

# SEMINOLE HIGH SCHOOL

## EXTENDED ESSAY

---

# The development of the fractal dimension and its applications

---

*Author:*

Samuel BRENNER

*Supervisor:*

Ms. Margaret LANGFIELD

August 2013

SEMINOLE HIGH SCHOOL

# *Abstract*

Faculty Name

Department or School Name

Extended Essay

**The development of the fractal dimension and its applications**

by Samuel BRENNER

---

add an  
abstract

# *Acknowledgements*

The acknowledgements and the people to thank go here, don't forget to include your project advisor...

# Contents

<b>Abstract</b>	<b>i</b>
<b>Acknowledgements</b>	<b>ii</b>
<b>List of Figures</b>	<b>iv</b>
<b>List of Tables</b>	<b>v</b>
<b>1 Fractals and their dimensions</b>	<b>1</b>
1.1 Why a fractal dimension is necessary . . . . .	1
1.1.1 Example of a fractal: the Koch curve . . . . .	1
Construction of the Koch curve. . . . .	1
1.1.2 Another example of a fractal: the middle-third Cantor set . . . . .	3
Construction of the middle-third Cantor set. . . . .	3
1.1.3 The ambiguity of nonfractal geometry . . . . .	4
1.2 What “dimension” means . . . . .	5
Dimension of the Koch curve. . . . .	6
1.3 Fractal dimensions . . . . .	6
1.3.1 Spaces, sets, and measures . . . . .	7
1.3.2 Box-counting dimension . . . . .	8
<b>2 Applications</b>	<b>11</b>
<b>3 Multifractals: a brief discussion and their applications</b>	<b>12</b>
<b>A Source Code of Box-Counting Module</b>	<b>13</b>
<b>B Source Code of Multifractal Spectrum Module</b>	<b>22</b>
<b>C Source Code of Iterative Function System Generator</b>	<b>27</b>
<b>Bibliography</b>	<b>30</b>

# List of Figures

1.1	Koch Curve . . . . .	2
1.2	Middle-third Cantor set . . . . .	3
1.3	Middle-fifteenth Cantor set . . . . .	5

# List of Tables

# Chapter 1

## Fractals and their dimensions

### 1.1 Why a fractal dimension is necessary

We introduce two fractal shapes and observe various properties that lead them to be called fractals. We then show how characterizations of their geometry without fractal dimensions are not useful in differentiating one from another.

#### 1.1.1 Example of a fractal: the Koch curve

We first discuss the Koch curve, one of the earliest fractals to be discovered. Shown in Fig. 1.1, the curve was first constructed from a line segment by a recursive procedure [1]. However, it will be more useful to describe its construction in the following manner.

**Construction of the Koch curve.** A curve  $E_1$  is constructed by taking some line segment  $E_0$  and dividing it into thirds. An equilateral triangle is constructed on the middle third of the segment  $E_0$ , such that the middle third of  $E_0$  is one of the triangle's sides. The middle third of  $E_0$  is then removed. Each successive iteration  $E_n$  of the curve's generation is performed as follows: Each line segment in the  $n$ th iteration of the curve, having length  $L_n$ , is replaced by a scaled-down version of  $E_1$  with length  $4L_n/3$ .

The Koch curve  $F$  is the result of an infinite number of applications of the iteration described above.

$$F = \lim_{n \rightarrow \infty} E_n \tag{1.1}$$

We note the most obvious properties of this fractal that separate it from a non-fractal shape.

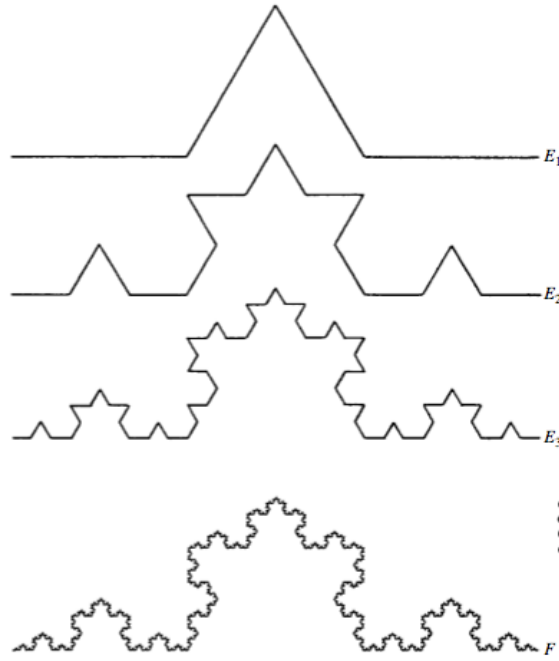


FIGURE 1.1: Iterations of the Koch curve, a simple fractal exhibiting many characteristics typical of fractals. The initial figure at the top is made more and more textured with each iteration  $E_n$ , becoming rougher and more fractal-like as it approaches the final Koch curve  $F$ . Image credit: [2].

**Observation 1.** *Its shape is defined recursively.* The Koch curve is a line segment on which an infinite number of the iterations described above is performed. This results in self-similarity at all scales: any section of the Koch curve contains an infinite number of repetitions of the original curve.

**Observation 2.** *The length of the Koch curve is infinite.*

*Proof.* Because of the recursive rule defining the Koch curve, the curve's length is increased by a factor of  $4/3$  with each of the infinite iterations. We express the length of the curve after the  $n$ th iteration as  $L_n$ .

$$L_n = \left(\frac{4}{3}\right)^n L_0. \quad (1.2)$$

The total length  $L_\infty$  can be expressed as an infinite geometric sequence with a common ratio of  $4/3$ :

$$L_\infty = \lim_{n \rightarrow \infty} L_n = \lim_{n \rightarrow \infty} \left(\frac{4}{3}\right)^n L_0. \quad (1.3)$$

Limits are linear with respect to multiplication by a constant, so we find:

$$\lim_{n \rightarrow \infty} L_n = L_\infty = L_0 \lim_{n \rightarrow \infty} \left(\frac{4}{3}\right)^n, \quad (1.4)$$



which clearly increases without bound.

$$\lim_{n \rightarrow \infty} L_n = \infty \quad (1.5)$$

The length of the Koch curve is thus infinite.  $\square$

**Observation 3.** *The Koch curve has no area.* Though we see that the additional length added with each iteration expands the fractal's size in the plane, the added segments have no thickness, so the area filled by the curve is thus zero.

### 1.1.2 Another example of a fractal: the middle-third Cantor set

We introduce a second fractal, the middle-third Cantor set, and show that it has drastically different properties from the Koch curve described above. It is shown in Fig. 1.2 and was first described in [3].



FIGURE 1.2: Iterations of the middle-third Cantor set, another simple fractal. The initial figure at the top is made more and more sparse with each iteration  $E_n$ , before finally approaching the middle-third Cantor set, a figure that has no length, as we show below.

**Construction of the middle-third Cantor set.** A curve  $E_1$  is constructed by taking some line segment  $E_0$  and removing the middle third. Each successive iteration  $E_n$  of the set's generation is performed by replacing each line segment in the  $n$ th iteration, having length  $L_n$ , with a scaled-down version of  $E_1$  of length  $L_n/3$ . The middle-third Cantor set  $F$  is the result of an infinite number of applications of the iteration described here, and is given by Eq. 1.1.

We again note properties of this fractal that differentiate it from a non-fractal shape.

**Observation 4.** *Its shape is defined recursively.* Just as with the Koch curve, this fractal is defined by a simple, recursive rule. This results in self-similarity at all scales: any portion of the set contains an infinite number of repetitions of the original set.

**Observation 5.** *Its length is zero.*

*Proof.* We show that the length of the Cantor set is zero in the same way that we showed that the length of the Koch curve is infinite. Because of the recursive rule defining the Cantor set, the curve's length is decreased by a factor of  $2/3$  with each of the infinite iterations. We express the length of the curve after the  $n$ th iteration as  $L_n$ .

$$L_n = \left(\frac{2}{3}\right)^n L_0. \quad (1.6)$$

The total length  $L_\infty$  can be expressed as an infinite geometric sequence with a common ratio of  $2/3$ :

$$L_\infty = \lim_{n \rightarrow \infty} L_n = \lim_{n \rightarrow \infty} \left(\frac{2}{3}\right)^n L_0. \quad (1.7)$$

As before,

$$\lim_{n \rightarrow \infty} L_n = L_\infty = L_0 \lim_{n \rightarrow \infty} \left(\frac{2}{3}\right)^n \quad (1.8)$$

which clearly tends toward zero.

$$\lim_{n \rightarrow \infty} L_n = 0. \quad (1.9)$$

The length of the Cantor set is thus zero.  $\square$

**Observation 6.** *Its area is zero.* The Cantor set consists of a line segment—something that has no thickness and thus fills no area—from which sections are removed. Because this process of removal does not add any thickness to the figure, the Cantor set must not have any thickness or area.

### 1.1.3 The ambiguity of nonfractal geometry

We have now introduced two fractals. The first, the Koch curve, has infinite length but no area. The second, the middle-third Cantor set, has no length and no area.

These characteristics do not, however, provide us much information about the fractals' shape: we can envisage many other fractals that, like the middle-third Cantor set, have neither length nor width but that also have entirely different appearances. In the middle-third Cantor set, we removed the middle  $1/3$  of every line segment to generate the next iteration. Suppose instead that we construct a middle-fifteenth Cantor set. We perform an identical procedure to generate this fractal, but, instead of removing the middle third of each line segment, we remove the middle fifteenth. That is, we divide the line segment into fifteen equal parts and remove the middle one. The construction of such a set is depicted in Fig. 1.3.

**Observation 7.** This middle-fifteenth Cantor set, though it has a different structure than the middle-third Cantor set, also has no length.

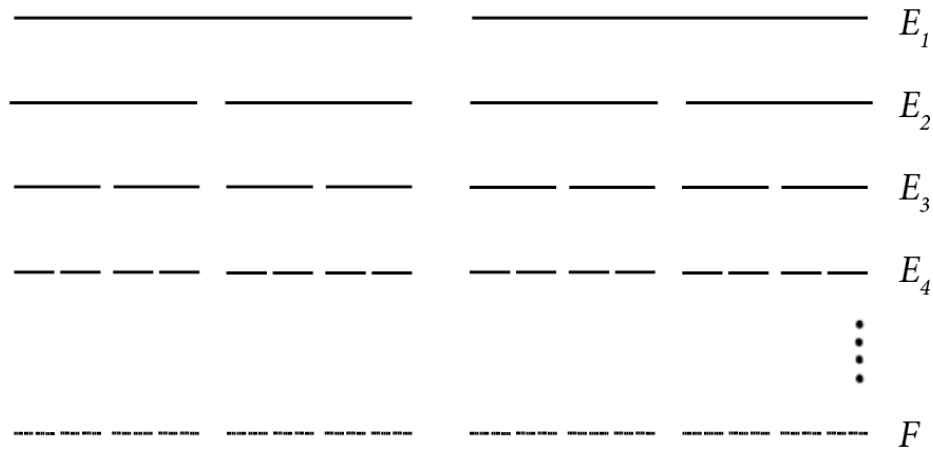


FIGURE 1.3: The middle-fifteenth Cantor set. This fractal is generated like middle-third Cantor set, except the middle fifteenth of each line segment is removed in every iteration. We see that the resulting fractal has a very different shape but show that it, like the middle-third Cantor set, has neither length nor area.

*Proof.* The length of the middle-fifteenth Cantor set can be expressed as an infinite geometric sequence, just as we did with the middle-third Cantor set. Here, though, the length is decreased by a factor of  $14/15$  with each iteration, so the length  $L_n$  of the set after  $n$  iterations is given by

$$L_n = \left(\frac{14}{15}\right)^n L_0. \quad (1.10)$$

The final length of the curve  $L_\infty$  is

$$L_\infty = \lim_{n \rightarrow \infty} L_n = \lim_{n \rightarrow \infty} \left(\frac{14}{15}\right)^n L_0, \quad (1.11)$$

which still tends to zero as  $n$  tends toward infinity. Hence the length of the middle-fifteenth Cantor set is also zero.  $\square$

Instead of classifying fractals by their length and area, then, we examine their “roughness”. This, as we will discuss below, leads to the creation of different *fractal dimensions* to classify the space-filling properties of these fractals.

## 1.2 What “dimension” means

Consider a line segment of length  $L_0$  and some number  $\epsilon > 1$ . Note that when the segment is scaled by  $\epsilon$  into smaller pieces of equal size, each segment produced has length  $L_0/\epsilon$ . Now we repeat our exploration with a square of side length  $L_0$ . Here, when the square is scaled by a factor  $\epsilon$ , the smaller squares produced have an area  $1/\epsilon^2$

the original. Again, when we repeat this procedure with a cube of side length  $L_0$  we find that the cubes produced all have volumes  $1/\epsilon^3$  the original. Hence, we generalize that the number of pieces  $n$  produced from scaling by a factor of  $\epsilon$  is given by

$$n = 1/\epsilon^D \quad (1.12)$$

where  $D$  is the dimension of the shape, and we can now solve for dimension:

$$D = \frac{\log n}{\log (1/\epsilon)} \quad (1.13)$$

**Dimension of the Koch curve.** We now return to the Koch curve. We recognize the first iteration  $E_1$  as the fundamental unit of all other iterations of the curve. That is, we can replace each of the four line segments that make up  $E_1$  with a scaled-down copy of the original  $E_1$  in order to reach the next iteration, and by repeating this process ad infinitum we can reach the final curve  $F$ . We can, however, view this process in the framework established in the example above, so that each iteration replaces one line segment with a length  $L_0$  with four others, each of which has a length of  $L_0/3$ . This is akin to the method described above: our fractal produces  $n = 4$  smaller pieces each time it is scaled, and each piece is  $1/\epsilon = 1/3$  the size of the original. Hence, we can apply the equation for dimension (Equation 1.13) to the Koch curve:

$$D = \frac{\log n}{\log (1/\epsilon)} = \frac{\log 4}{\log 3} \approx 1.26 \quad (1.14)$$

**Remark 1.** What does this dimension mean? Again, it seems intuitive that the dimension would be between one and two—this generalization of dimension should resemble the more familiar case of non-fractional dimensions, and the Koch curve is certainly greater than one-dimensional while not filling the entire plane as a two-dimensional shape would.

elaborate  
on this:  
why it  
should  
be be-  
tween  
one and  
two

### 1.3 Fractal dimensions

In the previous section we introduced the concept of dimension to quantify how the size of a figure is related to its scaling factor. We found that a figure in two dimensions, for example, is scaled into four parts when scaled by a factor of two. We call this the *similarity dimension*. Yet there are still other ways to quantify the scaling of a shape. We must remember:

“There is no one dimension that can only describe a fractal set; many different types of dimension exist, and all provide very different conceptions of a fractal’s properties” [2].

That is, dimension is just some quantity that describes the scaling relationship of a shape. We will use the understanding we develop here to understand box-counting dimension and Hausdorff dimension.

### 1.3.1 Spaces, sets, and measures

Before we venture further into classifying fractals by their dimension, we need the following definitions.

We use the word “set” to more precisely refer to what might be simply called a shape; every shape is just a set, possibly infinite, of points. As seen above, whereas non-fractal sets have well-defined, intuitively understood geometric properties such as length or area, fractal sets can be characterized by various concepts of dimension. Here, we will discuss *box-counting dimension* in the greatest detail.

**Definition 1.** An  $n$ -dimensional space  $\mathbb{R}^n$  is the set of all points that can be identified by an ordered  $n$ -ple of real numbers. Hence, for example, any point  $\alpha$  in  $\mathbb{R}^5$  can be identified by the ordered 5-ple  $\alpha = (\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5)$  where  $\alpha_1 \dots \alpha_5 \in \mathbb{R}$ .

We will now work by analogy to discuss the concept of a *measure*<sup>1</sup>. Consider a geographical map of some landmass, with a function  $\mu(\mathcal{S})$  that denotes the volume of groundwater below any subset  $\mathcal{S}$  of the landmass. We note four intuitive properties of this measure  $\mu$ :

1. The measure of any subset of the space—or the volume of groundwater under any plot of land on the landmass—gives a numerical indication of the plot of land’s magnitude. That is, it describes the plot’s size when measured by groundwater content.
2. There should be no water under a plot of land with no size.
3. If a plot of land  $A$  contains another plot of land  $B$ , there should be less or as much groundwater under  $B$  as there is under  $A$ .

---

<sup>1</sup>Analogy adapted from [4]

4. If we find the volume of groundwater under a plot of land on the landmass, we should get the same value or less than when we split the plot up into many pieces with overlap allowed and add up the volume of groundwater found under each piece.

We now formalize these properties below.

**Definition 2.** A measure  $\mu$  is a function on a space  $\mathbb{R}^n$  that assigns a positive number to each subset  $\mathcal{S}$  of  $\mathbb{R}^n$  such that:

- Because  $\mu$  should characterize a set by its size in some sense, it is meaningless to have a nonzero size for any set with no elements. This is reflected in Property 2 above.

$$\mu(\emptyset) = 0 \tag{1.15}$$

- The size of a set should be smaller than (or equal to, if  $A = B$ , i.e.,  $B$  is not a proper subset of  $A$ ) the size of a set that contains the first set and other elements too. This is reflected in Property 3 above.

$$\mu(B) \leq \mu(A) \quad \text{if} \quad B \subset A. \tag{1.16}$$

- The measure of a union of subsets should never be larger than the sum of the measures of the individual subsets:

$$\mu\left(\bigcup_{i=1}^{\infty} A_i\right) \leq \sum_{i=1}^{\infty} \mu(A_i) \tag{1.17}$$

Allowing for the measure of the union to be smaller than the sum of the individual measures accounts for overlap between the subsets; if, for all pairs of subsets  $A_1$  and  $A_2$ ,  $A_1 \cap A_2 = \emptyset$ , the measure of the union should be equal to the sum of the individual measures. In short, the measure should be distributive over addition. This is reflected in Property 4 above.

We call the space on which the measure resides the *support* of the measure.

### 1.3.2 Box-counting dimension

Recall that in the example of the Koch curve above, we failed to usefully classify the shape by non-fractal means. Though we could see that the figure filled a space, we lacked language to discuss precisely how it filled that space; we could not quantify its

“roughness”. We introduce the box-counting dimension as an analogy to the Euclidean dimension in order to quantify how a fractal set fills a space.

We now seek to formalize our understanding of dimension. Furthermore, we wish to formalize this understanding in a way that allows us to discuss fractal objects that are not strictly self-similar<sup>2</sup>. For this we introduce a specific dimension, the *box-counting dimension*. In addition, the box-counting dimension is computationally much more simple than other fractal dimensions, such as the Hausdorff dimension [2].

The box-counting dimension gives an analogue of the dimension defined by Equation 1.13: the fractal set is covered by a number of cubes<sup>3</sup> with side length  $\epsilon$ . We extend the number of replacements  $n$  from Equation 1.13 to take the smallest number of cubes that can cover the fractal set. The box-counting dimension is given by the limit of this extension of Equation 1.13 as the side  $\epsilon$  approaches zero.

**Definition 3.** The box-counting dimension of a fractal set  $F$ ,  $\dim_B F$ , is given by:

$$\dim_B F = \lim_{\epsilon \rightarrow 0} \frac{\log N_\epsilon(F)}{-\log \epsilon} \quad (1.18)$$

where  $N_\epsilon(F)$  is the number of cubes of side-length  $\epsilon$  that cover the fractal set  $F$ .

**Example 1.** We can apply numerical techniques to verify the agreement of this definition with that established in Section ??. A computational module to calculate the box-counting dimension is included in A and is here applied to a Koch curve of size ??? pixels end-to-end. \*\*\*\*include graphics of curve and results, compare dimensions and error\*\*\*\*

move  
these to  
Ch. 2

**Example 2.** Same thing, Sierpinski pyramid, perhaps?

<sup>2</sup>As mentioned in Section ??, fractals are not as concretely defined as one might hope, and we tend only to classify a shape as a fractal when it possesses a reasonably large number of the fractal properties identified in Section ??—when a shape has sufficient “roughness” that the language of fractal geometry becomes more useful in describing its characteristics.

<sup>3</sup>A “cube” here is not strictly three-dimensional; the shape has analogues in all other dimensions as well. A cube in  $\mathbb{R}^n$  is more generally a figure with  $2n$  sides of equal size where each side is orthogonal to the others it touches, if the others exist, and each side is itself a cube in  $\mathbb{R}^{n-1}$ . Thus a cube in  $\mathbb{R}^1$  is a line segment; a cube in  $\mathbb{R}^2$  is a square; and a cube in  $\mathbb{R}^4$  is a tesseract.

# Todo list

add an abstract . . . . .	i
elaborate on this: why it should be between one and two . . . . .	6
move these to Ch. 2 . . . . .	9



## Chapter 2

# Applications

## Chapter 3

# Multifractals: a brief discussion and their applications

# Appendix A

## Source Code of Box-Counting Module

---

```
/** Fractal analysis module utilizing boxcounting to determine
    the fractal dimension of a shape in the input text/binary file.

    Prints to terminal:
    - the results at each level of analysis
    - the overall dimension as determined by least-squares regression
    - an R^2 value

    @author Samuel Brenner
    @version July 11, 2013

    **/
```

```
#include <fstream>
#include <string>
#include <sstream>
#include <math.h>
#include <iostream>
#include "regressionModule.h"
#include "dataReader.h"
#include "interpolation.h"

using namespace std;

const char* fileName = "c_hdf5_plt_cnt_1000.txt";
//const char* fileName = "first(phillip's).txt";

const char* outFileName = "logN-vs-logE.txt";

void printArray(int** arrayIn, int HEIGHT, int WIDTH){
    cout << endl;
```

```

    for(int i = 0; i < HEIGHT; i++){
        for(int j = 0; j < WIDTH; j++){
            printf("%3d", arrayIn[i][j]);
        }
        printf("\n");
    }
}

void printArray(double** arrayIn, double HEIGHT, double WIDTH){
    cout << endl;
    printf("%7s%7s\n", "level", "log(n)");
    for(int i = 0; i < HEIGHT; i++){
        for(int j = 0; j < WIDTH; j++){
            printf("%7.3f ", arrayIn[i][j]);
        }
        printf("\n");
    }
}

void printToFile(double** arrayIn, int HEIGHT, int WIDTH, double slope, double
    ↪ yint){
    FILE * pFile;
    pFile = fopen(outFileName, "w");
    fprintf(pFile, "slope: %f\nintercept: %f\n", slope, yint);
    for(int i = HEIGHT - 1; i >= 0; i--){
        for(int j = 0; j < WIDTH; j++){
            fprintf(pFile, "%-2.6f ", arrayIn[i][j]);
        }
        fprintf(pFile, "\n");
    }
    fclose(pFile);
}

/**
    Returns the log base 2 of the number of cells filled in a given level
    of fractal analysis.
    @param arrayIn is the 2-3-D array of integers that contains the data
    @param HEIGHT
    @param WIDTH
    @param DPETH
    @param level is the level of analysis to be performed. Level 0, for example,
    examines only the individual data points, whereas at level 1 they are merged
    into boxes of side length  $2^1$ , and for level  $l$  size  $2^l$ .
    @return log base 2 of the number of cells filled in a given level.

    */
double boxCounting(int*** arrayIn, int HEIGHT, int WIDTH, int DEPTH, int level){
    int numberFilled = 0; //number of boxes with an element of the figure
    int boxDimension = (int) pow(2, level);
    int boxDimensionZ;

    if(DEPTH != 1){
        boxDimensionZ = boxDimension;
    }
    else{

```

```

    boxDimensionZ = 1;
}
//iterates through all boxes
//cout << "here!" << endl;
for(int i = 0; i < HEIGHT; i += boxDimension){
    for(int j = 0; j < WIDTH; j += boxDimension){
        for(int k = 0; k < DEPTH; k += boxDimensionZ){
            int boxSum = 0;

            //sums each box
            for(int boxSumX = 0; boxSumX < boxDimension; boxSumX++){
                for(int boxSumY = 0; boxSumY < boxDimension; boxSumY++){
                    for(int boxSumZ = 0; boxSumZ < boxDimensionZ; boxSumZ++){
                        //if(boxSumX + i >= HEIGHT || boxSumY + j >= WIDTH || boxSumZ + k
➔ >= DEPTH){ This is to deal with dimensions that aren't
                        // cout << endl << "Dimensions must be powers of two!" << endl;
➔ powers of two.
                        // boxSum += 0;
                        //}
                        //else{
                            boxSum += arrayIn[boxSumX + i][boxSumY + j][boxSumZ + k];
                        //}
                    }
                }
            }

            if(boxSum > 0){
                numberFilled++;
            }

        }
    }

    printf("%s%d\n%s%d\n%s%d\n\n", "level: ", level,
    "--box dimension: ", boxDimension, "--filled boxes: ", numberFilled);

    return log2(numberFilled);
}

int main () {
    int HEIGHT, WIDTH, DEPTH;
    double*** elements;
    bool haveZeros = false;
    double arraySum;
    int*** elementsInterpolated;

    elements = dataReaderASCII<double>(fileName, HEIGHT, WIDTH, DEPTH, haveZeros,
➔ arraySum); //change to dataReaderASCII to read in text files
    elementsInterpolated = interpolate(elements, HEIGHT, WIDTH, DEPTH, 0.5);
    //printToFile(elementsInterpolated, HEIGHT, WIDTH);

    int LOWESTLEVEL = 0;

    //initialize out-array

```

```

int largestDimension = fmin(HEIGHT, WIDTH);
if(DEPTH > 1){
    largestDimension = fmin(largestDimension, DEPTH);
}
int outArrayLength = int(log2(largestDimension) - LOWESTLEVEL + 1);
double** outDataArray = new double* [outArrayLength];

//printf("\n\n%d\n\n", outArrayLength);

for(int i = 0; i < outArrayLength; i++){
    outDataArray[i] = new double[2];
}

for(int k = 0; k < outArrayLength; k++)
{
    outDataArray[k][0] = outArrayLength - 1 - (k + LOWESTLEVEL);
    outDataArray[k][1] = boxCounting(elementsInterpolated, HEIGHT, WIDTH, DEPTH,
    ↪ k + LOWESTLEVEL);
}

printArray(outDataArray, outArrayLength, 2);

double* regressionArray = new double[3];

slope(outDataArray, outArrayLength, regressionArray);

printf("\n\n%s%s\n\n%.5f\n\n%.13f\n\n", "Curve: ", fileName, "Dimension: ",
    ↪ regressionArray[0], "R^2: ", regressionArray[1]);

printToFile(outDataArray, outArrayLength, 2, regressionArray[0],
    ↪ regressionArray[2]);

delete[] elements;
delete[] elementsInterpolated;
delete[] outDataArray;
delete[] regressionArray;

return 0;
}

/**
Module to interpolate the isocontour at some pre-specified value inside a 2- or
    ↪ 3-D array.

Implemented in boxCounter3D.cpp:
Fractal analysis module utilizing boxcounting to determine
the fractal dimension of a shape in the input text/binary file.

Prints to terminal:
- the results at each level of analysis
- the overall dimension as determined by least-squares regression
- an R^2 value

@author Samuel Brenner
@version July 11, 2013

```

*@author Samuel Brenner*  
*@version July 11, 2013*

```

**/

#ifndef interpolation_h
#define interpolation_h

int*** interpolate(double*** arrayIn, int HEIGHT, int WIDTH, int DEPTH, double
    ↪ valueToFind){
    //Declare and initialize arrayOut. If any values in the arrayIn happen to be
    ↪ the valueToFind,
    //we'll initialize the corresponding arrayOut to a 1; otherwise it'll be a 0.
    int*** arrayOut = new int**[HEIGHT];
    for(int i = 0; i < HEIGHT; i++){
        arrayOut[i] = new int*[WIDTH];
        for(int j = 0; j < WIDTH; j++){
            arrayOut[i][j] = new int[DEPTH];
            for(int k = 0; k < DEPTH; k++){
                //performs a preliminary check so that we don't miss any values that are
                ↪ exactly == valueToFind
                if(arrayIn[i][j][k] == valueToFind){
                    arrayOut[i][j][k] = 1;
                }
                else{
                    arrayOut[i][j][k] = 0;
                }
            }
        }
    }
}

//Makes the interpolater able to handle planar data
int startingDepthIndex = 0;
if(DEPTH != 1){
    startingDepthIndex = 1;
}
else{
    DEPTH++;
}

/*
    Iterates over all entries in the array that aren't on the edge of the array,
    ↪ unless the array is 2D: for 2D
    arrays the program ignores the edge condition in the third dimension.

    For each entry analyzed, we check to see if the value is lower than
    ↪ valueToFind. If so, the neighbors on all
    six sides (four if planar data) are checked against it to determine if there
    ↪ is some crossing over valueToFind
    in between the two. Then we determine which cell should contain that crossing
    ↪ and assign it a 1 in the
    corresponding cell of the outArray.
*/
for(int i = 1; i < HEIGHT - 1; i++){

```

```

for(int j = 1; j < WIDTH -1; j++){
    for(int k = startingDepthIndex; k < DEPTH - 1; k++){
        if(arrayIn[i][j][k] < valueToFind){
            if(arrayIn[i + 1][j][k] > valueToFind){
                if(int((valueToFind - arrayIn[i][j][k]) / (arrayIn[i + 1][j][k] -
↪ arrayIn[i][j][k])) == 0){
                    arrayOut[i][j][k] = 1;
                }
            }
            else{
                arrayOut[i + 1][j][k] = 1;
            }
        }
        if(arrayIn[i][j + 1][k] > valueToFind){
            if(int((valueToFind - arrayIn[i][j][k]) / (arrayIn[i][j + 1][k] -
↪ arrayIn[i][j][k])) == 0){
                arrayOut[i][j][k] = 1;
            }
            else{
                arrayOut[i][j + 1][k] = 1;
            }
        }
        if(arrayIn[i][j - 1][k] > valueToFind){
            if(int((valueToFind - arrayIn[i][j][k]) / (arrayIn[i][j - 1][k] -
↪ arrayIn[i][j][k])) == 0){
                arrayOut[i][j][k] = 1;
            }
            else{
                arrayOut[i][j - 1][k] = 1;
            }
        }
        if(arrayIn[i - 1][j][k] > valueToFind){
            if(int((valueToFind - arrayIn[i][j][k]) / (arrayIn[i - 1][j][k] -
↪ arrayIn[i][j][k])) == 0){
                arrayOut[i][j][k] = 1;
            }
            else{
                arrayOut[i - 1][j][k] = 1;
            }
        }
        if(arrayIn[i][j][k + 1] > valueToFind){
            if(int((valueToFind - arrayIn[i][j][k]) / (arrayIn[i][j][k + 1] -
↪ arrayIn[i][j][k])) == 0){
                arrayOut[i][j][k] = 1;
            }
            else{
                arrayOut[i][j][k + 1] = 1;
            }
        }
        if(arrayIn[i][j][k - 1] > valueToFind){
            if(int((valueToFind - arrayIn[i][j][k]) / (arrayIn[i][j][k - 1] -
↪ arrayIn[i][j][k])) == 0){
                arrayOut[i][j][k] = 1;
            }
            else{
                arrayOut[i][j][k - 1] = 1;
            }
        }
    }
}

```



```

        }
    }

    }
}

}

return arrayOut;
}

#endif

/**
    Module to run regression for boxcounting.

    Changes a double[3] such that regressionArray[0] = slope,
    regressionArray[1] = R^2, and regressionArray[2] = y-int.

    Implemented in boxCounter3D.cpp:
    Fractal analysis module utilizing boxcounting to determine
    the fractal dimension of a shape in the input text/binary file.

    Prints to terminal:
    - the results at each level of analysis
    - the overall dimension as determined by least-squares regression
    - an R^2 value

    @author Samuel Brenner
    @version July 11, 2013

    @author Samuel Brenner
    @version July 11, 2013

**/

#include "regressionModule.h"

#include <math.h>

double stdev(double* arrayIn, double arrayMean, int arrayLength);

double corrCoeff(double* arrayX, double* arrayY, double meanX,
    double meanY, double stdevX, double stdevY, int length);

double mean(double* arrayIn, int arrayLength);

void slope(double** arrayIn, int arrayLength, double* regressionArray){

    double* arrayX = new double[3];

    for(int i = 0; i < 3; i++){
        arrayX[i] = arrayIn[i][0];
    }

```

```

double* arrayY = new double[3];
for(int i = 0; i < 3; i++){
    arrayY[i] = arrayIn[i][1];
}

double meanX = mean(arrayX, 3);
double meanY = mean(arrayY, 3);

double stdevX = stdev(arrayX, meanX, 3);
double stdevY = stdev(arrayY, meanY, 3);

double r = corrCoeff(arrayX, arrayY, meanX, meanY, stdevX, stdevY, 3);

double slope = r
    * stdevY / stdevX;

delete[] arrayX;
delete[] arrayY;

regressionArray[0] = slope;
regressionArray[1] = pow(r, 2);
regressionArray[2] = meanY - slope * meanX;
}

double stdev(double* arrayIn, double arrayMean, int arrayLength){

    double squaresSum = 0;

    for(int i = 0; i < arrayLength; i++){
        squaresSum += pow(arrayIn[i] - arrayMean, 2);
    }

    return sqrt(squaresSum / (arrayLength - 1));
}

double corrCoeff(double* arrayX, double* arrayY, double meanX,
    double meanY, double stdevX, double stdevY, int length){

    double sum = 0;

    for(int i = 0; i < length; i++){
        sum += (arrayX[i] - meanX) * (arrayY[i] - meanY);
    }

    return sum / (stdevX * stdevY * (length - 1));
}

double mean(double* arrayIn, int arrayLength){
    double sum = 0;

    for(int i = 0; i < arrayLength; i++){
        sum += arrayIn[i];
    }

```

---

```
    return sum / arrayLength;  
}
```

---

## Appendix B

# Source Code of Multifractal Spectrum Module

---

```
/**
    Multifractal analysis module utilizing boxcounting to determine
    the multifractal spectrum of a measure in the input text/binary file.

    Prints the data that forms the multifractal spectrum to a text file
    that can later be analyzed in Matlab or another visualization program.

    The algorithm used is described in:
        A. Chhabra and R. V. Jensen, Phys. Rev. Lett. 62, 1330 (1989).

    @author Samuel Brenner
    @version July 11, 2013
**/
#include <fstream>
#include <string>
#include <sstream>
#include <math.h>
#include <iostream>
#include <algorithm>
#include "dataReader.h"
#include "dataCorrection.h"

using namespace std;

const char* fileName = "multifractal.txt"; //filename of input data--can also be
    ↪ binary

double qMin = -10; //minimum and maximum values of the moment used in
double qMax = 10;
double qIncrement = 1.0; //the step used between each value of q tested.
```

```

void printArray(int** arrayIn, int HEIGHT, int WIDTH){
    cout << endl;
    for(int i = 0; i < HEIGHT; i++){
        for(int j = 0; j < WIDTH; j++){
            printf("%7d", arrayIn[i][j]);
        }
        printf("\n");
    }
}

void printArray(double** arrayIn, int arrayHeight, int WIDTH){
    cout << endl;
    for(int i = 0; i < arrayHeight; i++){
        for(int j = 0; j < WIDTH; j++){
            printf("%3.3f ", arrayIn[i][j]);
        }
        printf("\n");
    }
}

void printToFile(double** arrayIn, int level, int HEIGHT){
    FILE * pFile;
    char* fileOutName = new char[20];
    sprintf(fileOutName, "%s%d%s", "falphaout/falpha_level_", level, ".txt");
    pFile = fopen(fileOutName, "w");
    for(int i = 0; i < HEIGHT; i++){
        fprintf(pFile, "%.8f %.8f", arrayIn[i][0], arrayIn[i][1]);
        fprintf(pFile, "\n");
    }
    fclose(pFile);
    delete[] fileOutName;
}

/**
    Returns in a double** the data points that make up the plot of  $f(\alpha)$  vs.
    ↪  $\alpha$ .
    The analysis is performed using the method of moments described in Chhabra and
    ↪ Jensen,
    1989 (Phys. Rev. Lett.).
    @param arrayIn is the 2-/3-D array of doubles that contains the data
    @param HEIGHT
    @param WIDTH
    @param DPETH
    @param level is the level of analysis to be performed. Level 0, for example,
    examines only the individual data points, whereas at level 1 they are merged
    into boxes of side length  $2^1$ , and for level  $l$  size  $2^l$ .
    @return  $\alpha$  vs.  $f(\alpha)$ 
**/
double** boxCounting(double*** arrayIn, int HEIGHT, int WIDTH, int DEPTH, int
    ↪ level){
    int boxDimension = (int) pow(2, level); //side length of the boxes that we'll
    ↪ use to coarse-grain the data, in pixels.
    int boxDimensionZ;

    if(DEPTH != 1){

```

```

    boxDimensionZ = boxDimension;
}
else{
    boxDimensionZ = 1;
}

int nDataPts = HEIGHT * WIDTH * DEPTH / pow(boxDimension, 2) / boxDimensionZ;
int falphaSize = qMax - qMin + 1;

//creates array to hold falpha
double** falpha = new double*[falphaSize];
for(int i = 0; i < qMax - qMin + 1; i++){
    falpha[i] = new double[2];
}

/** iterates over all values of q to be tested, where q is the "microscope for
    ↪ exploring different regions of the measure",
    an exaggerating exponent that gives us the qth moment of the measure. We
    ↪ parametrize the relationship f(alpha) vs. alpha
    in terms of q to find functions f(q) and alpha(q). These values are then
    ↪ added to the falpha out-array.
**/
for(double q = qMin; q <= qMax; q += qIncrement){
    double* probability = new double[nDataPts]; //an array of all the non-
    ↪ normalized measures taken in.
    double sumOfProbabilitiesQthMoment = 0.0;
    int arrayCounter = 0;
    double fOfQ = 0.0;
    double alphaOfQ = 0.0;

    //iterates through all boxes
    for(int i = 0; i < HEIGHT; i += boxDimension){
        for(int j = 0; j < WIDTH; j += boxDimension){
            for(int k = 0; k < DEPTH; k += boxDimensionZ){
                double boxSum = 0.0;

                //sums each box
                for(int boxSumX = 0; boxSumX < boxDimension; boxSumX++){
                    for(int boxSumY = 0; boxSumY < boxDimension; boxSumY++){
                        for(int boxSumZ = 0; boxSumZ < boxDimensionZ; boxSumZ++){

                            //This is to deal with dimensions that aren't powers of two.
                            //if(boxSumX + i >= HEIGHT || boxSumY + j >= WIDTH || boxSumZ +
                            ↪ k >= DEPTH){
                                // cout << endl << "Dimensions must be powers of two!" << endl
                                ↪ ;

                                // boxSum += 0;
                                //}
                                //else{
                                    boxSum += arrayIn[boxSumX + i][boxSumY + j][boxSumZ + k];
                                //}
                            }
                        }
                    }
                }
                probability[arrayCounter] = boxSum;

```

```

        arrayCounter++;
    }
}

for(int i = 0; i < nDataPts; i++){
    sumOfProbabilitiesQthMoment += pow(probability[i], q); //the denominator in
    ↪ eqn. 6 of Chhabra and Jensen
}

double log2size = log2(double(boxDimension) / HEIGHT);

for(int i = 0; i < nDataPts; i++){
    double normalizedMeasure = pow(probability[i], q) /
    ↪ sumOfProbabilitiesQthMoment; //eqn. 6 of Chhabra and Jensen
    //cout << log2(normalizedMeasure) << endl;
    fOfQ += normalizedMeasure * log2(normalizedMeasure); //the numerator of eqn
    ↪ . 7 of Chhabra and Jensen
    alphaOfQ += normalizedMeasure * log2(probability[i]); //the numerator of
    ↪ eqn. 8 of Chhabra and Jensen
}

falpha[int(q - qMin)][0] = alphaOfQ / log2size; //the final divisions in each
    ↪ equation
falpha[int(q - qMin)][1] = fOfQ / log2size;
delete[] probability;
}

return falpha;
}

int main () {
    int HEIGHT, WIDTH, DEPTH;
    double*** elements;
    bool haveZeros = false;
    double arraySum;

    elements = dataReaderASCII<double>(fileName, HEIGHT, WIDTH, DEPTH, haveZeros,
    ↪ arraySum);
    //elements = dataReaderBinary<double>(fileName, HEIGHT, WIDTH, DEPTH,
    ↪ haveZeros, arraySum); //for reading in binary data

    /**
     * Corrects the data by removing zeros (does so by adding one to every value
    ↪ in the array)
     * and normalizing the array so that the sum of all the elements is one.
    **/
    dataCorrection<double>(elements, HEIGHT, WIDTH, DEPTH, haveZeros, arraySum);

    int LOWESTLEVEL = 0; //the lowest level of coarse-graining, where level == 0
    ↪ examines each individual pixel as its own box.
        //level = log2(box's side length) so that side length = 1 when
    ↪ level == 0.

```

```
//iterates over all levels to be tested, going from LOWESTLEVEL to the highest  
↪ possible level permitted by the arrayIn size.  
for(int k = 0; k < log2(HEIGHT) - LOWESTLEVEL; k++){  
    cout << endl << "Level: " << k + LOWESTLEVEL << endl;  
    double** falpha = boxCounting(elements, HEIGHT, WIDTH, DEPTH, k + LOWESTLEVEL  
↪ );  
    printArray(falpha, qMax - qMin + 1, 2);  
    printToFile(falpha, k + LOWESTLEVEL, qMax - qMin + 1); //This spectrum output  
↪ can then be analyzed in Matlab or another visualization program.  
}  
  
//delete[] buffer;  
delete[] elements;  
return 0;  
}
```

---



## Appendix C

# Source Code of Iterative Function System Generator

---

```
//Program to generate fractals in text.
#include <fstream>
#include <string>
#include <sstream>
#include <math.h>
#include <iostream>

using namespace std;

int HEIGHT = pow(2, 9);
int WIDTH = HEIGHT;
//double normalizer = 1;//double(pow(5.0, log2(HEIGHT)));

struct point{
    double probability;
    int value;
};

void assignProbabilities(point** fractalArray){
    double fractalSum = 0.0;
    for(int levelDivision = HEIGHT; levelDivision > 1; levelDivision /= 2){
        for(int i = 0; i < HEIGHT; i++){
            for(int j = 0; j < WIDTH; j++){

                if((i % levelDivision >= (levelDivision / 2)) && (j % levelDivision >= (
↪ levelDivision / 2))){
                    fractalArray[i][j].probability *= 2.0 / 5.0;
                }
                else{
                    fractalArray[i][j].probability *= 1.0 / 5.0;
                }

                fractalSum += fractalArray[i][j].probability;
            }
        }
    }
}
```

```

    }
}
double fractalSumNew = 0.0;

//divide each element by a certain factor so that the total probability is
    ↪ still one.

//for(int i = 0; i < HEIGHT; i++){
    //for(int j = 0; j < WIDTH; j++){
        // fractalArray[i][j].probability /= normalizer;
        // fractalSumNew += fractalArray[i][j].probability;
    // }
//}
//cout << normalizer << endl;
//cout << fractalSumNew << endl;
}

void printToTerminal(point** arrayIn){
    cout << endl;
    for(int i = 0; i < HEIGHT; i++){
        for(int j = 0; j < WIDTH; j++){
            printf("%-.8f ", arrayIn[i][j].probability);
        }
        printf("\n\n");
    }
}

void printToFile(point** arrayIn){
    FILE * pFile;
    pFile = fopen("multifractal.txt", "w");
    fprintf(pFile, "%d\n%d\n1\n", HEIGHT, WIDTH);
    for(int i = 0; i < HEIGHT; i++){
        for(int j = 0; j < WIDTH; j++){
            fprintf(pFile, "%-.10f ", arrayIn[i][j].probability);
        }
        fprintf(pFile, "\n");
    }
    fclose(pFile);
}

int main(){
    //initialize array
    point** fractalArray = new point*[HEIGHT];
    for(int i = 0; i < HEIGHT; i++){
        fractalArray[i] = new point[WIDTH];
    }

    //fill array with zeroes
    for(int i = 0; i < HEIGHT; i++){
        for(int j = 0; j < WIDTH; j++){
            fractalArray[i][j].probability = 1;

```

```
    }  
}  
  
    //fill array with fractal shape  
    assignProbabilities(fractalArray);  
  
    //printToTerminal(fractalArray);  
    printToFile(fractalArray);  
  
  
  
    delete[] fractalArray;  
    return 0;  
}
```

---

# Bibliography

- [1] H. von. Koch. Sur une courbe continue sans tangente, obtenue par une construction gomtrique lmentaire. *Archiv fr Matemat., Astron. och Fys.*, 1:681–702, 1904.
- [2] K. Falconer. *Fractal Geometry: Mathematical Foundations and Applications*. Wiley, 2003.
- [3] Henry J.S. Smith. On the integration of discontinuous functions. *Proceedings of the London Mathematical Society, Series 1*, 6:140–153, 1874.
- [4] *Chaos and Fractals*, chapter Appendix B. Springer-Verlag, 1992.