MULTIFRACTAL ANALYSIS OF DNA

By

Rasekh Rifaat

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, Manitoba, Canada

Thesis Advisor: W. Kinsner, Ph. D., P. Eng.

(ix+62+A38) =109 pp.

0-612-32231-9

MULTIFRACTAL ANALYSIS OF DNA

BY

RASEKH RIFAAT

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University

of Manitoba in partial fulfillment of the requirements of the degree

of

MASTER OF SCIENCE

ABSTRACT

This thesis presents two techniques for analyzing DNA using a multifractal methodology. The DNA analysis presented in the thesis is motivated by the intriguing possibility of identifying biological functionality using information contained within the DNA sequence. In addition, the analysis may give insight into the nature of DNA complexity, and provide guidelines for the selection of operating parameters such as the minimum DNA sequence length which can be analyzed. The first technique breaks a DNA sequence into four subsequences based on the individual constituent bases, and treats each of these as strange attractors from which the multifractal dimension may be estimated. Results show that the generated subsequences exhibit multifractal properties which can be localized to different positions along the sequences. A minimum window size of 256 bases, and a scaling range from 64 to 256 bases is needed for estimation of the multifractal measures.The second technique estimates the multifractal spectrum of DNA based on *n*-block entropies. The minimum window size was selected to be 1024 bases along with a scale range of one to three base pair sequence lengths. Experimental results show that DNA has a multifractal characteristic using this measure, and that the multifractality changes depending upon the position in a sequence. The phylogeny of organisms based on their multifractality was demonstrated with only two misclassifications, which may have other unresolved issues.

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Dr. W. Kinsner, for his guidance and support throughout this journey. I have always appreciated his encouragement, and I am extremely grateful that I have been able to work on such an outstanding research topic.

I would like to thank Dr. McAlpine for her encouragement, and support in making this initial collaboration possible.

I would like to thank Tina Ehtiati for all of her help and ideas in writing this thesis. Thanks also to Megan Tate for also reviewing the text of this document.

I would like to thank all of my colleagues in the Delta Research Group, past and present, who influenced my surroundings everyday and continually challenged me to become better: Tina Ehtiati, Richard Dansereau, Epiphany Vera, Eric Jang, Pradeepa Yahampath, Steve Miller, Jason Toonstra, Luotao Sun, Fan Mo, Shamit Bal, Alexi Denis, Jonathan Greenberg, and Hongjin Chen.

I would also like to thank all of my friends who have been there for me over the past two years. In particular, I would like to thank Andrew Dalgarno and John Tajima for their endless friendship over the past two years.

I would like to thank my entire family, all of whom have supported me throughout this research. This thesis is dedicated to my mother, Laila Bassim.

Partial financial support is acknowledged from the Natural Sciences and Engineering Research Council (NSERC) of Canada.

Table of Contents

LIST OF FIGURES

CHAPTER I
INTRODUCTION

1.1 Background and Motivation

Deoxyribonucleic acid (DNA) is one of the most examined molecules on the planet. Scientists around the world are trying to discover its secrets for many purposes. Currently genetic information is used to raise better plants and animals, create enhanced pharmaceuticals for humans and in medicine for gene therapy. Science as a whole has benefited from the study of genetics because of increased understanding of the biological processes that all organisms share.

In recent years, a significant amount of research has been directed towards sequencing and understanding the entire human genome in the form of the Human Genome Project (HGP). There are many conceivable applications which would be available once the entire human genome is understood. The majority of progress so far, however, has been only to determine the sequence, with only limited breakthroughs in understanding what it does. The approach from the lowest level has not been able to provide the answers to questions regarding higher level knowledge within the DNA sequence.

Recently, the approach has been modified to search for the higher level of knowledge or "meta" information. Researchers have been examining DNA sequences for statistical properties which might explain a portion of the DNA's function [ABGW95] [BeRO96] [PBGH92] [ScHe97]. In this thesis, we turn our eye to a technique developed in our research group which has proved successful at finding the meta knowledge in other areas of study.

Multifractal techniques [Kins94] have proven successful in the examination of a wide range of signals such as computer images, radio signals, human fingerprints and speech [Lang96] [Chen97] [Shaw97] [Jang97] [Grie96]. By using them to analyze DNA, the prospect of finding some higher level information is very good.

1.2 Thesis Statement and Objectives

The objective of this thesis is to achieve estimations of multifractal measures for DNA sequences and to identify ranges of values for those measures. Through the use of established techniques, such as the multifractal spectrum [Kins94], we will show that DNA can be analyzed and characterized from fractal methods. DNA from various sources (bacteria, mitochondria) will be analyzed. We will also determine how the multifractal spectrum changes over different parts of the DNA sequence.

1.3 Thesis Organization

This thesis is organized into six chapters. Chapter 2 provides a general introduction to DNA, fractals, multifractals and how they have been used together in previous research. Chapter 3 describes the theoretical basis for calculating the two definitions of estimating the multifractal dimension of DNA. Experimental results which show the multifractal signature of DNA for single base subsequence attractors are given in Chapter 4, and experiments with multiple base sequence attractors are given in Chapter 5. Conclusions, recommendations, and contributions are presented in Chapter 6.

CHAPTER II
BACKGROUND ON DNA AND MEASURES

In order to analyze the information content in DNA sequences, we must first understand how and what they are, and what their constituent parts are. Following that, we will describe how the amount of information in an object may be quantified using information theory techniques. Power-law relationships will be introduced as a method for understanding complex behavior which cannot normally be characterized.Multifractal measures are then introduced to provide a framework for discussions about the complexity of objects. Other research which has combined DNA and fractality is reviewed in order to understand what has already been accomplished.

2.1 DNA

Deoxyribonucleic acid (DNA) and ribonucleic acid (RNA) is the information source for all organisms on Earth today. They serve to encode the entire genotype in each cell of an organism, and then allow for a sharing of the knowledge with offspring. The main DNA functions are transcription, translation, and replication.

DNA is a complex molecule which can be represented compactly as a string of nucleotide bases. It is a double stranded linear molecule composed of four compounds: adenine (A), guanine (G), thymine (T), and cytosine (C). DNA sequencing consists of determining the particular order of nucleotides in a given organism's DNA strands. Due to the nature of the binding properties, DNA has a complementary strand structure where the information on one strand is encoded in a complementary form on the other strand. The complements of A, G, C, and T are, respectively, T, C, G, and A. In RNA, thymine is replaced by the nucleic acid uracil which then complements adenine. The effect of the complementary bases, is that only one of the strands in a DNA molecule needs to be sequenced in order for all of the knowledge to be captured. Adenine and guanine are grouped together in a class of nucleic acids known as purines. Cytosine, thymine, and uracil are known as pyrimidines. Because of the pairing of two strands, DNA sequences are said to have lengths consisting of base pairs (bp), even though only one of the complementary strands may be given.

DNA is transcribed from nucleotides into RNA, specifically messenger RNA (mRNA), which is then translated into a series of amino acids. The amino acids are strung together in a linear fashion in order to make up a protein. A series of three nucleotides, known as a codon, is used to specify a particular amino acid. There are $4^3 = 64$ possible codons, but because some are redundant, they are used to specify only 21 different amino acids. Three codon combinations are used to stop a protein chain from extending, and one is always used to start a translation. Codons appear in a sequential non-overlapping manner within the DNA string. Some bases in the DNA sequence do not directly code for proteins, and may be used to regulate the transcription, translation and replication processes.

There are three possible interpretations, known as reading frames, of a DNA coding sequence depending on which nucleotide is selected as the start of the translation process. If either of the next nucleotides is selected instead, a completely different protein sequence will be translated. If a nucleotide three positions away is selected, an amino acid is deleted or added from the protein chain.

From the discovery of DNA as the information carrier in biological systems, there has been an ongoing analysis of how DNA translates into a physical description of organisms, or phenotype. There have been great advances in understanding the nature of codons, and their translation into proteins. It has been shown that the DNA of any organism is sufficient to code all of the information needed to construct and maintain it [Lewi97]. The organization of genetics is focused around the gene, which has many interpretations. In general, a gene is considered to be a continuous portion of DNA which codes for a certain polypeptide chain, one or more of which may form a functional protein. Genes are very wide ranging, and are generally very difficult to classify. Some genes are made up of both coding (exon) and non-coding (intron) regions. The introns are spliced out of the mRNA before being translated to a protein, but the mechanism of their identification is currently poorly understood.

Different organisms have different DNA characteristics. Bacterial DNA does not have any introns, while eukaryotic DNA has introns and exons. As well, mitochondrial DNA does not have any introns, and even has regions whereby two different proteins are are coded in the same portion of DNA sequence (overlapping genes). It is also important to consider if the whole DNA sequence will be considered, or only parts of it. Perhaps, DNA which has already had all of the introns sliced out might be used. In general, the target application will determine what portion of the DNA will be used. If the search is for all DNA components, then the entire string might be used, while for gene function only a portion of DNA might be used.

One particular feature is repeated sequences within a piece of DNA. For example, in human DNA, there are several locations which have 40-50 repeats of the sequence GC. There are many other repeats which abound. Some DNA fingerprinting makes use of this fact, and uses the number of repeats as an identifier for a particular person.

Research in genetics in the past has focused on the direct understanding of DNA and the transcription of DNA into RNA and then protein. New techniques to determine DNA sequences, such as "shotgun sequencing", along with experimentally determine the gene function has been the primary emphasis.

There has been a trend recently to examine DNA at a higher level and search for patterns or correlations which exist in the DNA string. If DNA were a completely random pattern, then these correlations shouldn't exist. Recent research has shown that these correlations exist, and can be exploited.

2.2 Information Content of Symbolic Sequences

The concept of measuring information was first introduced by Shannon, and has been used extensively in many different subject areas. Shannon's entropy is defined for a sequence S with symbols $s_i (i = 1, ..., \lambda)$ as

$$H = -\sum_i p_i \log_2 p_i \qquad (2.1)$$

where p_i is the probability that s_i occurs, and i extends over the alphabet of symbols. The most common interpretation of (2.1) is that H represents the minimum number of bits which are required in order to code the sequence. An extension of (2.1) occurs by defining block-entropies as

$$H_r = -\sum_i p_i^{(r)} \log_2 p_i^{(r)} \tag{2.2}$$

where $p_i^{(r)}$ are the probability that an ordered combination of r elementary symbols occurs. The number of compound symbols is now λ^r.

The most difficult aspect of calculating entropy is estimating the probability density function of the different block sequences. The accepted practical algorithm is to estimate probability using a relative frequency method where the probability of an event is the number of times the event occurs averaged over the total number of events examined. Unfortunately, it is difficult to estimate the number of samples which are needed before an accurate estimate of the probability density can be established. Some guidelines are to continue evaluating events until the minimum sample count is ten (i.e. the smallest count for any event is ten). A method has been suggested which can extend the accuracy of the estimated entropies for DNA sequences and literary texts using short sequences [ScHe97]. In some cases, the above guideline is impractical, and can be reduced if there is evidence that some sequences are simply likely to never occur.

2.3 Multifractal Measures

Fractals are now a common concept in physics and mathematics. They embody the idea that patterns exist in deterministic systems which appear chaotic. The fact that there is a relationship among different scales provides a powerful measuring tool with which to characterize "natural" signals. In general, these natural signals are difficult to analyze using classical techniques such as Fourier analysis because of their nonstationary behavior.

2.3.1 Power Law Relationship

Fractals are related to the two fundamental concepts of scale and measurement. In a fractal, these two properties of an object are intertwined, and their relationship can be exploited and characterized.

Fig. 2.1 The coastline of Britain.

One of the most famous examples is a discussion of the length of the coast of England, shown in Fig. 2.1 [PeJS92] (Along with other coastlines [Kins97][Fede86]). The first approach would be to use a compass on a map by fixing the compass setting and walking the compass around the island. The number of steps multiplied by the compass setting would give an approximation to the length of coastline. However, if we choose a smaller compass setting, and repeat the measurement of length, there is an increase in the value. The increase is explained by the bays, bends, and general jaggedness of the coastline, which are counted toward the length when the compass setting is reduced. In fact, the compass setting could be reduced again, with another measurement of length. In fact, as the compass setting keeps becoming smaller and smaller, the length of the coastline approaches infinity.

The lesson in the coastline experiment is that it may be impossible to measure something accurately because it depends on the scale of measurement. The relationship between the scale and coastline length can be measured. By plotting the scale versus coastline length, there appears to be an exponential relationship. Changing to a grid whereby the axis are plotted using log scales (log-log plots), a very clear linear relationship appears whereby the points fit to a straight line. The mathematical expression is given as

$$L(s) = c\left(\frac{1}{s}\right)^d \qquad (2.3)$$

where L is the measured length, s is the scale of the plot, and d and c are constants. It turns out that the value of d is characteristic of the complexity of the object being examined. In the coastline example, an increase in d means that there are many bays, bends and jags which are being added to the measurement at each scale reduction. A decrease in d shows that the shape of the coastline is relatively flat, and that decreasing the scale doesn't increase the measurement of length as much.

For objects which are not "complicated," d tends to go to zero since there is no variation when the scale is changed. Objects like straight lines, squares, and circles exhibit a power-law relationship of zero, indicating the lowest possible complexity.

The primary value of d, is that it provides an estimate of the complexity of the length of an object. Other measures can be used in the calculation, and for a single object, a different measure can provide a different level of complexity.

2.3.2 Euclidean and Fractal Dimensions

In mathematics, there has been a significant amount of discussion regarding the concept of dimension. The most common is Euclidean dimension, D_E, the smallest integer space that an object can be placed into. For instance, a point has a dimension zero, a line has dimension one, a square has dimension two, and a cube has three dimensions. No special method of calculation is needed for D_E, and it is sometimes referred to as topological dimension. Note that an object like the coast of Britain is considered to have a Euclidean dimension of one since it can be stretched out and represented by a single variable.

Dimensionality can be further generalized by considering non-integer dimensions, also known as fractal dimension. Fractal dimension is based on the power-law relationship, and constitutes a measure of complexity for an object. There are numerous fractal dimensions which have been used in research.

The box-counting dimension is the most popular since it is extremely easy to calculate and understand, which leads to its common occurrence. Any object may be covered using a grid of boxes (volume elements or vels) which have homogeneous sizes. The number of vels which covers the object is taken as the measurement for the power-law relationship, while the size of the vels is the scale factor. Reworking (2.3), the box-counting dimension becomes

$$D_b = \frac{d}{ds}\left(\frac{\log L(s)}{\log \frac{1}{s}}\right) \tag{2.4}$$

where $L(s)$ is the number of covering boxes for a given vel size.

The box-counting dimension is an example of a morphological dimension. Several classes of dimension have been identified, such as morphological, entropy, spectrum, and variance classes [Kins94]. A feature of the box-counting dimension which is not very desirable, is that it does not provide any information regarding the distribution of the object, merely an estimate of the complexity of its outline.

The information dimension, D_I, alleviates this problems somewhat by considering the information content of the object. It is defined as

$$D_I = \lim_{r \to 0} \frac{H}{\log(1/r)} \tag{2.5}$$

where H is Shannon's Entropy (2.1), and r is the size of the vel. The probability is the relative frequency with which the object intersects the ith vel.

The box-counting and information dimension are only two of many other fractal dimensions which can be computed, and which can give different results for the same object. There has been an argument in recent years that a single dimension does not provide enough information about the complexity of an object, and that it would be wise to use several of these measures to characterize the complexity.

Since it is quite possible to apply these measures to objects which do not exhibit complex behavior, further clarification is needed. More specifically, an object is only considered to have fractal properties when its fractal dimensions are non-integer. The application of fractal dimension calculations to simple objects such as circles, lines, and cubes will result in an integer dimension which matches the Euclidean dimension.

2.3.3 Rényi Generalized Entropy and Dimension

As seen in (2.1) and (2.2), Shannon's entropy is defined as a measure of the amount of information in a sequence of symbols. Rényi [Rény70] [Kins94] then extended the idea to a generalized entropy given by

$$H(q) = \frac{1}{1-q}\log_2 \sum_{j=1}^{N} p_j^q \tag{2.6}$$

where q is called the moment-order. Shannon's Entropy is a special case of (2.6) with $q = 1$, and is treated below. A proper interpretation of the generalized entropy must be include consideration of the statistical technique used to estimate the probabilities.

As with the previous definitions of fractal dimension, evaluating the generalized entropy at different scales, r, a power law relationship may be discovered using

$$D_q = \lim_{r \to 0} \frac{1}{q-1} \frac{\log_2 \sum_{j=1}^{N_r} p_j^q}{\log_2 r} \tag{2.7}$$

which is called the multifractal spectrum or the Rényi (multifractal) dimension. As with any power-law relationship, the estimate of D_q comes from the linear fit of the denominator to the numerator of (2.7). An effective multifractal spread exists if D_q is monotonic, nonincreasing within error, and insensitive to small changes in the scaling range with respect to the moment-order.

We can also consider how D_q responds to extremely high and low moment orders. For $q = \infty$, (2.7) becomes

$$D_{\infty} = \lim_{q \to \infty} \lim_{r \to 0} \frac{1}{q-1} \frac{\log_2 \sum_{j=1}^{N_r} p_j^q}{\log_2 r} = \lim_{r \to 0} \frac{\log_2 p_{max}}{\log_2 r} \tag{2.8}$$

where p_{max} is the highest probability, because of the sifting property of the infinite (Chebyshev) norm [Kins94]. Similarly, for $q = -\infty$, (2.7) becomes

$$D_{-\infty} = \lim_{q \to -\infty} \lim_{r \to 0} \frac{1}{q-1} \frac{\log_2 \sum_{j=1}^{N_r} p_j^q}{\log_2 r} = \lim_{r \to 0} \frac{\log_2 p_{min}}{\log_2 r} \tag{2.9}$$

where p_{min} is the lowest probability. There are several special cases which can be considered in regards to D_q. If we consider $q = 0$, (2.7) simplifies to the box counting dimension (2.4), where the number of the covering vels is used instead of the probability of occurring within a vel. By setting $q = 1$, D_q reduces to the information dimension (2.5). More specifically, $H(1)$ cannot be evaluated directly, so the limit as q approaches one is applied

$$H(1) = \lim_{q \to 1} \frac{1}{q-1} \log \sum_j p_j^q \tag{2.10}$$

Using L'Hopital's rule, we get

$$H(1) = -\lim_{q \to 1}\left(\frac{1}{\sum_j p_j^q}\right)\left(\frac{d}{dq}\sum_j p_j^q\right)\log_2 e \tag{2.11}$$

In order to evaluate the derivative in (2.11), we use $\frac{d}{dx}c^x = c^x \ln c$ and get

$$H(1) = -\lim_{q \to 1}\frac{\sum_j p_j^q \ln p_j}{\sum_j p_j^q}\left(\frac{1}{\ln 2}\right) = -\sum_j p_j \log_2 p_j \tag{2.12}$$

which is Shannon's Entropy (2.2).

To clarify, a window is a piece of DNA which is a continuous portion of another DNA sequence. Usually, the entire DNA sequence for an organism will be the reference sequence, and a window could consist of any section of the sequence, or the entire sequence itself. The value in approaching each sequence from a window perspective is that investigations into how the multifractality of different windows compares can be made. The result is to determine whether there are complexity changes within a DNA sequence, or whether the DNA sequence has a homogeneous complexity.

(2.6) and (2.7) describe the theoretical values of $H(q)$ and D_q respectively, but for some practical applications, the window size should be incorporated into the definition. If the size of the window to be analyzed is too small, then estimates of the probabilities within the window will not be valid.

As in the previous section, for simple objects the multifractal spectrum is a constant integer. Single fractal objects have a constant non-integer multifractal spectrum. Objects which are multifractal in nature will have a non-increasing spectrum, while single fractal objects will have a single uniform dimension for all values of q.

2.3.4 Mandelbrot Spectrum of Dimensions

Another method of analyzing multifractals is presented with the Mandelbrot dimension. In single fractal dimension calculations, such as the information and box-counting dimensions, a single region size is considered, and as a result only a single-valued power-law relationship (2.3) was measured. We can improve that by considering a nonuniform set of regions, r_j, with varying probabilities, p_j. The local power-law relationship then becomes

$$p_j(r_j) \sim r_j^{\alpha_j} \tag{2.13}$$

where α_j is a noninteger which depends on the selected region of the measure. The local scaling exponent, α_j, is called the Hölder exponent [Kins94]. Additionally, it is possible to cover the regions with a set of vels, and determine the number of vels with a specific α, $N_\alpha(r)$. A power law relationship ship now exists in the form of

$$N_\alpha(r) \sim \frac{1}{r^{f(\alpha)}} \tag{2.14}$$

where $f(\alpha)$ is the fractal Mandelbrot dimension, D_{Man}. The Mandelbrot dimension is related directly from the Rényi dimension by a Legendre transformation with the following explicit result [Kins94]

$$\alpha_q = \frac{d}{dq}[(q-1)D_q] \tag{2.15}$$

and $f(\alpha) \equiv f_q$ with

$$f_q = q\alpha_q - (q-1)D_q \tag{2.16}$$

The Mandelbrot dimension provides an alternative viewpoint to the multifractality of a given object, and provides different visual features than the Rényi dimension.

2.4 Analysis of DNA sequences using statistical and fractal measures

Most of the current research in the deciphering the meaning of DNA sequences is approached from the lowest level. Analysis of codons, amino acids, and proteins are the main subjects of research. In this thesis, the perspective is changed, and a higher level of information is sought. Searching for the higher level, or "meta," knowledge can take various forms.

In recent years, there has been an interest in the properties of DNA from a statistical standpoint. There have been several investigations into the statistical distribution of bases over the DNA string. Attempts have been made to link the statistics of a piece of DNA to whether or not it codes for protein (exons vs. introns).

Voss brought to the forefront the idea that there were self-similar characteristics in DNA sequences which could be measured [Voss92]. He outlines a particular method of translating DNA into a random walk which is then measured. As well, he examines the power spectrum of the bases of DNA and finds that they exhibit 1/f (pink) noise which is a strong indication of fractality. Also noted is a very strong periodicity at a frequency of three, which is believed to be related to the codon size.

Several published results show that the calculations using a random walk model can be used to characterize different DNA sequences [BeGS92][BeGR94][GBSR95]. In particular, they use the multifractal spectrum to reconstruct the phylogeny of mitochondrial DNA. They apply the technique to entire mitochondrial genomes and classify the results using a hierarchical clustering technique. The random walk methods maps the DNA sequence into a two-dimensional image, which provides a useful visualization tool.

Other researchers have recognized the importance of the higher level knowledge which is present in DNA [PBGH92][BGHP93][BeRO96]. There have been several views on what the implications of "long-range correlations" are. As of now, there is no widely accepted opinion as to the source of the correlations, or what they mean in a biological sense.

A multifractal spectrum trajectory may be considered for DNA whereby the multifractal spectrum is calculated for adjacent windows and a curve showing how the spectrum changes over the DNA sequence is generated. We have not included results for the multifractal trajectory due to limited time, but it has been considered.

In this study of higher level knowledge in DNA, we expect to show that there is a possibility that the multifractal estimates can be used to characterize a given DNA sequence. As well, it is important to determine whether there is local complexity within a DNA sequence, or if the multifractal dimensions are constant throughout a sequence. Guidelines for selecting the proper operating parameters should also be established.

2.5 Summary

Background information on DNA and measures which are used throughout this thesis are discussed. A simplified explanation of DNA and its attributes is presented in order to understand the properties of the sets which will be experimented on. The information and fractal measures present possible measuring tools which we will apply to DNA sequences in the following chapters. A review of some related research shows that long-range correlations are present, but not completely understood.

CHAPTER III
MULTIFRACTAL MEASURES OF DNA

The objective of this thesis is to analyze DNA at a higher level than has previously been accomplished. The multifractal measurements which have been described will serve to provide a basis with which characterization of the DNA sequence may follow. Two definitions of the probability distribution will be given along with guidelines which describe how to find the correct window sizes and scales for regression. The sequences which will be analyzed will also be presented.

3.1 Calculation of Rényi's Generalized Entropy

Previous attempts at calculating the fractal dimension of DNA have used a random walk model introduced by Voss [Voss92] and used in several different papers [PBGH92] [BeGS92]. Instead of this approach, we consider that the DNA sequence is itself a strange attractor, and as such can be measured directly. Two interpretations of the Rényi dimension are outlined and discussed in this thesis.

In fractal dimension calculations, the most important quantities to determine are the necessary probabilities. Since the nature of DNA sequences is significantly different from a discrete time sampled signal such as audio or video [Chen97] [Shaw97] [Jang97] [Grie96], the selection of a probability measure needs to be explored.

3.1.1 *Single Base Subsequence Attractors (SBSA)*

Unlike discrete time signals which have real values at each sample, DNA sequences consist of symbols which can take one of four values at any given position within a sequence. As well, there is no relative importance between the symbols which makes it difficult to convert them to a discrete time series. The first approach implemented is to consider that each base makes up a strange attractor within the DNA, and to calculate the probabilities for each individual base separately. The result is a Rényi dimension description for each of the four bases. One technique to visualize this is to convert a DNA sequence to a portion of the real line, where a point exists if the target base exists in the sequence. The result is a set whose multifractal spectrum may be estimated. Four such sets are inherent for any DNA sequence. Alternatively, a single subsequence could be created if we considered whether or not each base is a purine or pyrimidine. The probability (relative frequency) that an attractor for a given base exists in a chosen vel is

$$p_j = \frac{n_{bj}}{N_b} \tag{3.1}$$

where n_{bj} is the count of base b in vel j, and N_b is the total count of base b in the sequence. This definition satisfies the requirements that the sum of all the probabilities must equal one.

From the above probability definition (3.1), Rényi's generalized entropy given in (2.6) can be calculated for various vel sizes. In this particular instance, the vels are non-overlapping and cover the sequence completely. The entropy, when plotted against vel size, provides a means of calculating the power law relationship given by (2.7), which exists for this strange attractor.

3.1.2 *Multiple Base Sequence Attractors (MBSA)*

Another view of the spectrum of fractal dimensions can be determined from an information theoretic point of view. We can consider Rényi's generalized entropy (2.6) to be a measure of block information which can be applied to DNA symbols directly. As in Section 2.2, we define a set of disjoint ordered sequences of length r which make up the event space. The sequences are made up of all of the possible combinations of the four bases of DNA (A,G,C,T). For a fixed scale, r, the number of elements in the event space is

$$N_r = 4^r \tag{3.2}$$

because of the alphabet size of DNA. The definition of probability then becomes

$$p_j = \frac{n_j}{N} \tag{3.3}$$

where n_j is the number of times the jth element occurs, and N is the total number of elements examined. This definition is also consistent with that of n-block entropies.

A pitfall of this technique is that in order to calculate the probabilities for large vel sizes, r, an impractical amount of bases is necessary. For example, consider $r = 8$, which gives $N_8 = 65536$. In order to get a valid probability distribution function for such a large event space, sequences which are at least ten times as long need to be evaluated. As well, the structure of natural DNA includes a significant amount of repeat strings which concentrates the probability distribution function at certain symbols [ScHe97]. The obvious advantage over the SBSA is that only a single entropy and dimension set need to be calculated, while for the subsequences a set of four dimensions is needed for a complete representation.

3.2 Selection of Vel Sizes

In most fractal measurements, a dyadic vel size of 2^n is used because computer systems operate efficiently for powers of two. This stems from the nature of current computer architectures which favor binary numbers and representations. As well, using a dyadic sizing provides some efficiencies for calculating logarithms of base 2.

Using a dyadic sizing, an estimate of the entropy can be obtained. However, due to the codon nature of DNA, a triadic vel size of 3^n may be more appropriate. This size would likely have less interference from codon boundaries than others. It might also prove useful in the identification of open reading frames.

In practical applications, a dyadic or triadic vel scaling may not be feasible. In considering the MBSA calculations, as the vel size increases, the number of elements increases exponentially. In this case, the vel size is selected to be a range of small integers such as $1, 2, 3, \ldots, r$ in order to accurately estimate the probability distribution.

3.3 Selection of the Scale Range for Regression

In order to calculate the multifractal spectrum, the slope of the log-log plot must be determined. The standard technique is to use least squares regression to fit the data to a line thereby determining the slope[PeJS92]. In many cases the entropy values at certain scales cannot be used due to outliers and behavior which clearly do not follow the power law relationship. In practice, a continuous range of scales is selected over which the power law relationship holds, and the dimension is estimated using only these scales.

In determining which bases to use for the slope calculation, it is important to have a general idea of the structure of the generalized entropy vs. vel size plot. This figure will clearly show any regions which do not adhere to the power law relationship necessary for the fractal dimension calculation. The outliers might take one of many forms. For instance, many log-log plots are erratic or curved over a particular range of scales.

The determination of scales will greatly affect the final applicability of the multifractal spectrum. By determining the range of scales to be used, maximum and minimum window sizes will be determined. The minimum window size will determine the smallest resolution within the DNA sequence which can be identified. The maximum size will pinpoint the necessary sequence lengths which are required for the calculations.

A distinct possibility may also be that no maximum window size exists, and that one may be established which, for practical purposes, is sufficient to calculate the multifractal spectrum accurately. A minimum window size should exist due to the discrete nature of the

DNA sequence. In this thesis, the minimum and maximum scales used for any given regressions will be denoted as R_{min} and R_{max}.

3.4 Dividing the Dimension Space

Part of the novelty in the proposed technique is that the entire sequence is not required in order to get an estimate of the fractal dimension, and that small windows can be used. In fact, large windows will tend to average out changes in the different parts of the sequence. An experiment will be to examine the global multifractal dimension in comparison to its parts. This example should show what the relationship is between the overall and constituent parts. It would also help determine what a reasonable window size is. If it turns out that the global and constituent parts are the same, then, there is no variation of the multifractal dimension along the DNA sequence, and the whole sequence could be used for characterization. This is highly unlikely, since the various genes are already known to have different statistical properties[Lewi96]. It would be useful to determine whether or not a single gene has a given characteristic which could be used to isolate that gene.

3.5 Selection of DNA Sequences

Due to the ever increasing volume of sequenced DNA data, there is no shortage of sequences to experiment on. As a result, two main sequences were selected as the main benchmarks for these experiments. The *Escherichia coli* genome is fully sequenced and annotated. *Methanococcus jannashii* is an archaeabacteria which is also fully sequenced and annotated. These two sequences were chosen because of the wealth of "extra" information in addition to the full sequence, such as gene identification and annotation. These sequences also represent some of the longest contiguous sequences available with 4.6 and 1.5 million base pairs respectively. The long sequences allow for experimentation over a very large range of vel and window sizes.

In [GBSR95], the multifractal dimensions of DNA sequences were calculated using a random walk model of DNA. It was shown that the multifractal curves could be used to reconstruct the phylogeny of the various DNA sequences without any additional information. In order to provide a comparison to the random walk multifractal spectrum, the MBSA multifractal spectrums of the same sequences will be computed in Section 5.5. The sequences used are shown in Fig. 3.1.

Sequence	GenBank Code	Length
Home sapiens (human)	HUMMTCG	16 569
Bos taurus (cow)	MIBTXX	16 338
Rattus norvegicus (rat)	MIRNXX	16298
Phoca vitulina (harbor seal)	MIPVDNA	16 826
Balaenoptera physalus (fin whale)	MIBPCG	16 398
Xenopus laevis (toad)	XELMTCG	17 553
Cyprinus carpio (carp)	MICCCG	16 364
Crossotoma lacustre (fish)	CRQMTGENOM	16 558
Drosophilia yakuba (fruit fly)	MIDYRRN	16 017
Apis mellifera (honey bee)	AMFGENOM	16 343
Strongylocentrotus purpuratus (sea urchin)	MISPXX	15 650
Paracentrotus lividus (sea urchin)	PALMTCG	15 696
Caenorhabditis elegans (nematode)	MTCE	13 794
Ascaris suum (nematode)	MTAS	14 284

Fig. 3.1 Mitochondrial DNA sequences accessed from GenBank.

In addition to natural biological sequences, several synthetic sequences were used as a control group to test the calculations. Synthetic sequences were created with the following patterns: random uniform base distribution, single base, short periodic repeats.

3.6 Calculation of Multifractal Measures

All of the calculations presented were implemented in C/C++ on a Sun UltraSparc 1 using Sun Microsystems C/C++ compiler. The structure charts and code are presented in the Appendix. The code which implements linear regression is taken verbatim from [PTVF92].

In the design of the software, there were several objectives. First of all, the code was designed so that as much of it as possible could be reused for a different data set. Secondly, the code was designed to be portable. The software is split up into three main modules as described in Fig. 3.2. The final implementation is a command line program which takes as arguments the parameters in Fig. 3.3.

1. MultifractalCalc.{h|cc}

   This file contains the classes which implement the Multifractal calculations for DNA used in this thesis, along with a couple of support classes.
2. genecalc.cc

   This file contains the source for the driver which reads all of the parameters, sets up the calculation objects, and then writes all of the results to the appropriately named files.
3. PatternIO.{h|cc}

   The object which is used to input DNA sequences is given in these files. As well, the generation of synthetic sequences is handled here.
4. fit.{h|c}

   This code from [PTVF92] performs a linear regression given x and y data.

Fig. 3.2 List of source code files.

1. Sequence file name

   The directory and file name of the flat file containing the DNA sequence for processing. The file should be pre-processed to remove and other symbols or notation. The only characters allowed in the file are A,G,C, or T.

2. Start and End base

   These two parameters define how big the window to be used should be. The end base parameter may be substituted for by 0, which uses the minimum number of bases necessary for the SBSA calculation according to the scaling, or 1, which uses all of the DNA until the end of the file.

3. Scaling scheme

   For the SBSA calculation, this parameter indicates the base of the vel scaling. For example, 2 indicates a dyadic scale, while 3 indicates a triadic scaling.

4. Maximum scales

   Two parameters are needed, the first indicating the maximum scale for the SBSA technique, and the second indicating the maximum scale for the MBSA technique.

5. type of dimension calculation

   This parameter is a single letter which indicates which dimensions to calculate. An 'r' indicates the full Rényi spectrum, while a 'v' calculates only the Variance dimension, and an 'i' calculates only the Information dimension.

6. Start/End Regression scales

   These two parameters indicate which scales to start and stop the regression parameter. If a one is entered for both, then the dimensions are calculated for all possible combinations of scales which are continuous.

7. Output file prefix

   This parameter specifies what the prefix of all output files are. The output files are names with the prefix, and then various suffixes are added indicating the calculations within that file, and for SBSA calculations, which base was used.

Fig. 3.3 Parameters input into the genecalc program.

The software was tested continuously as new modules were added. Intermediate results of calculations were stored and displayed to confirm that there were no mathematical errors. As well, once code was running correctly, an effort was made to re-use it verbatim. Both the SBSA and MBSA approaches use the same piece of code to perform linear regression and calculate the Rényi spectrums. Through the use of C++, data hiding and encapsulation was accomplished resulting in simplified interfaces.

3.7 Summary

The sequences which will be analyzed in this thesis are described along with two multi-fractal approaches, SBSA and MBSA, for measuring the sequences. A framework of how the entropies and scale ranges should be determined was discussed. Experiments are described which can highlight the differences in characteristics between sequences, and within sequences.

The calculation of the multifractal measures using a specific implementation was presented, along with an outline of the software designed for this thesis. Software validation was performed at each step to ensure that all calculations performed are correct.

CHAPTER IV
EXPERIMENTAL RESULTS FOR SINGLE BASE SUBSEQUENCE ATTRACTORS

The first technique which we will examine is the single base subsequence attractor method for calculating the fractality of DNA. This method presents an interesting viewpoint because it treats each of the constituent parts of DNA as its own entity within the entire sequence. The entropy calculations will show if there is any connection between scales for this measure. As well, we will discuss the minimum window size using this technique. The relationship between a sequence of DNA and its pieces are also explored.

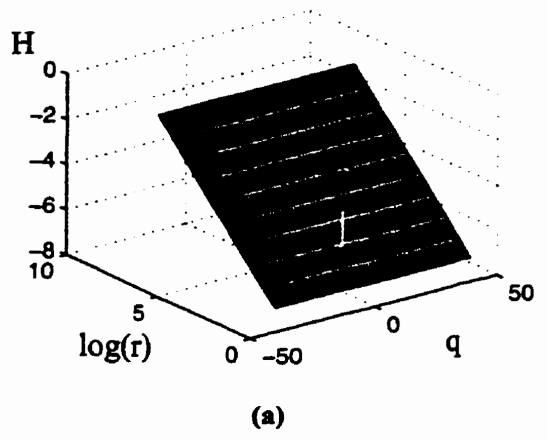
4.1 Rényi's Generalized Entropy Calculation

The generalized entropy values were determined for several DNA sequences, as it is a fundamental calculation necessary for calculating the multifractal spectrum. Several examples of various calculations are presented in three dimensional form. The horizontal axis are q and $\log_2 r$. The vertical axis shows the Rényi entropy as defined in (2.6). All of the figures depict similar behavior. All of the plots in this section used a dyadic scale for calculation unless stated otherwise.

4.1.1 Synthetic Sequence Tests

The entropy calculation was first tested using synthetic sequences. For sequences which consist of a single base, the calculations show the expected entropy for the selected base. For the other bases, since the probabilities are all zero, taking the logarithm results in undefined values. Synthetic short repeats (Figure 4.1b uses the repeating sequence of

AGCT) also exhibit flat simple behavior and showed the same results in structure as a single base.



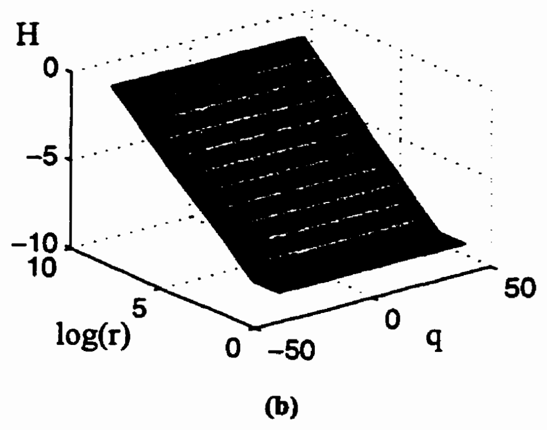


Fig. 4.1 Generalized entropy of synthetic sequences containing (a) all A, and (b) repeated AGCT

The behavior of random sequences gives the same structures as those found in natural sequences, discussed in the next section. The random sequences were generated using a uniform random number generator which had an equal probability of generating any of the four bases for a given position.

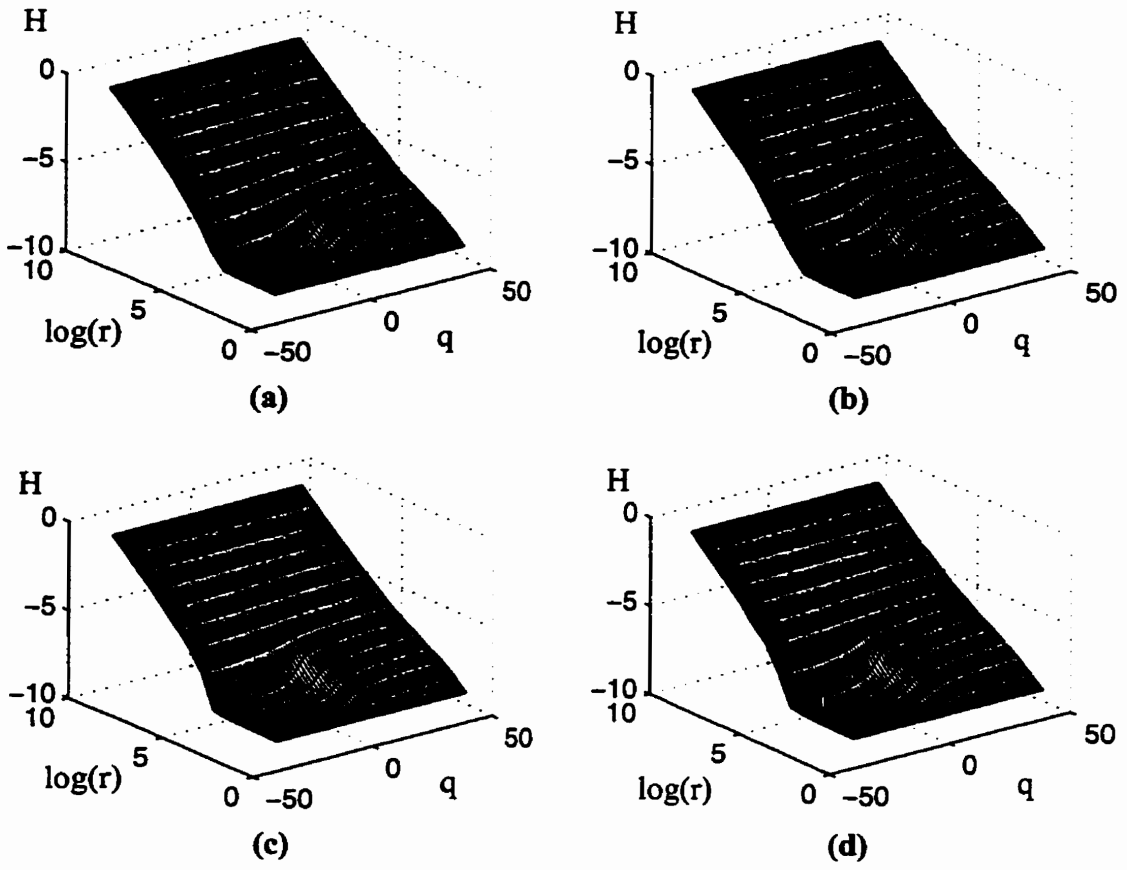


Fig. 4.2 Generalized entropy of random sequences for the four bases of DNA (a) adenine, (b) cytosine, (c) guanine and (d) thymine.

4.1.2 Natural Whole Sequence Tests

The generalized entropy, calculated using the entire sequences of *E. coli* and *M. jannashii*, is shown in Figure 4.3. Several features become evident in viewing the results of natural sequences:

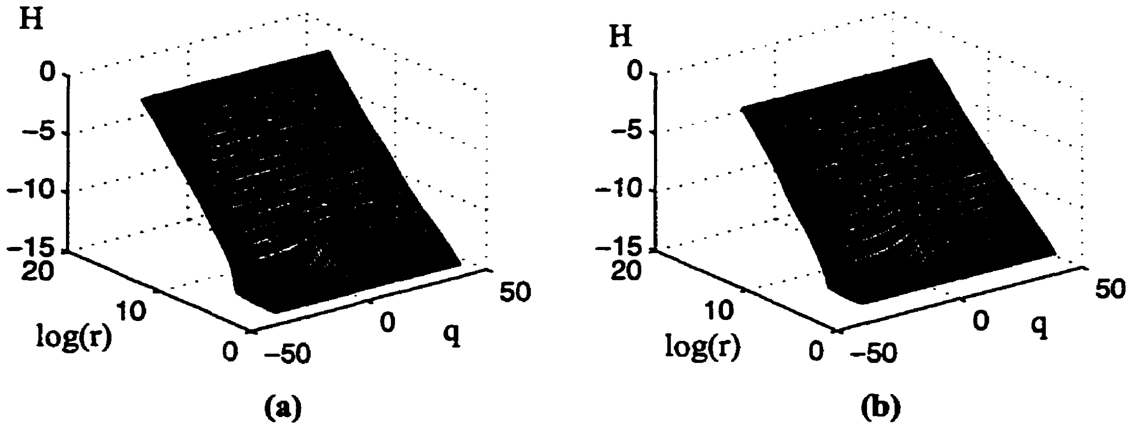


Fig. 4.3 Generalized entropy of adenine for the entire sequence of (a) *E. coli* and (b) *M. jannashii.*

- There is a flattened portion for small scales ($r < 50$) and negative moment orders ($q < 0$). This region appears for all SBSA generalized entropy calculations in natural and random DNA sequences. Because of the sifting property of negative norms, the results tend toward the smallest probability which for sequences smaller than 50 base pairs, is usually zero. The reason for a minimum probability of zero is that it seems for vel sizes up to fifty base pairs, it is likely that at least one vel will not have any instances of the targeted base. As a result, for small vel sizes, outliers occur for negative moment orders.
- As the scales increase, there is a tendency for the entropies to become equal across the values of q. That is, the value of q seems to have no effect for scales larger than $r > 2^{12} = 4096$. This feature appears for all generalized entropy calculations for natural DNA sequences.
- All of the bases exhibit similar behavior. When looking at the four entropies generated by a single sequence, all have a common shape, and range over the same entropy values.

4.2 Selecting Vel Sizes

A direct comparison of the generalized entropies calculated using different dyadic bases shows that they follow the same slopes. The use of a different dyadic base does not affect the slope entropy for the same region, but has a slight modification to the entropy itself.

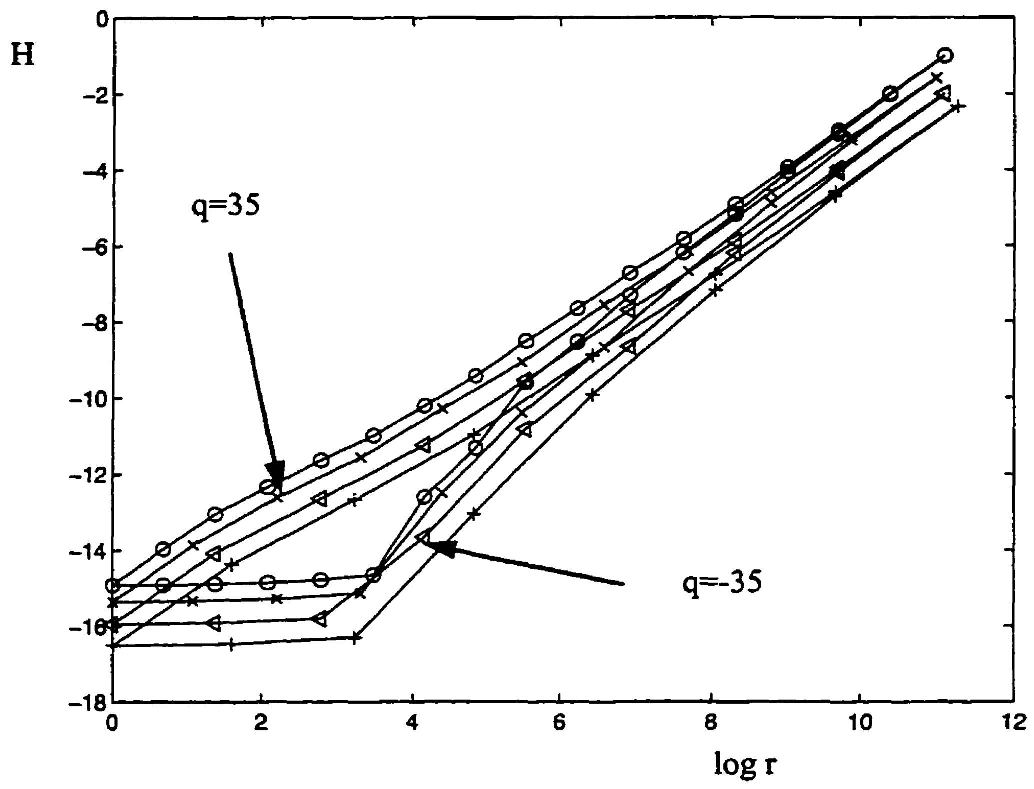


Fig. 4.4 Entropy of E.coli using different dyadic scalings. Circles are a scaling of two, crosses have scale three, diamonds have scale four, and plus signs have scale five.

4.3 Selecting Scales for Regression

In order to effectively determine the effects of the range of scales on the multifractal dimension calculation, a full range of possibilities is calculated. Figures 4.5, 4.6, and 4.7 show that by selecting different starting and stopping ranges (R_{min}, R_{max}), the dimensionality has a wide variation.

It can be seen that as the vel sizes used increases, there is a converging effect, and the dimensions start to approach a fixed value for all q. As a result, for the SBSA, the scale range between 64 and 256 base pairs is chosen. By keeping the range away from the convergent area, the calculated dimensions should provide the maximum signature for any given DNA sequence.

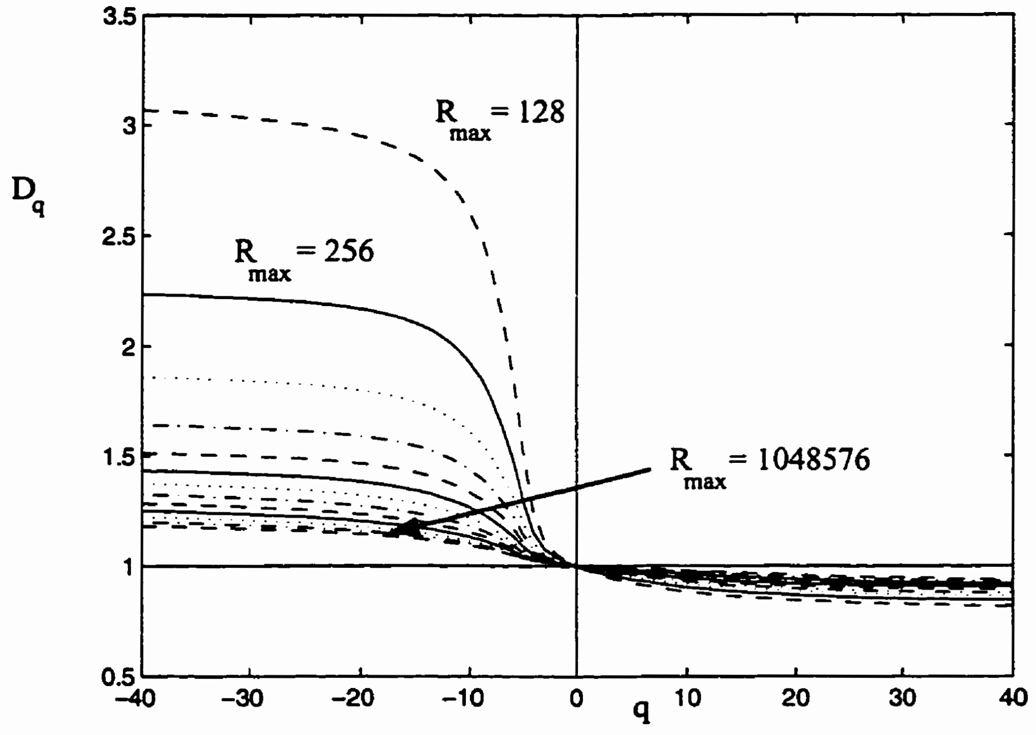


Fig. 4.5 SBSA calculation of D_q for *E. coli* (adenine) using R_{min} = 64 and varying R_{max} from 128 to 1048576.

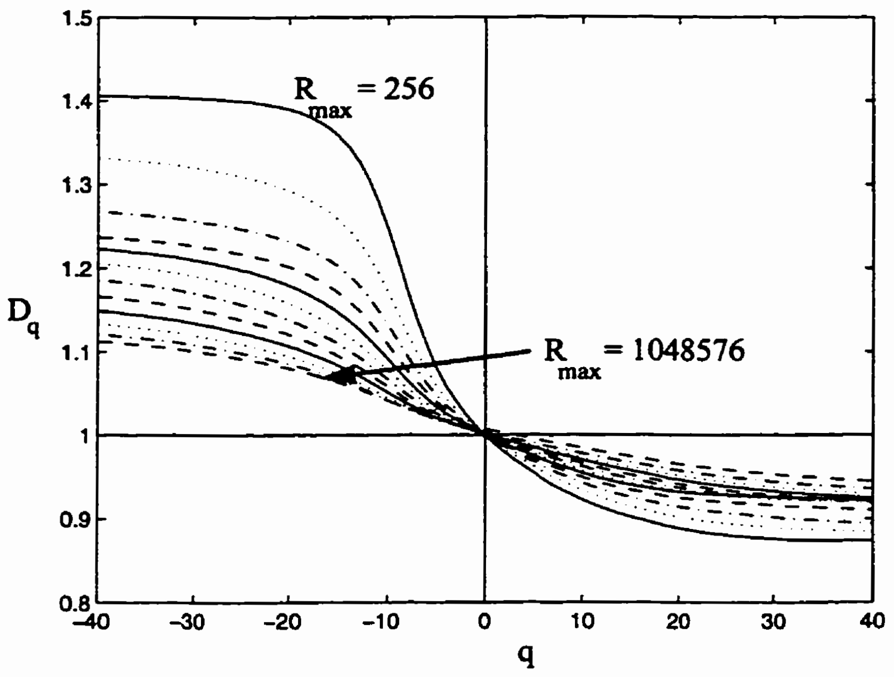


Fig. 4.6 SBSA calculation of D_q for *E. coli* (adenine) using R_{min} = 128 and varying R_{max} from 256 to 1048576.

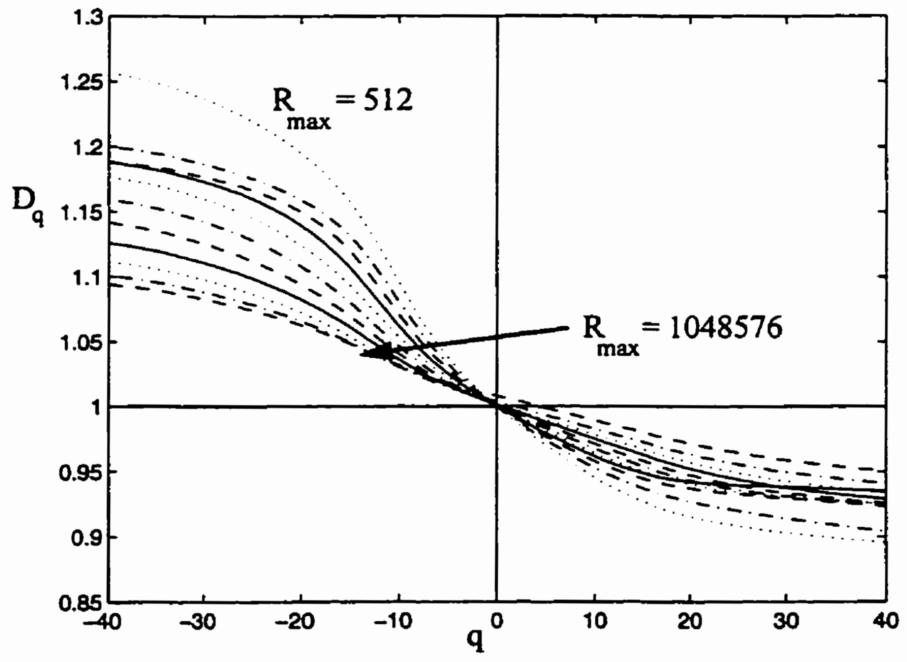


Fig. 4.7 SBSA calculation of D_q for *E. coli* (adenine) using R_{min} = 256 and varying R_{max} from 512 to 1048576.

Because of this choice of scales, the smallest window of DNA which can be examined using the single base dust technique is 256 bp. In order to verify this, Figure 4.8 shows how the multifractal spectrum varies with window size and constant scale ranges. The calculated dimensions using 256 bp vary significantly, while the larger window sizes are relatively similar. The practical impact of this is that for scale ranges of 64 to 256 bp, the minimum window size needed for estimation is 512 bp. The slight differences between the larger window sizes can be attributed to differences in each part of the sequence, and might be useful in characterizing each of the sequences.

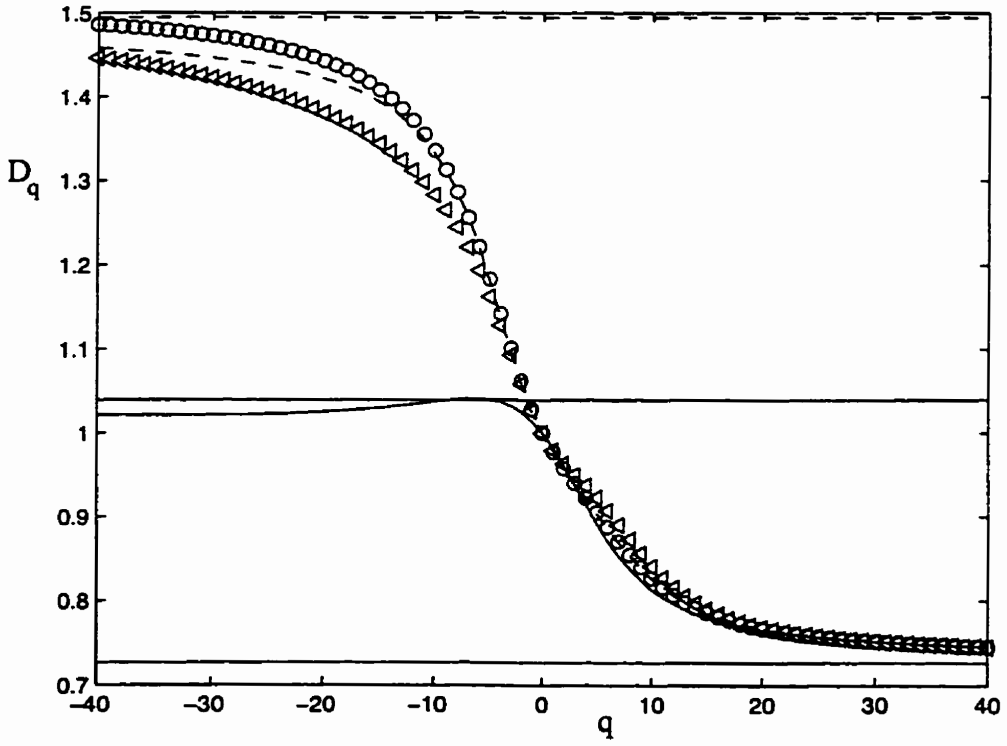


Fig. 4.8 Rényi Dimension of *E. coli* using varying window sizes. Solid Line for 256bp, Dash for 512bp, Diamonds for 1024bp, and Circles for 2048bp.

Of particular interest is that observation that for $q = 0$, the morphological dimension D_0 is always one. If we re-examine (2.7) for $q = 0$, we get

$$D_0 = \lim_{r \to 0} \frac{1}{q-1} \frac{\log \sum_{i=1}^{N_r} p_i^q}{\log(r)} = \lim_{r \to 0} \frac{\log N_r}{\log r} \tag{4.1}$$

Further examination shows that the number of vels, N_r, is proportional to the scale, r. As a result, the limit in (4.1) approaches unity for any sequence in which at least one target symbol appears in every vel. This observation relates very well established theory, since the DNA sequence is an object which exists rigidly within one dimension. Since the morphological dimension does not take into account the distribution of an object over the covering vels, it will always be one since the number of covering vels changes with scale.

4.4 Global Calculations of the Generalized Dimension

Several experiments were performed which give comparative results of the multifractal dimension. Figure 4.9 shows how the fractal dimension of various partitions of *E. coli* differ from each other. As is visible, there is significant variety at higher and lower moment orders. Figure 4.10 shows how the fractal dimension of various partitions of *M. jannashii* are related. There is much more similarity between the regions of *M. jannashii* than *E. coli*. Breaking the sequence even further, however, might show some variety which is hidden for these particular regions. Choosing smaller windows would provide a much better view of the "locality "of multifractal complexity, which might give an indication of the biological function.

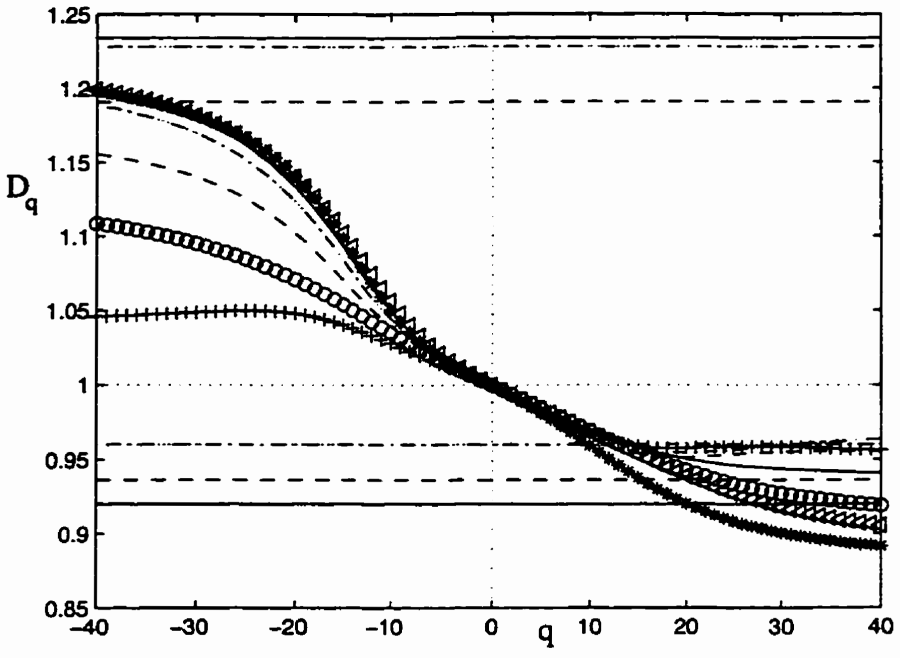


Fig. 4.9 SBSA Rényi dimensions of different regions of *E. coli*. (a) The entire sequence is the dash-dotted line, and the max and min are given. (b) The first half is solid, and second half is dashed, and the max and min values appear. (c) The quarters are given by diamonds, circles, plus signs, and stars respectively, and the min and max are not shown.

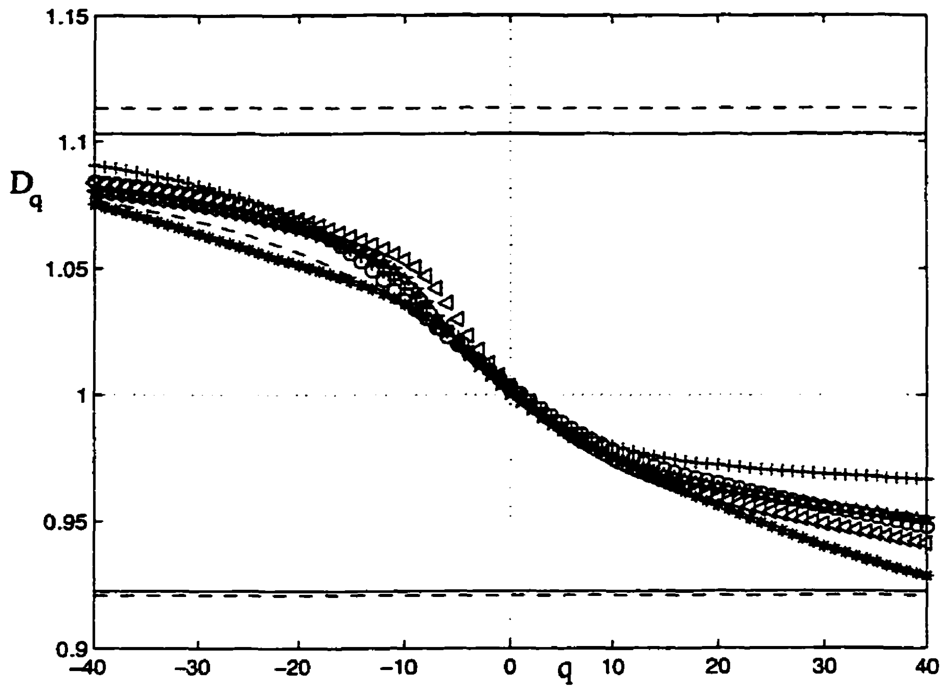


Fig. 4.10 SBSA Rényi dimensions of different regions of *M. jannashii.* (a) The entire sequence is the dash-dotted line, and the max and min are given. (b) The first half is solid, and second half is dashed, and the max and min values appear. (c) The quarters are given by diamonds, circles, plus signs, and stars respectively, and the min and max are not shown.

4.5 Summary

The SBSA multifractal measurement techniques show that there are indeed relationships between various scales, indicating that an isolated single base possesses fractal properties. As well, guidelines for the scales which should be used for regression and the minimum window size were established. There does not appear to be an explicit relationship between a sequence and its parts in the Rényi spectrum domain, although there is a large variation which indicates that the multifractal properties of DNA vary between its different segments.

CHAPTER V
EXPERIMENTAL RESULTS FOR MULTIPLE BASE SEQUENCE ATTRACTORS

The multiple base sequence attractor technique of analyzing DNA is explored in this chapter. Synthetic and natural sequences are analyzed, looking for a power-law relationship between the various scales. In order to properly apply the Rényi spectrum, certain adjustments are made within the power-law framework to provide a consistent approach.

As with the SBSA analysis, the scales for regression and window sizes are selected. Along with the *E. coli* and *M. jannashii* sequences, an experiment in mitochondrial phylogeny is performed, which duplicates the results obtained in a different multifractal analysis technique.

5.1 Rényi's Generalized Entropy Calculation

5.1.1 Synthetic Sequence Tests

Experiments were performed which show what the calculated MBSA entropies of synthetic sequences are. In Fig. 5.1, we can see that for non-random sequences, the size of the blocks do not affect the calculation of entropy. For a sequence which consists of only one base, the number of bits of information is zero since there is essentially no information being transmitted. A sequence which has a repeated structure has an entropy of two, since only two bits are required to send the entire sequence.

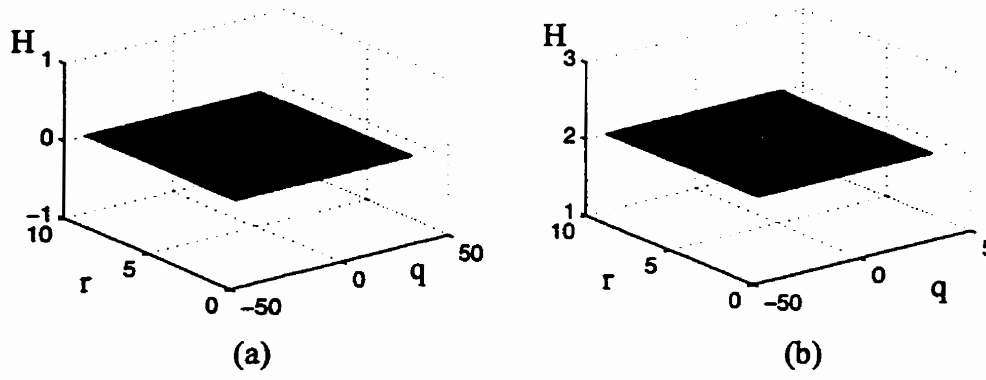


Fig. 5.1 Entropies of Synthetic Sequences (a) with all Adenine, and (b) with a repeating pattern of AGCT.

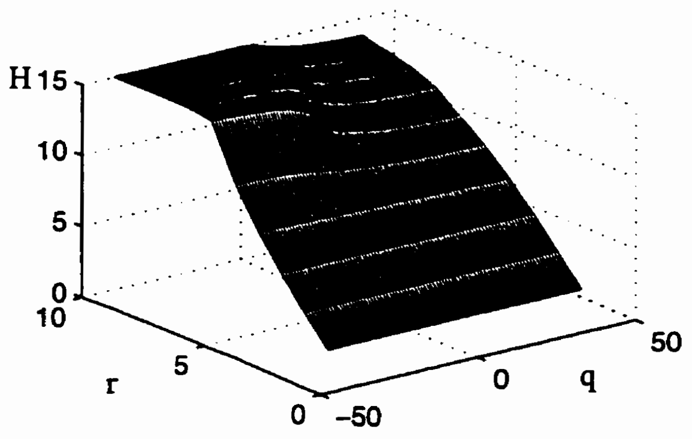
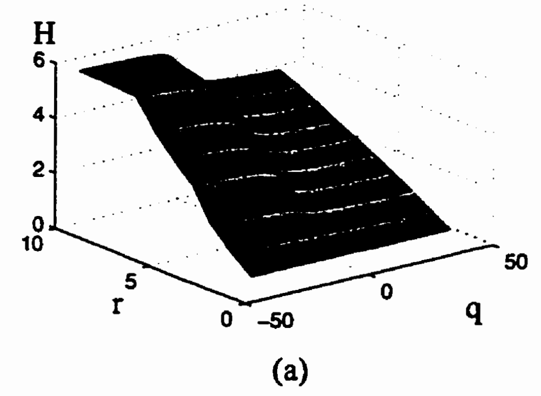


Fig. 5.2 Entropy of Random Sequence of DNA with each base having equal probability of occurring

The entropy of a random DNA sequence is shown in Fig. 5.2. A flat portion exists for scales past six. The random sequence used was of length 2^{20} bases. For vels greater than six, the probability estimates were not accurate enough for a sequence of this length. A longer sequence would be needed. From the vels which were estimated properly, a linear relationship exist between the entropy and the vel size. This is unexpected, since for calculations of fractal dimension, we require a relationship between entropy and the logarithm of vel size, i.e. a power-law relationship.

5.1.2 Natural Whole Sequence Tests

Figure 5.3 shows the entropies calculated for the entire sequences of *E. coli* and *M. jannashii*. There clearly is a power-law relationship, but the larger vels clearly diverge due to inaccurate probability estimation.



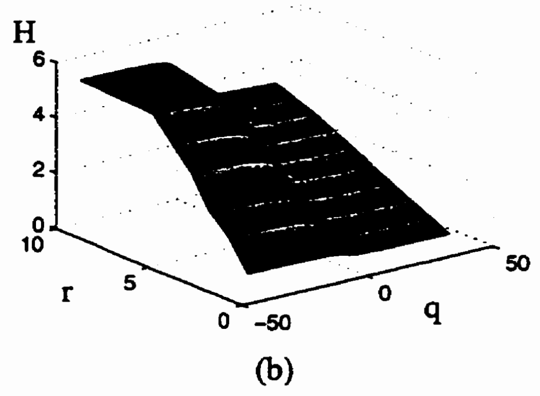


Fig. 5.3 MBSA Entropies over the entire sequences of (a) *E. coli*, and (b) *M. jannashii*

An important item which needs to be determined is the minimum window size which can be used to accurately estimate the probability density functions for the calculations of generalized entropy. We expect that as the window size increases, the estimate of the probabilities used in calculating entropy should converge to a particular value. It should be possible to estimate the minimum window size needed for any given choice of vel size, *r*. From Fig. 5.4, a logarithmic relationship exists between the minimum window needed and the vel size. The plot shows how when the window size in increased, the entropies eventually stabilize. The relationship takes the form of

$$w_{min} = c_1 2^{c_2 r} \tag{5.1}$$

where c_1 and c_2 can be estimated from the data. A further complication is revealed by examining Figs. 5.5 and 5.6. The minimum window size needed clearly depends on the moment order, q. For smaller values of q, c_1 and c_2 increase, while for larger values of q, c_1 and c_2 decrease. Re-examining (2.6), we can see that for negative moment-orders, smaller probabilities are emphasized, while for positive moment-orders, the larger probabilities are emphasized. This effect is related to the sifting property of infinite norms discussed in Section 2.3.3. The overall effect is that in order to estimate the minimum window size, the smallest value of q to be calculated must be considered.

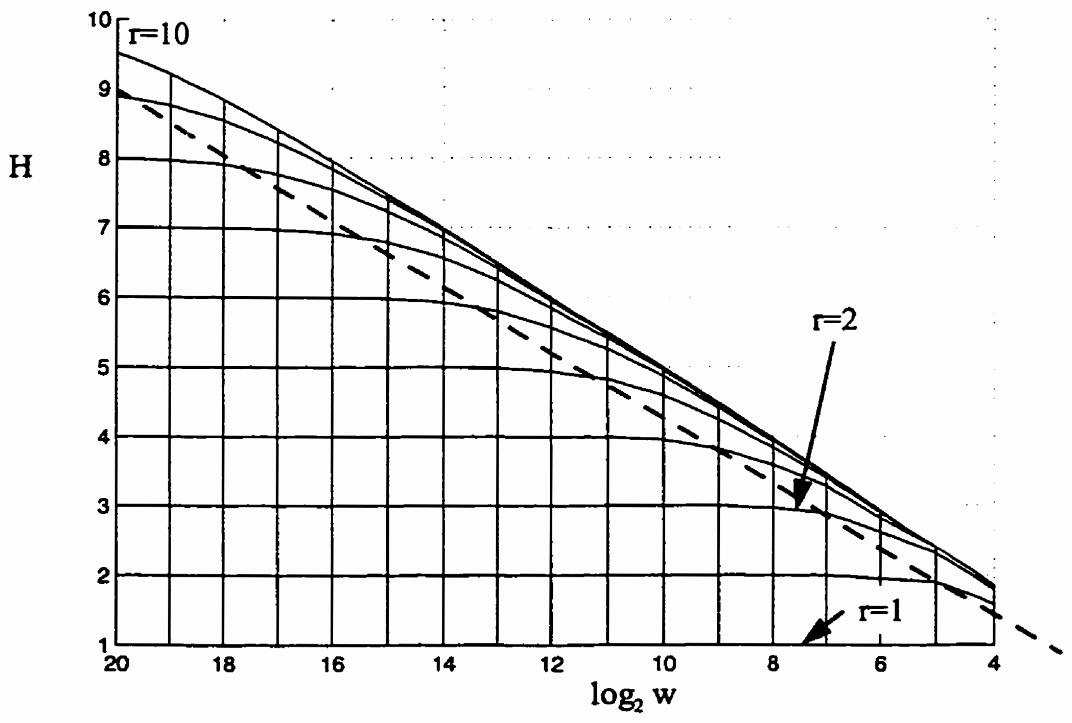


Fig. 5.4 Entropy versus Window size for q=0. Vel sizes increase from 1 to 10.

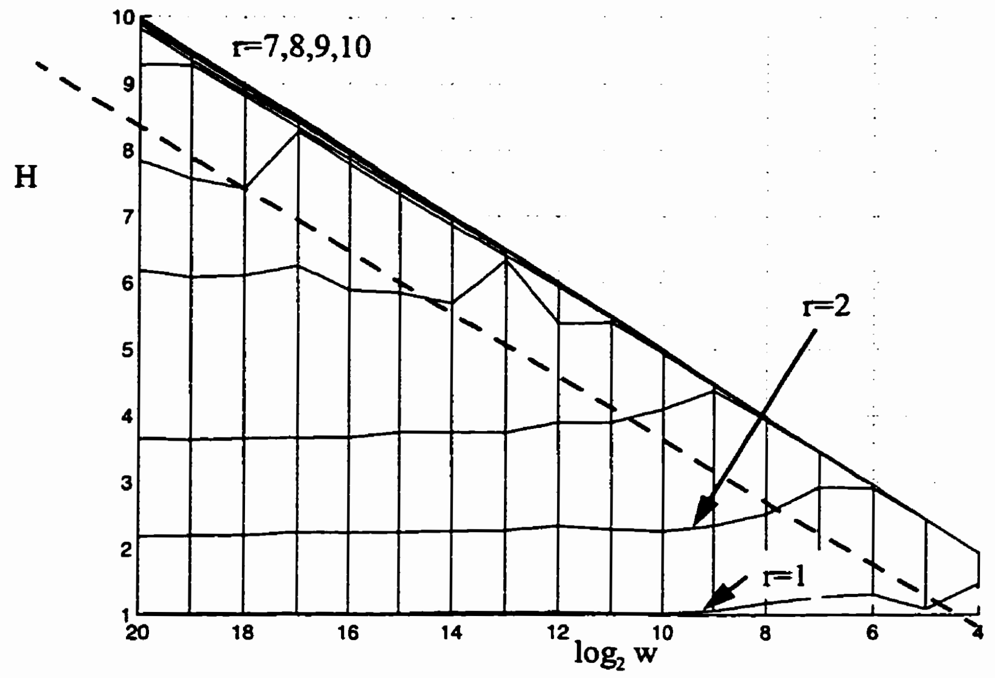


Fig. 5.5 Entropy versus window size for q=-38. Vel sizes increase from 1 to 10.

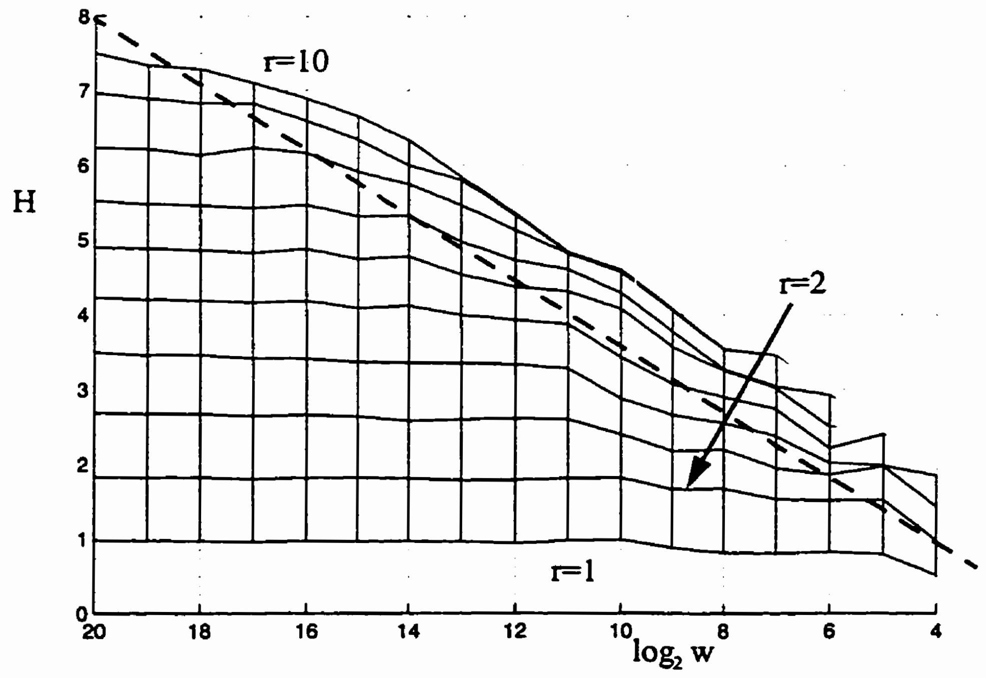


Fig. 5.6 Entropy versus window size for q=38. Vel sizes increase from 1 to 10.

In Fig. 5.6, it can be seen that the estimates of entropy vary widely as the window size in increased, and that as the window size increases past a critical value, the values converge. An idea which will be explored is that the changing estimates of entropy are very significant of the structure of the underlying DNA sequence, and that using the smaller vel sizes will give localized information about the sequence, depending upon the position of the window. It can be argued, that when the window size increases too much, any local detail is averaged out with detail from other local sections which have been included in the larger window size. This hypothesis provides an explanation of why the entropy values converge so convincingly at higher window sizes. The averaged entropy does not differ along the sequence, while the local values have significant fluctuations. It is the use of the local fluctuations which will give the best results in terms of localizing the multifractal characteristic, and relating it to known underlying biological information.

5.2 Selection of Vel Sizes

Figures 5.7 and 5.8 illustrate what appears to be a problem facing the calculation of D_q. The graphs show that as the vel size increases, for a constant window size, there is some point where the entropy is no longer estimated correctly. More information is added with each larger window until the final value is reached for any particular vel size.

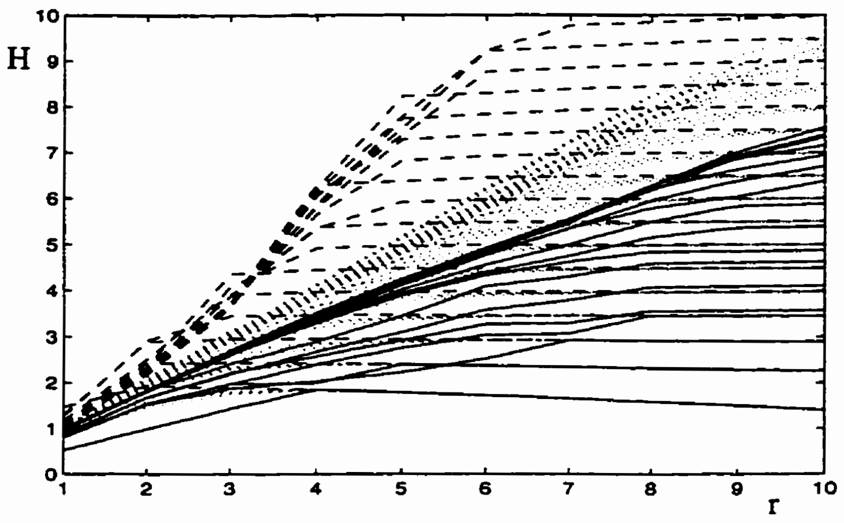


Fig. 5.7 MBSA Entropies for increasing window sizes of *E. coli* and varying *q*. For q=-30, the dashed lines show increasing window sizes. For q=-1,0,1 the dotted lines indicate the log-log plot. For q=30, the filled lines show increasing window size.

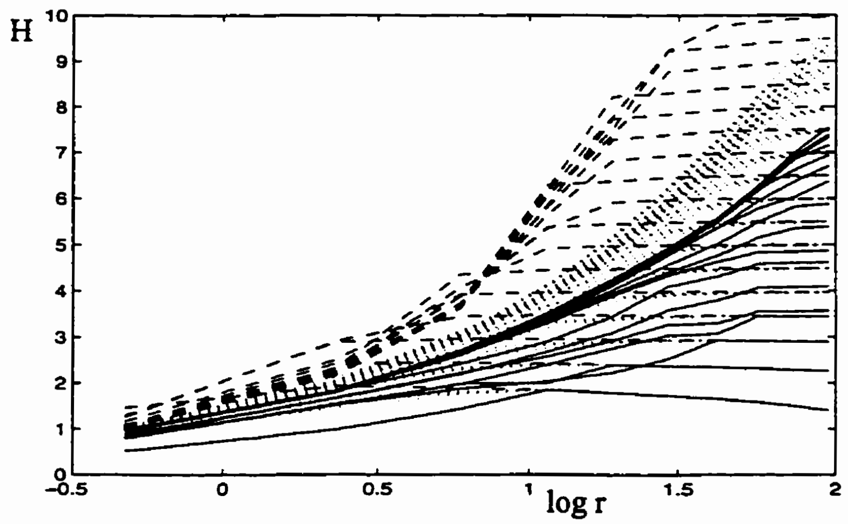


Fig. 5.8 MBSA Entropies as in Figure 5.7, but with logarithmic scaling along the horizontal (scale) axis.

There seems to exist a linear relationship between the vel size and the entropies, while the idea presented for fractal dimensions indicate that a power-law relationship is expected. While this fact was not predicted, it can be explained. Instead of considering the scale to be r, we can consider it to be N_r. The relationship between these two values is given in (3.2). Our definition of the Rényi dimensions becomes

$$D_q = \lim_{r \to 0} \frac{1}{q-1} \frac{\log_2 \sum_{j=1}^{N_r} p_j^q}{\log_2 N_r} = \lim_{r \to 0} \frac{1}{q-1} \frac{\log_2 \sum_{j=1}^{N_r} p_j^q}{2r} \tag{5.2}$$

From this relationship, all of the experimental behavior fits properly. The value of D_0, the box-counting dimension, becomes one which is consistent with the idea of morphological dimensions (it is always 1). As a result, a previous disadvantage (Section 3.1.2) regarding the range of vel sizes has been resolved.

5.3 Selecting Scales for Regression

Unlike the SBSA entropies, there are no outliers at small scales. At the larger vel sizes, the probability distribution is not estimated accurately, resulting in entropies with errors. The maximum vel size is then chosen such that the probability at that scale is estimated properly. For a vel size of 4, the minimum window size is around 8096 bases, while for a vel size of 3, the minimum window size is around 1024 bases. In order to maximize the possible locality, we will choose that $R_{min} = 1$ and $R_{max} = 3$ for all of the multifractal calculations performed using MBSA analysis for the remainder of the thesis.

Figures 5.9 to 5.11 show how the Rényi dimension changes for various scale ranges. The choice of vel sizes is validated by these plots which show that by using only the first three scales, the results are still comparable to including larger vel sizes for which the probabilities are still accurate. In particular, in Fig. 5.9, the first four estimations of D_q, are so closely packed, that we might consider only using the first two scales to estimate the fractal dimensions.

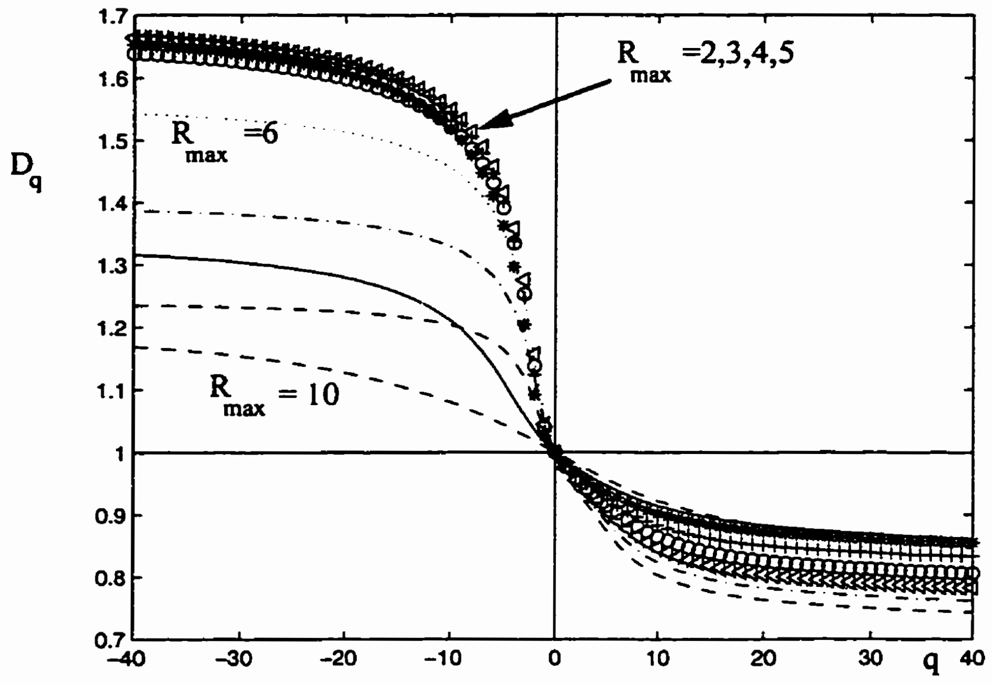


Fig. 5.9 MBSA calculation of D_q for *E. coli* using $R_{min} = 1$ and varying R_{max} from 2 to 10.

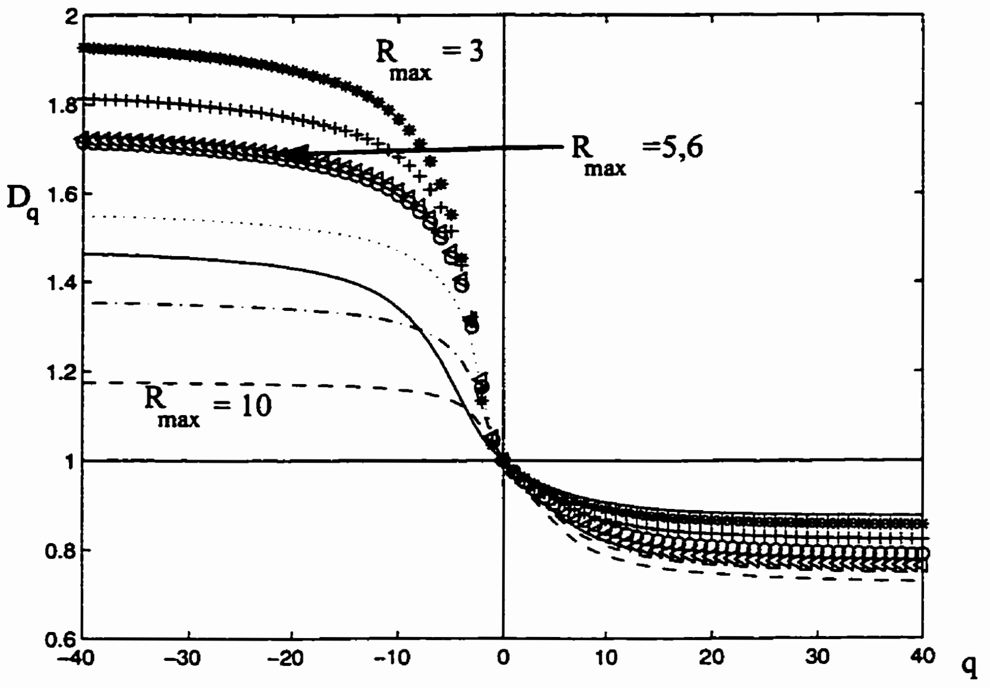


Fig. 5.10 MBSA calculation of D_q for *E. coli* using R_{min} =2 and varying R_{max} from 3 to 10.

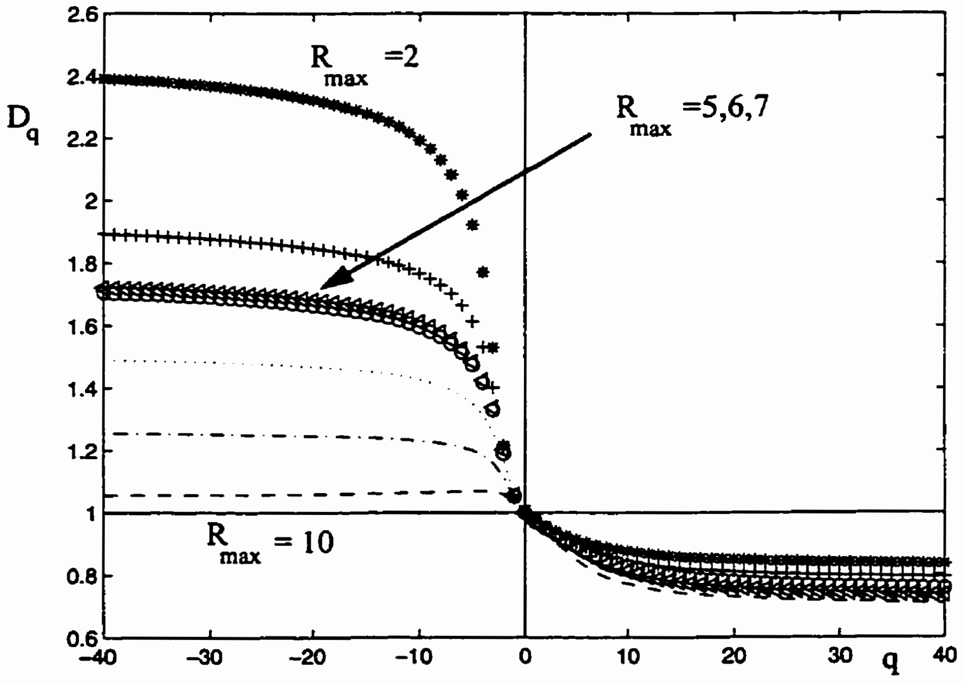


Fig. 5.11 MBSA calculation of D_q for *E. coli* using R_{min} =3 and varying R_{max} from 3 to 10.

5.4 Global Calculations of the Generalized Dimension

As in the previous chapter, we can examine what happens when we analyze the fractal dimensions of parts of the DNA sequence as compared to the whole. Figures 5.12 and 5.13 show how the halves and quarters of *E. coli* are related. It appears that when examining a sequence and its halves, the fractal dimension of the sequence lies exactly halfway between the dimensions of the halves. The same relationship is observed for all of the quarters and can be examined mathematically.

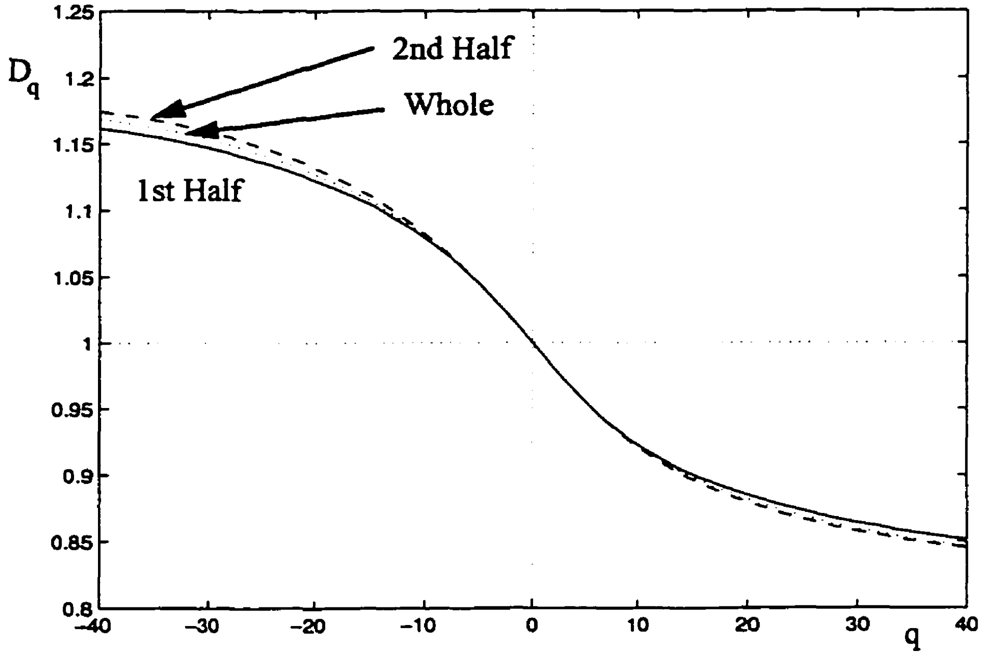


Fig. 5.12 MBSA Rényi dimensions of different regions of E.Coli. The dotted line is the whole sequence, the first half is solid, and the second half is dashed.

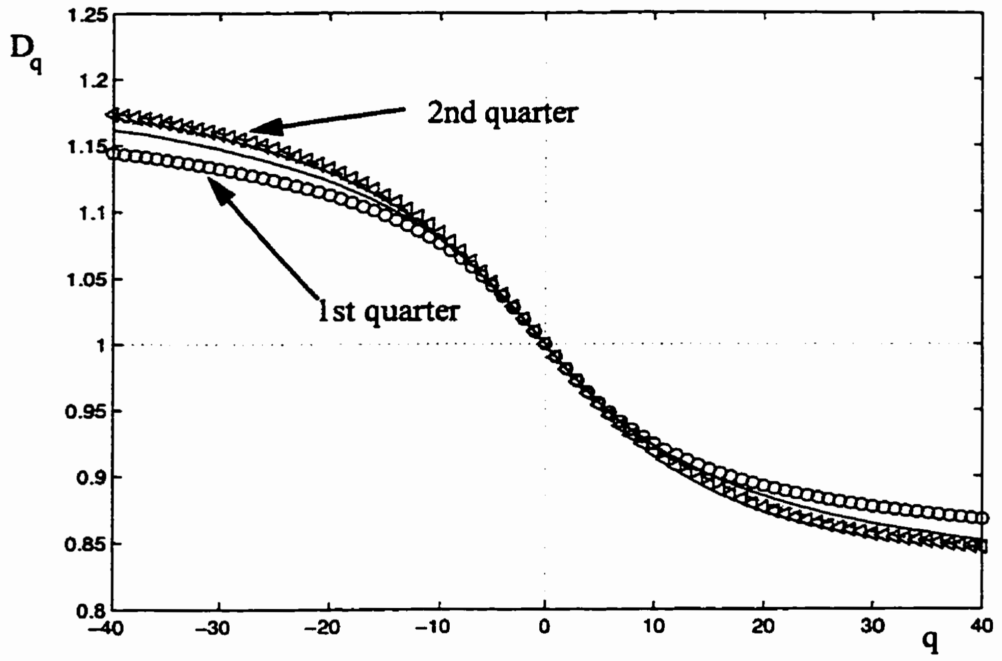


Fig. 5.13 MBSA Rényi dimensions of different regions of E.Coli. The dotted line is the whole sequence, the first half is solid, the second half is dashed (as in Fig. 5.12), the first quarter is circles, and the second quarter is plus signs.

The definition of probability, (3.3), can be modified such that for the first, second, and whole sequences, we have

$$p^{(h1)} = \frac{n^{(h1)}}{N/2} \tag{5.3}$$

$$p^{(h2)} = \frac{n^{(h2)}}{N/2} \tag{5.4}$$

$$p^{(w)} = \frac{n^{(w)}}{N} = \frac{n^{(h1)} + n^{(h2)}}{N} = \frac{p^{(h1)} + p^{(h2)}}{2} \tag{5.5}$$

where $n^{(h1)}$, $n^{(h2)}$, and $n^{(w)}$ are the counts in the first half, second half, and whole sequence. From (5.5) it seems that every MBSA Rényi dimension curve should appear exactly averaged between its first and second halves. The moment-order, q, plays a part in determining the effect of the averaging. For moment orders close to zero, the averaging disappears, and the there is no difference between the three curves. For larger (or smaller) orders, the sifting property (Section 2.3.3) tends to pull out the averaging feature, which is evident in the results.

This averaging property shows the significant drawbacks which can be associated with computing only global dimensions. Pieces of the DNA where the fractality might be extremely varied are absorbed into the global calculations, and are ultimately ignored.

5.5 Mitochondrial Phylogeny

The MBSA multifractal spectrum was calculated for a group of mitochondrial DNA sequences (from [GBRS95]). As outlined in Section 3.5, a comparison of the results is desirable. Figure Fig. 5.14 shows the multifractal curves of the various mtDNA sequences. It is particularly striking how there appear to be groupings of similar organisms. There are very clear distinctions between insects, sea urchins, and the rest of the organisms. This provides a link between the evolution of organisms and complexity of their DNA sequences. It should be noted that there appear to be several organisms which are not of the same taxonomy which have closely related multifractal spectrums.

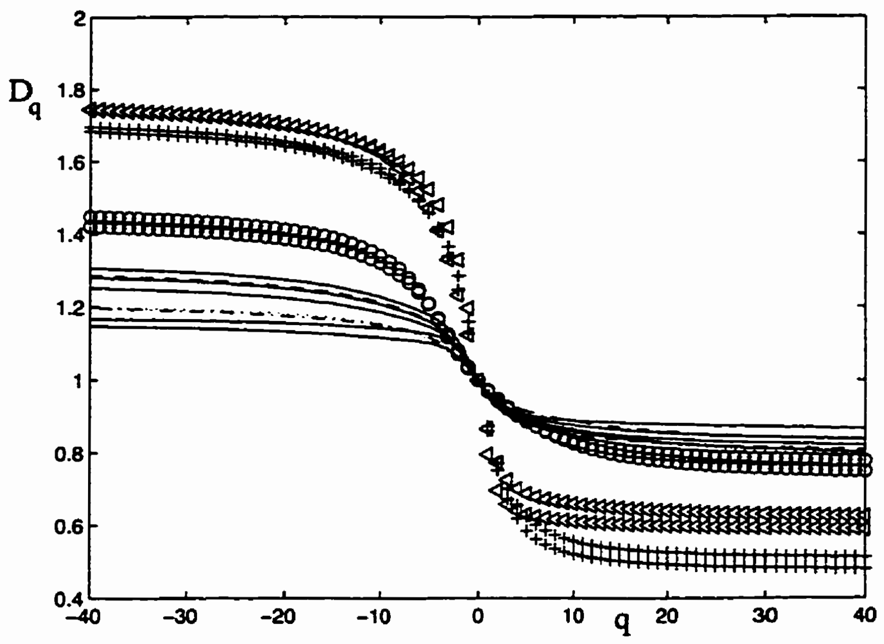


Fig. 5.14 Shows comparison of D_q curves with rmin=1,rmax=3.insect (Δ), nematode (+), urchin (o), mammal (-), toad (--),carp (-.),fish (..).

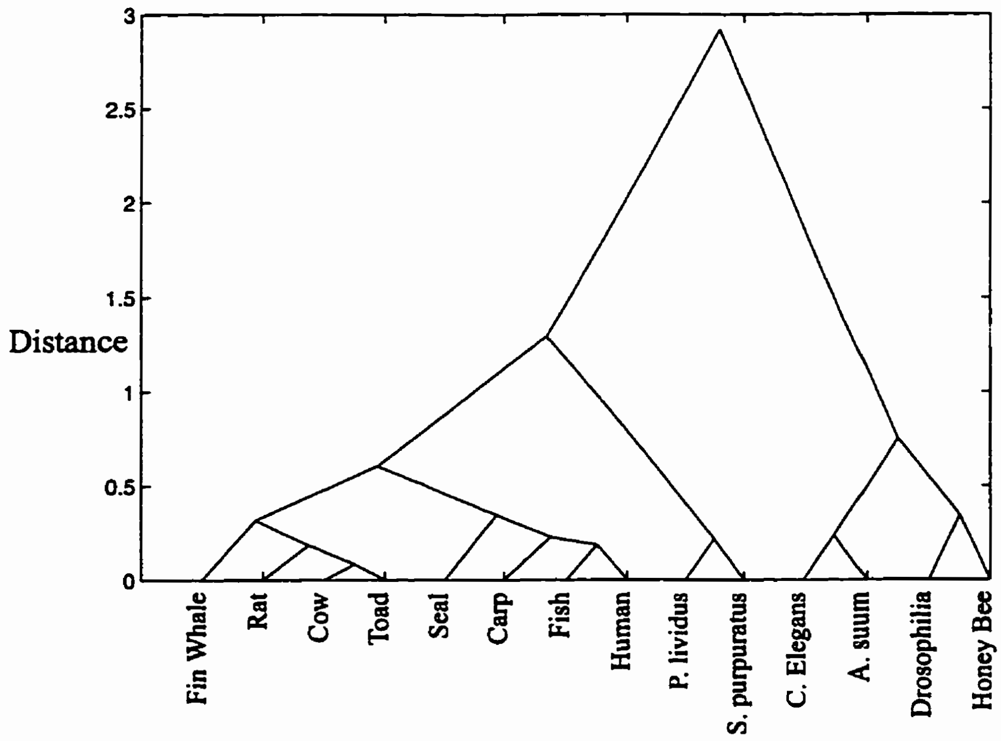


Fig. 5.15 Dendogram classification using cluster analysis of D_q curves from mitochondrial DNA.

Figure 5.15 shows a classification of the multifractal spectrums from the mitochondrial DNA sequences. The majority of classifications relate to known evolutionary patterns, but two glaring misclassifications occur with humans and seals. These two mammals are group with the fish. As well, the toad should be related to the mammals at a higher level.

5.5.3 Special Consideration

The Mandelbrot spectrum of dimensions, shown in Fig. 5.16, provides an alternative representation of the complexity of DNA sequences. There are several anomalies which are clearly visible from this diagram. First of all, there are anomalies near the edges of the Mandelbrot spectra. These anomalies have been encountered in previous research, and are based on the numerical properties of the underlying implementation.

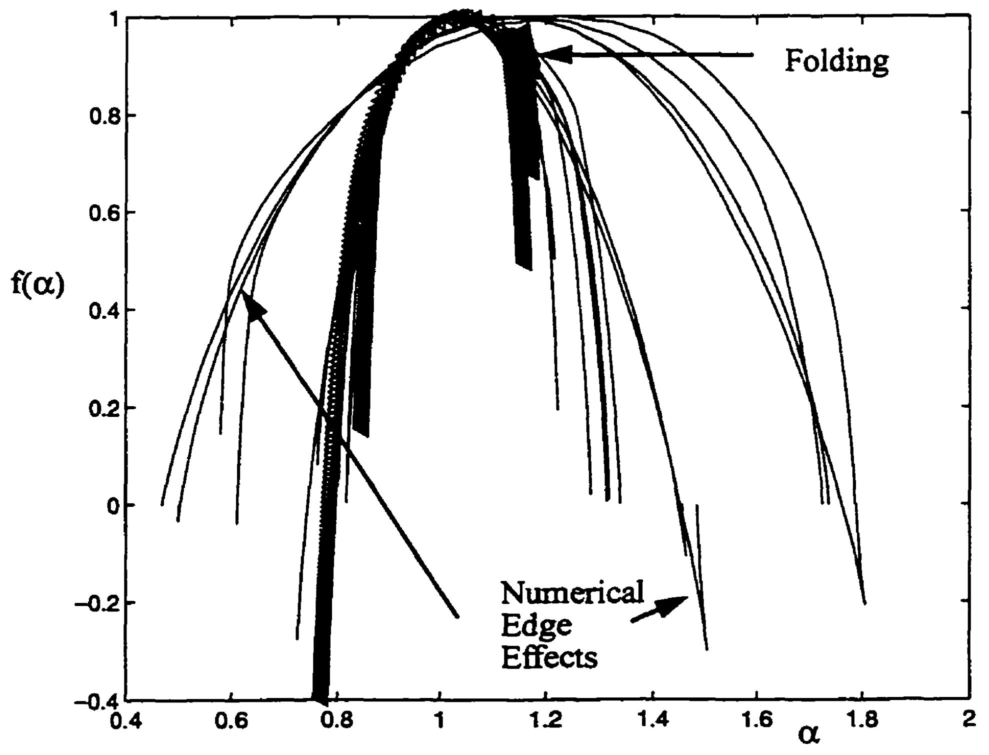


Fig. 5.16 Comparison of D_{man} curves using a scale range 1 to 3 bp. All curves are shown by lines except human and seal which use triangles.

Secondly, there is a folding behavior which appears only for the human and seal spectra. Figure 5.17 shows a close-up of the multifractal spectra behavior. There appears to be a folding back in the curve, then a continuation of the normal behavior. Note that only two out of the fourteen curves exhibit this characteristic. By changing the scale range, different curves produce the same folding property. This property is not related to the numerical edge effects because q ranges from -10 to 0 over the folded interval. This curve is being presented with reservations, because we are unsure of the nature of the folding property. It is interesting to note that the behavior occurred for the two organisms which had been misclassified as to taxonomic class.

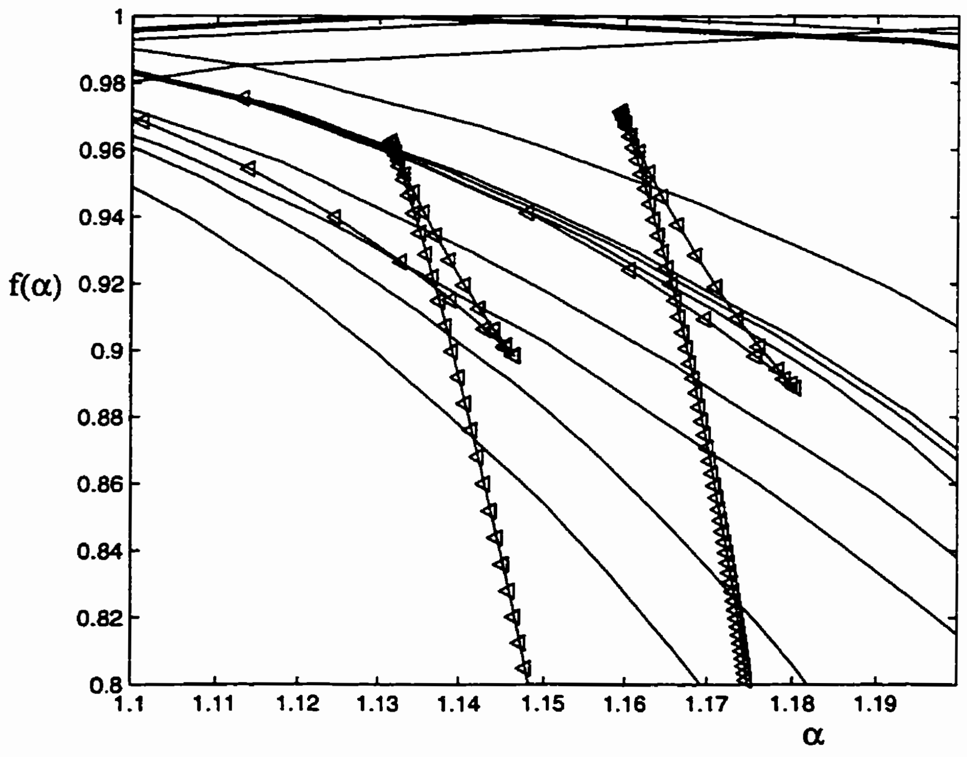


Fig. 5.17 Zoomed view of Fig. 5.16.

If not for the folding properties which have been observed, we would recommend classifying using the Mandelbrot spectrum. From Fig. 5.16, the various classes are still visible and are somewhat highlighted.

5.6 Summary

The multiple base sequence attractors are shown to be a very powerful analysis technique for DNA, which exhibits a power-law relationship. It is important to consider a slight change in the concept of scale which provides us with consistent behavior and accurate results. As well, the minimum window size, and scales for regression were outlined.

Of particular importance is the relationship of a sequence to its constituent parts, which was shown to be an average of high and low moment orders. This analysis also highlights the fact that DNA seems to have different characteristics at different parts of the sequence.

The results using mitochondrial DNA show that while the Rényi dimension has some correct features, the Mandelbrot dimension can also highlight the differences between different phylogenetic classes very well. Unfortunately, due to a folding property whose source is unknown, the Mandelbrot spectra cannot be used for classification.

CHAPTER VI
CONCLUSIONS, RECOMMENDATIONS, AND CONTRIBUTIONS

6.1 Conclusions

In this thesis, a framework for the analysis of DNA sequences using multifractal techniques was developed. Two definitions of multifractal dimension, single base subsequence attractors (SBSA) and multiple base sequence attractors (MBSA) of DNA sequences were presented and explored, both of which show that natural DNA sequences exhibit multifractal behavior in the form of non-integer dimensionality, and non-uniform multifractal spectra.

The SBSA approach showed that the components of DNA, when treated separately exhibit power-law relationships and multifractality. The multifractality was shown to be related to the position of the window within a DNA sequence for E. coli and M. jannashii. A scale range of 64 to 256 base pairs, a dyadic scaling of two, and a minimum window size of 512 base pairs was shown to be sufficient for accurately estimating the multifractality of E. coli and M. jannashii.

The MBSA approach showed that the information content of DNA sequences exhibits a power-law relationship and multifractal behavior. The multifractality of a larger sequence was shown to consist of an average of the constituent parts of sequence. As well, the multifractality was shown to be varied for different windows of E. coli and M. jannashii. A scale range of one to three, and a minimum window size of 1024 was shown to be sufficient for estimating the multifractal spectrum of E. coli, M. jannashii, and mitochondrial

DNA sequences. Mitochondrial DNA sequences from different organisms were classified according to taxa using the MBSA method with two misclassifications.

6.2 Contributions

This thesis has made the following contributions:

- A technique for the estimation of the multifractality of the individual separate bases of DNA including study of the minimum window sizes and scale ranges necessary for accurate estimations;
- A technique for the estimation of the multifractality of composite DNA including study of the minimum window sizes and scale ranges necessary for accurate estimations;
- A study of how portions of a DNA sequence are related to the whole sequence using multifractality;
- A verification of the results that mitochondrial DNA can be used to classify organisms according to evolutionary and taxonomic groups;
- A software system which estimates the two multifractal measures (SBSA and MBSA) simultaneously.

6.3 Recommendations

Based on the research conducted in this thesis, recommendations are as follows:

- Both of the multifractal techniques should be applied to a larger group of DNA sequences to determine if there are DNA sequences for which the results of this thesis do not apply.
- A multifractal spectrum trajectory using adjacent windows should be analyzed to determine if local complexity can be used as a feature for DNA classification.
- The Mandelbrot spectrum of human and seal mitochondrial DNA should be re-examined to determine the source of the folding behavior observed.

- The single-base subsequence attractor analysis should be examined for redundancy, and if possible compressed.
- The software developed in this thesis is flexible enough to be applied to other signals such as text which have a discrete symbol structure.
- An automated system for determining the minimum window size, and valid ranges for the scales in linear regression could be developed.

REFERENCES

[ABGW95] P. Allegrini, M. Barbi, P. Grigolini, and B. J. West, "Dynamical model for DNA sequences," *Phys. Rev. E*, vol. 52, no. 5, pp.5281-5296, 1995.

[BeRO96] P. Bernaola-Galvàn, R. Román-Roldán, and J. L. Oliver, "Compositional segmentation and long-range fractal correlations in DNA sequences," Phys. Rev. E, vol. 53, no. 5, pp. 5181-5189, 1996.

[BeGS92] C. L. Berthelsen, J. A. Glazier, and M. H. Skolnick, "Global fractal dimension of human DNA sequences treated as pseudorandom walks," *Phys. Rev. A*, vol. 45, no. 12, pp. 8902-8913, 1992.

[BeGR94] C. L. Berthelsen, J. A. Glazier, and S. Raghavachari, "Effective multifractal spectrum of a random walk," *Phys. Rev. E*, vol. 49, no. 3, pp. 1860-1864, 1994.

[BGHP93] S. V. Buldyrev, A. L. Goldberger, S. Havlin, C.-K. Peng, H. E. Stanley, M. H. R. Stanley, and M. Simons, "Fractal Landscapes and Molecular Evolution: Modeling the Myosin Heavy Chain Gene Family," *Biophysical J.*, vol. 65, pp. 2673-2679, December 1993.

[Chen95] H. Chen, *Accuracy of Fractal and Multifractal Measures for Signal Analysis*, M.Sc. Thesis, Winnipeg, MB:Department of Electrical and Computer Engineering, University of Manitoba, 1997, 193 pp.

[Fede88] J. Feder, *Fractals*, New York, NY: Plenum Press, 1988, 283pp. { ISBN 0-306-42851-2}

[FrER96] J. Freund, W. Ebeling, and K. Rateitschak, "Self-similar sequences and universal scaling of dynamical entropies," *Phys. Rev. E*, vol. 54, no. 5, pp. 5561-5566, 1996.

[GRBS95] J.A. Glazier, S. Raghavachari, C. L. Berthelsen, M. H. Skolnick, "Reconstructing phylogeny from the multifractal spectrum of mitochondrial DNA," Phys. Rev. E, vol. 51. no. 3, pp. 2665-2668, 1995.

[Grie96] W. S. Grieder, *Variance Fractal Dimension for Signal Feature Enhancement and Segmentation from Noise*, M.Sc. Thesis, Winnipeg, MB:Department of Electrical and Computer Engineering, University of Manitoba, 1996, 369 pp.

[Jang97] E. Jang, *Compression of Fingerprints based on Wavelet Packet Decomposition and Fractal Singularity Measures*, M.Sc. Thesis, Winnipeg, MB:Department of Electrical and Computer Engineering, University of Manitoba, 1997, 206 pp.

[Kins94] W. Kinsner, *Fractal Dimensions: Morphological, Entropy, Spectrum, and Variance Classes*, Tech. Report, DEL94-4, Department of Electrical and Computer Engineering, University of Manitoba, Winnipeg, Manitoba, Canada, May 1994, 140 pp.

[Kins97] W. Kinsner, Fractal and Chaos Engineering, 24.721 Course Notes, University of Manitoba, 1997.

[Lang96] A. Langi, *Wavelet and Fractal Processing and Compression of Nonstationary Signals*, Ph. D. Thesis, Winnipeg, MB:Department of Electrical and Computer Engineering, University of Manitoba, 1996, 466 pp.

[Lewi97] B. Lewin, *Genes VI*, New York, NY: Oxford University Press, 1997, 1260 pp.

[PeJS92] H. Peitgen, H. Jürgens, and D. Saupe, *Chaos and Fractals: New Frontiers of Science*, New York, NY: Springer-Verlag, 1992, 894 pp. { ISBN 0-387-97903-4}

[PBGH92] C.-K. Peng, S. V. Buldyrev, A. L. Goldberger, S. Havlin, F. Sciortino, M. Simons, and H. E. Stanley, "Long-range correlations in nucleotide sequences," *Nature*, vol. 356, pp. 168-170, March 12, 1992.

[PTVF92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, New York, NY: Cambridge University Press, 1992, pp. 661-666.

[RiKM98] R. Rifaat, W. Kinsner, and P. McAlpine, "Multifractal analysis of DNA," *Canadian Conference on Electrical and Computer Engineering*, May 24-28, 1998, pp. 766-769.

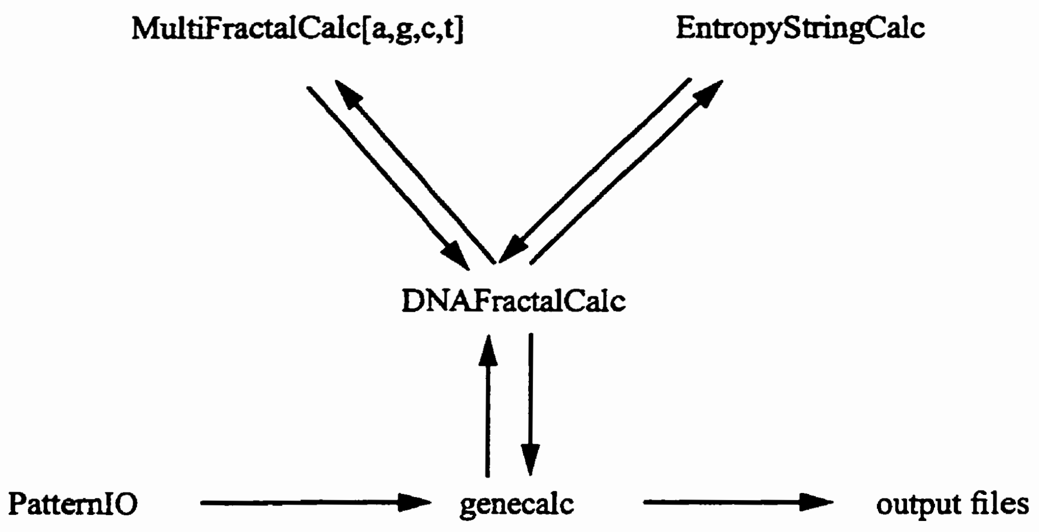
[ScHe97] Armin O. Schmitt and Hanspeter Herzel, "Estimating the entropy of DNA sequences," *Journal of Theoretical Biology*, no. 1888, pp. 369-377, 1997.

[Shaw97] D. B. Shaw, *Classification of Transmitter Transients using Fractal Measures and Probabilistic Neural Networks*, M.Sc. Thesis, Winnipeg, MB:Department of Electrical and Computer Engineering, University of Manitoba, 1997, 375 pp.

[Voss92] R.F. Voss, "Evolution of long-range fractal correlations and 1/f noise in DNA base sequences," *Phys. Rev. Lett.*, vol. 68, no. 25, pp. 3805-3808, 1992.

APPENDIX

A.1 Structure Chart



This chart shows the relationship between the modules of this program. genecalc acts as a shell to parse the command line, and pass data between the separate modules. DNAFractalCalc acts as a holder for the different fractal calculations so that the data passing can occur simultaneously and transparently.

A.2 Source Code

A.2.1 genecalc.cc

```
/* Genecalc.cc

   This program does multifractal dimension calculations on DNA sequence
data.

*/


#include <iostream.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "PatternIO.h"
#include "fit.h"
#include "MultiFractalCalc.h"

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#define NUMARGUMENTS 11
#define PROGRAMNAME 0
#define INPUTFILEPARM (PROGRAMNAME+1)
#define STARTBASEFILEPARM (INPUTFILEPARM+1)
#define ENDBASEFILEPARM (STARTBASEFILEPARM+1)
#define SCALEEXPFILEPARM (ENDBASEFILEPARM+1)
#define SCALEMAX1FILEPARM (SCALEEXPFILEPARM+1)
#define SCALEMAX2FILEPARM (SCALEMAX1FILEPARM+1)
#define FTYPEFILEPARM (SCALEMAX2FILEPARM+1)
#define STARTREGRESSPARM (FTYPEFILEPARM+1)
#define ENDREGRESSPARM (STARTREGRESSPARM+1)
#define PREFIXFILEPARM (ENDREGRESSPARM+1)

#define NO_DIMENSION 0
#define RENYII_DIMENSION 1
#define INFORMATION_DIMENSION 2
#define VARIANCE_DIMENSION 3


#define EXTRANAMELENGTH 10
#define MINTHEOQ -50
#define MAXTHEOQ 50

//-------------------------------------------------------
// displayUsage
//   This function prints to standard out the parameters of the program.
//-------------------------------------------------------

void displayUsage()
{
  cout << "Usage: genecalc <<filename>> startbase endbase scaleExp
scaleMax1 scaleMax2 ftype startregress endregress <<prefix>>"<< endl;
  cout << "where <<filename>> is the input flat file" << endl;
  cout << "      startbase is the base number to start at" << endl;
  cout << "      endbase is the ending base number. There are two special
cases:" << endl;
  cout << "         endbase = 0  calculates endbase = dyad^scaleMax +
startbase" << endl;
  cout << "         endbase = 1  calculates endbase = total number of
bases." << endl;
```

```
  cout << "     scaleExp is the scale exponent to use. Normally 2." <<
endl;
  cout << "     scaleMax1 is the unibase   number of scales to keep
track of." << endl;
  cout << "     scaleMax2 is the multibase number of scales to keep
track of." << endl;
  cout << "     ftype is the type of fractal characaterization."<<endl;
  cout << "            r,s - Renyii Dimension with q spacing of 1 and
0.25."<<endl;
  cout << "            i - Information Dimension."<<endl;
  cout << "            v - Variance Dimension."<<endl;
  cout << "     startregress - scale to start regression at."<<endl;
  cout << "     endregress - scale to end regression at."<<endl;
  cout << "         Note: 0 indicates to use max/min scale."<<endl;
  cout << "         Note: 1 for both indicates to fit for all possible
lines."<<endl;
  cout << "     <<prefix>> is a filename prefix. Four output"<<endl;
  cout << "            files are written with <<prefix>>.a.asc
where"<<endl;
  cout << "            a signifies data based on adenine. The
other"<<endl;
  cout << "            files will have g,c, and t respectively. As
well,"<<endl;
  cout << "            a base of e indicates the multiple base
calculations."<<endl;
  cout << " Output file for each base:"<<endl;
  cout << "    <prefix><base>r.asc - dimension calculated with
regression."<<endl;
  cout << "    <prefix><base>f.asc - generalized entropy."<<endl;
  cout << "    <prefix><base>h.asc - holder exponents."<<endl;
  cout << "    <prefix><base>m.asc - Mandelbrot dimensions."<<endl;
  cout << "    <prefix><base>p.asc - Legend of scales used in regression
for r.asc."<<endl;
  cout << "          <prefix>d.asc - Lists the values of q. With min and
max being the first and last."<<endl;
  cout << "          <prefix>s.asc - Lists the scales and there
logs."<<endl;
}

//-----------------------------------------------------
// Main Program.
//-----------------------------------------------------

int main(unsigned int argc, char* argv[])
{
  long startingBase,scaleMax1,scaleMax2,endingBase,dyadicBase,ftypeSel;
  long qqMin, qqMax, numBases, offsetBase, fitStart, fitEnd, fitIndex;
  int baseMax, dyadicIndex, dimIndex, baseIndex;
  char aString;
  PatternIO pattern;
  int outFd;
  FILE *fs;
  DNAFractalCalc calc;
  DimensionIterator dims;
  // Check for the number of arguments, and print out error if they
aren't met.
  if (argc != NUMARGUMENTS) {
    cout << argv[PROGRAMNAME] <<": Incorrect Parameters."<<endl;
    displayUsage();
    exit(-1);
  }

  if (!pattern.setFileName(argv[INPUTFILEPARM])) { // Open File Failed.
    cout << argv[PROGRAMNAME] << ": Error opening input file."<<endl;
    displayUsage();
    exit(-1);
  }

  startingBase = endingBase = scaleMax1=scaleMax2 = -1;
  sscanf(argv[STARTBASEFILEPARM], "%d", &startingBase);
  sscanf(argv[ENDBASEFILEPARM], "%d", &endingBase);
  sscanf(argv[SCALEEXPFILEPARM], "%d", &dyadicBase);
  sscanf(argv[SCALEMAX1FILEPARM], "%d", &scaleMax1);
  sscanf(argv[SCALEMAX2FILEPARM], "%d", &scaleMax2);

  if (startingBase < 0) {
    cout << argv[PROGRAMNAME] <<": Error selecting starting base pair.
Value = " << argv[STARTBASEFILEPARM] << endl;
    displayUsage();
    exit(-1);
  }

  if (endingBase < 0) {
    cout << argv[PROGRAMNAME] <<": Error selecting ending base pair.
Value = " << argv[ENDBASEFILEPARM] << endl;
    displayUsage();
    exit(-1);
  }

  if (dyadicBase < 0) {
    cout << argv[PROGRAMNAME] <<": Error selecting dyad. Value = " <<
argv[SCALEEXPFILEPARM] << endl;
    displayUsage();
    exit(-1);
  }

  if (scaleMax1 < 0) {
    cout << argv[PROGRAMNAME] <<": Error selecting scaleMax1. Value = "
<< argv[SCALEMAX1FILEPARM] << endl;
    displayUsage();
    exit(-1);
  }

  if (scaleMax2 < 0) {
    cout << argv[PROGRAMNAME] <<": Error selecting scaleMax2. Value = "
<< argv[SCALEMAX2FILEPARM] << endl;
    displayUsage();
```

```
      exit(-1);
    }

    if (endingBase == 0) {
      endingBase=1;
      for (int s=0; s<scaleMax1; s++) endingBase *= dyadicBase;
      endingBase+=startingBase;
    } else if (endingBase == 1) {
      endingBase = pattern.fileSize();
    }

    ftypeSel = NO_DIMENSION;
    switch(argv[FTYPEFILEPARM][0]) {
    case 'r':
    case 'R':
      ftypeSel = RENYII_DIMENSION;
      qqMin=-40;qqMax=40;
      dims.copy(DimensionIterator(qqMin,qqMax,1));
      break;
    case 's':
    case 'S':
      ftypeSel = RENYII_DIMENSION;
      qqMin=-40;qqMax=40;
      dims.copy(DimensionIterator((double)qqMin,(double)qqMax,0.25));
      break;
    case 'i':
    case 'I':
      ftypeSel = INFORMATION_DIMENSION;
      qqMin=1;qqMax=1;
      dims.copy(DimensionIterator(qqMin,qqMax,1));
      break;
    case 'v':
    case 'V':
      ftypeSel = VARIANCE_DIMENSION;
      qqMin=2;qqMax=2;
      dims.copy(DimensionIterator(qqMin,qqMax,1));
      break;
    }

    if (ftypeSel == NO_DIMENSION) {
      cout << argv[PROGRAMNAME] <<": Error selecting calculating
dimension. Value = " <<argv[FTYPEFILEPARM] << endl;
      displayUsage();
      exit(-1);
    }

    fitStart = fitEnd = -1;
    sscanf(argv[STARTREGRESSPARM], "%d", &fitStart);
    sscanf(argv[ENDREGRESSPARM], "%d", &fitEnd);

    if (fitStart < 0) {
     cout << argv[PROGRAMNAME] <<": Error selecting startRegress. Value =
" << argv[STARTREGRESSPARM] << endl;
      displayUsage();
    exit(-1);
  }

  if (fitEnd < 0) {
    cout << argv[PROGRAMNAME] <<": Error selecting endRegress. Value = "
<< argv[ENDREGRESSPARM] << endl;
    displayUsage();
    exit(-1);
  }

  if (fitEnd == 0) {
    fitEnd = scaleMax1;
  } else if (fitStart == fitEnd) {
    fitStart = fitEnd = -1;
  }

  // Find the largest scale kmax
  calc.setTotalBasesAndDims(endingBase - startingBase, dims, dyadicBase,
scaleMax1, scaleMax2, fitStart, fitEnd, 12);


  // Allocate memories

  numBases = endingBase - startingBase;
  // Over all of the scales calculate the probabilities.
  for (offsetBase = 0; offsetBase < numBases; offsetBase++){
    aString = pattern.getBase(startingBase+offsetBase);
    calc.addBase(aString);
    if (offsetBase%1000==0) cout << "Progress Base = " << offsetBase<<
endl;
  }

  calc.doFinalCalc();

  // Output the values of each scale to a file.

  for (baseIndex = 0; baseIndex < calc.numBases(); baseIndex++) {
    int prefixLen = strlen(argv[PREFIXFILEPARM]);
    char* outName = new char[prefixLen + EXTRANAMELENGTH];
    strcpy(outName, argv[PREFIXFILEPARM]);
    strcat(outName, calc.basesString(baseIndex));
    strcat(outName, "f.asc");
    outFd = open(outName,O_WRONLY+O_CREAT+O_TRUNC,511);
    if (outFd < 0) {
      cout << argv[PROGRAMNAME] <<": Error creating output file "
<<outName << endl;
      exit(-1);
    }

    cout << "Starting Base " << baseIndex+1<< endl;

    fs = fdopen(outFd, "w");
    // This loop sends all of the data needed in order to reconstruct the
log-log plots.
    for (dyadicIndex = 0; dyadicIndex < calc.dyadicMax(baseIndex);
dyadicIndex++){

fprintf(fs,"%#1E\t",log(calc.minDimFinalTot(baseIndex,dyadicIndex)));
      for (dims.start();dims.cond();dims.next()) {
fprintf(fs,"%#1E\t",calc.finalTot(baseIndex, dyadicIndex,
dims.count()));
      }

fprintf(fs,"%#1E",log(calc.maxDimFinalTot(baseIndex,dyadicIndex)));
      fprintf(fs,"\n");
    }

    delete []outName;
    fclose(fs);
    close(outFd);

    // Now we output the values to the file which have been analyzed
using
    // linear regression.

    prefixLen = strlen(argv[PREFIXFILEPARM]);
    outName = new char[prefixLen + EXTRANAMELENGTH];
    strcpy(outName, argv[PREFIXFILEPARM]);
    strcat(outName, calc.basesString(baseIndex));
    strcat(outName, "r.asc");
    outFd = open(outName,O_WRONLY+O_CREAT+O_TRUNC,511);
    if (outFd < 0) {
      cout << argv[PROGRAMNAME] <<": Error creating output file "
<<outName << endl;
      exit(-1);
    }

    fs = fdopen(outFd, "w");
    for (fitIndex = 0; fitIndex  < calc.fitSize(baseIndex); fitIndex++){
      fprintf(fs,"%#1E\t",calc.minDimFinalFit(baseIndex,fitIndex));
      for (dims.start();dims.cond();dims.next()) {
fprintf(fs,"%#1E\t",calc.finalFit(baseIndex, fitIndex, dims.count()));
      }
      fprintf(fs,"%#1E\n",calc.maxDimFinalFit(baseIndex,fitIndex));
    }
    delete []outName;
    fclose(fs);
    close(outFd);

    // Now we output the values to the file which has the Holder
exponent.

    prefixLen = strlen(argv[PREFIXFILEPARM]);
    outName = new char[prefixLen + EXTRANAMELENGTH];
    strcpy(outName, argv[PREFIXFILEPARM]);
    strcat(outName, calc.basesString(baseIndex));
    strcat(outName, "h.asc");
    outFd = open(outName,O_WRONLY+O_CREAT+O_TRUNC,511);
    if (outFd < 0) {
      cout << argv[PROGRAMNAME] <<": Error creating output file "
<<outName << endl;
      exit(-1);
    }

    fs = fdopen(outFd, "w");
    for (fitIndex = 0; fitIndex  < calc.fitSize(baseIndex); fitIndex++){
      for (dims.start();dims.cond();dims.next()) {
fprintf(fs,"%#1E\t",calc.manAlpha(baseIndex, fitIndex, dims.count()));
      }
      fprintf(fs,"\n");
    }
    delete []outName;
    fclose(fs);
    close(outFd);

    // Now we output the values to the file which has the Mandelbrot
spectrum.

    prefixLen = strlen(argv[PREFIXFILEPARM]);
    outName = new char[prefixLen + EXTRANAMELENGTH];
    strcpy(outName, argv[PREFIXFILEPARM]);
    strcat(outName, calc.basesString(baseIndex));
    strcat(outName, "m.asc");
    outFd = open(outName,O_WRONLY+O_CREAT+O_TRUNC,511);
    if (outFd < 0) {
      cout << argv[PROGRAMNAME] <<": Error creating output file "
<<outName << endl;
      exit(-1);
    }

    fs = fdopen(outFd, "w");
    for (fitIndex = 0; fitIndex  < calc.fitSize(baseIndex); fitIndex++){
      for (dims.start();dims.cond();dims.next()) {
fprintf(fs,"%#1E\t",calc.manSpectrum(baseIndex, fitIndex,
dims.count()));
      }
      fprintf(fs,"\n");
    }
    delete []outName;
    fclose(fs);
    close(outFd);

    cout << "Output fit size file."<<endl;
    // Output a file with the list of start and end scales for fitting.
    prefixLen = strlen(argv[PREFIXFILEPARM]);
    outName = new char[prefixLen + EXTRANAMELENGTH];
    strcpy(outName, argv[PREFIXFILEPARM]);
    strcat(outName, calc.basesString(baseIndex));
    strcat(outName, "p.asc");
    outFd = open(outName, O_WRONLY+O_CREAT+O_TRUNC,511);
    if (outFd < 0) {
      cout << argv[PROGRAMNAME] <<": Error creating output file "
<<outName << endl;
      exit(-1);
    }

    fs = fdopen(outFd, "w");
    for (fitIndex = 0; fitIndex  < calc.fitSize(baseIndex); fitIndex++){
fprintf(fs,"%#ld\t%#ld\n",
calc.dyadicStore(baseIndex, calc.fitTableMin(baseIndex, fitIndex)),
calc.dyadicStore(baseIndex, calc.fitTableMax(baseIndex, fitIndex)));
    }
    delete []outName;
    fclose(fs);
    close(outFd);
  }

  cout << "Output scales file. " << endl;
  // Output a file with list of dyadic scales used
  int prefixLen = strlen(argv[PREFIXFILEPARM]);
  char* outName = new char[prefixLen + EXTRANAMELENGTH];
  strcpy(outName, argv[PREFIXFILEPARM]);
  strcat(outName, "s.asc");
  outFd = open(outName,O_WRONLY+O_CREAT+O_TRUNC,511);
  if (outFd < 0) {
   cout << argv[PROGRAMNAME] <<": Error creating output file " <<outName
<< endl;
    exit(-1);
  }

  fs = fdopen(outFd, "w");
  for (dyadicIndex = 0; dyadicIndex < calc.dyadicMax(0); dyadicIndex++){
    fprintf(fs,"%ld\t%ld\n",dyadicIndex,calc.dyadicStore(0,
dyadicIndex));
  }
  delete []outName;
  fclose(fs);
  close(outFd);

  cout << "Output dimensions file." << endl;
  // Output a file with a list of dimensions used.
  prefixLen = strlen(argv[PREFIXFILEPARM]);
  outName = new char[prefixLen + EXTRANAMELENGTH];
  strcpy(outName, argv[PREFIXFILEPARM]);
  strcat(outName, "d.asc");
  outFd = open(outName,O_WRONLY+O_CREAT+O_TRUNC,511);
  if (outFd < 0) {
   cout << argv[PROGRAMNAME] <<": Error creating output file " <<outName
<< endl;
    exit(-1);
  }

  fs = fdopen(outFd, "w");
  fprintf(fs,"%d\n",qqMin-1);
  for (dims.start();dims.cond();dims.next()) {
    fprintf(fs,"%f\n",dims.current());
  }
  fprintf(fs,"%d\n",qqMax+1);
  delete []outName;
  fclose(fs);
  close(outFd);

  pattern.closeFile();
  cout << "Finished." <<endl;

  return 0;
}
```

A.2.2 MultiFractalCalc.h

```
#include <math.h>

#define PROBTYPE long

// Notes:
//  qMin and qMax will be the start and end dimensions to calculate. They
//    may be the same dimension.


class DimensionIterator {

  protected:
    int _usage; // Should really be a boolean value;
    long _iqMin, _iqMax, _istep, _iCurrent;
    double _dqMin, _dqMax, _dstep, _dCurrent;
    double* _arrayVals;
    long _totalNum, _count;
    enum {USEINTS,USEFLOATS,USEARRAY};

  public:
    DimensionIterator();
    DimensionIterator(long qMin, long qMax, long step);
    DimensionIterator(double qMin, double qMax, double step);
   DimensionIterator(double qMin, double qMax, double stepShort, double
inqMin, double inqMax, double stepLong);
    DimensionIterator(DimensionIterator&);
    virtual ~DimensionIterator() {if (_arrayVals) delete []_arrayVals;}

    void copy(DimensionIterator&);
    void start();
    int cond();
    void next();

    double current() {return _dCurrent;}
    long count() {return _count;}
    long totalNum() {return _totalNum;}
};

#define BUFFERTYPE long

class RoundBuffer {

  private:
    long _size, _start, _count;
    BUFFERTYPE *_items;

  public:
    RoundBuffer();
    ~RoundBuffer();

    void setSize(int aSize); // After this, buffer contents are
scrambled.
    void append(BUFFERTYPE anItem);
    BUFFERTYPE item(int index);
    BUFFERTYPE revItem(int index) {return item(_size-index-1);}
    long size() {return _size;}
    void printit();
    int filled() {return _count>=_size;}
};

class BasicFractalCalc {

  private:
    void relMemory();

  protected:
    double _tolerance;
    static double power(double base, long mant);
    static long factorial(long expr);
    double evalProbRunning(double prob, double dimNum);
    double evalProbFinal(double sumProb, double dimNum, long nTot);
    double logb2(double val) {return log(val)/_log2;}
    double flog(double val) {return logb2(val);} // Just used to switch
between logb2 and log

    double **_finalTotal, *_minFinal, *_maxFinal;
    double **_finalFit, *_minFinalFit, *_maxFinalFit;
    long *_fitTableMin, *_fitTableMax;
    double *_scaleStore;
    double **_modelFit;
    double **_reconFit;
    double **_manSpectrum, **_manAlpha;

    DimensionIterator _q;
    long _scaleMax, _fitStart, _fitEnd, _fitNum, _fitParmNum;
    double _log2;

    void setSizes(long scaleMax, DimensionIterator& q, long fitStart,
long fitEnd, long fitParmNum);
```

```
    void doRegressFit();

  public:
    BasicFractalCalc();
    virtual ~BasicFractalCalc();

    virtual void doFinalCalc() {};

    // Access Functions.
    long scaleMax() {return _scaleMax;}
    double finalTot(int scaleIndex, int dimIndex)
      {return _finalTotal[scaleIndex][dimIndex];}
    double minDimFinalTot(int scaleIndex)
      {return _minFinal[scaleIndex];}
    double maxDimFinalTot(int scaleIndex)
      {return _maxFinal[scaleIndex];}

    double finalFit(int fitIndex, int dimIndex)
      { return _finalFit[fitIndex][dimIndex];}
    double minDimFinalFit(int fitIndex)
      { return _minFinalFit[fitIndex];}
    double maxDimFinalFit(int fitIndex)
      { return _maxFinalFit[fitIndex];}
    long fitSize()
      { return _fitNum;}

    double modelFit(int fitIndex, int parmIndex)
      { return _modelFit[fitIndex][parmIndex];}
    double reconFit(int fitIndex, int dimIndex)
      { return _reconFit[fitIndex][dimIndex];}
    long fitParmNum()
      { return _fitParmNum;}

    long fitTableMin(int fitIndex)
      { return _fitTableMin[fitIndex];}
    long fitTableMax(int fitIndex)
      { return _fitTableMax[fitIndex];}

    double manSpectrum(int fitIndex, int dimIndex)
      { return _manSpectrum[fitIndex][dimIndex];}
    double manAlpha(int fitIndex, int dimIndex)
      { return _manAlpha[fitIndex][dimIndex];}
};

class MultiFractalCalc : public BasicFractalCalc {

  private:
    long _dyadicBase;
    PROBTYPE *_prob, *_minProb, *_maxProb, *_totalProbs,
*_runTotalProbs; // These could be doubles.
    double **_runningTotal;

  protected:
    void releaseMemory();
  public:
    MultiFractalCalc();
    virtual ~MultiFractalCalc();

    void setTolerance(double tol) {_tolerance = tol;}
    void setDyadicMaxAndDims(long dyadicBase, long  dyadicMax,
DimensionIterator& q,
     long fitStart, long fitEnd, long fitParmNum);

    void addProbabilityOne();
    void updateRunningTotals(long uptoIndex);
    virtual void doFinalCalc();

};

class EntropyStringCalc : public BasicFractalCalc {

  private:
    long _pos, _windowSize, _alphabetSize;
    char* _alphabet;
    long *_counted, **_probs, *_indexSizes;
    DimensionIterator _q;
    RoundBuffer _buffer;

  protected:
    void releaseMemory();

  public:
    EntropyStringCalc();
    virtual ~EntropyStringCalc();

    void setAlphabet(char *alphabet, long windowSize, long maxVelSize,
DimensionIterator q,
     long fitStart, long fitEnd, long fitParmNum);
    void nextLetter(char letter);
    virtual void doFinalCalc();
};

class DNAFractalCalc {

  private:
    enum {_NUMBASES=5};
    enum {_NUMMULTIFRACS=4};
    enum {_ENTROPYFRAC=4};
    char _bases[_NUMBASES];
    char* _basesString[_NUMBASES];

    long _numBases, _dyadicMax, _dyadicBase;
    long _totalBases, _currentNumBases;
    long *_dyadicStore; // Previously n. Holds the different scales.
    BasicFractalCalc* _mfCalcs[_NUMBASES];  // Array of calculations.
One for each base.
```

```
  protected:

  public:
    DNAFractalCalc();
    virtual ~DNAFractalCalc();

    void setTotalBasesAndDims(long nBases, DimensionIterator& q, long
dyadicBase, long  scaleMaxU, long scaleMaxM,
      long fitStart, long fitEnd, long fitParmNum);
    void addBase(char inString);
    void doFinalCalc();

    // Access Functions.
    long numBases() {return _numBases;}
    long dyadicStore(int baseIndex, int index) {
      if (baseIndex < _ENTROPYFRAC) return
(_dyadicStore)?_dyadicStore[index]:-1;
      else return index+1;
    }

    char* basesString(int baseIndex) {return _basesString[baseIndex];}
    char bases(int baseIndex) {return _bases[baseIndex];}

    long dyadicMax(int baseIndex)
      {return _mfCalcs[baseIndex]->scaleMax();}

    // Access to _mfCalc values;
    double finalTot(int baseIndex, int dyadicIndex, int dimIndex)
      {return _mfCalcs[baseIndex]->finalTot(dyadicIndex, dimIndex);}
    double minDimFinalTot(int baseIndex, int dyadicIndex)
      {return _mfCalcs[baseIndex]->minDimFinalTot(dyadicIndex);}
    double maxDimFinalTot(int baseIndex, int dyadicIndex)
      {return _mfCalcs[baseIndex]->maxDimFinalTot(dyadicIndex);}


    double finalFit(int baseIndex, int fitIndex, int dimIndex)
      { return _mfCalcs[baseIndex]->finalFit(fitIndex,dimIndex);}
    double minDimFinalFit(int baseIndex, int fitIndex)
      { return _mfCalcs[baseIndex]->minDimFinalFit(fitIndex);}
    double maxDimFinalFit(int baseIndex, int fitIndex)
      { return _mfCalcs[baseIndex]->maxDimFinalFit(fitIndex);}
    long fitSize(int baseIndex)
      { return _mfCalcs[baseIndex]->fitSize();}

    double modelFit(int baseIndex, int fitIndex, int parmIndex)
      { return _mfCalcs[baseIndex]->modelFit(fitIndex,parmIndex);}
    double reconFit(int baseIndex, int fitIndex, int dimIndex)
      { return _mfCalcs[baseIndex]->reconFit(fitIndex,dimIndex);}
    long fitParmNum(int baseIndex)
      { return _mfCalcs[baseIndex]->fitParmNum();}

    long fitTableMin(int baseIndex, int fitIndex)
      { return _mfCalcs[baseIndex]->fitTableMin(fitIndex);}
    long fitTableMax(int baseIndex, int fitIndex)
```

```
      { return _mfCalcs[baseIndex]->fitTableMax(fitIndex);}

    double manSpectrum(int baseIndex, int fitIndex, int dimIndex)
      { return _mfCalcs[baseIndex]->manSpectrum(fitIndex,dimIndex);}
    double manAlpha(int baseIndex, int fitIndex, int dimIndex)
      { return _mfCalcs[baseIndex]->manAlpha(fitIndex,dimIndex);}
};
```

A.2.3 MultifractalCalc.cc

```
#include "MultiFractalCalc.h"
#include <math.h>
#include <stdlib.h>
#include <iostream.h>
#include "fit.h"
#include "nonlinear.h"
#include <string.h>
#define PRESET_TOLERANCE 0.00000001

//
========================================================================
====================
//  DimensionIterator
//
========================================================================
====================

DimensionIterator::DimensionIterator()
{
  _usage = USEINTS;
  _iqMin = 0;
  _iqMax = 0;
  _istep = 1;
  _totalNum = 1;
  _arrayVals = NULL;
  start();
}

DimensionIterator::DimensionIterator(DimensionIterator& src)
{
  _arrayVals = NULL;
  copy(src);
}

void DimensionIterator::copy(DimensionIterator& src)
{
  _usage = src._usage;
  switch(_usage) {
  case USEINTS:
    _iqMin = src._iqMin;
    _iqMax = src._iqMax;
    _istep = src._istep;
    _iCurrent = src._iCurrent;
    break;
  case USEFLOATS:
    _dqMin = src._dqMin;
    _dqMax = src._dqMax;
    _dstep = src._dstep;
    _dCurrent = src._dCurrent;
    break;
  case USEARRAY:
    if (_arrayVals) delete []_arrayVals;
    _arrayVals = new double[src._totalNum];
    for (int i=0; i<src._totalNum; i++)
      _arrayVals[i]=src._arrayVals[i];
    break;
  }
  _totalNum = src._totalNum;
  _count = src._count;
}

DimensionIterator::DimensionIterator(long qMin, long qMax, long step)
{
  _usage = USEINTS;
  _iqMin = qMin;
  _iqMax = qMax;
  _istep = step;

  _totalNum = (_iqMax - _iqMin) /_istep +1;
  start();
}

DimensionIterator::DimensionIterator(double qMin, double qMax, double
step)
{
  _usage = USEFLOATS;
  _dqMin = qMin;
  _dqMax = qMax;
  _dstep = step;

  _totalNum = (long) (floor((_dqMax - _dqMin) / _dstep)) + 1;
  start();
}

DimensionIterator::DimensionIterator(double qMin, double qMax, double
stepLong, double inqMin, double inqMax, double stepShort)
{
  long i;

  _usage = USEARRAY;
  _dqMin = qMin;
  _dqMax = qMax;

  long firstPart = (long) (floor((_dqMax - inqMax) / stepLong));
  long secondPart = (long) (floor((inqMax - inqMin) / stepShort)) + 1;
  long thirdPart = (long) (floor((inqMin - _dqMin) / stepLong)) + 1;
  _totalNum = firstPart + secondPart + thirdPart;
  _arrayVals = new double[_totalNum];

  for (i=0; i < thirdPart; i++)
    _arrayVals[i] = _dqMin + ((double)i)*stepLong;
  for (i=0; i < secondPart; i++)
    _arrayVals[i+thirdPart] = inqMin + ((double)i)*stepShort;
  for (i=0; i < firstPart; i++)
    _arrayVals[i+thirdPart+secondPart] = inqMax +
((double)i+1)*stepLong;
}

void DimensionIterator::start()
{
  switch(_usage) {
  case USEINTS:
    _iCurrent = _iqMin;
    _dCurrent = _iCurrent;
    break;
  case USEFLOATS:
    _dCurrent = _dqMin;
    break;
  case USEARRAY:
    _dCurrent = _arrayVals[0];
    break; // The array uses the count variable.
  }
  _count = 0;
}

int DimensionIterator::cond()
{
  int returnVal;
  switch(_usage) {
  case USEINTS:
    returnVal = _iCurrent <= _iqMax;
    break;
  case USEFLOATS:
    returnVal = _dCurrent <= _dqMax;
    break;
  case USEARRAY:
    returnVal = _count < _totalNum;
    break;
  }
  return returnVal;
}

void DimensionIterator::next()
{
  switch(_usage) {
  case USEINTS:
    _iCurrent += _istep;
    _dCurrent = _iCurrent;
    break;
  case USEFLOATS:
    _dCurrent += _dstep;
    break;
  case USEARRAY:
    _dCurrent = _arrayVals[_count+1];
    break;
  }
  _count++;
}


//
======================================================================
====================
//  RoundBuffer
//
======================================================================
====================

RoundBuffer::RoundBuffer()
{
  _size = 0;
  _items = NULL;
  _start = 0;
  _count = 0;
}

RoundBuffer::~RoundBuffer()
{
  delete []_items;
}

void RoundBuffer::setSize(int aSize)
{
  _size = aSize;
  _start = 0;
  _count = 0;
  delete []_items;
  _items = new BUFFERTYPE[_size];
}

void RoundBuffer::append(BUFFERTYPE anItem)
{
  _items[_start] = anItem;
  _start++;
  if (_start>=_size) _start=0;
  _count++;
}

BUFFERTYPE RoundBuffer::item(int index)
{
  int tmp=index+_start;
  return _items[((tmp>=_size)?(tmp-_size):tmp)];
}
```

```
void RoundBuffer::printit()
{
  cout << "S:"<<_start<<" I:";
  for (int i=0;i<_size;i++) cout << _items[i] << " ";
}

//
========================================================================
====================
//  BasicFractalCalc
//
========================================================================
====================

BasicFractalCalc::BasicFractalCalc()
{
  _tolerance = PRESET_TOLERANCE;
  _minFinal = NULL;
  _maxFinal = NULL;
  _finalTotal = NULL;

  _finalFit = NULL;
  _minFinalFit = NULL;
  _maxFinalFit = NULL;
  _fitTableMin = NULL;
  _fitTableMax = NULL;
  _scaleStore = NULL;
  _modelFit = NULL;
  _reconFit = NULL;
  _manSpectrum = NULL;
  _manAlpha = NULL;

  _log2 = log(2.0);
}

BasicFractalCalc::~BasicFractalCalc()
{
  relMemory();
}

void BasicFractalCalc::relMemory()
{
  if (_finalTotal) {
    for (int scaleIndex=0; scaleIndex < _scaleMax; scaleIndex++) {
      delete []_finalTotal[scaleIndex];
    }
  }

  delete []_minFinal;
  delete []_maxFinal;
  free(_finalTotal);

  if (_finalFit) {
    for (int fitIndex=0; fitIndex < _fitNum; fitIndex++) {
      delete [](_finalFit[fitIndex]);
      delete [](_reconFit[fitIndex]);
      delete [](_manSpectrum[fitIndex]);
      delete [](_manAlpha[fitIndex]);
    }
  }
  if (_modelFit) {
    for (int fitItem=0; fitItem < _fitParmNum; fitItem++) {
      delete []_modelFit[fitItem];
    }
  }
  free(_finalFit);
  free(_modelFit);
  free(_reconFit);
  free(_manSpectrum);
  free(_manAlpha);

  delete []_minFinalFit;
  delete []_maxFinalFit;
  delete []_fitTableMin;
  delete []_fitTableMax;
  delete []_scaleStore;

  _minFinal = NULL;
  _maxFinal = NULL;
  _finalTotal = NULL;

  _finalFit = NULL;
  _minFinalFit = NULL;
  _maxFinalFit = NULL;
  _fitTableMin = NULL;
  _fitTableMax = NULL;
  _scaleStore = NULL;
  _manSpectrum = NULL;
  _manAlpha = NULL;

  _modelFit = NULL;
  _reconFit = NULL;

}
void BasicFractalCalc::setSizes(long scaleMax, DimensionIterator& q,
long fitStart, long fitEnd, long fitParmNum)
{
  relMemory();
  _scaleMax = scaleMax;
  _q.copy(q);

  _fitStart = fitStart;
  _fitEnd = fitEnd;
  _fitParmNum = fitParmNum;

  if (_fitStart == -1) {
    _fitNum = scaleMax*(scaleMax+1)/2;
  }else{
    _fitNum = 1;
  }

  _minFinal = new double[_scaleMax];
  _maxFinal = new double[_scaleMax];
  _minFinalFit = new double[_fitNum];
  _maxFinalFit = new double[_fitNum];
  _fitTableMin = new long[_fitNum];
  _fitTableMax = new long[_fitNum];
  _scaleStore = new double[_scaleMax];

  _finalFit = (double**)malloc(_fitNum * sizeof(double*));
  _reconFit = (double**)malloc(_fitNum * sizeof(double*));
  _manSpectrum = (double**)malloc(_fitNum * sizeof(double*));
  _manAlpha = (double**)malloc(_fitNum * sizeof(double));

  for (int fitIndex = 0; fitIndex < _fitNum; fitIndex++) {
    _finalFit[fitIndex] = new double[_q.totalNum()];
    _reconFit[fitIndex] = new double[_q.totalNum()];
    _manSpectrum[fitIndex] = new double[_q.totalNum()];
    _manAlpha[fitIndex] = new double[_q.totalNum()];
  }

  _modelFit = (double**)malloc(_fitParmNum * sizeof(double*));
  for (int parmIndex = 0; parmIndex < _fitParmNum; parmIndex++) {
    _modelFit[parmIndex] = new double[_q.totalNum()];
  }

  int count = 0;
  if (_fitStart == -1) {
    for (int upToIndex = scaleMax; upToIndex > 0; upToIndex--) {
      for (int minFitIndex=upToIndex-1; minFitIndex>=0;minFitIndex--) {
_fitTableMin[count] = minFitIndex;
_fitTableMax[count] = upToIndex;
count++;
      }
    }
  } else {
    _fitTableMin[0] = _fitStart;
    _fitTableMax[0] = _fitEnd;
  }

  _finalTotal = (double**)malloc(_scaleMax * sizeof(double*));

  for (int scaleIndex = 0; scaleIndex < _scaleMax; scaleIndex++) {
    _scaleStore[scaleIndex] = scaleIndex+1;
    _finalTotal[scaleIndex] = new double[_q.totalNum()];
    for (_q.start(); _q.cond(); _q.next()) {
      _finalTotal[scaleIndex][q.count()] = 0.0;
    }
  }
}
```

```
//------------------------------------------------------
// doRegressFit
//    This function performs the necessary regression for fitting,
// and modelling, and reconstruction. It requires that the sub
// class initialize _scaleStore properly.
//------------------------------------------------------
void BasicFractalCalc::doRegressFit()
{
  long scaleIndex, fitIndex;
  double *dimsUsed = new double[_q.totalNum()];
  double p1,p2,p3;

  // Create a temporary array for the dims.
  for (_q.start(); _q.cond(); _q.next()) {
    dimsUsed[_q.count()] = _q.current();
  }


  for (fitIndex = 0; fitIndex < _fitNum; fitIndex++) {
    // Need to create temporary for scale array.
    int fitStart = _fitTableMin[fitIndex], fitLength =
_fitTableMax[fitIndex] - fitStart;

    double *dyadicStore = _scaleStore+fitStart-1 , *probStore = new
double[fitLength];
    double *x = dyadicStore - 1, *y = probStore - 1;
    double answer, throw1, throw2, throw3, throw4, throw5;

    for (_q.start(); _q.cond(); _q.next()) {
      for (scaleIndex = fitStart; scaleIndex < fitStart+fitLength;
scaleIndex++)
probStore[scaleIndex-fitStart] = _finalTotal[scaleIndex][_q.count()];

      fit(x, y, fitLength, NULL, 0, &throw1, &answer, &throw2, &throw3,
  &throw4, &throw5);

      _finalFit[fitIndex][_q.count()] = answer;
    }

//    nonlinearFit(dimsUsed, _finalFit[fitIndex], _totalDims,
_modelFit[fitIndex], _fitParmNum);
//    nonlinearReconstruct(dimsUsed, _reconFit[fitIndex], _totalDims,
_modelFit[fitIndex], _fitParmNum);

    // Do fitting for min
    for (scaleIndex = fitStart; scaleIndex < fitStart+fitLength;
scaleIndex++)
      probStore[scaleIndex-fitStart] = _minFinal[scaleIndex];
    fit(x,y, fitLength, NULL, 0, &throw1, &answer, &throw2, &throw3,
&throw4, &throw5);
    _minFinalFit[fitIndex] = answer;
    for (scaleIndex = fitStart; scaleIndex < fitStart+fitLength;
scaleIndex++)
      probStore[scaleIndex-fitStart] = _maxFinal[scaleIndex];
    fit(x,y, fitLength, NULL, 0, &throw1, &answer, &throw2, &throw3,
&throw4, &throw5);
    _maxFinalFit[fitIndex] = answer;

    // Here we calculate the appropriate value of the Mandelbrot spectra.
    for (_q.start(); _q.cond(); _q.next()) {
      if (_q.count() == 0) {
p1 = (dimsUsed[_q.count()  ]-1)*_finalFit[fitIndex][_q.count()  ];
p2 = (dimsUsed[_q.count()+1]-1)*_finalFit[fitIndex][_q.count()+1];
_manAlpha[fitIndex][_q.count()] = (p1-p2)/(dimsUsed[_q.count()]-
dimsUsed[_q.count()+1]);
      }else if (_q.count() == _q.totalNum() -1) {
p1 = (dimsUsed[_q.count()  ]-1)*_finalFit[fitIndex][_q.count()  ];
p2 = (dimsUsed[_q.count()-1]-1)*_finalFit[fitIndex][_q.count()-1];
_manAlpha[fitIndex][_q.count()] = (p1-p2)/(dimsUsed[_q.count()]-
dimsUsed[_q.count()-1]);
      } else {
p1 = (dimsUsed[_q.count()-1]-1)*_finalFit[fitIndex][_q.count()-1];
p2 = (dimsUsed[_q.count()  ]-1)*_finalFit[fitIndex][_q.count()  ];
p3 = (dimsUsed[_q.count()+1]-1)*_finalFit[fitIndex][_q.count()+1];
_manAlpha[fitIndex][_q.count()] = (p3-p1)/(dimsUsed[_q.count()+1]-
dimsUsed[_q.count()-1]);
      }
      _manSpectrum[fitIndex][_q.count()] = _q.current() *
_manAlpha[fitIndex][_q.count()] -
(_q.current()-1)*_finalFit[fitIndex][_q.count()];

    }


    delete []probStore;
  }

  delete []dimsUsed;

}

//--------------------------------------------------------
// power
//    This function performs an double base to an integer power
calculation.
//--------------------------------------------------------

double BasicFractalCalc::power(double base, long mant)
{
  double returnVal = 1.0;
  int i;

  if (mant < 0) {
    for (i = 0; i > mant; i--)
      returnVal = returnVal / base;
  } else if (mant > 0) {
    for (i = 0; i < mant; i++)
      returnVal = returnVal * base;
  } else
    returnVal = 1.0;

  return returnVal;
}

//---------------------------------------------------
// factorial
//    This function performs a long factorial.
//---------------------------------------------------

long BasicFractalCalc::factorial(long expr) {
  long i;
  long returnVal=1;

  for (i=2;i<=expr;i++) {
    returnVal*=i;
  }
  return returnVal;
}


//---------------------------------------------------
// evalProbRunning
//   This function takes a probability and dimension, and returns the
//   correct power function. It's main importance is filtering the
//   information dimension, and not completing the computation if the
//   probability is extremely small.
//---------------------------------------------------

double BasicFractalCalc::evalProbRunning(double prob, double dimNum)
{
  double returnVal;
  int i;

  if (prob < _tolerance) {
    returnVal = 0.0;
  } else if (fabs(dimNum-1.0)<_tolerance) {
    returnVal = prob * flog(prob);
  } else {
    returnVal = pow(prob,dimNum);
  }

  return returnVal;
}

//---------------------------------------------------
// evalProbFinal
//    This function takes the total probability sum, dimension and total
//    number of probabilities used, and calculates the top logarithm. It
//    is mainly used to handle the special case of Information Dimension.
```

```
//------------------------------------------------------

double BasicFractalCalc::evalProbFinal(double sumProb, double dimNum,
long nTot)
{
  double returnVal;
  double dNTot = (double)nTot;

  if (fabs(dimNum-1.0)<_tolerance) { // Information Dimension
      returnVal = sumProb / dNTot - flog(dNTot);
  } else {
    returnVal = flog(sumProb/pow(dNTot,dimNum))/(dimNum-1.0);
  }

  return returnVal;
}

//
=========================================================================
====================
//  MultiFractalCalc
//
=========================================================================
====================

//------------------------------------------------------
//------------------------------------------------------
MultiFractalCalc::MultiFractalCalc()
{
  _prob = NULL;
  _minProb = NULL;
  _maxProb = NULL;
  _totalProbs = NULL;
  _runTotalProbs = NULL;
  _runningTotal = NULL;
}

//------------------------------------------------------
//------------------------------------------------------
MultiFractalCalc::~MultiFractalCalc()
{
  releaseMemory();
}

//------------------------------------------------------
//------------------------------------------------------
void MultiFractalCalc::releaseMemory()
{
  long dyadicIndex;

  if (_prob) delete []_prob;
  if (_minProb) delete []_minProb;
  if (_maxProb) delete []_maxProb;
  if (_totalProbs) delete []_totalProbs;
  if (_runTotalProbs) delete []_runTotalProbs;
  if (_runningTotal) {
    for (int dyadicIndex=0; dyadicIndex < _scaleMax; dyadicIndex++) {
      delete []_runningTotal[dyadicIndex];
    }
  }

  free(_runningTotal);
  _prob=NULL;
  _minProb = NULL;
  _maxProb = NULL;
  _totalProbs = NULL;
  _runTotalProbs = NULL;
  _runningTotal = NULL;
}

//------------------------------------------------------
//------------------------------------------------------


void MultiFractalCalc::setDyadicMaxAndDims(long dyadicBase, long
dyadicMax, DimensionIterator& q,
   long fitStart, long fitEnd, long fitParmNum)
{ // Should check that dyadicMax > 0

  releaseMemory();
  setSizes(dyadicMax, q, fitStart, fitEnd, fitParmNum);

  _dyadicBase = dyadicBase;

  _prob = new PROBTYPE[_scaleMax];
  _minProb = new PROBTYPE[_scaleMax];
  _maxProb = new PROBTYPE[_scaleMax];
  _totalProbs = new PROBTYPE[_scaleMax];
  _runTotalProbs = new PROBTYPE[_scaleMax];

  _runningTotal = (double**)malloc(_scaleMax * sizeof(double*));

  for (int dyadicIndex = 0; dyadicIndex < _scaleMax; dyadicIndex++) {
    _prob[dyadicIndex] = 0;
    _minProb[dyadicIndex] = -1;
    _maxProb[dyadicIndex] = -1;
    _totalProbs[dyadicIndex] = 0;
    _runTotalProbs[dyadicIndex] = 0;

    _runningTotal[dyadicIndex] = new double[_q.totalNum()];
    _finalTotal[dyadicIndex] = new double[_q.totalNum()];
    for (q.start(); q.cond(); q.next()) {
      _runningTotal[dyadicIndex][q.count()] = 0.0;
    }
  }
}

//------------------------------------------------------
```

```
//------------------------------------------------------

void MultiFractalCalc::addProbabilityOne()
{
  int dyadicIndex;

//  _totalProbs++;
  for (dyadicIndex = 0; dyadicIndex < _scaleMax; dyadicIndex++) {
    _prob[dyadicIndex]++;
    _runTotalProbs[dyadicIndex]++;
  }
}

//------------------------------------------------------
//------------------------------------------------------

void MultiFractalCalc::updateRunningTotals(long uptoIndex)
{
  int dyadicIndex, dimIndex;

  for (dyadicIndex = 0; dyadicIndex <= uptoIndex; dyadicIndex++) {
    double pTemp = (double) _prob[dyadicIndex];
    for (_q.start(); _q.cond(); _q.next()) {
      _runningTotal[dyadicIndex][_q.count()] += evalProbRunning(pTemp,
_q.current());
    }

    if (_minProb[dyadicIndex] == -1) {
      _minProb[dyadicIndex] = _maxProb[dyadicIndex] =
_prob[dyadicIndex];
    } else {
      if (_minProb[dyadicIndex] > _prob[dyadicIndex])
_minProb[dyadicIndex] = _prob[dyadicIndex];
      if (_maxProb[dyadicIndex] < _prob[dyadicIndex])
_maxProb[dyadicIndex] = _prob[dyadicIndex];
    }
    _prob[dyadicIndex] = 0;
    _totalProbs[dyadicIndex] += _runTotalProbs[dyadicIndex];
    _runTotalProbs[dyadicIndex] = 0;
  }
}

//------------------------------------------------------
// In this function, fitting has been added. In order to not fit
// the function under certain dyadic indexes, code could be
// added here!!!
//------------------------------------------------------

void MultiFractalCalc::doFinalCalc()
{
  long dyadicIndex;
  double pTemp;
  for (dyadicIndex = 0; dyadicIndex < _scaleMax; dyadicIndex++) {
    for (_q.start(); _q.cond(); _q.next()) {
      pTemp = _runningTotal[dyadicIndex][_q.count()];
      _finalTotal[dyadicIndex][_q.count()] =
evalProbFinal(pTemp,_q.current(),_totalProbs[dyadicIndex]);
    }
    _minFinal[dyadicIndex] = flog((double)_minProb[dyadicIndex] /
(double)_totalProbs[dyadicIndex]);
    _maxFinal[dyadicIndex] = flog((double)_maxProb[dyadicIndex] /
(double)_totalProbs[dyadicIndex]);
  }

  for (dyadicIndex = 0; dyadicIndex < _scaleMax; dyadicIndex++){
    _scaleStore[dyadicIndex] = flog(power(_dyadicBase,dyadicIndex));
  }

  doRegressFit();
}


//
========================================================================
====================
//  EntropyStringCalc
//
========================================================================
====================

EntropyStringCalc::EntropyStringCalc()
{
  _alphabet = NULL;
  _counted = NULL;
  _probs = NULL;
}

EntropyStringCalc::~EntropyStringCalc()
{
  releaseMemory();
}

void EntropyStringCalc::releaseMemory()
{
  int i;

  delete []_counted;
  delete []_indexSizes;

  for (i=0; i<_scaleMax; i++){
    delete []_probs[i];
  }
  free(_probs);

}

void EntropyStringCalc::setAlphabet(char *alphabet, long windowSize,
long maxVelSize, DimensionIterator q,
    long fitStart, long fitEnd, long fitParmNum)
{
  long i,a,j;

  releaseMemory();
  setSizes(maxVelSize, q, fitStart, fitEnd, fitParmNum);

  _q.copy(q);
  _windowSize = windowSize;
  _alphabet = alphabet;

  // The size of the alphabet will determine how many values to store.
  _alphabetSize = strlen(alphabet);
  _buffer.setSize(_scaleMax);

  _pos = 0;  // This is the number of letters which have passed us.


  _counted = new long[_scaleMax];
  _indexSizes = new long[_scaleMax];
  _probs = (long**)malloc(_scaleMax*sizeof(long*));
  for (i=0,a=1; i<_scaleMax; i++) {
    _indexSizes[i] = a;
    _counted[i] = 0;
    a*=_alphabetSize;
    _probs[i] = new long[a];
    for (j=0; j<a; j++) _probs[i][j]=0;
  }

}

// This function assumes that lookfor is guaranteed to be in the
// string.

int strposi(char* str, char lookfor)
{
  int i=0;
  while(str[i]!=lookfor) i++;
  return i;
}

void EntropyStringCalc::nextLetter(char letter)
{
  long index, vel, vpos;

  // Update History List
  long aPos = strposi(_alphabet,letter);
  _buffer.append(aPos);
  _pos++;

  if (_pos < 40) {
    cout << letter << " ";
    cout << aPos << " -- ";
    for (int j=0; j<_scaleMax;j++)
```

```
      cout << _buffer.revItem(j) << " ";
    cout << " -- ";
    for (int k=0; k <_scaleMax;k++)
      cout << _buffer.item(k) << " ";
    cout << " == ";
    _buffer.printit();
    cout << endl;
  }

  // Add Entry to each new new item.
  // Convert from RoundBuffer to index for each vel size.

  for (vel=0;vel<_scaleMax;vel++) {
    if (_pos>vel) { // This checks to make sure we don't try and do any
sequences at the beginning.
      for (vpos=0,index=0;vpos<=vel;vpos++) {
index+=_buffer.revItem(vpos)*_indexSizes[vpos];
      }
      _probs[vel][index]++;
      _counted[vel]++;
    }
  }
}

void EntropyStringCalc::doFinalCalc()
{
  int vel, index;
  double prob;
  long minProbTmp, maxProbTmp, nSymbols;

  // Calculate Entropies from stored probabilities.
  for (vel = 0; vel < _scaleMax; vel++) {
    nSymbols = _indexSizes[vel]*_alphabetSize;
    double count = (double)_counted[vel];
    long tmpCount = 0;
    for (index=0; index<nSymbols; index++) {
      prob = ((double)_probs[vel][index]);
      tmpCount += _probs[vel][index];
      for (_q.start(); _q.cond(); _q.next()) {
_finalTotal[vel][_q.count()] += evalProbRunning(prob, _q.current());
      }
      if (index==0) {
minProbTmp = maxProbTmp = prob;
      } else {
if (minProbTmp > prob) minProbTmp = prob;
else if (maxProbTmp < prob) maxProbTmp = prob;
      }
    }
    cout << "Count = "<<count<<"  Total="<<tmpCount;
    cout << endl;
    for (_q.start(); _q.cond(); _q.next()) {
      double *tmp = &(_finalTotal[vel][_q.count()]);
      *tmp = -evalProbFinal(*tmp, _q.current(), _counted[vel])/2.0;
    }
    _minFinal[vel] = flog(((double)minProbTmp)/count);
    _maxFinal[vel] = flog(((double)maxProbTmp)/count);
  }

  // Fit Entropies using linear regression.
  for (long scaleIndex = 0.0; scaleIndex < _scaleMax; scaleIndex++){
    _scaleStore[scaleIndex] = ((double)(scaleIndex+1.0));
  }
  doRegressFit();
}

//
==========================================================================
====================
//  DNAFractalCalc
//
==========================================================================
====================

//-------------------------------------------------------
// Constructor
//-------------------------------------------------------
DNAFractalCalc::DNAFractalCalc()
{
  _numBases = _NUMBASES;
  _bases[0] = 'a';
  _bases[1] = 'g';
  _bases[2] = 'c';
  _bases[3] = 't';
  _bases[4] = 'e';
  _basesString[0] = "a";
  _basesString[1] = "g";
  _basesString[2] = "c";
  _basesString[3] = "t";
  _basesString[4] = "e";

  for (int baseIndex = 0; baseIndex < _numBases; baseIndex++){
    if (baseIndex < _ENTROPYFRAC)
      _mfCalcs[baseIndex] = new MultiFractalCalc();
    else
      _mfCalcs[baseIndex] = new EntropyStringCalc();
  }
  _currentNumBases = 0;
  _totalBases = 0;
  _dyadicStore = NULL;
}

//-------------------------------------------------------
// Destructor
//-------------------------------------------------------
DNAFractalCalc::~DNAFractalCalc()
{
  for (int baseIndex = 0; baseIndex < _numBases; baseIndex++)
    delete _mfCalcs[baseIndex];
  if (_dyadicStore) delete []_dyadicStore;
}

//--------------------------------------------------
//--------------------------------------------------
void DNAFractalCalc::setTotalBasesAndDims(long nBases,
DimensionIterator& q, long  dyadicBase, long  scaleMaxU, long scaleMaxM,
  long fitStart, long fitEnd, long fitParmNum)
{
  _totalBases = nBases;
  _currentNumBases = 0;
  _dyadicMax = scaleMaxU;
  _dyadicBase = dyadicBase;

  for (int baseIndex=0; baseIndex < _numBases; baseIndex++) {
    if (baseIndex < _ENTROPYFRAC)
      ((MultiFractalCalc*)_mfCalcs[baseIndex])-
>setDyadicMaxAndDims(_dyadicBase, _dyadicMax, q, fitStart, fitEnd,
fitParmNum);
    else
      ((EntropyStringCalc*)_mfCalcs[baseIndex])->setAlphabet("agct",
nBases, scaleMaxM, q, fitStart, fitEnd, fitParmNum);
  }

  if (_dyadicStore) delete []_dyadicStore;
  _dyadicStore = new long[_dyadicMax];
  for (int dyadicIndex = 0; dyadicIndex < _dyadicMax; dyadicIndex++) {
    if (dyadicIndex == 0)
      _dyadicStore[dyadicIndex] = 1;
    else
      _dyadicStore[dyadicIndex] = _dyadicStore[dyadicIndex-1] *
_dyadicBase;
  }
}

//--------------------------------------------------
//--------------------------------------------------

void DNAFractalCalc::addBase(char inString)
{
  long baseIndex, dyadicIndex;
  long upToIndex = -1;

  _currentNumBases++;

  for (dyadicIndex = 0; dyadicIndex < _dyadicMax; dyadicIndex++) {
    if (_currentNumBases%_dyadicStore[dyadicIndex] == 0) {
      upToIndex = dyadicIndex;
    }
  }

  if ((inString>='A')&&(inString<='Z')) inString = inString - 'A' + 'a';

  int catchFlag = 0;
  for (baseIndex = 0; baseIndex < _ENTROPYFRAC; baseIndex++) {
    if (inString == _bases[baseIndex]){
      catchFlag++;
      ((MultiFractalCalc*)_mfCalcs[baseIndex])->addProbabilityOne();
    }
    if (upToIndex >= 0) {
      ((MultiFractalCalc*)_mfCalcs[baseIndex])-
>updateRunningTotals(upToIndex);
    }
  }
  if (catchFlag > 0)
   ((EntropyStringCalc*)_mfCalcs[_ENTROPYFRAC])->nextLetter(inString);
}

void DNAFractalCalc::doFinalCalc()
{
  long baseIndex;

  for (baseIndex = 0; baseIndex < _numBases; baseIndex++){
    _mfCalcs[baseIndex]->doFinalCalc();
  }
}
```

A.2.4 PatternIO.h

```
/*

   This class provides flat file IO. It also allows for test sequences to
be created
    by passing in the appropriate file name:
      xxSEQ will simulate a virtual file which contains SEQ repeated.
      xxx will simulate a virtual file which returns a uniform random
base.
*/
#define SString char*

class PatternIO {
  protected:
    SString _fileName;
    SString _repSeq;
    char _bases[4];
    int _fileId, _generateSeq, _repLength;
//    SString _buffer;
    long _offset;
    long _size;

  public:
    PatternIO();
    virtual ~PatternIO();

    int setFileName(SString& fileName);
    int openFile(); // Returns positive for success, negative for fail
    void closeFile();
    long fileSize() {return _size;}

//    void getBase(long position, SString& aBase) {aBase =
getBase(position);}
    char getBase(long position);
};
```

A.2.5 PatternIO.cc

```
#include "PatternIO.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <strings.h>
#include <iostream.h>
#include <stdlib.h>

#define GENERATE_NONE 0
#define GENERATE_FILE 1
#define GENERATE_UNIFORM 2
#define GENERATE_STRING 3

PatternIO::PatternIO()
{
  _fileId = -1;
  _offset = 0;
  _fileName = "";
  _generateSeq = GENERATE_NONE;
//  _buffer = "";
  _bases[0]='a';
  _bases[1]='g';
  _bases[2]='c';
  _bases[3]='t';
}

PatternIO::~PatternIO()
{
  closeFile();
}

int PatternIO::setFileName(SString& fileName)
{
  closeFile();
  _fileName = fileName;
  return openFile();
}

int PatternIO::openFile()
{
  mode_t mode;
  struct stat buf;
  int returnVal = 0;

  // Check to see if it's a special sequence.
  if ((_fileName[0] == 'x') &&
      (_fileName[1] == 'x')) {
    if (_fileName[2] == 'x') {
      _generateSeq = GENERATE_UNIFORM;
      srandom(time(0));
    } else {
      _generateSeq = GENERATE_STRING;
      _repLength = strlen(_fileName)-2;
      _repSeq = new char[_repLength + 5];
      strcpy(_repSeq, _fileName+2);
      cout << "Selected: Generate String:" << endl;
    }
    returnVal = 1;
  }
  else if (_fileId < 0) {
    _fileId = open(_fileName,O_RDONLY,&mode);
    if (_fileId > 0) {
      fstat(_fileId,&buf);
      _size = buf.st_size;
      _offset = 0;
    }
    _generateSeq = GENERATE_FILE;
    returnVal = (_fileId > 0 ? 1:0);
  }
  return returnVal;
}

void PatternIO::closeFile()
{
  if (_generateSeq == GENERATE_FILE) {
    if (_fileId > 0)
      close(_fileId);
    _fileId = -1;
  } else if (_generateSeq == GENERATE_STRING) {
      cout << "Selected: Generate String:" << endl;
    delete []_repSeq;
      cout << "Selected: Generate String:" << endl;
    _repSeq = NULL;
  }

  _generateSeq = GENERATE_NONE;

}

char PatternIO::getBase(long position)
{
  char returnVal;
  switch (_generateSeq) {
  case GENERATE_FILE:
    // Not going to use a buffer right now. Just do an lseek and grab it.
    if (_fileId > 0) {
      lseek(_fileId, position, SEEK_SET);
      read(_fileId, &returnVal, 1);
    }
    break;
  case GENERATE_UNIFORM:
    // Don't know yet.
    returnVal = _bases[random()%4];
    break;
  case GENERATE_STRING:
    returnVal = _repSeq[position % _repLength];
    break;
  }

  return returnVal;
}
```

A.2.6 fit.h

```
#ifndef FIT_H
#define FIT_H

void fit(double* x, double* y, int ndata, double* sig, int mwt,
 double *a, double *b, double *siga, double *sigb,
 double *chi2, double *q);


#endif
```

A.2.7 fit.c

```
#include <math.h>
#include "fit.h"

static float sqrarg;
#define SQR(a) ((sqrarg=(a)) == 0.0 ? 0.0 : sqrarg*sqrarg)

float gammq(float a, float x)
{
  printf("Error: gammq not implemented.");
  return 0.0;
}

void fit(double *x, double *y, int ndata, double *sig, int mwt,
 double *a, double *b,
 double *siga, double *sigb, double *chi2, double *q)

/*
   Given a set of data points x[1..ndata], y[1..ndata] with individual
standard
```

```
   deviations sig[1..ndata], fit them to a straight line y=a +bx by
minimizing
   chi square. Returned are a,b and their respective probable
uncertainties siga
   and sigb, the chi-square chi12, and the goodness-of-fit probability q
(that
   the fit would have chi2 this large or larger). If mwt=0 on input, then
the
   standard deviations are assumed unavailble: q is returned as 1.0 and
the
   normalization of chi12 is to unit standard deviation on all points.

   Taken from Numerical Recipes in C.
*/

{
  int i;
  double wt,t,sxoss,sx=0.0,sy=0.0,st2=0.0,ss,sigdat;

  *b=0.0;
  if (mwt) {
    ss=0.0;
    for (i=1;i<=ndata;i++) {
      wt=1.0/SQR(sig[i]);
      ss += wt;
      sx += x[i]*wt;
      sy += y[i]*wt;
    }
  } else {
    for (i=1;i<=ndata;i++) {
      sx += x[i];
      sy += y[i];
    }
    ss = ndata;
  }
  sxoss=sx/ss;
  if (mwt) {
    for (i=1;i<=ndata;i++) {
      t=(x[i]-sxoss)/sig[i];
      st2 += t*t;
      *b += t*y[i]/sig[i];
    }
  } else {
    for (i=1;i<=ndata;i++) {
      t=x[i]-sxoss;
      st2 += t*t;
      *b += t*y[i];
    }
  }
  *b /= st2;
  *a =(sy-sx*(*b))/ss;
  *siga=sqrt((1.0+sx*sx/(ss*st2))/ss);
  *sigb=sqrt(1.0/st2);
  *chi2=0.0;
  if (mwt==0) {
    for (i=1;i<=ndata;i++)
      *chi2+=SQR(y[i]-(*a)-(*b)*x[i]);
    *q=1.0;
    sigdat=sqrt((*chi2)/(ndata-2));
    *siga *=sigdat;
    *sigb *=sigdat;
  } else {
    for (i=1;i<=ndata;i++)
      *chi2 += SQR((y[i]-(*a)-(*b)*x[i])/sig[i]);
    *q=gammq(0.5*(ndata-2),0.5*(*chi2));
  }

}
```

IMAGE EVALUATION TEST TARGET (QA-3)