FRACTAL-BASED METHODS IN IMAGING

A Thesis

Presented to

The Faculty of Graduate Studies

of

The University of Guelph

by

MATTHEW DEMERS

In partial fulfilment of requirements

for the degree of

Master of Science

May, 2007

© Matthew Demers, 2007

Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 978-0-494-30558-4

Our file Notre référence

ISBN: 978-0-494-30558-4

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

FRACTAL BASED METHODS IN IMAGING

Matthew Demers
University of Guelph, 2007

Advisor:
Dr. H. Kunze

With the age of computing, fractal imaging has become a well-developed research area. A wide variety of applications are explored in this thesis. We discuss a novel method to colour fractal attractor sets. Since such sets can resemble real objects, the method has potential applicability in the computer graphics industry. We also consider the application of these ideas to the area of image cryptography and information hiding. Switching focus to general digital images, we look at the theory and practice of fractal image compression. Our goal, however, is to extend the ideas used in compression to produce schemes for image denoising and image recovery. Some encouraging results for such applications are presented.

Acknowledgements

My research was supported over the past two years by means of grants and/or scholarships generously provided by the National Sciences and Engineering Council and the University of Guelph. I am very grateful for this aid.

I wish to thank my advisory committee members Pal Fischer and Stephen Gismondi. Both have dedicated much time and effort towards providing me with constructive criticism and friendly encouragement.

Finally, I must thank my supervisor, Herb Kunze, for his constant and invaluable guidance and insight. He has been a phenomenal collaborator and true friend, and this thesis would not have been the same without his patience and support.

Contents

Acknowledgements	i
List of Figures	iv
1 Introduction	1
2 Fundamentals of Images	5
2.1 Closed Subsets of the Real Plane and Image Functions	5
2.2 Pixel Space and Pixel Image Functions	8
3 Background	13
3.1 Banach's Fixed Point Theorem	13
3.2 Iterated Function Systems	17
4 Producing and Colouring IFS Attractors	27
4.1 Plotting Fractal Attractors	27
4.2 Code Space and Fractal Tops	35
4.3 Colouring Pictures using the Random Algorithm	39
4.4 Applications of Colour Stealing	44
5 Fractal Imaging	47
5.1 The Collage Theorem	47
5.2 Iterated Function Systems with Grey-Level Maps	50
5.3 Local Iterated Function Systems with Grey-Level Maps	65
5.4 Practical Issues with LIFSM	74
5.5 Decompression	82
6 Applications of Near-Optimal Parent-Child Pairs	85
6.1 An Introduction to Fractal Image Denoising	85
6.2 Noise and Noisy Images	86
6.3 The Denoising Algorithm	88

6.4	Reconstruction	90
6.5	Discussion	92
6.6	An Introduction to Fractal Image Recovery	97
7	Future Work	102

List of Figures

2.1	Examples of computer-generated images. From left to right, the colour spaces used are $\mathbb{C} = \{0, 1\}$, $\{0, 1, \dots, 255\}$, and $\{0, 1, \dots, 255\}^3$, respectively.	10
3.1	The Hausdorff distance d_H is the largest of the two depicted set distances.	20
3.2	The Koch curve is a well known set attractor that looks much like one side of a snowflake.	25
3.3	Some attractors strongly resemble things in the natural world.	26
4.1	From left to right, the approximation of the attractor of $\hat{m}$ formulated in (4.5), with 500 points, 2000 points, and 15000 points plotted respectively	34
4.2	the point x on attractor A is coloured by “stealing” the colour from another picture, defined by $(\mathcal{P}_1 \circ \phi_B \circ \tau_A)(x)$	40
4.3	A 512×512 digital picture with colour space $\mathbb{C} = \{0, 1, \dots, 255\}^3$. The colours in this image will be used to paint the set attractor of an IFS.	43
4.4	The “leaf” set attractor has been painted using the colours from the digital picture.	44
4.5	When the same IFS is used to both draw <i>and</i> colour the digital picture, the result is the portion that is supported by the set attractor.	45
5.1	The Hausdorff distance between two visually similar sets can still be large.	53
5.2	Each $\hat{w}_i$ defines an interval in $[0, 1]$	54
5.3	A one-dimensional signal $y = \mathcal{P}(x)$	55
5.4	Copies of the original signal are shrunk to fit on each interval.	55
5.5	Each copy is then fit as closely to the original signal as possible.	56
5.6	Choosing an overlapping IFS can lead to questions about what happens to $\mathcal{Q}(x)$ in the overlapped region.	57
5.7	An example of a simple signal.	60

5.8	The signal is shrunk into two smaller copies according to $\hat{\mathbf{w}}$	61
5.9	The shrunk copies are then fit closely to the original curve using the functions ϕ_1 and ϕ_2	63
5.10	Another simple (but nonmonotone) signal.	64
5.11	The best-fit function of this signal does not preserve the shape of the original.	65
5.12	The signal is divided up into different parent blocks (P_1, P_2) and child blocks (C_1, C_2, C_3, C_4).	67
5.13	Using shrunk subsets of the original signal does a far better job at creating a minimal collage than shrinking down the entire curve. . . .	69
5.14	A partitioning of a space into a fixed grid of child blocks.	70
5.15	A fixed grid scheme for parent blocks divides the image up in a similar fashion as the child blocks.	72
5.16	A 256×256 pixel, 256-greyscale image.	75
5.17	The image is carved up into a grid of parent and child blocks.	76
5.18	Crude Decimation doesn't take into account most of the pixels in the current parent block. An unrotated, unflipped parent orientation is assumed in this picture.	79
5.19	Averaging Decimation uses all of the parent pixels in the calculation of the inner product. Again, an unrotated, unflipped parent orientation is assumed here.	80
5.20	An example of a best parent-child pairing.	81
5.21	Images obtained after applying the optimal IFS one, two, and ten times to a seed image.	83
5.22	An analog to Figure 5.21 but with smaller parent block and child block sizes.	83
6.1	Noisy Images created using $\nu_{i,j}$ normally-distributed random variables with standard deviation $\sigma = 10, 25$, and 40 , respectively, are pictured from left to right.	88
6.2	Parent Block size: 16×16 pixels; Child Block Size: 8×8 pixels. A child block is outlined in white. The eight parents that produce the best parent-child pairings with that child are outlined in black.	89
6.3	Parent Block size: 8×8 pixels; Child Block Size: 4×4 pixels. Pictured from left to right: The original digital image, the noisy image, and the denoised image, averaging the best 25 parent blocks.	93
6.4	Parent Block size: 8×8 pixels; Child Block Size: 4×4 pixels. Pictured from left to right: The noisy image produced with $\sigma = 100$, and the denoised image using the best 20 parent blocks.	95
6.5	Parent Block size: 8×8 pixels; Child Block Size: 4×4 pixels. Pictured from left to right: An even noisier image produced with $\sigma = 200$, and the denoised image using again the best 20 parent blocks.	96

6.6	A 256×256 pixel, 256-greyscale image, with a small 4x4-pixel block labeled as “bad.” The bad pixels are coloured white.	98
6.7	Eight near-optimal pairs of parents are used in this recovery, which no longer contains the area of bad pixels that was evident in Figure 6.6. The original digital image with bad pixels is pictured on the left; the approximation of the digital image is pictured in the centre, and the recovered digital image is on the right.	100

Chapter 1

Introduction

With the dawn of imaging in the age of personal computing, fractal imaging arose first as an area of interest more than twenty years ago. The keen minds that first explored the ideas behind fractal image compression are still working on brand new fractal imaging applications today. There are many prominent reference books on fractal imaging and related topics [1, 2, 3, 4]. Some of the brand new results, as well as some of the older ones, will be discussed and further developed in this thesis. What, exactly, *is* fractal imaging, though?

The fundamental principle behind all fractal-based methods is Banach's Fixed Point Theorem, which states that a contraction map on a complete metric space has a unique, globally attractive fixed point. In other words, certain types of functions will act to map all points closer to a certain special point that is called fixed. By iterating such functions repeatedly, any point in the space that is being worked with can be mapped arbitrarily close to that fixed point. Iterated Function Systems (IFSs)

are systems of set-valued contraction maps that are often used to construct approximations of fractal images. As with single contraction mappings, there is a fixed point of any Iterated Function System; it is called the attractor of the IFS. An attractor is always self-similar in that pieces of itself are just shrunken copies of the whole. In practice, drawing the actual attractor is impossible, because attractors possess infinitesimally tiny details. Thus, one can only draw an approximation of it.

One such method of doing this is by employing the so-called chaos game. By choosing a starting point and selecting contraction maps at random, the sequence of points generated quickly approaches the set attractor. Thus, by graphing the set of points, a good approximation of the attractor may be made.

By further selecting an image and a second IFS, as first suggested by M. Barnsley [4], a second chaos game may be played simultaneously in order to paint the attractor of the first IFS with colours that are “stolen” from the image. This is a new and intriguing application that holds a lot of promise practically; it could have many benefits in the industry of computer graphics. This method will be explored in Section 4.3.

Barnsley also looked into the inverse problem that forms the basis of fractal imaging. Given a target image, is it possible to construct an IFS such that its attractor is a good approximation to the target? In fractal imaging applications, geometrically contractive mappings are chosen. To do this, a desired target picture is divided up into child pieces that are similar in shape to larger parent pieces. For each child

block, then, it is possible to find the parent block that is mathematically closest to it after application of the appropriate contraction map and a grey-level or colour-level adjustment. The collection of parameters that define the best parent-child pairings and colour adjustments define the fractal transform of the image, which is dubbed as a *local* iterated function system with grey/colour-level maps (LIFSM). This block-encoding scheme was introduced in the thesis of Jacquin [5]. Under iteration upon any initial image, the fractal transform produces an approximation of the original target image. These parameters can be thought of as a representation of the target image, and can typically be stored in a compact file, often yielding significant image compression.

In this thesis, the ideas of image compression first posed by Barnsley are taken a step further. Given a target image that is noisy, corrupted, or unreadable, the idea is to find a fractal transform that, under iteration, produces a clearer “denoised” image. This is achieved without having any knowledge of what the original, unobscured image looks like, which makes this idea especially exciting. Some encouraging results are presented in Chapter 6.

However, the basics that provide the framework for all of these ideas and innovative methods must first be discussed before some of the more complicated notions are tackled ambitiously. The structure of this thesis is as follows.

The second chapter of this thesis rigourously defines the notions of images, image functions, and colour spaces. It also defines a pixel, a pixelization, and a pixel image

function. These definitions and notations are essential for the rest of the thesis.

The third chapter discusses the background behind iterated function systems, exploring Banach's Fixed Point Theorem, Hutchinson's Theorem, and some examples of set attractors.

The fourth chapter details methods of plotting attractors with a computer, and outlines a random algorithm known as the chaos game. Using a variation of the chaos game, a method of colouring attractors called colour stealing will be discussed.

The fifth chapter explains the inverse problem and the collage theorem, and then discusses iterated function systems with grey-level maps before moving to the more complex LIFSM algorithm. The method of fractal image compression is then introduced, before practical considerations are taken into account.

The sixth chapter uses the ideas from fractal image compression and extends them to new applications. Two applications, fractal image denoising and fractal image recovery, are discussed.

The thesis concludes with a brief chapter outlining possible future work in the area of imaging.

Chapter 2

Fundamentals of Images

2.1 Closed Subsets of the Real Plane and Image Functions

Throughout the thesis, images will be discussed in many different contexts. This section discusses the distinction between image functions in the real plane and digital pictures that exist in the discrete space of pixels in order to establish a backbone for the chapters to come. Some of the notation and discussion points are motivated by [4]. To begin, some important concepts are defined.

Notation 1 *Let $\square = [0, 1] \times [0, 1]$. Observing that any closed, bounded, and rectangular subset of $\mathbb{R}^2$ can be scaled to $\square$, without loss of generality, the domain $\square$ is typically used in our image-related applications.*

When we are not working in the unit square, the symbol X will be used to represent a general space or domain.

Notation 2 *The symbol $\mathbb{C}$ is used to denote a **colour space**. Each point in the*

colour space $\mathbb{C}$ represents a colour or shade of grey.

Definition 3 An *image function* $\mathcal{P}$ is a function that associates to each point $(x, y) \in \square$ a point in a colour space $\mathbb{C}$. In other words,

$$\mathcal{P} : \square \mapsto \mathbb{C}. \quad (2.1)$$

In Notation 2, $\mathbb{C}$ could be one of a variety of possible spaces. For example,

1. $\{0, 1\}$, where each of the two values corresponds to a single colour choice. For example, 0 could be associated with the colour black and 1 with the colour white. Black and white pictures like these can actually be viewed as a visual rendering of a set: Each point in X is either in the set, if mapped to one colour in $\mathbb{C}$, or not, if mapped to the other. For example, points in the set may be coloured black, leaving those that are not in the set white. This will be discussed further in Chapter 3, where certain very special sets will be rendered as pictures;
2. a closed interval, such as $[0, 1]$, which allows any choice from a continuum of values to be assigned. Again, the values 0 and 1 may be associated to black and white respectively, while values in between represent some shade of grey. This is a colour space that will be used in Chapter 5;
3. a set of integers $\{0, 1, \dots, 255\}$, where every integer corresponds to a specific grey level in a computer image. 0 refers to the colour black, while 255 represents

white. This colour space will be used very extensively through Chapters 5 and 6; or

4. an ordered RGB triplet (R, G, B) in $[0, 1]^3$, the red, green, and blue colour components of a colour picture. In this colour space, $(1, 1, 1)$ corresponds to white and $(0, 0, 0)$ corresponds to black. The point $(1, 0, 0)$ represents the colour red, $(0, 1, 0)$ green, and $(0, 0, 1)$ blue. The general triplets (R, G, B) correspond to the colour obtained by mixing the associated amounts of red, green, and blue. For example, $(0.663, 0, 0.863)$ represents a light purple. This colour space will be utilized in Chapter 4, when a novel way of attaching colour to points in certain sets is discussed.

Sometimes, to refer to the set of coloured points of $\square$, the image function and the set on which it acts will be paired together.

Definition 4 Let $\mathcal{P} : \square \mapsto \mathbb{C}$ be an image function. Then $(\square; \mathcal{P})$ is called an **image**.

Seeking a computationally convenient way to work with image functions, the space of all image functions is established.

Definition 5 Let $\square$ be the unit square and $\mathbb{C}$ be a colour space. Then, **image space** is defined by

$$\mathcal{F}(\square, \mathbb{C}) = \{\mathcal{P} \mid \mathcal{P} : \square \mapsto \mathbb{C}\}. \quad (2.2)$$

In this thesis, the question will often arise as to whether one image function is “close” to another image function in this space of images. To determine a distance like this,

a metric is required.

Definition 6 *Given an image function $\mathcal{P} \in \mathcal{F}(\square, \mathbb{C})$, the norm of $\mathcal{P}$ is calculated as*

$$\|\mathcal{P}\|_2 = \left[\int_{(x,y) \in \square} \|\mathcal{P}(x,y)\|_2^2 dA \right]^{1/2} < \infty.$$

The distance between two functions $\mathcal{P}_1, \mathcal{P}_2 \in \mathcal{F}(\square, \mathbb{C})$ is then defined by

$$d_2(\mathcal{P}_1, \mathcal{P}_2) = \|\mathcal{P}_1 - \mathcal{P}_2\|_2.$$

The notation $\|\mathcal{P}\|_2$ should not be confused with $\|\mathcal{P}(x,y)\|_2$ in the definition, which is calculated as the usual Cartesian norm of the colour space value $\mathcal{P}(x,y)$. We also assume that certain restrictions on the image function are in place in order to ensure that the integral in Definition 6 exists.

In theory, these definitions establish well the relationship between points and colours. An image function maps any point $(x,y) \in \square$ to a point in the appropriate colour space. A point, however, is an infinitesimally tiny thing. Because of this, for practical purposes, it is impossible to work with functions like the ones defined in this section, because computer screens possess only finite resolution. This issue is addressed in the next section.

2.2 Pixel Space and Pixel Image Functions

The fundamental building block of any computer-generated image is a pixel, representing the smallest possible unit of space that can be depicted on a screen. The

precise mathematical definition of what we mean by a pixel will now be established.

Definition 7 *The pixel $\mathbf{P}(i, j)$ of area $1/r^2$, where i, j , and r are integers such that $1 \leq i \leq r$ and $1 \leq j \leq r$, is a square-shaped set of points in $\square$ given by*

$$\mathbf{P}(i, j) = \left\{ (x, y) \mid \frac{i-1}{r} \leq x < \frac{i}{r}, \frac{j-1}{r} \leq y < \frac{j}{r} \right\}. \quad (2.3)$$

Definition 8 *The set of all pixels of area $1/r^2$ is called the **pixelization** with resolution r of $\square$, given by*

$$\mathbf{P} = \{\mathbf{P}(i, j) \mid 1 \leq i \leq r, 1 \leq j \leq r\}. \quad (2.4)$$

A pixel is thus a square-shaped box of points $(x, y) \in \square$. A pixelization $\mathbf{P}$ is the set of pixels that, when arranged in a rectangular array, fills up the area in $\square$. $\square$ can then be thought of as a box of pixels, with a height and a length of r pixels. Noting that the set of pixels is a kind of rectangular array, it is possible to easily refer to any single pixel $\mathbf{P}(i, j)$ by giving its column i and row j . Thus, for example, $\mathbf{P}(1, 1)$ represents the bottom left pixel while $\mathbf{P}(r, r)$ refers to the top right pixel.

The assumption that pixels in $\square$ are square-shaped leads to pixelizations that are also square in shape, but this assumption can be softened to allow rectangular pixelizations of a domain other than $\square$. The effect of this assumption is that, for the purposes of this thesis, resolution can be defined as a single number rather than two numbers (for example, “1024 × 768”), which is more common.

Under this formulation, the resolution of a pixelization is the square root of the number of pixels $\mathbf{P}(i, j) \in \mathbf{P}$. In this thesis, a resolution of 256 will be assumed for

any pixelization $\mathbf{P}$ unless explicitly stated otherwise.

In computer-produced images like the ones in Figure 2.1, as well as the images that will be dealt with in later chapters, image functions as defined in Definition 3 have notable characteristics. For such images, $\mathcal{P}$ is a piecewise constant function with possible discontinuities at the edge of each pixel. A piecewise-constant image function $\mathcal{P} : \square \mapsto \mathbb{C}$ always maps *all* points (x, y) in any single pixel $\mathbf{P}(i, j)$ to the same element of the colour space $\mathbb{C}$. Thus, images that have a higher resolution can include more detail. For example, a 256×256 pixel image has lower resolution and thus less detail than a 512×512 image.

Figure 2.1: Examples of computer-generated images. From left to right, the colour spaces used are $\mathbb{C} = \{0, 1\}$, $\{0, 1, \dots, 255\}$, and $\{0, 1, \dots, 255\}^3$, respectively.

It will be useful at times to define another sort of image function that explicitly maps any pixel $\mathbf{P}(i, j)$ to its colour or greyscale value, as opposed to an image function, which maps a *point* $(x, y) \in \square$ to a point in $\mathbb{C}$.

Definition 9 Let $\mathbb{C}$ be a colour space, $\mathcal{P} : \square \mapsto \mathbb{C}$, and $\mathbf{P}(i, j)$ represent a pixel in the pixelization $\mathbf{P}$ of resolution r . For each i, j , assume that $\mathcal{P}(x, y)$ has the same

value for all $(x, y) \in \mathbf{P}(i, j)$. Then define a **pixel image function** $\tilde{\mathcal{P}} : \mathbf{P} \mapsto \mathbb{C}$ by

$$\tilde{\mathcal{P}}(\mathbf{P}(i, j)) = \mathcal{P}(x, y) \quad \forall (x, y) \in \mathbf{P}(i, j). \quad (2.5)$$

To refer to the coloured points of a pixelization $\mathbf{P}$, the pixel image function and the domain on which it acts will be paired together.

Definition 10 Let $\tilde{\mathcal{P}} : \mathbf{P} \mapsto \mathbb{C}$ be a pixel image function. Then the pairing $(\mathbf{P}; \tilde{\mathcal{P}})$ is called a **digital image**.

The space of all pixel image functions is now defined.

Definition 11 Let $\mathbf{P}$ be a pixelization of resolution r and $\mathbb{C}$ be a colour space. Then, **pixel image space** is defined by

$$\tilde{\mathcal{F}}(\mathbf{P}, \mathbb{C}) = \{\tilde{\mathcal{P}} \mid \tilde{\mathcal{P}} : \mathbf{P} \mapsto \mathbb{C}\}. \quad (2.6)$$

The norm on pixel image space will now be defined.

Definition 12 Given a pixel image function $\tilde{\mathcal{P}} \in \tilde{\mathcal{F}}(\mathbf{P}, \mathbb{C})$, the norm of $\tilde{\mathcal{P}}$ is defined as

$$\|\tilde{\mathcal{P}}\|_2 = \left[\sum_{\mathbf{P}(i, j) \in \mathbf{P}} (\tilde{\mathcal{P}}(\mathbf{P}(i, j)))^2 \right]^{1/2} < \infty,$$

with the distance between two functions $\tilde{\mathcal{P}}_1, \tilde{\mathcal{P}}_2 \in \tilde{\mathcal{F}}(\mathbf{P}, \mathbb{C})$ defined by

$$d_2(\tilde{\mathcal{P}}_1, \tilde{\mathcal{P}}_2) = \|\tilde{\mathcal{P}}_1 - \tilde{\mathcal{P}}_2\|_2.$$

These definitions will come in very handy in the sections to come. In Chapter 4, interesting sets of points called IFS attractors will be coloured by image functions in a unique way. In later chapters, while talking about fractal image compression and related applications, the ideas presented in this chapter will again surface.

Chapter 3

Background

In this chapter, we present some background material from analysis. Further discussions of this and related material can be found in various publications [1, 6].

3.1 Banach's Fixed Point Theorem

To set up the framework for this important chapter, a complete metric space (X, d_X) is considered.

Definition 13 *A mapping $f : X \mapsto X$ is **contractive** if there exists a $c \in [0, 1)$ such that $d_X(f(x), f(y)) \leq c d_X(x, y), \forall x, y \in X$. The value c is called the **contraction factor** of f . f is called a **contraction mapping**.*

The nearer that c is to zero, the more powerful the contraction is. A special type of contraction mapping is now defined.

Definition 14 A *linear* or *affine* mapping f is a function

$$f(t) = at + b. \quad (3.1)$$

f is contractive with respect to absolute value if and only if $-1 < a < 1$.

Affine maps on the plane are very commonly used in Iterated Function Systems, which will be made quite evident in Section 3.2 and beyond.

It is useful to consider the space of all contraction maps, which we will now define.

Definition 15 The set of all contraction maps on X is denoted as

$$\begin{aligned} \text{Con}(X) = \{ & f : X \rightarrow X \mid \exists a \ c_f \in [0, 1) \text{ such that} \\ & d_X(f(x), f(y)) \leq c_f d_X(x, y) \ \forall x, y \in X \} \end{aligned} \quad (3.2)$$

$\text{Con}(X)$ will prove to be an essential concept in the pages to come. One important property of this space is stated now in a theorem.

Theorem 16 If $f \in \text{Con}(X)$, then f is uniformly continuous on (X, d_X) .

Proof: f is uniformly continuous on (X, d_X) if for any $\varepsilon > 0$, $\exists$ a $\delta > 0$ such that for all $x, a \in X$, whenever $d_X(x, a) < \delta$, $d_X(f(x), f(a)) < \varepsilon$.

For all $x, a \in X$, $d_X(f(x), f(a)) \leq c d_X(x, a)$, because f is a contraction mapping. If $c = 0$, then $d_X(f(x), f(a)) \leq 0$, and so $d_X(f(x), f(a)) \leq \varepsilon$ no matter what value of δ is chosen. If $c > 0$, choose $\delta = \frac{\varepsilon}{c}$. Then, if $d_X(x, a) < \delta = \frac{\varepsilon}{c}$, it is true that $d_X(f(x), f(a)) \leq c d_X(x, a) < c \frac{\varepsilon}{c} = \varepsilon$ for any $\varepsilon > 0$. Thus, f is uniformly continuous on (X, d_X) if $f \in \text{Con}(X)$.

□

A crucial theorem, first proposed in 1922 by Banach, now follows. This theorem will set the stage for a large part of the rest of this thesis.

Theorem 17 (*Banach's Fixed Point Theorem*): *Let (X, d_X) be a complete metric space and let $f \in \text{Con}(X)$ with contraction factor $c \in [0, 1)$. Then there exists a unique fixed point $\bar{x} \in X$ such that $\bar{x} = f(\bar{x})$. Additionally, for any $x \in X$, $d_X(f^{\circ n}(x), \bar{x}) \rightarrow 0$ as $n \rightarrow \infty$.*

It is common to state the conclusion of Banach's Fixed Point Theorem by saying that there exists a unique, globally attractive fixed point.

Proof: Choose a point $x_0 \in X$ and create a sequence by letting $x_{n+1} = f(x_n)$, for $n \geq 0$. In that case, $x_n = f^{\circ n}(x_0)$, where $f^{\circ 0}(x_0) = x_0$.

Let m and n be positive integers satisfying $m < n$. Then,

$$\begin{aligned} d_X(x_m, x_n) &= d_X(f^{\circ m}(x_0), f^{\circ n}(x_0)) \\ &= d_X(f(f^{\circ(m-1)}(x_0)), f(f^{\circ(n-1)}(x_0))) \\ &\leq c d_X(f^{\circ(m-1)}(x_0), f^{\circ(n-1)}(x_0)), \quad \text{by contractivity.} \end{aligned}$$

$$\begin{aligned}
\text{Continuing the process, } &= c d_X(f(f^{\circ(m-2)}(x_0)), f(f^{\circ(n-2)}(x_0))) \\
&\leq c^2 d_X(f^{\circ(m-2)}(x_0), f^{\circ(n-2)}(x_0)) \\
&\quad \dots \\
&\leq c^m d_X(x_0, f^{\circ(n-m)}(x_0))
\end{aligned}$$

Now consider $d_X(x_0, f^{\circ k}(x_0))$ for $k \geq 1$.

$$d_X(x_0, f^{\circ k}(x_0)) = d_X(x_0, x_k)$$

$$\begin{aligned}
\text{Using triangle inequality, } &\leq d_X(x_0, x_1) + d_X(x_1, x_k) \\
&\leq d_X(x_0, x_1) + d_X(x_1, x_2) + d_X(x_2, x_k)
\end{aligned}$$

$$\begin{aligned}
\text{Repeating sufficiently, } &\leq d_X(x_0, x_1) + d_X(x_1, x_2) + \dots + d_X(x_{k-1}, x_k) \\
&= d_X(x_0, x_1) + d_X(f(x_0), f(x_1)) + \\
&\quad \dots + d_X(f^{\circ(k-1)}(x_0), f^{\circ(k-1)}(x_1)) \\
&\leq d_X(x_0, x_1) + c d_X(x_0, x_1) + \dots + c^{k-1} d_X(x_0, x_1) \\
&= (1 + c + c^2 + \dots + c^{k-1}) d_X(x_0, x_1) \\
&= \frac{1 - c^k}{1 - c} d_X(x_0, x_1) \\
&\leq \frac{1}{1 - c} d_X(x_0, x_1)
\end{aligned}$$

$$\text{In conclusion, } d_X(x_m, x_n) \leq c^m \frac{1}{1 - c} d_X(x_0, x_1)$$

Since $0 \leq c < 1$, then given any $\varepsilon > 0$, a number $M > 0$ may be found such that

$\frac{c^m}{1 - c} d_X(x_0, x_1) < \varepsilon$ for all $m \geq M$. Thus, $d_X(x_m, x_n) < \varepsilon$ for all $n > m$, when $m \geq M$.

Hence, the sequence $\{x_n\}$ is Cauchy in X . X is complete, and so the Cauchy sequence must converge to some $\bar{x} \in X$. (i.e. for any $\delta > 0$, there exists an $N > 0$ such that $d_X(x_n, \bar{x}) < \delta$ whenever $n \geq N$.) Therefore, the limit of $x_{n+1} = f(x_n)$ may be taken as $n \rightarrow \infty$ to obtain $\bar{x} = f(\bar{x})$, noting that $\lim_{n \rightarrow \infty} f(x_n) = f(\lim_{n \rightarrow \infty} x_n)$, by continuity of $f(x)$. In other words, $\bar{x}$ is a fixed point of f .

To show that $\bar{x}$ is a *unique* fixed point, the opposite is assumed in order to derive a contradiction. Specifically, we assume that there are two different fixed points $\bar{x}$ and $\bar{z}$. Thus, we assume that $f(\bar{x}) = \bar{x}$ and $f(\bar{z}) = \bar{z}$, for $\bar{x}, \bar{z} \in X$, with $\bar{x} \neq \bar{z}$. Then,

$$d_X(\bar{x}, \bar{z}) = d_X(f(\bar{x}), f(\bar{z})) \leq c d_X(\bar{x}, \bar{z})$$

$d_X(\bar{x}, \bar{z}) > 0$ by our assumption, and thus each side of the inequality may be divided by it to get $1 \leq c$. This is a contradiction, since $f \in \text{Con}(X)$ and so c must be less than 1! Thus, $\bar{x} = \bar{z}$, and the fixed point must be a unique one. □

3.2 Iterated Function Systems

The ideas behind contraction mappings extend to the more complex and interesting idea of iterated function systems, commonly abbreviated as “IFSs” [1, 7]. Let (X, d_X) be a complete metric space. Furthermore, define a number of contraction mappings; to each of these, associated mappings are defined that act to map subsets of X to subsets of X . The concept of an iterated function system is now formally defined.

Definition 18 Let (X, d_X) be a complete metric space, and let $w_i : X \mapsto X$ such that $w_i \in \text{Con}(X)$, with $i = 1, \dots, N$. Then,

$$\hat{w}_i(X) = \{w_i(x) | x \in X\} \quad (3.3)$$

$$\text{and } \hat{w}(X) = \bigcup_{i=1}^N \hat{w}_i(X) \quad (3.4)$$

Then, $(\hat{w}, X)$ is said to be an **iterated function system**, or **IFS**. An IFS that includes N mappings is often called an **N -map IFS**.

Since each of the mappings involved in the IFS are contraction mappings, we often say that the system is a contractive IFS. Now, to show that $\hat{w}$ is a contraction map on X , some sort of metric space of sets must be used; it is important to recognize that the “points” mapped by $\hat{w}$ are in fact sets themselves!

Some metrics will now be established in order to determine distances between sets.

First, we define the distance between a point and a set.

Definition 19 Let $x \in X$ be a point and $S \subseteq X$ be set. Then, let

$$d'_X(x, S) = \inf_{y \in S} d_X(x, y). \quad (3.5)$$

$d'_X(x, S)$ is the **point-set distance** of X .

$\inf_{y \in S} d_X(x, y)$ is well defined because X is a complete space. Using this definition, the distance from one set to another set may now be defined.

Definition 20 Let $R \subseteq X$ and $S \subseteq X$. Then,

$$\begin{aligned} d_{X,set}(R, S) &= \sup_{x \in R} d'_X(x, S) \\ &= \sup_{x \in R} \left(\inf_{y \in S} d_X(x, y) \right) \end{aligned} \quad (3.6)$$

is the *set-set distance* of X .

Example: Although it might not seem immediately clear, it is usually the case that $d_{X,set}(R, S) \neq d_{X,set}(S, R)$. Take X to be a closed, bounded interval of the real line, with the natural metric $d_X(x, y) = |x - y|$, and let $R = [0, 0.4]$, and $S = [0.3, 1.0]$. Then, consider the following:

$$\begin{aligned} d_{X,set}(R, S) &= \sup_{x \in R} d'_X(x, S) \\ &= \sup_{x \in R} \inf_{y \in S} d_X(x, y) \\ &= 0.3, \end{aligned}$$

whereas,

$$\begin{aligned} d_{X,set}(S, R) &= \sup_{x \in S} d'_X(x, R) \\ &= \sup_{x \in S} \inf_{y \in R} d_X(x, y) \\ &= 0.6. \end{aligned}$$

The two set distances in this particular example are clearly not the same, and the same is true in general.

Now, the Hausdorff distance between two sets may at last be defined.

Definition 21 *The Hausdorff distance between two sets $R \subseteq X$ and $S \subseteq X$ is given by*

$$d_{\mathcal{H}} = \max\{d_{X, \text{set}}(R, S), d_{X, \text{set}}(S, R)\}, \quad (3.7)$$

where $d_{X, \text{set}}(R, S)$ and $d_{X, \text{set}}(S, R)$ are calculated as in Figure 3.1.

Figure 3.1: The Hausdorff distance $d_{\mathcal{H}}$ is the largest of the two depicted set distances.

This metric is very useful, because it can be employed in order to show that the IFS $(\hat{w}, X)$ is indeed contractive.

Theorem 22 *$(\hat{w}, X)$ is a contraction mapping on $(\mathcal{H}(X), d_{\mathcal{H}})$, where $\mathcal{H}(X)$ is the set of all nonempty compact subsets of X .*

Proof: This theorem will now be proven for the case when the IFS consists of only two mappings; the proof extends easily to include IFSs consisting of greater numbers of mappings too, by induction.

It can be shown that for any four sets $S_1, S_2, S_3, S_4 \in \mathcal{H}(X)$,

$$d_{\mathcal{H}}(S_1 \cup S_2, S_3 \cup S_4) \leq \max\{d_{\mathcal{H}}(S_1, S_3), d_{\mathcal{H}}(S_2, S_4)\} \quad (3.8)$$

Let $R \in \mathcal{H}(X)$ and $S \in \mathcal{H}(X)$. Then, using the identity in (3.8), we obtain the following:

$$\begin{aligned} d_{\mathcal{H}}(\hat{w}_1(R) \cup \hat{w}_2(R), \hat{w}_1(S) \cup \hat{w}_2(S)) &\leq \max\{d_{\mathcal{H}}(\hat{w}_1(R), \hat{w}_1(S)), d_{\mathcal{H}}(\hat{w}_2(R), \hat{w}_2(S))\} \\ &\leq \max\{c_1, c_2\} d_{\mathcal{H}}(R, S) \end{aligned}$$

(since $d_{\mathcal{H}}(\hat{w}_1(R), \hat{w}_1(S)) \leq c_1 d_{\mathcal{H}}(R, S)$ and $d_{\mathcal{H}}(\hat{w}_2(R), \hat{w}_2(S)) \leq c_2 d_{\mathcal{H}}(R, S)$; taking the larger of the two c_i assures the result.)

Both c_1 and c_2 are in $[0, 1)$, and therefore $\max\{c_1, c_2\}$ certainly is! Let $c_{\hat{w}} = \max\{c_1, c_2\}$. Then, $d_{\mathcal{H}}(\hat{w}(R), \hat{w}(S)) \leq c_{\hat{w}} d_{\mathcal{H}}(R, S)$. Thus, $\hat{w} = \hat{w}_1 \cup \hat{w}_2$ is indeed a contraction mapping on $(\mathcal{H}(X), d_{\mathcal{H}})$. □

Having proved contractivity in the preceding result, a recasting of Banach's theorem for iterated function systems unfolds rather naturally. This theorem, due to Hutchinson, is stated here.

Theorem 23 *Suppose that $\hat{w} = \{\hat{w}_1, \dots, \hat{w}_N\}$ is an N -map IFS in a complete metric space $(\mathcal{H}(X), d_{\mathcal{H}})$. Then, there exists a unique compact set A such that*

$$A = \bigcup_{i=1}^N \hat{w}_i(A)$$

Furthermore, for any compact set $S \subseteq X$, the sequence of sets $\{S_i\}$ defined by

$$\begin{aligned} S_0 &= S \\ \text{and } S_{m+1} &= \bigcup_{i=1}^N \hat{w}_i(S_m) \end{aligned}$$

converges to A in the Hausdorff metric. A is referred to as the **attractor** or **set attractor** of the IFS.

Proof: $(\mathcal{H}(X), d_{\mathcal{H}})$ is a complete metric space. Since each $\hat{w}_i$ is contractive on $(\mathcal{H}(X), d_{\mathcal{H}})$, by Theorem 16, we know that $\hat{w}_i$ is uniformly continuous on $\mathcal{H}(X)$. This means that $\hat{w}_i : \mathcal{H}(X) \mapsto \mathcal{H}(X)$. Furthermore, because it is the union of a finite number of such maps, $\hat{\mathbf{w}} : \mathcal{H}(X) \mapsto \mathcal{H}(X)$. That is, $\hat{\mathbf{w}}$ satisfies the hypotheses of Banach's Fixed Point Theorem on the complete metric space $(\mathcal{H}(X), d_{\mathcal{H}})$. The conclusion of Banach's Theorem, modified appropriately, follows. $\square$

This result was originally presented by Hutchinson in [8]. Hutchinson's Theorem guarantees that every contractive IFS possesses a unique attractor A . Iterated Function Systems sometimes have very interesting attractors, depending on the maps w_i that are involved. In some cases, when working in a two-dimensional space, the fixed point can take the form of a shape resembling objects that exist in the natural world. A few examples of IFSs are now presented in order to demonstrate this.

Examples:

1. In the space $X = [0, 1]$ with $d_X(x, y) = |x - y| \forall x, y \in X$, define the IFS $(\hat{\mathbf{w}}, X)$ with mappings as follows:

$$\begin{cases} w_1(x) &= \frac{1}{3}x \\ w_2(x) &= \frac{1}{3}x + \frac{2}{3} \end{cases} \quad (3.9)$$

Both w_1 and w_2 are affine maps, and both are contractive with respect to d_X , because the coefficient of the first-degree term is less than one. Furthermore, the associated set-valued maps $\hat{w}_1$ and $\hat{w}_2$ each map X to X . Begin with the set $I_0 = [0, 1]$, defining $I_j = \hat{\mathbf{w}}(I_{j-1})$, for integers $j \geq 1$. Then,

$$\begin{aligned}\hat{\mathbf{w}}(I_0) &= \hat{w}_1(I_0) \cup \hat{w}_2(I_0) = \left[0, \frac{1}{3}\right] \cup \left[\frac{2}{3}, 1\right] \\ \hat{\mathbf{w}}(I_1) &= \hat{w}_1(I_1) \cup \hat{w}_2(I_1) = \left[0, \frac{1}{9}\right] \cup \left[\frac{2}{9}, \frac{1}{3}\right] \cup \left[\frac{2}{3}, \frac{7}{9}\right] \cup \left[\frac{8}{9}, 1\right] \\ &\dots\end{aligned}$$

By iterating repeatedly, I_j converges to the famous middle-thirds Cantor Set as $j \rightarrow \infty$. This Cantor Set is indeed the set attractor of the IFS given in (3.9).

2. The same idea is extended to the real plane in this example. In the space $X = [0, 1]^2$, with points $(x, y) \in X$ and metric $d_X(x, y) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$, the von Koch attractor is produced by the

IFS defined by:

$$\left\{ \begin{array}{lcl} k_1(x, y) & = & \begin{bmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{3} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ k_2(x, y) & = & \begin{bmatrix} \frac{1}{6} & -\frac{\sqrt{3}}{6} \\ \frac{\sqrt{3}}{6} & \frac{1}{6} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ k_3(x, y) & = & \begin{bmatrix} \frac{1}{6} & \frac{\sqrt{3}}{6} \\ -\frac{\sqrt{3}}{6} & \frac{1}{6} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{2} \end{bmatrix} \\ k_4(x, y) & = & \begin{bmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{3} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{2}{3} \\ 0 \end{bmatrix} \end{array} \right. \quad (3.10)$$

If, as in the previous example, an initial set $I_0 = [0, 1] \times [0, 1]$ is chosen, letting $I_j = \hat{\mathbf{k}}(I_{j-1})$, $j \geq 1$, then applying $\hat{\mathbf{k}}$ repeatedly brings the resulting sets ever closer to a set attractor. The attractor of this IFS is often called the “snowflake curve” for reasons that are obvious once an approximation of its graph is observed. See Figure 3.2.

3. Let X be $[-3, 3] \times [0, 11]$, with d_X as in the previous example. One may construct the Barnsley Spleenwort Fern with the four-map IFS consisting of:

Figure 3.2: The Koch curve is a well known set attractor that looks much like one side of a snowflake.

$$\left\{ \begin{array}{l} s_1(x, y) = \begin{bmatrix} 0 & 0 \\ 0 & 0.16 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ s_2(x, y) = \begin{bmatrix} 0.2 & -0.26 \\ 0.23 & 0.22 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 1.6 \end{bmatrix} \\ s_3(x, y) = \begin{bmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0.14 \end{bmatrix} \\ s_4(x, y) = \begin{bmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 1.6 \end{bmatrix} \end{array} \right. \quad (3.11)$$

The attractor of this IFS, an approximation of which can be seen in Figure 3.3, is strikingly natural in appearance.

It is important to remember that any picture of an IFS attractor is only an approximation of the true attractor. While computers have finite resolution, set attractors

Figure 3.3: Some attractors strongly resemble things in the natural world.

possess infinitesimally tiny details. How exactly are these pictures generated using a computer, then, especially if these details are so small? This question will be answered in the next chapter, as methods of generating graphical approximations to fractal attractors will be discussed. Then, a unique way of applying colour to an attractor, the method of *colour stealing*, will be explored.

Chapter 4

Producing and Colouring IFS Attractors

4.1 Plotting Fractal Attractors

Drawing an accurate picture of an attractor of many iterated function systems is an impossible task. No matter how one may try, there is an *infinite* amount of detail in any set attractor with a fractal nature. How, then, does one produce pictures of such a set? There are a couple of methods that one can use to plot good approximations of these mathematical objects, and those will be discussed in this chapter. Afterwards, a method of colouring these attractors in interesting ways will be presented, along with the necessary background.

Let (X, d_X) be a complete metric space, and $(\hat{\mathbf{w}}, X)$ be an N-map IFS made up of maps $\hat{w}_1, \hat{w}_2, \dots, \hat{w}_N$, with $\hat{w}_i \in \text{Con}(X)$. Then, by Theorem 23, there exists a unique fixed point $A \in \mathcal{H}(X)$ such that $\hat{\mathbf{w}}(A) = A$. Moreover, for any set $S \in \mathcal{H}(X)$, $d_{\mathcal{H}}(\hat{\mathbf{w}}^{\circ n}(S), A) \rightarrow 0$ as $n \rightarrow \infty$.

Two possible algorithms for generating approximations of set attractors are now

presented.

1. Deterministic Algorithm

Begin with $S = \{x_0\}$, a single point in X . In that case,

$$\begin{aligned}\hat{\mathbf{w}}(S) &= \bigcup_{i=1}^N \hat{w}_i(S) \\ &= \bigcup_{i=1}^N w_i(x_0) \\ &= \{w_1(x_0), \dots, w_N(x_0)\}\end{aligned}$$

Note that this set of points is not necessarily a set of N distinct points! (It is possible to have two maps w_p and w_q such that $w_p(x_0) = w_q(x_0)$.)

Second and further iterations of $\hat{\mathbf{w}}$ are then applied to the initial point:

$$\hat{\mathbf{w}}(\hat{\mathbf{w}}(S)) = \bigcup_{i,j=1}^N w_i(w_j(x_0)) \tag{4.1}$$

$$\begin{aligned}&= \{w_1(w_1(x_0)), \dots, w_1(w_N(x_0)), w_2(w_1(x_0)), \\ &\quad \dots, w_2(w_N(x_0)), \dots, w_N(w_N(x_0))\}\end{aligned} \tag{4.2}$$

...

$$\hat{\mathbf{w}}^{on}(S) = \{w_{i_1} \circ w_{i_2} \circ \dots \circ w_{i_n}(x_0)\} \tag{4.3}$$

The set in (4.3) is a set of points containing all possible combinations of w -maps being applied in different orders, with $i_j \in \{1, 2, \dots, N\}$, giving a total of at most N^n points; this is a number that would obviously grow massive very quickly. One could list them using a natural counting order, starting with

$w_1 \circ w_1 \circ \dots \circ w_1(x_0)$ and ending with $w_N \circ w_N \circ \dots \circ w_N(x_0)$. Note that if a point x_0 on A is chosen as the initial point, all of the points mapped by $\hat{w}$ will also be elements of A , and so $\hat{w}(x_0)$ will be a subset of the set attractor.

Assuming that the initial point does not lie in A , then the set of points obtained at each iteration is guaranteed to be closer to the attractor A than the set before, because the IFS is contractive. A may be approximated by stopping this algorithm short after any number of iterations, recording all of the points, and graphing them (presumably using a computer program). A method like this is sure to work, but the storage space required with every step grows faster than exponentially! Any program would require an exorbitant amount of space after applying $\hat{w}$ just a few times to a single point. If the initial set were *more* than a single point, the problem would arise even more quickly. So, for practical purposes, a different algorithm is necessary to obtain an acceptable approximation to the fractal attractor A .

2. Random Algorithm

The random algorithm, quite often referred to as “the chaos game,” [1] is a far faster, more efficient method of producing a good approximation of the attractor A of a contractive IFS. First, a seed point $x_0 \in X$ is chosen. The sequence $\{x_1, x_2, \dots\}$ is formed by letting

$$x_{n+1} = w_{\sigma_n}(x_n). \quad (4.4)$$

σ_n is an index chosen randomly for every $n \geq 1$, with $\sigma_n \in \{1, 2, \dots, N\}$, and the choice is independent of choices made for previous n values. (Recall that N is the total number of maps in the IFS.) A probability is assigned for each possible selection of mapping for all n . Let p_k represent the probability of selecting map $w_k, k \in \{1 \dots N\}$. Of course, the stipulation that $\sum_{k=1}^N (p_k) = 1$ must hold. The resulting IFS with probabilities is sometimes referred to as an **IFSP**.

Theorem 24 *Let (X, d_X) be a complete metric space. Let $(\hat{w}, X)$ be a contractive IFS consisting of N mappings. Define the random sequence $\{x_n\}$ by*

$$\begin{aligned} x_1 &= w_{\sigma_1}(x_0) \\ x_2 &= w_{\sigma_2}(x_1) = w_{\sigma_2} \circ w_{\sigma_1}(x_0) \\ &\dots \\ x_n &= w_{\sigma_n}(x_n) = w_{\sigma_{n-1}} \circ \dots \circ w_{\sigma_2} \circ w_{\sigma_1}(x_0) \end{aligned}$$

Finally, let A be the attractor of the IFS $(\hat{w}, X)$. Then, for all random sequences $\{\sigma_n\}_{n=1}^{\infty}$, the corresponding sequence $\{x_n\}_{n=0}^{\infty}$ contains points that are arbitrarily close to A .

Proof: Given a seed point, $x_0 \in X$, then by Definition 19,

$$\begin{aligned} d'_X(x_0, A) &= \inf_{y \in A} d_X(x_0, y) \\ \text{So, } d'_X(w_{\sigma_1}(x_0), A) &= \inf_{y \in A} d_X(w_{\sigma_1}(x_0), y) \end{aligned}$$

Let A_{σ_1} represent $\hat{w}_{\sigma_1}(A)$; this set is a subset of A itself, of course. Then,

$$\begin{aligned} d'_X(w_{\sigma_1}(x_0), A) &\leq \inf_{y \in A_{\sigma_1}} d_X(w_{\sigma_1}(x_0), y) \\ &= \inf_{z \in A} d_X(w_{\sigma_1}(x_0), w_{\sigma_1}(z)) \\ &\leq c \inf_{z \in A} d_X(x_0, z), \text{ where contraction factor } c = \max_{1 \leq i \leq N} \{c_i\} \\ &= c d'_X(x_0, A) \end{aligned}$$

If this knowledge is applied to any point x_i produced by the random algorithm after, then

$$\begin{aligned} d'_X(x_k, A) &= d'_X(w_{\sigma_k}(x_{k-1}), A) \\ &\leq c d'_X(x_{k-1}, A) \\ &\leq c^2 d'_X(x_{k-2}, A) \\ &\dots \\ &\leq c^k d'_X(x_0, A) \end{aligned}$$

Because $0 < c < 1$, $d'_X(x_n, A)$ must converge to zero as $n \rightarrow \infty$. Thus, the points in the sequence $\{x_n\}_1^k$ must contain points that are arbitrarily close to points on A as $k \rightarrow \infty$.

□

We can even go a step further with the following theorem.

Theorem 25 *Let (X, d_X) , $(\hat{w}, X)$, and $\{x_n\}$ be defined as in Theorem 24. Then, for any $x_0 \in X$ and almost all sequences $\{\sigma_n\}$, the random walk $\{x_n\}_{n=0}^\infty$ is dense in A . In other words, for any point $a \in A$ and any $\varepsilon > 0$, there exists an $m > 0$ such that $d_X(x_m, a) < \varepsilon$.*

Proof: See [1].

If, of course, the initial point x_0 lies on the attractor, then the sequence of $\{x_n\}$ consists *entirely* of points that lie on the attractor.

So, the random algorithm does indeed produce points that closely approximate any points on the set attractor after a sufficient number of iterations. This method has definite advantages, computationally, because of its iterative nature. After every new x_i is obtained through the randomly chosen mapping, the previous one (i.e. x_{i-1}) can be plotted on a graph and then replaced by the new piece of information, whereupon the procedure repeats itself until a desired number of points have been obtained. For example, only the second point must be known in order to calculate the third one, only the third point is necessary to calculate the fourth, and so on.

Remark :

It is important to note that computer plots have finite precision. When a computer program plots a point in the sequence $\{x_n\}$, it colours the pixel on the computer screen in which that point lies. Essentially, then, pixels can be thought of as box-shaped sets each consisting of points in X . If a term of the sequence $\{x_n\}$ (or a certain number of terms) happen to be points x_j in a particular pixel, the entirety of that pixel will be shaded, since a pixel represents the smallest possible unit of space on the “discrete” realm of a computer screen. As the random algorithm or chaos game is applied, pixels are gradually filled up, in a sense, and from the start, the set of shaded pixels grows. Oftentimes, the chaos game is only iterated until the number of shaded pixels on the computer screen *stops* growing.

An example of the chaos game at work is now demonstrated. Consider the IFS with probabilities given by the following mappings:

$$\left\{ \begin{array}{l} \hat{m}_1(x, y) = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} ; p_1 = \frac{1}{4} \\ \hat{m}_2(x, y) = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.5 \\ 0 \end{bmatrix} ; p_2 = \frac{1}{4} \\ \hat{m}_3(x, y) = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0.5 \end{bmatrix} ; p_3 = \frac{1}{4} \\ \hat{m}_4(x, y) = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} ; p_4 = \frac{1}{4} \end{array} \right. \quad (4.5)$$

The attractor from this simple-looking IFS becomes more and more clearly apparent, the more points are added to the sequence $\{x_n\}$ using the random algorithm. This is demonstrated in Figure 4.1.

Figure 4.1: From left to right, the approximation of the attractor of $\hat{\mathbf{m}}$ formulated in (4.5), with 500 points, 2000 points, and 15000 points plotted respectively

Clearly, the chaos game is a powerful tool that can be used to construct good approximations to set attractors. The chaos game can be used for more complex and interesting things, too, though, which will be described a little later in the chapter.

4.2 Code Space and Fractal Tops

A major idea of the last section involved using the random algorithm in order to closely approximate the set attractor A of a contractive IFS $(\hat{\mathbf{w}}, X)$, where (X, d_X) is a complete metric space. If there were some way to represent each different point in A , it would be possible to refer to it in order to perform operations. As a result, an interesting space is now introduced. For further discussion of the material in this section, the reader is referred to [4] and [9].

Definition 26 *Let $(\hat{\mathbf{w}}, X)$ be a contractive iterated function system with N maps. Then, the **code space** of the IFS is defined by:*

$$\begin{aligned}\Omega_N &= \{ \text{the set of all possible semi-infinite strings} \\ &\quad \text{that can be formed using the numbers } 0 \text{ through } N-1 \text{ inclusive} \} \\ &= \{ \sigma = \sigma_1 \sigma_2 \sigma_3 \dots, \text{ where } \sigma_i \text{ are integers } \in [0, N-1] \forall i \} \end{aligned} \quad (4.6)$$

A special mapping, which acts to map points in the code space to points on the set attractor of $(\hat{\mathbf{w}}, X)$, is also required.

Definition 27 *For each $x_0 \in (X, d_X)$, the **address map** is defined by $\phi_A(\sigma) : \Omega_N \mapsto$*

A , with

$$\phi_A(\sigma) = \lim_{n \rightarrow \infty} w_{\sigma_1}(w_{\sigma_2}(\dots w_{\sigma_{n-1}}(w_{\sigma_n}(x_0)))). \quad (4.7)$$

The address map ϕ takes numerical codes and transforms them into a composition of functions that takes a starting point x_0 ever closer to the attractor. Essentially, the codes σ can be used to represent different random walks taken when choosing maps using the chaos game. It should be made clear that the sequence is most properly read from right to left, which may not seem very intuitive. However, to help clarify how these address maps work, consider the following example. Take the IFS $(\hat{\mathbf{w}}, X)$ corresponding to the maps:

$$\begin{cases} w_1(x) &= 0.5x \\ w_2(x) &= 0.5x \\ w_3(x) &= 0.5x + 0.5 \end{cases} \quad (4.8)$$

$1333\bar{3}$, $3111\bar{1}$, $2333\bar{3}$, and $3222\bar{2}$ all map to the point $0.5 \in A$. (Note that it is perfectly possible to have two maps in an IFS that are identical to one another!) Different points on A will correspond to different addresses. As is demonstrated by this simple example, A could contain points with nonunique IFS codes, but it should be noted that any single code will map to one and *only* one point on A .

To help “order” all of these different codes, some way of measuring them is necessary. The following function is used:

Definition 28 Let Ω_N be a code space. Then

$$\|\sigma\|_{\Omega_N} = \sum_{i=1}^{\infty} \frac{\sigma_i}{(N+1)^i} \quad (4.9)$$

measures the size of any code $\sigma \in \Omega_N$, with the distance between two codes σ and $\omega \in \Omega_N$ defined as

$$d_{\Omega_N}(\sigma, \omega) = \|\sigma - \omega\|_{\Omega_N} \quad (4.10)$$

$\|\sigma\|_{\Omega_N}$ is quite easy to use because it induces a natural order on Ω_N . For two codes $\sigma \in \Omega_N$ and $\omega \in \Omega_N$, we say that $\sigma > \omega$ if $\sigma_i > \omega_i$ on the first instance where $\sigma_i \neq \omega_i$, where $i = 1, 2, \dots$

Now, as mentioned earlier, a single point $x \in A$ may be addressed to by many different codes. That is, for any $x \in A$, there exist a certain number M of different codes σ^j , $j = 1, \dots, M$ such that $\phi_A(\sigma^j) = x$. The set of all addresses for a given point $x \in A$, given by $\phi_A^{-1}(x)$, is closed and bounded, and thus must have a unique largest element; this is the maximum code for that set. A function may now be defined that will come in very handy in later discussions:

Definition 29 Let (Ω_N, d_{Ω_N}) be the code space for an N -map IFS $(\hat{w}, X)$ with attractor A . Let σ be an element of Ω_N and $\phi_A : \Omega_N \mapsto A$ be an address mapping. Then, define:

$$\tau_A(x) = \max\{\sigma \in \Omega_N | \phi_A(\sigma) = x\}. \quad (4.11)$$

τ_A is called the **tops function** of the IFS $(\hat{w}, X)$.

The “max” in this definition corresponds to the largest of the codes under consideration, as determined by the natural ordering of code space discussed earlier. Named by Barnsley, the tops function acts to select the largest code out of all possible codes that can be addressed to a given point x that lies on the attractor of the IFS. Thus, τ_A produces a *unique* address for every point x on the attractor.

While using the random algorithm to construct an approximation of the attractor A , it would be impossible to assign a “colouring function” that maps a colour to every point based on its address, since every point could have multiple addresses. The tops function τ_A resolves this problem.

For practical purposes, the constraints of the resolution of a computer screen must be dealt with. In a pixelization $\mathbf{P}$ of resolution r of $\square$, as established in Definition 2.2, with length and height of r pixels, an approximation of the attractor A may be plotted using the chaos game as described in Section 4.1. Each pixel $\mathbf{P}(i, j)$ (where i and j are integers in $[1, r]$) contains infinitesimally tiny points, possibly including points $x \in A$, each of these corresponding to a code uniquely defined by $\tau_A(x)$. While the points are plotted during the random algorithm, many of the pixels will typically be visited multiple times as the random maps are chosen and new points are found. In the limit as the number of iterations approaches infinity, an approximation of the tops function in pixel-space may be obtained. $\tau_A(x)$ may be approximated by:

$$\tau_A(x) \approx \max_{x \in \mathbf{P}(i, j)} (\tau_A(x)) \quad (4.12)$$

It would then be possible, implementing this approximation, to associate a colour with each of these pixels, by use of a colouring function. Immediately, the image functions defined in Chapter 2 come to mind! This connection, leading the way to a method called *colour stealing*, will be discussed in the next section.

4.3 Colouring Pictures using the Random Algorithm

Let $(\hat{\mathbf{w}}, \square)$ be a contractive IFS with its set attractor denoted as A . Given an image function $\mathcal{P}_1$, as defined in Chapter 2, *another* image function $\mathcal{P}_2$ may be produced which colours A using the colours obtained from $\mathcal{P}_1$. The scheme, due to Barnsley [4], is referred to as *colour stealing* and proceeds in the following way:

1. A contractive N -map IFS $(\hat{\mathbf{w}}, \square)$ is chosen. The attractor, A , of this IFS, is the attractor for which a colouring is desired.
2. Another contractive N -map IFS $(\hat{\mathbf{v}}, \square)$ is chosen with a set attractor denoted by B .
3. An image function $\mathcal{P}_1 : \square \mapsto \mathbb{C}$ is selected. The colours of $\mathcal{P}_1$ are the colours that will be used to colour the attractor A .
4. $\tau_A : A \mapsto \Omega_N$ and $\phi_B : \Omega_N \mapsto B$ are defined, as in Definitions 29 and 27.
5. Finally, a second image function, $\mathcal{P}_2 : \square \mapsto \mathbb{C}$, is defined by the following:

$$\mathcal{P}_2(x) = (\mathcal{P}_1 \circ \phi_B \circ \tau_A)(x), \forall x \in A. \quad (4.13)$$

This composition of functions may seem unwieldy at first glance, but the concept is not as difficult as it may first appear. The chaos game is played “on top of” the attractor A . At every point $x \in A$, the corresponding code uniquely defined by $\tau_A(x)$ is taken. ϕ_B then addresses that code to a point on the *other* attractor, B , which has already been coloured by $\mathcal{P}_1$. *That* is the colour that is used to shade the initial point x ; the series of operations is the composition of functions defined as $\mathcal{P}_2$. The same process is then applied to every starting x that is “landed on” as the random algorithm is carried out.

Figure 4.2: the point x on attractor A is coloured by “stealing” the colour from another picture, defined by $(\mathcal{P}_1 \circ \phi_B \circ \tau_A)(x)$

In practice, two chaos games are played at the same time, one each with $\hat{\mathbf{w}}$, the drawing IFS and with $\hat{\mathbf{v}}$, the colouring IFS. Each IFS has points that are produced after every iteration which jump from place to place on their respective attractors. A digital image $(\mathbf{P}_1; \tilde{\mathcal{P}}_1)$ occupying the unit square is chosen from which colours will be

stolen. The new digital image $(\mathbf{P}_2; \tilde{\mathcal{P}}_2)$ will be created that has a pixelization of the same resolution as that of $(\mathbf{P}_1; \tilde{\mathcal{P}}_1)$; let that resolution be r . A pixel, then, corresponds to a small box of points in each unit square, with an area of $\frac{1}{r^2}$. It might be desirable to begin with a blank $(\mathbf{P}_2; \tilde{\mathcal{P}}_2)$ consisting of all white pixels, though any initial digital image could be used. Then, consider iteration k of the “dual” random algorithm. Suppose that $x_k \in \mathbf{P}_2(i, j)$. If a new largest code for the pixel $\mathbf{P}_2(i, j)$ is obtained, the corresponding point $y_k \in B$ can immediately be found through the map $\phi_B \circ \tau_A$, as well as the pixel $\mathbf{P}_1(m, n)$ in which y_k lies. The colour of the pixel $\mathbf{P}_1(m, n)$, given by $\tilde{\mathcal{P}}_1(\mathbf{P}_1(m, n))$, is then stolen in order to imbue $\mathbf{P}_2(i, j)$ with colour. As the game is iterated, new biggest codes will become rarer and the digital image $(\mathbf{P}_2; \tilde{\mathcal{P}}_2)$ will cease to change. After a sufficient number of iterations, an approximation of an IFS attractor will have been drawn and shaded using the colours stolen from the digital picture $(\mathbf{P}_1; \tilde{\mathcal{P}}_1)$. The following result appears in [9].

Theorem 30 *If the two iterated function systems $(\hat{\mathbf{w}}, \square)$ and $(\hat{\mathbf{v}}, \square)$ are chosen such that $\hat{\mathbf{w}} = \hat{\mathbf{v}}$, then $\tilde{\mathcal{P}}_2$ is exactly the portion of $\tilde{\mathcal{P}}_1$ supported by the set attractor A . This can be denoted as:*

$$\tilde{\mathcal{P}}_2 = \tilde{\mathcal{P}}_1 \Big|_A. \quad (4.14)$$

Proof: $\hat{\mathbf{w}} = \hat{\mathbf{v}}$. Thus, the set attractor of both iterated function systems is the same; denote the set attractor as A . Then, $\tau_A : A \mapsto \Omega_N$ and $\phi_A : \Omega_N \mapsto A$. Therefore, the composition of maps $(\phi_A \circ \tau_A)(x) = x$ for all $x \in A$.

□

A few examples of painted attractors follow. Using carefully selected pictures to steal colours from, surprisingly beautiful colouring schemes can arise.

Example:

Consider the IFSs with probabilities given by

$$\left\{ \begin{array}{l} w_1(x, y) = \left(\frac{1.901x-0.072y+0.186}{0.563x-0.201y+2.005}, \frac{0.015x+1.609y+0.028}{0.563x-0.201y+2.005} \right); p_1 = 0.55 \\ w_2(x, y) = \left(\frac{0.002x-0.044y+0.075}{0.002x-0.088y+0.154}, \frac{0.003x-0.044y+0.104}{0.002x-0.088y+0.154} \right); p_2 = 0.01 \\ w_3(x, y) = \left(\frac{0.965x-0.352y+0.058}{1.348x-0.307y+0.075}, \frac{1.314x-0.065y-0.191}{1.348x-0.307y+0.075} \right); p_3 = 0.22 \\ w_4(x, y) = \left(\frac{-0.325x-0.0581y+0.029}{-1.281x-0.243y+0.058}, \frac{-1.229x-0.001y+0.199}{-1.281x+0.243y-0.058} \right); p_4 = 0.22 \end{array} \right. \quad (4.15)$$

$$\left\{ \begin{array}{l} v_1(x, y) = (0.8x + 0.1, 0.8x + 0.04y) \\ v_2(x, y) = (0.5x + 0.25, 0.5y + 0.4) \\ v_3(x, y) = (0.355x - 0.355y + 0.266, 0.355x + 0.355y + 0.078) \\ v_4(x, y) = (0.355x + 0.355y + 0.378, -0.355x + 0.355y + 0.35) \end{array} \right. \quad (4.16)$$

$\hat{w}$ is the drawing IFS with attractor A and $\hat{v}$ is the colouring IFS with attractor B . The digital picture $(\mathbf{P}_1; \tilde{\mathcal{P}}_1)$ consisting of 512×512 pixels coloured with RGB triplets is presented in Figure 4.3.

As the chaos game is carried out, different pixels of $\mathbf{P}_1$ will be picked out by the mapping ϕ_B and the colours of those selected pixels, determined by $\tilde{\mathcal{P}}_1$, are used to

Figure 4.3: A 512×512 digital picture with colour space $\mathbb{C} = \{0, 1, \dots, 255\}^3$. The colours in this image will be used to paint the set attractor of an IFS.

paint a pixel in the drawing of the attractor A . The result of this example is depicted in Figure 4.4.

The leaf-shaped set attractor A is shaded using colours that have been stolen from $(\mathbf{P}_1; \tilde{\mathcal{P}}_1)$. Now, if the IFS $\hat{\mathbf{v}}$ is chosen so that $\hat{\mathbf{v}} = \hat{\mathbf{w}}$, the result of Theorem 30 can be easily observed in Figure 4.5.

$\mathbf{P}_1 = \mathbf{P}_2$; the corresponding digital images have the same resolution, and indeed, $\tilde{\mathcal{P}}_2$ is exactly the portion of $\tilde{\mathcal{P}}_1$ that is supported by the set attractor A . (Note that the image appears to be rotated by ninety degrees, which is simply because of the orientation of the axes.)

The idea of colour stealing is a brand new application in the world of fractal imaging. It will be very exciting to see in the years to come what new and innovative graphical techniques may develop as a result of this interesting and unique method

Figure 4.4: The “leaf” set attractor has been painted using the colours from the digital picture.

of drawing and colouring digital pictures.

4.4 Applications of Colour Stealing

It was demonstrated in the previous section that colour stealing can paint pictures in a neat way. However, the idea poses some interesting possibilities, and may actually be extendable to some other potentially useful applications.

The possibility of using colour stealing as an information hiding scheme has been explored [9]. The idea is fairly straightforward:

1. Person A chooses a drawing IFS, a colouring IFS, and a colouring picture that contains information intended only for Person B to see. Person B must know what these IFSs are as well.

Figure 4.5: When the same IFS is used to both draw *and* colour the digital picture, the result is the portion that is supported by the set attractor.

2. Person A uses the colour stealing algorithm described in Section 4.3 to produce a coloured approximation of the attractor of the drawing IFS. This image is then transmitted to Person B.
3. Person B, knowing the IFSs, can use the same random number generator and seed value used by the generator in the colour stealing process.
4. Person B performs the colour stealing algorithm in reverse, returning the colours to an initially empty colouring picture.

Under certain conditions on the tops function and the IFSs, this “colour returning” process can reshuffle the stolen pixels back into their original configuration. This work was presented at an international conference and appears in the refereed conference proceedings [9]. More recently, M. Barnsley has investigated the idea of

homomorphisms between different IFS attractors and their associated tops functions [10].

In the next chapter, a very different problem is tackled —one of *image compression*. Different from the image generation discussed in this chapter, the methods explored will allow for the storage of image files in very efficient ways using ideas in the world of fractals.

Chapter 5

Fractal Imaging

5.1 The Collage Theorem

So far, it has been demonstrated in this thesis that if f is a contraction mapping on a complete metric space (X, d_X) , the fixed point of f may be approximated by repeatedly applying the mapping to any initial point in X . The fixed point might be some abstract set, but for some contraction mappings, such as certain iterated function systems, the fixed point can often be represented as an visually interesting picture.

In this chapter, however, a different problem is discussed: Is it possible that given any specific “target” point $x \in X$, we can find a map $f \in \text{Con}(X)$ such that x is close to the fixed point of f ? This is an inverse problem that can be stated more rigourously:

Inverse Problem:

Let (X, d_X) be a complete metric space. Given a target point $x \in X$ and $\varepsilon > 0$, find

an $f \in \text{Con}(X)$ with a fixed point $\bar{x} \in X$ such that $d_X(x, \bar{x}) < \varepsilon$.

There could be many different viable solutions to this problem; the inverse problem is not a well-posed one. It is important to observe that if $\text{Con}(X)$ is the set of all contraction maps on X , then it must contain the map that acts to send all points of X directly to the target x . Indeed, the map $f(x_0) = x$, for all $x_0 \in X$, is a contraction map on X with contraction factor $c = 0$ and fixed point $\bar{x} = x$. In practice, the set $\text{Con}(X)$ consists *only* of contraction maps of a particular form, and that particular form excludes the trivial solution described above.

Barnsley's Collage Theorem will help to shed some light on the puzzle of whether this Inverse Problem may be solved. [1]

Theorem 31 *Let (X, d_X) be a complete metric space. Given an $x \in X$, consider a mapping $f \in \text{Con}(X)$ with contraction factor $c_f \in [0, 1)$ such that $d_X(x, f(x)) < \varepsilon$, for some $\varepsilon > 0$. Denote the fixed point of f as $\bar{x}$; then*

$$d_X(x, \bar{x}) \leq \frac{\varepsilon}{1 - c_f} \quad (5.1)$$

Proof:

$$\begin{aligned} d_X(x, \bar{x}) &\leq d_X(x, f(x)) + d_X(f(x), \bar{x}) \text{ (by triangle inequality)} \\ \Rightarrow d_X(x, \bar{x}) &\leq d_X(x, f(x)) + d_X(f(x), f(\bar{x})) \text{ (\bar{x} is a fixed point)} \\ \Rightarrow d_X(x, \bar{x}) &\leq d_X(x, f(x)) + c_f d_X(x, \bar{x}) \text{ (contractivity)} \\ \Rightarrow d_X(x, \bar{x}) (1 - c_f) &\leq d_X(x, f(x)) \\ \Rightarrow d_X(x, \bar{x}) &\leq \frac{1}{1 - c_f} d_X(x, f(x)), \end{aligned}$$

and if $d_X(x, f(x))$ is taken to be $< \varepsilon$ for some $\varepsilon > 0$, the result follows. $\square$

The implication here is that the Collage Theorem allows us to reduce the error $d_X(x, \bar{x})$ by instead minimizing $d_X(x, f(x))$, subject to restrictions that ensure that c_f is sufficiently less than 1. In this sense, $d_X(x, f(x))$ is often given the special title of **collage distance**. The inverse problem is transformed by the collage theorem into the following:

Modified Inverse Problem:

Given an $x \in X$ and $\varepsilon > 0$, find a function $f \in \text{Con}(X)$ such that $d_X(x, f(x)) < \varepsilon$.

Interestingly, it has been shown that the reduction in error using the minimal-collage fixed point is in fact suboptimal. Further reading can be found in recent journal publications [11, 12].

Intuitively, it might seem that if a contraction mapping is closely approximated by a second contraction mapping, the fixed point of the first should also be well-approximated by the fixed point of the second. This is stated as a theorem [13]:

Theorem 32 *Let (X, d_X) be a complete metric space. Furthermore, let f_1 and f_2 be contraction mappings with contraction factors c_1 and c_2 , and fixed points $\bar{x}_1$ and $\bar{x}_2$ respectively. Then,*

$$d_X(\bar{x}_1, \bar{x}_2) \leq \frac{1}{1 - \min\{c_1, c_2\}} d_{X, \text{sup}}(f_1, f_2),$$

$$\text{where } d_{X, \text{sup}}(f_1, f_2) = \sup_{x \in X} d_X(f_1(x), f_2(x)).$$

This theorem says that if the distance between two mappings in a certain metric is small, then the distance between the two fixed points of those mappings has an upper bound. Thus, the fixed points of two contraction mappings must be reasonably “close together” if one of the mappings closely approximates the other!

This problem will be explored in greater depth in the next section, as ideas in the area of Fractal Imaging are explored.

5.2 Iterated Function Systems with Grey-Level Maps

One application that arises from the idea of the Collage Theorem is that of fractal image compression. With many potential real-life applications in different areas of science and technology, this area of mathematics is especially exciting. The basics are covered in this chapter, with a few interesting extensions to follow thereafter.

Given a black and white target image S , IFS maps $\{\hat{w}_1, \dots, \hat{w}_N\}$ must be found such that the attractor A of the IFS approximates S well. It seems intuitive that if $A \approx S$, then

$$A = \bigcup_{i=1}^N \hat{w}_i(A) \approx \bigcup_{i=1}^N \hat{w}_i(S) = \hat{\mathbf{w}}(S). \quad (5.2)$$

Because the IFS maps are contractive, the target S should be well-approximated by shrunken copies of itself. $\hat{\mathbf{w}}(S)$ is called the **collage of S under the IFS $\hat{\mathbf{w}}$** .

The goal is to approximate some target image by the fixed point of some contraction map, where the complete metric space involved is one with greyscale images as “points”. By finding a contraction mapping with a fixed point that is very close

to the target, Hutchinson's Theorem guarantees that upon repeated application of the contraction mapping to any initial point, the sequence of points obtained must converge to the fixed point. When Hutchinson's Theorem bends to fit in image space, the previous sentence may be rephrased as follows: Upon repeated application of the optimal contraction mapping to any initial *image*, the sequence of images obtained will converge to the fixed point, which is itself an image. The contraction mapping, then, must be found so that its fixed point is a close approximation of the target image. The target could then be represented by the parameters that define the optimal contraction map. Those parameters could then be stored in the place of the image itself, yielding potentially significant image compression.

Before plunging into fractal image compression wholeheartedly, however, it is important to realize that an image is not just a set of points; it is more complex. As was discussed in Chapter 2, an image of course possesses both geometry *and* shading. Somehow, the shading has to be taken into account. This is performed by considering the shading as a function that assigns a colour or greyscale value to every point in a set. The goal of fractal image compression, then, is to find a "double IFS" that both shrinks an image to fit a smaller portion of itself, while simultaneously adjusting the shading of each shrunken piece in a way that closely approximates the image's original grey-level values or colouration.

For relative simplicity, only greyscale images are considered in this section of the thesis. Such images are monochromatic, containing only shades of grey, and thus

do not deal with the added complication of colour. However, the ideas presented extend naturally into the world of colour graphics as well, with each of red, green, and blue shades being represented independently to one another and analogously to the simpler greyscale scheme.

In more mathematical language, assume that an image lies in a general space X . X could be the unit square $\square$ (and will be taken to be most of the time) since any rectangular image could be scaled to $\square$ with little difficulty. For a one-dimensional “image” or signal, $X = [0, 1]$ is similarly used for simplicity. In the greyscale images being discussed, every point $x \in X$ is assigned a shade of grey, in exactly the same manner as was discussed in Chapter 2. Explicitly, let $\mathbb{C} = [0, 1]$ be a colour space. Then, the shading of these images can be represented by a greyscale image function $\mathcal{P} : X \mapsto [0, 1]$, assigning points in X to points in the colour space that represent a shade of grey.

Let $\mathcal{F}(X, \mathbb{C}) = \{\mathcal{P} | \mathcal{P} : X \mapsto [0, 1]\}$ be the space of all greyscale image functions. In Definition 6, the $\mathcal{L}^2$ metric on this image function space was defined. Since that time, Definition 21 of the Hausdorff metric for sets has been presented. Some sort of modified Hausdorff metric, taking into account the underlying set of non-white points and the grey levels of those points, at first seems to be an attractive choice for measuring the distances between two image functions, until it becomes apparent that even the smallest difference between two functions could make $d_{\mathcal{H}}$ very large.

Considering the picture in Figure 5.1, the single isolated point alone makes $d_{\mathcal{H}}(R^*, R)$

Figure 5.1: The Hausdorff distance between two visually similar sets can still be large.

very large, even though visually, R^* and R are very similar. Thus, the space of greyscale images established in Definition 6 is instead chosen.

Now, we return to our previous discussion. More mathematically, given a target image, it is desirable to find an IFS with N maps that possesses an attractor A that closely approximates the target. Upon applying the collage theorem, the related inverse problem is as follows:

Inverse Problem: Given a target image function $\mathcal{P} \in \mathcal{F}(\square, \mathbb{C})$, and $\varepsilon > 0$, is it possible to find some contractive function T such that $\|\mathcal{P} - T\mathcal{P}\| < \varepsilon$?

Remarks:

1. The ideas presented are extendable to domains of $\mathbb{R}$ or even $\mathbb{R}^n$; one-dimensional “images” could be thought of as simple signals, and multidimensional signals could carry more complex information: Think of the constant streams of signals, sent through airwaves, for television, internet-related, or other purposes, in this current growing age of wireless technology.
2. In practice, recall that it is impossible to work with points on real intervals. When working in the confines of a computer-based environment, a pixelization and

discretized greyscale range must be adopted. Practical issues like this will be covered in greater depth in Section 5.4.

Application to 1-D signals:

Before jumping into the two-dimensional world, one-dimensional signals are considered. Thus, let $X = [0, 1]$ here. The aim is to build a double iterated function system that not only shrinks an image to fit a smaller portion of itself, but also adjusts the greyscale values of each shrunk piece to closely approximate the original shading of the image.

Start with an N -map geometrically-contractive IFS, denoted as $\hat{\mathbf{w}} = \{\hat{w}_1, \dots, \hat{w}_N\}$. In 5.2, an overlapping IFS with two maps is used just as a demonstration.

Figure 5.2: Each $\hat{w}_i$ defines an interval in $[0, 1]$

Now, a one-dimensional signal supported on $[0, 1]$ can be represented by $y = \mathcal{P}(x)$, where $x \in X$ and $y \in [0, 1] = \mathbb{C}$, the allowable grey range, and $\mathcal{P} \in \mathcal{F}(X, \mathbb{C})$. An example signal is demonstrated in Figure 5.3.

The following transformations are then applied:

1. For each i , $y = \mathcal{P}(x)$ is “shrunk” in order to fit onto $\hat{w}_i(X)$, the interval corre-

Figure 5.3: A one-dimensional signal $y = \mathcal{P}(x)$.

sponding to the associated $w_i(x)$. In doing so, shrunken copies of the original signal, defined by $a_i(x) = \mathcal{P}(w_i^{-1}(x))$, $x \in \hat{w}_i(X)$, are obtained. In Figure 5.4, the original signal is depicted using a dotted line and the two shrunken copies with solid lines.

Figure 5.4: Copies of the original signal are shrunken to fit on each interval.

2. The greyscale values of the shrunken copies are modified by using N functions ϕ_i (one function for every copy) that act upon each copy in order to fit the original signal $\mathcal{P}$ as closely as possible on each interval $\hat{w}_i(X)$. Though affine maps are often used

to do this, quadratic maps have also been experimented with by some researchers. In this thesis, these maps are restricted to the affine sort. New, modified shrunken copies $b_i(x)$ of the original signals are obtained, where

$$b_i(x) = \phi_i(a_i(x)) = \phi_i(\mathcal{P}(w_i^{-1}(x))), x \in \hat{w}_i(X).$$

Figure 5.5: Each copy is then fit as closely to the original signal as possible.

3. Using these shrunken and altered copies of $\mathcal{P}$, a new greyscale signal function $\mathcal{Q} \in \mathcal{F}(X, \mathbb{C})$ is built that approximates the original signal.

Complications arise when an IFS is chosen with overlapping regions $\hat{w}_i(X)$, because in that case, $\mathcal{Q}(x)$ in general would not be defined uniquely for x in those overlapping regions, as is made evident by Figure 5.6.

For all points x in overlapping regions, the average greyscale value might be taken. For example, if $\bigcap_{i=1}^M \hat{w}_i(X)$ is nonempty, then for $x \in \bigcap_{i=1}^M \hat{w}_i(X)$, let

$$\mathcal{Q}(x) = \sum_{i=1}^M \frac{b_i(x)}{M} = \sum_{i=1}^M \frac{\phi_i(\mathcal{P}(w_i^{-1}(x)))}{M} \quad (5.3)$$

Figure 5.6: Choosing an overlapping IFS can lead to questions about what happens to $\mathcal{Q}(x)$ in the overlapped region.

Alternatively, a non-overlapping IFS could be chosen in the first place, eliminating this difficulty. The series of operations is now formally defined.

Definition 33 Let (X, d_X) be a complete metric space, and let $(\hat{w}, X)$ be a contractive IFS with N mappings denoted by $\{\hat{w}_1, \dots, \hat{w}_N\}$. Let $\mathbb{C}$ be a colour space and $P : X \mapsto \mathbb{C}$ be an image function. Then,

$$\mathcal{Q}(x) = (T\mathcal{P})(x) = \sum_{i=1}^N \phi_i(\mathcal{P}(w_i^{-1}(x))), \text{ if the IFS is nonoverlapping} \quad (5.4)$$

is the minimal collage of $\mathcal{P}(x)$. T is called the **fractal transform** of $\mathcal{P}$.

Remarks:

1. Fractal image compression seeks the fractal transform that moves $\mathcal{P}$ only a small amount. Thus, we want $d_2(\mathcal{P}, T\mathcal{P})$ to be small. Then, by the collage theorem, $d_2(\mathcal{P}_g, \overline{\mathcal{P}_g})$ would be small as well, where $\overline{\mathcal{P}_g}$ is the fixed point of $T\mathcal{P}_g$, provided that the contraction factor c is sufficiently small. In practice, the contraction factor of T is controlled by choosing known geometrical contraction maps.

2. When each ϕ_i is an affine grey-level map, where

$$\phi_i(t) = \alpha_i t + \beta_i, \quad 0 \leq t \leq 1 \quad (5.5)$$

then α_i can be thought of as a “contrast” factor, as it acts to shrink or expand the range of shades of grey for each point, while β_i works to act as a “brightness” adjustment, increasing or decreasing the grey-level values by a constant.

3. The distance $\|\mathcal{P} - T\mathcal{P}\|_2$ is referred to as the **collage distance** between the signal $\mathcal{P}$ and its minimal collage $\mathcal{Q}$ using the fractal transform T . The square of the collage distance, however, is usually taken in order to avoid working with the square root.

4. The double IFS that takes into account both $\hat{\mathbf{w}}$ for geometry and $\hat{\Phi}$ for greyscale maps is often denoted by $(\hat{\Phi}, \hat{\mathbf{w}})$.

Definition 34 $(\hat{\Phi}, \hat{\mathbf{w}})$ is called an *Iterated Function System with Grey-Level Maps*, sometimes abbreviated **IFSM**.

Now, consider what happens as T is iterated. Let a sequence of image functions $\{\mathcal{P}_i\}$ be produced by letting

$$\mathcal{P}_n = (T^{\circ n} \mathcal{P}_0)(x)$$

where $\mathcal{P}_0$ is any seed image function. Then, that sequence converges as $n \rightarrow \infty$ to a unique fixed point of T , $\bar{\mathcal{P}} \in \mathcal{F}(X, \mathbb{C})$. This function is the attractor of T . This must be so by Theorem 22, because $(\hat{\Phi}, \hat{\mathbf{w}})$ is a contractive iterated function system.

In fractal image compression, T must be found such that the collage distance $d_2(\mathcal{P}, T\mathcal{P})$ is *minimized*. By doing this, a T will have been discovered that possesses

a fixed point that is only a small distance away from the original target $\mathcal{P}$, provided that the minimal collage distance is small as well. This is exactly what is required by the modified inverse problem.

Assume that the IFS $\hat{\mathbf{w}}$ is non-overlapping. Denote the collage distance as $\Delta = \|\mathcal{P} - T\mathcal{P}\|_2$. Then,

$$\begin{aligned}\Delta^2 &= \left[\int_X [\mathcal{P}(x) - T\mathcal{P}(x)]^2 dx \right] \\ &= \int_X \left[\mathcal{P}(x) - \sum_{i=1}^N \phi_i(\mathcal{P}(w_i^{-1}(x))) \right]^2 dx,\end{aligned}$$

where $w_i^{-1}(x)$ is defined only on $X_i = \hat{w}_i(X)$.

$$\begin{aligned}&= \sum_{i=1}^N \left[\int_{X_i} [\mathcal{P}(x) - \phi_i(\mathcal{P}(w_i^{-1}(x)))]^2 dx \right] \\ &= \sum_{i=1}^N \left[\int_{X_i} [\mathcal{P}(x) - \alpha_i(\mathcal{P}(w_i^{-1}(x))) + \beta_i]^2 dx \right] \\ &= \sum_{i=1}^N \Delta_i^2,\end{aligned}$$

where Δ_i is the collage distance over X_i . Thus, the collage distance can be thought of as a function of the grey-level map parameters α_i and β_i . Thus, the question of minimizing collage distance boils down to a calculus problem: For each i , α_i and β_i must be found such that Δ_i^2 is a minimum. By doing so, a T will have been found such that $\mathcal{Q}(x) = (T\mathcal{P})(x)$ is as close of an approximation to the original signal $\mathcal{P}$ as possible.

This process of finding the minimal collage using shrunken copies of a target

signal is called the *IFSM algorithm*. To help elucidate this algorithm, an example is presented.

Example (IFSM algorithm): Consider the nonoverlapping 2-map IFS $(\hat{\mathbf{w}}, [0, 1])$ with the associated mappings given by

$$\begin{cases} w_1(x) = 0.5x \\ w_2(x) = 0.5x + 0.5 \end{cases} \quad (5.6)$$

Further, consider the one-dimensional signal function $\mathcal{P}(x) = x^2$ on $[0, 1]$, displayed in Figure 5.7. First the signal is shrunk into two copies of itself by the IFS mappings

Figure 5.7: An example of a simple signal.

$\hat{w}_i$. The graph in Figure 5.8 demonstrates this. Now, the collage distance Δ (and thus Δ^2) must be minimized, so that the new signal made of shrunk copies fits as close to the original signal as possible. Taking each piece in turn, Δ_1^2 , the squared

Figure 5.8: The signal is shrunk into two smaller copies according to $\hat{\mathbf{w}}$

collage distance of the leftmost piece, is dealt with first:

$$\begin{aligned}
 \Delta_1^2 &= \int_0^{\frac{1}{2}} [\mathcal{P}(x) - \phi_1(\mathcal{P}(w_1^{-1}(x)))]^2 dx \\
 &= \int_0^{\frac{1}{2}} [x^2 - \alpha_1 (2x)^2 - \beta_1]^2 dx \\
 &= \int_0^{\frac{1}{2}} [(1 - 4\alpha_1)^2 x^4 - (2 - 8\alpha_1)\beta_1 x^2 + \beta_1^2] dx \\
 &= \frac{1}{160}(1 - 4\alpha_1)^2 - \frac{1}{12}(1 - 4\alpha_1) + \frac{1}{2}\beta_1^2 \\
 &= \frac{1}{10}\alpha_1^2 - \frac{1}{20}\alpha_1 + \frac{1}{2}\beta_1^2 - \frac{1}{12}\beta_1 + \frac{1}{3}\alpha_1\beta_1 + \frac{1}{160}
 \end{aligned}$$

Thus, Δ_1^2 is a function of two variables, the greyscale coefficients α_1 and β_1 . From calculus, the collage distance is minimized when

$$\begin{cases} \frac{\partial \Delta_1^2}{\partial \alpha_1} = 0 \\ \frac{\partial \Delta_1^2}{\partial \beta_1} = 0 \end{cases} \quad (5.7)$$

Taking these partial derivatives yields:

$$\begin{cases} \frac{\partial \Delta_1^2}{\partial \alpha_1} = \frac{1}{5}\alpha_1 - \frac{1}{20} + \frac{1}{3}\beta_1 = 0 \\ \frac{\partial \Delta_1^2}{\partial \beta_1} = \beta_1 - \frac{1}{12} + \frac{1}{3}\alpha_1 = 0 \end{cases} \quad (5.8)$$

It is now a simple task to solve the system to obtain

$$\begin{cases} \alpha_1 = \frac{1}{4} \\ \beta_1 = 0 \end{cases} \quad (5.9)$$

Thus, to map the leftmost shrunken copy, the affine map

$$\phi_1(t) = \frac{1}{4}t$$

should be used in order to minimize the collage distance, Δ_1^2 . (In this case, Δ_1^2 is in fact zero!)

This is then repeated for Δ_2^2 , the squared collage distance for the rightmost shrunken copy. Here, the interval is between $\frac{1}{2}$ and 1, and $y = \mathcal{P}(2x - 1)$ is the equation of the right-hand shrunken copy.

$$\Delta_2^2 = \int_{\frac{1}{2}}^1 [x^2 - \alpha_2 (2x - 1)^2 - \beta_2]^2 dx \quad (5.10)$$

Calculating the integral and then taking partial derivatives with respect to α_2 and β_2 , another system is obtained:

$$\begin{cases} \frac{\partial \Delta_2^2}{\partial \alpha_2} = \frac{1}{5}\alpha_2 + \frac{1}{3}\beta_2 - 0.25833 = 0 \\ \frac{\partial \Delta_2^2}{\partial \beta_2} = \frac{1}{3}\alpha_2 + \beta_2 - 0.58333 = 0 \end{cases} \quad (5.11)$$

Solving this system gives $(\alpha_2, \beta_2) = (0.71875, 0.34375)$, and thus a grey-level map of

$$\phi_2(t) = 0.71875t + 0.34375. \quad (5.12)$$

Finally, ϕ_1 is applied to the left shrunken copy and ϕ_2 is applied to the right one in order to closely approximate the original signal. The graphical results are presented in Figure 5.9.

Figure 5.9: The shrunken copies are then fit closely to the original curve using the functions ϕ_1 and ϕ_2

The curve clearly follows the general shape of the parabola that was the original signal, and illustrates the power of the mathematics above. This picture is the minimal collage of the original signal. By repeatedly applying the corresponding fractal transform T that maps the original signal to the one shown in Figure 5.9 to any initial signal, a good approximation of the original signal will be obtained.

Thus, knowing only the grey-level map coefficients and the mappings of $(\hat{\mathbf{w}}, X)$,

it is possible to create a signal that is quite close to the original, but requiring only a very small amount of information, pertaining to what makes up T .

Example:

Unfortunately, such satisfactory results as those achieved in the previous example cannot be obtained all of the time. Consider the case where $\mathcal{P}(x) = \sin(\pi x)$ is the target on $X = [0, 1]$.

Figure 5.10: Another simple (but nonmonotone) signal.

Using the same two-map IFS $(\hat{\mathbf{w}}, X)$ as in the previous example, the technique illustrated above attempts to create a minimal collage for the signal function by mapping two smaller copies of it onto each half of the signal. However, the best approximation that can be found is depicted in Figure 5.11.

This will clearly not do! The shape of the signal is not preserved at all, and this is a problem because the collage distance cannot be made arbitrarily small, which

Figure 5.11: The best-fit function of this signal does not preserve the shape of the original.

means that the collage theorem cannot be applied to say that the true error between the signal and the fixed point of such a fractal transform can be made small. This shape cannot be represented by two smaller copies of itself each supported by half the interval, because while each piece of the function is monotone, the entire signal *itself* is not. An alternative idea must be put into practice to deal with cases like these in order to obtain better approximations. That idea, of using subsets of the signal rather than the signal's entirety, forms the basis for the next section.

5.3 Local Iterated Function Systems with Grey-Level Maps

Sometimes, we cannot make the collage distance $d_2(\mathcal{P}, T\mathcal{P})$ arbitrarily small using the IFSM algorithm of Section 5.2. Some signals simply cannot be well-approximated by nonoverlapping unions of shrunken copies of themselves, due to their mathematical

nature. Instead of trying to map shrunken copies of the entire signal to each piece (each section of the signal supported by the intervals $\hat{w}_i(X)$) of the signal, shrunken copies of *portions* of the signal could be used.

In the previous example, for instance, the difficulty with the approximation was that the function was not a monotone one. Thus, instead of shrinking the entire function down and fitting two smaller copies onto each half of the signal, this alternative approach is taken.

Opening the problem to include this approach of using portions of the signal to find an approximation of the target signal is referred to as **Local Iterated Function Systems with Grey-Level Maps**, abbreviated **LIFSM**. This method for signals will be discussed in the following example, before presenting it more formally as an algorithm.

Example:

The one-dimensional example from the previous section is revisited, with $X = [0, 1]$, $\mathbb{C} = [0, 1]$, and $y = \mathcal{P}(x) = \sin(\pi x)$, for $x \in X$. First, the target signal is divided into two pieces supported on intervals I_1 and I_2 , where $I_1 \cup I_2 = X = [0, 1]$. For simplicity, each piece could be taken to be half of X , for example, with $P_1 = [0, 0.5]$ and $P_2 = [0.5, 1]$, though the signal could be divided in other ways. Then, the target function is re-divided into smaller pieces. For instance, the signal could be divided into the four portions of the signal defined on the intervals C_1, C_2, C_3, C_4 , so that again $C_1 \cup C_2 \cup C_3 \cup C_4 = X = [0, 1]$. This is depicted in Figure 5.12, and stated now

in a more formal definition.

Definition 35 *Let X be a domain for an image or signal and let $\mathbb{C}$ be a colour space.*

Let $\mathcal{P} : X \mapsto \mathbb{C}$ be a target image function. We partition X into

- 1. n child blocks $C_i, 1 \leq i \leq n$, with $\bigcup_{i=1}^n C_i = X$, and*
- 2. m parent blocks $P_j, 1 \leq j \leq m$, each larger than any child block.*

Figure 5.12: The signal is divided up into different parent blocks (P_1, P_2) and child blocks (C_1, C_2, C_3, C_4).

To each parent block P_i corresponds a geometric IFS $(\hat{\mathbf{w}}, X)$ that acts to map the interval to all of the child blocks. The aim is to choose grey-level mappings so that the portion of the signal supported by each parent block, when shrunken and translated to lie in a child block, fits the piece of the signal supported by that child block as closely as possible.

Now, the method proceeds as in the last section, except with one important dif-

ference. The goal now is to approximate each portion of the target function (each child block) by a shrunken copy of just a *piece* of the target —the parent block that does the best job of doing so.

In slightly more mathematical language, consider a target function divided into n child blocks and m parent blocks, each numbered from left to right. For each child block C_i , every parent block P_j is considered together with it as a parent-child pairing (C_i, P_j) , with $i \in \{1 \dots n\}$, and $j \in \{1 \dots m\}$ is considered. For each parent block, the grey-level maps ϕ_{ij} are found that minimize the collage distance in the same manner as described in the last section. After all minimal collage distances have been calculated for all parent blocks, the parent-child pairing that produces the *smallest* collage distance is taken to be the “best” one. Then, along with the optimal grey-level map coefficients α_{ij} and β_{ij} , the number of the best parent is also stored. A graphical representation of the resulting best-fit function of the signal in the example is presented in Figure 5.13:

The shrunken copy of one *half* of the original target signal is shrunk down to fit on each of the four sections (child blocks), and the shrunken signal is then adjusted using the corresponding grey-level adjustment ϕ_{ij} . The parent block that yields the smallest collage distance after calculating the optimal grey-level map coefficients is the one that is used. Compare Figure 5.13 with Figure 5.11, and it is visually obvious that a much smaller collage distance is obtained. Thus, a much better approximation may be obtained after iteration of the LIFSM $(\hat{\Phi}, \hat{\mathbf{w}})$ on any initial signal. In general,

Figure 5.13: Using shrunken subsets of the original signal does a far better job at creating a minimal collage than shrinking down the entire curve.

this method allows for the approximation of more complicated signals using ideas in the world of iterated function systems. Moreover, it opens the door to fractal image compression, a two-dimensional variation of the same method.

In the two-dimensional case, as discussed in Chapter 2, the unit square $\square = [0, 1] \times [0, 1]$ provides the space in which a picture will lie, and an image function $\mathcal{P}$ is considered that acts to map points $x \in \square$ to a value in the greyscale colour space $\mathbb{C} = [0, 1]$. Familiarly by now, the value 0 corresponds to the colour black, and 1 to the colour white. It is helpful to write x as a coordinate pair (x, y) instead of just x for reasons of clarity in this context of two-dimensional images, and this is the approach that will be adopted in this section. The following algorithm is developed.

[1]

The LIFSM Algorithm:

1. *The unit square is partitioned into q child blocks.*

For simplicity, assume that the squares are all non-overlapping and tile the entirety of the space perfectly without remainder, as depicted in Figure 5.14.

Figure 5.14: A partitioning of a space into a fixed grid of child blocks.

Each child block contains a portion of the target image for which a “best” parent block must be found.

2. *The image is re-partitioned into r parent blocks.*

It is very important that the parent blocks are larger than the child blocks, because otherwise, the functions mapping parent blocks to child blocks will not be geometric contraction maps! It is common that parent blocks are chosen to be four times the size of children, though any size that is larger than the child size might be used.

A choice must now be made, for there are various possible approaches in selecting which parents to use.

i) The Exhaustive Approach

An exhaustive approach may be employed, where *every* possible parent block is used. An obvious problem here is that there are an infinite number of potential parents that could be chosen when working in $[0, 1] \times [0, 1]$, a space consisting of infinitely many points. Even in the finite pixel-space of a computer screen, thousands of parents might have to be considered for each child block; the calculation will inevitably take a long time to finish unless a very powerful computer is used.

ii) Co-Centric Parent Blocks

Another approach, for every child block, considers only a single parent block that surrounds the child. Since the edges in the blocks may line up, the chances might be high that such a parent-child pair could yield a desirably low collage distance.

iii) Combination Approach

A mix of approaches i) and ii), the Combination Approach considers a number of parent blocks for every child block. Each of those parents must contain the child somewhere inside them. The parent block that yields the smallest collage distance is taken to be the optimal parent for that child.

iv) Fixed Grid

A simpler case, the image is simply re-partitioned into a grid of parent blocks in a similar fashion to the child blocks. This method allows for both ease of programming and the consideration of a potentially wide variety of different parent blocks. In the applications that follow in this thesis, the fixed grid approach is used, though it is

important to note that any method will achieve similar results.

P1	P2	P3	...

Figure 5.15: A fixed grid scheme for parent blocks divides the image up in a similar fashion as the child blocks.

3. For each child block, every single parent block is considered as a **parent-child pairing** (C_k, P_l) , where $k \in \{1..q\}$ and $l \in \{1..r\}$.

Every parent, in turn, is shrunk through a known contraction mapping to fit the size of the current child. Then, the parameters in the grey-level map ϕ_{kl} are chosen in such a way as to minimize the collage distance between the current parent and the current child.

More rigourously, denote by Δ_{kl} the collage distance between child block C_k and parent block P_l , with the geometric contraction mapping reducing the parent block to the child block denoted as w_{kl} , and the grey-level maps used to modify the greyscale values of the shrunk copy of the parent block as ϕ_{kl} . Then,

$$\Delta_{kl}^2 = \int_{C_k} \left[\mathcal{P}(x, y) - \phi_{kl}(\mathcal{P}((w_{kl})^{-1}(x, y))) \right]^2 dA \quad (5.13)$$

Now since ϕ_{kl} is taken to be an affine map, then

$$\phi_{kl}(t) = \alpha_{kl}t + \beta_{kl}. \quad (5.14)$$

Thus, Δ_{kl}^2 can be thought of as a function of two variables, α_{kl} and β_{kl} . From calculus, the minimum $\mathcal{L}^2$ distance is obtained when

$$\begin{cases} \frac{\partial \Delta_{kl}^2}{\partial \alpha_{kl}} = 0 \\ \frac{\partial \Delta_{kl}^2}{\partial \beta_{kl}} = 0 \end{cases} \quad (5.15)$$

Using (5.13) and taking the partial derivatives gives the following:

$$\begin{cases} \|\mathcal{P}\|_2^2 \alpha_{kl} + \|\mathcal{P}\|_1 \beta_{kl} = \langle \mathcal{P} \circ w_{kl}, \mathcal{P} \rangle \\ \|\mathcal{P}\|_1 \alpha_{kl} + \beta_{kl} = \|\mathcal{P} \circ w_{kl}\|_1 \end{cases} \quad (5.16)$$

The optimal α_{kl} and β_{kl} values may then be found by using Cramer's Rule:

$$\begin{cases} \alpha_{kl} = \frac{\langle \mathcal{P} \circ w_{kl}, \mathcal{P} \rangle - \|\mathcal{P} \circ w_{kl}\|_1 \|\mathcal{P}\|_1}{\|\mathcal{P}\|_2^2 - \|\mathcal{P}\|_1^2} \\ \beta_{kl} = \frac{\|\mathcal{P} \circ w_{kl}\|_1 \|\mathcal{P}\|_2^2 - \langle \mathcal{P} \circ w_{kl}, \mathcal{P} \rangle \|\mathcal{P}\|_1}{\|\mathcal{P}\|_2^2 - \|\mathcal{P}\|_1^2}, \end{cases} \quad (5.17)$$

where $\langle \mathcal{P} \circ w_{kl}, \mathcal{P} \rangle$ represents the inner product between the picture functions of the parent block and child block.

The collage distance using these optimal α_{kl} and β_{kl} coefficients is calculated and recorded, and then, the same calculations as described above are performed for *every single* parent to be considered for the current child.

4. The best parent-child pairing is found.

After the calculations in Step 3 are completed for all parent blocks, the parent yielding the smallest collage distance is deemed to be the best parent for the current

child. The number of the best parent is stored, along with the corresponding grey-level map coefficients α_{kl} and β_{kl} .

5. *The process is then repeated for every child block.*

After all children have been considered, and all parent blocks for every child block, a complete set of best parent-child pairings is obtained. This information, along with the optimal grey-level map coefficients, may be stored and *decompressed* later to obtain an approximation of the target image $\mathcal{P}$.

Remark:

In the case of square parent and child blocks, in fact, a parent block may be mapped to a child block in *eight* different contractive ways, each of which performs different flips and rotations of the parent. Differently-oriented parents will not in general produce the same collage distances, and thus recording the orientation that produces the smallest collage distance is of benefit.

Putting this somewhat involved process into practice is the subject of the next section, along with the difficulties that are associated with doing so.

5.4 Practical Issues with LIFSM

The previous section described how basic Fractal Image Compression works. The idea of imaging, though, is obviously applicable to the world of computing, when real numbers cannot be put into use. Parent and child blocks cannot possibly consist of an infinite number of points; they are made up of a *finite* number of pixels. The

theoretical discussion of fractal image compression in Section 5.3 leads to the practical discussion of actually implementing it, but to do so, some things must be taken into consideration.

As briefly mentioned previously, greyscale values cannot be expressed simply as a real-valued number in the interval $[0, 1]$, either; discretized space must be considered instead, typically with integer values between 0 and 255. In practice, it becomes quickly apparent that the LIFSM algorithm must be altered slightly in order to produce the desired results.

First, a pixelization $\mathbf{P}$ of $\square$ must be defined with resolution r , as in Section 2.2. Without loss of generality, a 256-greyscale digital image is used with resolution $r = 256$. A “256-greyscale” digital image is one with a colour space $\mathbb{C} = \{0, 1, \dots, 255\}$, with 0 corresponding to black and 255 to white. An example of such a digital image is given in Figure 5.16.

Figure 5.16: A 256×256 pixel, 256-greyscale image.

Further, a pixel image function $\tilde{\mathcal{P}} : \mathbf{P} \mapsto \mathbb{C}$ is introduced. As in Definition 9, $\tilde{\mathcal{P}}$

assigns a greyscale value to each pixel $P(i, j) \in P$. The greyscale value is a point in the colour space $\mathbb{C}$.

Following Step 1 of the LIFSM Algorithm, the digital image is divided up into a number of child blocks that tile the image perfectly. In Step 2, we choose a parent block scheme. The parent blocks must be larger than the child blocks, and though many different ways of selecting parent blocks exist, a grid pattern is chosen here as an analogous example to the grid chosen in Section 5.3. This is illustrated in Figure 5.17.

Figure 5.17: The image is carved up into a grid of parent and child blocks.

Now, for every child block C_k , each parent block P_l is considered in turn, yielding a parent-child pairing. As in the LIFSM algorithm, affine grey-level maps must be found that adjust greyscale values of the shrunk parent block to closely fit those of the child block. Denote these grey-level maps by $\tilde{\phi}_{kl} : \mathbb{C} \mapsto [0, 255]$, with

$$\tilde{\phi}_{kl}(t) = \tilde{\alpha}_{kl}t + \tilde{\beta}_{kl},$$

where $[0, 255]$ is the real-valued greyscale range containing the integer-valued colour

space $\mathbb{C}$ with the same maximum and minimum bounds; the optimal coefficients that are found might not map integers to integers (and don't, in general). The coefficients $\tilde{\alpha}_{kl}$ and $\tilde{\beta}_{kl}$ that correspond to contrast and brightness act on the greyscale value of the parent pixel, given by $\tilde{\mathcal{P}}(w_{kl})^{-1}(\mathbf{P}(i, j))$, for all pixels $\mathbf{P}(i, j) \in C_k$, the current child.

In the discrete world of a pixelization, the collage distance must be represented in terms of a summation rather than an integral. The collage distance may then be calculated by

$$\Delta_{kl}^2 = \sum_{\mathbf{P}(i,j) \in C_k} \left[\tilde{\mathcal{P}}(\mathbf{P}(i, j)) - \tilde{\phi}_{kl}(\tilde{\mathcal{P}}((w_{kl})^{-1}(\mathbf{P}(i, j))) \right]^2, \quad (5.18)$$

as first discussed in Chapter 2, where i and j are both integers in the interval $[0, 255]$. It is important to keep in mind that the choice of i and j corresponds to a unique pixel $\mathbf{P}(i, j)$ in the image.

In order to perform the necessary calculations to minimize Δ_{kl}^2 , though, a significant obstacle surfaces: What, exactly, is $\mathcal{P}((w_{kl})^{-1}(\mathbf{P}(i, j)))$? In a pixelization, it is impossible to simply reduce the size of the parent in order to compute the inner product between it and the current child, because pixels represent the elementary unit of space; the idea of “reducing” a parent block with more pixels than a child block and doing point-by-point calculations does not make sense. A parent block will always contain more pixels, and thus more information, than a child block. This is in stark contrast to the theoretical case, when parent and child blocks consist of an infinite number of points in a subset of the real plane. Hence, under the inverse

map $(w_{kl})^{-1}$, the parent information $\mathcal{P}((w_{kl})^{-1})$ is not completely known. Instead, **decimation** schemes must be considered, by which some information must be lost. Two such schemes will be outlined, but first, some variables must be defined.

Let C be the child block currently being considered and P be the current parent block. Temporarily, rename the pixels in C and P , so that they are numbered in “natural counting order” from bottom-left to top-right as $C^{(i)}$ and $P^{(j)}$, $i \in \{0, \dots, N_C - 1\}$, $j \in \{0, \dots, N_P - 1\}$, where N_C is the total number of pixels in the child block, and N_P is the total number of pixels in the parent block. (Note that any $C^{(i)}$ or $P^{(j)}$ could be re-written as a unique pixel using the “**P**” notation introduced before; they are renamed here purely for convenience.) Define k to be the integer-valued magnification factor between the size of the child block and the parent block width. For example, if parent blocks are 16×16 pixels, and child blocks are 4×4 pixels, then k would be 4. Finally, let z be the width chosen for any child block.

Possible Decimation Schemes:

1. *Crude Decimation*

The crude decimation scheme throws away a lot of information in favour of speedy calculation and programming ease. The inner product is calculated as follows:

$$IP_{crude} = \sum_{i=0}^{N_C-1} \tilde{\mathcal{P}}(C^{(i)}) \tilde{\mathcal{P}}(P^{(ki)}) \quad (5.19)$$

That is, every child pixel is multiplied by every k^{th} parent pixel, adding them together, associating similarly shaded pixels, as is illustrated in Figure 5.18.

Figure 5.18: Crude Decimation doesn't take into account most of the pixels in the current parent block. An unrotated, unflipped parent orientation is assumed in this picture.

A scheme like this is executed quickly, but is quite brutal in that it does not even consider the majority of parent pixels in the calculation; much detail can be lost.

2. Averaging Decimation

A different, more complicated scheme can be used in order to include more information from each parent in the calculation instead of entirely disregarding some pixels. The inner product is calculated as:

$$IP_{avg} = \sum_{i=0}^{N_C-1} \tilde{P}(C^{(i)}) \left(\sum_{n=0}^{k-1} \sum_{j=0}^{k-1} \frac{\tilde{P}(P^{(ki+j+nz)})}{k^2} \right) \quad (5.20)$$

This scheme might appear to be an alphabet soup of variables at first glance, but the idea is simple: The nearby parent pixels are all averaged, and the average value is used in the calculation of the inner product against the corresponding child pixel.

Figure 5.19 demonstrates this association.

This method takes longer to execute in a computer program, but none of the pixels are simply thrown away, creating a more “accurate” inner product calculation, in a sense.

Figure 5.19: Averaging Decimation uses all of the parent pixels in the calculation of the inner product. Again, an unrotated, unflipped parent orientation is assumed here.

Returning to the previous notation, optimal $\tilde{\alpha}_{kl}$ and $\tilde{\beta}_{kl}$ values are easy to calculate now, using either method of decimation. If (C_k, P_l) is taken to be the current parent-child pairing, then assuming the Averaging Decimation Approach is used, the following formulae are obtained:

$$\begin{cases} \tilde{\alpha}_{kl} = \frac{IP_{avg} - \|\tilde{\mathcal{P}} \circ w_{kl}\|_1 \|\tilde{\mathcal{P}}\|_1}{\|\tilde{\mathcal{P}}\|_2^2 - \|\tilde{\mathcal{P}}\|_1^2} \\ \tilde{\beta}_{kl} = \frac{\|\tilde{\mathcal{P}} \circ w_{kl}\|_1 \|\tilde{\mathcal{P}}\|_2^2 - IP_{avg} \|\tilde{\mathcal{P}}\|_1}{\|\tilde{\mathcal{P}}\|_2^2 - \|\tilde{\mathcal{P}}\|_1^2}, \end{cases} \quad (5.21)$$

All eight possible orientations of every parent are taken into account and the optimal collage distance is calculated for each. The parent block yielding the smallest collage distance is taken to be the best parent and is recorded along with its orientation, and the optimal $\tilde{\alpha}_{kl}$ and $\tilde{\beta}_{kl}$ values. In the example digital image, the best parent for Child 19, as an example, is depicted in Figure 5.20. Child 19 is outlined in white while the best parent for Child 19 is outlined in black.

To carry out these extensive calculations, time is needed. Adding more considerations (including different orientations, using the Averaging Decimation Approach,

Figure 5.20: An example of a best parent-child pairing.

etc.) increases the time required to execute the algorithm. One way to reduce the time is to decide upon a threshold value, δ , for the squared collage distance Δ_{kl}^2 . Then, when considering any parent-child pair, if the minimal value for Δ_{kl}^2 is determined to be less than δ , the pairing is considered to be adequate, and the search for a better parent-child match for the current child is discontinued. If no parent-child pairing yields a $\Delta_{kl}^2 < \delta$, then the parent-child pairing with the minimal collage distance is taken as usual.

Fractal image compression requires the storage of a relatively small amount of data: For every child, the number of the best parent is stored, as well as the number corresponding to the geometrical contraction map that was used (a reference to the orientation of that parent block), and the optimal grey-level map parameters. This list of information is referred to as the *fractal code*. The fractal code, of course,

depends upon the framework chosen.

5.5 Decompression

Once fractal image compression has been completed, decompression is a very simple task. The IFS obtained through the LIFSM algorithm is iterated upon any initial seed image that possesses the same resolution and colour space as the target image. At every iteration, blocks of the seed image are shrunk down and mapped to smaller blocks, determined by the fractal code. The corresponding affine maps are applied to the blocks appropriately. Since the IFS obtained is globally attractive with a unique fixed point (one that is as close as possible to the target image), the seed image may be any digital image at all; even a completely black or white image will work! After iterating a sufficient number of times, an approximation to the original is obtained, and the digital image is said to be *decompressed*.

The minimal-collage grey-level mappings will not in general map integers in the greyscale range to integers. Indeed, the obtained shading IFS may determine that, for example, a pixel should be shaded using a value of $113.878 \in [0, 255]$. Pixels cannot, however, be shaded by a computer using fractional greyscale values. Thus, after each iteration, the value of $\tilde{\phi}_{kl} \circ \tilde{\mathcal{P}}(\mathbf{P}(i, j))$ is rounded to the nearest integer.

With every iteration, the seed image is mapped “closer” to its attractor, and thus, the digital image becomes gradually clearer. Using the fractal code obtained from the example digital image depicted in Figure 5.16 and a purely black seed image, observe

the progression after one, two, and ten applications, shown in Figure 5.21.

Figure 5.21: Images obtained after applying the optimal IFS one, two, and ten times to a seed image.

The quality of the approximations in Figure 5.21 is obviously not terribly good, but if the parent blocks and child blocks are reduced in size, the result improves quite significantly, as illustrated in Figure 5.22. The digital image is again displayed after application of one, two, and ten iterations of the code.

Figure 5.22: An analog to Figure 5.21 but with smaller parent block and child block sizes.

The tradeoff, of course, is that the compression is not as great. More information must be stored to obtain an improved approximation, due to an increased number of child blocks. In the first example with the larger parent and child blocks, the compressed file is just 29 kB in size while the uncompressed target image is far larger;

the necessary storage space is reduced by a significant amount. With the smaller parent and child blocks, and accompanied improvement in image approximation, however, the compression decreases. Thus, for purposes of compression, the algorithm outlined in this chapter is most useful when only a rough approximation of an image is needed, though fractal image compression performs similarly to the JPEG standard at moderate compression levels. At high levels of image compression, the JPEG format does perform better at retaining image quality. However, such fractal algorithms are implemented as part of the JPEG-2000 standard. In addition, these techniques have been used to encode streaming video, such as videophone technologies and the RealMedia format used by the Real Player.

In addition, the ideas behind LIFSM provide the basis for some interesting and potentially very useful applications outside of compression as well. Some of these applications will be discussed in the next chapter.

Chapter 6

Applications of Near-Optimal Parent-Child Pairs

While image compression uses each minimal-collage parent-child pair to build a fractal transform, it is the case that there are typically many parent-child pairs that produce a collage distance that is very close to the minimum. This fact isn't very helpful when compression is the goal, though, because the point of compression is to keep the fractal code as small as possible; storing this extra information does not contribute any benefit. In this chapter, however, possible benefits of exploiting "near-optimal" pairs will be explored in applications other than image compression.

6.1 An Introduction to Fractal Image Denoising

The ideas behind Fractal Image Compression present the opportunity for other applications besides the simple compression of image files. One of those applications is image denoising.

In the world today, a world of endless transmissions through wires and wireless

technology, it is often the case that signals and images that are transferred are not transferred perfectly. Different obstacles may cause *interference* or *noise*, resulting in a received signal that is not completely true to the original. A “noisy” signal can manifest itself in such everyday things as a snowy image on a TV screen, or as a static crackle during a radio broadcast during a storm. If it were possible, the ability to produce a clearer image out of one that is obscured by noise would truly be a feat, and doing so without any prior knowledge of what the true (unobscured) signal “should be” might seem to require a magic trick. However, fractal image denoising is often able to achieve just that, sometimes with surprising success. By using the ideas of self-similarity and pairing parent and child blocks from last chapter, along with new ideas of near-optimal pairs which will be explored in this chapter, a good approximation of the original image may often be recovered from a noisy version of that image.

In this chapter, only two-dimensional greyscale digital images will be considered. However, the ideas of fractal image denoising extend to many different realms, including one-dimensional or multi-dimensional signals, colour images, and audio.

6.2 Noise and Noisy Images

Before beginning, a mathematical definition of what “noise” actually comprises is necessary. In this thesis, noise is a function involving random variables that act to alter (or interfere with) an existing image function.

Definition 36 Let $\mathbf{P}$ be a pixelization of the unit square $\square$ with resolution r . Denote any single pixel of $\mathbf{P}$ as $\mathbf{P}(i, j)$, with $i \in \{1, \dots, r\}$ and $j \in \{1, \dots, r\}$. Consider an image function $\tilde{\mathcal{P}} : \mathbf{P} \rightarrow \mathbb{C}$, where $\mathbb{C}$ is a colour space. Further, define r^2 normally distributed random variables $\nu_{i,j}$, with a mean of 0 and variance of σ^2 . Finally, let $\mathbb{C}^+$ be the largest element and $\mathbb{C}^-$ be the smallest element of the colour space $\mathbb{C}$ (in the Cartesian norm). Then,

$$\tilde{u}(\mathbf{P}(i, j)) = \begin{cases} \mathbb{C}^-, & \text{if } \tilde{\mathcal{P}}(\mathbf{P}(i, j)) + \nu_{i,j} \leq \mathbb{C}^- \\ \mathbb{C}^+, & \text{if } \tilde{\mathcal{P}}(\mathbf{P}(i, j)) + \nu_{i,j} \geq \mathbb{C}^+ \\ \tilde{\mathcal{P}}(\mathbf{P}(i, j)) + \nu_{i,j}, & \text{otherwise} \end{cases} \quad (6.1)$$

is the **noisy image function** for all pixels $\mathbf{P}(i, j) \in \mathbf{P}$.

In essence, each random variable $\nu_{i,j}$ determines exactly how much noise affects the associated pixel $\mathbf{P}(i, j)$. $\tilde{u}(\mathbf{P}(i, j))$ is then the “new” shade of grey assigned to the pixel, which will differ from $\tilde{\mathcal{P}}(\mathbf{P}(i, j))$ unless $\nu_{i,j}$ happens to be zero. The amount, on average, that each pixel differs will depend on the variance of the distribution of $\nu_{i,j}$, given by σ^2 . $\tilde{u}(i, j)$ is limited on each side by the boundaries of the allowable greyscale range, $\mathbb{C}^-$ and $\mathbb{C}^+$, which, using the greyscale colour space $\mathbb{C} = \{0, \dots, 255\}$, we take to be 0 and 255 respectively.

σ^2 could be thought of as the *intensity* of the noise affecting the image. Larger values of σ^2 result in a higher probability that pixels will be changed significantly, and thus, to the human eye, the “snowiness” of the noisy image appears to be worse.

Using the example image from the previous chapter, noise is demonstrated in Figure 6.1 at different values of the standard deviation σ . The images obviously become increasingly obscured as the variance of the random variables $\nu_{i,j}$ increases.

Figure 6.1: Noisy Images created using $\nu_{i,j}$ normally-distributed random variables with standard deviation $\sigma = 10, 25$, and 40 , respectively, are pictured from left to right.

The aim of fractal denoising is then to use ideas from fractal image compression in order to help reduce the amount of interference, producing an image that is more true to what it should be. To measure how different one greyscale image is from another, a greyscale image metric must be used; the natural choice is the formula given in Definition 12.

The goal of image denoising is to find an image that is closer to the original image under this metric than the noisy image, using *only* the noisy image to achieve this.

6.3 The Denoising Algorithm

At last, the denoising algorithm will be discussed. The method actually proceeds in a very similar fashion to the LIFSM algorithm detailed in Chapter 5. In fact, it is

identical to the LIFSM algorithm except for when it comes to step four and beyond.

In fractal image compression, this step determines the best parent-child pairing. This is the parent-child pairing (C_k, P_l) that yields the smallest collage distance. In a fractal denoising algorithm, however, this step goes further. For the current child, after all calculations are completed for all parent-child pairings, the values of Δ_{kl}^2 for every pairing are considered. Instead of taking only the optimal pairing (that is, the parent-child pair that produces the smallest value of Δ_{kl}^2), a number of *near-optimal pairings* are also taken. These are the parent-child pairings that produce the second-least collage distance, third-least collage distance, and so on. Any number of near-optimal pairings can be taken; let that number be denoted as p . Then, the numbers of the best parents are all stored, along with the corresponding optimal grey-level map coefficients α_{kl_i} and β_{kl_i} , where $i \in \{1, \dots, p\}$. All of these parents will be factored into play during the reconstruction phase. An example showing a child block and the corresponding best parent blocks is demonstrated in Figure 6.2.

Figure 6.2: Parent Block size: 16 x 16 pixels; Child Block Size: 8 x 8 pixels. A child block is outlined in white. The eight parents that produce the best parent-child pairings with that child are outlined in black.

The calculations to discover the p near-optimal parent-child pairs are then repeated for every child block.

After all of the steps of the algorithm are repeated for every single child block, a complete set of the qp near-optimal parent-child pairings is obtained (recalling that q is the total number of child blocks in the digital image). This information, along with the optimal grey-level map coefficients, may be stored in a file. The digital image may then be reconstructed either immediately or later, using the information in that file, to obtain a more accurate approximation of the original pixel image function $\tilde{\mathcal{P}}$.

6.4 Reconstruction

Once the file described above has been completed, reconstructing the image is not a difficult thing to do. Through the denoising algorithm, p contractive iterated function systems $(\hat{\mathbf{w}}, \hat{\Phi})_a$, $a = 1, \dots, p$, are formed, with $(\hat{\mathbf{w}}, \hat{\Phi})_1$ corresponding to the IFS obtained using the best parent-child pairings, $(\hat{\mathbf{w}}, \hat{\Phi})_2$ corresponding to that using the second-best pairings, and so on through the top p . Now, the parent blocks are chosen such that they are the p best possible selections (i.e. the collage distance is as small as possible). Thus, it might be expected that while $(\hat{\mathbf{w}}, \hat{\Phi})_1$, the IFS using the best possible pairings, will produce the best approximation of the initial image, it must be remembered that the initial image was a noisy one! Pixels all differ from the true image by some random value determined by the random variables $\nu_{i,j}$.

How is this effect countered? Since the noise influencing the image is assumed to

be random with a mean value of zero, then by considering the approximations produced by other near-optimal parent-child pairings, through the IFSs $(\hat{\mathbf{w}}, \hat{\Phi})_1, (\hat{\mathbf{w}}, \hat{\Phi})_2, (\hat{\mathbf{w}}, \hat{\Phi})_3, \dots, (\hat{\mathbf{w}}, \hat{\Phi})_p$ and *averaging* the resulting greyscale values at each pixel with every iteration of the IFS (with the first iteration upon some seed image of the same dimensions), a clearer image can be obtained after repeating the process a sufficient number of times.

More mathematically, we note that the fractal code gives us p near-optimal fractal transforms T_a , $a \in \{1, \dots, p\}$, each representing the transformations associated with the IFS $(\hat{\mathbf{w}}, \hat{\Phi})_a$. Each of these fractal transforms will map the greyscale values of the pixels of any seed digital image to new values. To utilize all of these at once while recreating the image, we take the average of them all. That is, we let

$$T_{avg}(\tilde{\mathcal{P}}_0) = \sum_{a=1}^p \frac{T_a(\tilde{\mathcal{P}}_0)}{p} \quad (6.2)$$

where $(\mathbf{P}; \tilde{\mathcal{P}}_0)$ is any initial digital image with a pixelization $\mathbf{P}$ of the same resolution as the target digital image and a pixel image function $\tilde{\mathcal{P}}_0$ that maps to the same colour space as the target digital image. By applying T_{avg} repeatedly to the initial digital image, then the resulting approximation of the target image should be pretty good, as long as the p fractal transforms all have small collage distances. T_{avg} approaches a unique image in $\tilde{\mathcal{F}}(\mathbf{P}, \mathbb{C})$ if and only if T_{avg} is contractive.

Theorem 37 *The function defined by*

$$T_{avg}(\tilde{\mathcal{P}}) = \sum_{a=1}^p \frac{T_a(\tilde{\mathcal{P}})}{p}$$

is a contraction mapping on $\mathcal{F}(\mathbf{P}, \mathbb{C})$, where T_a is the fractal transform yielding the a^{th} -smallest collage distance.

Proof: T_{avg} is the average of p contraction maps all with the same geometrically determined contraction factor c under our fixed grid parent selection scheme, equal to child size divided by parent size, and so T_{avg} has a contraction factor of c as well. Therefore, T_{avg} is contractive on $\mathcal{F}(\mathbf{P}, \mathbb{C})$. □

Thus, the function defined in Theorem 37 is indeed a contraction mapping, and the sequence of digital images obtained by iterating T_{avg} does indeed converge to an attractor digital image when applied repeatedly to any initial seed image.

6.5 Discussion

The random noise variables $\nu_{i,j}$ are assumed to be normally distributed. Because of this, it becomes more and more unlikely that the average of p noisy parents will preserve the most inaccurate pixels as the number of averaged parents increases. This is true as long as the best p parents all provide reasonably low collage distances under the optimal grey-level maps that are applied. This is clarified by the following simple example.

Example:

For a normal distribution with mean $\mu = 0$, the probability that any single observation from a sample deviates from μ by the standard deviation, σ , or greater is

approximately 0.32. So, the probability that any single sample is *greater* than μ by one standard deviation or more is approximately 0.16.

Similarly, in this application, consider any pixel $P(i, j)$ in a given child block. The probability that the corresponding (noise-influenced) grey-level of the pixel in the best parent block is greater than the “true” grey-level, $\tilde{P}(P(i, j))$, by more than σ is then, on average, approximately 0.16. Imagine now, instead, that the best *two* parents are averaged for every child, and that those two parents provide good approximations. Then, for that amount of error to appear in the final approximation, the grey-level map of the second-best parent would also have to be “off” on the same corresponding pixel by at least σ as well. The chances of that occurring are approximately $0.16 \times 0.16 = 0.0256$, and it quickly becomes clear that it would be highly unlikely, with randomly distributed noise, that this could happen. By averaging high-quality parent blocks (those that produce low collage distances), the noise is, most of the time, diminished, by the Law of Large Numbers.

Figure 6.3: Parent Block size: 8 x 8 pixels; Child Block Size: 4 x 4 pixels. Pictured from left to right: The original digital image, the noisy image, and the denoised image, averaging the best 25 parent blocks.

Some examples of these improvements are finally exhibited in Figure 6.3. Visually, the smoothness of the 256-greyscale image is remarkable in contrast to the noisy picture, produced with standard deviation $\sigma = 25$; the snow appears to have been completely removed from the picture. Let $\tilde{\mathcal{R}} : \mathbf{P} \mapsto \{0, 1, \dots, 255\}$ be the image function that gives the greyscale value of the denoised image pictured in Figure 6.3 at the pixel $\mathbf{P}(i, j)$. Mathematically, the recovery can be verified if

$$d_2(\tilde{\mathcal{P}}, \tilde{\mathcal{R}}) \leq d_2(\tilde{\mathcal{P}}, \tilde{\mathbf{u}})$$

where d_2 is the metric given in Definition 12. It is possible to measure this distance exactly using a computer utility. For this example, the results are listed in the following table:

$d_2(\tilde{\mathcal{P}}, \tilde{\mathbf{u}})$	$d_2(\tilde{\mathcal{P}}, \tilde{\mathcal{R}})$	Noise Reduction
9857.984	4349.951	55.87%

The denoised image is 55 percent closer to the original image than the noisy image, under the metric used. This demonstrates just how successful this method can be at creating more accurate images out of ones afflicted by random interference.

In this chapter, interference is taken to be randomly generated, adding or subtracting from the values of the greyscale value of each pixel $\mathbf{P}(i, j)$ based on the variable $\nu_{i,j}$. However, it is possible that the alterations from these variables could cause the modified grey-level values $\tilde{\mathbf{u}}(\mathbf{P}(i, j))$ to be greater than 255 or less than 0, in which

case, $\tilde{u}(\mathbf{P}(i, j))$ is taken to be simply 255 or 0 respectively, as dictated by Definition 6.1. This keeps the greyscale values inside the allowable range, but the result is that there is a greater number of pixels shaded completely black or white as the variance of the random variable $\nu_{i,j}$ increases. The resulting appearance is remarkably similar to that of a snowy TV screen. This effect is depicted in Figures 6.4 and 6.5.

Figure 6.4: Parent Block size: 8 x 8 pixels; Child Block Size: 4 x 4 pixels. Pictured from left to right: The noisy image produced with $\sigma = 100$, and the denoised image using the best 20 parent blocks.

As the interference increases, the image becomes harder and harder to recover accurately. The noisy image in Figure 6.4 was produced by using a normal distribution for the variable $\nu_{i,j}$ with $\sigma = 100$. When σ is increased to 200, as in Figure 6.5, the recovery is still good, but to the eye, it becomes extremely difficult to detect.

The results of the attempts at image recovery for both cases are displayed in the following table.

Figure 6.5: Parent Block size: 8 x 8 pixels; Child Block Size: 4 x 4 pixels. Pictured from left to right: An even noisier image produced with $\sigma = 200$, and the denoised image using again the best 20 parent blocks.

σ	$d_2(\tilde{\mathcal{P}}, \tilde{\mathbf{u}})$	$d_2(\tilde{\mathcal{P}}, \tilde{\mathcal{R}})$	Noise Reduction
100	20231.82	6830.10	66.24%
200	27216.70	10128.89	62.78%

It is clear that if the noise changes the face of the image too severely, good image recovery becomes a nigh-impossible task. However, significant improvements can still be made, and even in cases of moderate noise, noticeable differences are easily demonstrated.

In conclusion, image recovery is a potentially powerful application that can aid in the reconstruction of images with data that has been affected adversely by interference. In the next section, however, one more fractal imaging application that seeks to restore images that are actually *missing* pieces of data entirely will be explored.

6.6 An Introduction to Fractal Image Recovery

The final application of fractal imaging that will be discussed in this thesis is one that tackles a tough problem. Suppose one has an image with unreliable, unreadable content or that the image is missing pieces of information completely. Then, is there a way to recover the image, building a new image with a complete set of information that is close to the true image, using ideas in fractal imaging?

The real-world application here is obvious. Image files are sometimes not transferred correctly over the internet for various reasons, resulting in files that are corrupted and often unreadable. Very old photographs may have holes or tears that are impossible to restore —impossible, that is, unless a way to somehow fill in the lost information can be found.

In fact, there is a way to accomplish this. The denoising scheme discussed earlier in this chapter can extend to meet this task. Using the “good” data from the corrupted image, the aim is to find a fractal transform that strategically selects only those parent blocks that do not contain any “bad” data. The fixed point of that fractal transform will, again, be a unique digital image. Once a minimal-collage contractive IFS is found using the good parents, then iterating repeatedly will produce an approximation of that fixed point, except that information will be written into the parts of the image that were previously corrupt. The result is a complete image that approximates the original well.

The set of bad pixels is now defined.

Definition 38 Given a pixelization $\mathbf{P}$ of the unit square $\square$ of resolution r , labeling the pixels $\mathbf{P}(i, j)$, $i \in \{1, \dots, r\}$, $j \in \{1, \dots, r\}$, define a new set by

$$\mathbf{P}_b = \{\mathbf{P}(i, j) \in \mathbf{P} \mid \mathbf{P}(i, j) \text{ is bad}\}. \quad (6.3)$$

$\mathbf{P}_b$ is the set of *bad pixels* of $\mathbf{P}$.

The number of bad pixels is denoted $|\mathbf{P}_b|$.

Any set of pixels can be taken to be bad, and thus they could be selected randomly. They could also be arranged together, representing a “hole” in an image. In this thesis, $\mathbf{P}_b$ is taken to be a small box-shaped set of pixels, as is demonstrated in Figure 6.6. Bad pixels represent places where the pixel image function of the original digital image is left undefined.

Figure 6.6: A 256×256 pixel, 256-greyscale image, with a small 4×4 -pixel block labeled as “bad.” The bad pixels are coloured white.

The LIFSM algorithm proceeds in the familiar way, with a few minor differences. For every child block, every parent block is considered with it as a parent-child pair, *unless* the parent contains pixels in the bad pixel set $\mathbf{P}_b$. Again, the p parent blocks

that produce the smallest collage distances with the child block are considered as near-optimal pairs.

If a child block contains pixels in the set $\mathbf{P}_b$, the aim is still to find the parent blocks that are closest to the child block; however, only the good pixels in the child block may be utilized. The calculation in (5.20) must be modified slightly so that the inner product is only made using the pixels with good data. Using all of the variables previously defined for (5.20), the formula becomes

$$IP_{avg} = \sum_{C^{(i)} \notin \mathbf{P}_b} \tilde{\mathcal{P}}(C^{(i)}) \left(\sum_{n=0}^{k-1} \sum_{j=0}^{k-1} \frac{\tilde{\mathcal{P}}(P^{(ki+j+nw)})}{k^2 - |\mathbf{P}_b|} \right). \quad (6.4)$$

Once the algorithm is complete, a set of pq parent-child pairings is obtained, where q is the total number of child blocks. The information stored in the fractal code is identical to that stored for fractal image denoising; no information needs to be kept pertaining to which pixels are bad. Because of the selection scheme utilized, no parent block will contain any bad pixels at all. Hence, the IFS obtained using this method is defined for *every* pixel, and upon iteration of this IFS on a seed digital image, new information is written into the bad pixels, creating a digital image that contains no bad pixels at all. An example is presented in Figure 6.7.

To avoid the block-effect that is introduced through the algorithm, we can define a new digital image as follows.

Definition 39 *Let $\mathbf{P}$ be a pixelization of $\square$ of resolution r , and let $(\mathbf{P}; \tilde{\mathcal{P}})$ be a digital image with a set of bad pixels $\mathbf{P}_b$ where the pixel image function $\tilde{\mathcal{P}} : \mathbf{P} \mapsto \mathbb{C}$ is not*

defined. Let $\mathbb{C}$ be a colour space, and let $\tilde{\mathcal{R}} : \mathbf{P} \mapsto \mathbb{C}$ be the image function produced by iterating the corresponding minimal collage IFS upon any initial digital image. Then, the **recovered pixel image function** $\tilde{\mathcal{S}} : \mathbf{P} \mapsto \mathbb{C}$ is given by

$$\tilde{\mathcal{S}}(\mathbf{P}(i, j)) = \begin{cases} \tilde{\mathcal{P}}(\mathbf{P}(i, j)), & \text{if } \mathbf{P}(i, j) \notin \mathbf{P}_b \\ \tilde{\mathcal{R}}(\mathbf{P}(i, j)), & \text{if } \mathbf{P}(i, j) \in \mathbf{P}_b \end{cases} \quad (6.5)$$

The resulting digital image can also be found in Figure 6.7.

Figure 6.7: Eight near-optimal pairs of parents are used in this recovery, which no longer contains the area of bad pixels that was evident in Figure 6.6. The original digital image with bad pixels is pictured on the left; the approximation of the digital image is pictured in the centre, and the recovered digital image is on the right.

In practice, this technique could prove to be very useful. It is the case that if a digital image is corrupted or contains bad pixels, it may not load whatsoever. A fractal image recovery application, however, could help prevent this from happening. A computer program could, for example, check any image before loading it to ensure that it does not contain any bad information. If it does, however, then instead of choosing not to display the picture, the program could run a recovery algorithm like the one described above. Then, at least, a good approximation to the original image

is provided that contains only good data.

Fractal image compression lends itself to many interesting, exciting, and potentially very useful applications. The two applications detailed in this chapter clearly exemplify the promise held by these techniques. Further extensions of these applications will be touched upon in the next chapter, which will conclude this thesis by discussing possibilities for future work in the area of imaging.

Chapter 7

Future Work

There is much more to be done in the area of Fractal Imaging. While this thesis has presented several new and innovative avenues, many of these ideas may be extended in future work. A brief list of possibilities is as follows.

1. Denoising with other Noise Distributions

In Chapter 4, we explored ways of denoising images with a noise function determined by normal variables with a mean μ of 0. However, it is possible that given a noise function of another distribution with a mean of $\mu \neq 0$, a good approximation might still be made. For example, with noise that is normally distributed with a mean of $a \neq 0$, we might expect that, following the denoising algorithm of 6, the averaging might produce a digital image whose pixels' greyscale values are, on average, different from those of the original image by a . Thus, by subtracting this value from the greyscale value of all pixels in the denoised image, a good approximation of the original might be obtained.

2. Image Recognition

A new area of imaging generating much current interest is that of image recognition. Based on an image function, the content of an image might be determined. This has many applications in many different areas. For example, medical image or fingerprint archives might be quickly searched for certain characteristics or features, using ideas in imaging. Hospitals are now storing digital versions of X-rays, CT scans, and the like, and new companies have arisen in recent years that are trying to implement such capabilities using fractal and/or wavelet-based imaging techniques.

3. Fractal Audio and Video Compression and Applications

The ideas explored in imaging might be naturally extendable to the world of sound or video. Using a similar foundation in Banach's Fixed Point Theorem and the Collage Theorem, it may be possible to find new ways to compress audio or video files. Going further, fractal audio or video denoising is a potentially *very* interesting topic, since radio signals are regularly obscured to this day by interference. Perhaps it is possible to use the applications posed in Chapter 6 in order to combat this effect.

4. Wavelet Exploration

Wavelets provide a commonly used basis in which images can be represented conveniently, and the theory behind wavelets is utilized in other types of image

compression. Work has already been done to combine the applications of fractal image compression with some of these other ideas somehow in order to establish new, hybridized ideas that utilize the two approaches.

The following is the main C++ code that uses the ideas from Chapters 5 and 6 to compress, denoise, or recover images. FreeImage, an open-source C++ library, was used to help process the images.

```
//-----
//
// Name:      FractalImageDlg.cpp
// Author:    Matthew Demers
// Created:   1/14/2006 2:20:46 PM
//
//-----

#include "FractalImageDlg.h" #include <iostream> #include <stdio.h>
#include <stdlib.h> #include <math.h> #include <time.h> #include
<wx/string.h> #include <freeimage.h>

//Do not add custom headers.
//wx-dvcpp designer will remove them
////Header Include Start
////Header Include End

//-----
// FractalImageDlg
//-----

//Add Custom Events only in the appropriate Block.
// Code added in other places will be removed by wx-dvcpp
////Event Table Start
BEGIN_EVENT_TABLE(FractalImageDlg,wxDialog)
////Manual Code Start
////Manual Code End

EVT_CLOSE(FractalImageDlg::FractalImageDlgClose)
EVT_CHECKBOX(ID_BADPIXELCHECKBOX,FractalImageDlg::
    BadPixelCheckBoxClick)
EVT_CHECKBOX(ID_PARENTCHILDMATCHCHECKBOX,FractalImageDlg::
    ParentChildMatchCheckboxClick)
EVT_BUTTON(ID_ENCODEBUTTON,FractalImageDlg::EncodeButtonClick)
EVT_COMMAND_SCROLL(ID_CHILDSLIDER,FractalImageDlg::
    ChildSliderScroll)
```

```

EVT_COMMAND_SCROLL(ID_PARENTSLIDER,FractalImageDlg::
    ParentSliderScroll)
EVT_BUTTON(ID_LOADIMAGEBUTTON,FractalImageDlg::
    LoadImageButtonClick)
END_EVENT_TABLE()
////Event Table End

//Global Variables
FIBITMAP *InputImage; FIBITMAP *DemoImage; FIBITMAP *BadPixelImage;
BYTE *InputImageVector; BYTE *DemoImageVector; BYTE *BadPixelVector;

wxString TempStr; char TempChar[20];

FractalImageDlg::FractalImageDlg( wxWindow *parent, wxWindowID id,
const wxString &title, const wxPoint &position, const wxSize& size,
long style )
: wxDialog( parent, id, title, position, size, style)
{
    CreateGUIControls();
}

FractalImageDlg::~FractalImageDlg() { }

void FractalImageDlg::CreateGUIControls(void) {
    //Do not add custom Code here
    //wx-devcpp designer will remove them.
    //Add the custom code before or after the Blocks
    ////GUI Items Creation Start

    WxPanel1 = new wxPanel(this, ID_WXPANEL1, wxPoint(-3,-2),
        wxSize(522,414) );

    LoadImageButton = new wxButton(WxPanel1, ID_LOADIMAGEBUTTON,
        _("Load Image") , wxPoint(6,18),wxSize(89,28) );

    ProgressMemo = new wxTextCtrl(WxPanel1, ID_PROGRESSMEMO, "" ,
        wxPoint(272,13),wxSize(238,245) , wxTE_READONLY |
        wxTE_MULTILINE);
    ProgressMemo->SetMaxLength(0);
    ProgressMemo->AppendText(_("ProgressMemo"));

    InputFileMemo = new wxTextCtrl(WxPanel1, ID_INPUTFILEMEMO, "" ,

```

```

        wxPoint(99,21),wxSize(168,21) , wxTE_MULTILINE);
InputFileMemo->SetMaxLength(0);
InputFileMemo->AppendText(_("InputFileMemo"));

ParentSlider = new wxSlider(WxPanel1, ID_PARENTSLIDER,0,0,64,
        wxPoint(107,73),wxSize(158,42) , wxSL_AUTOTICKS |
        wxSL_LABELS | wxSL_HORIZONTAL);
ParentSlider->SetRange(0,64);
ParentSlider->SetValue(0);

WxStaticLine1 = new wxStaticLine(WxPanel1, ID_WXSTATICLINE1,
        wxPoint(6,61),wxSize(260,2) );

WxStaticText1 = new wxStaticText(WxPanel1, ID_WXSTATICTEXT1 ,
        _("Parent Size (pixels)") ,wxPoint(9,89),wxSize(93,17) );

WxStaticText2 = new wxStaticText(WxPanel1, ID_WXSTATICTEXT2 ,
        _("Child Size (pixels)") ,wxPoint(9,136),wxSize(85,17) );

ChildSlider = new wxSlider(WxPanel1, ID_CHILDSLIDER,4,4,64,
        wxPoint(107,120),wxSize(158,42), wxSL_AUTOTICKS
        | wxSL_LABELS | wxSL_HORIZONTAL);
ChildSlider->SetRange(4,64);
ChildSlider->SetValue(4);

ParentShiftSlider = new wxSlider(WxPanel1, ID_PARENTSHIFTSLIDER,1,
        1,64,wxPoint(107,170),wxSize(158,42) , wxSL_AUTOTICKS |
        wxSL_LABELS | wxSL_HORIZONTAL);
ParentShiftSlider->SetRange(1,64);
ParentShiftSlider->SetValue(1);

WxStaticText3 = new wxStaticText(WxPanel1, ID_WXSTATICTEXT3 ,
        _("Parent Shift (pixels)") ,wxPoint(9,185),wxSize(94,17) );

ChildOverlapSlider = new wxSlider(WxPanel1, ID_CHILDOVERLAPSLIDER,
        0,0,63,wxPoint(108,220),wxSize(158,42) , wxSL_AUTOTICKS |
        wxSL_LABELS | wxSL_HORIZONTAL);
ChildOverlapSlider->SetRange(0,63);
ChildOverlapSlider->SetValue(0);

WxStaticText4 = new wxStaticText(WxPanel1, ID_WXSTATICTEXT4 ,
        _("Child Overlap (pixels)") ,wxPoint(9,235),wxSize(102,17));

```

```

EncodeButton = new wxButton(WxPanel1, ID_ENCODEBUTTON,
    _("E N C O D E") , wxPoint(22,351),wxSize(225,42) );

WxStaticLine2 = new wxStaticLine(WxPanel1, ID_WXSTATICLINE2,
    wxPoint(7,270),wxSize(256,2) );

AveragingSlider = new wxSlider(WxPanel1, ID_AVERAGINGSLIDER,
    1,1,25,wxPoint(108,299),wxSize(158,37) , wxSL_AUTOTICKS |
    wxSL_LABELS | wxSL_HORIZONTAL);
AveragingSlider->SetRange(1,25);
AveragingSlider->SetValue(1);

WxStaticText6 = new wxStaticText(WxPanel1, ID_WXSTATICTEXT6 ,
    _("# of Avgd Parents") ,wxPoint(12,306),wxSize(96,17) );

WxStaticText5 = new wxStaticText(WxPanel1, ID_WXSTATICTEXT5 ,
    _("Show Parent-Child Match?") ,wxPoint(282,302),
    wxSize(130,17));

ParentChildMatchCheckbox = new wxCheckBox(WxPanel1,
    ID_PARENTCHILDMATCHCHECKBOX, _("ParentChildMatchCheckbox") ,
    wxPoint(477,299),wxSize(17,18) );

WhichChildTxt = new wxStaticText(WxPanel1, ID_WHICHCHILDTXT ,
    _("Which Child?") ,wxPoint(284,326),wxSize(67,17) );

ChildMemo = new wxTextCtrl(WxPanel1, ID_CHILDMEMO, _("(1") ,
    wxPoint(450,322),wxSize(41,21) );
ChildMemo->SetMaxLength(0);
ChildMemo->Enable(false);

InputFileDialog = new wxFileDialog(this, "Choose a file" , "" ,
    "" , "*. *" , wxOPEN);

WxStaticText7 = new wxStaticText(WxPanel1, ID_WXSTATICTEXT7 ,
    _("Bad Pixels?") ,wxPoint(284,352),wxSize(59,17) );

BadPixelCheckBox = new wxCheckBox(WxPanel1, ID_BADPIXELCHECKBOX,
    _("BadPixelCheckBox") , wxPoint(476,350),wxSize(14,17) );

WxStaticText8 = new wxStaticText(WxPanel1, ID_WXSTATICTEXT8 ,

```

```

        _("How Many?") ,wxPoint(284,375),wxSize(61,17) );

BadPixelMemo = new wxTextCtrl(WxPanel1, ID_BADPIXELMEMO, _("1") ,
    wxPoint(451,373),wxSize(39,21) );
BadPixelMemo->SetMaxLength(0);
BadPixelMemo->Enable(false);

this->SetSize(8,8,537,457);
this->SetTitle(_("FractalImage"));
this->Center();
this->SetIcon(wxNullIcon);

/////GUI Items Creation End
ParentSlider->SetValue(64);
ChildSlider->SetValue(32);
ParentShiftSlider->SetValue(64);
ChildOverlapSlider->SetValue(0);
}

void FractalImageDlg::FractalImageDlgClose(wxCloseEvent& event) {
    // --> Don't use Close with a wxDialog,
    // use Destroy instead.
    Destroy();
}

/*
 * LoadImageButtonClick
 */
void FractalImageDlg::LoadImageButtonClick(wxCommandEvent& event) {
    unsigned width,height;           // dimensions of input image
    wxString widthStr,heightStr;     // string versions, for
        troubleshooting purposes
    wxString string1, string2;       // more strings
    InputFileDialog->ShowModal(); //selects file to load
    string1=InputFileDialog->GetDirectory();
    string2=InputFileDialog->GetFilename();
    InputFileMemo->SetValue(string1+"\\ "+string2);

    InputImage = FreeImage_Load(FIF_BMP, InputFileMemo->GetValue(),
        BMP_DEFAULT);
    DemoImage = FreeImage_Load(FIF_BMP, InputFileMemo->GetValue(),
        BMP_DEFAULT);

```

```

BadPixelImage = FreeImage_Load(FIF_BMP, InputFileMemo->GetValue(),
    BMP_DEFAULT);

if (InputImage) { // bitmap successfully loaded!
    ProgressMemo->AppendText("Successfully loaded!\n\n");
    if (FreeImage_GetWidth(InputImage) == 256){
        if (FreeImage_GetHeight(InputImage) == 256){;
            width=FreeImage_GetWidth(InputImage);
            height=FreeImage_GetHeight(InputImage);
            sprintf(TempChar,"%d",width);
            widthStr=TempChar;
            sprintf(TempChar,"%d",height);
            heightStr=TempChar;
            ProgressMemo->AppendText("Dimensions: "+widthStr+" x "
                +heightStr+"\n\n");
            //These lines dump image data into vectors so we can
                work with them.
            InputImageVector = (BYTE *) FreeImage_GetBits(InputImage);
            DemoImageVector = (BYTE *) FreeImage_GetBits(DemoImage);
            BadPixelVector = (BYTE *) FreeImage_GetBits(BadPixelImage);
        }
        else ProgressMemo->AppendText("Currently, this program only
            supports 256x256 images.\n\n");
    }
    else ProgressMemo->AppendText("Currently, this program only
        supports 256x256 images.\n\n");
    }
    else ProgressMemo->AppendText("File not located.\n\n");
    event.Skip();
} /*
 * ParentSliderScroll
 */
void FractalImageDlg::ParentSliderScroll(wxScrollEvent& event) {
    //Changes childslider so that its maximum value remains less
        than the parent size. This is important, since children must
        be smaller in size than parents for the contractivity to come
        into play.
    ChildSlider->SetRange(4, ParentSlider->GetValue());
    if (ChildSlider->GetValue() > ParentSlider->GetMax()){
        ChildSlider->SetValue(ParentSlider->GetMax());
    }
    //Changes parentshiftslider so that it is impossible to create

```

any gaps when considering parents- i.e. the whole picture is covered. This is perhaps unnecessary, but it's here to simplify matters.

```

ParentShiftSlider->SetRange(1, ParentSlider->GetValue());
if (ParentShiftSlider->GetValue() > ParentSlider->GetMax()){
    ParentShiftSlider->SetValue(ParentSlider->GetMax());
}
// insert your code here
event.Skip();
} /*
 * ChildSliderScroll
 */
void FractalImageDlg::ChildSliderScroll(wxScrollEvent& event) {
    //Ensures that the entire picture is covered by children, while
    simultaneously disallowing children to overlap by more than
    their own width, which would be a nonsensical notion.
    ChildOverlapSlider->SetRange(0, ChildSlider->GetValue()-1);
    if (ChildOverlapSlider->GetValue() > ChildSlider->GetMax()-1){
        ChildOverlapSlider->SetValue(ChildSlider->GetMax());
    }
    // insert your code here
    event.Skip();
} /*
 * EncodeButtonClick
 */
void FractalImageDlg::EncodeButtonClick(wxCommandEvent& event) {
    //variable declarations (lots of them)
    unsigned ParentWidth, ParentHeight, ChildWidth, ChildHeight;
    unsigned XParentShift, YParentShift, XOverlap, YOverlap;
    unsigned NumOfParentsAcross, NumOfParentsDown;
    unsigned NumOfChildrenAcross, NumOfChildrenDown;
    unsigned NumOfOverlaps;
    unsigned NumParents, NumChildren;
    unsigned WidthScaling, HeightScaling;
    unsigned x, y, i, j, m, n, q, p, r, CurrentChild, CurrentParent;
    unsigned ParentFirstPixel[4096];
    unsigned ChildFirstPixel[4096];
    unsigned LoopStart, LoopEnd;
    unsigned ParentL1Norm[4096], ParentL2Norm[4096], ChildL1Norm;
    unsigned BestParent[4096][30];
    unsigned ParentOrientation[4096][30];
    unsigned NumberToAverage;

```



```

unsigned width, height;
unsigned FirstRand, SecondRand;
unsigned BoxedChild;
unsigned BadPixel[20];
unsigned FirstBadPixel;
unsigned InnerProductDivisor, InnerProductDivisorR,
    InnerProductDivisorU, InnerProductDivisorUR;
unsigned InnerProductDivisor90, InnerProductDivisor90R,
    InnerProductDivisor270, InnerProductDivisor270R;
bool skip = false;
bool YepBadPixel = false;
float InnerProductR, InnerProductU, InnerProductUR,
    InnerProduct90, InnerProduct90R;
float InnerProduct270, InnerProduct270R;
float ParentL1NormF, ParentL2NormF, ChildL1NormF, InnerProductF;
float InnerProduct, InnerProductRF, InnerProductUF,
    InnerProductURF;
float InnerProduct270F, InnerProduct270RF, InnerProduct90F,
    InnerProduct90RF;
float InnerProductAdd;
float Alpha, Beta, AlphaR, BetaR, Collage, CollageR, AlphaU,
    BetaU, CollageU;
float AlphaUR, BetaUR, CollageUR, Alpha90, Beta90, Collage90,
    Alpha90R, Beta90R, Collage90R;
float Alpha270, Beta270, Collage270, Alpha270R, Beta270R,
    Collage270R;
float BestAlpha[4096][30], BestBeta[4096][30], BestCollage[30];
float TempFloat;
wxString ParentWidthStr, ParentHeightStr, ChildWidthStr,
    ChildHeightStr;
wxString XParentShiftStr, YParentShiftStr, XOverlapStr,
    YOverlapStr;
wxString NumOfParentsAcrossStr, NumOfParentsDownStr;
wxString NumOfChildrenAcrossStr, NumOfChildrenDownStr;
wxString NumParentsStr;
wxString ParentOrientationStr, CurrentChildStr;
wxString TempStr;
wxString OutputFilename;
wxString widthStr, heightStr;
FILE *OutputFile;
const unsigned LOW = 1;
const unsigned HIGH = 60;

```

```

//assigning values to variables based on input and hard-coding.
i = 0;
j = 0;
q = 0;
p = 0;
r = 0;
BoxedChild = 1;
width=FreeImage_GetWidth(InputImage);
height=FreeImage_GetHeight(InputImage);
sprintf(TempChar,"%d",width);
widthStr=TempChar;
sprintf(TempChar,"%d",height);
heightStr=TempChar;
ParentWidth = ParentSlider->GetValue();
ParentHeight = ParentWidth;
ChildWidth = ChildSlider->GetValue();
ChildHeight = ChildWidth;
WidthScaling = int(ParentWidth/ChildWidth);
HeightScaling = int(ParentHeight/ChildHeight);
XParentShift = ParentShiftSlider->GetValue();
YParentShift = XParentShift;
XOverlap = ChildOverlapSlider->GetValue();
YOverlap = XOverlap;
InnerProductDivisor = ChildWidth*ChildHeight;
InnerProductDivisorR = ChildWidth*ChildHeight;
InnerProductDivisorU = ChildWidth*ChildHeight;
InnerProductDivisorUR = ChildWidth*ChildHeight;
InnerProductDivisor90 = ChildWidth*ChildHeight;
InnerProductDivisor90R = ChildWidth*ChildHeight;
InnerProductDivisor270 = ChildWidth*ChildHeight;
InnerProductDivisor270R = ChildWidth*ChildHeight;

sprintf(TempChar,"%d",ParentWidth);
ParentWidthStr = TempChar;
sprintf(TempChar,"%d",ParentHeight);
ParentHeightStr = TempChar;
sprintf(TempChar,"%d",ChildWidth);
ChildWidthStr = TempChar;
sprintf(TempChar,"%d",ChildHeight);
ChildHeightStr = TempChar;
sprintf(TempChar,"%d",XParentShift);
XParentShiftStr = TempChar;

```

```

sprintf(TempChar,"%d",YParentShift);
YParentShiftStr = TempChar;
sprintf(TempChar,"%d",XOverlap);
XOverlapStr = TempChar;
sprintf(TempChar,"%d",YOverlap);
YOverlapStr = TempChar;
ProgressMemo->AppendText("Parent Dimensions:
    "+ParentWidthStr+" x "+ParentHeightStr+"; ");
ProgressMemo->AppendText("Parent (X,Y) Shift:
    ("+XParentShiftStr+", "+YParentShiftStr+")\n");
ProgressMemo->AppendText("Child Dimensions:
    "+ChildWidthStr+" x "+ChildHeightStr+"; ");
ProgressMemo->AppendText("Child (X,Y) Overlap:
    ("+XOverlapStr+", "+YOverlapStr+")\n");

if (ParentWidth>=XParentShift) {
    NumOfParentsAcross = int((width+XParentShift-
        ParentWidth)/XParentShift);
}
else NumOfParentsAcross = int(width/XParentShift);

if (ParentHeight>=YParentShift) {
    NumOfParentsDown = int((height+YParentShift-
        ParentHeight)/YParentShift);
}
else NumOfParentsDown = int(height/YParentShift);

//NumOfOverlaps is an integer ONLY when the picture can be
    tiled perfectly by the children with the specified size and
    overlap! This could be useful in the future, when we're
    trying to decide when to modify things or make up
    'remainder' children, possibly considering child overlaps.
NumOfOverlaps = int((width - ChildWidth)/
    (ChildWidth - XOverlap));
NumOfChildrenAcross = int((width +
    XOverlap*NumOfOverlaps)/ChildWidth );
NumOfOverlaps = int((height - ChildHeight)/
    (ChildHeight - YOverlap));
NumOfChildrenDown = int((height +
    YOverlap*NumOfOverlaps)/ChildHeight);
NumParents = NumOfParentsAcross * NumOfParentsDown;
NumChildren = NumOfChildrenAcross * NumOfChildrenDown;

```

```

sprintf(TempChar, "%d", NumOfParentsAcross);
NumOfParentsAcrossStr = TempChar;
sprintf(TempChar, "%d", NumOfParentsDown);
NumOfParentsDownStr = TempChar;
sprintf(TempChar, "%d", NumOfChildrenAcross);
NumOfChildrenAcrossStr = TempChar;
sprintf(TempChar, "%d", NumOfChildrenDown);
NumOfChildrenDownStr = TempChar;
sprintf(TempChar, "%d", NumParents);
NumParentsStr = TempChar;
ProgressMemo->AppendText("There are "+NumOfParentsAcrossStr+"
    parents across the image.\n");
ProgressMemo->AppendText("There are "+NumOfParentsDownStr+"
    parents down the image.\n");
ProgressMemo->AppendText("There are "+NumOfChildrenAcrossStr+"
    children across the image.\n");
ProgressMemo->AppendText("There are "+NumOfChildrenDownStr+"
    children down the image.\n");
ProgressMemo->AppendText("There are "+NumParentsStr+" parents
    total.\n");

NumberToAverage = AveragingSlider->GetValue();

if (BadPixelCheckBox->IsChecked() == true) {

    for (i=1; i<=16; i++) {
        BadPixel[i] = 1;
    }
    time_t seconds;
    //Get value from system clock and
    //place in seconds variable.
    time(&seconds);
    //Convert seconds to a unsigned integer.
    srand((unsigned int) seconds);
    //Get random numbers between 1 and 61200 very crudely, ensuring
    that they are at least four away from the edge of the
    image. Due to the nature of the crudeness, the first bad
    pixels will more likely be in the middle of the range
    [1, 61200] than on the outsides.
    for (i = 0; i < 1; i=i+0){
        FirstRand = rand() % (HIGH - LOW + 1) + LOW;
    }
}

```

```

SecondRand = rand() % (HIGH - LOW + 1) + LOW;
if ((FirstRand*SecondRand*17 % 256) < 252){
    i = 1;
}
}
//assigns a first "bad" pixel
BadPixel[1] = FirstRand * SecondRand * 17;
sprintf(TempChar,"%d",BadPixel[1]);
TempStr = TempChar;
ProgressMemo->AppendText("The first bad pixel is "+TempStr+".");
FirstBadPixel = BadPixel[1];
//assigns a 4x4 block of bad pixels based on the first bad pixel.
for (i=2; i<5; i++) {
    BadPixel[i] = FirstBadPixel + (i-1);
}
for (i=0; i<=3; i++) {
    BadPixel[i+5] = BadPixel[1] + width + i;
}
for (i=0; i<=3; i++) {
    BadPixel[i+9] = BadPixel[1] + 2*width + i;
}
for (i=0; i<=3; i++) {
    BadPixel[i+13] = BadPixel[1] + 3*width + i;
}
for (i=1; i<=16; i++){
    BadPixelVector[BadPixel[i]] = 255;
}
}
i=0;

for (y=0; y < NumOfParentsDown; y++) {
    for (x=0; x< NumOfParentsAcross; x++) {
        ParentFirstPixel[i]=x*XParentShift+y*YParentShift*width;
        //Calculates each Parent's first pixel (the one in the
        bottom-left corner)
        i++;
    }
}

i=0;
for (y=0; y<NumOfChildrenDown; y++) {
    for (x=0; x<NumOfChildrenAcross; x++) {

```

```

//Preliminary Easy-Way-Out
ChildFirstPixel[i]=x*ChildWidth + y*ChildWidth*width;
i++;
//End easy-way-out. The following might eventually be a way to
deal with different "conditions" on the borders in future
work...if the overlapping doesn't work out perfectly.
Right now, it is just skeletal.
if (y == 0) {
    if (x == 0) {
        //x, y both at min (bottom left corner)
    }
    else if (x == NumOfChildrenAcross - 1) {
        //x at max, y at min (bottom right corner)
    }
    else; // y at min (bottom edge)
}
else if (x == 0) {
    if (y == NumOfChildrenDown - 1) {
    }
    //x at min, y at max (top left corner)
    else; //x at min (left edge)
}
else if (y == NumOfChildrenDown - 1) {
    if (x == NumOfChildrenAcross - 1) {
        //x, y both at max (top right corner)
    }
    else; //y at max (top edge)
}
else if (x == NumOfChildrenAcross - 1) {
    //x at max (right edge)
}
else; //x,y not at max: we're somewhere in the middle.
//end giant if
}
}

//initializes the norm calculations, best coefficient values.
for (i=0; i<4096; i++) {
    ParentL1Norm[i] = 0;
    ParentL2Norm[i] = 0;
}
for (i=0; i<4096; i++) {

```

```

    for (j=0; j<=(NumberToAverage + 4); j++) {
        ParentOrientation[i][j] = 0;
        BestAlpha[i][j] = 0;
        BestBeta[i][j] = 0;
    }
}
//First Steps to Calculating Parent L1/L2 norms...
//Later on, we'll finish this off, but we'll have to change to
    floats.
for (i=0; i<NumParents; i++) {
    for (y=0; y<ParentHeight; y++) {
        LoopStart=ParentFirstPixel[i]+y*width;
        LoopEnd = LoopStart+ParentWidth;
        for (x=LoopStart; x<LoopEnd; x++) {
            ParentL1Norm[i]=ParentL1Norm[i]+InputImageVector[x];
            ParentL2Norm[i]=ParentL2Norm[i]+
                InputImageVector[x]*InputImageVector[x];
        }
    }
}
for (CurrentChild=0; CurrentChild < NumChildren; CurrentChild++){
    //Calculating L1 ChildNorm
    ChildL1Norm = 0;
    //initializes best collage to start at something really big.
        If a parent-child pairing in this loop is considered that
        produces a smaller collage distance, that distance becomes
        the new "Best Collage". BestCollage[i] refers to the
        i'th-best collage distance, all of which are recorded.
    for (i=1; i<=(NumberToAverage + 4); i++) {
        BestCollage[i] = 99999999;
    }
    BestCollage[0] = -1;
    for (y=0; y < ChildHeight; y++) {
        LoopStart=ChildFirstPixel[CurrentChild]+y*width;
        LoopEnd = LoopStart+ChildWidth;
        for (x=LoopStart; x<LoopEnd; x++) {
            ChildL1Norm=ChildL1Norm+InputImageVector[x];
        }
    }
    for (CurrentParent=0; CurrentParent<NumParents;
        CurrentParent++) {
        skip = false;

```

```

//Checking to make sure there are NO BAD PIXELS!!!
for (q = 0; q<ParentWidth; q++) {
  for (p = 0; p<ParentHeight; p++) {
    for (r = 1; r<16; r++) {
      if (BadPixel[r] == ParentFirstPixel[CurrentParent] +
          p*width + q) {
        skip = true;
      }
    }
  }
}
if (skip == true) {
  //do nothing! (disregard parent) if bad pixels are found in
  the current parent!!
}
else {
  InnerProduct = 0;
  InnerProductR = 0;
  InnerProductU = 0;
  InnerProductUR = 0;
  InnerProduct90 = 0;
  InnerProduct90R = 0;
  InnerProduct270 = 0;
  InnerProduct270R = 0;
  InnerProductAdd = 0;
  for (y=0; y<ChildHeight; y++) {
    LoopStart=ChildFirstPixel[CurrentChild]+y*width;
    LoopEnd = LoopStart+ChildWidth;
    for (x=LoopStart; x<LoopEnd; x++) {
      //Calculates Inner Product... goes through Child Pixels, and
      multiplies, by the associated "average" grey-level value
      of the pixels of the Parent, according to the averaging
      decimation scheme. Remember! Scaling represents the
      DECIMATION that occurs when we shrink from the parent to
      the child, and is equal to the parentwidth divided by the
      childwidth. Right now, of course, this only takes into
      account square parents and children. The other loops to
      follow are calculated differently according to the
      orientation being considered.
      for (n = 0; n < HeightScaling; n++) {
        for (m = 0; m < WidthScaling; m++) {
          for (p = 1; p < 16; p++) {

```



```

//checking for bad pixels...
if (((ParentFirstPixel[CurrentParent] +
    (x-LoopStart)*WidthScaling + y*HeightScaling*width +
    m + n*width) == BadPixel[p]) &&
    (BadPixelCheckBox->IsChecked() == true)){
    //The current pixel is labeled as "bad". We must
    decrement the InnerProduct Divisor by 1 pixel.
    InnerProductDivisor = InnerProductDivisor - 1;
    YepBadPixel = true;
}
}
if (YepBadPixel == false) {
    InnerProductAdd = InnerProductAdd +
        (InputImageVector[ParentFirstPixel[CurrentParent]
        + (x-LoopStart)*WidthScaling + y*HeightScaling*width
        + m + n*width]);
}
YepBadPixel = false;
}
}
if (((HeightScaling*WidthScaling) - (ChildWidth*ChildHeight
    - InnerProductDivisor)) <= 0) {
    InnerProductAdd = InnerProductAdd/1 ;
}
else InnerProductAdd = InnerProductAdd/
    ((HeightScaling*WidthScaling) - (ChildWidth*ChildHeight
    - InnerProductDivisor));
InnerProduct = InnerProduct + InnerProductAdd;
InnerProductAdd = 0;
}
for (x=LoopStart; x<LoopEnd; x++) {
    //the same procedure goes for this loop, which considers
    orientation that flips the parent block horizontally
    while being reduced.
    for (n = 0; n < HeightScaling; n++) {
        for (m = 0; m < WidthScaling; m++) {
            for (p = 1; p < 16; p++) {
                //checking for bad pixels...
                if (((ParentFirstPixel[CurrentParent] +
                    (LoopEnd-x-1)*WidthScaling + y*HeightScaling*width +
                    m + n*width) == BadPixel[p]) &&
                    (BadPixelCheckBox->IsChecked() == true)){

```

```

        //The current pixel is labeled as "bad". We must
        decrement the InnerProduct Divisor by 1 pixel.
        InnerProductDivisorR = InnerProductDivisorR - 1;
        YepBadPixel = true;
    }
}
if (YepBadPixel == false) {
    InnerProductAdd = InnerProductAdd +
        (InputImageVector[ParentFirstPixel[CurrentParent] +
        (LoopEnd-x-1)*WidthScaling + y*HeightScaling*width
        + m + n*width]);
}
YepBadPixel = false;
}
}
if (((HeightScaling*WidthScaling) - (ChildWidth*ChildHeight
    - InnerProductDivisorR)) <= 0) {
    InnerProductAdd = InnerProductAdd/1;
}
else InnerProductAdd = ((HeightScaling*WidthScaling) -
    (ChildWidth*ChildHeight - InnerProductDivisor));
InnerProductR = InnerProductR + InnerProductAdd;
InnerProductAdd = 0;
}
for (x=LoopStart; x<LoopEnd; x++) {
    //Loop for the Vertical Flip orientation.
    for (n = 0; n < HeightScaling; n++) {
        for (m = 0; m < WidthScaling; m++) {
            for (p = 1; p < 16; p++) {
                //checking for bad pixels...
                if (((ParentFirstPixel[CurrentParent] +
                    (ChildHeight-(y+1))*HeightScaling*width) +
                    (x-LoopStart)*WidthScaling + m + n*width) ==
                    BadPixel[p]) && (BadPixelCheckBox->IsChecked()
                    == true)) {
                    //The current pixel is labeled as "bad". We must
                    decrement the InnerProduct Divisor by 1 pixel.
                    InnerProductDivisorU = InnerProductDivisorU - 1;
                    YepBadPixel = true;
                }
            }
        }
    }
    if (YepBadPixel == false) {

```

```

        InnerProductAdd = InnerProductAdd +
            (InputImageVector[(ParentFirstPixel[CurrentParent]
                + (ChildHeight-(y+1))*HeightScaling*width) +
                (x-LoopStart)*WidthScaling + m + n*width]);
    }
    YepBadPixel = false;
}
}
if (((HeightScaling*WidthScaling) - (ChildWidth*ChildHeight -
    InnerProductDivisorU)) <= 0) {
    InnerProductAdd = InnerProductAdd/1;
}
else InnerProductAdd = InnerProductAdd/
    ((HeightScaling*WidthScaling) - (ChildWidth*ChildHeight
        - InnerProductDivisor));
InnerProductU = InnerProductU + InnerProductAdd;
InnerProductAdd = 0;
}
for (x=LoopStart; x<LoopEnd; x++) {
    //loop considering horizontal + vertical flip orientation
    for (n = 0; n < HeightScaling; n++) {
        for (m = 0; m < WidthScaling; m++) {
            for (p = 1; p < 16; p++) {
                //checking for bad pixels...
                if (((ParentFirstPixel[CurrentParent] - 1 +
                    (ChildHeight-(y+1))*HeightScaling*width) +
                    (LoopEnd-x)*WidthScaling + m + n*width) ==
                    BadPixel[p]) && (BadPixelCheckBox->IsChecked()
                        == true)){
                    //The current pixel is labeled as "bad". We must
                    decrement the InnerProduct Divisor by 1 pixel.
                    InnerProductDivisorUR = InnerProductDivisorUR - 1;
                    YepBadPixel = true;
                }
            }
        }
        if (YepBadPixel == false) {
            InnerProductAdd = InnerProductAdd +
                (InputImageVector[(ParentFirstPixel[CurrentParent]
                    - 1 + (ChildHeight-(y+1))*HeightScaling*width) +
                    (LoopEnd-x)*WidthScaling + m + n*width]);
        }
        YepBadPixel = false;
    }
}

```

```

    }
  }
  if (((HeightScaling*WidthScaling) - (ChildWidth*ChildHeight -
    InnerProductDivisorUR)) <= 0) {
    InnerProductAdd = InnerProductAdd/1;
  }
  else InnerProductAdd = InnerProductAdd/
    ((HeightScaling*WidthScaling) - (ChildWidth*ChildHeight
    - InnerProductDivisor));
  InnerProductUR = InnerProductUR + InnerProductAdd;
  InnerProductAdd = 0;
}
for (x=LoopStart; x<LoopEnd; x++) {
  //90-degree clockwise rotation orientation
  for (n = 0; n < HeightScaling; n++) {
    for (m = 0; m < WidthScaling; m++) {
      for (p = 1; p < 16; p++) {
        //checking for bad pixels...
        if (((ParentFirstPixel[CurrentParent] - 1 -
          y*WidthScaling + ParentHeight +
          (x-LoopStart)*HeightScaling*width + m + n*width)
          == BadPixel[p]) && (BadPixelCheckBox->IsChecked()
          == true)){
          //The current pixel is labeled as "bad". We must
            decrement the InnerProduct Divisor by 1 pixel.
          InnerProductDivisor90 = InnerProductDivisor90 - 1;
          YepBadPixel = true;
        }
      }
    }
    if (YepBadPixel == false) {
      InnerProductAdd = InnerProductAdd +
        (InputImageVector[ParentFirstPixel[CurrentParent]
        -1 - y*WidthScaling + ParentHeight +
        (x-LoopStart)*HeightScaling*width + m + n*width]);
    }
    YepBadPixel = false;
  }
}
if (((HeightScaling*WidthScaling) - (ChildWidth*ChildHeight
  - InnerProductDivisor90)) <= 0) {
  InnerProductAdd = InnerProductAdd/1;
}

```

```

else InnerProductAdd = InnerProductAdd/
    ((HeightScaling*WidthScaling) - (ChildWidth*ChildHeight
    - InnerProductDivisor));
InnerProduct90 = InnerProduct90 + InnerProductAdd;
InnerProductAdd = 0;
}
for (x=LoopStart; x<LoopEnd; x++) {
    //90-degree clockwise rotation + horizontal flip orientation
    for (n = 0; n < HeightScaling; n++) {
        for (m = 0; m < WidthScaling; m++) {
            for (p = 1; p < 16; p++) {
                //checking for bad pixels...
                if (((ParentFirstPixel[CurrentParent] - 1
                    - y*WidthScaling + ParentHeight +
                    (LoopEnd - x - 1)*HeightScaling*width + m + n*width)
                    == BadPixel[p]) && (BadPixelCheckBox->IsChecked()
                    == true)){
                    //The current pixel is labeled as "bad". We must
                    decrement the InnerProduct Divisor by 1 pixel.
                    InnerProductDivisor90R = InnerProductDivisor90R - 1;
                    YepBadPixel = true;
                }
            }
        }
        if (YepBadPixel == false) {
            InnerProductAdd = InnerProductAdd +
                (InputImageVector[ParentFirstPixel[CurrentParent]
                - 1 - y*WidthScaling + ParentHeight +
                (LoopEnd - x - 1)*HeightScaling*width + m +
                n*width]);
        }
        YepBadPixel = false;
    }
}
if (((HeightScaling*WidthScaling) - (ChildWidth*ChildHeight
    - InnerProductDivisor90R)) <= 0) {
    InnerProductAdd = InnerProductAdd/1;
}
else InnerProductAdd = InnerProductAdd/
    ((HeightScaling*WidthScaling) - (ChildWidth*ChildHeight
    - InnerProductDivisor));
InnerProduct90R = InnerProduct90R + InnerProductAdd;
InnerProductAdd = 0;

```

```

}
for (x=LoopStart; x<LoopEnd; x++) {
    //270 degree clockwise rotation orientation.
    for (n = 0; n < HeightScaling; n++) {
        for (m = 0; m < WidthScaling; m++) {
            for (p = 1; p < 16; p++) {
                //checking for bad pixels...
                if (((ParentFirstPixel[CurrentParent] - width
                    - y*WidthScaling + (LoopEnd - x)*HeightScaling*width
                    + m + n*width) == BadPixel[p]) &&
                    (BadPixelCheckBox->IsChecked() == true)){
                    //The current pixel is labeled as "bad". We must
                    decrement the InnerProduct Divisor by 1 pixel.
                    InnerProductDivisor270 = InnerProductDivisor270 - 1;
                    YepBadPixel = true;
                }
            }
        }
        if (YepBadPixel == false) {
            InnerProductAdd = InnerProductAdd +
                (InputImageVector[ParentFirstPixel[CurrentParent]
                    - width - y*WidthScaling +
                    (LoopEnd - x)*HeightScaling*width + m + n*width]);
        }
        YepBadPixel = false;
    }
}
if (((HeightScaling*WidthScaling) - (ChildWidth*ChildHeight -
    InnerProductDivisor270)) <= 0) {
    InnerProductAdd = InnerProductAdd/1;
}
else InnerProductAdd = InnerProductAdd/
    ((HeightScaling*WidthScaling) - (ChildWidth*ChildHeight
    - InnerProductDivisor));
InnerProduct270 = InnerProduct270 + InnerProductAdd;
InnerProductAdd = 0;
}
for (x=LoopStart; x<LoopEnd; x++) {
    //270 degree clockwise rotation + horizontal flip orientation.
    for (n = 0; n < HeightScaling; n++) {
        for (m = 0; m < WidthScaling; m++) {
            for (p = 1; p < 16; p++) {
                //checking for bad pixels...

```

```

    if (((ParentFirstPixel[CurrentParent] + y*WidthScaling
        + (x - LoopStart)*HeightScaling*width + m + n*width)
        == BadPixel[p]) && (BadPixelCheckBox->IsChecked()
        == true)){
        //The current pixel is labeled as "bad". We must
        decrement the InnerProduct Divisor by 1 pixel.
        InnerProductDivisor270R = InnerProductDivisor270R - 1;
        YepBadPixel = true;
    }
}
if (YepBadPixel == false) {
    InnerProductAdd = InnerProductAdd +
        (InputImageVector[x]*InputImageVector[ParentFirstPixel
        [CurrentParent] + y*WidthScaling +
        (x - LoopStart)*HeightScaling*width + m + n*width]);
}
YepBadPixel = false;
}
}
if (((HeightScaling*WidthScaling) - (ChildWidth*ChildHeight
    - InnerProductDivisor270R)) <= 0) {
    InnerProductAdd = InnerProductAdd/1;
}
else InnerProductAdd = InnerProductAdd/
    ((HeightScaling*WidthScaling)
    - (ChildWidth*ChildHeight - InnerProductDivisor));
InnerProduct270R = InnerProduct270R + InnerProductAdd;
InnerProductAdd = 0;
}
}
//changing unsigned integers into floats, to avoid
type-related problems.
ParentL1NormF = ParentL1Norm[CurrentParent]/
    (ParentWidth*ParentHeight);
ParentL2NormF = ParentL2Norm[CurrentParent]/
    (ParentWidth*ParentHeight);
//Note- ParentL2Norm is actually the L2-norm, squared,
for simplicity.
ChildL1NormF = ChildL1Norm/InnerProductDivisor;
InnerProductF = InnerProduct/InnerProductDivisor;
InnerProductRF = InnerProductR/InnerProductDivisorR;
InnerProductUF = InnerProductU/InnerProductDivisorU;

```

```

InnerProductURF = InnerProductUR/InnerProductDivisorUR;
InnerProduct90F = InnerProduct90/InnerProductDivisor90;
InnerProduct90RF = InnerProduct90R/InnerProductDivisor90R;
InnerProduct270F = InnerProduct270/InnerProductDivisor270;
InnerProduct270RF = InnerProduct270R/InnerProductDivisor270R;
//Resetting the divisor in case of bad pixels;
InnerProductDivisor = ChildWidth*ChildHeight;
InnerProductDivisorR = ChildWidth*ChildHeight;
InnerProductDivisorU = ChildWidth*ChildHeight;
InnerProductDivisorUR = ChildWidth*ChildHeight;
InnerProductDivisor90 = ChildWidth*ChildHeight;
InnerProductDivisor90R = ChildWidth*ChildHeight;
InnerProductDivisor270 = ChildWidth*ChildHeight;
InnerProductDivisor270R = ChildWidth*ChildHeight;
//Calculating the alpha/beta using Kramer's Rule...
Alpha = ((InnerProductF * 1) - (ChildL1NormF * ParentL1NormF))/
        ((ParentL2NormF * 1) - (ParentL1NormF * ParentL1NormF));
Beta = ((ParentL2NormF * ChildL1NormF) -
        (InnerProductF*ParentL1NormF))/
        ((ParentL2NormF * 1) - (ParentL1NormF * ParentL1NormF));
//Doing the same for the REVERSE and other orientations...
AlphaR = ((InnerProductRF * 1) - (ChildL1NormF *
        ParentL1NormF))/((ParentL2NormF * 1) -
        (ParentL1NormF * ParentL1NormF));
BetaR = ((ParentL2NormF * ChildL1NormF) -
        (InnerProductRF*ParentL1NormF))/((ParentL2NormF * 1)
        - (ParentL1NormF * ParentL1NormF));
AlphaU = ((InnerProductUF * 1) -
        (ChildL1NormF * ParentL1NormF))/((ParentL2NormF * 1)
        - (ParentL1NormF * ParentL1NormF));
BetaU = ((ParentL2NormF * ChildL1NormF)
        - (InnerProductUF*ParentL1NormF))/((ParentL2NormF * 1)
        - (ParentL1NormF * ParentL1NormF));
AlphaUR = ((InnerProductURF * 1) -
        (ChildL1NormF * ParentL1NormF))/((ParentL2NormF * 1)
        - (ParentL1NormF * ParentL1NormF));
BetaUR = ((ParentL2NormF * ChildL1NormF)
        - (InnerProductURF*ParentL1NormF))/((ParentL2NormF * 1)
        - (ParentL1NormF * ParentL1NormF));
Alpha90 = ((InnerProduct90F * 1)
        - (ChildL1NormF * ParentL1NormF))/((ParentL2NormF * 1)
        - (ParentL1NormF * ParentL1NormF));

```



```

Beta90 = ((ParentL2NormF * ChildL1NormF)
  - (InnerProduct90F*ParentL1NormF))/((ParentL2NormF * 1)
  - (ParentL1NormF * ParentL1NormF));
Alpha90R = ((InnerProduct90RF * 1)
  - (ChildL1NormF * ParentL1NormF))/((ParentL2NormF * 1)
  - (ParentL1NormF * ParentL1NormF));
Beta90R = ((ParentL2NormF * ChildL1NormF)
  - (InnerProduct90RF*ParentL1NormF))/((ParentL2NormF * 1)
  - (ParentL1NormF * ParentL1NormF));
Alpha270 = ((InnerProduct270F * 1)
  - (ChildL1NormF * ParentL1NormF))/((ParentL2NormF * 1)
  - (ParentL1NormF * ParentL1NormF));
Beta270 = ((ParentL2NormF * ChildL1NormF)
  - (InnerProduct270F*ParentL1NormF))/((ParentL2NormF * 1)
  - (ParentL1NormF * ParentL1NormF));
Alpha270R = ((InnerProduct270RF * 1)
  - (ChildL1NormF * ParentL1NormF))/((ParentL2NormF * 1)
  - (ParentL1NormF * ParentL1NormF));
Beta270R = ((ParentL2NormF * ChildL1NormF)
  - (InnerProduct270RF*ParentL1NormF))/((ParentL2NormF * 1)
  - (ParentL1NormF * ParentL1NormF));
//initializing all of the collage values for each
  possible orientation.
Collage = 0;
CollageR = 0;
CollageU = 0;
CollageUR = 0;
Collage90 = 0;
Collage90R = 0;
Collage270 = 0;
Collage270R = 0;
for (y=0; y<ChildHeight; y++) {
  LoopStart=ChildFirstPixel[CurrentChild]+y*width;
  LoopEnd = LoopStart+ChildWidth;
  for (x=LoopStart; x<LoopEnd; x++) {
    TempFloat = (InputImageVector[x] -
      (Alpha*InputImageVector[ParentFirstPixel[CurrentParent]
      + (x-LoopStart)*WidthScaling + y*HeightScaling*width]
      +Beta));
    Collage = Collage + TempFloat*TempFloat;
  }
  //Not ACTUALLY the collage distance... this is still
    squared, however, the minimal value of THIS will

```

```

        certainly correspond to the minimal collage distance,
        so we leave it.
    }
    for (x=LoopStart; x<LoopEnd; x++) {
        TempFloat = (InputImageVector[x] -
            (AlphaR*InputImageVector[ParentFirstPixel[CurrentParent]
            + (x-LoopStart)*WidthScaling + y*HeightScaling*width]
            +BetaR));
        CollageR = CollageR + TempFloat*TempFloat;
        //Now, we do the same work with the second set, that of
        the reverse tallies.
    }
    for (x=LoopStart; x<LoopEnd; x++) {
        TempFloat = (InputImageVector[x] -
            (AlphaU*InputImageVector[ParentFirstPixel[CurrentParent]
            + (x-LoopStart)*WidthScaling + y*HeightScaling*width]
            +BetaU));
        CollageU = CollageU + TempFloat*TempFloat;
    }
    for (x=LoopStart; x<LoopEnd; x++) {
        TempFloat = (InputImageVector[x]
            - (AlphaUR*InputImageVector[ParentFirstPixel
            [CurrentParent] + (x-LoopStart)*WidthScaling
            + y*HeightScaling*width]+BetaUR));
        CollageUR = CollageUR + TempFloat*TempFloat;
    }
    for (x=LoopStart; x<LoopEnd; x++) {
        TempFloat = (InputImageVector[x]
            - (Alpha90*InputImageVector[ParentFirstPixel
            [CurrentParent] + (x-LoopStart)*WidthScaling
            + y*HeightScaling*width]+Beta90));
        Collage90 = Collage90 + TempFloat*TempFloat;
    }
    for (x=LoopStart; x<LoopEnd; x++) {
        TempFloat = (InputImageVector[x]
            - (Alpha90R*InputImageVector[ParentFirstPixel
            [CurrentParent] + (x-LoopStart)*WidthScaling
            + y*HeightScaling*width]+Beta90R));
        Collage90R = Collage90R + TempFloat*TempFloat;
    }
    for (x=LoopStart; x<LoopEnd; x++) {
        TempFloat = (InputImageVector[x]

```

```

        - (Alpha270*InputImageVector[ParentFirstPixel
          [CurrentParent] + (x-LoopStart)*WidthScaling
          + y*HeightScaling*width] +Beta270));
    Collage270 = Collage270 + TempFloat*TempFloat;
}
for (x=LoopStart; x<LoopEnd; x++) {
    TempFloat = (InputImageVector[x]
        - (Alpha270R*InputImageVector[ParentFirstPixel
          [CurrentParent] + (x-LoopStart)*WidthScaling
          + y*HeightScaling*width]+Beta270R));
    Collage270R = Collage270R + TempFloat*TempFloat;
}
//One pass through this calculates the collage distance
//for one single parent, for the current child. Next pass
//will do the next parent, etc, etc, until all parents have
//been considered for the current child! We could STOP it
//and take a "reasonably good" parent that produces a
//collage distance less than some threshold value, if we
//wanted to speed things up slightly.
}
for (j=NumberToAverage; j>=1; j=j-1) {
    if ((Collage < BestCollage[j]) &&
        (Collage >= BestCollage[j-1])) {
        //This collage distance is one of the top best! We must
        //shuffle the ranks.
        for (i=NumberToAverage; i>=j; i=i-1){
            BestCollage[i+1] = BestCollage[i];
            BestAlpha[CurrentChild][i+1] = BestAlpha[CurrentChild][i];
            BestBeta[CurrentChild][i+1] = BestBeta[CurrentChild][i];
            BestParent[CurrentChild][i+1] = BestParent[CurrentChild][i];
            ParentOrientation[CurrentChild][i+1] =
                ParentOrientation[CurrentChild][i];
        }
        BestCollage[j] = Collage;
        BestAlpha[CurrentChild][j] = Alpha;
        BestBeta[CurrentChild][j] = Beta;
        BestParent[CurrentChild][j] = CurrentParent;
        ParentOrientation[CurrentChild][j] = 1;
    }
}
for (j=NumberToAverage; j>=1; j=j-1) {
    if ((CollageR < BestCollage[j]) &&

```

```

        (CollageR >= BestCollage[j-1])) {
//The same is repeated for the reverse and other orientations
    in the loops below.
    for (i=NumberToAverage; i>=j; i=i-1){
        BestCollage[i+1] = BestCollage[i];
        BestAlpha[CurrentChild][i+1] = BestAlpha[CurrentChild][i];
        BestBeta[CurrentChild][i+1] = BestBeta[CurrentChild][i];
        BestParent[CurrentChild][i+1] = BestParent[CurrentChild][i];
        ParentOrientation[CurrentChild][i+1]
            = ParentOrientation[CurrentChild][i];
    }
    BestCollage[j] = CollageR;
    BestAlpha[CurrentChild][j] = AlphaR;
    BestBeta[CurrentChild][j] = BetaR;
    BestParent[CurrentChild][j] = CurrentParent;
    ParentOrientation[CurrentChild][j] = 2;
}
}
for (j=NumberToAverage; j>=1; j=j-1) {
    if ((CollageU < BestCollage[j]) &&
        (CollageU >= BestCollage[j-1])) {
        for (i=NumberToAverage; i>=j; i=i-1){
            BestCollage[i+1] = BestCollage[i];
            BestAlpha[CurrentChild][i+1] = BestAlpha[CurrentChild][i];
            BestBeta[CurrentChild][i+1] = BestBeta[CurrentChild][i];
            BestParent[CurrentChild][i+1] = BestParent[CurrentChild][i];
            ParentOrientation[CurrentChild][i+1]
                = ParentOrientation[CurrentChild][i];
        }
        BestCollage[j] = CollageU;
        BestAlpha[CurrentChild][j] = AlphaU;
        BestBeta[CurrentChild][j] = BetaU;
        BestParent[CurrentChild][j] = CurrentParent;
        ParentOrientation[CurrentChild][j] = 3;
    }
}
for (j=NumberToAverage; j>=1; j=j-1) {
    if ((CollageUR < BestCollage[j]) &&
        (CollageUR >= BestCollage[j-1])) {
        for (i=NumberToAverage; i>=j; i=i-1){
            BestCollage[i+1] = BestCollage[i];
            BestAlpha[CurrentChild][i+1] = BestAlpha[CurrentChild][i];

```

```

    BestBeta[CurrentChild][i+1] = BestBeta[CurrentChild][i];
    BestParent[CurrentChild][i+1] = BestParent[CurrentChild][i];
    ParentOrientation[CurrentChild][i+1]
        = ParentOrientation[CurrentChild][i];
}
BestCollage[j] = CollageUR;
BestAlpha[CurrentChild][j] = AlphaUR;
BestBeta[CurrentChild][j] = BetaUR;
BestParent[CurrentChild][j] = CurrentParent;
ParentOrientation[CurrentChild][j] = 4;
}
}
for (j=NumberToAverage; j>=1; j=j-1) {
    if ((Collage90 < BestCollage[j]) &&
        (Collage90 >= BestCollage[j-1])) {
        for (i=NumberToAverage; i>=j; i=i-1){
            BestCollage[i+1] = BestCollage[i];
            BestAlpha[CurrentChild][i+1] = BestAlpha[CurrentChild][i];
            BestBeta[CurrentChild][i+1] = BestBeta[CurrentChild][i];
            BestParent[CurrentChild][i+1] = BestParent[CurrentChild][i];
            ParentOrientation[CurrentChild][i+1]
                = ParentOrientation[CurrentChild][i];
        }
        BestCollage[j] = Collage90;
        BestAlpha[CurrentChild][j] = Alpha90;
        BestBeta[CurrentChild][j] = Beta90;
        BestParent[CurrentChild][j] = CurrentParent;
        ParentOrientation[CurrentChild][j] = 5;
    }
}
for (j=NumberToAverage; j>=1; j=j-1) {
    if ((Collage90R < BestCollage[j]) &&
        (Collage90R >= BestCollage[j-1])) {
        for (i=NumberToAverage; i>=j; i=i-1){
            BestCollage[i+1] = BestCollage[i];
            BestAlpha[CurrentChild][i+1] = BestAlpha[CurrentChild][i];
            BestBeta[CurrentChild][i+1] = BestBeta[CurrentChild][i];
            BestParent[CurrentChild][i+1] = BestParent[CurrentChild][i];
            ParentOrientation[CurrentChild][i+1]
                = ParentOrientation[CurrentChild][i];
        }
        BestCollage[j] = Collage90R;
    }
}

```

```

    BestAlpha[CurrentChild][j] = Alpha90R;
    BestBeta[CurrentChild][j] = Beta90R;
    BestParent[CurrentChild][j] = CurrentParent;
    ParentOrientation[CurrentChild][j] = 6;
}
}
for (j=NumberToAverage; j>=1; j=j-1) {
    if ((Collage270 < BestCollage[j]) &&
        (Collage270 >= BestCollage[j-1])) {
        for (i=NumberToAverage; i>=j; i=i-1){
            BestCollage[i+1] = BestCollage[i];
            BestAlpha[CurrentChild][i+1] = BestAlpha[CurrentChild][i];
            BestBeta[CurrentChild][i+1] = BestBeta[CurrentChild][i];
            BestParent[CurrentChild][i+1] = BestParent[CurrentChild][i];
            ParentOrientation[CurrentChild][i+1]
                = ParentOrientation[CurrentChild][i];
        }
        BestCollage[j] = Collage270;
        BestAlpha[CurrentChild][j] = Alpha270;
        BestBeta[CurrentChild][j] = Beta270;
        BestParent[CurrentChild][j] = CurrentParent;
        ParentOrientation[CurrentChild][j] = 7;
    }
}
for (j=NumberToAverage; j>=1; j=j-1) {
    if ((Collage270R < BestCollage[j]) &&
        (Collage270R >= BestCollage[j-1])) {
        for (i=NumberToAverage; i>=j; i=i-1){
            BestCollage[i+1] = BestCollage[i];
            BestAlpha[CurrentChild][i+1] = BestAlpha[CurrentChild][i];
            BestBeta[CurrentChild][i+1] = BestBeta[CurrentChild][i];
            BestParent[CurrentChild][i+1] = BestParent[CurrentChild][i];
            ParentOrientation[CurrentChild][i+1]
                = ParentOrientation[CurrentChild][i];
        }
        BestCollage[j] = Collage270R;
        BestAlpha[CurrentChild][j] = Alpha270R;
        BestBeta[CurrentChild][j] = Beta270R;
        BestParent[CurrentChild][j] = CurrentParent;
        ParentOrientation[CurrentChild][j] = 8;
    }
}
}

```

```

    }
    //if the parent is ignored due to bad pixels,
    the program skips here.
}
}
if (ParentChildMatchCheckbox->IsChecked() == true) {
    //Drawing boxes for display purposes!
    BoxedChild = atoi(ChildMemo->GetValue());
    if (BoxedChild > NumChildren) {
        //the child selected doesn't exist, and nothing is done.
    }
    else {
        //set pixels around the selected child to white.
        for (i=0; i<ChildWidth; i++) {
            DemoImageVector[ChildFirstPixel[BoxedChild]+i] = 255;
        }
        for (i=0; i<ChildWidth; i++) {
            DemoImageVector[ChildFirstPixel[BoxedChild]+i*width] = 255;
            DemoImageVector[ChildFirstPixel[BoxedChild]+i*width
                +(ChildWidth-1)] = 255;
        }
        for (i=0; i<ChildWidth; i++) {
            DemoImageVector[ChildFirstPixel[BoxedChild]
                +(ChildWidth-1)*width+i] = 255;
        }
    }
}
for (i=1; i<=NumberToAverage; i++) {
    //set pixels around the best parents to black.
    for (j=0; j<ParentWidth; j++) {
        DemoImageVector[ParentFirstPixel[BestParent[BoxedChild][i]]
            + j] = 0;
    }
    for (j=0; j<ParentWidth; j++) {
        DemoImageVector[ParentFirstPixel[BestParent[BoxedChild][i]]
            + j*width] = 0;
        DemoImageVector[ParentFirstPixel[BestParent[BoxedChild][i]]
            + j*width + (ParentWidth - 1)] = 0;
    }
    for (j=0; j<ParentWidth; j++) {
        DemoImageVector[ParentFirstPixel[BestParent[BoxedChild][i]]
            + (ParentWidth-1)*width+j] = 0;
    }
}

```

```

    }
}
OutputFilename = "fractalcode.fic";
ProgressMemo->AppendText("Writing to: "+OutputFilename+"\n");
if ((OutputFile = fopen(OutputFilename,"w")) != NULL) {
    // "w"=write
    //writing to a file that will contain our fractal code.
    sprintf(TempChar,"%d",width);
    fprintf(OutputFile,"%s\n",TempChar);
    sprintf(TempChar,"%d",height);
    fprintf(OutputFile,"%s\n",TempChar);
    sprintf(TempChar,"%d",ParentWidth);
    fprintf(OutputFile,"%s\n",TempChar);
    sprintf(TempChar,"%d",ParentHeight);
    fprintf(OutputFile,"%s\n",TempChar);
    sprintf(TempChar,"%d",ChildWidth);
    fprintf(OutputFile,"%s\n",TempChar);
    sprintf(TempChar,"%d",ChildHeight);
    fprintf(OutputFile,"%s\n",TempChar);
    sprintf(TempChar,"%d",XParentShift);
    fprintf(OutputFile,"%s\n",TempChar);
    sprintf(TempChar,"%d",YParentShift);
    fprintf(OutputFile,"%s\n",TempChar);
    sprintf(TempChar,"%d",XOverlap);
    fprintf(OutputFile,"%s\n",TempChar);
    sprintf(TempChar,"%d",YOverlap);
    fprintf(OutputFile,"%s\n",TempChar);
    sprintf(TempChar,"%d",NumberToAverage);
    fprintf(OutputFile,"%s\n", TempChar);
    for (i=0; i<NumChildren; i++) {
        for (j=1; j<=NumberToAverage; j++) {
            sprintf(TempChar,"%d",BestParent[i][j]);
            fprintf(OutputFile,"%s\n",TempChar);
            sprintf(TempChar,"%f",BestAlpha[i][j]);
            fprintf(OutputFile,"%s\n",TempChar);
            sprintf(TempChar,"%f",BestBeta[i][j]);
            fprintf(OutputFile,"%s\n",TempChar);
            sprintf(TempChar,"%d",ParentOrientation[i][j]);
            fprintf(OutputFile,"%s\n",TempChar);
        }
    }
}
if (ParentChildMatchCheckbox->IsChecked() == true) {

```



```

    if (BoxedChild > NumChildren) {
        ProgressMemo->AppendText("The chosen child does not exist.");
    }
    else if (FreeImage_Save(FIF_BMP, DemoImage,
        "fractaldemo.bmp", 0)){
        ProgressMemo->AppendText("Saved demo picture!");
    }
}
if (BadPixelCheckBox->IsChecked() == true) {
    if (FreeImage_Save(FIF_BMP, BadPixelImage,
        "fractalbadpixel.bmp", 0)){
        ProgressMemo->AppendText("Saved bad pixels picture!");
    }
}
fclose(OutputFile);
}
event.Skip();
}/*
 * ParentChildMatchCheckboxClick
 */
void FractalImageDlg::ParentChildMatchCheckboxClick(wxCommandEvent&
event) {
    if (ParentChildMatchCheckbox->IsChecked() == true) {
        ChildMemo->Enable(true);
    }
    else ChildMemo->Enable(false);
    event.Skip();
} /*
 * BadPixelCheckBoxClick
 */
void FractalImageDlg::BadPixelCheckBoxClick(wxCommandEvent& event) {
    if (BadPixelCheckBox->IsChecked() == true) {
        BadPixelMemo->Enable(true);
    }
    else BadPixelMemo->Enable(false);
    event.Skip();
}

```

The following is the main program for the Fractal Image Decoder that produces decompressed, denoised, and/or recovered digital images.

```
//-----
//
// Name:      FicDecoderDlg.cpp
// Author:     Matthew Demers
// Created:    6/14/2006 1:33:03 AM
//
//-----

#include "FicDecoderDlg.h" #include <iostream> #include <stdio.h>
#include <stdlib.h> #include <math.h> #include <wx/string.h>
#include <freeimage.h>
//Do not add custom headers.
//wx-dvcpp designer will remove them
////Header Include Start
////Header Include End

//-----
// FicDecoderDlg
//-----

//Add Custom Events only in the appropriate Block.
// Code added in other places will be removed by wx-dvcpp
////Event Table Start
BEGIN_EVENT_TABLE(FicDecoderDlg,wxDialog)
////Manual Code Start
////Manual Code End

EVT_CLOSE(FicDecoderDlg::FicDecoderDlgClose)
EVT_BUTTON(ID_DECODEBUTTON,FicDecoderDlg::DecodeButtonClick)
EVT_BUTTON(ID_LOADCODEBUTTON,FicDecoderDlg::LoadCodeButtonClick)
EVT_CHECKBOX(ID_WXCHECKBOX1,FicDecoderDlg::WxCheckBox1Click)
EVT_BUTTON(ID_LOADIMAGEBUTTON,FicDecoderDlg::LoadImageButtonClick)
END_EVENT_TABLE()
////Event Table End

FIBITMAP *InputImage;
//FIBITMAP *InputImage2;
```

```
FIBITMAP *OutputImage; BYTE *InputImageVector; BYTE
*OutputImageVector; BYTE *OutputImageVector2;
```

```
wxString TempStr; char TempChar[20];
```

```
FicDecoderDlg::FicDecoderDlg( wxWindow *parent, wxWindowID id, const
wxString &title, const wxPoint &position, const wxSize& size, long
style )
```

```
    : wxDialog( parent, id, title, position, size, style)
{
    CreateGUIControls();
}
```

```
FicDecoderDlg::~FicDecoderDlg() { }
```

```
void FicDecoderDlg::CreateGUIControls(void) {
    //Do not add custom Code here
    //wx-devcpp designer will remove them.
    //Add the custom code before or after the Blocks
    ///GUI Items Creation Start
```

```
WxPanel1 = new wxPanel(this, ID_WXPANEL1, wxPoint(2,5),
    wxSize(502,354) );
```

```
ProgressMemo = new wxTextCtrl(WxPanel1, ID_PROGRESSMEMO, "" ,
    wxPoint(253,24),wxSize(232,307) , wxTE_MULTILINE);
ProgressMemo->SetMaxLength(0);
ProgressMemo->AppendText(_("ProgressMemo"));
```

```
LoadImageButton = new wxButton(WxPanel1, ID_LOADIMAGEBUTTON,
    _("Load Image") , wxPoint(12,49),wxSize(105,38) );
```

```
LoadImageMemo = new wxTextCtrl(WxPanel1, ID_LOADIMAGEMEMO,
    _("LoadImageMemo") , wxPoint(126,56),wxSize(118,21) );
LoadImageMemo->SetMaxLength(0);
```

```
WxCheckBox1 = new wxCheckBox(WxPanel1, ID_WXCHECKBOX1,
    _("WxCheckBox1") , wxPoint(14,23),wxSize(15,15) );
```

```
WxStaticText1 = new wxStaticText(WxPanel1, ID_WXSTATICTEXT1 ,
    _("Use a Black Image") ,wxPoint(34,24),wxSize(94,17) );
```

```

LoadCodeButton = new wxButton(WxPanel1, ID_LOADCODEBUTTON,
    _("Load Code") , wxPoint(13,106),wxSize(103,38) );

LoadCodeMemo = new wxTextCtrl(WxPanel1, ID_LOADCODEMEMO,
    _("LoadCodeMemo") , wxPoint(126,111),wxSize(116,21) );
LoadCodeMemo->SetMaxLength(0);

WxStaticLine1 = new wxStaticLine(WxPanel1, ID_WXSTATICLINE1,
    wxPoint(11,163),wxSize(231,2) );

DecodeButton = new wxButton(WxPanel1, ID_DECODEBUTTON,
    _("D E C O D E") , wxPoint(49,240),wxSize(145,39) );

LoadImageDialog = new wxFileDialog(this, "Choose a file" ,
    "" , "" , "*.*", wxOPEN);

LoadCodeDialog = new wxFileDialog(this, "Choose a file" , "" ,
    "" , "*.*", wxOPEN);

IterationsSlider = new wxSlider(WxPanel1, ID_ITERATIONSSLIDER,
    0,0,50,wxPoint(52,187),wxSize(138,40) , wxSL_LABELS |
    wxSL_LEFT | wxSL_TOP | wxSL_HORIZONTAL);
IterationsSlider->SetRange(0,50);
IterationsSlider->SetValue(0);

this->SetSize(8,8,519,398);
this->SetTitle(_("FicDecoder"));
this->Center();
this->SetIcon(wxNullIcon);

////GUI Items Creation End
}

void FicDecoderDlg::FicDecoderDlgClose(wxCloseEvent& event) {
    // --> Don't use Close with a wxDialog,
    // use Destroy instead.
    Destroy();
} /*
    * DecodeButtonClick
    */
void FicDecoderDlg::DecodeButtonClick(wxCommandEvent& event) {
    unsigned ParentWidth, ParentHeight, ChildWidth, ChildHeight;

```

```

unsigned XParentShift, YParentShift, XOverlap, YOverlap;
unsigned NumOfParentsAcross, NumOfParentsDown;
unsigned NumOfChildrenAcross, NumOfChildrenDown;
unsigned NumParents, NumChildren;
unsigned NumOfOverlaps;
unsigned NumberToAverage;
unsigned WidthScaling, HeightScaling;
unsigned x, y, i, j, test, CurrentChild;
i = 0;
j = 0;
NumberToAverage = 1;
unsigned NumOfIterations;
unsigned width, height;
unsigned LoopStart, LoopEnd;
unsigned BestParent[4096][30];
unsigned ParentFirstPixel[4096];
unsigned ChildFirstPixel[4096];
unsigned ParentOrientation[4096][30];
float BestAlpha[4096][30], BestBeta[4096][30];
wxString ParentWidthStr, ParentHeightStr, ChildWidthStr,
    ChildHeightStr;
wxString XParentShiftStr, YParentShiftStr, XOverlapStr,
    YOverlapStr;
wxString NumberToAverageStr;
wxString widthStr, heightStr;
wxString CodeFilename;
wxString xStr, yStr, CurrentChildStr;
FILE *CodeFile;

CodeFilename=LoadCodeMemo->GetValue();
ProgressMemo->AppendText("Reading from: "+CodeFilename+"\n");
if ((CodeFile = fopen(CodeFilename,"r")) != NULL) { // "r"=read

    fscanf(CodeFile,"%s",TempChar);
    widthStr=TempChar;
    ProgressMemo->AppendText("Dimensions: "+widthStr+" x ");
    width=atoi(TempChar);

    fscanf(CodeFile,"%s",TempChar);
    heightStr=TempChar;
    ProgressMemo->AppendText(heightStr+"\n");
    height = atoi(TempChar);

```

```
fscanf(CodeFile,"%s",TempChar);
ParentWidthStr=TempChar;
ProgressMemo->AppendText(ParentWidthStr+"\n");
ParentWidth = atoi(TempChar);

fscanf(CodeFile,"%s",TempChar);
ParentHeightStr=TempChar;
ProgressMemo->AppendText(ParentHeightStr+"\n");
ParentHeight = atoi(TempChar);

fscanf(CodeFile,"%s",TempChar);
ChildWidthStr=TempChar;
ProgressMemo->AppendText(ChildWidthStr+"\n");
ChildWidth = atoi(TempChar);

fscanf(CodeFile,"%s",TempChar);
ChildHeightStr=TempChar;
ProgressMemo->AppendText(ChildHeightStr+"\n");
ChildHeight = atoi(TempChar);

fscanf(CodeFile,"%s",TempChar);
XParentShiftStr=TempChar;
ProgressMemo->AppendText(XParentShiftStr+"\n");
XParentShift = atoi(TempChar);

fscanf(CodeFile,"%s",TempChar);
YParentShiftStr=TempChar;
ProgressMemo->AppendText(YParentShiftStr+"\n");
YParentShift = atoi(TempChar);

fscanf(CodeFile,"%s",TempChar);
XOverlapStr=TempChar;
ProgressMemo->AppendText(XOverlapStr+"\n");
XOverlap = atoi(TempChar);

fscanf(CodeFile,"%s",TempChar);
YOverlapStr=TempChar;
ProgressMemo->AppendText(YOverlapStr+"\n");
YOverlap = atoi(TempChar);

fscanf(CodeFile,"%s",TempChar);
```

```

NumberToAverageStr=TempChar;
ProgressMemo->AppendText(NumberToAverage+"\n");
NumberToAverage = atoi(TempChar);

if (ParentWidth>=XParentShift) {
    NumOfParentsAcross = int((width+XParentShift-ParentWidth)/
        XParentShift);
}
else NumOfParentsAcross = int(width/XParentShift);

if (ParentHeight>=YParentShift) {
    NumOfParentsDown = int((height+YParentShift-ParentHeight)/
        YParentShift);
}
else NumOfParentsDown = int(height/YParentShift);

NumOfOverlaps = int((width - ChildWidth)/
    (ChildWidth - XOverlap));
NumOfChildrenAcross = int((width + XOverlap*NumOfOverlaps)/
    ChildWidth );
NumOfOverlaps = int((height - ChildHeight)/
    (ChildHeight - YOverlap));
NumOfChildrenDown = int((height + YOverlap*NumOfOverlaps)/
    ChildHeight);
NumParents = NumOfParentsAcross * NumOfParentsDown;
NumChildren = NumOfChildrenAcross * NumOfChildrenDown;

WidthScaling = int(ParentWidth/ChildWidth);
HeightScaling = int(ParentHeight/ChildHeight);

for (y=0; y < NumOfParentsDown; y++) {
    for (x=0; x< NumOfParentsAcross; x++) {
        ParentFirstPixel[i]=x*XParentShift+y*YParentShift*width;
        i++;
    }
}
i=0;
for (y=0; y<NumOfChildrenDown; y++) {
    for (x=0; x<NumOfChildrenAcross; x++) {
        //Preliminary Easy-Way-Out. Defines child-first pixels, like in
        //the encoding program, assuming the the child blocks tile the
        //image perfectly.

```

```

ChildFirstPixel[i]=x*ChildWidth + y*ChildWidth*width;
i++;
//End easy-way-out
if (y == 0) {
    if (x == 0) {
        //x, y both at min (bottom left corner?)
    }
    else if (x == NumOfChildrenAcross - 1) {
        //x at max, y at min (bottom right corner?)
    }
    else; // y at min (bottom edge?)
}

else if (x == 0) {
    if (y == NumOfChildrenDown - 1) {
    }
    //x at min, y at max (top left corner?)
    else; //x at min (left edge?)
}

else if (y == NumOfChildrenDown - 1) {
    if (x == NumOfChildrenAcross - 1) {
        //x, y both at max (top right corner?)
    }
    else; //y at max (top edge?)
}

else if (x == NumOfChildrenAcross - 1) {
    //x at max (right edge?)
}
else; //x,y not at max: we're somewhere in the middle.
}
}

for (CurrentChild=0; CurrentChild<NumChildren; CurrentChild++) {
    for (j=1; j<=NumberToAverage; j++) {
        //reads the information from the fractal code file
        fscanf(CodeFile,"%s",TempChar);
        BestParent[CurrentChild][j]=atoi(TempChar);
        fscanf(CodeFile,"%s",TempChar);
        BestAlpha[CurrentChild][j]=atof(TempChar);
        fscanf(CodeFile,"%s",TempChar);
        BestBeta[CurrentChild][j]=atof(TempChar);
    }
}

```



```

        fscanf(CodeFile,"%s",TempChar);
        ParentOrientation[CurrentChild][j]=atoi(TempChar);
    }
}
fclose(CodeFile);
//Decoding time!
NumOfIterations = IterationsSlider->GetValue();
// Create the initial output image / seed.
if (WxCheckBox1->GetValue()==true) { // load from file
    OutputImage=FreeImage_Allocate(width,height,8);
    // use a pattern; black is the only choice offered here.
}
else {
    OutputImage=FreeImage_Load(FIF_BMP,
        LoadImageMemo->GetValue(),BMP_DEFAULT);
}

if (OutputImage) { // we've got an output image!
    ProgressMemo->AppendText("Bitmap successfully initialized.\n");
    // Crucial step! FreeImage needs me to define a palette!
    RGBQUAD *pal = FreeImage_GetPalette(OutputImage);
    for (i=0; i<256; i++) {
        pal[i].rgbRed = i;
        pal[i].rgbGreen = i;
        pal[i].rgbBlue = i;
    }
    // Get the OutputImage's bits.
    BYTE *OutputImageVector=(BYTE *)FreeImage_GetBits(OutputImage);

    //Initializing values for a "clone" image...
    OutputImageVector2 = OutputImageVector;

    //This is it! The IFS obtained during compression is iterated
    //with this loop. After enough iterations, an approximation
    //of the target image is produced.
    for (i=0; i<NumOfIterations; i++){
        for (CurrentChild=0; CurrentChild<NumChildren; CurrentChild++){
            for (y=0; y<ChildHeight; y++) {

                LoopStart=ChildFirstPixel[CurrentChild]+y*width;
                LoopEnd = LoopStart+ChildWidth;
            }
        }
    }
}

```

```

for (x=LoopStart; x<LoopEnd; x++) {
    test = 0;
    for (j=1; j<=NumberToAverage; j++) {
        if (ParentOrientation[CurrentChild][j] == 1) {
            test = test + int(BestAlpha[CurrentChild][j]*
                OutputImageVector2[int(ParentFirstPixel[BestParent
                    [CurrentChild][j]]+y*width*HeightScaling+(x-LoopStart)
                    *WidthScaling))+BestBeta[CurrentChild][j]);
        }
        else if (ParentOrientation[CurrentChild][j] == 2) {
            test = test + int(BestAlpha[CurrentChild][j]*
                OutputImageVector2[int(ParentFirstPixel[BestParent
                    [CurrentChild][j]]+ (LoopEnd-x-1)*WidthScaling +
                    y*HeightScaling*width))+BestBeta[CurrentChild][j]);
        }
        else if (ParentOrientation[CurrentChild][j] == 3) {
            test = test + int(BestAlpha[CurrentChild][j]*
                OutputImageVector2[int((ParentFirstPixel[BestParent
                    [CurrentChild][j]]+ (ChildHeight-(y+1))*
                    HeightScaling*width) + (x-LoopStart)*WidthScaling))
                    +BestBeta[CurrentChild][j]);
        }
        else if (ParentOrientation[CurrentChild][j] == 4) {
            test = test + int(BestAlpha[CurrentChild][j]*
                OutputImageVector2[int((ParentFirstPixel[BestParent
                    [CurrentChild][j]]-1 + (ChildHeight-(y+1))*
                    HeightScaling*width) + (LoopEnd-x)*WidthScaling))
                    +BestBeta[CurrentChild][j]);
        }
        else if (ParentOrientation[CurrentChild][j] == 5) {
            test = test + int(BestAlpha[CurrentChild][j]*
                OutputImageVector2[int(ParentFirstPixel[BestParent
                    [CurrentChild][j]]-1 - y*WidthScaling + ParentHeight
                    + (x - LoopStart)*HeightScaling*width))
                    +BestBeta[CurrentChild][j]);
        }
        else if (ParentOrientation[CurrentChild][j] == 6) {
            test = test + int(BestAlpha[CurrentChild][j]*
                OutputImageVector2[int(ParentFirstPixel[BestParent
                    [CurrentChild][j]]-1 - y*WidthScaling + ParentHeight
                    + (LoopEnd - x - 1)*HeightScaling*width))
                    +BestBeta[CurrentChild][j]);
        }
    }
}

```

```

    }
    else if (ParentOrientation[CurrentChild][j] == 7) {
        test = test + int(BestAlpha[CurrentChild][j]*
            OutputImageVector2[int(ParentFirstPixel[BestParent
            [CurrentChild][j]]- width - y*WidthScaling
            + (LoopEnd - x)*HeightScaling*width))
            +BestBeta[CurrentChild][j]);
    }
    else if (ParentOrientation[CurrentChild][j] == 8) {
        test = test + int(BestAlpha[CurrentChild][j]*
            OutputImageVector2[int(ParentFirstPixel[BestParent
            [CurrentChild][j]] + y*WidthScaling + (x - LoopStart)
            *HeightScaling*width))+BestBeta[CurrentChild][j]);
    }
    }
    OutputImageVector[x]=int(test/NumberToAverage);
}
}
}
OutputImageVector2 = OutputImageVector;
}
}
if (FreeImage_Save(FIF_BMP, OutputImage,
    "fractalimagenew.bmp", 0)) {
    ProgressMemo->AppendText("Successfully saved!\n");
    // bitmap successfully saved!
}
FreeImage_Unload(OutputImage);
ProgressMemo->AppendText("Unloaded! Byebye!\n");
}
// insert your code here
event.Skip();
} /*
 * LoadImageButtonClick
 */
void FicDecoderDlg::LoadImageButtonClick(wxCommandEvent& event) {
    unsigned width,height;          // dimensions of input image
    wxString widthStr,heightStr;    // string versions
    wxString string1, string2;
    LoadImageDialog->ShowModal(); //select file to load
    string1=LoadImageDialog->GetDirectory();
    string2=LoadImageDialog->GetFilename();

```

```

LoadImageMemo->SetValue(string1+"\\\\"+string2);

InputImage = FreeImage_Load(FIF_BMP, LoadImageMemo->GetValue(),
    BMP_DEFAULT);

if (InputImage) { // bitmap successfully loaded!
    ProgressMemo->AppendText("Successfully loaded!\n\n");
    if (FreeImage_GetWidth(InputImage) == 256){
        if (FreeImage_GetHeight(InputImage) == 256){;
            width=FreeImage_GetWidth(InputImage);
            height=FreeImage_GetHeight(InputImage);
            sprintf(TempChar,"%d",width);
            widthStr=TempChar;
            sprintf(TempChar,"%d",height);
            heightStr=TempChar;
            ProgressMemo->AppendText("Dimensions: "+widthStr+" x "
                +heightStr+"\n\n");
            InputImageVector = (BYTE *) FreeImage_GetBits(InputImage);
        }
        else ProgressMemo->AppendText("Currently, this program only
            supports 256x256 images.\n\n");
    }
    else ProgressMemo->AppendText("Currently, this program only
        supports 256x256 images.\n\n");
    }
    else ProgressMemo->AppendText("File not located.\n\n");
    event.Skip();
} /*
 * WxCheckBox1Click
 */
void FicDecoderDlg::WxCheckBox1Click(wxCommandEvent& event) {
    if (WxCheckBox1->GetValue()==true) {
        LoadImageButton->Enable(false);
        LoadImageMemo->Enable(false);
    }
    else if (WxCheckBox1->GetValue()==false) {
        LoadImageButton->Enable(true);
        LoadImageMemo->Enable(true);
    }
    event.Skip();
} /*

```

```
* LoadCodeButtonClick
*/
void FicDecoderDlg::LoadCodeButtonClick(wxCommandEvent& event) {
    wxString string1, string2;
    LoadCodeDialog->ShowModal();
    string1=LoadCodeDialog->GetDirectory();
    string2=LoadCodeDialog->GetFilename();
    LoadCodeMemo->SetValue(string1+"\\ "+string2);
    event.Skip(); }
```

Bibliography

- [1] M.F. Barnsley. *Fractals Everywhere*. Academic Press, New York, 1988.
- [2] Y. Fisher. *Fractal Image Compression, Theory and Applications*. Springer, New York, 1995.
- [3] N. Lu. *Fractal Imaging*. Academic Press, New York, 1997.
- [4] M.F. Barnsley. *Superfractals*. Academic Press, London, 2006.
- [5] A. E. Jacquin. Fractal image coding based on a theory of iterated contractive image transformations. In M. Kunt, editor, *Proc. SPIE Vol. 1360, p. 227-239, Visual Communications and Image Processing '90: Fifth in a Series, Murat Kunt; Ed.*, pages 227–239, September 1990.
- [6] R.S. Strichartz. *The Way of Analysis*. Jones and Bartlett Publishers, Inc., Sudbury, Massachussets, 2000.
- [7] M.F. Barnsley, V. Ervin, D. Hardin, and J.Lancaster. Solution of an inverse problem for fractals and other sets. *Proc. Natl. Acad. Sci.*, 83:1975–1977, 1985.

- [8] J. Hutchinson. Fractals and self-similarity. 30:713–747, 1981.
- [9] H. Kunze, K. Levere, and M. Demers. Fractals tops and colour stealing. In *Maple Conference 2006 Proceedings*, 2006.
- [10] M.F. Barnsley. Transformations between attractors of hyperbolic iterated function systems. 2006.
- [11] E. Vrscay and D. Saupe. Can one break the ‘collage barrier’ in fractal image coding. In E. Lutton M. Dekking, J. Levy-Vehel and C. Tricot, editors, *Fractals: Theory and Applications in Engineering*, pages 307–323. Springer Verlag, 1999.
- [12] H. Kunze, J. Hicken, and E.R. Vrscay. Inverse problems for ODEs using contraction maps: Suboptimality of the “collage method”. *Inverse Problems*, 20:977–991, 2004.
- [13] P. Centore and E.R. Vrscay. Continuity of attractors and invariant measures of iterated function systems. *Can. Math. Bull.*, 37:315–329, 1994.