FRACTAL INTERPOLATION TECHNIQUES

A THESIS

The Honors College

The University of Southern Mississippi

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science

In the Department of Mathematics

by

Shywanda Ruffin Moore

May 1999

Approved by:

Joseph Kolibal
Department of Mathematics

Wallace C. Pye, Chair
Department of Mathematics

Maureen Ryan, Dean
Honors College

CONTENTS

1 Introduction **1**
1.1 Background 1
1.2 Background on Interpolation 2
1.3 Higher Order Approximations 4
1.4 Fractal Interpolation 6
1.5 Computational Considerations 8

2 Fractals and Iterated Functions Systems **9**
2.1 History and Background 9
2.2 Basic Mathematical Concepts Used in Understanding Fractals 10
2.3 Iterated Function Systems (IFS) 13
2.4 Fractal Interpolation Techniques 15

3 Comparisons **17**
3.1 Linear Interpolation 17
3.2 Linear Iterated Function Systems, Linear Interpolation 18
3.3 Piecewise Linear Interpolation vs. Piecewise Linear IFS 20
3.4 Quadratic Interpolation 21
3.5 Comparing Smooth versus Rough Interpolation 23

4 Conclusion **28**
4.1 Summary 28
4.2 Conclusions 29

A COMPUTER PROGRAMS **30**
A.1 Linear Interpolation 30
A.2 Piecewise Linear Interpolation 31
A.3 Quadratic Interpolation 33
A.4 Cubic Splines 35
A.5 Linear IFS 37
A.6 Piecewise Linear IFS 38
A.7 Quadratic IFS 40
A.8 General IFS 41
A.9 Poisson Distribution 43

LIST OF ILLUSTRATIONS

2.1 Illustration of graph of non-self similar fractal 10

3.1 Graph of linear interpolation function 18
3.2 Comparison of standard and linear IFS 20
3.3 Graph of piecewise linear IFS 22
3.4 Quadratic IFS interpolation 23
3.5 Cubic spline interpolation compared with general IFS 25

Chapter 1

Introduction

1.1 Background

In this thesis the investigations center around mechanisms for interpolating data and the implications and attributes of assessing computational techniques for the efficient construction of approximations to functions which are known only at a discrete, fixed number of points. While the properties of standard approximation methods have been well cataloged, approximations based on fractal methods using iterative techniques present new challenges.

In constructing polynomial approximations, it is well known that there are a significant number of standard measures of quality which must be considered, and that problems can develop in attempting to utilize polynomial approximation methods. Issues which affect polynomial interpolation, such as data sampling (i.e., selectively sampling the space in place of using regularly spaced intervals), convergence rates and error estimates, are well established for standard polynomial interpolation methods. In contrast, fractal interpolation methods and the possible advantages of using interative methods are less well established. If the graphs of the approximation becomes extremely oscillatory, other methods such as spline interpolation or piecewise interpolation are used. Thus, it is important to consider the design of algorithms which can be used to construct fractal interpolation splines as well.

There are many other techniques for finding approximations to data other than interpolation. In interpolation the approximation p which is constructed has the property that $p(x_i) = F_i$ for each data point (x_i, F_i). Alternative methods for fitting data include least squares approximation and other best-fit methods. While

these are important, they are beyond the scope of this work and the objective is to compare and contrast classical techniques using polynomial interpolation with fractal interpolation methods.

1.2 Background on Interpolation

Consider a set of data points of the form $\{(x_i, F_i) \in \mathbb{R}^2 : i = 0, 1, 2, \ldots, N\}$, where

$$a = x_0 < x_1 < x_2 < \ldots < x_N = b. \tag{1.1}$$

An interpolation function corresponding to this set of data is a continuous function f which satisfies

$$f(x_i) = F_i \qquad \text{for } i = 0, 1, 2, \ldots, N. \tag{1.2}$$

The points (x_i, F_i) are called the interpolation points and the function f defined on the interval $[a, b]$ interpolates the data when the graph of f passes through the interpolation points [1]. Note that if there are $N + 1$ data points as given in (1.2), then there is a unique polynomial of degree N which interpolates these points. This is because a general polynomial of degree N has $N + 1$ coefficients $a_0, a_1, \ldots a_{N-1}, a_N$ which can be uniquely specified. Polynomial interpolation [11] provides for the construction of a polynomial functions which are derivable from the data. For example, the unique first-degree interpolating polynomial $a_1 x + a_0 = f(x)$ fits the data in the sense that

$$a_1 x_0 + a_0 = F_0 \qquad a_1 x_1 + a_0 = F_1 \tag{1.3}$$

In other words, the interpolant is the straight line passing through points (x_0, F_0) and (x_1, F_1).

As will be seen, the interpolant, when it is a polynomial, is rarely of degree higher than three because of problems which develop in attempting to construct approximations in which the degree of the polynomial approaches the number of data points. The difficulty is intrinsic to polynomial approximation. Thus whether the polynomial approximation is constructed using classical methods based on finite differences or based on fractal methods, the degree of the polynomial will be constrained to be three or less, from practical considerations. Instead of attempting to construct a single polynomial over the entire field of data, the technique is to break up the domain into smaller subdomains, on which a lower degree polynomial is constructed.

There are many methods for doing the decomposition. These approaches are classified as piecewise polynomial approximations. For example, instead of finding the polynomial

$$p_N(x) = a_0 + a_1 x + \ldots + a_{N-1} x^{N-1} + a_N x^N$$

which interpolates the data points in (1.2) on the interval $[a, b]$, the linear polynomials, $p_1^j(x)$, are constructed on each of the subdomains $I_j = [x_j, x_{j+1}]$, $j = 0, 1, \ldots N-1$, where

$$[x_0, x_1] \cup [x_1, x_2] \cup [x_2, x_3] \cup \ldots \cup [x_{N-1}, x_N] = [a, b]. \tag{1.4}$$

Let $p : [a, b] \to \mathbb{R}$ denote the unique function that passes through the interpolation points and which is linear on each of the subintervals I_j. Then $p(x)$ is given by specifying

$$p_1^j(x) = F_{j-1} + [(x - x_j)/(x_{j+1} - x_j)] (F_{j+1} - F_j), \qquad x \in I_j, \tag{1.5}$$

that is, the function $p(x)$ is defined to be p_1^j on each interval I_j and is called the piecewise linear interpolation function for the given data.

The difficulty with polynomial interpolation on any interval is that as the number of interpolation points N increases, even as the approximation improves near the center of the interval, the behavior tends to be oscillatory at the ends of the interval. If it is possible to choose the x_i freely on the interval of definition of the data, it is know that a better, less oscillatory fit can be achieved than for evenly spaced intervals [3]. Thus, at least part of the task in extending interpolation methods to fractal methods must involve the consideration of techniques which do not rely on uniformly spaced data.

1.3 Higher Order Approximations

For this thesis, the construction of quadratic interpolants using standard and fractal methods was examined. The extension to cubic approximations focused on cubic splines, illustrating the use of these to construct smooth approximations (up to the second derivative) to data. At the other extreme, instead of attempting the construction of cubic fractal interpolation methods, the investigation compared these to the construction of extremely rough interpolants which can be generated using fractal methods (discussed in Chapter 2).

1.3.1 Constructing Quadratic Interpolants

The construction of quadratic interpolation functions [3] can be accomplished by taking a polynomial of the form

$$p_2(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) \tag{1.6}$$

for appropriate constants a_0, a_1, and a_n, and determining the divided differences of f with respect to x_0, x_1, and x_2.

1.3.2 Constructing Splines

A continuous function $S(x)$, defined on a finite interval $[a, b]$, is called a spline function of degree $k > 0$ (order $k + 1$), having as knots the strictly increasing sequence,

$$\lambda_j, \qquad j = 0, 1, \ldots, N, \qquad \lambda_0 = a, \qquad \lambda_N = b. \tag{1.7}$$

provided on each knot interval $[\lambda_j, \lambda_{j+1}]$, $S(x)$ is given by a polynomial of at most degree k.

1.3.3 Cubic Splines

The most common piecewise polynomial method uses cubic splines. Cubic spline interpolants obey the following conditions:

(a) S is a cubic polynomial, denoted S_j, on the subinterval $[x_j, x_{j+1}]$ for each $j = 0, 1, \ldots, N - 1$;

(b) $S(x_j) = f(x_j)$ for each $j = 0, 1, \ldots, N$;

(c) $S_{j+1}(x_{j+1}) = S_j(x_{j+1})$ for each $j = 0, 1, \ldots, N - 2$;

(d) $S'_{j+1}(x_{j+1}) = S'_j(x_{j+1})$ for each $j = 0, 1, \ldots, N - 2$;

(e) $S''_{j+1}(x_{j+1}) = S''_j(x_{j+1})$ for each $j = 0, 1, \ldots, N - 2$; and,

(f) One of the following conditions is satisfied:

  (i) $S''(x_0) = S''(x_N) = 0$ (free or natural boundary); or,

  (ii) $S'(x_0) = f'(x_0)$ and $S'(x_N) = f'(x_N)$ (clamped boundary).

When the free boundary conditions occur, the spline is called a natural spline, and its graph approximates the shape a long flexible rod would assume if forced to go through each of the data points [3].

To solve the system of equations which result from using the cubic spline, the Crout factorization for tridiagonal linear systems can be used. Thus, requiring the solution of the following $N \times N$ linear system,

$$\begin{aligned}
E_1 &: \quad a_{11}x_1 + a_{12}x_2 &&= a_{1,N+1} \\
E_2 &: \quad a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &&= a_{2,N+1} \\
\vdots & \qquad \vdots && \quad \vdots \\
E_{N-1} &: \quad a_{N-1,N-2}x_{N-2} + a_{N-1,N-1}x_{N-1} + a_{N-1,N}x_N &&= a_{N-1,N+1} \\
E_N &: \quad a_{N,N-1}x_{N-1} + a_{N,N}x_N &&= a_{N-1,N+1}
\end{aligned} \tag{1.8}$$

where the a_{ij} are determined by the coefficients. The interpolation and convergence properties of spline theory have been extensively studied [9].

1.4 Fractal Interpolation

The history of iterated function systems and fractals is discussed in [7]. The derivation of the formula for the dimension of self-similar sets is contained in Moran in 1946. Later information involving computers and other technology can be found in the works of Mandelbrot in 1982, Dekking in 1982, Barnsley and Demko in 1985, and Barnsley in 1988. A discussion of self-affine sets is given by Mandelbrot in 1986 and applications to image compression and processing were described in Barnsley (1988) and Barnsley and Sloan in 1988. The first account of iterated function systems is that of Hutchinson in 1981, although there were other similar ideas around earlier.

The transition from fractals to Iterated Function Systems (IFS) is natural in that the IFS provides a framework for the description, classification, and generation of fractals. This will be discussed more fully in Sec. 3.2. The technique of fractal interpolation using IFS's differs from classical interpolation in that there are no difference quotients involved; that is, the approximating polynomial is not constructed by the

use of any finite difference scheme. Instead, the polynomial is obtained by iterating a system of equations until the desired result, i.e., the polynomial approximation, is obtained.

There is a more substantial difference in these two approaches to constructing polynomial interpolants. In classical interpolation, the coefficients of the polynomial are determined by an algorithm which can be generalized to polynomials of any degree. In the case of fractal interpolation, the technique requires the construction of functions, whose composition determines the graph of the approximation. The construction of these IFS's is more complicated.

1.4.1 The Comparison of IFS to Standard Interpolation

In this thesis standard interpolation techniques are examined for constructing linear, piecewise linear, piecewise quadratic and piecewise cubic interpolants. These are compared to four IFS systems which were constructed. The first constructed was a simple linear IFS that took two vector valued functions, w_1 and w_2, which connected the pair of points (x_1, y_1) and (x_2, y_2) using an iteration sequence . In other words, the attractor of the IFS, is the graph of the interpolation function $f(x)$ over the interval $[x_1, x_2]$. The second IFS which was constructed is a piecewise linear IFS which took individual linear IFS and pieced them together to form a new function. The next IFS constructed was a parabola. This particular IFS was used to show that a specific quadratic could be formed. However, a more generalized quadratic IFS would be difficult to construct. The last IFS constructed was simply a general IFS. This particular IFS is constructed in $\mathbb{R}^2$, such that its attractor is the graph of a continuous function $f : [x_0, x_N] \to \mathbb{R}$ which interpolates the data.

1.5 Computational Considerations

The computation of all numerical examples and work performed in this study is done using the C programming language. There are any number of alternative languages which could have been used to construct the algorithms, however it was felt that C was consistent with current computational techniques and standards. All of the software developed for this study is listed in Appendix A.

Chapter 2

Fractals and Iterated Functions Systems

2.1 History and Background

The term "fractals" was coined in 1977 by Mandelbrot. It was taken from the Latin word *fractus* meaning broken pieces. The term was used to describe objects that were not easily fit into traditional geometrical settings [7]. Fractals are typically constructed using iterative algorithms. The applications for fractals are numerous, including fractal art, however the usual purpose in constructing fractals lies in their ability to represent how objects in nature occur. They are used in weather prediction models, population growth and decline models and in models describing the flow of fluids. For the purpose of this thesis, fractals provide a mathematical tool which is applicable to the generation and interpolation of graphical data.

In discussing fractals, it is important to observe that there appears to be some qualitative differences between fractals. Some fractals are self-similar, while others are not. Fractals which are self similar have the property that an enlarged plot of a small subinterval of the fractal curve looks like the original plot [15]. Fractals which are not self similar, or which have different scales of self-similarity are more typical of the randomness found in nature.

To compare fractals, various measures are used. A geometrical set A is considered to be a fractal if

$$D = \lim_{\epsilon \to 0} \frac{\ln \mathcal{N}(A, \epsilon)}{\ln(1/\epsilon)} \tag{2.1}$$

exists, where $\mathcal{N}(A, \epsilon)$ is the least number of balls of radius $\epsilon > 0$ which covers A. If the limit exists, then D is called the fractal dimension of A. Fractal dimension

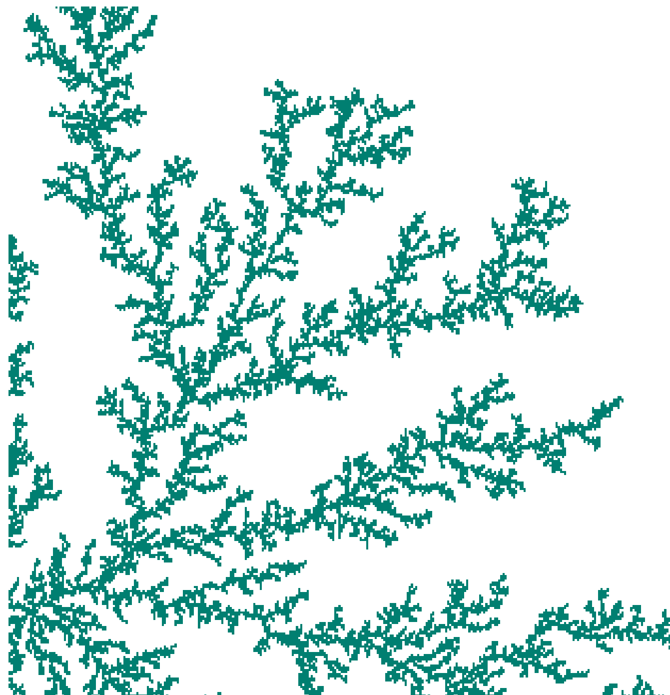

Figure 2.1: Illustration of the graph of non-self similar fractal from [10].

affords a measure of how densely populated a metric space is by a fractal set. A plane is a fractal with dimension 2 and a line is a fractal having dimension 1, while a fractal having dimension between 1 and 2 can be considered a "proper" fractal. The dimension of a fractal can be computed numerically, or computed theoretically if it is a fractal which is self similar. An example of a self-similar fractal is the Cantor Set, and a fractal which is not self-similar is illustrated in Fig 2.1.

2.2 Basic Mathematical Concepts Used in Understanding Fractals

To understand how fractal curves can be used to approximate data, it is necessary to develop some basic topological ideas of fractals. Since fractals are often constructed algorithmically by repeated applications of some elementary rules, it is natural to embed the development of fractals into a framework which can be described by Iterated Function Systems (IFS's). The construction of the fractal can be understood in the context of an algorithm, however in order to understand the convergence properties of the construction it is necessary that the IFS construction

or algorithm be considered with a background in the mathematics of metric spaces, particularly the Hausdorff metric space, transformations, and contraction mappings.

2.2.1 Metric Spaces

A metric space $(\mathbf{X}, d)$ is a space $\mathbf{X}$ coupled with a real-valued function

$$d : \mathbf{X} \times \mathbf{X} \rightarrow \mathbb{R},$$

which measures the distance between pairs of points in $\mathbf{X}$. The function d must satisfy certain requirements:

(i) $d(x, y) = d(y, x), \;\; \forall x, y \in \mathbf{X}$;

(ii) $0 < d(x, y) < \infty, \;\; \forall x, y \in \mathbf{X}, x \neq y$;

(iii) $d(x, x) = 0, \;\; \forall x \in \mathbf{X}$; and,

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

The function d is called a metric or distance function on the space $\mathbf{X}$. The usual metric is given by the Euclidean distance between two points.

A metric space is complete if every Cauchy sequence converges to a point $x \in \mathbf{X}$. A Cauchy sequence is a sequence $\{x_n\}_{n=1}^{\infty}$ of points in a metric space $(\mathbf{X}, d)$ such that for any given number $\varepsilon > 0$, there is $N \in \mathcal{Z}$, where $N > 0$ such that $d(x_n, x_m) < \varepsilon, \;\; \forall n, m > N$.

There is a mathematical space in which the convergence properties of the IFS construction used to generate fractals are naturally set. If $(\mathbf{X}, d)$ is a complete metric space then the Hausdorff metric space, denoted $\mathcal{H}(\mathbf{X})$, is the space whose points are the compact subsets of $\mathbf{X}$. The distance $d(x, \mathrm{B})$ defined

$$d(x, \mathrm{B}) = \min\{d(x, y) : y \in B\}.$$

is the distance form the point x to the set B. The distance from a set $A \in \mathcal{H}(\mathbf{X})$ to

the set $B \in \mathcal{H}(\mathbf{X})$ is given by

$$d(A, B) = \max_{x \in A} d(x, B).$$

The Hausdorff distance function, d_h which is used to construct the Hausdorff metric space, is then given by

$$d_h = \max\{d(A, B), d(B, A)\}. \tag{2.2}$$

Thus in a Hausdorff metric space contained in $\mathbb{R}^2$, for example, it is not the individual coordinate points (x, y) which are the members of the set, $\mathcal{H}(\mathbf{X})$, instead the members are the compact subsets.

2.2.2 Transformations

In order to work within the geometry of fractals there is also a need to introduce transformations acting on the space. A transformation on the space $\mathbf{X}$ is a function $f : \mathbf{X} \to \mathbf{X}$, which assigns exactly one point $f(x)$ to each point $x \in \mathbf{X}$. A transformation f on a metric space is a contraction mapping if there is a constant $0 \leq s < 1$ such that

$$d(f(x), f(y)) \leq s \cdot d(x, y), \;\; \forall \;\; x, y \in \mathbf{X};$$

the scalar s is called a contractivity factor for f.

More importantly, to later understand the concept of certain iterated function systems, a specific type of transformation must be discussed. This is called the *shear transformation*. This thesis uses shear transformations which map lines parallel to the y-axis into lines parallel to the y-axis. The transformation obeys four linear equations specified by five real numbers, a_n, c_n, d_n, e_n, and f_n, with $n = 0, 1, \ldots, N$:

$$a_n x_0 + e_n = x_{n-1} \tag{2.3}$$

$$a_n x_N + e_n = x_n \tag{2.4}$$

$$c_n x_0 + d_n F_0 + f_n = F_{n-1} \tag{2.5}$$

$$c_n x_N + d_n F_N + f_n = F_n \tag{2.6}$$

where d_n is taken to be a free parameter and where F_j is the value of the function at x_j, $j = 0, 1, \ldots, N$. Then, this will define the shear transformation w_n which is introduced in Sec. 3.5 to develop the fractal interpolation schemes.

2.3 Iterated Function Systems (IFS)

The first noted account of iterated function schemes or systems was in 1981 by Hutchinson, however, there were similar ideas around before this [7]. An IFS is defined to be a complete metric space $(\mathbf{X}, d)$ coupled with a finite set of contraction mappings

$$w_n : \mathbf{X} \to \mathbf{X}, \tag{2.7}$$

with contractivity factors s_n. The notation for an IFS is

$$\{\mathbf{X};\ w_n, n = 1, 2, \ldots, N\}. \tag{2.8}$$

An IFS has a unique fixed point $A \in \mathcal{H}(\mathbf{X})$. This fixed point is called the attractor of the IFS, and is obtained from a finite set of maps acting on a complete metric space [1].[1]

[1] Sometimes it may seem that data points move at random but they always appear to remain close to a particular set for a convergent IFS.

The purpose for developing this machinery is in order to demonstrate that the iterated maps (IFS's) used to generate the fractal set, are convergent, i.e., that there is indeed a unique set which is obtained by iterating the functions infinitely. The Hausdorff metric is used to measure the convergence of the iterates generated by the IFS to the desired, converged set. One technique for constructing functional iterations which converge is to construct them so that they are contraction mappings. In this manner, on each iteration, the points are mapped by the functional iteration to new points which lie closer to each other than on the previous iteration by a factor s.

The demonstration of the convergence of the IFS's is beyond the scope of this thesis, however it is important to note that the convergence properties of the IFS's used in the thesis can be demonstrated using the construction outlined. In fact, the IFS's used in this thesis can be shown to consist of contraction mappings which have the required properties to assure convergence.

In this study, the IFS's were constructed using affine linear maps, i.e., linear transformations of the plane which include a translation in the plane based on the Deterministic Algorithm of Barnsley [1]. To create visual images of the attractors of an IFS, the affine linear maps are iterated to an acceptable level of convergence (typically requiring a large number of iterations) and the results are displayed by plotting the coordinates of the points. Since the IFS's consist of affine linear maps, w_i, this can be expressed as the matrix equation

$$w_i(x) = A_i x + t_i = x' \tag{2.9}$$

or, explicitly

$$w_i(x) = w_i \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \end{bmatrix} = \begin{bmatrix} x_1' \\ x_2' \end{bmatrix} \tag{2.10}$$

in which the point $(x_1, x_2) \in \mathbb{R}^2$ is transformed to the point (x_1', x_2') and where a_i,

b_i, c_i, d_i, e_i and f_i are constants for each map w_i.

Barnsley's Deterministic Algorithm is constructed from these affine linear maps acting on the space $\mathbf{X}$. Let $X; w_1, w_2, \ldots, w_N$ be an IFS. Choose a compact set A_0, then construct the sets

$$A_{n+1} = \bigcup_{j=1}^{N} w_j(A_n) \qquad \text{for } n = 0, 1, 2, \ldots. \tag{2.11}$$

This constructs a sequence, $\{A_n; n = 0, 1, 2, 3, \ldots\}$, of compact sets generated by mappings on a metric space which can be shown (under suitable conditions) to converge to the attractor, A, in the Hausdorff metric.

2.4 Fractal Interpolation Techniques

The application of Barnsley's Deterministic Algorithm to sets in the plane is the technique used in this thesis to interpolate data to produce fractal interpolations. The traditional methods for analyzing experimental data have been based on Euclidean geometry and elementary functions. As discussed in Chapter 1 these typically yield approximations which are polynomials. Fractal interpolation techniques are different both in construction and in the results that are produced. While fractal interpolation can produce polynomial approximations, i.e., where the graph of the attractor is that of a polynomial, they can also produce interpolants which are fractal curves in the plane. The graphs of these are typically like the graph of the non-self similar fractal shown in Fig. 2.1. They have a random appearance. The purpose in constructing these interpolants is clearly different from the purpose in constructing polynomial interpolants. If the data are assumed to be taken from a function which is relatively smooth, then using polynomial interpolation, either constructed using standard methods, or constructed using an IFS provides a consistent representation of the data. If however, the data are not assumed to be smooth, then

the fractal set generated by the IFS may be more consistent with the shape of the data.

Fractal interpolation functions are comprised of simple mathematical expressions which can be computed rapidly by computer. Clearly, the application of (2.10) and (2.11) manually is tedious and difficult, however algorithmically these are not more difficult to compute than the discrete differences used to construct standard polynomial interpolants.

Chapter 3

Comparisons

3.1 Linear Interpolation

Linear interpolation is the approximation of a function f by using a first degree polynomial. In the general case, the linear polynomial passing through $(x_0, f(x_0))$ and $(x_1, f(x_1))$ is constructed using the quotients

$$L_0(x) = (x - x_1)/(x_0 - x_1) \tag{3.1}$$

and

$$L_1(x) = (x - x_0)/(x_1 - x_0) \tag{3.2}$$

When $x = x_0, L_0(x_0) = 1$ and $L_1(x_0) = 0$. When $x = x_1, L_0(x_1) = 0$ and $L_1(x_1) = 1$.

For the purposes of this study, two points, (x_1, y_1) and (x_2, y_2) were chosen. The linear interpolation program used those points to construct a line segment. These points were then connected by N interpolation points by using an iterative method that calculated y_i for each new x_i on the interpolation function. The program used to generate the linear polynomial which interpolates the two points was tested and a typical output from the program in which ten interpolation points were specified between the endpoints $(0, 0)$ and $(1, 2)$, is shown in Fig. 3.1.

The resulting line which fills in the data between the two endpoints is not remarkable, however it does illustrate some attributes of linear interpolation. The interpolation points introduce no local maxima or minima. This can be shown to be typical of linear or quadratic interpolation, however it is not the case with higher order approximations in which oscillatory behavior of the interpolant can become a problem.

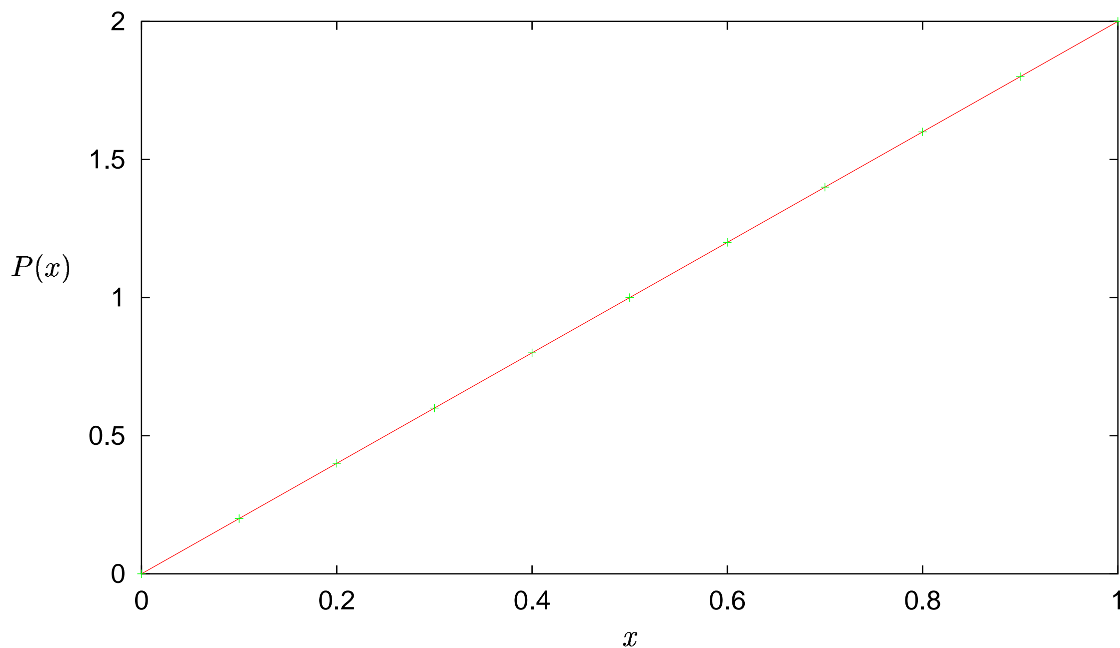

Figure 3.1: Graph of linear interpolation function.

3.2 Linear Iterated Function Systems, Linear Interpolation

The first comparison that was done was between the linear interpolation function constructed using standard methods in Sec. 3.1 and the linear IFS shown in Fig. 3.2 The linear IFS constructs the interpolation points using the Deterministic algorithm (2.3)–(2.6). In this program, an interactive technique was used to input the data. The user was asked to enter the number of interpolation points and to enter two points (x_1, y_1) and (x_2, y_2) to be interpolated by the program. Those points were then taken by the program and entered into an IFS $\{\mathbb{R}^2; w_1, w_2\}$ where

$$w_1 \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 0.5 \end{pmatrix}, \tag{3.3}$$

and

$$w_2 \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0.5 \\ 1 \end{pmatrix}. \tag{3.4}$$

To construct the desired interpolation data, the line must be computed from both the left and right sides. To compute the left side, the initial point was set to $(x_2, y_2) = (1, 2)$. These points were then decremented by a set value. To compute the right side, the initial values were set to $(x_1, y_1) = (0, 1)$ and incremented by the set value. Once incremented, the values were then output to a file to be plotted.

Notice that both graphs are similar in construction and appearance. For this particular example, there is no difference in the methods in regard to the output results, except that it is evident from Fig. 3.2 that the IFS generates interpolation data which are not uniformly spaced, while the standard polynomial construction produces interpolants which (in this case) are uniformly spaced. The clustering of the output data at the endpoints of the interpolation interval is typical of the results which were obtained using the Deterministic algorithm. It is important to realize, though, that the set of interpolation points drawn in the figure is collectively the single point in the Hausdorff space which is the attractor for the system. In this case the attractor is seen to be the desired straight line joining the two specified endpoints. Less evident from the static picture of the attractor is the mechanism by which the point set for IFS interpolant is constructed.

Computationally, the linear interpolating function and the linear IFS took comparable amounts of time to execute for the same number of output data, however in order to fill in the points in the center of the graph, it was required to start from more initial points. In general, in attempting to construct smooth interpolants using the IFS method, this is a problem and the computer time required to fill in the graph is actually several times longer.

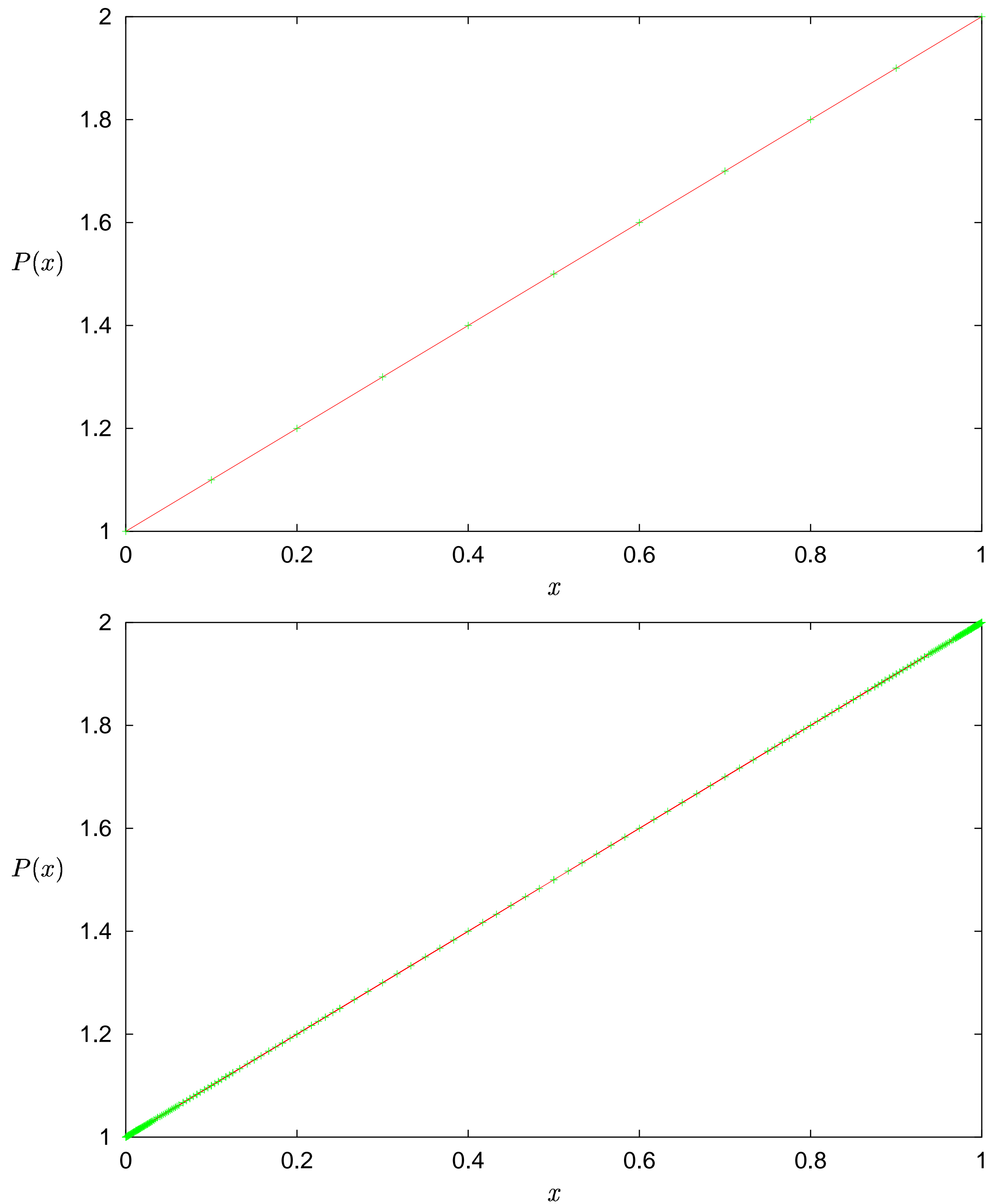

Figure 3.2: Comparison of standard (top) and linear IFS (bottom) to interpolate between the points (0,1) and (1,2) showing the interpolation points (dots).

3.3 Piecewise Linear Interpolation vs. Piecewise Linear IFS

The simplest piecewise polynomial approximation is piecewise linear interpolation. This is the joining of a set of data points

$$(x_0, f(x_0)), (x_1, f(x_1)), \ldots, (x_n, f(x_n)) \tag{3.5}$$

by a series of straight lines. A disadvantage of linear approximation is that differentiability is not guaranteed, i.e., the function may not be "smooth" at these points. The piecewise linear interpolation function pieces together several different line segments and creates a new, continuous function. The piecewise polynomial approximation can be constructed using standard interpolation methods as described in (1.4)–(1.5), or by using fractal methods.

For the fractal construction, the graph of the piecewise linear approximation $f(x)$ is also the attractor of an IFS of the form $\{\mathbb{R}^2; w_n, n = 1, 2, \ldots, N\}$ where the maps are affine. The transformation is of the form:

$$w_n \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_n & 0 \\ c_n & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e_n \\ f_n \end{pmatrix} \tag{3.6}$$

where the coefficients are obtained from the shear transformation described in (2.3)–(2.6), i.e., the piecewise linear function was constructed similarly to the general IFS, except that the d_n have been set to 0.

3.4 Quadratic Interpolation

The quadratic IFS examined in this thesis has an attractor defined by $f(x) = 2x - x^2$ on the interval $[0, 2]$. This IFS is defined by:

$$w_1 \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1/2 & 0 \\ 1/2 & 1/4 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{3.7}$$

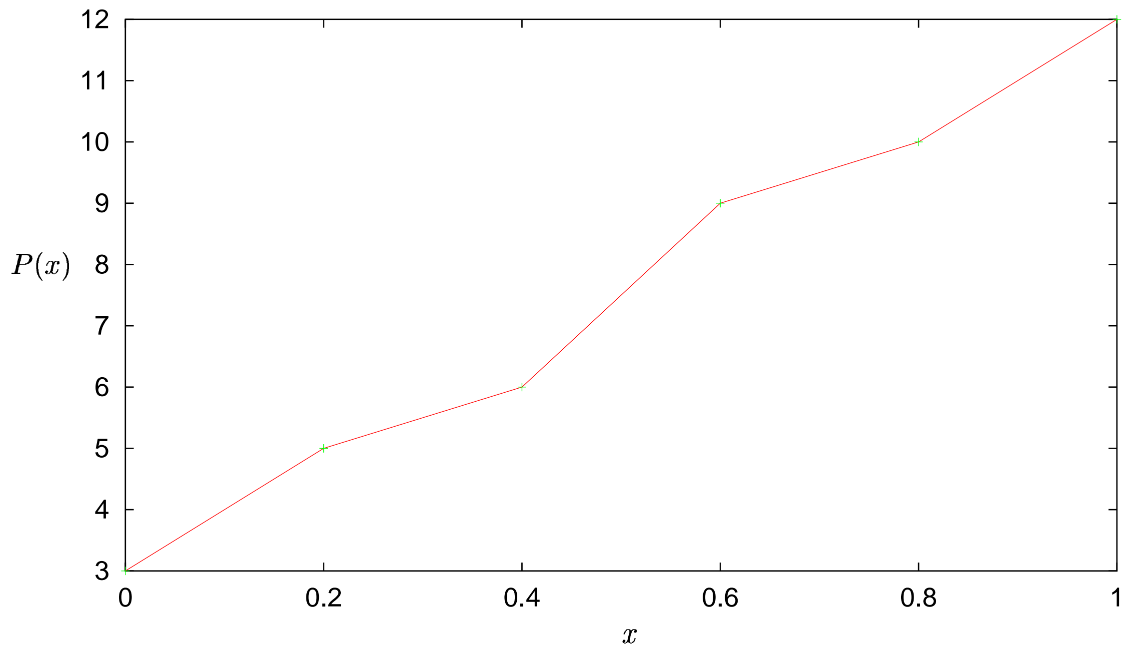

Figure 3.3: Graph of piecewise linear IFS.

and

$$w_2 \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1/2 & 0 \\ -1/2 & 1/4 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{3.8}$$

where the graph of $f(x)$, G, is equal to $\{(x, 2x - x^2) : x \in [0, 2]\}$ and is the attractor of the IFS. Like the linear IFS, this IFS was constructed in two parts. The right side of the graph was computed by first setting the value of both x and y. This was done using decrements and increments, as well. This amounted to computing points satisfying the equations

$$x' = \frac{x}{2} + 1; \quad \text{and} \quad y' = -\frac{x}{2} + \frac{y}{4} + 1; \tag{3.9}$$

and yielding the part of the fractal that covered the interval $[1, 2]$. The left side was computed in a similar fashion with the equations being

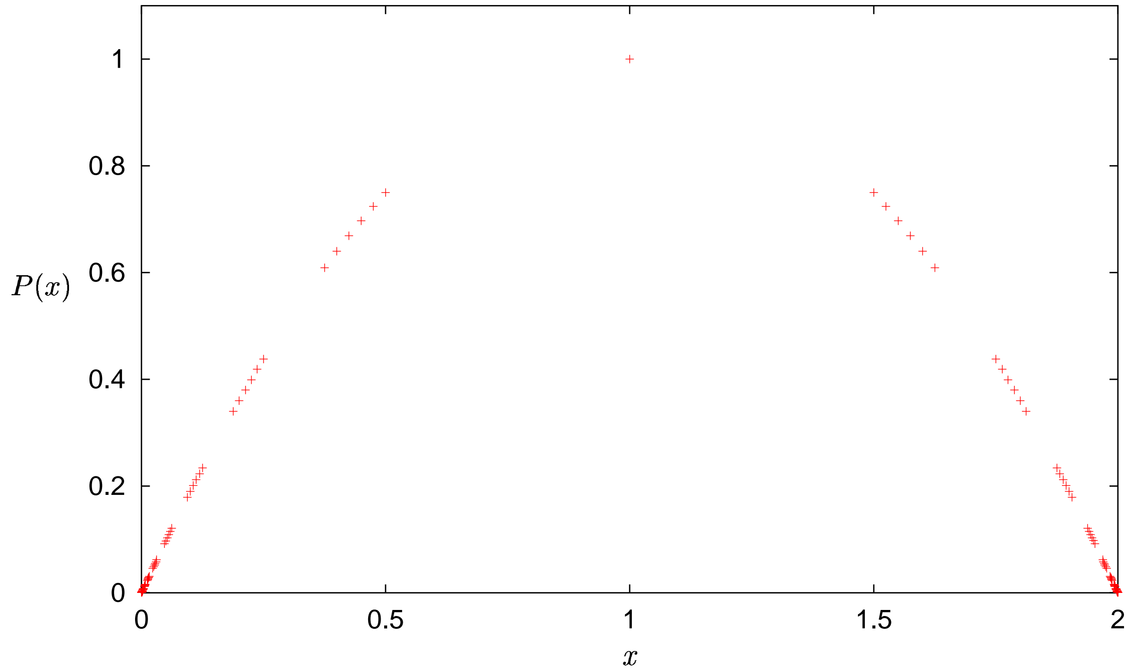

Figure 3.4: Quadratic IFS interpolation for data taken from the function $y = 2x - x^2$ showing that the IFS recovers the data on the quadratic curve.

$$x' = \frac{x}{2}; \quad \text{and} \quad y' = \frac{x}{2} + \frac{y}{4} \tag{3.10}$$

constructing the interval $[0, 1]$. Hence, $G = w_1(G) \cup w_2(G)$. Note that the graph generated by the IFS, shown in Fig. 3.4, is sparse toward the center of the interval $[0, 2]$, and contains most of the points clustering toward the endpoints. This is typical of the behavior of the linear IFS's used.

3.5 Comparing Smooth versus Rough Interpolation

The construction of the cubic spline assures that there is both a continuous first derivative and a continuous second derivative. Interpolation using a general IFS produces results which are quite rough. The general IFS, which is of the form $\{\mathbb{R}^2; w_n, n = 1, 2, \ldots, N\}$, is an affine transformation with the composition

x	y
-1.0	-1.0
0.0	0.0
1.0	1.0
2.0	8.0
3.0	9.0

$a[j]$	$b[j]$	$c[j]$	$d[j]$
-1.00	1.54	0.00	-0.54
0.00	-0.07	-1.61	2.68
1.00	4.75	6.43	-4.18
8.00	5.07	-6.11	2.04

Table 3.1: Sample data used to construct cubic spline approximation and IFS approximation (left) and computed coefficients of cubic spline approximation (right) used in computing $S(x)$ in (3.14).

$$w_n \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_n & 0 \\ c_n & d_n \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e_n \\ f_n \end{pmatrix} \tag{3.11}$$

Unlike the other IFS structures, this particular IFS has constraints with

$$w_n \begin{pmatrix} x_0 \\ F_0 \end{pmatrix} = \begin{pmatrix} x_{n-1} \\ F_{n-1} \end{pmatrix} \tag{3.12}$$

and

$$w_n \begin{pmatrix} x_N \\ F_N \end{pmatrix} = \begin{pmatrix} x_n \\ F_n \end{pmatrix} \quad \text{for } n = 1, 2, \ldots, N. \tag{3.13}$$

This particular IFS also has a free parameter. In this case it was chosen to be d_n which served as the vertical scaling factor for the IFS. The data points (x_n, F_n) were read from a data file and the d_n were generated and a random iteration function was then applied to the IFS. This algorithm is a modified version of the Basic Random Iteration Algorithm given by Barnsley such that the graph of the IFS passes through all interpolation points and constructs a unique attractor to the IFS[1].

The results of applying cubic spline versus a general IFS to the problem of interpolating the same data are illustrated in Fig. 3.5. Notice the oscillatory nature of the general IFS.

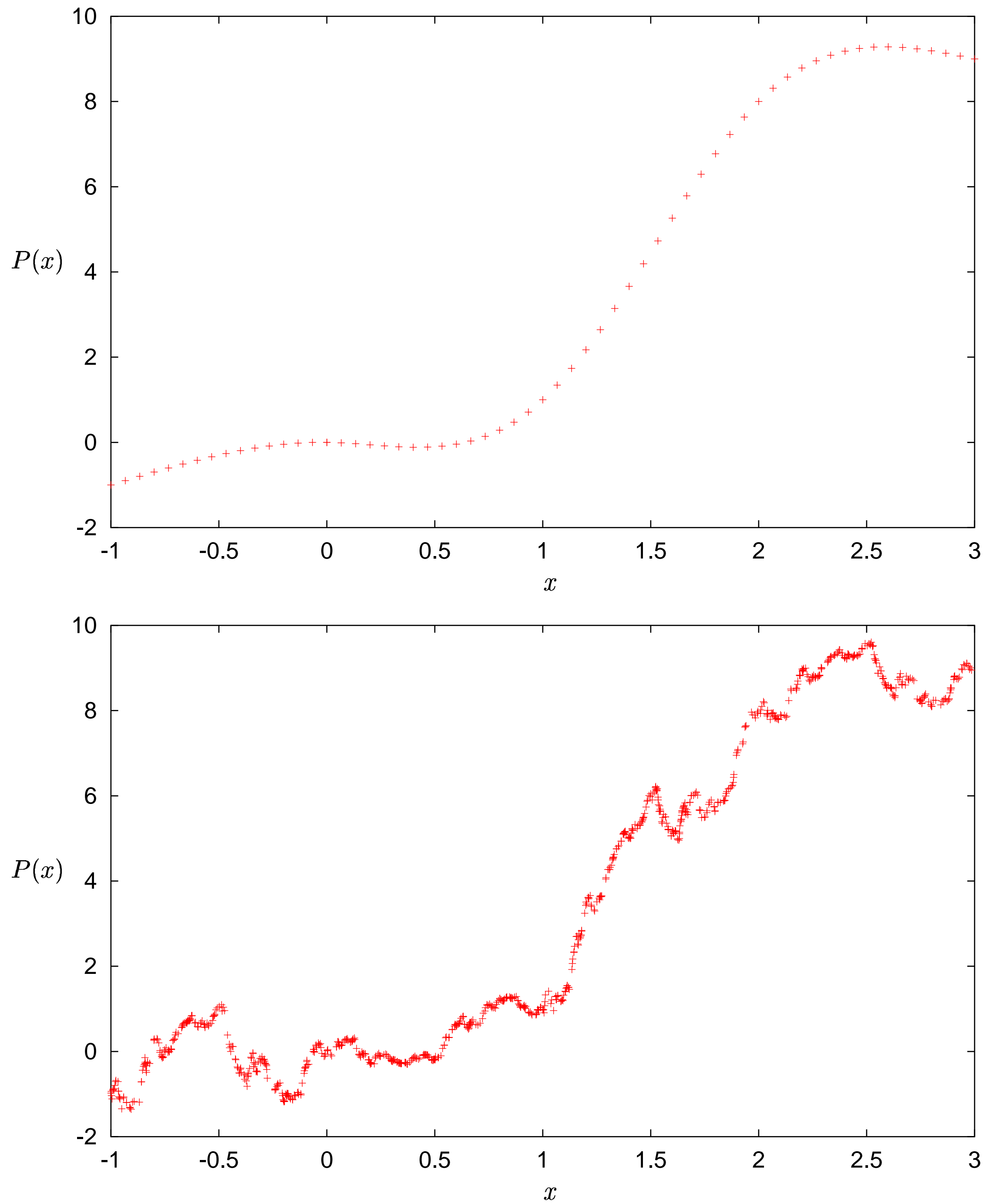

Figure 3.5: Approximating data using cubic spline interpolation (top) compared with approximating data using a general IFS (bottom).

In the implementation of the algorithm, the resulting output will depend on the choice of coefficients for the IFS. Clearly, this allows for the construction of a range of possible (acceptable) interpolants. In this particular example, the data used to

d_1	d_2	d_3	d_4	d_5
0.50000	-0.50000	0.23000	-0.51080	-0.35670

Table 3.2: Example of arbitrary coefficients d_n used to generate IFS approximation.

generate the cubic spline interpolant are shown in Table 3.1. In the cubic spline algorithm, these values are used to calculate the coefficients of the cubic spline equation, which are also shown in Table 3.1 The coefficients are then substituted into the equation

$$S(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3, \qquad x_j \leq x < x_{j+1}. \tag{3.14}$$

The general IFS is constructed and generated much differently than the cubic spline. For the same input data set (again from Table 3.1) the output which results is substantially different from that obtained by using the cubic spline algorithm. In this example, the data were input into the General IFS program. Unlike, the cubic spline, the general IFS has vertical scaling factors. For the program to run properly, the total number of scaling factors, d_n, must be equal to the number of coordinates read in. The coefficients d_n were arbitrarily generated (random generation would provide the greatest variability in the shape of the resulting fractal). The vertical scaling factors for this example are shown in Table 3.2.

The data were then entered into the program in order to calculate the shear transformation for this particular IFS example. After the shear transformations are calculated using these values in (2.3)–(2.6), the data are then pushed through a Random Iteration algorithm[8]. The random iteration produces an integer value of k between 1 and N. This value of k is then used as an index for the transformation matrices. The new values for x and y are then calculated and output. The details are provided in the code in Appendix A. In this example, there were 1000 data points produced. As is evident from Fig. 3.5 the random iteration inherent in the

IFS produces a graph that is quite oscillatory and fractal-like in nature.

Clearly, the randomization which is produced in iterating the functional compositions need not be entirely random in the sense that the distribution is taken from a uniformly distributed set. Instead, it is possible with this algorithm to construct graphs which are generated from other pdfs, for example the Poisson or Gaussian distributions. While these produce some interesting graphs (not too dissimilar in some cases from those where uniformly random random numbers are used) it is an unresolved as to whether the resulting output data retains any of the statistical character of the original pdf. These are considerations that can be examined in future investigations. The details of sampling from a Poisson distribution are provided in the code in Appendix A.

The lack of smoothness, or the roughness of the IFS-generated graph is not entirely a drawback. While it is not a useful technique for the construction of data in the solution of problems requiring smoothness, such as the solution of partial differential equations, the technique is applicable if the interest is in constructing images involving natural data such as the outline of a mountain.

Chapter 4

Conclusion

4.1 Summary

While the techniques for data fitting have been developed over the last century, the use of computers in data fitting has been a recent development. The early techniques for interpolation, being motivated by a desire to provide interpolated function values, were designed to be efficient for manual computation. The extension of these techniques to computer based interpolation was natural and the techniques have been redesigned to improve computational efficiency. However, the development of IFS systems to provide interpolation is a relatively new technique, which like the construction of fractals, requires the use of computers. These approaches to interpolation are less well developed and the possibilities for their use are less well developed.

In this thesis we have attempted to examine several basic interpolation schemes and to examine their properties. We have also examined the use of iterated function systems to accomplish interpolation. Clearly, these are different from the standard techniques used to construct polynomial interpolants to data. They appear to require more computer time (the time could be determined by coding efficiency which was not addressed, however it is more likely that this is true since the algorithms are substantially more complex than for polynomial interpolation), and these techniques are more difficult to implement, especially in regard to developing the traditional smoothed interpolation functions used in classical interpolation. In fact, these techniques are quite the opposite of the classical techniques in that they produce interpolated graphs which can be quite rough (that is, they are not smooth).

Perhaps the most interesting observation is that, of the fractal interpolation techniques which we examined, the ones which produced classical polynomial curves are hardly fractal, i.e., since the attractor is a line, or piecewise continuous polyline, or a quadratic, the dimension D of the 'fractal' is one, hence a degenerate fractal. The issues which are touched on in regard to the mathematics of the structure of the fractals produced are beyond the scope of this paper, however it is important to observe that the IFS technique is being used to produce rough interpolations of data to create artificial landscapes and environments. In these applications it is the standard interpolation techniques which fail.

4.2 Conclusions

In the area of piecewise linear interpolation, the techniques agree, and the IFS produces results using a seemingly meaningless iteration technique. There is one difficulty apparent in the construction which is a drawback from the traditional approach: the resulting output data are not sorted, that is, the values generated are produced in the order of the IFS and not in ascending, line number order. Sorted output data are valuable, and the cost of sorting the data needs to be factored into the cost of producing usable output results. Still, the technique is useful and interesting, providing some interesting alternatives to classical interpolation method.

Apendix A

COMPUTER PROGRAMS

The programs were written in C on a unix system and compiled using the gnu c compiler, gcc. All of the graphics were produced using the package `gnuplot`.

A.1 Linear Interpolation

```
%NOTE THAT THE SECTIONS HAVE CAPITALIZATION ON ALL EXCEPT THE CONNECTIVES.
/*Simple Interpolation */

#include <stdio.h>

main ()

{
        float t,y,x1,y1,x2,y2,N,m,b,x,j;
        FILE *linear; /*linear.gnu*/

        printf ("Enter number of interpolation points.");
        scanf ("%f",&N);
        printf ("Enter x1 coordinate:");
        scanf ("%f",&x1);
        printf ("Enter y1 coordinate:");
        scanf ("%f",&y1);
        printf ("Enter x2 coordinate:");
        scanf ("%f",&x2);
        printf ("Enter y2 coordinate:");
        scanf ("%f",&y2);
        m=((y2-y1)/(x2-x1));
        b=(y2-(m*x2));
        printf ("The slope of the line is %f\t",m);
        printf ("The intercept of the line is %f\n",b);
        printf ("N=%f\n",N);
        x=(-1.0)/N;
        printf ("x=%f\n",x);
        if ((linear=fopen("linear.gnu","w"))==NULL)
                printf ("\n");
        else
                printf ("\n");
        j=x2;
        while (j>=x1) {

                y=m*j+b;
                fprintf (linear,"%.3f\t%.3f\n",j,y);
```

```
                j=j+x;

        }

        if ((j!=x1)&&(j<x1))
                fprintf (linear,"%.3f\t%.3f\n",x1,y1);
        else
                printf ("\n");
        return 0;
}
```

A.2 Piecewise Linear Interpolation

```
/* Piecewise Linear Interpolation */
# define size1 99
#include <stdio.h>
main ()
{
        float n,N,x,y,m,b,temp, xpoints[size1], ypoints[size1],
             xintpoints[size1],yintpoints[size1],a0,aN,slope[size1],
             intercept[size1],aI;
        int i,size,isize,count,current,terminate;

        FILE *pwdata;    /* pwdata=pwl.dat */
        FILE *pwintpts;  /* pwintpts=pwlint.dat */
        FILE *pwout;     /* pwout=pwout.out */
        FILE *pwgnu;     /* pwgnu=pwl.gnu */


        /* Inserted opening statement for gnuplot file */
        if ((pwgnu=fopen("pwl.gnu","w"))==NULL)
                printf ("FIle could not be opened. \n\n");
        else {
                printf ("File was opened.\n");}
        /*  This part of the program reads in the data */


        size=0;
        isize=0;
        if ((pwdata=fopen ("pwl.dat","r"))==NULL)
                printf ("You goofed Sue !!!!!!\n");
        else {
                while (!feof(pwdata)) {
                        fscanf (pwdata,"%f%f",&xpoints[size],&ypoints[size]);
                        size=size+1;}
        }
        size = size - 1;

        if ((pwout=fopen ("pwout.out","w"))==NULL)
                printf ("There has been a mistake");
        else {
                fprintf (pwout,"This program is ready to execute.\n\n");}


        fprintf (pwout,"Program PWL: Piecewise linear interpolation\n\n");
        fprintf (pwout,"SIZE =  %d\n\n", size);
```

```
        fprintf (pwout,"N\t\tX\t\tY\n");
        fprintf (pwout,"__\t\t__\t\t__\n");
        for (count=0; count<(size);){
                fprintf(pwout,"%d\t\t",count);
                fprintf(pwout,"%.3f\t\t",xpoints[count]);
                fprintf(pwout,"%.3f\n",ypoints[count]);
                count=count+1;}


        isize=0;
        if ((pwintpts=fopen ("pwlint.dat","r"))==NULL)
                fprintf (pwout,"You goofed Sue !!!!!!\n");
        else {
                while (!feof(pwintpts)) {
                        fscanf (pwintpts,"%f",&xintpoints[isize]);
                        isize=isize+1;}

        }
        isize = isize - 1;

        /* This part of the program calculates the slope of the line and
                interpolates the data */

        fprintf (pwout,"\nNumber of int. xpoints = \%d\n",isize);
        count=0;
        while (count<=(size-1)){
                m=(ypoints[count+1]-ypoints[count])/
                        (xpoints[count+1]-xpoints[count]);
                b=(ypoints[1]-(m*xpoints[0]));
                slope[count]=m;
                intercept[count]=b;
                fprintf (pwout,"\n \%d \t  m = \%.3f \t b = \%.3f\n",count,m,b);
               count=count+1;}

        size=size-1;
/************************************************************************/
        /* This part of the program scales the data to [0,1] */

        a0 = xpoints[0];
        aN = xpoints[size];
        xpoints[0] = 0.0;
        fprintf (pwout,"\n\n");
        fprintf (pwout," x \t  y\n");
        fprintf (pwout,"___\t ___\n\n");
        fprintf (pwout,"%.3f\t %.3f\n",xpoints[0],ypoints[0]);
        fprintf (pwgnu,"%.3f\t %.3f\n",xpoints[0],ypoints[0]);
        for (count=1; count<=(size-1);count++){
                xpoints[count] = ((xpoints[count] - a0)/(aN-a0));
                fprintf (pwout,"%.3f\t %.3f\n",xpoints[count],ypoints[count]);
                fprintf (pwgnu,"%.3f\t %.3f\n",xpoints[count],ypoints[count]);}
        xpoints[size] = 1.0;
        fprintf (pwout,"%.3f\t %.3f\n",xpoints[size],ypoints[size]);
        fprintf (pwgnu,"%.3f\t%.3f\n",xpoints[size],ypoints[size]);
        aI=(aN-a0);
        fprintf (pwout,"\n\n aN-a0= \%.3f\n\n",aI);
```

```
        for (count=0; count<(isize);){
                xintpoints[count] = (xintpoints[count] - a0)/(aN-a0);
                fprintf (pwout,"%.3f\n",xintpoints[count]);
                count = count + 1;
                 }

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

        /* This part of the program finds the position
           of the interpolation point */

        count=0;
        current=0;

          while (count!=size){
              if (xintpoints[count]<0.0){
                        fprintf (pwout,"This number isn't in this
function domain.\n");
                        count=size;}
              else if (xintpoints[count]>1.0) {
                        fprintf (pwout,"The number isn't in this
function domain.\n");
                        count=size;}
              else if ((pwout,xintpoints[count])==(xpoints[count])) {
                        fprintf (pwout,"This number is equal to a0.\n");
                        count=size;}
              else {    fprintf (pwout,"\n x= %.3f\n",xintpoints[count]);
                        if (xintpoints[count]<=xpoints[count+1]) {
                                fprintf (pwout,"We've found its location!!!!\n\n
");


yintpoints[count]=((slope[count]*xintpoints[count])+
                                                intercept[count]);
                                fprintf (pwout,"\n\n");
                                fprintf (pwout,"The coordinates for the
 interpolation value is: (");
                                fprintf (pwout,"%.3f\t ",xintpoints[count]);
                                fprintf (pwout,",%.3f).\n",yintpoints[count]);
                                count=size;}
                        else
                                {count=count+1;}
               }
        }

return 0;
}
```

A.3 Quadratic Interpolation

```
/* Quadratic Interpolation */

#include <stdio.h>
#define row 1000

main ()
```

```
{

/* Variables */

int size,n,i,j,N;
float a[row],f[row],x[row],b[row],c[row],h[row],
      xpoint[row],incx,delta,square,S[row];

FILE *quad; /*quad=quad.inp*/
FILE *qgnu; /*qgnu=quad.gnu*/
FILE *qpt; /*qpt=qpoint.dat*/

/* READ IN THE DATA POINTS */

size=0;
if ((quad=fopen("quad.inp","r"))==NULL)
        printf ("File could not be opened.\n");
else {
       while (!feof(quad)) {
                fscanf (quad,"%f%f\n",&x[size],&f[size]);
                a[size]=f[size];
                size=size+1;}
}
size=(size-1);

if ((qpt=fopen("qpoint.dat","r"))==NULL)
        printf ("File could not be opened.\n");
else {
        while (!feof(qpt)) {
                fscanf (qpt,"%d\n",&N);
        }
}

if ((qgnu=fopen("quad.gnu","w"))==NULL)
        printf ("\n");
else{
        printf ("\n"); }
/* CALCULATIONS */

n=size;
for (i=0; (i<=(n-1));) {

        h[i]=x[i+1]-x[i];
        i=i+1;}

c[0]=0.0;
b[0]=((f[1]-f[0])/h[0]);

for (i=0; (i<=n-1);) {
        b[i+1]=((2*(f[i+1]-f[i]))/h[i])-b[i];
        i=i+1;}

c[n]=0.0;

for (j=n-1; j>0;) {
        c[j]=(b[j+1]-b[j])/(2*h[j]);
        j=j-1;}
```

```
/* OUTPUTTING THE COEFFICIENTS */

printf ("a\tb\tc\n");
for (j=0; j<=n-1;) {
        printf ("%f\t%f\t%f\n",a[j],b[j],c[j]);
        j=j+1;}
printf ("N=");
printf (" %d\n",N);
printf ("size=");
printf (" %d\n",size);


/* CALCULATING THE QUADRATIC EQUATION*/


i=0;
for (j=0; j<=(n-1);) {
        xpoint[i]=x[j];
        incx=(x[j+1]-x[j])/N;
        while ( (xpoint[i]<x[j+1])&&(xpoint[i]>=x[j]) ) {
                S[i]=(a[j] + (b[j]*(xpoint[i] - x[j]))+
                     (c[j]*(xpoint[i] - x[j])*(xpoint[i] - x[j])));
                printf ("%.4f\t%.4f\n",xpoint[i],S[i]);
                fprintf (qgnu,"%.4f\t%.4f\n",xpoint[i],S[i]);
                i=i+1;
                xpoint[i]=(xpoint[i-1] + incx);
        }
        j=j+1;
}

return 0;

}
```

A.4 Cubic Splines

```
/* CUBIC SPLINES */

#include <stdio.h>
#define size1 1000

main ()

{
int count,n,size,i,j,I,dummy,N;
float x[size1],f[size1],a[size1],h[size1],alpha[size1],mu[size1],b[size1],
        c[size1],d[size1],z[size1],l[size1],xint,xpoint[size1],S[size1],incx;

FILE *cubic;     /*cubic=cubic.dat*/
FILE *cubicgnu;  /*cubicgnu=cubic.gnu*/
FILE *cubicinp;  /*cubicinp=cubic.inp*/

/* READ IN THE DATA POINTS */

size=0;
```

```
if ((cubic=fopen("cubic.dat","r"))==NULL)
        printf ("File could not be opened.\n");
else {
        while (!feof(cubic))  {
                fscanf (cubic,"%f%f\n",&x[size],&a[size]);
                size=size+1;}
     }

size=size-1;
if ((cubicinp=fopen("cubic.inp","r"))==NULL)
        printf ("File could not be opened.\n");
else
        fscanf (cubicinp,"%d\n",&N);

/* CALCULATING THE COEFFICIENTS  */

n=size;

for (i=0; (i<=(n-1));) {
        h[i]=x[i+1] - x[i];
        i=i+1;}

for (i=1; (i<=n-1);) {
        alpha[i]=(3./h[i])*(a[i+1] - a[i]) - (3./h[i-1])*(a[i] - a[i-1]);
        i=i+1;}

l[0]=1;
mu[0]=0;
z[0]=0;

for (i=1; (i<=n-1);) {
/*      l[i]=(2.*(x[i+1] - x[i]) - (h[i-1]*mu[i-1])); ERROR IN SUBSCRIPT ON x[i]
*/
        l[i]= 2.*(x[i+1] - x[i-1]) - (h[i-1]*mu[i-1]);
        mu[i]=h[i]/l[i];
        z[i]=(alpha[i] - h[i-1]*z[i-1])/l[i];
        i=i+1;}

l[n]=1;
z[n]=0;
c[n]=0;

for (j=n-1; (j>=0);) {
        c[j]=z[j] - (mu[j]*c[j+1]);
        b[j]=(a[j+1] - a[j])/h[j] - (h[j]*(c[j+1] + 2.*c[j])/3.);
        d[j]=(c[j+1] - c[j])/(3.*h[j]);
        j=j-1;}



/* Outputting Coefficients */

for (j=0; (j<=n);) {
        printf ("%.2f\t",a[j]);
        printf ("%.2f\t",b[j]);
        printf ("%.2f\t",c[j]);
```

```
        printf ("%.2f\n",d[j]);
        j=j+1;
}
/*******************************************************************/



/* CALCULATING THE CUBIC SPLINE EQUATION */


if ((cubicgnu=fopen("cubic.gnu","w"))==NULL)
        printf ("An error has occurred.  File can't be opened.\n\n");
else {
        i=0;
        for (j=0; j<=n-1;) {
                xpoint[i]=x[j];
                incx=((x[j+1] - x[j])/N);

                while ( (xpoint[i]<x[j+1])&&(xpoint[i]>=x[j]) ) {
                  S[i]=(a[j] + (b[j]*(xpoint[i] - x[j])) + (c[j]*(xpoint[i] - x[
j])
                        *(xpoint[i] - x[j])) + (d[j]*(xpoint[i] - x[j])
                        *(xpoint[i] - x[j])*(xpoint[i] - x[j])));
                  fprintf (cubicgnu,"%.4f\t%.4f\n",xpoint[i],S[i]);
                  i=i+1;
                  xpoint[i]=(xpoint[i-1] + incx);}

                I=i-1;
                j=j+1;}
}

return 0;
}
```

A.5 Linear IFS

```
/* Linear IFS */


#include<stdio.h>

main ()
{

int n,i,size;
float x1,x2,y1,y2,xpoint,ypoint,inc,a,b,
        incx,incy;

FILE *ifs; /* ifs=ifs.dat */
FILE *ifsgnu;/*ifsgnu=ifs.gnu*/
/*      READ IN THE DATA POINTS         */

if ((ifs=fopen("ifs.dat","r"))==NULL)
        printf ("This file could not be opened.\n");
else {
        fscanf (ifs,"%f%f\n",&x1,&y1);
```

```
        fscanf (ifs,"%f%f\n",&x2,&y2);
}

if ((ifsgnu=fopen("ifs.gnu","w"))==NULL)
        printf ("\n");



        fprintf (ifsgnu,"%.3f\t%.3f\n",a,b);
int=-0.1;
inc=0.1;
ypoint=(y1/10)-0.1;;

for (n=0; n<=10;) {
        a=0.5*xpoint;
        b=(0.5*ypoint)+0.5;
        xpoint=xpoint+inc;
        ypoint=ypoint+inc;
        fprintf (ifsgnu,"%.3f\t%.3f\n",a,b);
        n=n+1;
}

xpoint=x1;
ypoint=y1;
inc=0.1;

while (ypoint<=y2) {
        a=(0.5*xpoint)+0.5;
        b=(0.5*ypoint)+1.0;
        xpoint=xpoint+inc;

        ypoint=ypoint+inc;
        fprintf (ifsgnu,"%.3f\t%.3f\n",a,b);
        i=i+1;
}

return 0;
}
```

A.6 Piecewise Linear IFS

```
/* Piecewise linear IFS */


#include <stdio.h>
#define size1 1000
#define size2 1000
main ()

{

int N,n,num,iteration,rand(void),seed,j,k,d;
float x[size1],F[size1],xpoint,ypoint,a[size1],c[size1],e[size1],f[size1],
        w[size1][size2],deltax,xvalue[size1],yvalue[size1],changex,
        xpt,ypt,newx,newy;
```

```
FILE *pwlifs; /*pwlifs=pwlifs.dat*/
FILE *pwlinp; /*pwlinp=pwlifs.inp*/
FILE *pwlignu;/*pwlignu=pwlifs.gnu*/

/* READ IN THE DATA POINTS */

seed=2;
srand(seed);
N=0;

if ((pwlifs=fopen("pwlifs.dat","r"))==NULL)

        printf ("File could not be opened.\n");
else {
        while (!feof(pwlifs)) {
                fscanf (pwlifs,"%f%f\n",&x[N],&F[N]);
                N=N+1;}
}
N=N-1;


if ((pwlignu=fopen("pwlifs.gnu","w"))==NULL)
        printf ("\n");
else
        printf ("\n");

for (n=1; (n<=N);) {
        deltax=(x[N]-x[0]);

        a[n]=(x[n]-x[n-1])/deltax;
        c[n]=(F[n]-F[n-1])/deltax;
        e[n]=((x[N]*x[n-1])-(x[0]*x[n]))/deltax;
        f[n]=((x[N]*F[n-1])-(x[0]*F[N]))/deltax;
        n=n+1;
}

xpt=0.0;
ypt=0.0;

for (j=1; j<=50;) {
        deltax=(a[N]*xpt+e[N])-(a[1]*xpt+e[1]);
        k=1+rand()*(N/2147483648.0);
        newx=(a[k]*xpt+e[k]);
        newy=c[k]*xpt+f[k];
        fprintf (pwlignu,"%.5f\t%.5f\n",newx,newy);
        xpt=newx;
        ypt=newy;
        j=j+1;  }



return 0;

}
```

A.7 Quadratic IFS

```
/* Quadratic IFS */
/* Example 2.4, p.210 Barnsely, Fractals Everywhere */


#include<stdio.h>

main ()
{

int k,n,m,i,isize;
float x1,x2,y1,y2,x,y,xpoint,ypoint,inc,a,b,
        incx,incy;

FILE *ifs; /* ifs=ifs.dat */
FILE *ifsgnu;/*ifsgnu=ifs.gnu*/
/*      READ IN THE DATA POINTS         */

if ((ifs=fopen("q-ifs.dat","r"))==NULL)
printf ("This file could not be opened.\n");
else {
        fscanf (ifs,"%i\n",&m);
        printf ("Number of inner iterations = %i\n",m);
        fscanf (ifs,"%i\n",&isize);
        printf ("Number of initial coordinates = %i\n",isize);

        printf ("%i\n",m);
        fscanf (ifs,"%f%f\n",&x1,&y1);
        fscanf (ifs,"%f%f\n",&x2,&y2);
}

if ((ifsgnu=fopen("q-ifs.gnu","w"))==NULL)
        printf ("\n");

/*      CALCULATIONS    */


/*      Compute the left side of the line       */
inc=0.05;
xpoint=x1;
ypoint=y1;

for (k=1; k<=isize;) {
        x=xpoint-(k-1)*inc;
   y=2.0*x - x*x;
for (n=0; n<=m;) {
         y=0.5*x + 0.25*y;
x=0.5*x;
        fprintf (ifsgnu,"%.3f\t%.3f\n",x,y);
        n=n+1;
}
        k=k+1;
        fprintf (ifsgnu,"\n");
}
```

```
/*      Compute the right side of the line      */
xpoint=x1;
ypoint=y1;

for (k=1; k<=isize;) {
        x=xpoint+(k-1)*inc;
        y=2.0*x - x*x;

for (n=0; n<=m;) {i
 y=-0.5*x + 0.25*y + 1.0;
        x=0.5*x + 1.0;
        fprintf (ifsgnu,"%.3f\t%.3f\n",x,y);
        n=n+1;
}
        k=k+1;
        fprintf (ifsgnu,"\n");
}


return 0;

}
```

A.8 General IFS

```
/* GENERAL IFS */
/* EXAMPLE 2.5 P. 210 BARNSLEY, FRACTALS EVERYWHERE*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define row 1000

main ()

{


int size,n,N,i,p,mean,m[row],rand(void),seed,j,k;
float x[row],F[row],d[row],a[row],e[row],c[row],
        f[row],b,xpt,ypt,number,number2,
        newx,newy;


FILE *gen;
FILE *gengnu;
FILE *genpts;

seed=50;
srand(seed);


/* READ IN THE DATA POINTS*/

n=0;
```

```
if ((gen=fopen("gen.dat","r"))==NULL)
        printf ("This file could not be opened.\n");
else {
        while (!feof(gen)) {
                fscanf (gen,"%f%f\n",&x[n],&F[n]);
                n=n+1;}
        }

N=n-1;

if ((genpts=fopen("gen_pts.dat","r"))==NULL)
        printf ("File could not be opened.\n");
else {
        for (n=0; n<=N; ) {
                fscanf (genpts,"%f\n",&d[n]);
                n=n+1;

        }
}
/*      CALCULATIONS */

b=x[N]-x[0];

for (n=1; n<=N; ) {
        a[n]=(x[n]-x[n-1])/b;
        e[n]=((x[N]*x[n-1])-(x[0]*x[n]))/b;
        c[n]=((F[n]-F[n-1])/b)-(d[n]*((F[N]-F[0])/b));
        f[n]=(((x[N]*F[n-1])-(x[0]*F[n]))/b)-(d[n]*((x[N]*F[0]-x[0]*F[N])/
                b));
        n=n+1;}

if ((gengnu=fopen("gen.gnu","w"))==NULL)
        printf ("\n");
else{
        printf ("\n");}

xpt=0.0;
ypt=0.0;

/*CALCULATING RANDOM NUMBERS */

for (j=1; j<=1000;) {
        k=1+rand()*(N/2147483648.0);
        newx=a[k]*xpt+e[k];
        newy=c[k]*xpt+d[k]*ypt+f[k];
        fprintf (gengnu,"%.5f\t%.5f\n",newx,newy);
        xpt=newx;
        ypt=newy;
        j=j+1;  }

return 0;

}
```

A.9 Poisson Distribution

```
/* POISSON DISTRIBUTION */

#include <stdio.h>
#define size 1000

main ()
{
int  i,n,Y,k,j,ind,size2,r,lambda,lambda2[size];
float p[size],expected[size],prob,exp,total, factorial[size],
        elam;

FILE *poisson;    /*poisson=poisson.dat*/
FILE *poissongnu; /*poissongnu=poisson.gnu*/
FILE *gen;        /*gen=gen.dat*/


/*      READ IN THE DATA        */



if ((poisson=fopen("poisson.dat","r"))==NULL)
        printf ("File could not be opened.\n");
else {
        fscanf (poisson,"%d%d\n",&lambda,&size2);
        printf ("Lambda= ");
        printf ("%d\t%d\n",lambda,size2);
}


exp=2.7183;
k=0;
elam=1.0;


printf ("Size2= ");
printf ("%d\n",size2);

/* Calculating Poisson Distributed Values */

if ((poissongnu=fopen("poisson.gnu","w"))==NULL)
        printf ("\n");
else
        printf ("\n");


/* Calculate the value of e to the minus lambda */


for (r=1; r<=lambda; ) {
        elam=elam*exp;
        r=r+1;
}

elam=(1/elam);
printf ("e to the minus lambda = ");
```

```
printf ("%.6f\n",elam);


/* Calculate the value of lambda for each succesive X value */


lambda2[0]=1;
printf ("Lamda2= ");
for (r=1; r<=size2; ) {
        lambda2[r]=lambda2[r-1]*lambda;
        printf ("%d\n",lambda2[r]);
        r=r+1;
}

printf ("\n");

/* Calculate the value of x! */


factorial[0]=1.;
printf ("Factorial=\n");
for (r=1; r<=size2; ) {
        factorial[r]=factorial[r-1]*r;
        printf ("%f\n",factorial[r]);
        r=r+1;


/* Calculate the value of lambda for each succesive X value */


lambda2[0]=1;
printf ("Lamda2= ");
for (r=1; r<=size2; ) {
        lambda2[r]=lambda2[r-1]*lambda;
        printf ("%d\n",lambda2[r]);
        r=r+1;
}

printf ("\n");

/* Calculate the value of x! */


factorial[0]=1.;
printf ("Factorial=\n");
for (r=1; r<=size2; ) {
        factorial[r]=factorial[r-1]*r;
        printf ("%f\n",factorial[r]);
        r=r+1;
}


/* Calculating the values of the expected probabilities */

printf ("\n");
printf ("P(X)=\n");
while (k<=size2) {
```

```
        if (k==0) {
                factorial[k]=1;
                p[k]=(elam/factorial[k]);
                 printf ("%.6f\n",p[k]);

                }

        else if (k==1) {
                p[k]=(lambda*elam)/(factorial[k]);
                 printf ("%.6f\n",p[k]);

        }


        else {
                p[k]=(lambda2[k]*elam)/factorial[k];
                printf ("%.6f\n",p[k]);

        }
        k=k+1;

}

/* Calculating the expected values */

for (k=0; k<=size2; ) {
        expected[k]=p[k]*size2;
        fprintf (poissongnu,"%d\t%.60f\n",k,expected[k]);
        k=k+1;
}
}


if ((gen=fopen("gen.dat","w"))==NULL)
        printf ("\n");
else
        printf ("\n");


        for (j=0; j<=size2;) {
                fprintf (gen,"%d\t%.60f\n",j,expected[j]);
                j=j+1;
        }

return 0;
}
```

BIBLIOGRAPHY

[1] Michael F. Barnsley. *Fractals Everywhere*. Academic Press Inc., London, 2nd edition, 1993.

[2] John Briggs. *Fractals the Patterns of Chaos*. Simon and Schuster, New York, 1992.

[3] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. PWS, Publishing Company, Boston, fifth edition, 1993.

[4] Janet C. Carr-Burgess. *Interpolation With Shape Control*. PhD thesis, University of Southern Mississippi, Hattiesburg, 1987.

[5] Philip J. Davis. *Interpolation and Approximation*. Blaisdell Publishing, 1963.

[6] Paul Dierckx. *Curve and Surface Fitting with Splines*. Oxford University Press, New York, 1993.

[7] Kenneth Falconer. *Fractal Geometry*. John Wiley and Sons, Ltd., Baffins Lane, Chichester England, 1990. Mathematical Foundations and Applications.

[8] Richard Johnsonbaugh and Martin Kalin. *Applications Programming in ANSI C*. Prentice Hall, New Jersey, third edition, 1996.

[9] Samuel Karlin, Charles Micchelli, Allan Pinkus, and I. J. Schoenberg. *Studies in Spline Functions and Approximation Theory*. Academic Press, Inc., New York, 1976.

[10] Joseph Kolibal and Lynn Ladner. Assessing numerical error in aggregation phenomena. Technical report, Mississippi Academy of Sciences, 1997.

[11] Peter Lancaster and Kestius Salkauskas. *Curve and Surface Fitting, An Introduction*. Academic Press Inc., London, 1986.

[12] Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman and Company, San Francisco, 1983.

[13] Hamed Parsiani and William Navas. Analysis of iterated function system fractals for image compression. *Computers and Industrial Engineering*, 33:441–444, 1997.

[14] Heinz-Otto Peitgen, Hartmut Jurgens, and Dietman Saupe. *Chaos and Fractals New Frontiers of Science*. Springer-Verlag Inc., New York, 1992.

[15] Roger T. Stevens. *Fractal Programming in C*. M and T Books, Redwood City, CA, 1989.