Seminole High School

Extended Essay

The development of the fractal dimension and its applications

Author:
Samuel Brenner

Supervisor:
Ms. Margaret Langfield

August 2013

Abstract

Extended Essay

The development of the fractal dimension and its applications

by Samuel Brenner

Many physical phenomena cannot be characterized by the idealizations of Euclidean geometry alone; they exhibit "roughness" of morphology. That is, they have a detailed structure at any arbitrarily small size scale. We can quantify this roughness with concepts of fractal dimension, which—just like the Euclidean dimension—tell how some figure fills a space. We explore the develpoment and utility of this concept, investigating three fractal dimensions: the similarity dimension and the box-counting dimension. We then describe the implementation of a computer algorithm to calculate the box-counting dimension of various fractals. Finally, we examine the application of fractal dimension to select physical and economic phenomena.

Acknowledgements

I would like to thank my advisor, Ms. Langfield, for beating my first drafts into a bloody pulp and showing me how I could turn a disorganized mess into a piece of writing that I'm beginning to be proud of. I would also like to thank Prof. Tomasz Plewa and Tim Handy at the Department of Scientific Computing at Florida State University for giving me the impetus to study fractal and multifractal geometry and implement my knowledge computationally.

Contents

List of Figures

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 nth 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.

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

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

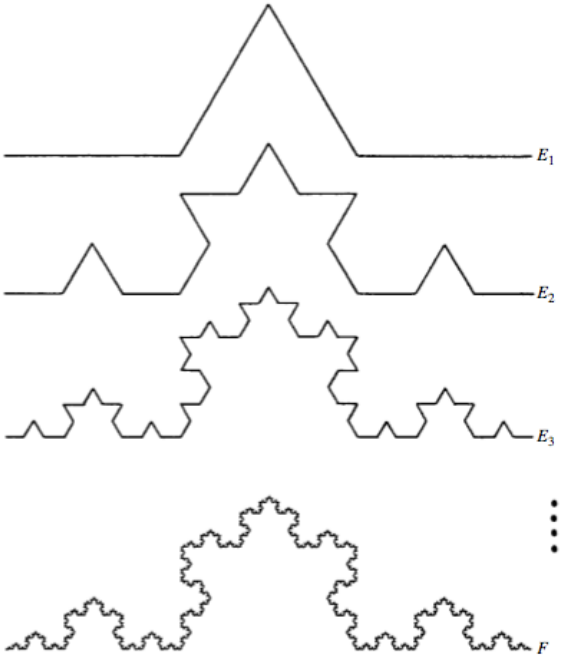

FIGURE 1.1: Iterations of the Koch curve, a simple fractal exhibiting many characteristics typical of fractals. The initial figure at the top is made more and more textured with each iteration E_n, becoming rougher and more fractal-like as it approaches the final Koch curve F. Image credit: [2].

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 nth iteration as L_n.

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

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

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

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

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

which clearly increases without bound.

$$\lim_{n\to\infty} L_n = \infty \tag{1.5}$$

The length of the Koch curve is thus infinite. □

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].

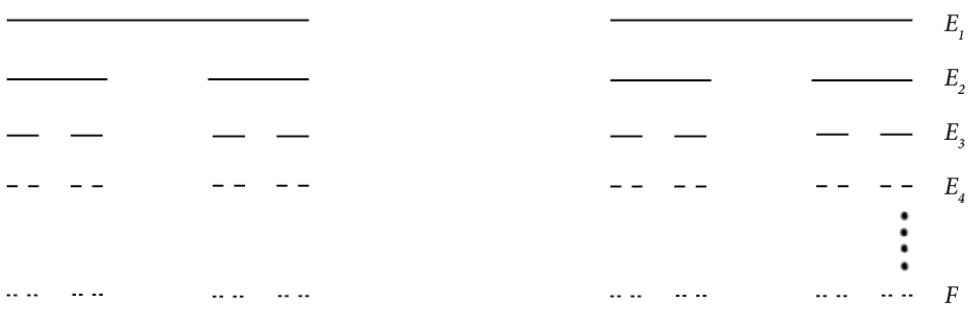

FIGURE 1.2: Iterations of the middle-third Cantor set, another simple fractal. The initial figure at the top is made more and more sparse with each iteration E_n, before finally approaching the middle-third Cantor set, a figure that has no length, as we show below.

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 nth 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 nth iteration as L_n.

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

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

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

As before,

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

which clearly tends toward zero.

$$\lim_{n\to\infty} L_n = 0. \tag{1.9}$$

The length of the Cantor set is thus zero. □

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.

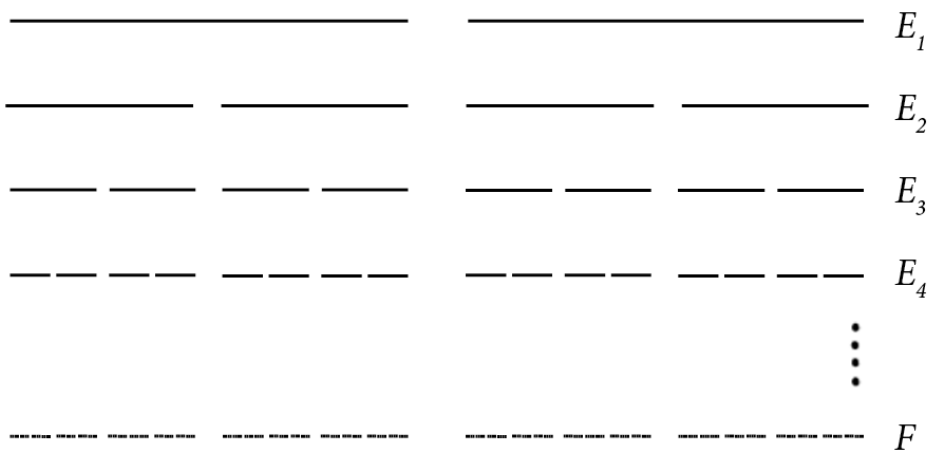

FIGURE 1.3: The middle-fifteenth Cantor set. This fractal is generated like middle-third Cantor set, except the middle fifteenth of each line segment is removed in every iteration. We see that the resulting fractal has a very different shape but show that it, like the middle-third Cantor set, has neither 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

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

The final length of the curve L_∞ is

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

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

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: an exploration of similarity dimension

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

when the square is scaled by a factor ϵ, 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 ϵ is given by

$$n = 1/\epsilon^D \tag{1.12}$$

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

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

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:

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

Remark 1. What does this dimension mean? Again, it seems intuitive that the dimension would be between one and two. If the dimension were two or greater, the curve would completely fill the plane in which it lies. Likewise, if the dimension were one or less, the curve would necessarily be broken into many discontinuous segments. This generalization of dimension resembles the more familiar case of non-fractional, Euclidean dimensions.

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. This is only one of many ways that we can 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 call the quantity described in the previous section the *similarity dimension*. We will use the understanding we develop here to understand box-counting dimension and similarity dimension.

1.3.1 Spaces and sets

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 α in $\mathbb{R}^5$ can be identified by the ordered 5-ple $\alpha = (\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5)$ where $\alpha_1 ... \alpha_5 \in \mathbb{R}$.

A "cube" is not strictly three-dimensional; the shape has analogues in all other dimensions as well.

Definition 2. 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.

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[1]. 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. To find it, the fractal set is covered by a number of cubes with side length ϵ. 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 ϵ approaches zero.

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

$$\dim_B F = \lim_{\epsilon \to 0} \frac{\log N_\epsilon(F)}{-\log \epsilon} \tag{1.15}$$

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

Note that the argument of this limit is almost identical to Eq. 1.13. If we make the substitution $-\log \epsilon = \log(1/\epsilon)$, we see

$$\dim_B F = \lim_{\epsilon \to 0} \frac{\log N_\epsilon(F)}{\log(1/\epsilon)} \tag{1.16}$$

which further accentuates the analogues between the two definitions of dimension. Instead of the replacement length n used in the numerator of the similarity dimesnion, we have here the number of boxes needed to cover the fractal set. And, here, the ϵ represents the side length of the box used instead of the scaling length used in the similarity dimension.

It is simple, computationally, to discretize the cubes—or, boxes—that cover the fractal set. In addition, it is simple to implement an algorithm that evaluates the limit in Eq. 1.15 by assessing which boxes do or do not cover the fractal set. A computational implementation of a box-counting algorithm is described in the following chapter. We use this implementation to examine the relationship between the box-counting and similarity dimensions of the Koch curve.

[1] Fractals are not as concretely defined as one might hope. We tend only to classify a shape as a fractal when it possesses a reasonably large number of the fractal properties identified in Sec 1.1—when a shape has sufficient "roughness" that the language of fractal geometry becomes more useful in describing its characteristics.

Chapter 2

A computational approach to box-counting dimension

The box-counting dimension of any object, as we established before, quantifies its scaling behavior—i.e., how its size increases when scaled at increasing magnification. Unlike the more complicated Hausdorff dimension (which requires a computer to undertake difficult optimization problems) or the similarity dimension (which requires the set under examination to be self-similar), the box-counting dimension can be calculated by a computer with relative ease.

2.1 Description of the computational implementation

A module was written in the C++ language to find the box-counting dimension of sets in 1, 2, and 3 Euclidean dimensions; the module's source code is included in Appendix A.

2.1.1 The data structures used

Suppose for the purposes of explaining the module's implementation of a box-counting algorithm that we want to examine the box-counting dimension of a flame front in a Type Ia supernova, an example to which we will later return. Discretized data is first read in from a binary or text file. We call the grid—the set of lattice points—represented by the data a "mesh". Each point on the mesh has some number associated with it; in our example, this will be the local progress of burning $p_{burning}$ of the star within each cell on the mesh, and this number will range from 0 for entirely unburnt cells to 1 for entirely burnt cells.

2.1.2 Finding the location of the quantity of interest by n-linear interpolation

A procedure called interpolation is then performed on the data set to determine the location of the contour defining the flame front. We know from the physical properties of the system, which we will discuss later, that the progress of burning is continuous. We specify some value $p_{burning} = p_{contour}$ at which the burning is still in progress at any point in time. We say that any region with local progress of burning $p_{burning} = p_{contour}$ contains part of the flame front—if the progress of burning is between 0 and 1, it must be in the progress of burning; the flames under study do not "go out". We neglect all other regions with $p_{burning} \in]1, 0[$ because the flame front is sufficiently thin (its thickness is on the order of 1μm, while the length of the flame front is on the order of 10^6m) [4]. Hence, at any practical resolution, any cell with $p_{burning} = p_{contour}$ will also contain all values of $p_{burning} \in [1, 0]$.

2.1.2.1 The implementation of n-linear interpolation

We first search for any cell with $p_{burning} = p_{contour}$, and mark that they contain part of the flame front. We then iterate through each cell c_i with $p_{burning} < p_{contour}$ and determine whether any neighboring cell[1] to c_i has $p_{burning} > p_{contour}$. If any one of those neighbors c_{i+1} does, we know by continuity that somewhere between those two cells there exists a region with $p_{burning} = p_{contour}$. To which of the two cells should we assign the flame front's location? Suppose the centers of two such cells in one Euclidean dimension lie a distance of 1 arbitrary unit from one another. We approximate the burning progress function on the interval between the two centers as a line. The slope of that line m is given by

$$m = \frac{p(i+1) - p(i)}{i + 1 - i} = p(i+1) - p(i) \tag{2.1}$$

and the line goes through the point $(i, p(i))$, so it has equation

$$p(r) - p(i) = [p(i+1) - p(i)]r, \tag{2.2}$$

where r is some positive distance from i. For what value of r does $p(r) = p_{contour}$? We solve for r:

$$r = \frac{p_{contour} - p(i)}{p(i+1) - p(i)}. \tag{2.3}$$

[1] In n Euclidean dimensions, each cell (except those on the borders of the data set) will have $2n$ neighbors.

If we find that $r > 0.5$ (that is, half the distance between the centers of each cell in our arbitrary units), the flame front must lie in cell c_{i+1}; otherwise, the flame front lies in cell c_i. In two and three Euclidean dimensions, an identical process is performed for every pair of cells identified. This process is repeated for all neighbors of all cells with $p_{burning} < p_{contour}$.

2.1.3 Applying the equation for box-counting dimension

Once the cells containing the flame front have been marked, we count them and apply the equation for box-counting dimension, Eq. 1.15, setting the initial side length of each cell to an arbitrary $\epsilon_0 = 1$. In order to numerically approximate this limit as the size of the boxes tends toward zero, we perform a linear regression on the points in the plot of $\log N_\epsilon(F)$ vs. $-\log \epsilon$ that tend toward some constant slope as ϵ tends to zero. The resulting slope then approximates the limit. To change the scale of the boxes used ϵ we coarse-grain the data. That is, we combine the cells of the original data set, originally of side-length ϵ_0, into cells of side length $2\epsilon_0$. We average the burning progress over all the included cells, and arrive at a new data set with 1/4 the number of cells as we had originally. We repeat this process until the entire data set has been coarse-grained into one large cell, at which point we can go no further.

2.1.4 Verification of the computational implementation

Our implementation of the box-counting algorithm was verified against computer-generated data with known fractal characteristics. The implementation was able to recover the theoretical dimension with 12% error on average, as can be seen in Table 2.1.

TABLE 2.1: Theoretical vs. Calculated Fractal Dimensions Found in Fractal Analysis Validation

Fractal Object	Theoretical Dimension	Calculated Dimension	Error	% error
Dragon boundary	1.5236	1.487	-0.037	2.4
Vicsek Fractal (Box)	1.4649	1.3264	-0.14	9.5
Fibonacci Word	1.6379	1.45964	-0.18	11
Gosper Curve	2	1.72811	-0.27	14
Boundary of Gosper Island	1.1292	1.2056	0.077	6.8
Julia Set	2	1.49408	-0.51	25
Julia $z^2 + 1/4$	1.0812	1.19677	0.12	11
Levy C boundary	1.934	1.5656	-0.37	19
Pythagoras Tree	2	1.74588	-0.25	13
			Average:	12

2.2 Calculation of the fractal dimension of the Koch curve using the computational implementation

We can apply the techniques described above to find the box-counting dimension of the Koch curve. The computational method is here applied to a Koch curve of size 1024 pixels end-to-end. Pictured in Fig. **??** is the plot of $\log N_\epsilon(F)$ vs. $\log(1/\epsilon)$ along with the line resulting from the linear regression performed.

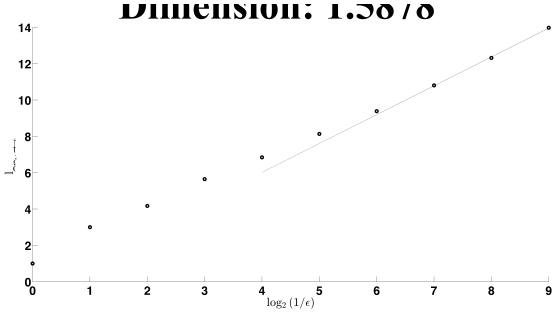

FIGURE 2.1: Plot of $\log N_\epsilon(F)$ vs. $\log(1/\epsilon)$ along with the line resulting from the linear regression performed. The slope of the plot clearly changes as $\log(1/\epsilon)$ increases to its last tested value of 9, indicating the importance of performing the regression only on the last points. The dimension, the slope of the line, is printed above the diagram.

We find that the curve has a dimension of 1.588, differing from the similarity dimension of the curve (≈ 1.28) by $\approx 25\%$.

2.3 Applications

How supernovae work...flame fronts...yep... random walks...

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%s%.5f\n%s%1.3f\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;
}
```

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] F. X. Timmes. On the acceleration of nuclear flame fronts in white dwarfs. *Astrophysical Journal*, 423:L131, 1994.