UNIVERSITY OF SOUTHAMPTON

THE ORIGIN OF LANGUAGE-LIKE FEATURES IN DNA

Allan Christopher Hurworth

Submitted for the degree of Doctor of Philosophy

FACULTY OF SCIENCE

CHEMISTRY

August 2000

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF SCIENCE

CHEMISTRY

Doctor of Philosophy

THE ORIGIN OF LANGUAGE-LIKE FEATURES IN DNA

Allan Christopher Hurworth

Non-coding DNA is known to account for a significant proportion of the genomes of many organisms. The discovery by the use of three statistical tests – the Zipf analysis, the fluctuation analysis and the Shannon entropy – of linguistic features and long-range correlations within non-coding DNA has given rise to the suggestion that this higher-order structure may form the basis of a biological language in non-coding DNA.

This work describes the development of a model to explain the origin of these language-like features in DNA. The model is based on observed genome reshaping processes – namely, transposable element insertion/excision events and point mutations – and involves the repeated duplication of transposable element target sites to build up repetitive blocks of DNA.

This model shows that the observed language-like features can be generated by known genetic rearrangements and therefore suggests that any function of non-coding DNA has been acquired opportunistically, through the use of these language-like features.

Contents

Abstract 2
Contents 3
Acknowlegements 6
Abbreviations 7
1 Non-coding DNA 8
1.1 Structures in Non-coding DNA 9
2 The Statistical Properties of DNA 14
2.1 Long-range Correlations 15
2.1.1 Fluctuation Analysis 15
2.2 Linguistic Features 20
2.2.1 The Zipf Plot 20
2.2.2 Shannon Entropy 25
3 The Possible Origins of Linguistic Features in DNA 27
3.1 Genome Reshaping Processes 27
3.1.1 Point Mutations 27
3.1.2 Slippage 29
3.1.3 Transposable Elements 29
3.1.4 Recombination 34
3.1.5 Summary 35
3.2 Modelling the Formation of Linguistic Features 36
3.2.1 Generalised Lévy Walk Model 36
3.2.2 Insertion–Deletion Model 38
3.2.3 Markov Processes 39
4 The Transposable Element Model 40
4.1 Sequence Generation 42
4.1.1 Starting Sequences Used 42
4.1.2 Target Site Selection 42

4.1.3 Target Site Duplication 43
4.1.4 Iteration 43
4.2 Sequence Analysis 44
4.3 Results 44
4.4 Initial Observations 44
4.5 The Effect of Iteration 49
4.6 Effect of Word Length 56
4.7 Effect of Copy Length 57
4.8 Effect of Target Site Bias 59
4.9 Effect of Starting Sequence 63
4.10 Variable Target Site Model 67
4.11 Differences Between Simulated and Real DNA 70
5 Initial Improvements to the Transposable Element Model 76
5.1 Initial Modifications 76
5.2 Comparison with Previous Results 80
5.3 Effect of Target Site Bias 82
5.4 Using More Target Sites 85
5.5 The Effect of Target Site Length 88
5.6 The Effect of Copy Length 96
5.7 The Fluctuation Anomaly 102
5.8 Further Modifications 108
6 Further Improvements to the Transposable Element Model 110
6.1 Point Mutations 110
6.2 Transitions vs Transversions 112
6.3 The Effect of Point Mutation and Transposable Element Insertions 116
6.4 Imprecise Insertion and Excision of Transposable Elements 124
6.5 The Effect of Imperfect Transposable Element Excisions 126
6.6 The Effect of a Range of Insertions and Deletions 134
7 Final Improvements to the Transposable Element Model 139
7.1 The Effect of Variable Copy Length 139
7.2 The Effect of Independent Excision Parameters 144

7.3 The Effect of Modelling Transposable Element End Sequences 149
8 Final Results and Conclusions 155
8.1 Future Work 166
Appendix: Program listings 170
5.1 dnasim2.c 170
5.2 dnasim2cfg.h 200
References 205

Acknowlegements

First of all, thanks go to George Attard, my supervisor. The fact that the original idea behind this work is his, coupled with the countless suggestions and help he has provided along the way mean that this project simply wouldn't have been possible otherwise. I'd also like to thank George for his considerable patience when dealing with my awful adherence to deadlines and general reliability over the last 5 years!

Thank you to the members of the Attard group past and present, Jason, Marcus, Nick (for windsurfing and kebabs), Stephane (for letting me win at squash), Steve, and the others for keeping me entertained on the rare occasions I have been in the office! Particular thanks go to ex-group-member Jon Jack who helped start the work which has become this thesis and who made the Zepler basement a more sociable place to be.

Extra special thanks to Lucy, without who's support (emotional and financial!) this certainly wouldn't have been finished. Also to friends who have helped me to forget DNA - Andy, Coops, DC, Em, Helen, Luan, Marky, Nick and all the rest.

Finally and most importantly, to my parents. Thank you for everything. This is for you. (And yes, Mum, it *is* finished now!)

Thanks, everyone!

Abbreviations

A	adenine
ANSI	American National Standards Institute
bp	base pairs
C	cytosine
DNA	deoxyribonucleic acid
cDNA	complementary deoxyribonucleic acid
exon	expressed sequence
G	guanine
HTLV II	human T-cell leukaemia virus type II
intron	intervening sequence
kb	kilobases
Kbyte	kilobyte
LINE	long interspersed nuclear element
MHC	myosin heavy chain
nt	nucleotide
RNA	ribonucleic acid
mRNA	messenger ribonucleic acid
T	thymine
TE	transposable element

1 Non-coding DNA

DNA is the molecular store of genetic information; the design from which an organism is constructed. This design is stored in the sequence of bases making up the DNA molecule, and acts as the template for the synthesis of the proteins that characterise a given organism. The DNA sequences which code for proteins are organised into genes, but between, and indeed within, these genes is a remarkable amount of DNA which does not code for proteins – so-called non-coding DNA.

The proportion of non-coding DNA within an organism's genome varies from species to species; in humans around 97% of the genome is non-coding. The proportion of non-coding DNA is not directly related to an organism's complexity; salamanders have a greater proportion of non-coding DNA than humans[1], while the puffer fish has a remarkably low non-coding to coding DNA ratio in comparison with other vertebrates[2]. The proportion of non-coding DNA is related to the genome sizes of various species – the genomes of salamanders are around forty times the size of the human genome, while the puffer fish genome is about an eighth of the size of those of other vertebrates. Although there are some differences in the amount of coding DNA from species to species, these cannot account for such large variations in genome size.

The variation in the proportion of non-coding DNA from species to species has lead to a debate on whether or not non-coding DNA has some important role within the genome[1, 3]. It seems unlikely that organisms such as humans and salamanders would tolerate such a large proportion of excess DNA if it served no useful purpose – replication of the genome is an expensive process in terms of cell resources, and has to be carried out at every round of cell division. Indeed, studies have shown that organisms that have high rates of cell division have less

non-coding DNA relative to those with slower development rates[1]. This suggests that organisms will tolerate the presence of non-coding DNA up to a certain point, beyond which the cost of repeated duplication becomes too high. The argument against a function for non-coding DNA stems from the inference that non-coding DNA cannot be essential if some organisms, such as the puffer fish, can virtually do without it.

Why some organisms have so much non-coding DNA is clearly a mystery, although several theories to explain its presence have been put forward. It has been proposed that non-coding regions of the genome may be areas that contain biological codes of some kind, possibly to initiate the unwinding of stretches of DNA from the nucleosome proteins which help packaging - this unwinding is necessary to allow the transcription apparatus access to a gene[4]. Other suggestions are that the non-coding stretches are important in determining the three-dimensional structure of DNA, through their effect on the distances between coding sequences. The three-dimensional structure of DNA seems to be an important factor affecting initiation of transcription (see Figure 2, later)[5,6]. Many simply believe that non-coding DNA has no function whatsoever, and is no more than 'junk DNA'.

1.1 STRUCTURES IN NON-CODING DNA

Non-coding DNA can be found both between and within the genes of an organism. The regions between the genes are known as intergenic spacers, while areas of non-coding DNA which occur within genes are termed introns (intervening sequences) - these are present alternating with the discontinuous coding exons (expressed sequences) which make up the gene. Figure 1 shows the relationship between intergenic spacers, introns, and exons. Within the regions of non-coding DNA are well-characterised sequences, some of which are known to have specific functions.

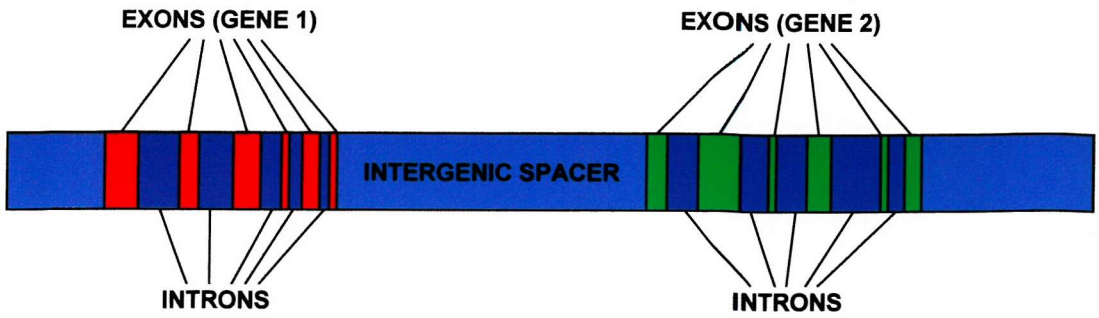


Figure 1 Introns, exons, and intergenic spacers

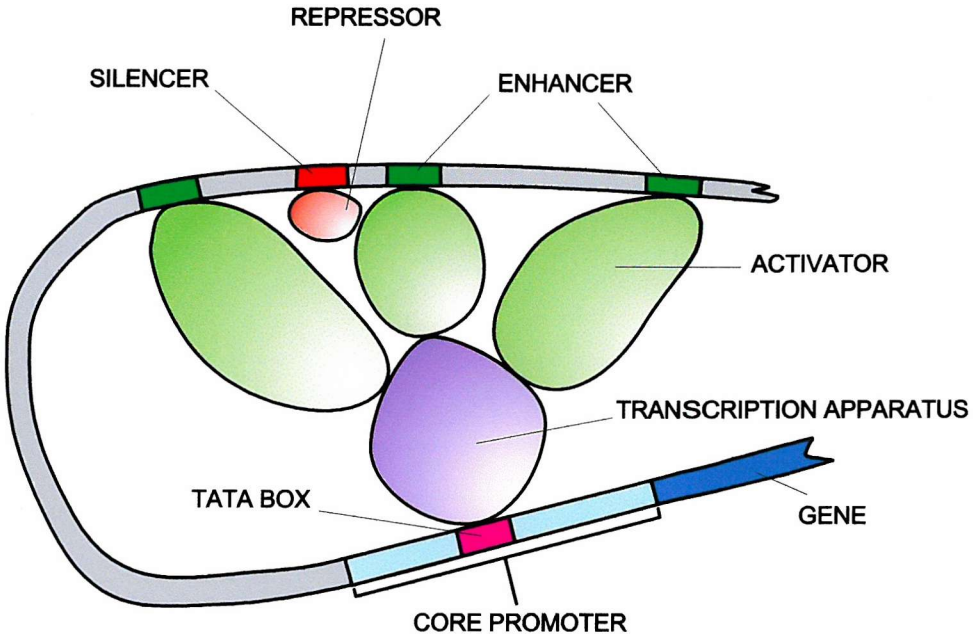


Figure 2 DNA–protein interactions Involved in Gene Regulation

Many of these sequences are related to the expression of genes. The frequency at which a gene is transcribed into RNA before translation into the corresponding protein is critical in regulating the final amount of the protein in the cell. The amount of the protein produced may need to change under certain conditions, and in multi-cellular organisms in which the entire genome is present in most

cells, a given cell need only make use of certain genes. This control is achieved through gene regulator sequences, present in the intergenic spacers. In eukaryotes these regulator sequences consist of the core promoter, usually situated almost immediately preceding the coding domain, and silencers and enhancers, which can be several thousand nucleotides distant from the core promoter[5, 6]. The core promoter is the site where binding of the transcription apparatus takes place, and includes the TATA box, a conserved binding sequence within the core promoter[5, 6]. Enhancer sequences bind proteins known as activators, which promote binding of the transcription apparatus to the core promoter. Silencers bind proteins known as repressors, which seem to interfere with the action of activators. The interaction between these components is shown in Figure 2. Examples of transcription systems in eukaryotes include the heat shock transcription factor system in *Drosophila*; the gal4 protein stimulated transcription of galactose metabolising enzymes in yeast[7]; and the glutamine synthatase system in mice[8].

Introns contain conserved sequences that mark their beginning and end, known as splice sites. In particular, introns always begin with the bases GT, and end with AG. These sites are involved in the removal of introns from newly transcribed RNA sequences. This removal of introns is necessary to generate mRNA for translation that contains only the information required for protein synthesis. The splice sites are recognised by protein–RNA complexes known as spliceosomes, which catalyse the removal of introns from the initial RNA transcript, while stitching together the exons to form mRNA. The resulting mRNA molecule is then ready for translation, containing only coding information for the appropriate protein[9].

Perhaps the most striking features found in the non-coding regions of eukaryotes, however, are the large numbers of repeated sequences[10, 11]. These include satellite, minisatellite, and microsatellite[12-25] sequences – so-called

because of the satellite bands they produce in CsCl density-gradient centrifugation relative to the main band formed by most of the DNA - and transposable elements. The satellite, minisatellite, and microsatellite sequences are repeated in tandem, while transposable elements are dispersed. Satellite sequences are highly repetitive with a repeat length of 5–10 base pairs (bp), or longer (~100 bp), and can have total lengths as long as 10^8 bp. Micro- and minisatellite sequences are less repetitive, with repeat lengths of 2–5 bp and ~15 bp respectively, and total lengths of ~400 bp and 0.5–30 kb respectively[10]. Table 1 gives examples of minisatellite, microsatellite and satellite sequences.

Repetitive sequence	*Repeat length (bp)*	*Total length*	*E.g. (Name) (Sequence)*
Minisatellite	~15	0.5kb-30kb	XTA of *Xenopus laevis*[19]: CCAACAGGCCTGCCCATCCAT
Microsatellite	2-5	100bp	GA of Tomato[26]: GA
Satellite	5-10 or ~100	Up to 10^8bp	Satellite I of *Drosophila virilis*[7]: ACAAACT

Table 1 Minisatellite, microsatellite and satellite sequences

Transposable elements are mobile sequences, capable of inserting themselves into new locations in the genome. They can be broadly split into two categories, according to the mechanism by which they move[7]. Retroelements, also known as retrotransposons, or retroposons (e.g. *LINE*s, the *gypsy* and *copia* elements of *Drosophilia*, yeast *Ty* elements) insert themselves into DNA via an RNA intermediate through the use of reverse transcriptase, an enzyme which catalyses the formation of DNA from an RNA template (the reverse of transcription); these are very similar to the retroviruses[27-29]. Indeed, it has been postulated that

retrotransposons originate from retroviral genomes that become part of the host germ-line DNA in a manner analogous to that of endogenous retroviruses e.g. mouse mammary tumour virus. The second group of transposable elements known as transposons (e.g. *mariner* elements, *Alu* sequences, *hobo* elements of *Drosophilia*) exist as DNA throughout their transposition cycle.

Repetitive sequences such as those described above are present in large numbers within the genomes of eukaryotes, indeed, they are thought to account for over 30% of the human genome – around ten times more DNA than the coding sequences. In certain species the proportion of repetitive DNA is far higher e.g. around 80% in the case of *Vicia faba*[30].

2 The Statistical Properties of DNA

Statistical analysis has revealed several differences between the coding and non-coding regions of DNA. These differences are interesting in that they indicate that a higher order structure, like that found in human languages, might be present in *non-coding* DNA, but is virtually absent in areas of *coding* DNA. These findings seemed to add strength to the argument that non-coding DNA may represent some previously undiscovered biological language.

The first of these findings concerned the presence of long-range correlations within DNA sequences[31-37]. The analysis carried out showed that long-range correlations were present in the non-coding DNA sequences studied, but were absent in the coding regions. The importance of this observation is that long-range correlations suggest a structure, which could be the basis for a biological code in non-coding areas of the genome.

In 1994, Mantegna et al published their work which described the presence of "linguistic" features in DNA sequences[3, 38, 39]. The authors applied two tests, the Zipf test, and the Shannon entropy calculation and found that DNA sequences gave results similar to those obtained when the tests are applied to human languages. It was observed that these language-like characteristics were far more apparent in non-coding sequences than in the coding regions.

These findings have prompted the extensive use of statistical methods in the analysis of DNA sequences, with a view to identifying and explaining the apparent language-like nature of non-coding DNA.

2.1 LONG-RANGE CORRELATIONS

2.1.1 Fluctuation Analysis

Fluctuation analysis is used to identify the presence of long-range sequence correlations[40]. The basis of the fluctuation analysis is to take a sequence, and by splitting the sequence into units of a given length, to calculate the difference between each individual unit and the 'average' unit. This difference is termed the fluctuation, and by analysing the effect of unit length on the fluctuation it is possible to determine if long-range correlations exist in the sequence[31-37].

To analyse DNA in this way it is first necessary to map the base sequence onto a numerical sequence to produce a 'DNA walk' - basically a mathematical representation of the DNA. This involves 'walking' along the DNA string, one base at a time, and at each step i along the 'walk' altering the value y (termed the net displacement) by a value $u(i)$, according to the nature of the base. The most common mapping rule employed is the purine-pyrimidine (RY) rule, in which purines (adenine and guanine) result in $u(i) = -1$, and pyrimidines (thymine and cytosine) $u(i) = +1$. So, for a 'walk' of l steps, $y(l)$ is given by:

$$y(l) \equiv \sum_{i=0}^{l} u(i) . \tag{1}$$

Using this quantity, the root mean square fluctuation $F(l)$ can be calculated from the square root of $F^2(l)$:

$$\begin{aligned} F^2(l) &\equiv \overline{\left[\Delta y(l) - \overline{\Delta y(l)}\right]^2} \\ &= \overline{\left[\Delta y(l)\right]^2} - \left[\overline{\Delta y(l)}\right]^2 , \end{aligned} \tag{2}$$

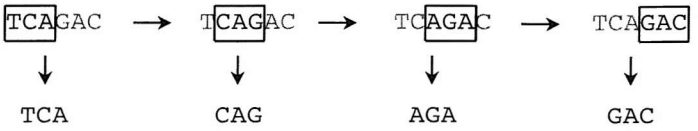
where:

$$\Delta y(l) \equiv y(l_0 + l) - y(l_0), \tag{3}$$

and the bars in equation (2) indicate averaging over all positions l_0 in the sequence. In essence the procedure involves taking a window of length l, and moving its start point (l_0) along the sequence one base at a time, calculating the value Δy for each window. On reaching the end of the sequence, the Δy values are used to obtain $F(l)$ through equation (2). The use of the fluctuation analysis on a short sequence is demonstrated in Figure 3.

The value $F(l)$ can be used to identify the presence of long-range correlations in the sequences studied. If the sequence is random, or contains correlations which extend up to a characteristic range, then $F(l) \propto l^{1/2}$. Alternatively, if the sequence contains correlations which extend over all measured length scales then $F(l) \propto l^{\alpha}$, with $\alpha > ½$. This behaviour of $F(l)$ is assessed by plotting log $F(l)$ against log l and measuring the slope to obtain α. An example of such a plot carried out on a DNA sequence is shown in Figure 4.

STAGE 1 **Split sequence into sub-sequences**



STAGE 2 **Apply RY rule to sub-sequences**

RY rule; T and C = +1, A and G = -1

e.g. TCA = +1+1-1=+1

	TCA	CAG	AGA	GAC
$\Delta y(l)$	+1	-1	-3	-1
$\Delta y(l)^2$	+1	+1	+9	+1

STAGE 3 **Calculate** $F(l)$

a) From stage 2: $\overline{\Delta y(l)} = \frac{1-1-3-1}{4} = -1$ From stage 2: $\overline{[\Delta y(l)]^2} = \frac{1+1+9+1}{4} = 3$

$$\left[\overline{\Delta y(l)}\right]^2 = 1$$

b) $$F^2(l) = \overline{[\Delta y(l)]^2} - \left[\overline{\Delta y(l)}\right]^2$$

$$= 3 - 1$$

$$F(l) = \sqrt{2}$$

Figure 3 The fluctuation analysis of a short sequence TCAGAC. Each window length generates one data point on the fluctuation plot, this example shows how the value would be calculated for the point at window length 3. In stage 1 the sliding window of length 3 is used to produce the 4 sub-sequences shown. Then, in stage 2, Δy(l) is calculated for each sub-sequence using the RY rule (purine -1, pyrimidine +1). Each Δy(l) value is also squared to give a list of $\Delta y(l)^2$ values, one for each sub-sequence. In stage 3a, the average values of Δy(l) and $\Delta y(l)^2$ over all sub-sequences are calculated. The average value obtained for Δy(l) is squared. Finally, in stage 3b the value of $F(l)$ is calculated. This calculation results in a value of √2 at window length 3. In order to ascertain whether a DNA sequence contained long-range correlations, the above procedure would be repeated at a number of different window lengths, l, and a plot of log $F(l)$ versus log l obtained. Long range correlations would be indicated by a line of gradient > ½ on this plot.

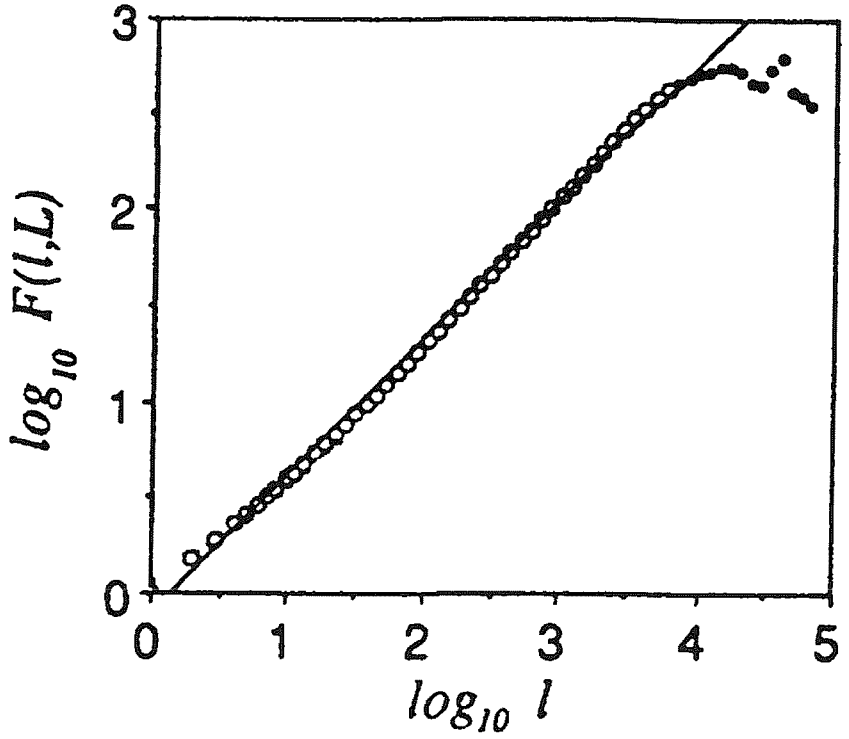


Figure 4 Fluctuation plot of a non-coding DNA sequence (adapted from reference 33). Plot of the entire human beta-globin intergenomic sequence (HUMHBB) (non-coding). The slope of the linear fit (omitting solid circles) is 0.71.

Application of this analysis to DNA sequences[31-37] has shown that the coding regions of sequences show values of α of $\approx \frac{1}{2}$, which would indicate no correlations, or correlations over a characteristic length scale. In fact the latter is true, as the coding regions are structured from triplet codons, which generate correlations over a well-defined length scale. Non-coding DNA sequences, however, give values of α which are consistently greater than $\frac{1}{2}$, indicating the presence of long-range correlations within these sequences. Examples of results obtained from fluctuation analysis are shown in Table 2 (some of the sequences are from cDNA – this is complementary DNA reverse transcribed from a mRNA transcript after removal of introns from the RNA i.e. a DNA gene sequence with the introns removed.)

Sequence	Code	Comments	Length (nt)	% Introns	α
GROUP A					
Adenovirus type 2	ADBCG	Intron-cont. virus	35,937	n.d.	0.56
Caenorhabditis elegans MHC 1	CELMY01A	Gene	12,241	541	0.61
C. elegans MHC gene 2	CELMY02A	Gene	10,780	44	0.54
C. elegans MHC gene 3	CELMY03A	Gene	11,621	49	0.61
C. elegans MHC unc 54 gene	CELMYUNC	Gene	9,000	25	0.58
Chicken c-*myb* oncogene	CEKMYB15	Gene (5'-end)	8,200	92	0.60
Chicken embryonic MHC	CHKMYHE	Gene	31,111	78	0.65
*Drosophila melanogaster*MHC	DROMHC	Gene	22,663	72	0.56
Goat β-globin	GOTGLOBE	Gene	10,194	n/a	0.58
Human β-globin	HHUMHBB	Chromosomal region	73,326	n/a	0.71
Human metallothionein	HUMMETIA	Gene*	2,941	91	0.61
Human α-cardiac MHC	HUMMHCAG1	Chromosomal region	2,366	72	0.65
Human β-cardiac MHC	HUMBMYH7	Gene	28,438	73	0.67
Rat embryonic skeletal MHC	RATMHCG	Gene	25,759	76	0.63
			$\alpha_{mean}\pm$(2 s.e.m.)=0.61±0.03		
GROUP B					
Bacteriophage λ	LAMCG	Intronless Virus	48,502	0	0.53
Chicken c-*myb* oncogene	CHKCMYBR	cDNA	2,218	0	0.50
Chicken nonmuscle MHC	CHKMYHN	cDNA	7,003	0	0.47
Dictiostelium discodium MHC	DDIMYHC	cDNA	6,681	0	0.49
Drosophilia melanogaster MHC	DROMYONMA	cDNA	6,338	0	0.47
Human β-cardiac MHC	HUMBMYH7CD	cDNA	6,008	0	0.49
Human dystrophin	HUMDYS	cDNA	13,957	0	0.53
Human embryonic MHC	HUMSMHCE	cDNA (partial)	3,382	0	0.51
Human mitochondrion	HUMMT	Intronless gene	16,569	0	0.49
Yeast MHC	SCMY01G•	Intronless gene	6,108	0	0.50
			$\alpha_{mean}\pm$(2 s.e.m.)=0.50±0.01		

Table 2 Fluctuation analysis of DNA (From reference 32). Sequences are divided into two groups on the basis of their intron content; within each group the sequences are ordered alphabetically. Note that $\alpha > 0.5$ implies the existence of long-range correlations, whereas $\alpha \approx 0.5$ implies only short-range correlations. The second column (code) lists the GenBank names (unless specified otherwise.)

nt, Number of nucleotides per sequence regions
*, eta-globin activating region (nontranscribed DNA)
n.d., no data; exon/intron map not fully known
n/a, not applicable, nontranscribed DNA
MHC, myosin heavy chain
•, EMBL name

Long-range correlations arise due the presence of a high-order structure in the sequence studied - there is a relationship which occurs between individual units of a sequence no matter how far apart they are. This sort of structure is present in languages, and so the presence of long-range correlations in non-coding regions of DNA suggests a possible biological code. Of course, just because a sequence shows long-range correlations doesn't mean that it is a language; sequences generated from electronic voltages, traffic, economic data, and music all display long-range correlations.

2.2 *LINGUISTIC FEATURES*

2.2.1 *The Zipf plot*

The Zipf plot was devised by the linguist George Zipf, who applied his test to several languages[41]. The test involves counting the frequency of occurrence of each word in a given text, and assigning each word a rank, the most common word being rank 1, the second most common word rank 2, and so on up to the least common word rank M. The log of the probability of each word is then plotted against the log of its rank. Somewhat surprisingly this gives a straight line, with slope $-\zeta$. The slope can vary depending on the text studied, but typical values are in the range $\zeta = 0.7$–1.6. Stated mathematically, the relationship between word probability $P(R)$ and word rank R obeys a power law:

$$P(R) = AR^{-\zeta}, \tag{4}$$

where A is a constant. The exponent $-\zeta$ is obtained from the slope on a log-log plot of $P(R)$ against R. The value of A can be determined from the normalisation condition:

$$\sum_{R=1}^{M} P(R) = 1,$$

such that:

$$(A)^{-1} = \sum_{R=1}^{M} R^{-\zeta} .$$

Figure 5 shows the Zipf analysis carried out on a human language.

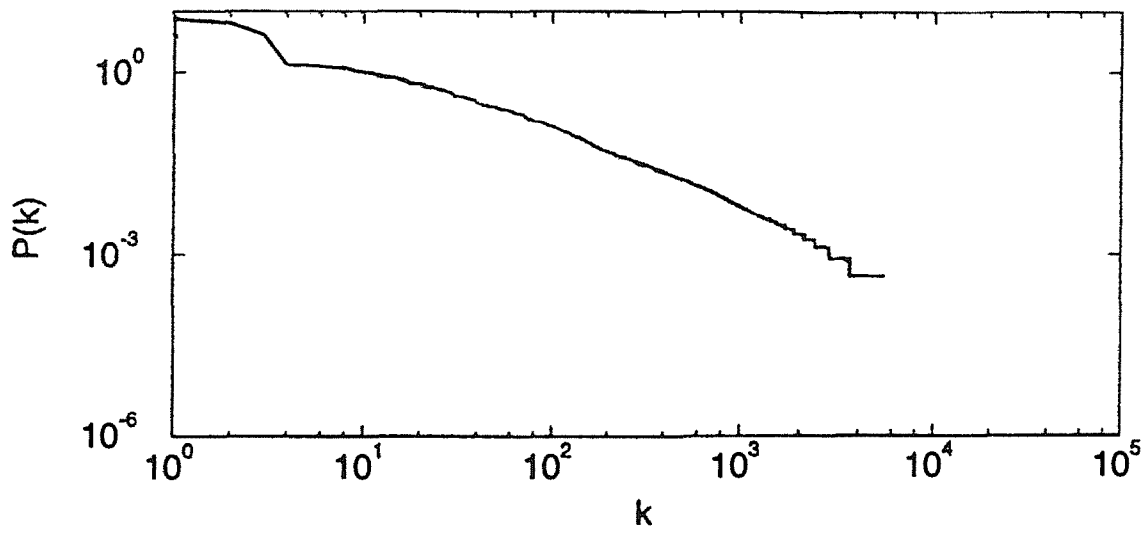


Figure 5 Zipf Analysis carried out on the Bible (Adapted from Reference 43)

This test can be used to analyse sequences of DNA, although a slightly modified method has to be applied, given that aside from the triplet codons in coding regions it is not known if any "words" exist in DNA sequences, let alone what they might be. The DNA sequence is split into "words" of a given length n (where n is usually in the range 3–8) by passing a window of length n along the sequence one character at a time. For a sequence of length l, this will give $l-(n-1)$ "words". Plots of log frequency against log word rank give a slope which is linear over a significant portion, with ζ in the range 0.2–0.5. Notably, the values of ζ achieved for non-coding sequences are consistently higher than those achieved for coding sequences, as shown in Table 3.

Although the values obtained for non-coding sequences are still lower than those for human languages, this is at least partly a result of the different ways in which words are defined in each case. If human text is analysed in the same way as a DNA sequence, by using a sliding window to generate "words" (the n-tuple approach), then the value ζ obtained is lower than when the text is split up into real words. For example, a standard Zipf test on a number of articles taken from an encyclopaedia gave $\zeta = 0.85$, but when the test was carried out using the n-tuple approach $\zeta = 0.57$ [39]. If the Zipf plot is carried out on a random sequence, a slope $\zeta \approx 0$ is obtained.

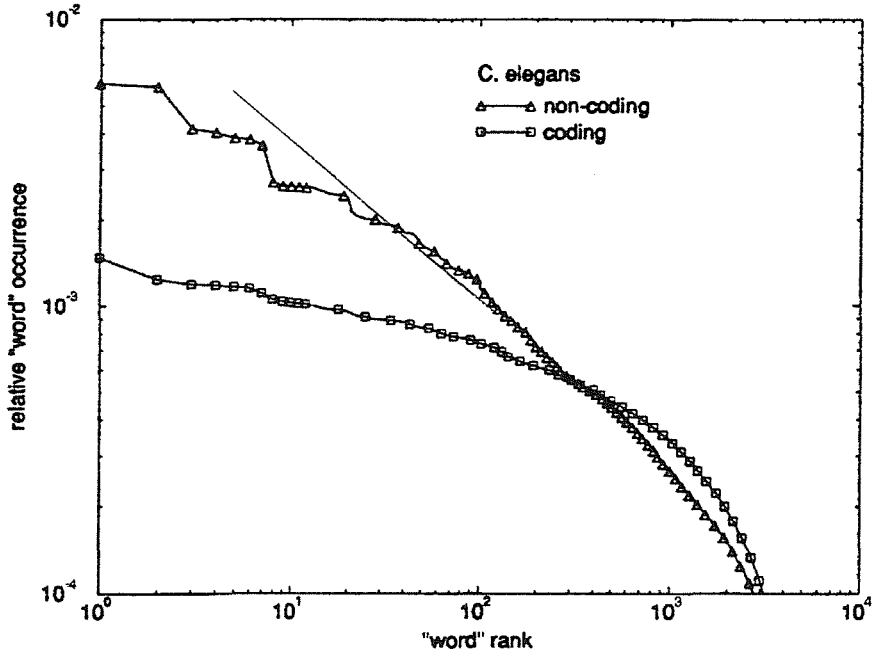


Figure 6 Zipf plots of the coding and non-coding regions of *C. elegans* (from reference 3). The Zipf exponent is measured at 0.54 for the coding region and 0.24 for the non-coding sequence. Word length 6 used.

	Length	% coding	ζ	H(4)
I. Eukaryotes				
A. Mammals (14 sequences)				
All	1,078,100	5	0.283±0.002	7.784
Coding	50,687	100	0.208±0.004	7.832
Noncoding	1,027,413	0	0.289±0.002	7.776
B. Invertebrates				
1. C. elegans				
Complete sequence	2,176,983	29	0.465±0.002	7.648
Coding	633,029	100	0.244±0.004	7.776
Noncoding	1,543,954	0	0.537±0.003	7.552
2. Other invertebrates (3 sequences)				
All	120,966	32	0.403±0.004	7.696
Coding	38,361	100	0.21±0.01	7.768
Noncoding	82,605	0	0.477±0.006	7.592
C. Yeast chromosome III				
Complete sequence	315,338	67	0.289±0.003	7.808
Coding	211,091	100	0.225±0.005	7.840
Noncoding	104,247	0	0.391±0.005	7.680
II. Eukaryotic viruses (11 sequences)				
All	1,616,928	84	0.263±0.002	7.936
Coding	1,361,411	100	0.194±0.001	7.960
Noncoding	255,517	0	0.362±0.003	7.880
III. Prokaryote (7 sequences)				
All	784,344	83	0.203±0.002	7.896
IV. Bacteriophages (3 sequences)				
All	140,735	87	0.158±0.003	7.936

Table 3 Results from the Zipf analysis of coding and non-coding sequences (adapted from reference 3). Linguistic analysis of the 37 sequences in GenBank Release No. 81.0 of 154 February 1994 that exhibit more than 50,000 base pairs (bp) apiece. These 40 sequences are partitioned into groups: Group IA contains 14 sequences from mammals. Group IB contains the 74 sequences comprising part of the 2.2×10^6 bp sequence of chromosome III of *C. elegans*[42], and 3 sequences of invertebrate. Group IC consists of the 315,338 bp yeast chromosome III sequence. Group II contains 11 sequences of eukaryotic viruses. Group III Contains 7 sequences of bacteria, while group IV contains 3 sequences of phage. The groups are arranges in increasing order of percent of *coding* regions. H(4) is the Shannon entropy for the sequence using word length 4.

Do the Zipf plots of non-coding DNA, with coefficients similar to those of human languages, mean that there is a language in non-coding DNA? In fact, Zipf language-like behaviour is observed in a wide range of situations in which there is no underlying language present. Population sizes of cities, the surface areas of islands and the distribution of income all give Zipf plots that display the properties of human languages. It has also been shown that two-parameter Markov processes, which involve biased random transitions between states, can generate Zipf behaviour[43]. It cannot, therefore, be concluded that a source that is characterised by a non-zero Zipf exponent ζ is in some way a language - but a source that is a language will undoubtedly show Zipf law behaviour.

The basic shape of the Zipf plot (a negative gradient) is generated by a source in which certain "words" are more common than others. In order to generate a plot in which there is a large gradient there must be a large bias towards certain "words", and against others; this does not occur in the random case in which each "word" is present at roughly the same frequency as all others. This bias is seen in languages - certain words occur far more frequently than others, and so a slope is generated with a large coefficient. This is due to the fact that languages are highly structured - only certain combinations of letters, words, and sentences are allowed if the language is to make sense and, as a result, certain words, letters and syllables are far more common than others. It is the absence of this type of hierarchical structure which leads to the low value of ζ observed in the coding regions of DNA. Although the coding regions are not random, they do not show the so-called higher order structures found in languages, having more 'freedom' in how "words" can be arranged, and so generate lower values of ζ.

A species must be capable of evolutionary change to be successful, and this is achieved mainly through the mutation of coding DNA sequences. By lacking higher-order structure, coding DNA sequences are not subjected to strict rules that would prevent certain arrangements of codons occurring. The 'language' of

coding DNA tolerates many different "word" combinations, and so can potentially code for many different proteins, allowing maximum flexibility in the range of proteins that can be produced.

At a trivial level the nature of the structure seen in non-coding regions could be related to the presence of repetitive elements which are known to occur within these sequences, as described earlier. Repetition of certain motifs will increase their proportion within a sequence studied, resulting in a bias of observed "words" which will generate Zipf plots with high values of ζ. As will be shown later, genome-reshaping processes that produce such tandem repeats provide good models for the origin of non-coding DNA.

2.2.2 *Shannon Entropy*

The Shannon entropy is another statistical approach that can give information on the complexity of a sequence. As with the Zipf analysis, the Shannon entropy is determined for DNA sequences by first splitting the sequence into units of length n, using a sliding window. The Shannon n-entropy ($H(n)$) is defined as[3]:

$$H(n) = -\sum_{i=1}^{\lambda^n} p_i \log_2 p_i , \qquad (5)$$

where p_i is the probability of a "word", i is the index for a "word", and λ is the number of letters in the alphabet of the sequence being studied (i.e. 4 for DNA). The summation is therefore over all the "words" (n-tuples) generated from a sequence. Because the Shannon entropy uses the same distribution of "words" produced by the sliding window as the Zipf test, the two tests are not independent. A sequence giving a high Zipf exponent ζ will also give a low value of $H(n)$.

When this test is carried out on DNA sequences it is found that non-coding regions show lower $H(n)$ than coding regions. Some examples are given in Table 3. The significance of this result is that low values of $H(n)$ indicate strongly biased systems, whereas high values correspond to systems that are more random in nature. The entropy can be thought of as a measure of the information content of a sequence, high $H(n)$ indicating high information content. The maximum value of $H(n)$ would be obtained for a purely random sequence.

The Shannon entropy is also used to determine the redundancy, $R(n)$ of a sequence,

$$R(n) = 1 - \frac{H(n)}{H'(n)}, \tag{6}$$

where $H'(n)$ is the theoretical Shannon entropy for a random sequence (i.e. a sequence in which each word has the same probability). Redundancy is the property of languages to often remain intelligible after the removal of a letter or word from a text. The lower the value of $H(n)$, the greater the degree of redundancy and the easier it would be to understand a text despite a missing word. The low values of $H(n)$ and relatively high redundancy of non-coding sequences once again highlight their language-like nature.

3 The Possible Origins of Linguistic Features in DNA

While in the past the genome was thought of as a highly stable, relatively unchanging system, current opinion suggests that this is far from the case. DNA must be well maintained, in order to make an organism viable, but variation is essential if the species is to adapt to evolutionary pressures. Alterations in the DNA can occur at several different levels. Point mutations are perhaps the most basic modification in which a single nucleotide is swapped for another. Insertion or deletion of bases can occur during replication of repetitive sequences, via a process known as strand slippage. Modifications on a far larger scale are also possible. Recombination occurs between homologous sequences, resulting in the movement of large stretches of DNA between sister chromosomes. Transposable elements are also active, mediating large-scale change within the genome.

3.1 GENOME RESHAPING PROCESSES

3.1.1 Point mutations[7,44]

Point mutations occur during DNA replication, when the DNA replicating enzyme inserts an incorrect base against the parent template strand. This leads to a base-pair mismatch, and although most such mismatches are removed by the proof-reading apparatus, some go undetected. On the next round of replication the strand containing the incorrect nucleotide will act as the template for a new DNA strand, and will result in a daughter DNA duplex with a different sequence from the original grandparent DNA. In the human genome it is thought that after proof-reading around 10 such point mutations occur per round of replication (about 10^{-9} mutations per nucleotide per generation[44]). In bacterial genomes the rate of mutation at a given nucleotide is approximately 10^{-9}-10^{-10} mutations per round of replication[7].

Point mutations may fall into one of two categories. The first are transitions, in which one purine (adenine or guanine) is substituted for the other purine, or one pyrimidine (thymine or cytosine) is substituted for the other pyrimidine. The second category is transversions in which a purine is swapped for a pyrimidine or vice versa. Transitions are more common than transversions, with a ratio of between 2-7 transitions for every transversion[44].

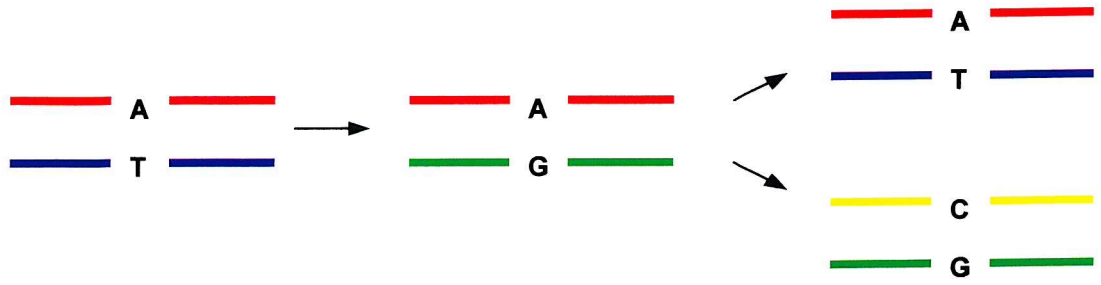


Figure 7 A mistmatch during DNA replication causes a point mutation. A mismatch occurs during the synthesis of the blue strand, resulting in A pairing with G. If this is not corrected then the red strand will go on to produce a DNA duplex with the same structure as the original, but the green strand will match a G against C, resulting in a change in the DNA sequence at that position, in comparison with the original grandparent strand. In this case the mutation is a transversion as the original T (pyrimidine) is substituted for G (a purine).

A point mutation may have one of a number of effects on an organism. Assuming the mutation occurs in a coding region, then a codon within that region will be altered. This can lead to a different amino acid sequence in the expressed protein, which can affect its function. Although most mutations that affect a protein's operation will impair its function, occasionally a mutation may result in a protein with characteristics that benefit the organism. Such mutations should lead to organisms that are favoured by natural selection.

Degeneracy in the genetic code reduces the effect of mutations on the protein product. Most amino acids are coded for by more than one codon, and these codons are generally similar to one another. In this way, if one of the nucleotides of the codon is changed the codon may still code for the same amino acid. This applies specifically to the final nucleotide position of the codon. If the final

nucleotide is changed then half of the 64 codons that make up the genetic code will always code for the same amino acid, and most of the remaining codons could code for the same amino acid as the original, if an appropriate substitution is made. Codons with similar sequences tend to code for amino acids with similar properties, so if mutation generates a codon for a different amino acid there is a good chance that the actual effect on the final protein will be minimal.

3.1.2 *Slippage*[45,46]

Slippage is a process by which short lengths of DNA (several base pairs) in repeated sequences may be deleted or duplicated. It is thought to be an important mechanism in the expansion and contraction of lengths of microsatellite sequence and acts to limit the size of tandemly-repeated blocks[10]. Slippage can occur within sequences which consist of tandem repeats, by the DNA of one strand moving laterally over the other to re-establish hydrogen bonding to another repeat unit further along the strand. This forms a 'bulge' in one or other strand (depending on which way the slippage occurs) which is targeted by repair mechanisms. This can yield either duplication or deletion of a length of DNA corresponding to the bulge. Slippage is a fairly common event thought to be responsible for the observed variations in the lengths of repetitive elements between individuals of a species, an estimation for the rate in *E. Coli* suggests one slippage mutation every 100 generations (approximately 10^{-8} per nucleotide per generation)[46]. This variation from one individual to another enables individuals to be distinguished from one another through the use of DNA-fingerprinting.

3.1.3 *Transposable elements*[7, 47, 48]

As described previously, transposable elements can be split into two main categories, depending on whether transposition occurs via DNA or RNA. Both types can have a major effect on the structures of DNA sequences.

Transposable elements that mobilise via DNA (transposons) include *mariner* elements, *hobo* elements of *Drosophilia*, and *Alu* sequences. Structurally these elements consist of a region that codes for the proteins involved in the transposition process, flanked by inverted repeats. Expression of these coding regions by the cell's normal transcription and translation apparatus provides the proteins necessary for transposition. Transposition of this type of element can occur by one of two methods, replicative or non-replicative. Replicative transposition involves duplication of the element before insertion of the copy into a new position, while in non-replicative transposition the element is excised from its original location and is inserted into its new location in a cut-and-paste mechanism. Characteristic to both types of transposition is duplication of the target sequence at which insertion occurs, this being due to staggered cutting of the DNA, shown in Figure 8. A consequence of target site duplication is that following excision of the transposable element the duplicated sequence and its original lie adjacent to each other, as shown. Excision is not always precise, however, and transposable elements can remove too many or too few bases, so that an exact duplication of the target sequence is not always observed[7, 59] (Figure 9, below). The length of the duplicated sequence varies from element to element (typical lengths are in the range 5–9)[49-53], as does the target site preference – some appear to insert at random, while others favour defined target sequences[54-57]. E.g. The *Tc4* and *Tc5* elements of *C. elegans* target the sequence CTNAG (where N represents any nucleotide)[58] while the *Tc3* transposons of *C. elegans* target the sequence TA[59]. *T2* elements of *Xenopus* insert at the target site TTAA.[60]

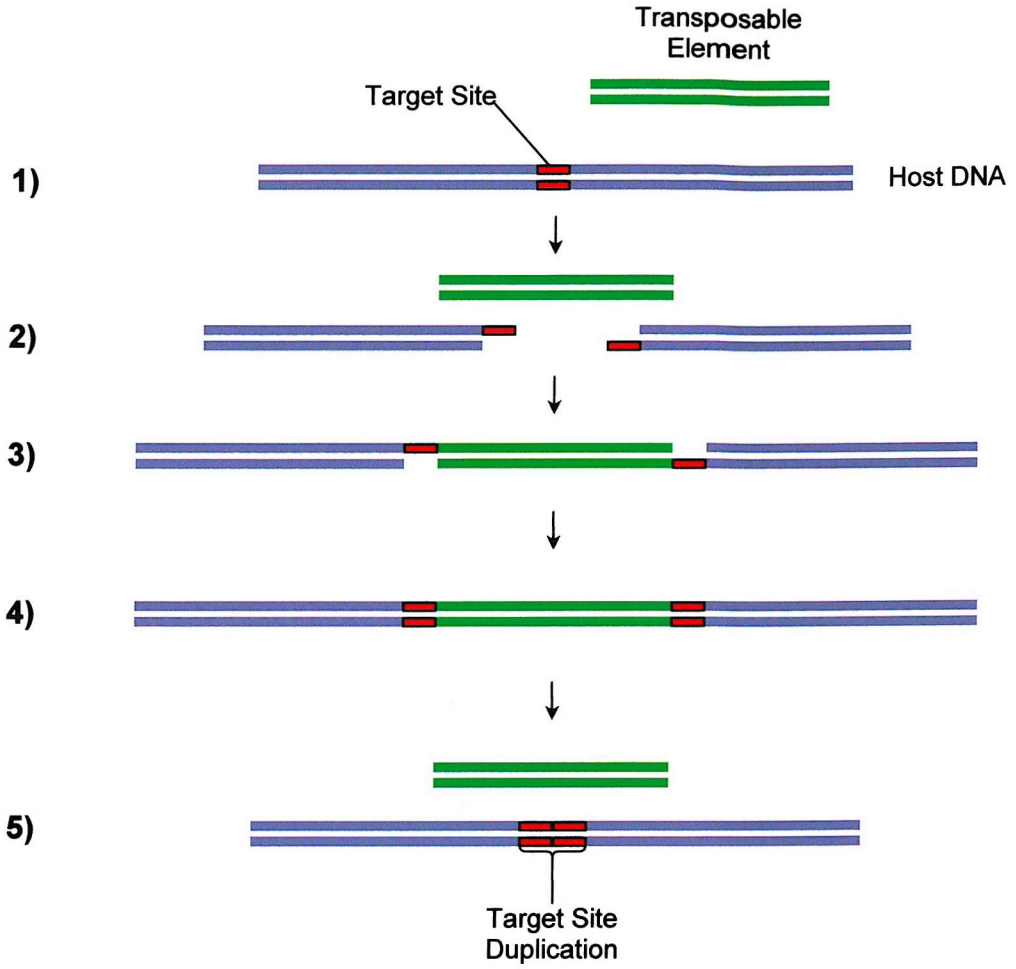


Figure 8 The mechanism of target site duplication by transposable elements. The structure in 1 represents a double stranded DNA molecule plus a transposable element. Staggered cuts are made at stage 2, followed by insertion of the transposable element at stage 3. This leaves two regions of complementary single stranded DNA, which are then filled in stage 4. This filling in of the single stranded DNA results in the duplication of the target site initially cleaved. Excision of the transposable element leaves the target site duplication.

Transposable elements that are mobilised through an RNA intermediate are termed retrotransposons (or retroposons)[27]. Examples include *LINEs*, the *gypsy* and *copia* elements of *Drosophilia*, and yeast *Ty* elements. They are related to the retroviruses in their life cycle, but lack the ability to leave the cell. From its DNA form in the genome the retrotransposon is transcribed to RNA in the usual way, this then inserts a DNA sequence into the genome through the use of the enzyme reverse transcriptase, which utilises RNA as a template to produce DNA. The gene for reverse transcriptase is present as part of the retrotransposon, as are the genes for other protein factors involved in the transposition cycle. As with the

DNA transposable elements, insertion into the genome is coupled with a duplication of the target site, and there is variation in the length of this duplication and in the nature of the target site.

Figure 9 Imprecise insertion and excision of a transposable elements. The target site in the host sequence is denoted in red. A transposable element (blue) inserts at the target site, causing a target site duplication (bold black). The transposable element may excise itself imprecisely in one of two ways, either leaving bases behind in the host sequence (an insertion), or by excising additional bases (a deletion).

Transposable elements are typically several thousand base pairs in length, and so the transposition event itself represents a fairly major change in the structure of the genome. Further to this, two transposable elements of exactly the same type may act as homologous sites for recombination, which can result in extreme genome restructuring, as described below. The transposition of an element into the coding region of a gene will destroy its function, and this has been observed to cause haemophilia A in individuals with no family history of the condition, by insertion of a *LINE* into the gene for factor VIII.

Transposable elements contain their own promoter sequences, to activate the genes involved in the transposition mechanism – on insertion into DNA these promoters can also activate native genes, enhancing their activity. It is also possible that insertion of an element can affect gene transcription by altering the three dimensional structure of DNA. This may be related to the effect of distances between gene regulator sequences and the core promoter, as described in chapter 1.

To give a general idea of the rates of transposition of transposable elements, the *Drosophila melanogaster* retrotransposons *Doc, roo* and *copia* elements have been estimated as having transposition rates of 4.2×10^{-5}, 3.1×10^{-4} and 1.3×10^{-3} transpositions per element per generation[61, 62]. The rate of excision of the *roo* element has been estimated at 9.0×10^{-6} per element per generation.[63]

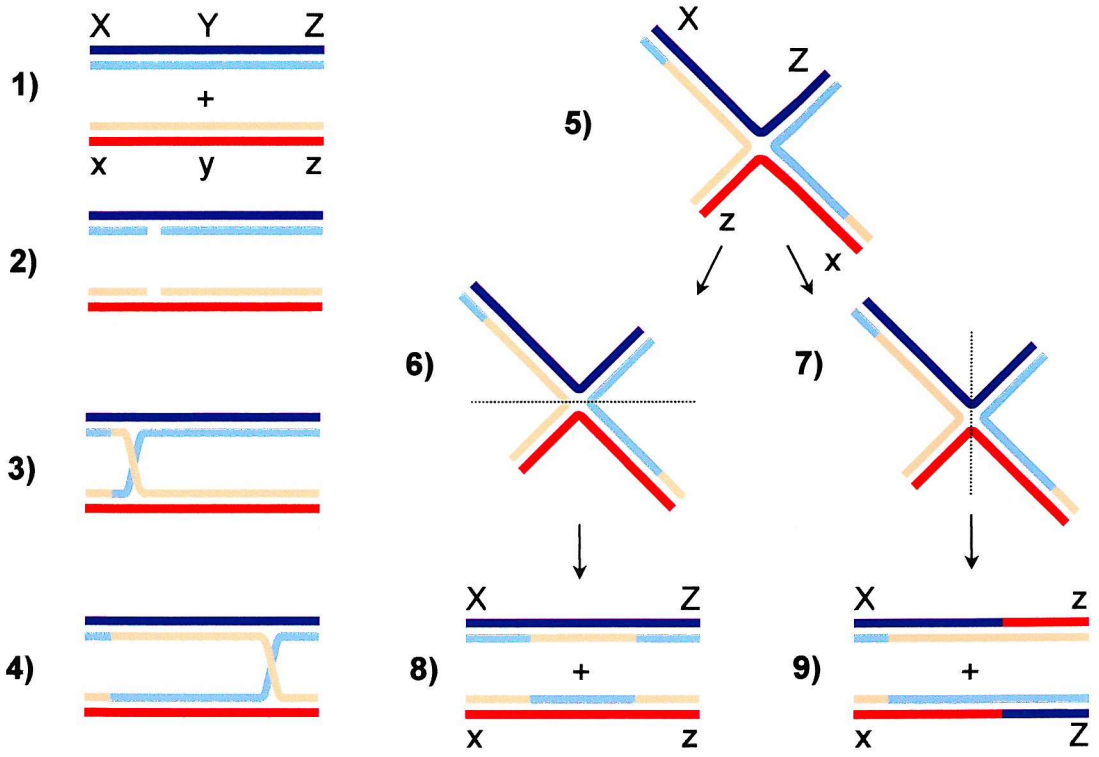


Figure 10 The mechanism of recombination (adapted from reference 7). 1 represents two DNA duplexes where X, Y and Z are genes, and x, y and z are alleles. At stage 2, nicks in the DNA strands occur, and homologous strands cross over resulting in structure 4, which can be re-drawn as structure 5, the Holliday structure. Cleavage of structure 5 can proceed in two different positions (6 and 7) giving rise to different products (8 and 9). Cleavage of the Holliday structure vertically gives rise to DNA in which large stretches of DNA are exchanged (stage 9).

3.1.4 *Recombination*[7, 9]

Recombination is the mechanism by which the crossing-over of chromatids on sister chromosomes takes place during meiosis. This crossing-over generates new combinations of alleles on chromosomes, and so enhances genetic variability.

Recombination must always occur between homologous sequences, due to the fact that strands of DNA are exchanged during the process. The mechanism proceeds by initial nicking of the DNA strands followed by strand exchange. The Holliday structure is formed, and two possible products result. See Figure 10.

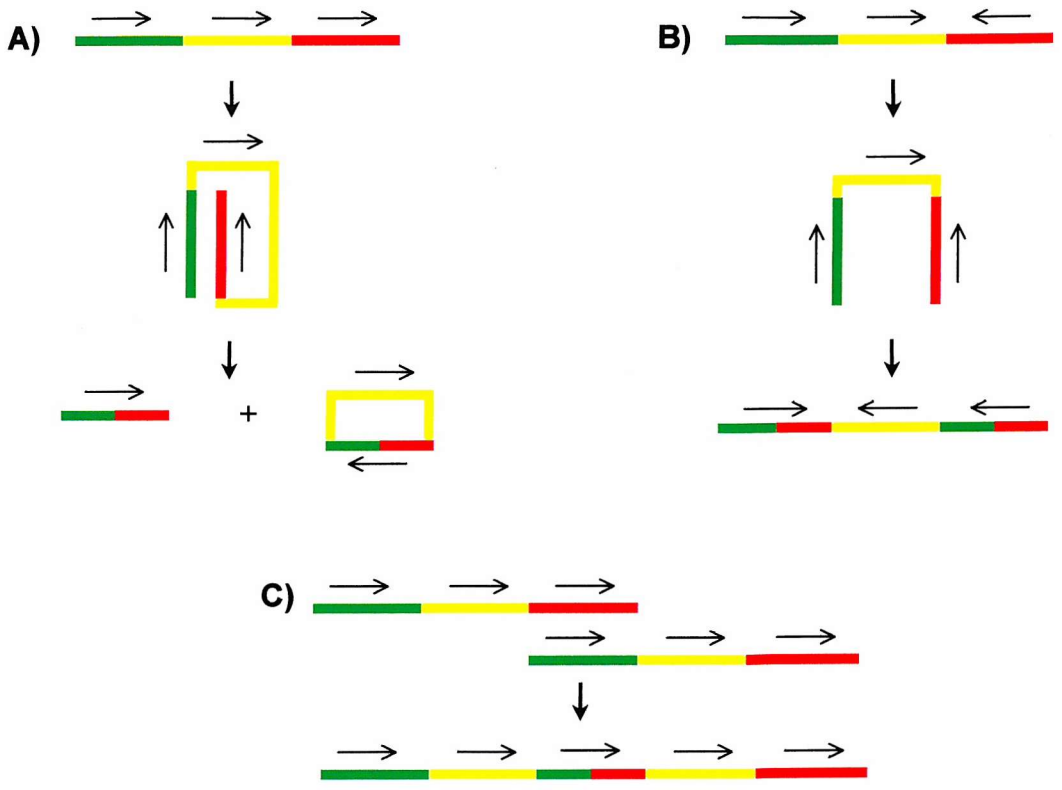


Figure 11 Rearrangements of the genome mediated by transposable elements. Homologous transposable elements (red and green) can act as sites for recombination, with one of three effects, as shown. In A) a gene (yellow) is deleted from the sequence; in B) the gene is inverted, and in C) a gene is duplicated by unequal crossing over, when sister chromosomes do not align exactly during recombination (from reference 9).

Two identical transposable elements present alternative homologous sequences to those on sister chromosomes that can lead to a number of possible outcomes [64]. Genes may be duplicated, deleted or inverted. Duplication of genes allows the original gene to continue to carry out an essential function, while mutation of the copy can result in a modified function. Inversion of a gene will almost certainly result in a change in its rate of expression. Figure 11 describes these changes. These modifications show further the effect that transposable elements can have on genome structure.

3.1.5 *Summary*

The genome reshaping processes described above are summarised in Table 4. It was thought that at least some of these processes might account for the formation of linguistic features in non-coding DNA, and this hypothesis could be tested by devising a model to describe such events. Before giving details of a simulation incorporating genome reshaping processes, three models giving alternative explanations for the emergence of language-like features are described.

Mutational process	***Rate***	***Number of bases affected***
Point mutation	10^{-9}-10^{-10} per nuceotide, per generation	1
Slippage	10^{-6}-10^{-8} per nucleotide, per generation	Depends on period of repetitive sequence, typically 2-15
Transposable element insertion/excision	10^{-3}-10^{-5} per transposable element, per generation	10^{3}-10^{4} due to transposable element itself, 5-9 due to duplication of bases on insertion
Recombination	10^{-8} per nucleotide, per generation	Depends on chromosome size; 10^{6}-10^{7} in humans

Table 4 Summary of genome reshaping processes

3.2 MODELLING THE FORMATION OF LINGUISTIC FEATURES

Since the observation of linguistic features in non-coding DNA, attempts have been made to model the hierarchical sequence correlations that underlie them. Buldyrev et al have produced two such models, based on a procedure known as the Lévy-walk[33]. In a basic Lévy-walk the method is to take l_1 steps in a given direction (up or down), followed by l_2 steps in a new, randomly determined direction, and so on. The lengths l_j of each string are drawn from an arbitrary probability distribution. The output of a basic Lévy-walk is shown in Figure 12.

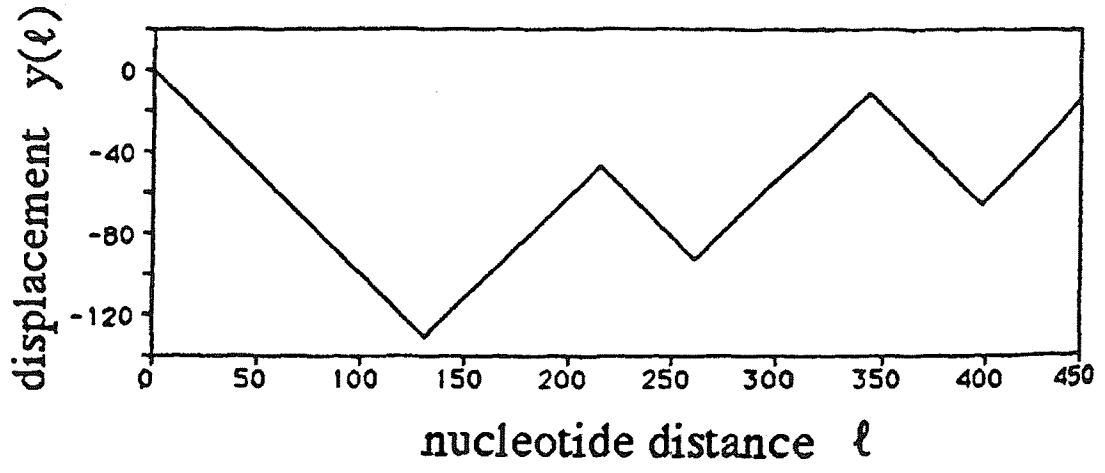


Figure 12 Output of the basic Lévy-walk (from reference 33). l_j steps (taken from a probability distribution) are taken in either the up or down direction, determined at random. The direction is then chosen again, at random, and another l_j steps are taken, and so on.

3.2.1 Generalised Lévy Walk Model[33]

The model is a basic Lévy-walk as described above, with several modifications. Rather than taking l_j steps in the same direction, each step is taken in a random direction. The direction of each step is chosen with a fixed bias probability that includes a parameter to affect the overall bias for each string l_j, which is randomly chosen as + or –. The overall bias means that for each string l_j the walk will tend to go either up or down as in the basic Lévy-walk. If for example,

the bias is + there will be more steps in the up direction than down, and vice versa.

The data generated appear similar to those shown in Figure 12, but the line is more 'jagged'. A sample output is shown in Figure 13. This is because the direction of each individual step in each string l_j is chosen at random. Even if the overall bias is +, for example, there will still be some steps down, and vice versa. The model proceeds by repeating the process and stitching together a number of strings l_j, to a given length.

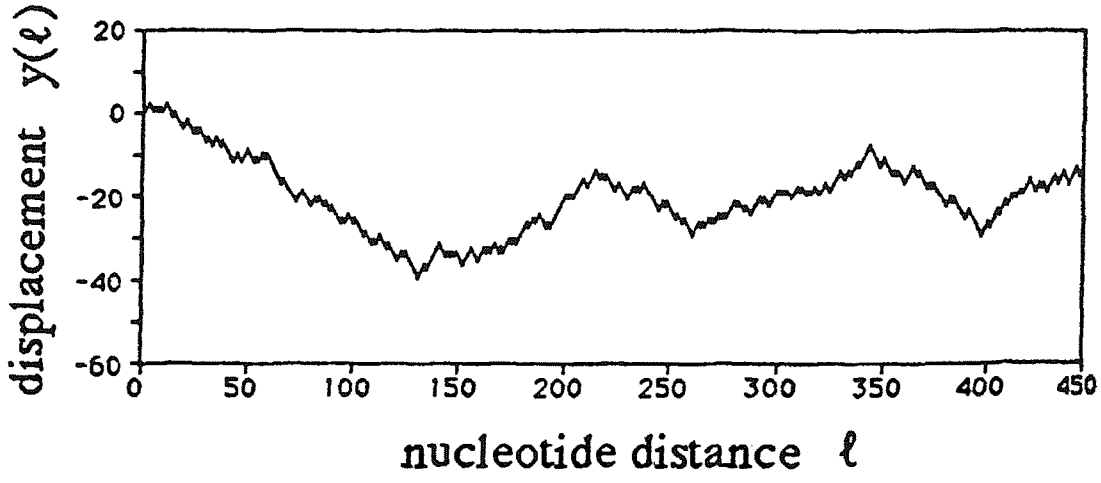


Figure 13 Output of the generalised Lévy-walk (from reference 33)

The map that is produced by the generalised Lévy-walk corresponds to that seen after applying the RY mapping rule to real DNA sequences. The map can be used to generate fluctuation analysis data.

By choosing the overall probability bias corresponding to the observed bias of pyrimidines vs. purines in real DNA, and by adjusting the probability distribution used to determine the lengths of each string, the Lévy-walk model can generate maps which resemble those of both coding and non-coding DNA sequences. When the probability distribution is modified such that a 'non-

coding' map is generated, fluctuation analysis of the map gives a value of α that is comparable to that found in non-coding regions of DNA[33].

While this model does indeed produce long-range correlations, it achieves this through a process that does not draw on the known genome reshaping processes. The model is interesting from a statistical point of view, but it is not a model of real-life events, and as a result cannot be used directly to explain how long-range correlations might have come about in non-coding DNA.

3.2.2 *Insertion-Deletion Model*[34]

This model consists of the following steps, starting with a 'coding' sequence generated by a biased random walk:

1) A length of DNA is cut from a sequence that shows no long-range correlations. The length of this string is determined from a power law distribution.
2) This sub-sequence may be inverted (i.e. all the pyrimidines are converted to pyridines and vice versa) with a probability of ½. With a low arbitrary probability the sub-sequence may be replaced by a random sub-sequence, with the same length.
3) The sub-sequence is inserted at a random position back into the original DNA sequence, from which it was initially cut.

This resulting sequence is then used as the starting point for another iteration of the process. The procedure is repeated for 100–2,000 cycles.

Analysis of the final sequence using the fluctuation approach gives values of α which correspond to those achieved with non-coding sequences[34].

This model does simulate some of the features found in real systems, such as the insertion of retroviral sequences and transposable elements, although such

events are not modelled in detail and in particular, target site duplications are not modelled.

3.2.3 *Markov Processes*[43, 65]

A Markov process is a procedure used to generate a sequence of values, in which the decision on the next value to be added to the sequence is affected only by the previous value in the sequence. As a result, Markov processes do not explicitly consider any long-range structure.

It has been shown that a Markov process involving biased random transitions between states (values of the sequence) can generate long-range correlations, and shows Zipf law behaviour[43]. The long-range correlations generated by the sequence extend over long, but finite intervals, which is not entirely consistent with the long-range correlations in non-coding DNA sequences which extend over all length scales.

4 The Transposable Element Model

The models described in the previous chapter demonstrate that language-like features in DNA sequences can be generated by processes that do not involve the simulation of physically realistic biological events. With this in mind, a model was proposed that might account for the generation of language-like features in DNA[66]. A key factor of this model was that unlike previous models it was based on *in vivo* genetic mutations - namely, the insertion and excision of transposable elements.

The model consists of the iterated duplication of specific target sites, and simulates the site-specific insertion and precise excision of a transposable element of the type mobilised through DNA. The model is a simulation of events that occur in the genome and therefore does not explicitly involve a biological language.

The model was implemented by iteration of the following general steps:

1) A sequence of DNA was scanned for target sites, which were arbitrarily chosen prior to the experiment.
2) At one of these sites, chosen with a given probability, a target site duplication event was simulated. A given number of bases were duplicated, and placed immediately after the target sequence.
3) The sequence generated was then used again in 1).

The action of the model is depicted graphically in Figure 14.

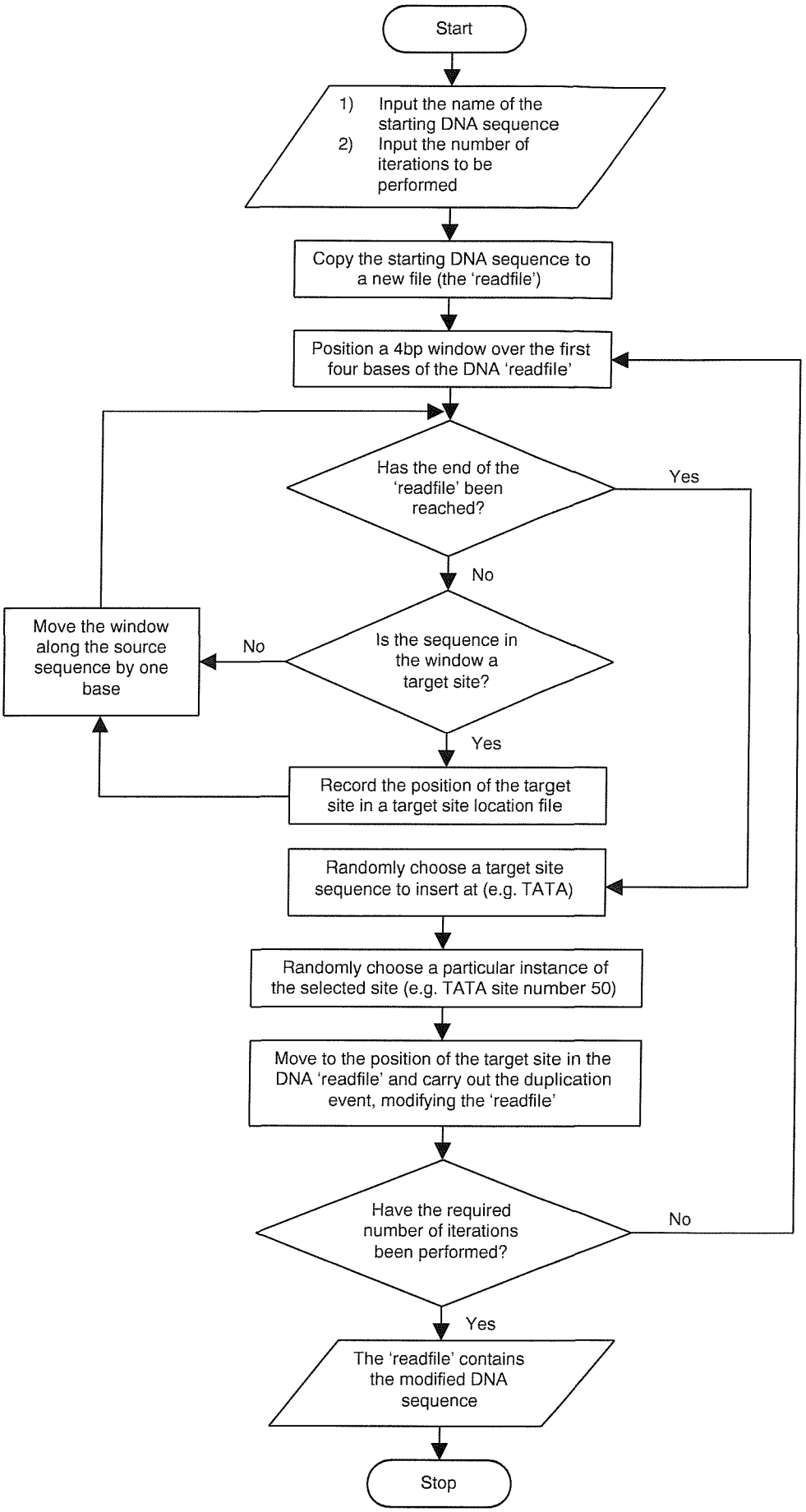


Figure 14 Flowchart summarising the action of the transposable element model

The sequence generated after a number of iterations was analysed using three statistical measures in order to assess whether the procedure had an effect on the language-like character of the sequence. The measures used in the analysis were the Zipf test, the Shannon entropy and the fluctuation exponent, all of which are described in detail in chapter 2.

4.1 SEQUENCE GENERATION

4.1.1 Starting Sequences Used

Two sequences were used in the course of this study. The first was the complete proviral genome of the human T-cell leukaemia virus type II (HTLV II, GenBank accession number M10060). This sequence is 8,952 bp in length, with a coding DNA : non-coding DNA ratio of 0.762[67].

Secondly, a random sequence was generated, such that the probability of any given base (A, G, T or C) occurring at a given position in the sequence was 0.25. This sequence was made to the same length as the HTLV II sequence.

4.1.2 Target Site Selection

Throughout the work on the initial model the three target sites used were the sequences TATA, TCAG and GGAC. These were chosen arbitrarily, but are typical of the insertion sites targeted by transposable elements[59,60,61].

The mechanism of selection was carried as follows:

1) Prior to the start of the simulation each of the three target sites was assigned a probability of selection (eg. TATA:TCAG:GGAC = 0.6 : 0.2 : 0.2).

2) During the simulation, the first stage of site selection was the random selection of a target site type (TATA, TCAG or GGAC), based on the selection probabilities assigned in 1).
3) One of the specific sites of the type chosen in 2) (eg. TATA) was then selected at random from those in the sequence, each with an equal probability of selection. For example, if target site type TATA was chosen in 2) and the host sequence contained 10 TATA sequences, one of those 10 specific sites would be selected at random, each with the same probability of selection.

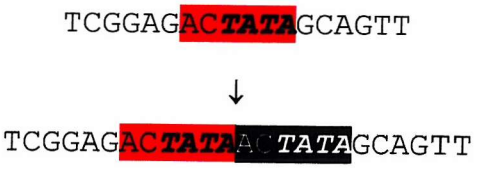


Figure 15 A 6-bp duplication on insertion. Insertion occurs at the TATA site marked in bold in the top line, and results in duplication of the bases highlighted in red. The duplicated sequence (highlighted in black) is inserted immediately following the target site. Note the new sequence now contains two target sites (shown in bold).

4.1.3 Target Site Duplication

On selection of a target site a number of bases were duplicated and inserted after the target site. Figure 15 shows the procedure for a 6 bp duplication. The lengths of duplication were chosen with the observed biological duplications in mind (i.e. in the range 5–9) [49, 50, 51]

4.1.4 Iteration

The starting sequences were typically run for 1,000, 5,000 or 10,000 iterations of the simulation before analysis. This was later extended to 100,000 iterations. Sets of results were generated by applying the simulation to a starting sequence for e.g. 1,000 cycles, recording the sequence, and then continuing to e.g. 5,000 cycles, and so on, so that each sequence generated was a modification of the previous. This is in contrast to the alternative approach (which was not used) in

which the simulation would have been re-started to obtain each individual sequence.

4.2 SEQUENCE ANALYSIS

Analysis of the sequences generated were carried out using the statistical tests defined in chapter 2, namely the Zipf test, the Shannon entropy calculation, and the fluctuation analysis.

For the Zipf test, the results were analysed using "word" lengths of n = 3–6. The Shannon entropy calculation was carried out using a word length 4.

The window lengths, l, used in carrying out the fluctuation analysis, to generate the plot of $\log_{10}F(l)$ against $\log_{10}l$, were l = 2, 5, 10, 20, 50, 100, 200, 500 and 1,000.

4.3 RESULTS

The aim of the initial simulations was to observe the effect of iterations of the simulation on the linguistic character of the sequences generated. Following these observations, various parameters of the model were altered to assess their effect on the linguistic features of the resulting sequences.

4.4 INITIAL OBSERVATIONS

The HTLV II genome was used as the starting sequence for initial testing of the proposed model. Figure 16 shows the Zipf plot of the HTLV II sequence, together with the results obtained by applying the simulation for varying numbers of iterations. It can be seen qualitatively that there is a progressive increase in the slope as the number of iterations increases. An actual measurement of the slope is difficult to obtain with any accuracy, due to the curvature of the slope at high ranks characteristic of many Zipf plots (a feature

observed in the Zipf analysis of both languages and DNA)[39]. An estimate was obtained by fitting a straight line to the first 30 points (ranks 1-30 – visual inspection suggested the line was linear in this region of the plot) using the linear regression technique. This approach revealed that from the initial sequence to the sequence obtained after 10,000 iterations the slope (ζ) increases from $\zeta \approx 0.34$ to $\zeta \approx 0.51$. Although measurement of the gradient is not straightforward due to the shape of the line, it is clear from visual inspection of the plot that the overall slope is increased with iterations – the line starts at higher probability and ends at lower probabilities as the iterations are increased.

The values obtained for the gradients are sensitive to the exact points chosen for the straight-line fit. The density of data points increases along the x-axis so an apparently small change in the range used (when assessed visually) can have a dramatic effect on the calculated value of ζ. For example, increasing the range from ranks 1-30 to ranks 1-70 (a 20% increase when viewed on a log scale) increases the gradient from 0.51 to 0.72. For this reason, the range used was set at ranks 1-30 for all measurements of ζ in this project – no attempt was made to alter the range to suit each individual plot. It should be noted that while the range chosen affects the exact value obtained for the gradient, the overall trend (higher iterations give a higher gradient) is unaffected by the choice of range. Although there are clearly disadvantages to fitting a straight line to the Zipf data, this approach does allow comparison with results obtained from the Zipf analysis of real DNA sequences.

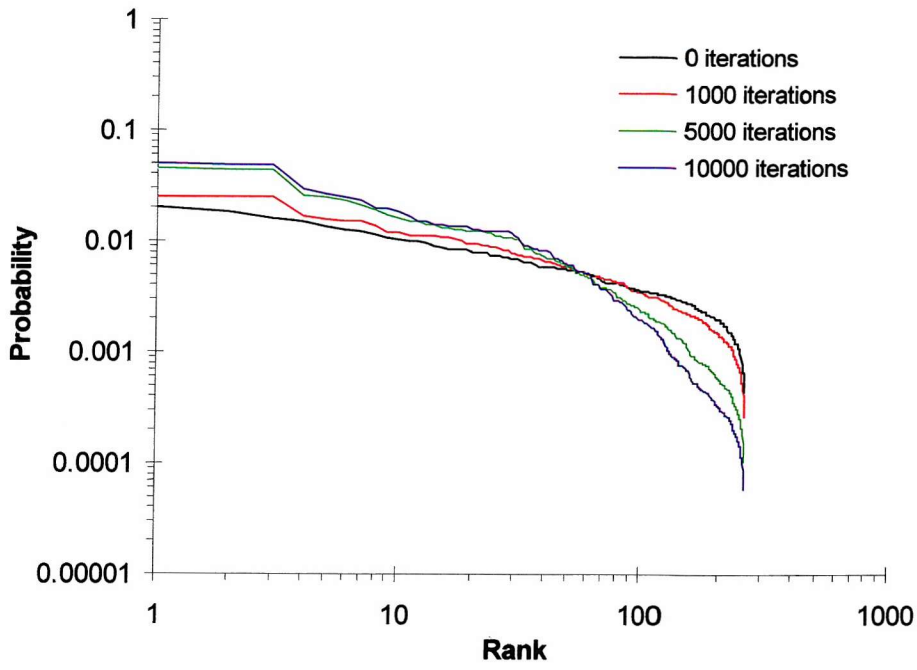


Figure 16 Zipf plots for initial run; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; analysed with word length 4

Fluctuation analysis of the same sequences is shown in Figure 17. The slope corresponding to the HTLV II sequence gives a value $\alpha = 0.54$, whereas the slope after 10,000 iterations is $\alpha = 0.86$.

The reproducibility of these results was assessed by repeating the simulation three times. In each case the random number generator used in the program was seeded with a different value (taken from the PC system clock). Qualitatively the shapes of the Zipf plots produced were very similar, and the approximate slopes corresponded well, with reproducibility to within ±0.04. The results achieved from fluctuation analysis were nearly identical in all three cases – at 1,000 iterations all three values of α were within ±0.05, and at 10,000 iterations all values were within ±0.01 of each other.

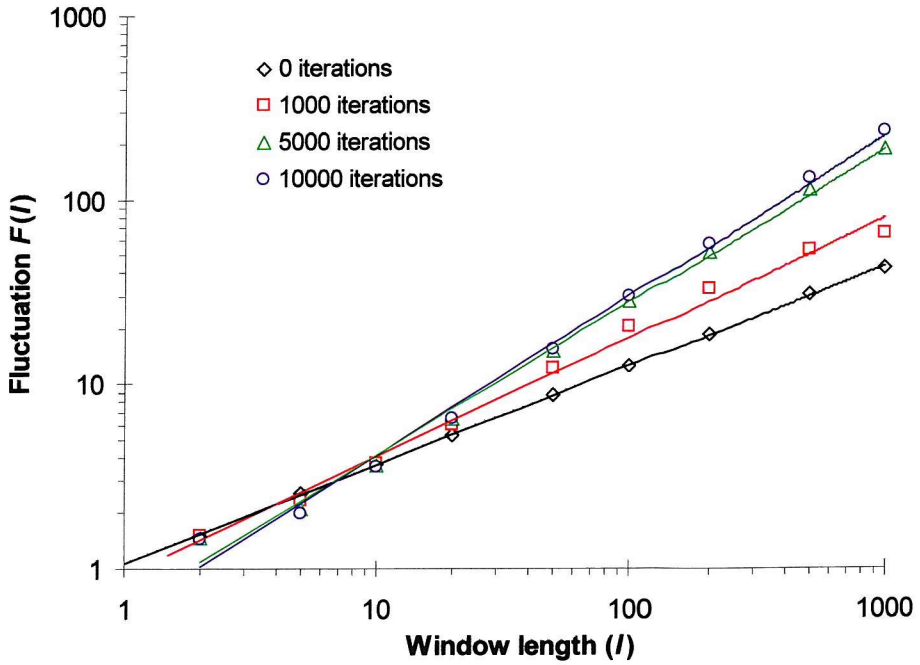


Figure 17 Fluctuation analysis of initial run; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6. The values of the fluctuation exponent are 0.65, 0.83, and 0.87 for 1000, 5000 and 10000 iterations respectively

These results show that, starting with a sequence which is highly coding and lacking linguistic features, the simulation can generate linguistic features characteristic of non-coding regions. The simulation proceeds by duplication of certain sequences. When the duplication length is at least as long as the target site (as it is in this case) then duplication of a target sequence will generate a new target sequence, increasing the probability of further selection of this site, and hence further duplication. The effect is to greatly increase the proportion of certain nucleotide motifs; a bias which will increase the slope generated by the Zipf plot. Long-range correlations emerge within the sequence as a result of the creation of structures formed by repeated duplications contained in blocks.

The Zipf plots do not show ideal Zipf behaviour, but this is partly a feature of Zipf plots carried out using "words" of an arbitrary length in which all "words" are the same length (such behaviour is observed in the Zipf analysis of both languages and DNA)[39], and partly as a result of the relatively basic nature of the model. It was described in chapter 3 how transposable elements show

considerable variability in their target site selection, and in the length of target site duplication upon insertion. The use of only three target sequences in the simulation, each with the same target site duplication length limits the number of different sequences that can be duplicated. This may lead to a bias which is too uneven to produce a smooth slope on a Zipf plot — a small number of "words" are present at a high frequency, while a large number of "words" are present at relatively low frequency.

A further side-effect of using three target sites and perfect copying events is the blocky nature observed when examining the sequences produced by the simulation. A sample of the sequence is given in Figure 18.

The results show that the model captures the physics of the emergence of linguistic features in sequences of DNA. The fact that the model is based on known biological processes means that this may be the mechanism which is responsible for the generation of linguistic features that have been reported for non-coding areas of DNA. It shows that linguistic features can arise without the presence of underlying biological information. This does not mean that organisms do not make use of the higher order structure of non-coding DNA, but these results suggest that if they do then it is an opportunistic use of such structure.

```
TGACAATGGCGACTAGCCTCCCAAGCCAGCCACCCAGGGCGAGTCATCGACCCAAAAGGT
CAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGACC
GTCTCACACAAACAATCCCAAGTAAAGGCTCTGACGTCTCCCCCTTTTTTTAGGAACTGA
AACCACGGCCCTGACGTCCCTCCCCCCTAGGAACAGGAACAGCTCTCCAGAAAAAAATAG
ACCTCACCCTTACCCACTTCCCCTAGCGCTGAAAAACAAGGCTCTGACGATTACCCCCTG
CCCATAAAATTTGCCTAGTCAAAATAAAAGATGCCGAGTCTATATCTATATCTATATCTA
TATCTATATCTATATCTATATCTATATCTATAAAAGCGCAAGGACAAGGACAAGGACAAG
GACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAG
GACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAG
GACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAG
GACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAG
GACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAG
GACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAG
GACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAG
GACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAGT
TCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGT
TCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGT
TCAGGTTCAGGTTCAGGAGGTGGCTCGCTCCCTCACCGACCCTCTGGTCACGGAGACTCA
CCTTGGGGATCCATCCTCTCCAAGCGGCCTCGGTTGAGACGCCTTCCGTGGGACTGGGAC
TGGGACTGGGACTGGGACTGGGACTGGGACTGGGACTGGGACTGGGACTGGGACTGGGAC
```

Figure 18 The blocky nature of sequences produced by the simulation. Colour coding helps to highlight the sequences produced by target site duplication. The first 1200 bases of a sequence generated after 5000 insertions into HTLV II are shown; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6

4.5 THE EFFECT OF ITERATION

Qualitative analysis of Figure 16 and Figure 17 shows that both α and ζ increase with iteration, but that the relationship is not linear. The relationship between iterations and linguistic character was investigated by measuring the variation of the Shannon entropy and the fluctuation exponent, α, as a function of iterations. Typical results are shown in Figure 19 and Figure 20.

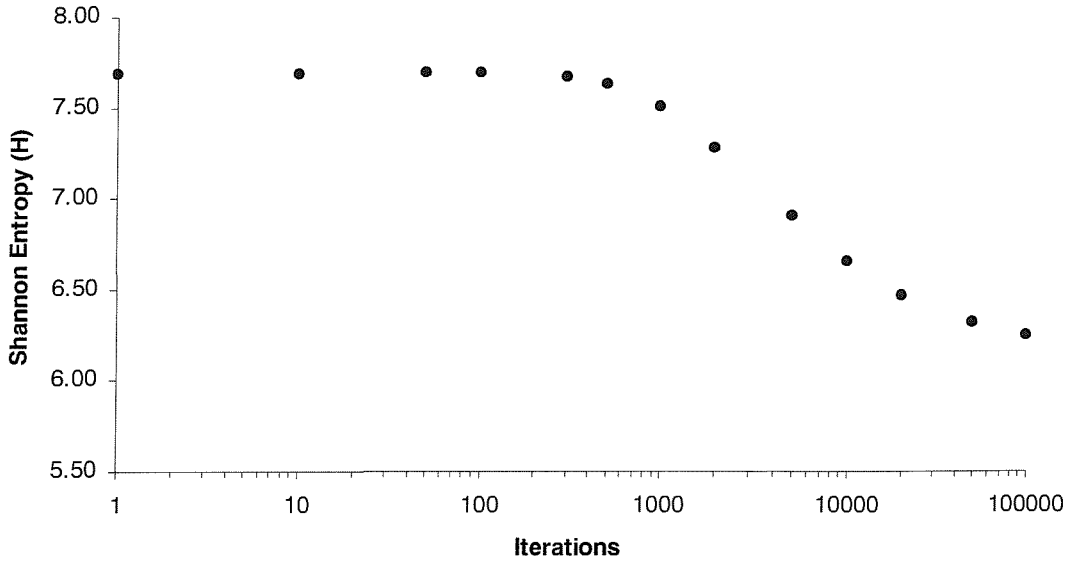


Figure 19 Variation of Shannon Entropy with iterations for initial run; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; analysed with word length 4

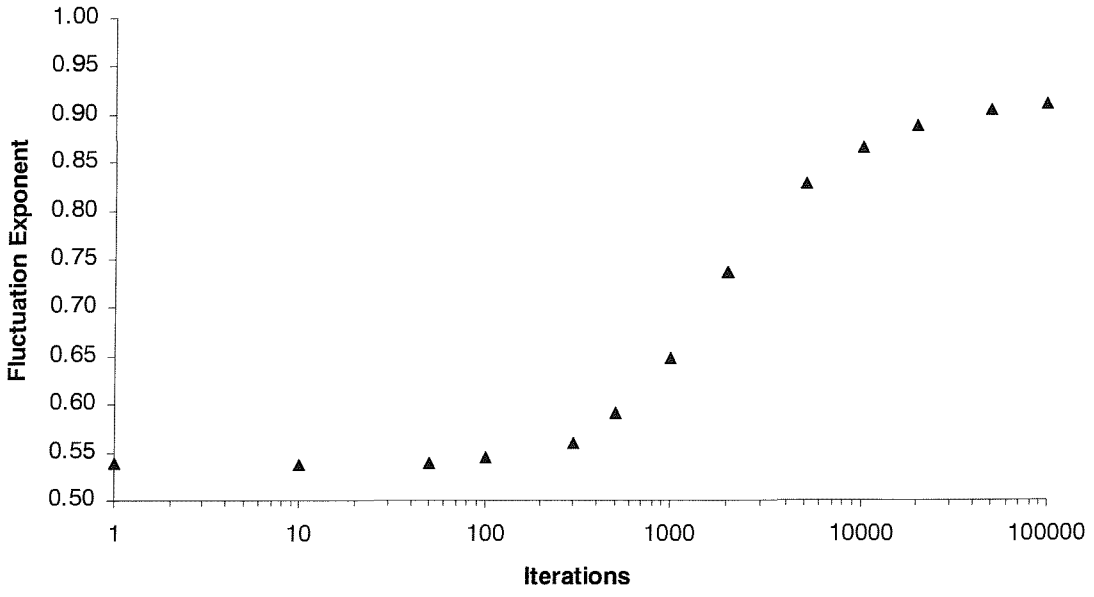


Figure 20 Variation of fluctuation exponent with iterations for initial run; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6

The results clearly show that as the number of iterations increases the linguistic character initially stays fairly constant, then increases quite sharply, before finally levelling off.

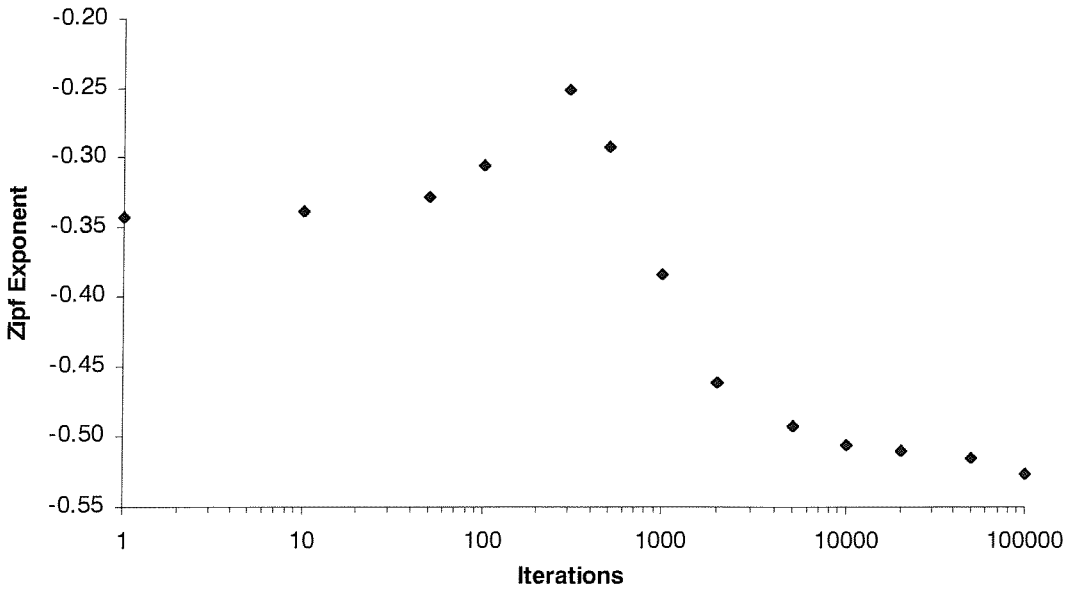


Figure 21 Variation of Zipf exponent with iterations for initial run; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; analysed with word length 4

Figure 21 shows the effect of iteration on the Zipf exponent (ζ). The Zipf exponent was measured using the data points for rank 1 to rank 30. The plot shows a general increase in the magnitude of the gradient (the gradient becomes more negative overall) but with a substantial peak in the gradient observed around 300 iterations. Aside from the peak observed at around 300 iterations the overall shape of the Zipf exponent vs iterations curve is similar to that seen with the Shannon entropy and fluctuation results. There is a sharp increase in the magnitude of the gradient, followed by a flattening off at higher iterations. The Zipf plot obtained after 100,000 iterations is shown in Figure 22. There is little difference between this plot and that obtained at 10,000 iterations at the lower ranks, although there is a more prominent drop-off in the line at just over rank 100.

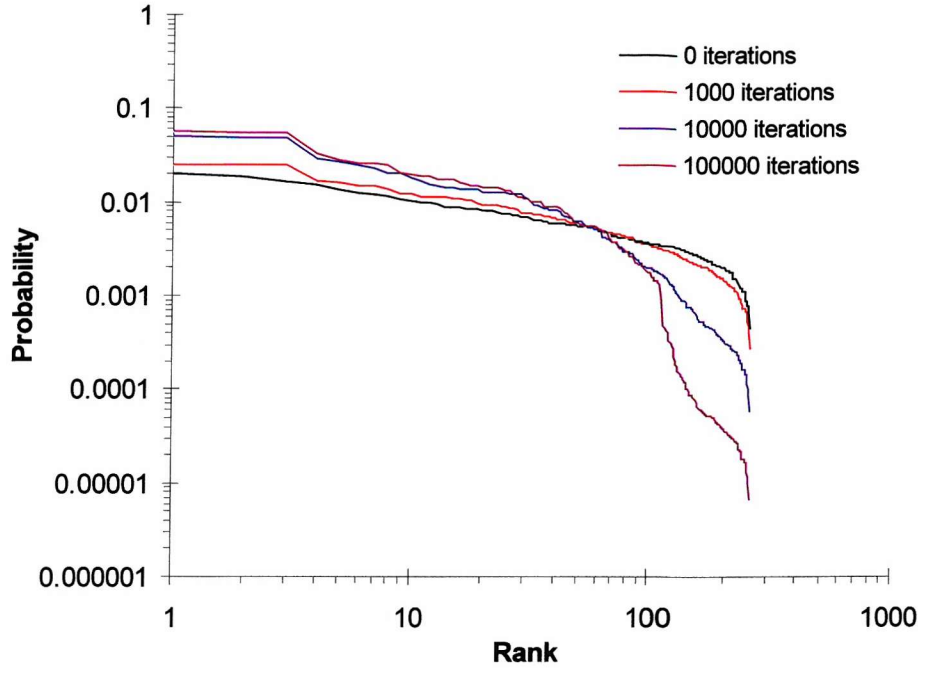


Figure 22 Zipf plots for initial run showing plot for 100000 iterations; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; analysed with word length 4

In order to investigate the cause of the peak in the gradient the Zipf plots obtained after 50, 300 and 1,000 iterations were examined – these are given in Figure 23. It can be seen that after 300 iterations the start of the curve (at low rank numbers) is lower than the starting sequence. The decrease in gradient appears to arise because the start of the plot becomes flattened – the most common words occur at roughly the same frequency as each other. Table 5 shows the most common words present after 100 and 300 iterations. It can be seen that after 300 iterations there is a smaller range of probabilities (0.0166 for rank 1 to 0.0099 for rank 12) than for 100 iterations (0.0187 for rank 1 to 0.0093 for rank 12), so the data after 300 iterations would give a flatter start to the Zipf plot. Looking at the words themselves shows that after 300 iterations a number of sequences are in the top 12 that were not there after 100 iterations, for example, the sequences TATA, TCAG and GGAC. The frequency of these sequences has been increased by target site duplication events. Because the overall number of words has increased due to duplication events, the probability of words that have not been duplicated decreases (e.g. CCCC is rank 1 in both charts – it

occurs with the same frequency, 179, after 100 iterations and 300 iterations, but the probability drops after 300 iterations because the total number of words is higher - the sequence is longer after 300 iterations). The new words that have appeared in the top 12 ranks all have roughly the same probability as one another and this, coupled with the lowering in probability of the non-duplicated words leads to a smaller overall range of probabilities in the top ranked words, resulting in a flatter Zipf plot. If the simulation is continued, then with further iterations the new words will rise to the top ranking positions (when these words first reach the top ranks may actually represent the situation where the Zipf exponent is at its minimum) from which continued selection will boost their probabilities and lead to a bias that increases the Zipf exponent.

A key reason for the decrease in gradient is the fact that the duplicated words all have roughly the same frequency and therefore appear in the top ranked words after approximately the same number of iterations. Table 6 gives details of the proportions of target sites present throughout the simulation. In the starting HTLV II sequence the sites are biased, there being approximately twice as many GGAC as TATA sites, for example. On running the simulation at 1:1:1 site selection bias, the proportion of each site gradually balances out to 0.33. This is expected due to the mechanism of site selection built into the simulation. The first step is to select a site type (TATA, TCAG or GGAC) depending on the selection bias (each equally probable in this case) and then to choose an individual occurrence of that site type. The number of target sites of a particular type in the starting sequence does not affect the probability of that type being selected for an insertion/excision event.

Rank	100 iterations			300 iterations		
	Word	requency	Probability	Word	Frequency	Probability
1	CCCC	179	0.0187	CCCC	179	0.0166
2	CTCC	164	0.0171	CCTC	168	0.0156
3	CCTC	154	0.0161	CTCC	164	0.0152
4	CCCT	134	0.0140	TCAG	136	0.0126
5	TCCC	121	0.0126	CCCT	134	0.0124
6	CCCA	112	0.0117	TATA	131	0.0121
7	TCCT	108	0.0113	GGAC	131	0.0121
8	ACCC	102	0.0106	CAGG	122	0.0113
9	CCTT	97	0.0101	TCCC	121	0.0112
10	CCAA	92	0.0096	CCCA	112	0.0104
11	TTCC	90	0.0094	TCCT	108	0.0100
12	CACC	89	0.0093	CAGC	107	0.0099

Table 5 The most common words present after 100 and 300 iterations; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6

It is possible that running the simulation with a bias towards certain target sites might reduce or eliminate altogether the peak in the Zipf exponent seen at 300 iterations. The bias would mean that words that were duplicated by the simulation might appear in the top ranks after different numbers of iterations rather than all together – this would help to maintain a bias in word frequencies in the most common words and therefore would lessen the flattening of the Zipf plot. This was investigated by running a simulation with a target site bias such that target sites were selected with a ratio TATA:TCAG:GGAC of 1:3:6. The effect on the Zipf exponent as a function of iterations is shown in Figure 24. This plot clearly shows that there is still a peak in the Zipf exponent at around 300 iterations but the magnitude of the peak is lower than that observed for the simulation run without target site bias.

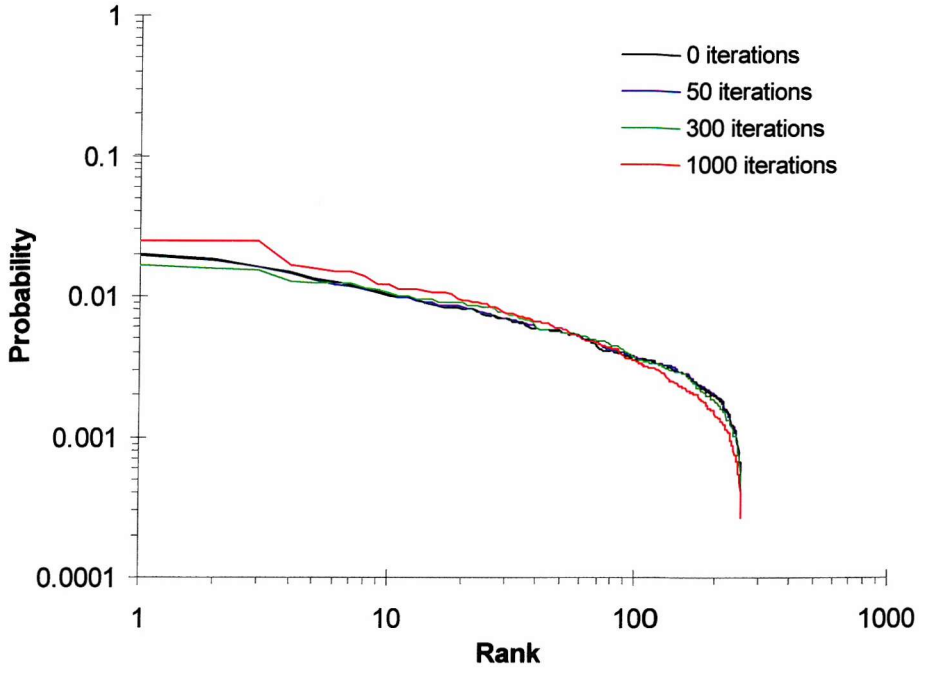


Figure 23 Zipf plots for initial run showing plots generating peak in Zipf exponent; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; analysed with word length 4

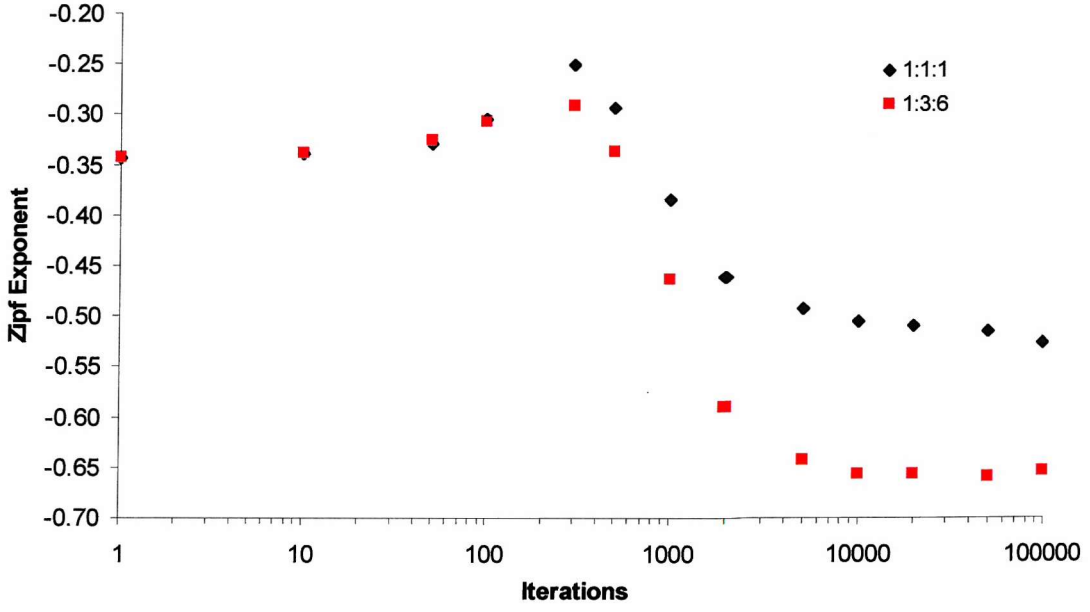


Figure 24 Variation of Zipf exponent with iterations showing effect of target site bias on peak in Zipf exponent; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities given in figure in form TATA:TCAG:GGAC; copy length 6; analysed with word length 4

Iterations	Number of TATA Sites	Proportion of TATA Sites	Number of TCAG Sites	Proportion of TCAG Sites	Number of GGAC Sites	Proportion of GGAC Sites	Total Number of Sites
0	22	0.22	34	0.35	42	0.43	98
10	24	0.22	36	0.33	48	0.44	108
100	59	0.30	66	0.33	73	0.37	198
1000	365	0.33	369	0.34	364	0.33	1098
10000	3304	0.33	3353	0.33	3441	0.34	10098
100000	33468	0.33	33410	0.33	33220	0.33	100098

Table 6 The relative proportions of target sites for the initial run; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6

4.6 *EFFECT OF WORD LENGTH*

The use of different word lengths in the Zipf analysis can alter the shape of the Zipf plot produced. This effect was investigated by using word lengths 3, 4, 5 and 6 in the Zipf analysis of the sequences generated from a single run. These results are shown in Figure 25. Qualitatively the higher word lengths appear to result in Zipf plots which are flatter at low ranks, while word length 3 gives a more curved line. Estimates of the gradient (obtained using the first 30 data points in each case) show values of $\zeta \approx 0.59$ for word length 3; $\zeta \approx 0.51$ for word length 4; $\zeta \approx 0.29$ for word length 5 and $\zeta \approx 0.11$ for word length 6.

The length of the sequence being analysed is important when considering which word length to use. For example, using word length 6 results in 4^6=4,096 possible words - a short sequence may not contain enough of these words to give a statistically reliable result. Given that the HTLV II starting sequence used has a length of 8,952bp it was not prudent to use the higher word lengths. Taking this into account, along with the appearance of the Zipf curves, word length 4 was chosen to be used for future Zipf plots.

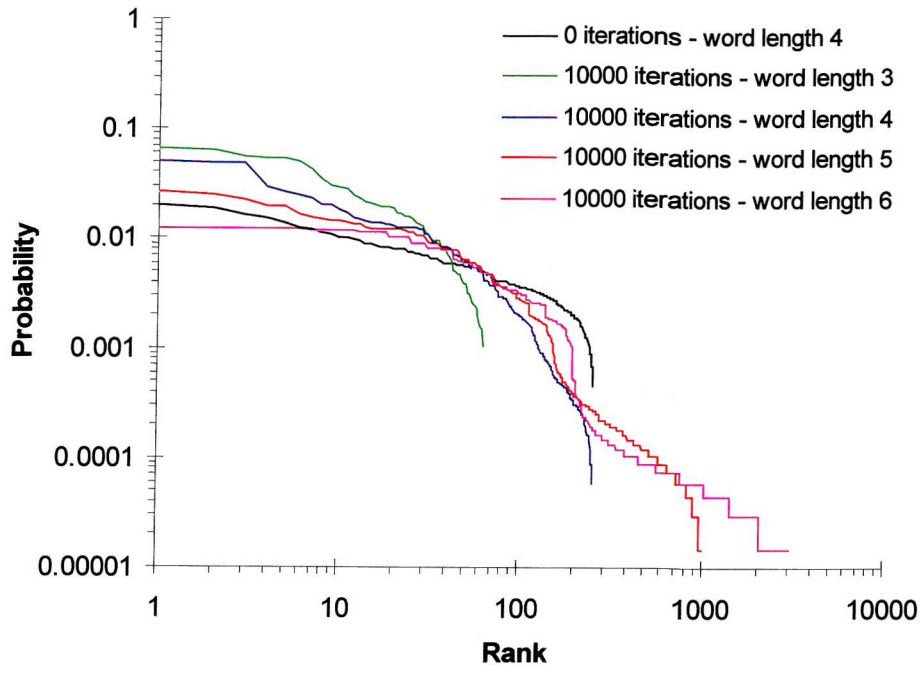


Figure 25 The effect of word length on Zipf plots; 10000 insertions into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; word lengths given in figure

4.7 EFFECT OF COPY LENGTH

Simulations were carried out using copy lengths 6 bp and 8 bp. At copy length 8 an interesting result is sometimes observed. Figure 26 shows Zipf plots of two separate runs of the simulation, with all parameters the same. It can be seen that there is a massive difference between the two results, due to the large plateau formed during run 2.

This plateau is formed because certain sequences are duplicated far in excess of others. This is due to the presence in the original sequence of overlapping or adjacent target sites, e.g. TCAGGAC or TCAGGGAC. If insertion occurs at the TCAG site then the GGAC site is also duplicated and two new sites are generated, rather than just one. This double target site duplication increases the probability of insertion at either of these two sites, which would lead to another two sites and so on, resulting in these sequences becoming unusually common in the final sequence. It seems that in order for this to result in the formation of a

plateau some threshold number of insertions must be passed, and that if by chance there are only a few insertions early on during the simulation at such a sequence then normal Zipf behaviour will be observed.

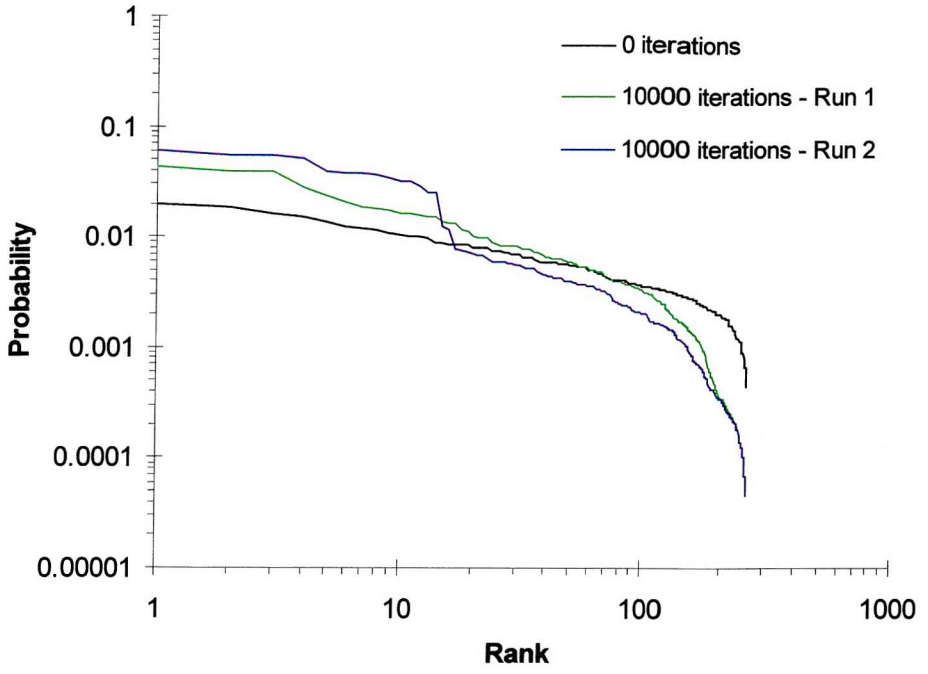


Figure 26 Zipf plots showing effect of sequence over-expression due to overlapping target sites; 10000 insertions into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 8; analysed with word length 4

Run Number	Number of TATA Sites	Proportion of TATA Sites	Number of TCAG Sites	Proportion of TCAG Sites	Number of GGAC Sites	Proportion of GGAC Sites	Total Number of Sites
1	3823	0.36	3439	0.32	3438	0.32	10700
2	5276	0.36	4742	0.33	4442	0.31	14460

Table 7 Amplification due to overlapping target sites; 10000 insertions into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 8

Table 7 shows the difference in numbers of target sites between run 1 and run 2 after 10,000 iterations. Following run 2 there are 35% more target sites present than after run 1. Note that in both cases there are more target sites than would

be expected if just one new target site were generated with each iteration (there are 98 target sites in the starting sequence, so 10,098 sites would be expected after 10,000 iterations). Table 6 shows an example of a simulation in which each iteration leads to only one new target site.

Amplification does not occur if the copy length is reduced to 6 bp, as insertion at GGAC only results in the formation of one new target site. As a result of the unreliable nature of simulations using long copy lengths, a 6 bp copy length was used for the majority of simulations.

4.8 EFFECT OF TARGET SITE BIAS

Simulations were carried out for a variety of target site probabilities, with all other parameters the same. The results showed no significant change in the fluctuation exponent α for a change in site bias but produced some variation in the quality of line obtained with Zipf plots. Figure 27 shows an example of a Zipf plot, for comparison with Figure 16, while the fluctuation data are shown in Figure 29 (compare with Figure 17). Figure 28 shows the effect of target site bias on the Zipf analysis by superimposing a series of Zipf plots with different target site biases. Visually, the greater the bias the steeper the initial part of the line appears. Fitting a straight line to the first 30 points gave Zipf exponents of 0.51 for the 1:1:1 run; 0.50 for 2:3:5 and 0.66 for 1:3:6 (bias expressed in form TATA:TCAG:GGAC). The Zipf plots for the simulations with bias 5:3:2 and 6:3:1 produced curves similar to their 2:3:5 and 1:3:6 equivalents. The 5:3:2 simulation gave a gradient of 0.54, and the 6:3:1 run gave a gradient of 0.59.

Note that in the 1:3:6 and 2:3:5 simulations, the bias was towards GGAC, which is the most common target site in the starting sequence, and in the 6:3:1 and 5:3:2 simulations the bias is for TATA - the least common. It might be expected that bias towards the least common site would produce a higher Zipf exponent. During the duplication event with a copy length of 6 the target site is copied

along with two adjacent bases – it is these two bases which give some variety to the "words" which are enriched during the simulation. With a relatively rare target site there is a smaller selection of "words" that may be enriched, so each of these "words" should appear at higher rank in the Zipf analysis. Consider an example where a target site (eg. TCAG) occurs only once in a DNA starting sequence. If the bases immediately to the left of it are, for example 'at', then for each insertion event at this site type, the sequence ATTCAG will be duplicated. Zipf words derived from this sequence would be very common, leading to a high gradient on the Zipf plot. In fact, it is interesting to note that bias towards an uncommon target site does not seem to generate higher Zipf exponents than bias towards a common site – it is only the overall bias that is important, not which target sites are biased for.

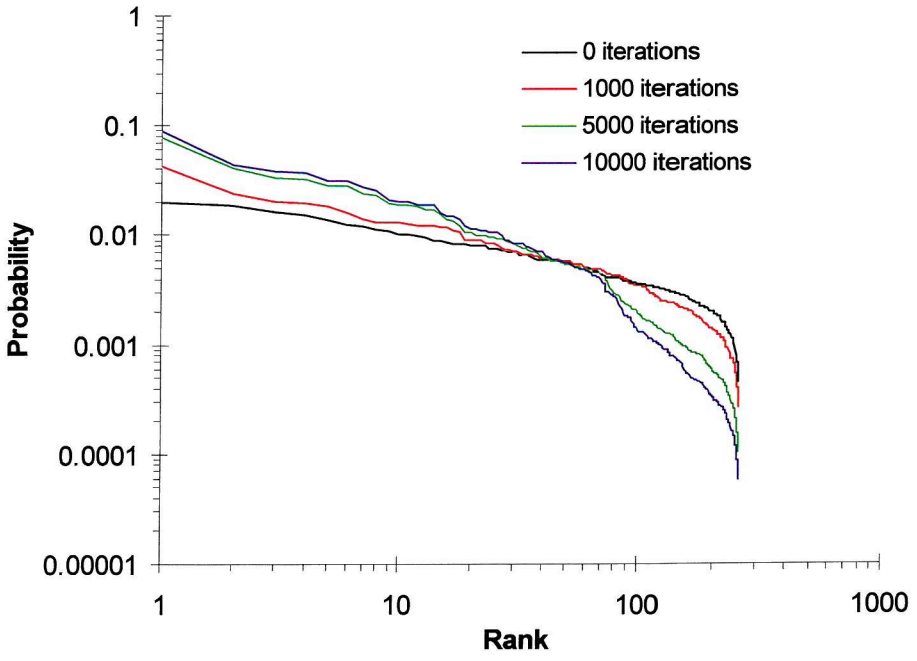


Figure 27 Zipf plots showing effect of target site bias; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:3:6 respectively; copy length 6; analysed with word length 4

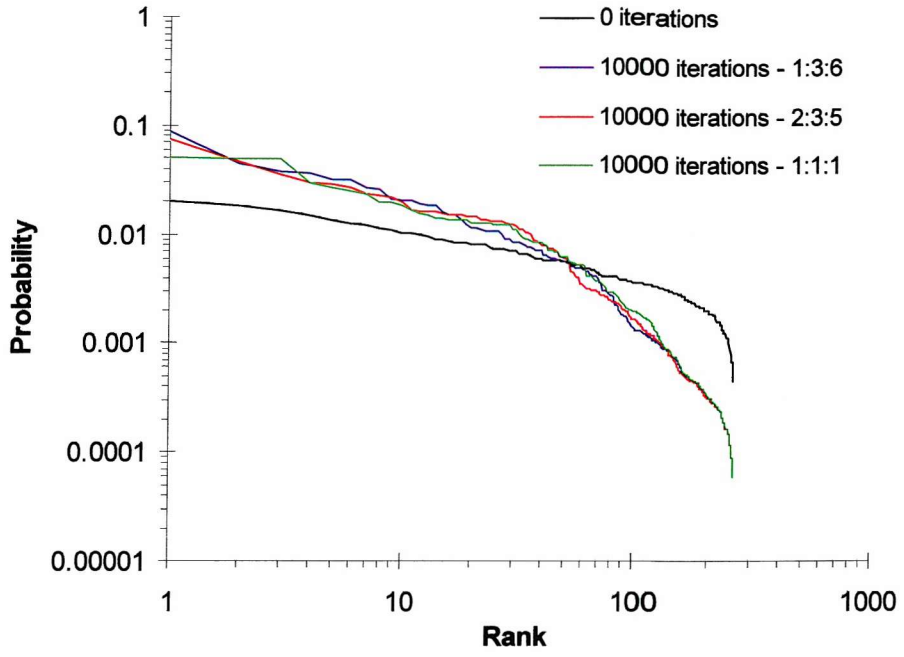


Figure 28 Zipf plots showing effect of target site bias; 10000 insertions into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities given in figure; copy length 6; analysed with word length 4

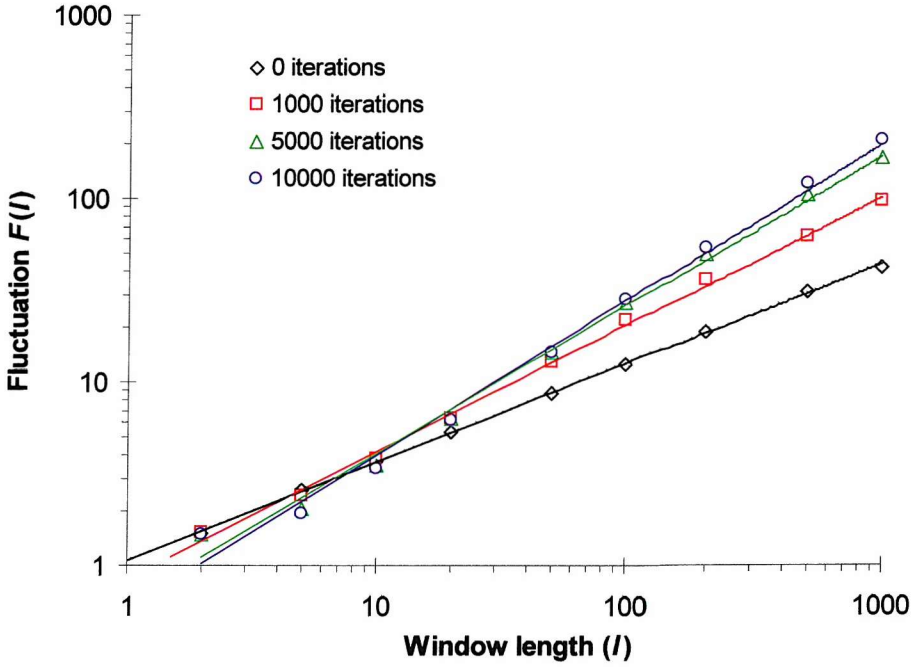


Figure 29 Fluctuation analysis showing effect of target site bias; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:3:6 respectively; copy length 6. The values of the fluctuation exponent are 0.69, 0.81, and 0.85 for 1000, 5000 and 10000 iterations respectively

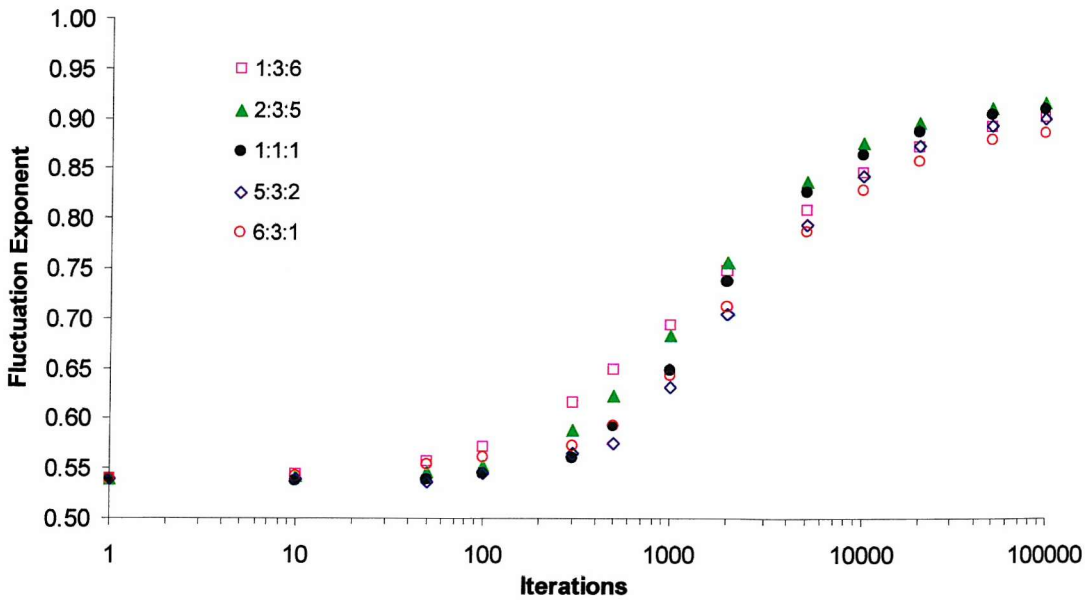


Figure 30 Variation of fluctuation exponent with iterations for sequences with target site bias; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities giveen in figure; copy length 6

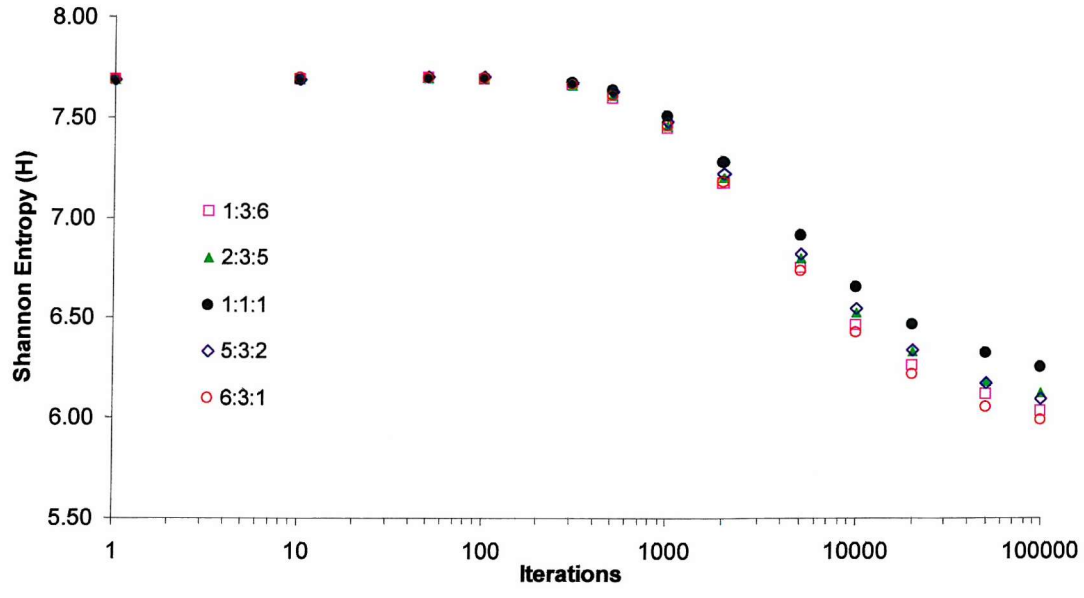


Figure 31 Variation of Shannon Entropy with iterations showing effect of target site bias; insertions into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities given in figure; copy length 6; analysed with word length 4

Figure 30 shows how the fluctuation exponent varies with iterations for runs with different target site bias. There is no obvious trend. Figure 31 shows the variation in the Shannon entropy with iterations. In this case the simulations with a greater bias (1:3:6 and 6:3:1) show lower entropy after 100,000 iterations. The unbiased run shows the highest entropy after 100,000 iterations, while the simulations run with bias 2:3:5 and 5:3:2 fit between the two extremes.

Target Site Bias	Number of TATA Sites	Proportion of TATA Sites	Number of TCAG Sites	Proportion of TCAG Sites	Number of GGAC Sites	Proportion of GGAC Sites	Total Number of Sites
1:1:1	3304	0.33	3353	0.33	3441	0.34	10098
2:3:5	1980	0.20	3140	0.31	4978	0.49	10098
1:3:6	1010	0.10	3029	0.30	6059	0.60	10098

Table 8 Proportions of target sites following biased simulation; 10000 insertions into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities given in the table in form TATA:TCAG:GGAC; copy length 6

Table 8 gives the numbers of target sites found in the sequences after 10,000 iterations. The proportions of target sites relative to one another confirm that the simulation is working correctly and that the proportions tend towards the bias built into each run.

4.9 *EFFECT OF STARTING SEQUENCE*

Figure 32 shows an example of a Zipf plot obtained using a random sequence of DNA as the start point. The details of the random sequence are given earlier in 'Starting Sequences Used'. This clearly shows the increase in gradient achieved by carrying out the simulation, from the random sequence with ζ=0.1, to the sequence obtained after 10,000 iterations with ζ=0.65. The final gradient achieved is very similar to that obtained when using HTLV II as the starting sequence and running with the same target site selection bias. At 10,000

iterations the Zipf plot shows three distinct steps - these may correspond to the three target sites used.

Iterations	Number of TATA Sites	Proportion of TATA Sites	Number of TCAG Sites	Proportion of TCAG Sites	Number of GGAC Sites	Proportion of GGAC Sites	Total Number of Sites
0	38	0.38	36	0.36	27	0.27	101
10	39	0.35	39	0.35	33	0.30	111
100	47	0.23	63	0.31	91	0.45	201
1000	125	0.11	338	0.31	638	0.58	1101
10000	1041	0.10	2987	0.30	6073	0.60	10101
100000	10011	0.10	30326	0.30	59764	0.60	100101

Table 9 The relative proportions of target sites using a random DNA sequence; insertion into random sequence; target sites TATA, TCAG, GGAC; site selection probabilities 1:3:6 respectively; copy length 6

Table 9 shows the number of target sites in the random DNA sequence before and during the simulation. The random starting sequence shows different proportions of each target site compared to HTLV II, but the total number of sites is quite similar (see Table 6). After 10,000 iterations the proportions of each target site are identical to those seen when using the HTLV II sequence (Table 8).

The fluctuation results in Figure 33 show no significant difference between starting sequences after iterations had been performed. The random starting sequence gave a fluctuation exponent of 0.48. The variation of fluctuation exponent with iterations for insertion into a random sequence is given in Figure 34, while Figure 35 shows the variation of Shannon entropy with iterations. These figures clearly demonstrate that the difference due to the initial value of the starting sequences diminishes as the simulation progresses to higher iterations. Also of interest is the slight difference that is observed between the HTLV II and random starting sequences when the simulation has reached 100,000 iterations. This is observed for both the Shannon entropy and

fluctuation exponent plots, and appears after the HTLV II and random sequence values appear to converge at 10,000 iterations.

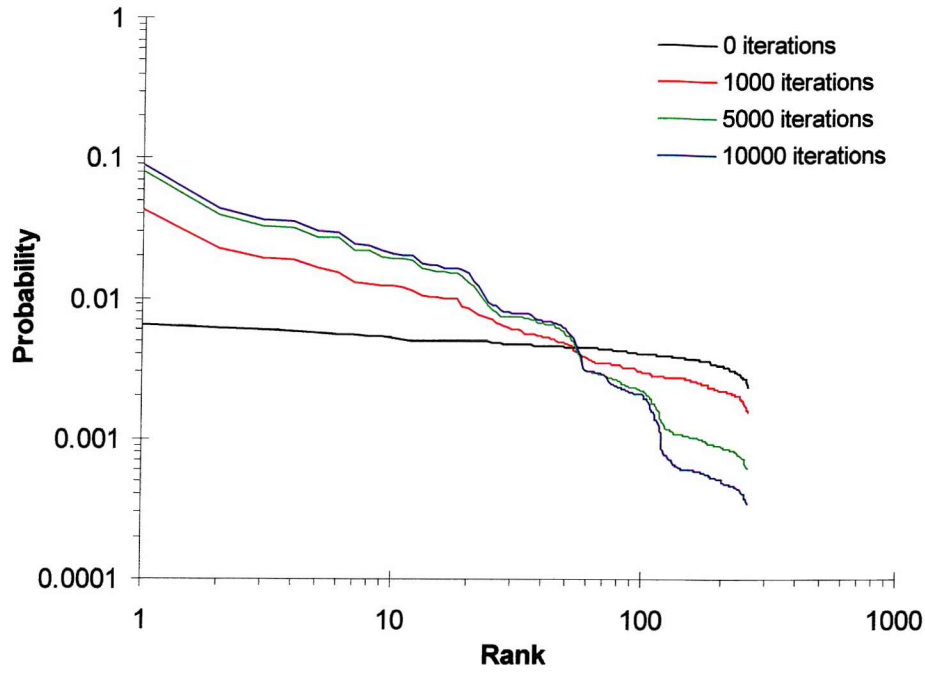


Figure 32 Zipf plots showing effect of starting sequence; insertion into random DNA sequence; target sites TATA, TCAG, GGAC; site selection probabilities 1:3:6 respectively; copy length 6; analysed with word length 4

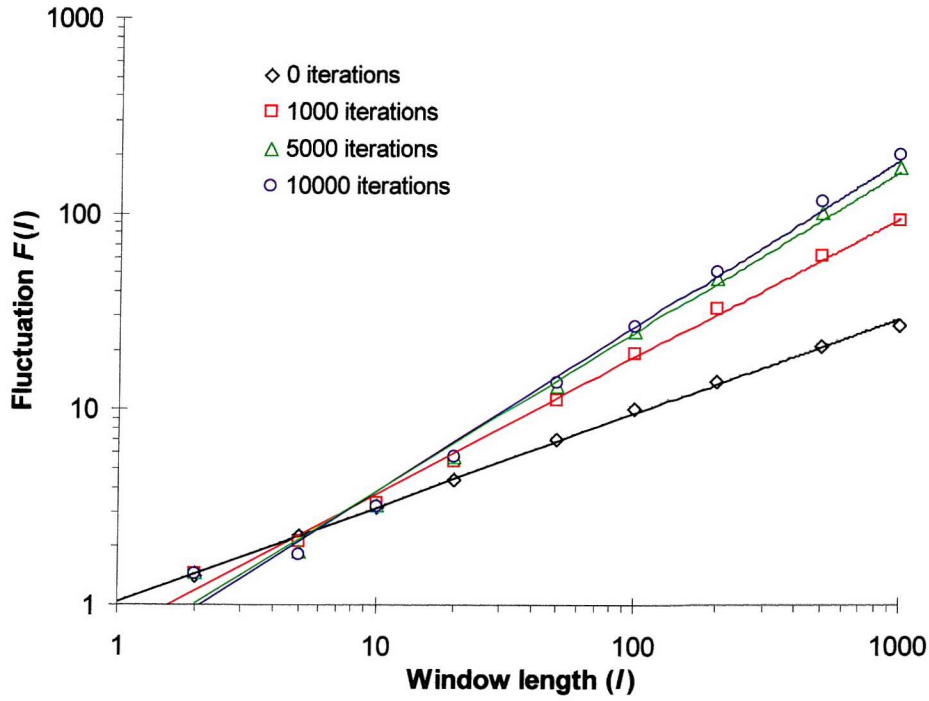


Figure 33 Fluctuation analysis showing effect of starting sequence; insertion into random DNA sequence; target sites TATA, TCAG, GGAC; site selection probabilities 1:3:6 respectively; copy length 6

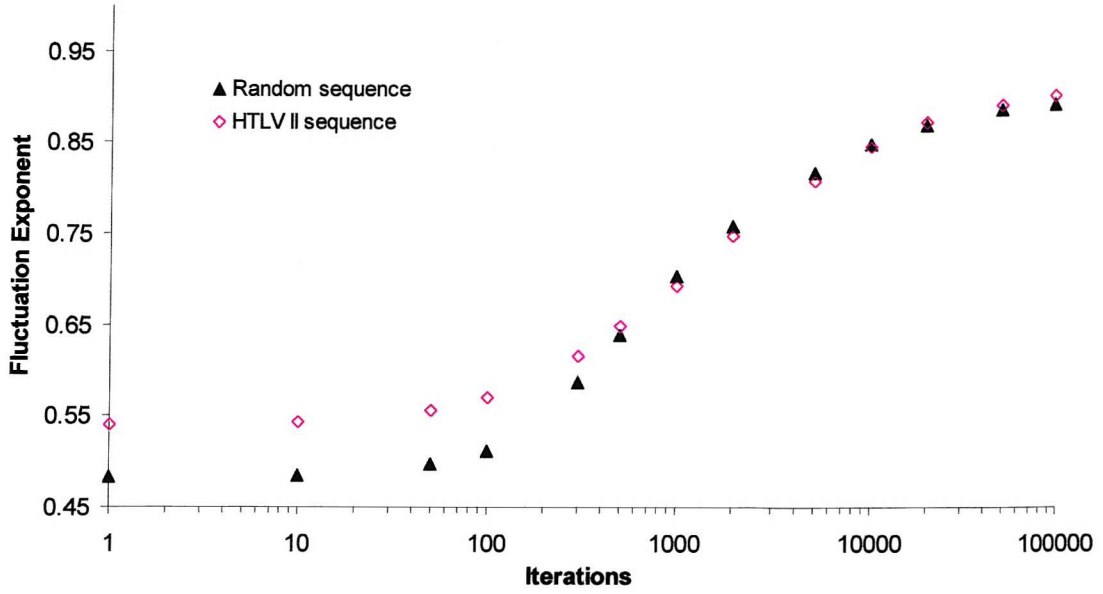


Figure 34 Variation of fluctuation exponent with iterations showing effect of starting sequence; starting sequences given in figure; target sites TATA, TCAG, GGAC; site selection probabilities 1:3:6 respectively; copy length 6

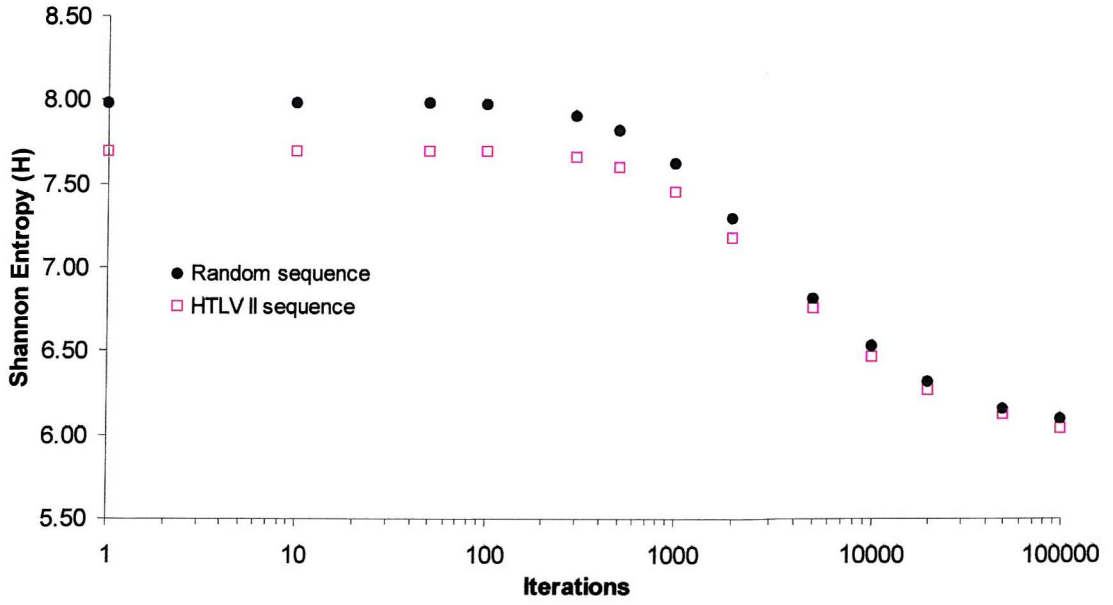


Figure 35 Variation of Shannon Entropy with iterations showing effect of starting sequence; starting sequences given in figure; target sites TATA, TCAG, GGAC; site selection probabilities 1:3:6 respectively; copy length 6; analysed with word length 4

4.10 VARIABLE TARGET SITE MODEL

In order to try and make the model more realistic, a modification was made to take account of the variable target site duplications seen in transposable elements[60, 59, 58]. The new simulation used target site copy lengths of 2 bp for TATA, 6 bp for TCAG, and 10 bp for GGAC.

Fluctuation results showed no significant differences from the previous model up to 10,000 iterations, although the quality of the Zipf plot seemed to be improved – see Figure 36 for Zipf plots and Figure 37 for Fluctuation results. Figure 38 gives a comparison between the simulation run with variable copy lengths, and that with all copy lengths equal. The variable target site model results in a Zipf plot which has a large plateau at lower ranks, but appears linear over a large portion in the centre of the plot. The slope of the line for the variable copy length simulation is 0.62 at 10,000 iterations showing a large increase in gradient over the non-variable simulation (which gave ζ=0.51).

The relationship of the fluctuation exponent and Shannon entropy with the number of iterations are given in Figure 39 and Figure 40. Both the fluctuation exponent and Shannon entropy values appear to level off sooner for the variable copy length model than the non-variable model. The final fluctuation exponent is lower for the variable model and the final Shannon entropy higher for the variable model compared to the non-variable model.

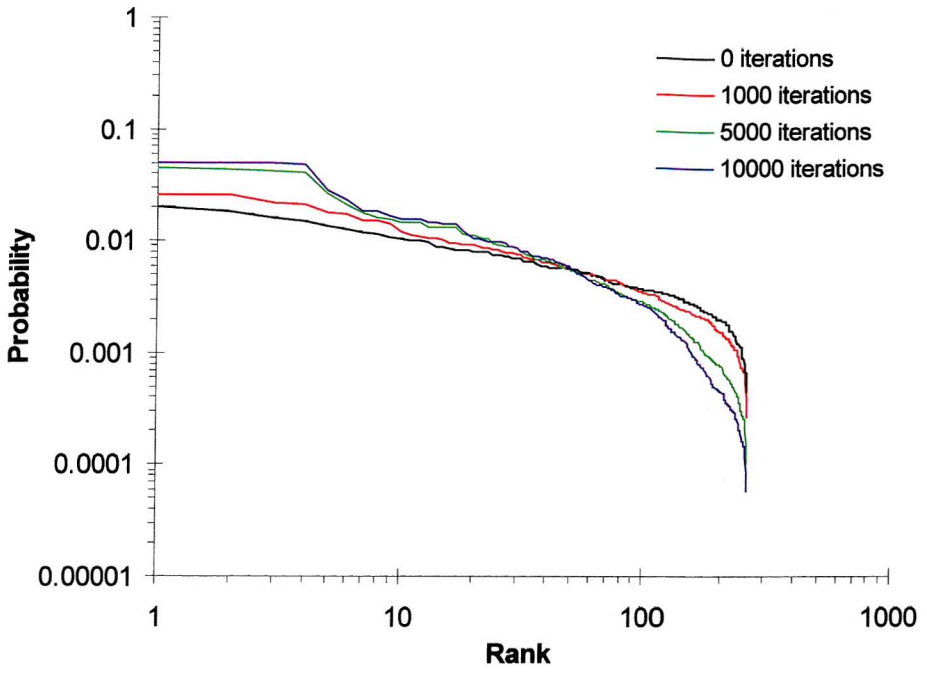


Figure 36 Zipf plots showing effect of a range of copy lengths; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy lengths 2:6:10 respectively; analysed with word length 4

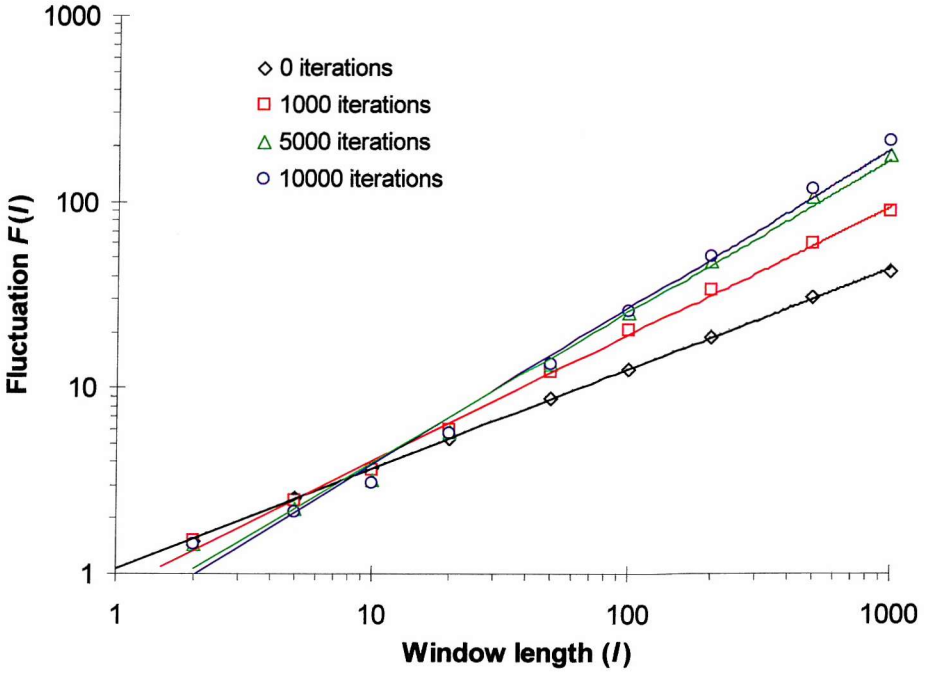


Figure 37 Fluctuation analysis showing effect of a range of copy lengths; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy lengths 2:6:10 respectively

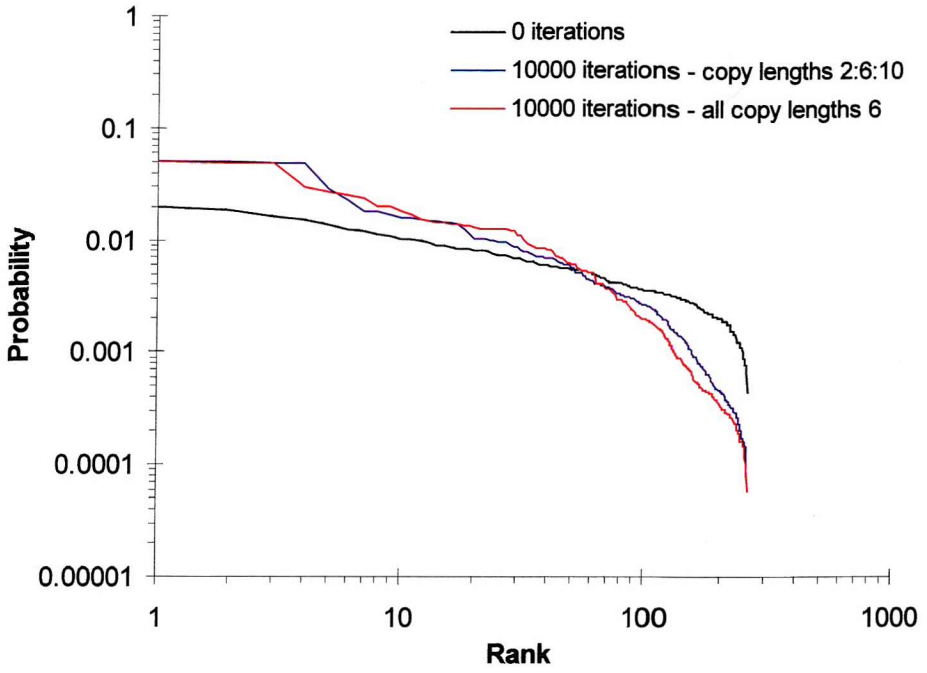


Figure 38 The effect of a range of copy lengths on Zipf plots; 10000 insertions into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy lengths given in the figure; analysed with word length 4

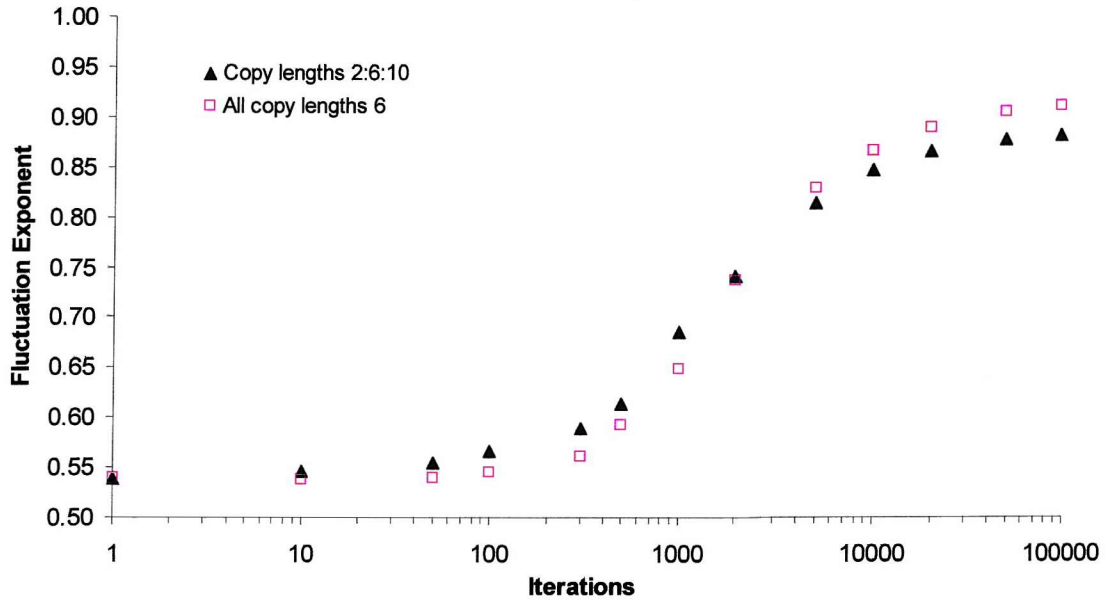


Figure 39 Variation of fluctuation exponent with iterations showing effect of a range of copy lengths; insertions into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy lengths given in the figure

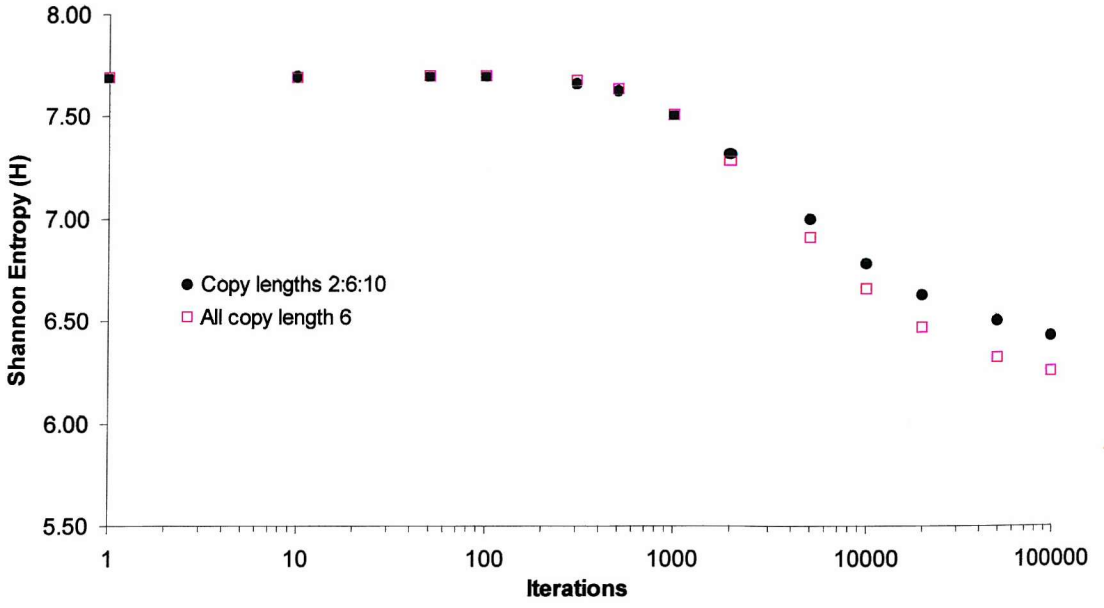


Figure 40 Variation of Shannon Entropy with iterations showing effect of a range of copy lengths; insertions into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy lengths given in the figure; analysed with word length 4

4.11 DIFFERENCES BETWEEN SIMULATED AND REAL DNA

The model that was developed clearly shows how transposable element insertion and perfect excision can generate statistical properties indicative of the language-like features that are observed in non-coding DNA sequences, from a starting sequence initially devoid of linguistic character. There are, however, two areas in which the sequences generated by the model do not match real non-coding DNA sequences. Firstly, on visual inspection of the sequences they appear very blocky. Figure 41 shows a typical sample of a sequence generated in which the repeated units are clearly visible. Secondly, close examination of the fluctuation analysis plots shows that the curves are not strictly linear in the region $\log l = 0.7\text{-}1$.

```
TGACAATGGCGACTAGCCTCCCAAGCCAGCCACCCAGGGCGAGTCATCGACCCAAAAGGT
CAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGT
CAGGGTCAGACCGTCTCACACAAACAATCCCAAGTAAAGGCTCTGACGTCTCCCCTTTT
TTTAGGAACTGAAACCACGGCCCTGACGTCCCTCCCCCCTAGGAACAGGAACAGCTCTCC
AGAAAAAAATAGACCTCACCCTTACCCACTTCCCCTAGCGCTGAAAAACAAGGCTCTGAC
GATTACCCCCTGCCCATAAAATTTGCCTAGTCAAAATAAAAGATGCCGAGTCTATATCTA
TATCTATATCTATATCTATATCTATATCTATATCTATATCTATATCTATATCTATATCTA
TATCTATATCTATATCTATAAAAGCGCAAGGACAAGGACAAGGACAAGGACAAGGACAAG
GACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAG
GACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAG
GACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAG
GACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAG
GACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAG
GACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAGGACAAG
GACAAGGACAAGGACAAGGACAGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGT
TCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGT
TCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGT
TCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGT
TCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGT
TCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGTTCAGGT
```

Figure 41 The blocky nature of sequences produced by the simulation. Colour coding helps to highlight the sequences produced by target site duplication. The first 1200 bases of a sequence generated after 5000 insertions into HTLV II are shown; target sites TATA, TCAG, GGAC; site selection probabilities 1:3:6 respectively; copy length 6

The 'blockiness' of the sequence is to be expected, bearing in mind the simple nature of the simulation. Up to 600,000 bases are added to the 8,952bp of the starting sequence, and all of the bases added are perfect copies of sites within the starting sequence, placed adjacent to those sites. It is no surprise that this procedure generates a sequence dominated by blocks of repetitive DNA. Figure 42 shows that using a different copy length for each target site still generates a blocky sequence. The origin of the depression in the fluctuation lines is not immediately obvious, however, and requires examination of the details of the fluctuation analysis.

```
TGACAATGGCGACTAGCCTCCCAAGCCAGCCACCCAGGGCGAGTCATCGACCCAAAAGGT
CAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGT
CAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGT
CAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGT
CAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGT
CAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGT
CAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGT
CAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGT
CAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGT
CAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGT
CAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGT
CAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGT
CAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGT
CAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGT
CAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGT
CAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGT
CAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGT
CAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGTCAGGGT
CAGGGTCAGGGTCAGACCGTCTCACACAAACAATCCCAAGTAAAGGCTCTGACGTCTCCC
CCTTTTTTTAGGAACTGAAACCACGGCCCTGACGTCCCTCCCCCCTAGGAACAGGAACAG
```

Figure 42 Sequence generated using range of copy lengths. Colour coding helps to highlight the sequences produced by target site duplication. The first 1200 bases of a sequence generated after 5000 insertions into HTLV II are shown; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy lengths 2:6:10 respectively

The fluctuation analysis should generate a straight line, with $\alpha > \frac{1}{2}$ for a sequence containing long-range correlations or $\alpha = \frac{1}{2}$ for a sequence containing no correlations, or correlations over a well-defined length scale. The fluctuation analysis is seen to give straight lines in both the cases of coding and non-coding DNA sequences, while the artificial sequences generated by the model show a curve in the line. The depression in the line can be seen more clearly by plotting the residuals of the curve. This is obtained by subtracting the linear fit from the fluctuation data to give a plot showing the deviation of the data from the linear fit. Such a plot is shown in Figure 43. The chart shows that the maximum depression in the fluctuation data line occurs roughly in the range of window

lengths 5-10 used in the fluctuation analysis. This range for the window length is approximately the same as the copy lengths that were used in the generation of the sequences - the dip in the line may therefore be related to the repeat period of the repetitive blocks that are found in the final sequences.

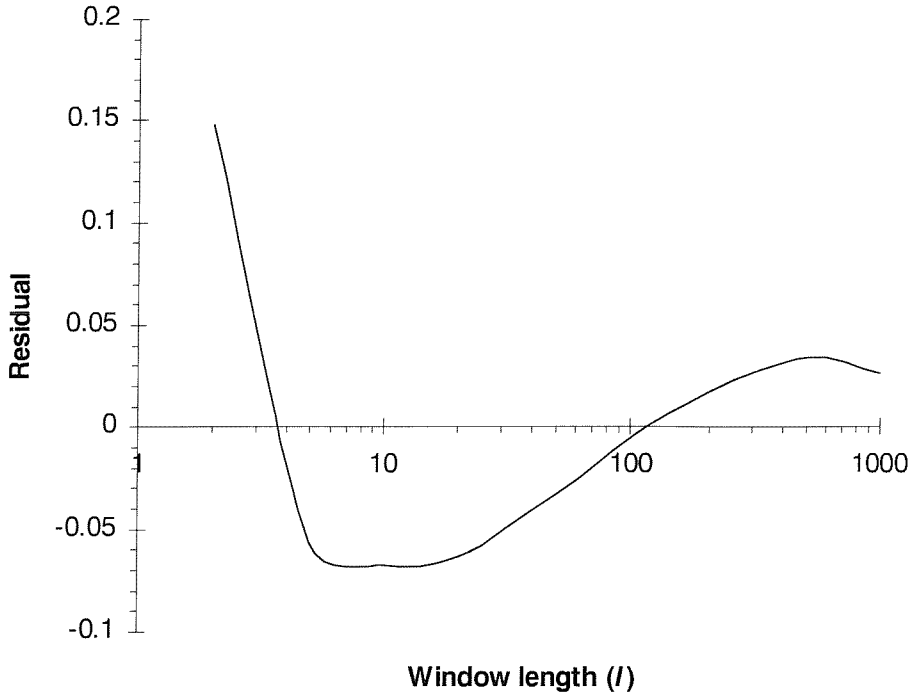


Figure 43 The residual of the fluctuation plot for the initial run after 10,000 iterations. Insertion into HTL VII; target sites tata, tcag, ggac; site selection probabilities 1:1:1 respectively; copy length 6

Figure 44 shows, in detail, the fluctuation analysis of a repetitive sequence when the window length used in the analysis is the same as the period of the repeat in the sequence. This shows that a low value of $F(l)$ is obtained at this window length, and therefore explains how a depression in the fluctuation line might be generated when analysing a repetitive sequence, and why the dip will appear at the value of l corresponding to the period of the repeats.

These two problems highlight the simplistic nature of the original model. In real systems, the insertion and excision of transposable elements rarely generate perfect copies of the target sequence, and other mutagenic processes would act

to break up the regular structure produced by the original simulation. We postulated that this regular structure produces the observed inconsistencies between the real DNA and the simulated DNA. With this in mind, the realism of the model was improved by simulating additional processes in the genome that would introduce 'noise' into the system, helping to break up the blocks of repeated sub-sequences. The aim of these new models was to produce non-coding DNA sequences that more closely resemble those found in real life and to produce a more complete simulation of the natural system by modelling more of the processes seen *in vivo*.

STAGE 1 ***Apply RY rule to sub-sequences***

Sequence: CTACCTAC

Window length: 4

	CTAC	TACC	ACCT	CCTA	CTAC
$\Delta y(l)$	+2	+2	+2	+2	+2
$\Delta y(l)^2$	+4	+4	+4	+4	+4

STAGE 2 ***Calculate*** $F(l)$

a) From stage 1: $\overline{\Delta y(l)} = \frac{2+2+2+2+2}{5} = 2$

$$\left[\overline{\Delta y(l)}\right]^2 = 4$$

From stage 1: $\overline{\left[\Delta y(l)\right]^2} = \frac{4+4+4+4+4}{5} = 4$

b) $F^2(l) = \overline{\left[\Delta y(l)\right]^2} - \left[\overline{\Delta y(l)}\right]^2$

$= 4 - 4$

$F(l) = 0$

Figure 44 The fluctuation analysis of a repetitive sequence. This diagram illustrates the effect of repetitive sequences on the fluctuation analysis, when the window length used is the same as the period of the repeats. In this case, every sub-sequence generated by the sliding window is identical, in terms of the RY rule. The overall effect is that the the difference between the square of the average and the average of the square is zero, and the value of $F(l)$ is therefore also zero. Compare this with the previous example in Figure 3.

5 Initial Improvements to the Transposable Element Model

Before additional genome reshaping processes were incorporated into the model, several other modifications were necessary. The original program had been written with the aim of testing the hypothesis that language-like features could be produced in DNA as a consequence of the accumulation of tandem repeats. The code was not optimised for high performance. Furthermore, the program was not particularly flexible in that it had been written to solve a very specific initial problem. This made it difficult to add new target sites that each had different copy lengths. The initial refinements to the original model concentrated on resolving these two problems of performance and flexibility.

5.1 *INITIAL MODIFICATIONS*

When writing the original version of the model, the primary consideration was to make sure that it worked correctly and for this reason it was written using relatively basic programming principles, using the C programming language. The program also had to run under Microsoft Windows and MS-DOS, as this was the operating environment available at the time. The main problem faced during the development of the program was how to handle the long continuous DNA sequences. In Windows and MS-DOS, a limit is placed on the maximum length of such strings (64 Kbytes, equivalent to a DNA sequence of length 64,000 bases). When the program was written is was felt that this would be too small for some of the simulations that would be run (e.g. the starting sequence HTLV II is 8,952 bp, so 10,000 insertions with a copy length of 6 produce a final sequence length of 68,952 bases). As a result, the code was written to hold the string in a file on disk, and carry out the modifications on the file. This approach

involves periodically reading from and writing to the disk, which is a very slow process compared to memory access. While the code was not prohibitively slow for the initial simulations (the runs of 10,000 cycles took approximately 2½ hours), it was clear that speed would decrease as more features were added to the model, for example, as longer sequences were used and as more iterations were performed. Indeed, increasing the number of iterations beyond 10,000 would slow the program significantly, bearing in mind that 5,000 iterations take 40 minutes, but continuing to 10,000 iterations takes a further 110 minutes. This slowdown is due to the increasing length of the sequence with each iteration, so is also a measure of how longer starting sequences would affect performance. It would be possible to circumvent the problems of large string size in Windows and MS-DOS by storing the string in a data structure such as a linked list (splitting the string into its component characters), but this would increase the complexity of the program. Instead it was decided to write the program to run under the Linux operating system (a UNIX clone), which does not impose the 64 Kbyte limit on arrays.

A later version of the program was written which used a linked list to store the DNA sequence. The program was still run under Linux as large arrays were still used in the program (but not for storing the DNA sequence). One of the advantages of a linked list is that new items can quickly be inserted into an existing list of items – an advantage applicable to the insertion of copied bases into the DNA sequence as performed in the simulation. The program written in this fashion was dramatically quicker than the versions using an array to hold the DNA sequence, as the simulation did not slow as the DNA sequence increased in length. Simulations run to 100,000 iterations might take several days to run using the array version of the simulation – using a linked list cut this down to several minutes. The linked list version of the simulation was the final version of the simulation and so incorporated all the features described here and in the following sections.

While the program was being re-written to use memory rather than disk storage, several other improvements were also made relating to the addition of new target sites to the model. These improvements were as follows:

1) Number of target sites: In previous work 3 x 4 base-pair sites were used as targets for simulated insertion/excision of transposable elements. In real systems there are many different types of transposable element, all with different target sequence preferences[58, 59, 60]. Additional target site sequences would improve the realism of the simulation, but these were difficult to add to the original program due to the way in which it was written. The new version of the program was altered to allow target sites to be easily added and removed.
2) Target site length: As noted above, the target sites originally used were all 4bp in length. This is not the case in living systems. The different target sites can be of various lengths[58, 59, 60]. In the original program, target site lengths could not easily be changed – this was resolved in the new version of the simulation.
3) Length of duplicated DNA: During an insertion by a transposable element a number of bases at the target site are duplicated. The length of this duplication varies from transposable element to transposable element, but is typically in the range 5-9 bases[49, 50, 51]. Selecting different copy lengths for different target sites was possible with the old program, but changes were not easy to make. The new version was modified so that the copy length could easily be altered.

Following the modifications to the original program, it was found that the time taken for 10,000 insertions into HTLV II was reduced from 2½ hours to approximately 30 minutes.

Testing of the new program showed that the new and the old programs produced the same results when initialised with the same parameters, except for a difference that was noted between the Zipf analyses produced using the two programs. A series of Zipf plots of sequences generated by the two programs

using the same parameters formed two distinct groups, one for each program. It was thought that this could be due to the random number generator in use, rand(), from the C programming language. The differences could arise from the different structures of the programs - the random number generator was not used identically in both programs - or from differences in the random number generator itself. In Linux the g++ compiler was used, while in Windows/MS-DOS the Borland Turbo C++ compiler was used. It is possible that the implementation of rand() was not identical in both compilers.

In any case, a study of reference 68, 'Numerical Recipies in C', containing a detailed analysis of random number generators showed that rand() was not appropriate for use in the program. Firstly, rand() is only required by the ANSI standard governing C compilers to generate random numbers in the range 0-32,675, which is not a large enough range if many numbers are to be generated in a single run, as is required by this simulation. Secondly, the mathematical method used by rand() to generate random numbers is known to be insufficient to produce a sequence of numbers which are sufficiently uncorrelated for use in serious applications, and finally, of the various versions of rand() at least some are known to be flawed in their implementation, again, giving rise to sequences of numbers which cannot be regarded as 'random'. As a result, rand() was substituted for a random number generator taken from ref. 68, ran2(). This has been shown to generate a series of numbers that are sufficiently random for use in applications such as the model described here, while still maintaining acceptable performance (about half the speed of rand()).

Testing of the model incorporating ran2() showed that the Zipf plots obtained did not fall into a clearly defined group when compared with those previously generated, but still appeared to show similar gradients (a comparison between the old and new versions is given below).

With these initial modifications carried out the effects of the new features of the program were investigated, with the aim of to trying to remove the observed dip in the fluctuation lines.

5.2 *COMPARISON WITH PREVIOUS RESULTS*

Figure 45 shows the Zipf plots for the new and old simulation when run with the same parameters. The lines appear closely matched and both have the same gradient of 0.53.

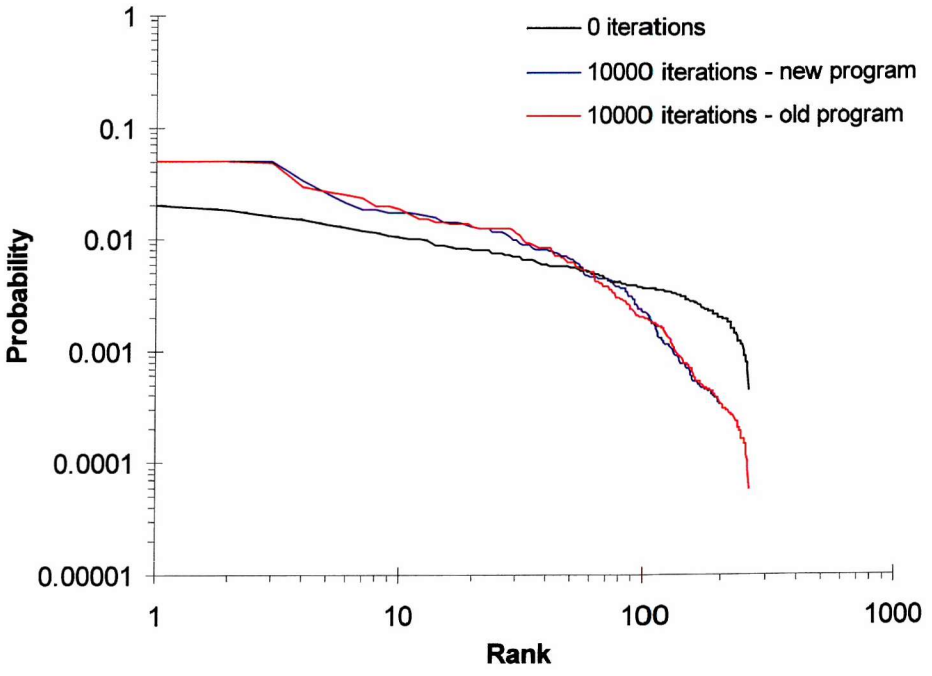


Figure 45 Zipf plots showing differences between old and new simulations; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; analysed with word length 4

The effect of iterations on the fluctuation exponent is shown in Figure 46 for both the old and new versions of the program. Again, the results appear similar. The fluctuation exponent at 10,000 iterations is 0.86 for the new simulation and 0.87 for the old.

Figure 47 shows the variation of Shannon entropy with iterations for both simulations. The two simulations produce very similar results.

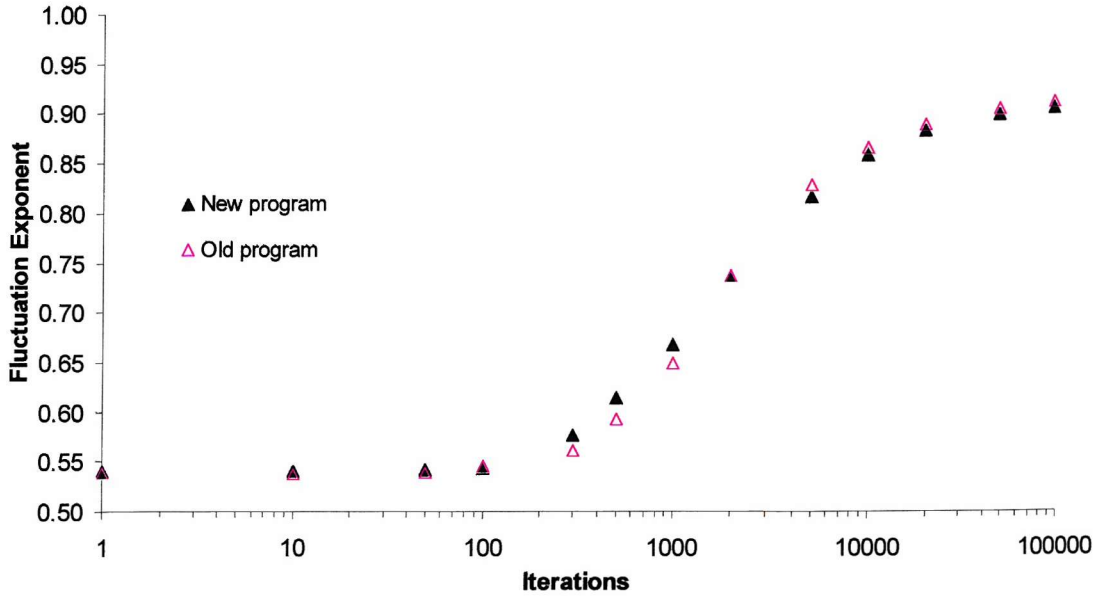


Figure 46 Variation of fluctuation exponent with iterations showing differences between old and new simulations; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6

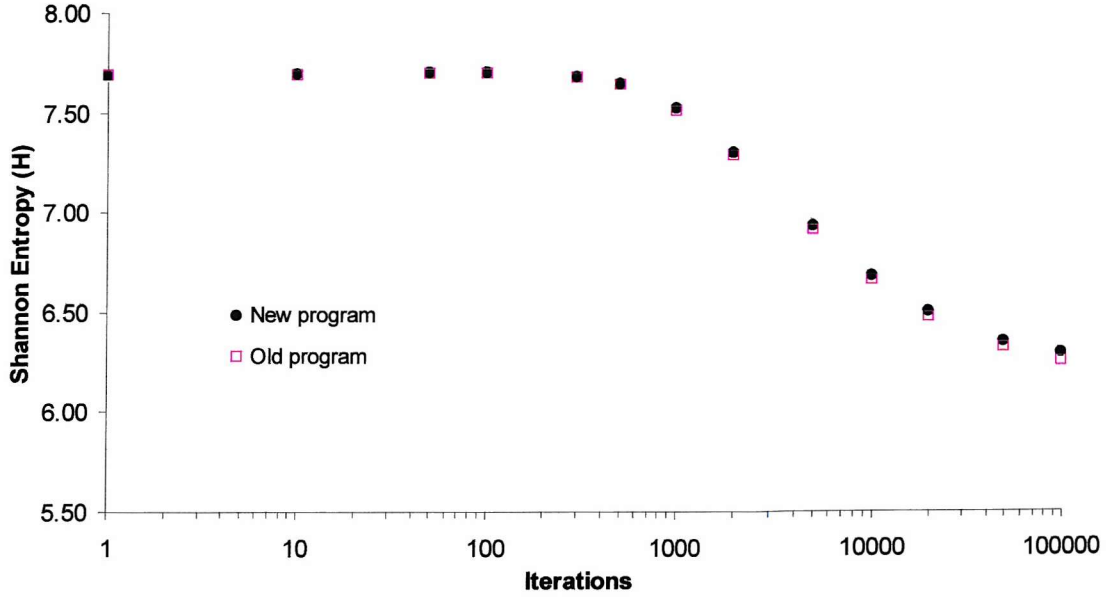


Figure 47 Variation of Shannon Entropy with iterations showing difference between old and new simulations; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; analysed with word length 4

It seems, therefore, that the alterations made to the program in order to improve its performance and flexibility have not affected its basic operation. Due to the problems with the old random number generator, however, results obtained with the new model are not directly comparable with those obtained using the old simulation. Where comparisons between two or more different runs are made, care has been taken to ensure that both were obtained using the same version of the model. All the results from this point were generated using the newer version of the program.

5.3 *EFFECT OF TARGET SITE BIAS*

This effect was investigated using the original model as presented in the previous section. The performance increase obtained with the new model made it possible to further investigate the effect by running additional simulations at an even higher bias than previously used. Three additional runs were made with target site selection bias 1:3:9, 1:4:16 and 1:5:25 (all given as TATA:TCAG:GGAC ratios). The highest ratio used previously was 1:3:6.

Figure 48 shows the Zipf plots for the runs with bias 1:3:6, 1:4:16 and 1:5:25. As with the previous results, the greater the degree of bias the higher the start of the line appears. The lines for the more highly biased runs appear more uneven than that obtained with the 1:3:6 run. Values of ζ for the lines are 0.59 for 1:3:6, 0.62 for 1:3:9 (Zipf plot omitted for clarity), 0.68 for 1:4:16 and 0.69 for 1:5:25. These results strengthen the conclusion that greater target site bias leads to a higher Zipf exponent.

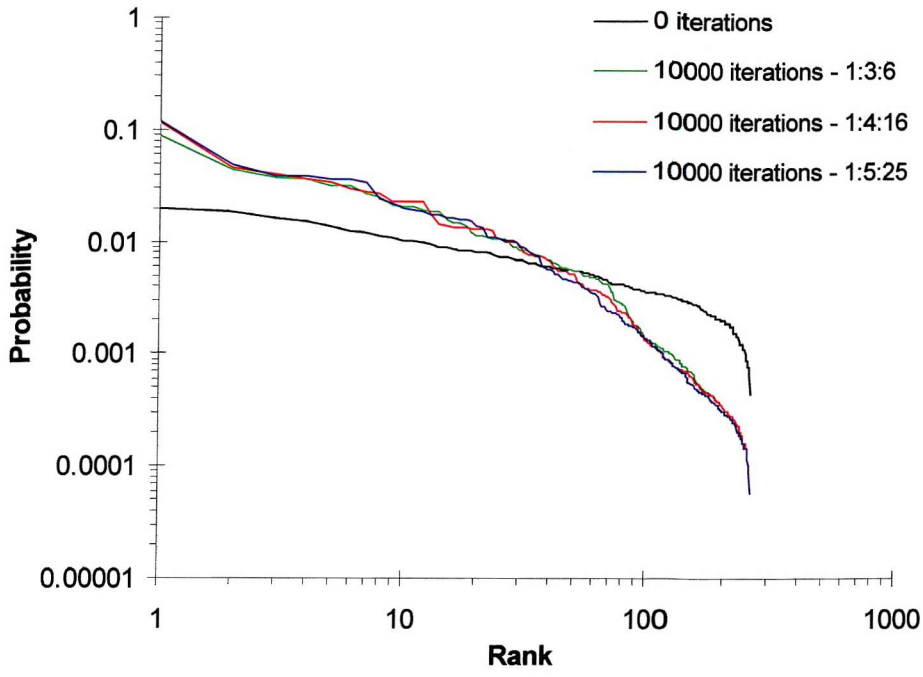


Figure 48 The effect of target site bias on Zipf plots; 10000 insertions into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities given in the figure; copy length 6; analysed with word length 4

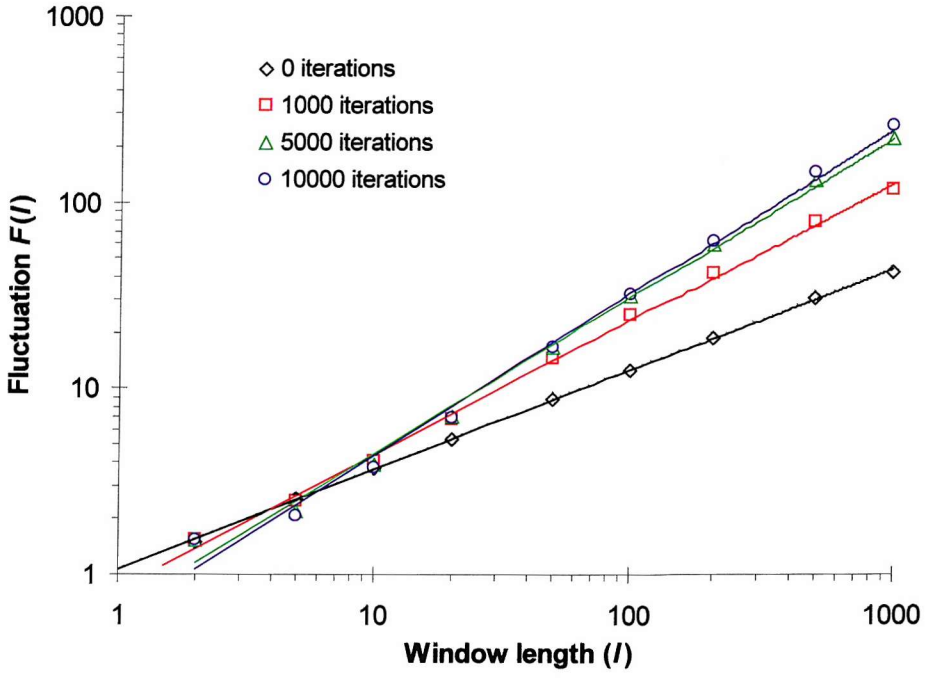


Figure 49 Fluctuation analysis showing effect of target site bias; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:3:9 respectively; copy length 6

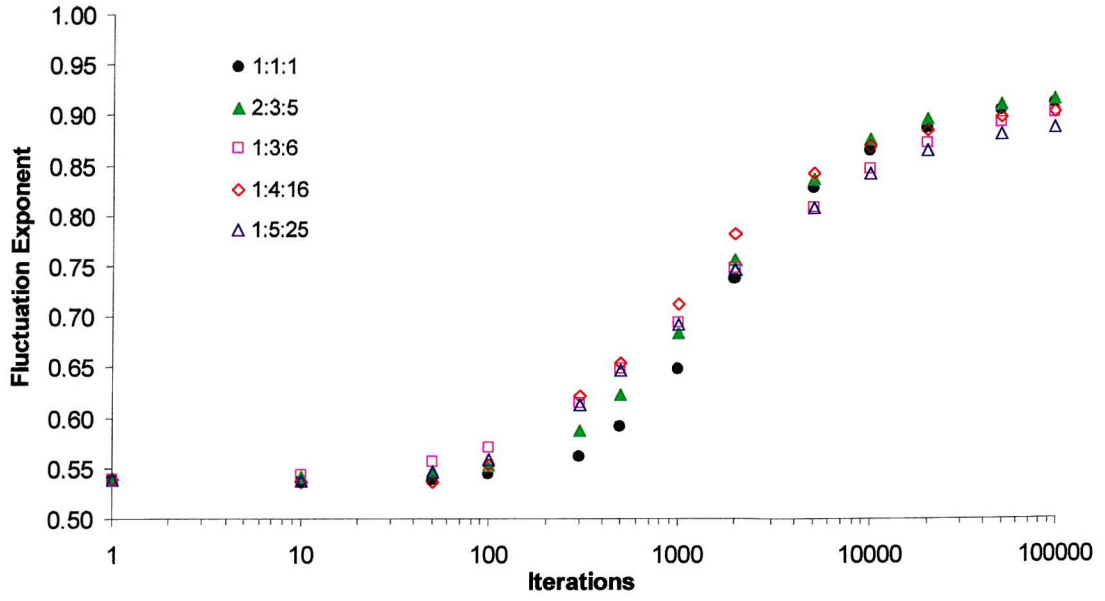


Figure 50 Variation of fluctuation exponent with iterations showing effect of target site bias; insertions into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities given in figure; copy length 6

The effect of iteration on the fluctuation exponent is presented in Figure 50. The results show that greater bias leads to a lower fluctuation exponent after many iterations, but that at lower iterations (in the region 100 – 2,000) the more heavily biased sequences lead to a higher value of α. As has been noted, the lines used to generate the fluctuation exponent exhibit a depression at around window lengths 5-10 which will affect the fitting of a straight line to the data. The depression is more noticeable at higher iterations (Figure 49 shows an example for a biased simulation – note the depression in the line for the 10,000 iterations sequence) and it may be that this affects the calculated value of the fluctuation exponent at these higher iterations. This could explain the low values of α seen at high iterations in Figure 50.

Figure 51 shows the variation of Shannon entropy with iterations for sequences with varying degrees of target site selection bias. There is a clear relationship - the greater the level of bias the lower the Shannon entropy.

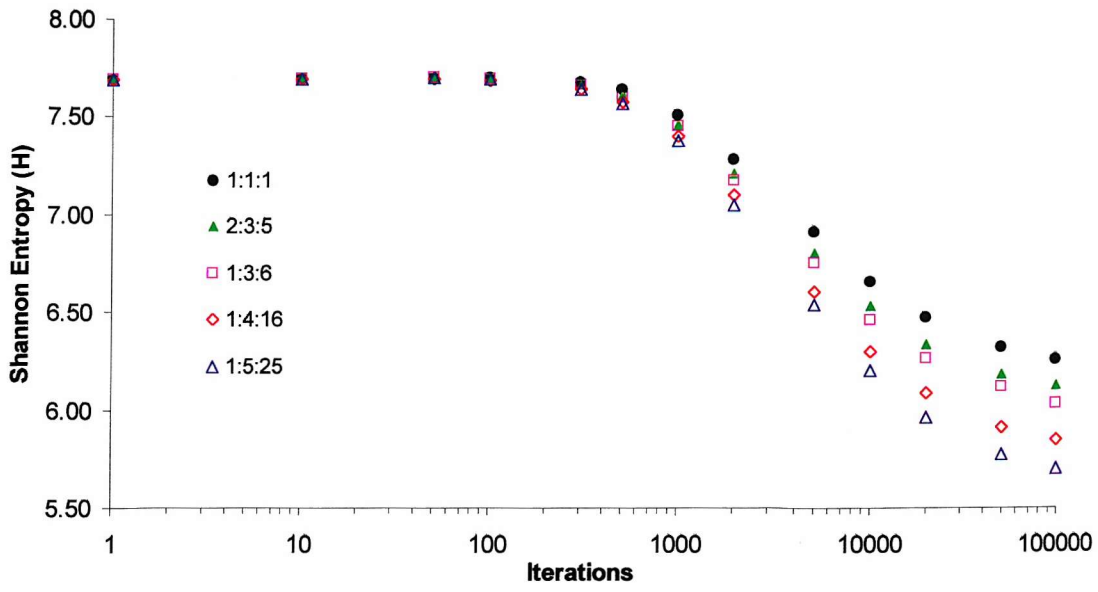


Figure 51 Variation of Shannon Entropy with iterations showing effect of target site bias; insertions into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities given in figure; copy length 6; analysed with word length 4

5.4 *USING MORE TARGET SITES*

The effect of adding more target sites was investigated by running the simulation with four and then five target sites. The runs were made once with no target site selection bias and again with a bias towards certain sites. Figure 52 shows a comparison of the Zipf plots obtained when 3, 4 and 5 target sites were used with no bias towards any site. Visually the lines appear quite different to one another, but with no obvious trend to follow going from 3 target sites to 5 sites. The measured Zipf exponents were 0.53 with 3 target sites, 0.63 with 4 target sites and 0.56 with 5 target sites. The Zipf plot for the sequence obtained when using 4 target sites does not appear to have a good linear portion, so the high value of ζ obtained for this run may be due to error in measuring the exponent.

Figure 53 shows how the fluctuation exponent varies with iterations for sequences obtained with 3, 4 and 5 target sites. There is little difference in the values obtained in each case.

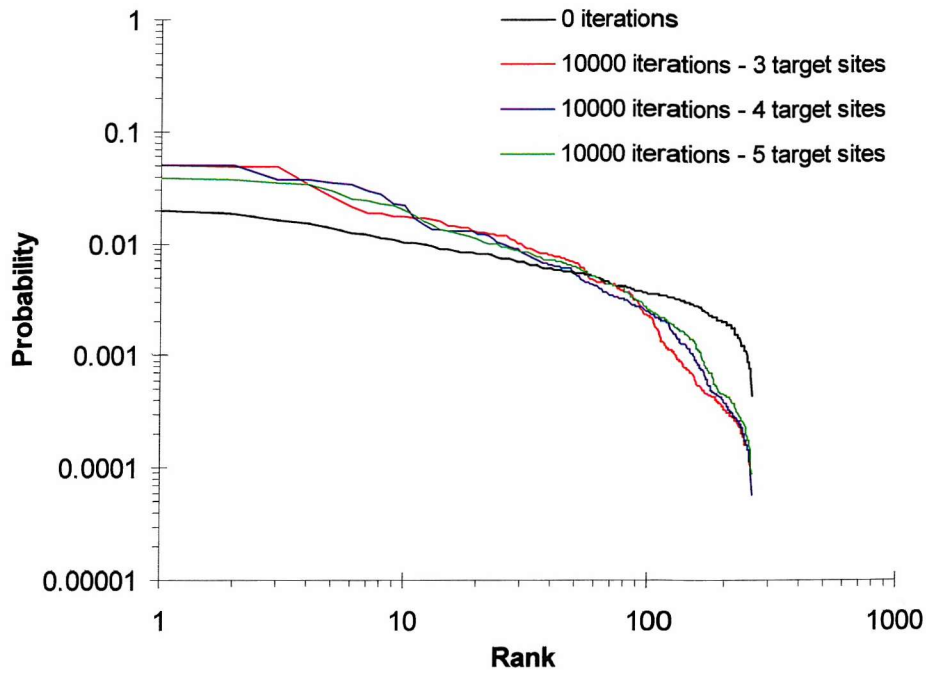


Figure 52 Zipf plots showing the effect of the number of target sites used; 10000 insertions into HTLV II; number of target sites given in figure; with 3 target sites – TATA, TCAG, GGAC site selection probabilities are 1:1:1 respectively; with 4 target sites – attc, TATA, TCAG, GGAC site selection probabilities are 1:1:1:1 respectively; with 5 target sites – cgtg, attc, TATA, TCAG, GGAC site selection probabilities are 1:1:1:1:1 respectively; copy length 6; analysed with word length 4

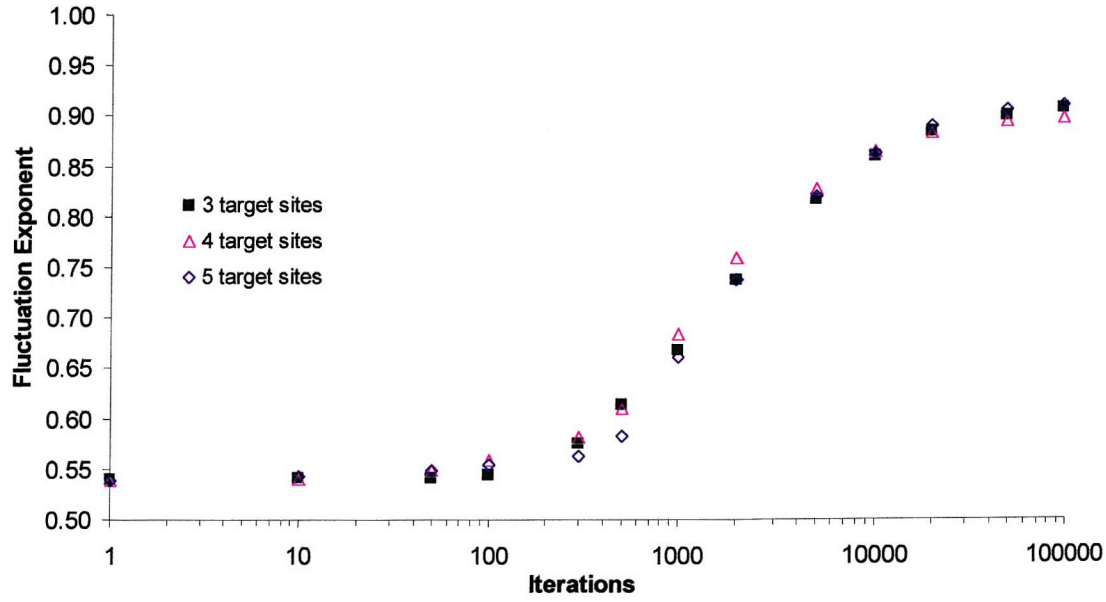


Figure 53 Variation of fluctuation exponent with iterations showing the effect of the number of target sites used; 10000 insertions into HTLV II; number of target sites given in figure; with 3 target sites – TATA, TCAG, GGAC site selection probabilities are 1:1:1 respectively; with 4 target sites – attc, TATA, TCAG, GGAC site selection probabilities are 1:1:1:1 respectively; with 5 target sites – cgtg, attc, TATA, TCAG, GGAC site selection probabilities are 1:1:1:1:1 respectively; copy length 6

There are, however, significant differences in the Shannon entropy data obtained using 3, 4 and 5 target sites, as shown in Figure 54. Using more target sites makes the Shannon entropy value start to decrease at higher iterations, and seems to make the end value of H lower (although it is not possible to see this for sure without going to higher iterations).

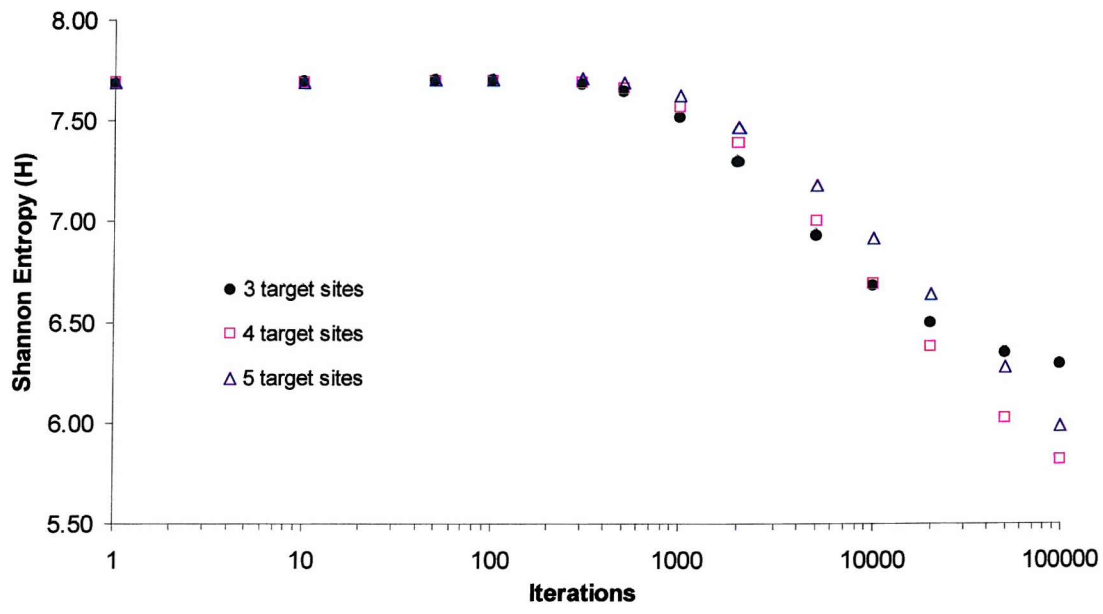


Figure 54 Variation of Shannon entropy with iterations showing the effect of the number of target sites used; 10000 insertions into HTLV II; number of target sites given in figure; with 3 target sites – TATA, TCAG, GGAC site selection probabilities are 1:1:1 respectively; with 4 target sites – attc, TATA, TCAG, GGAC site selection probabilities are 1:1:1:1 respectively; with 5 target sites – cgtg, attc, TATA, TCAG, GGAC site selection probabilities are 1:1:1:1:1 respectively; copy length 6; analysed with word length 4

The investigation into the effect of adding target sites was also performed with bias towards certain sites. These results showed only small changes in the Zipf plots, fluctuation exponents and Shannon entropy data as the number of target sites used was increased.

5.5 *THE EFFECT OF TARGET SITE LENGTH*

The simulation was run to ascertain the effect of changing the target site lengths. Up to this point the simulation had been run with target sites TATA, TCAG and GGAC all of length 4bp.

The first investigation was attempted using 3 target sites of length 8bp, but this failed to run as the starting sequence did not contain any of the sites. On average an 8bp sequence will occur every 4^8=65,536 bases (the starting sequence HTLV II was 8,952bp in length). The simulation was re-run using 3 target sites of length 6bp; the sequences TATAAT, TCAGCG and GGACTG. A copy length of 6 was used for each target site.

The results of the Zipf analysis are shown in Figure 55. The plots are striking due to the obvious presence of a large plateau at low ranks. This can be explained by the fact that the copy length is identical to the target site length. There are only three sequences that are duplicated - namely the three target sites TATAAT, TCAGCG and GGACTG. When the target site is length 4bp and the copy length is 6, there are 16 possible sequences duplicated for each target site eg. at target site TATA the sequence duplicated is TATAxx where x is any base - there are 4^2=16 combinations of xx. The lack of variety in the sequences copied when the copy length is the same as the target site length leads to a few words being highly probable in the final sequence, leading to the plateau effect. Examination of the words forming the plateau shows they are ranks 1-18 and are entirely derived from the target site sequences (6 words from the 6 bases of each of the three target sites).

The fluctuation analysis for the 6bp target site simulation is shown in Figure 56. Of note is the larger-than-usual depression in the line in the range of window lengths 2-10.

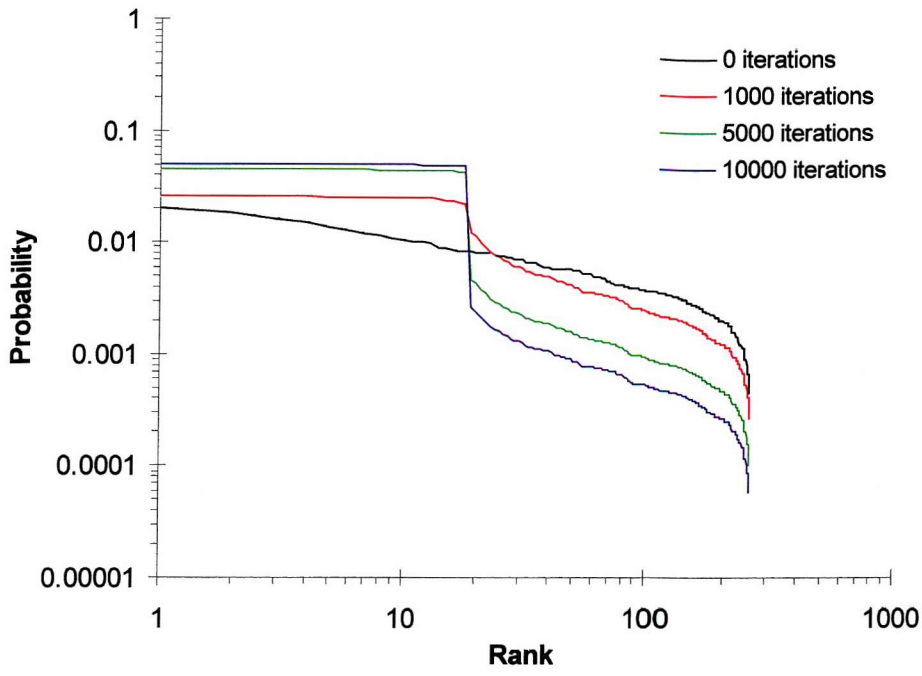


Figure 55 Zipf plots showing effect of target site length; insertion into HTLV II; target sites TATAAT, TCAGCG, GGACTG; site selection probabilities 1:1:1 respectively; copy length 6; analysed with word length 4

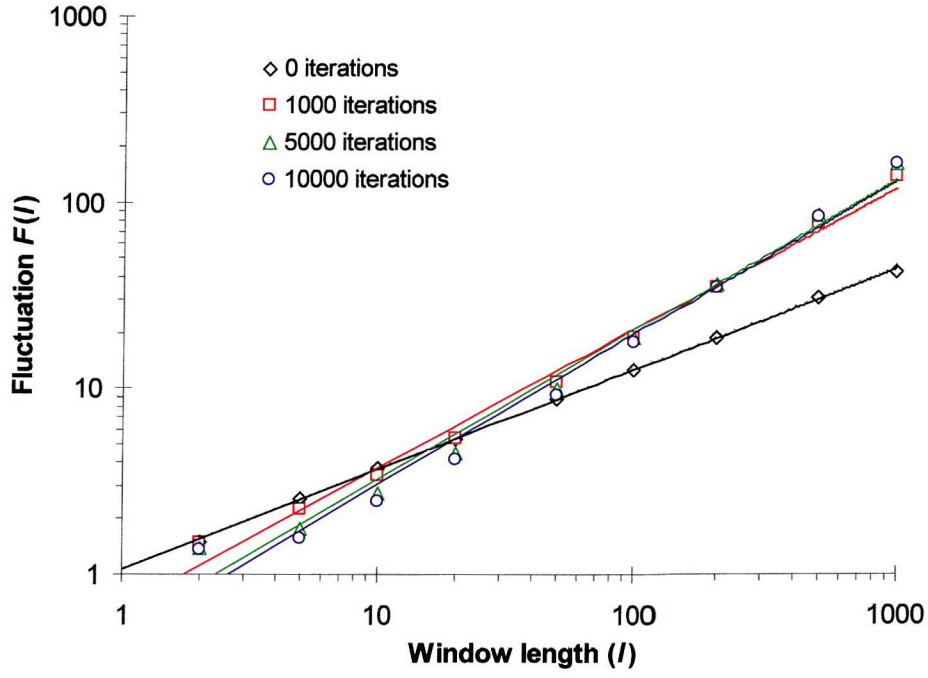


Figure 56 Fluctuation analysis showing effect of target site length; insertion into HTLV II; target sites TATAAT, TCAGCG, GGACTG; site selection probabilities 1:1:1 respectively; copy length 6

The simulation was also run with target sites of length 2bp. The Zipf data are given in Figure 57. Again, the result is striking with the curves appearing very flat, and changing little from the Zipf analysis of the starting sequence. The target sites occur much more frequently in the starting sequence and only 2 of the 6 bases that are copied are set due to the target site selected – for each target site duplication event there are 4^4=256 possible sequences that could be duplicated (e.g. duplication at the TA site leads to the insertion of TAxxxx where x is any base – there are 256 possible sequences that may be inserted). As a result a wider range of words are enriched in the sequence and the gradient of the Zipf plot is low. Table 10 shows the number of target sites in the HTLV II starting sequence depending on the length of the target site.

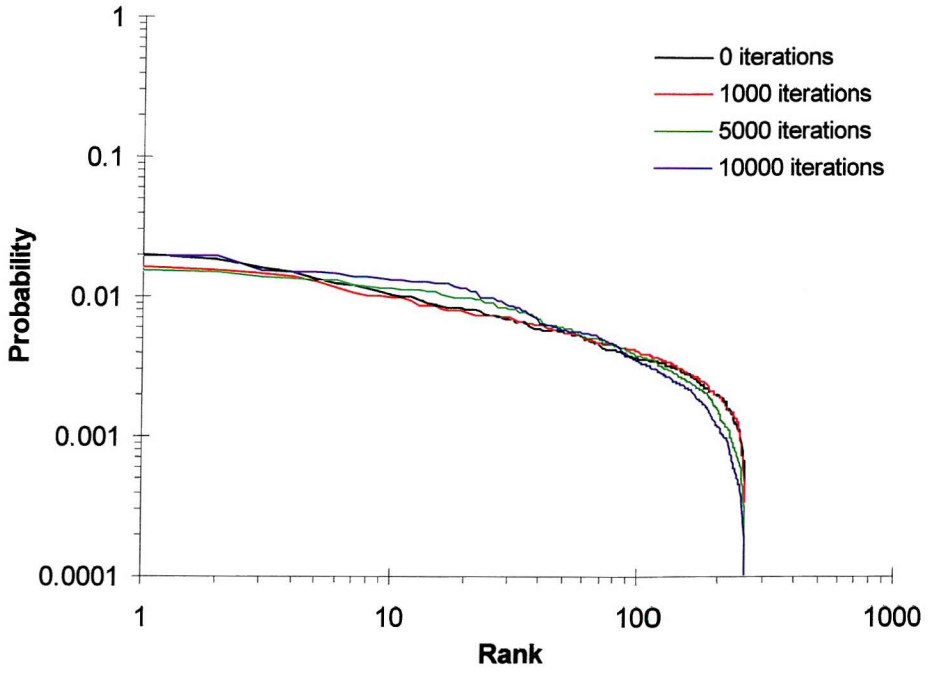


Figure 57 Zipf plots showing effect of target site length; insertion into HTLV II; target sites TA, TC, GG; site selection probabilities 1:1:1 respectively; copy length 6; analysed with word length 4

Target Site Length	Number of Site A	Proportion of Site A	Number of Site B	Proportion of Site B	Number of Site C	Proportion of Site C	Total Number of Sites
2	401	0.38	741	0.36	386	0.27	101
4	22	0.35	34	0.35	42	0.30	111
6	1	0.23	2	0.31	4	0.45	201

Table 10 The effect of target site length on the number of sites found in the host sequence. The exact target site sequence denoted by site A, site B and site C depends on the target site length. Site A: TA, TATA or TATAAT; Site B: TC, TCAG or TCAGCG; Site C: GG, GGAC or GGACTG

The fluctuation results for the run made with target site length 2 are given in Figure 58. The depression in the line seen with earlier fluctuation plots is greatly reduced. The likely reason for this is that there are fewer repetitive blocks in the sequence – it is these blocks that seem to cause the depression, as described previously.

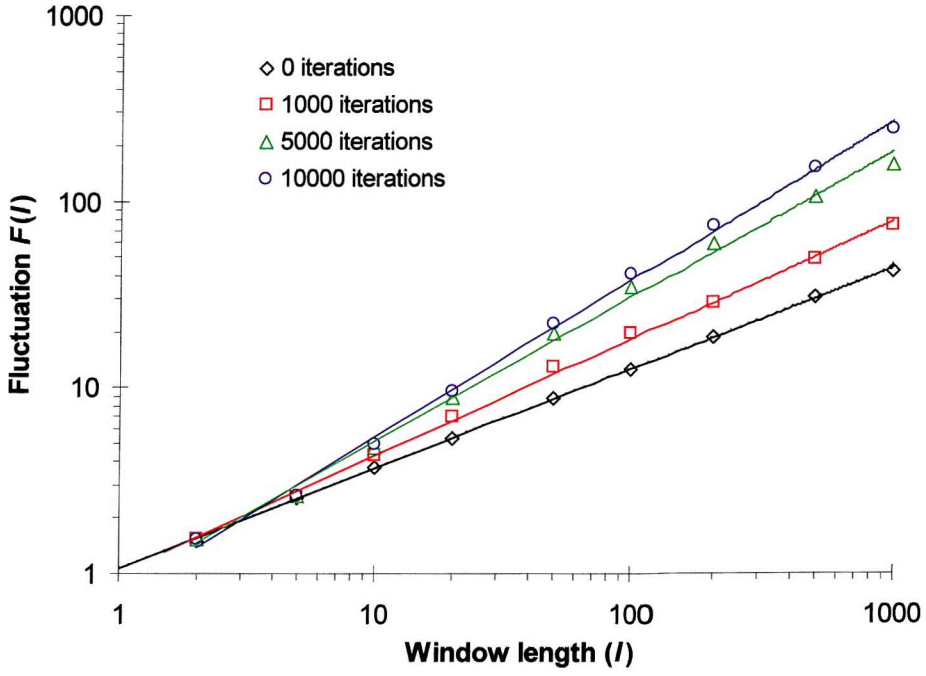


Figure 58 Fluctuation analysis showing effect of target site length; insertion into HTLV II; target sites TA, TC, GG; site selection probabilities 1:1:1 respectively; copy length 6

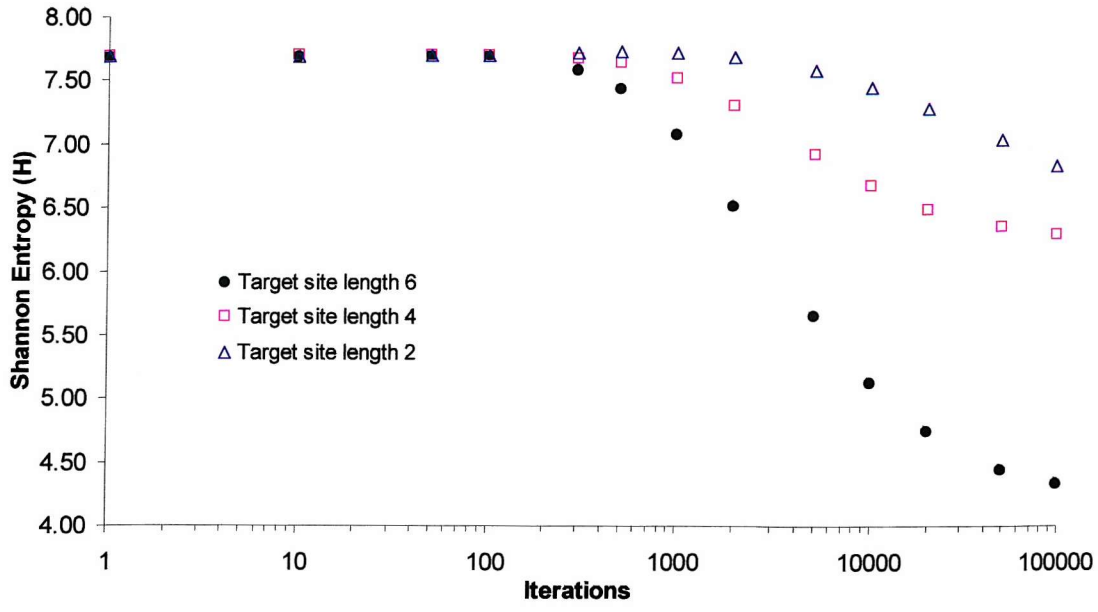


Figure 59 Variation of Shannon Entropy with iterations showing effect of target site length; insertion into HTLV II; target sites - length 6: TATAAT, TCAGCG, GGACTG; length 4: TATA, TCAG, GGAC; length 2: TA, TC, GG; site selection probabilities 1:1:1 respectively; copy length 6; analysed with word length 4

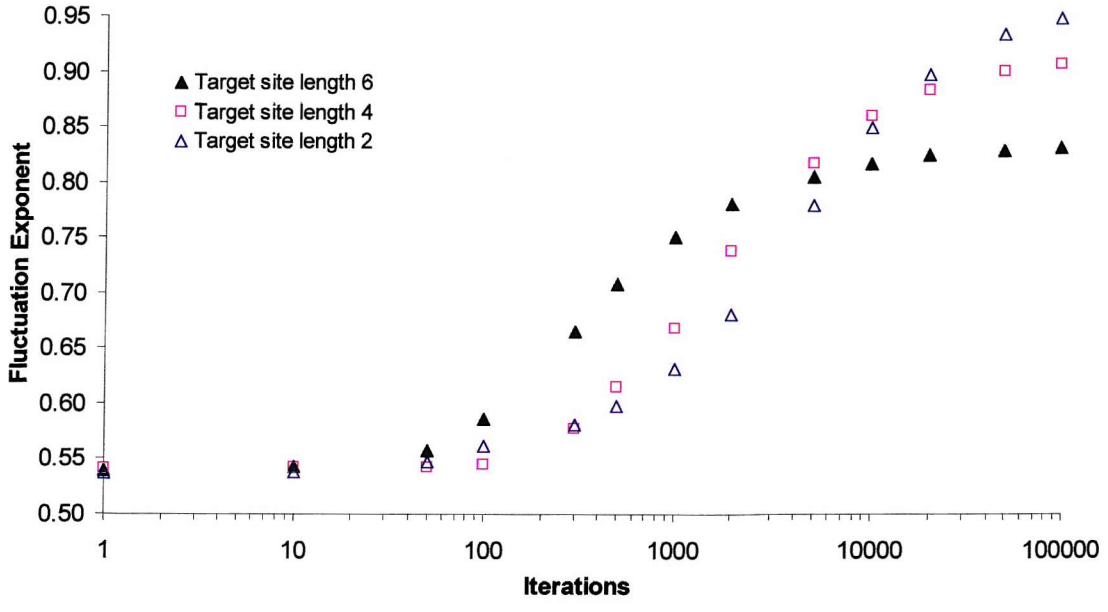


Figure 60 Variation of fluctuation exponent with iterations showing effect of target site length; insertion into HTLV II; target sites - length 6: TATAAT, TCAGCG, GGACTG; length 4: TATA, TCAG, GGAC; length 2: TA, TC, GG; site selection probabilities 1:1:1 respectively; copy length 6

Figure 59 shows the variation of Shannon entropy with iterations for sequences obtained using target sites of length 2, 4 and 6. The results show that the shorter the target site length used the lower the reduction in entropy caused by the simulation (although the simulation would have to be run to higher iterations to confirm this, particularly for the results obtained with target site length 2 as the line has not started to flatten out at 100,000 iterations). It appears that using a shorter target site also delays the reduction until higher iterations - the line obtained using target site length 2 stays at the entropy level of the starting sequence until a greater number of iterations have been performed before dropping away.

Figure 60 gives the variation of fluctuation exponent with iterations for the same sequences. These results show that the shorter the target site length used the higher the overall final fluctuation exponent. Like the Shannon entropy results, the shorter the target site length used, the greater the number of iterations that need to be performed before the fluctuation exponent starts to change from the value of the starting sequence.

As a final investigation into the effect of target site length on the statistical features of the DNA, a simulation was performed using the bases A, T, G and C as target sites i.e. every base in the sequence was a target site. Copy length 6 was used. The results of the Zipf analysis are given in Figure 61. The Zipf plots show a steady decrease in the Zipf exponent as the number of iterations increases - the sequence is being randomised by the action of the simulation. This result is interesting because all the bases added to the sequence come from the starting sequence itself, so it might be expected that a word that is particularly common in the starting sequence should be just as common in the final sequence. This does not seem to be the case - the lessening of the gradient shows that the probabilities of each word in the starting sequence are being equalised. A possible explanation is that the simulation acts to break up

particularly common words as they are more likely to be the target of an insertion event.

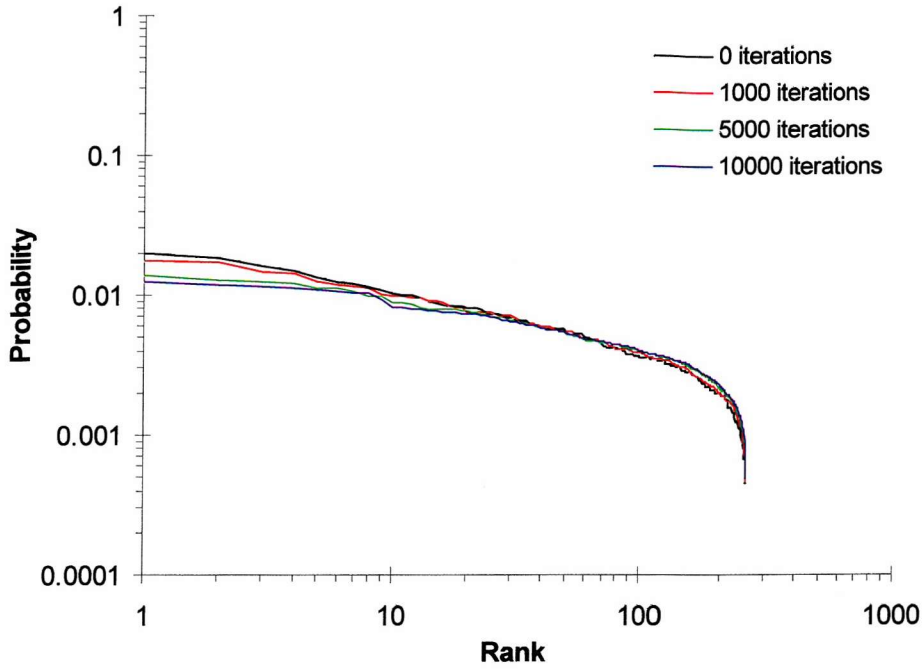


Figure 61 Zipf plots showing effect of target site length; insertion into HTLV II; target sites T, A, G, C; site selection probabilities 1:1:1:1 respectively; copy length 6; analysed with word length 4

The fluctuation analysis of the same sequences is shown in Figure 62. As with the results obtained using target sites of length 2bp, the plot is notable in that the depression usually seen at low window lengths is much reduced. Again, this is almost certainly due to the fact that there are fewer repetitive blocks in the final sequence. After 10,000 iterations the fluctuation exponent, α, was measured at 0.77.

Figure 63 shows the variation in Shannon entropy with increasing iterations. Unlike other Shannon entropy results the plot shows a gradually increasing level of Shannon entropy. As the sequence becomes randomised by the action of the simulation it becomes less ordered, so the Shannon entropy increases.

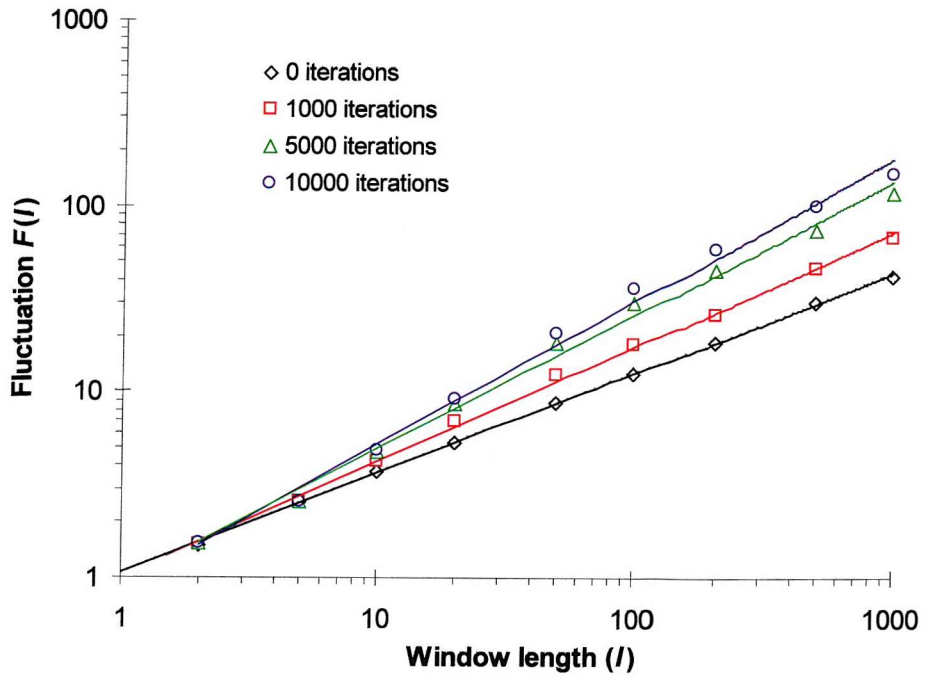


Figure 62 Fluctuation analysis showing effect of target site length; insertion into HTLV II; target sites T, A, G, C; site selection probabilities 1:1:1:1 respectively; copy length 6

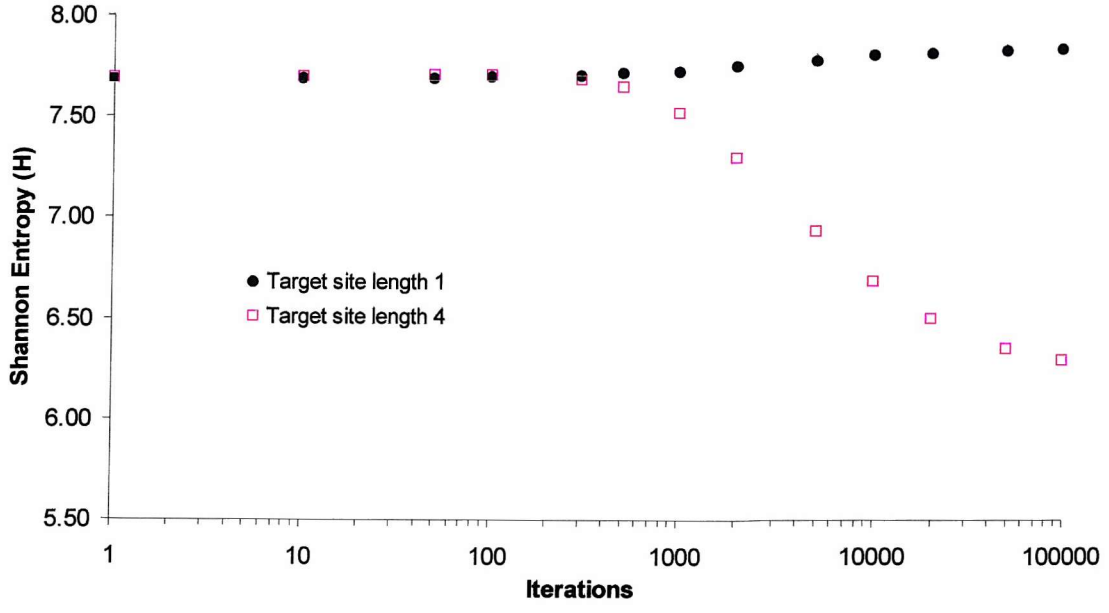


Figure 63 Variation of Shannon Entropy with iterations showing effect of target site length; insertion into HTLV II; for target site length 1 – target sites T, A, G, C; site selection probabilities 1:1:1:1 respectively; for target site length 4 – target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1; copy length 6; analysed with word length 4

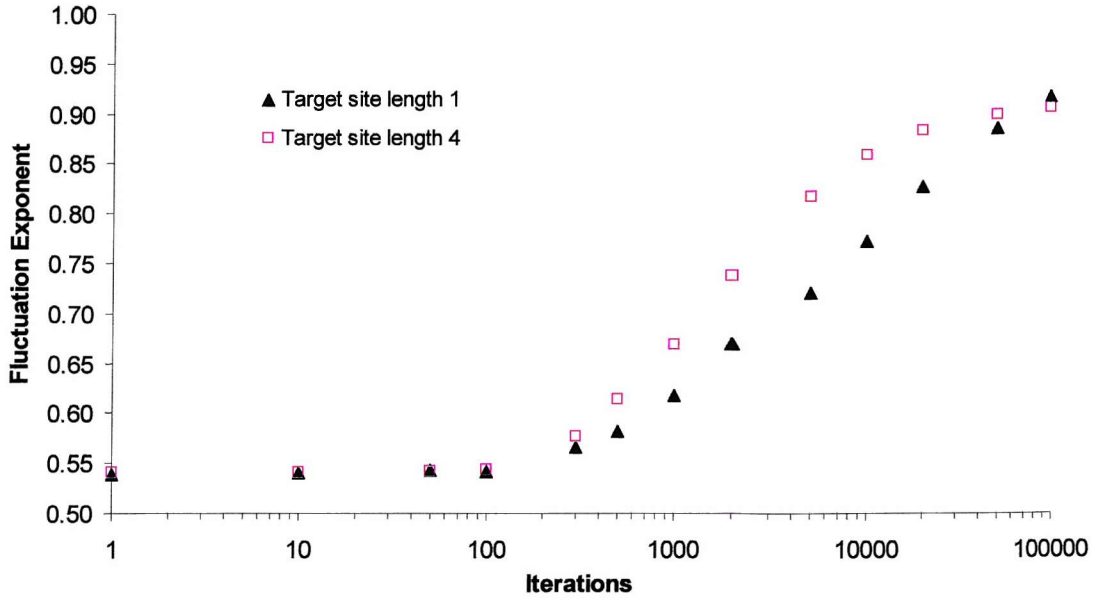


Figure 64 Variation of fluctuation exponent with iterations showing effect of target site length; insertion into HTLV II; for target site length 1 – target sites T, A, G, C; site selection probabilities 1:1:1:1 respectively; for target site length 4 – target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1; copy length 6

The variation of the fluctuation exponent with iterations is shown in Figure 64. Here the fluctuation exponent seems to steadily rise, with no apparent levelling off up to 100,000 iterations.

The results for the variation of Shannon entropy and fluctuation exponent with iterations for 1bp target sites are continuations of the trends seen when using target sites of 6bp, 4bp and 2bp length. As the target site length is reduced the Shannon entropy drops off later (in the case of the 1bp target sites not at all) and levels off at a higher value; the fluctuation exponent starts to increase at higher iterations and levels off at a higher final value.

5.6 *THE EFFECT OF COPY LENGTH*

Previous results showed the effect of using different copy lengths for each target site, but the overall effect of a short or long copy length had not been investigated. Simulations were run with the three target sites TATA, TCAG and

GGAC with copy lengths 20 and 50 to see if this had any effect on the results. Because using a longer copy length results in a longer final sequence and due to limitations of the program in handling such longer sequences, the simulations could only be run to 20,000 iterations rather than 100,000.

Figure 65 shows the Zipf analysis results for the two simulations run to 10,000 iterations with copy lengths 20 and 50 compared with the data obtained with copy length 6. It can be seen that the Zipf exponent appears lower the longer the copy length. Measurements of the Zipf exponent show that copy length 20bp gives a value of 0.43 and copy length 50bp gives 0.30. This compares with 0.34 for the starting sequence and 0.53 for the sequence with copy length 6. The reason for the higher Zipf exponent with shorter copy length is due to the number of words that are duplicated with each insertion event. When a copy length of 6bp is used, each insertion event generates a relatively small number of new words, so the frequencies of only a small number of words are increased in the final sequence. This will lead to a higher gradient on the Zipf plot. With a longer copy length, e.g. 50bp, a larger number of new words are obtained with each duplication so a large number of words are enriched in the sequence, leading to a lower Zipf exponent.

Figure 66 shows the fluctuation analysis for the sequence generated using copy length 20bp, while Figure 67 shows the fluctuation analysis using copy length 50bp. Of note in the two plots is the depression seen in the lines - this can be seen more clearly by examining the residuals of the two plots, given in Figure 68 and Figure 69. The residuals clearly show that the maximum depression in the fluctuation lines occurs around window length 20 when copy length 20 is used and window length 50 when copy length 50 is used. This fits the hypothesis presented previously that repetitive sequences would create a depression at the same window length as the copy length used in the simulation.

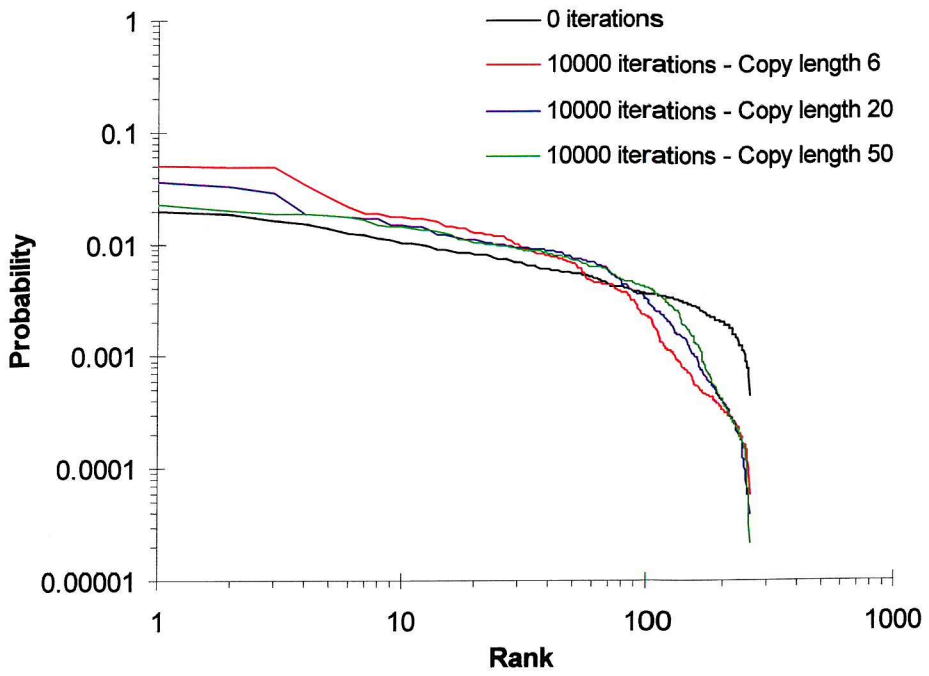


Figure 65 Zipf plots showing effect of copy length; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy lengths given in figure; analysed with word length 4

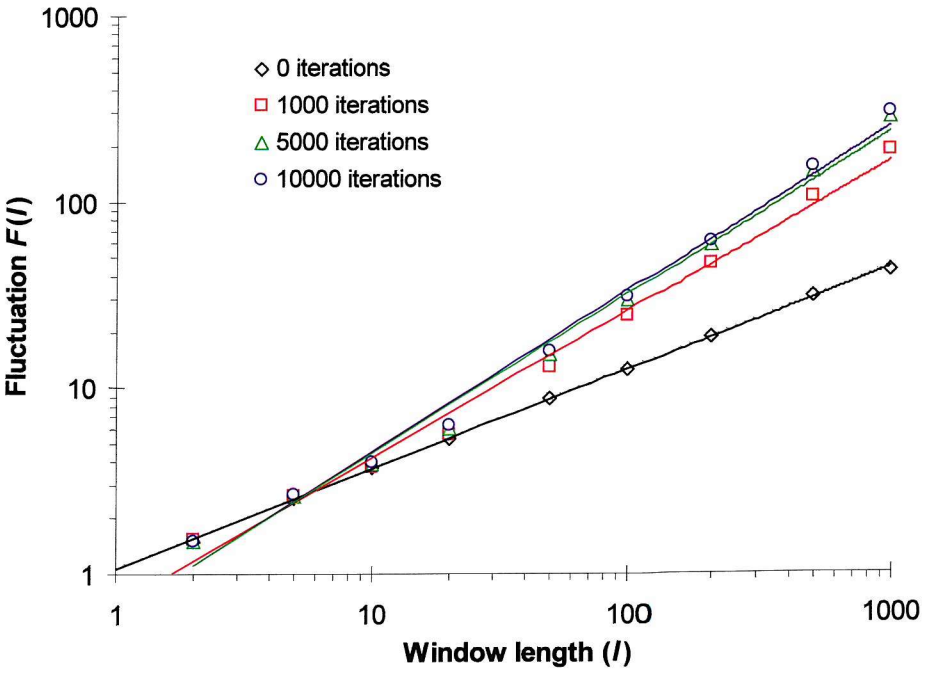


Figure 66 Fluctuation analysis showing effect of copy length; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 20

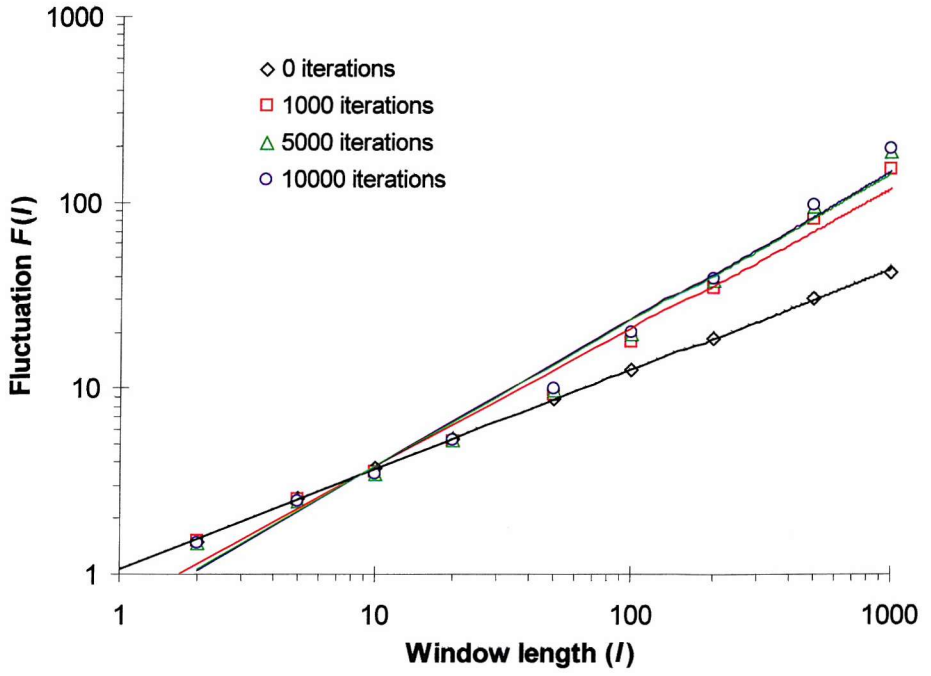


Figure 67 Fluctuation analysis showing effect of copy length; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 50

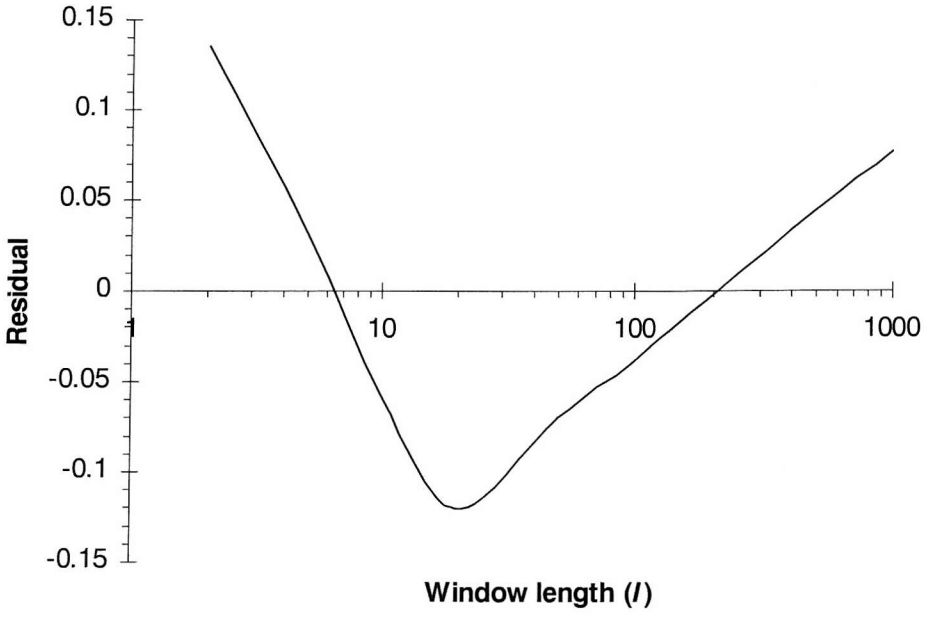


Figure 68 The residual of the fluctuation plot showing the effect of copy length. 10,000 insertions into HTL VII; target sites tata, tcag, ggac; site selection probabilities 1:1:1 respectively; copy length 20

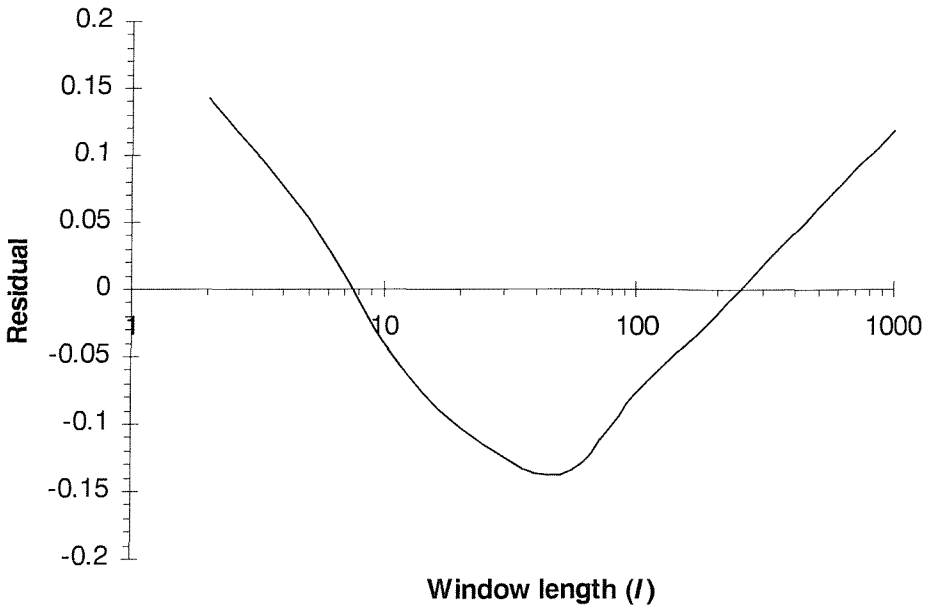


Figure 69 The residual of the fluctuation plot showing the effect of copy length. 10,000 insertions into HTL VII; target sites tata, tcag, ggac; site selection probabilities 1:1:1 respectively; copy length 50

The effect of iterations on Shannon entropy and fluctuation exponent values for the two simulations are given in Figure 70 and Figure 71. In line with the Zipf plots seen above, the Shannon entropy chart shows that the longer the copy length used, the higher the value of the Shannon entropy after the same number of iterations. When a short copy length is used there are a limited number of sequences that can be enhanced due to duplication e.g. using copy length 6 and target site length 4 there are $4^{(6-4)}=16$ possible sequences that may be enriched due to insertion at any one target site type. Using a copy length of 50 and target site length 4 there are $4^{(50-4)}=4.95\text{x}10^{27}$ possible sequences that may be duplicated. The end result is that using a short copy length results in a sequence where a relatively small number of words are very common. In this type of sequence there is a high degree of redundancy, a high degree of order and the Shannon entropy is correspondingly low.

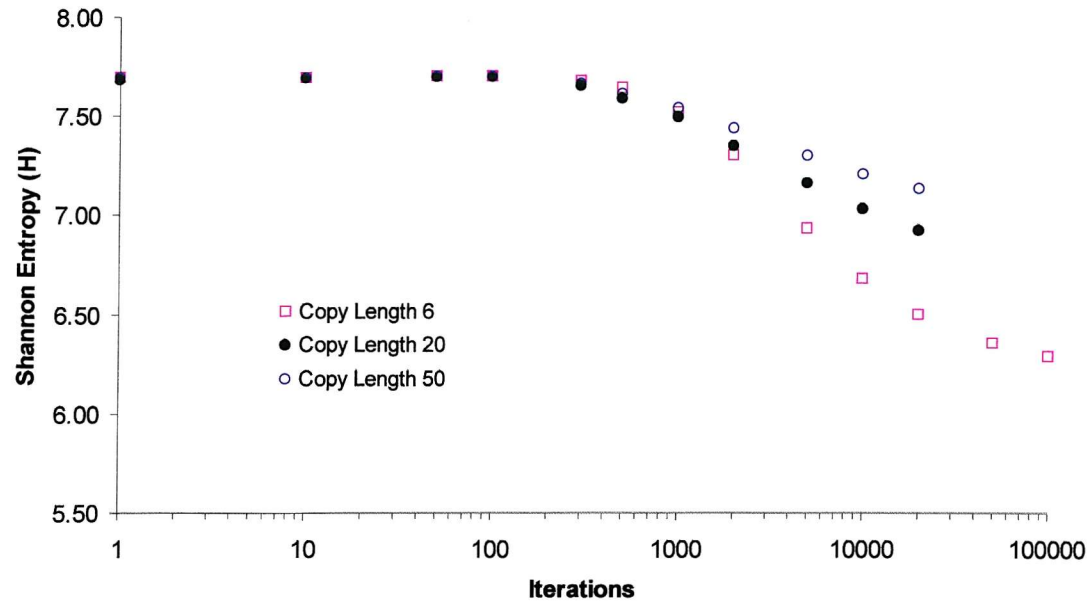


Figure 70 Variation of Shannon Entropy with iterations showing effect of copy length; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy lengths given in figure; analysed with word length 4

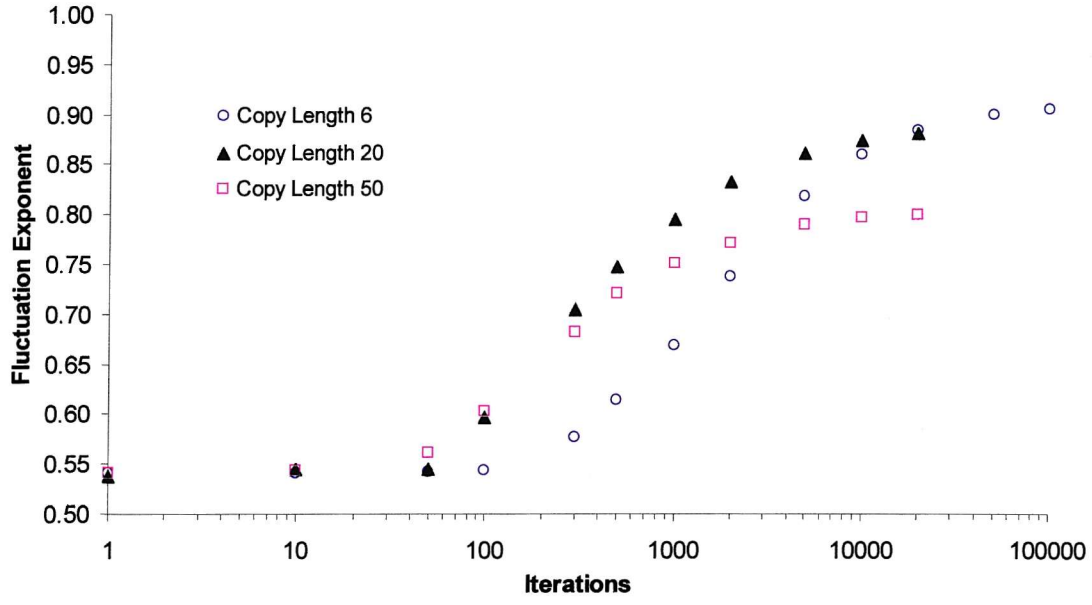


Figure 71 Variation of fluctuation exponent with iterations showing effect of copy length; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy lengths given in figure

The fluctuation results show that a longer copy length leads to the fluctuation exponent starting to increase at lower iterations, but reaching a lower final value. This may again be due to the fact that with a shorter copy length there are a smaller number of unique sequences which may be duplicated with each insertion event. There are identical blocks of repetitive sequences at many different points in the full sequence when a short copy length is used, but with a long copy length each block is more likely to be unique. The fact that blocks made up of the same repetitive sequence element are found throughout the final sequence may result in a higher fluctuation exponent.

5.7 THE FLUCTUATION ANOMALY

A primary aim of the modifications made to the simulation was to attempt to remove the depression seen in the fluctuation lines at around window length 5-10. It was earlier postulated that the anomaly was due to the copy length used in the simulation, and that the depression would be centred around the copy length used. This seems to have been confirmed by the investigations into the effect of copy length - increasing the copy length to 20bp and then to 50bp moved the anomaly to higher window lengths.

The other experiment which had a strong effect on the anomaly was the use of a different target site length. When target sites of 1bp or 2bp length were used, the depression in the line was greatly reduced. The explanation for the depression in the fluctuation line is that it occurs when a large number of identical sequences are generated from the sliding window used in the fluctuation analysis. To avoid this there needs to be more variety in the final sequence generated by the simulation. Using a short target site promotes this as the target site will occur in many different places in the starting sequence. There will be many short repetitive blocks such that when the sliding window is used a relatively large proportion of the generated sequences are from the edges of the blocks, incorporating bases from both the repetitive block and the non-repetitive

surrounding bases. Compare this with the result achieved if a long target site is used. The target site may occur only once in the host sequence so the action of the simulation will build up one long block of repetitive sequence. Most of the sequences generated by the fluctuation analysis sliding window will be the repetitive sequences from the interior of the block - relatively few will incorporate bases from the edges of the block, thus reducing the variety of sequences generated.

It is interesting to note that the aim of increasing the variety in the final sequence to improve the fluctuation data is set against the aim of biasing towards certain sequences in order to increase the Zipf exponent. Given that the Zipf exponents found earlier are high compared with those found for real non-coding DNA sequences and that visually the sequences appear blocky, it seemed that there was scope to increase the variety in the sequence while still maintaining a large enough bias to generate a suitable Zipf gradient. It should also be noted that the natural processes being replicated generate more sequence variety in their action than the corresponding simulated processes used in the model used up to this point. Tailoring the model to produce more "noise" in the system is, therefore, perfectly reasonable.

As previous results had showed that a short target site length and a long copy length could affect the depression seen in the fluctuation data, the simulation was run combining both to see what effect this would have. The simulation was performed with target sites TA, TC and GG using a copy length 20.

The Zipf results are given in Figure 72. The results are similar to those seen when using a 2bp target site with a 6bp copy length. The gradient appears to decrease slowly with iterations. The starting gradient is 0.34 and this falls gradually to 0.30 after 10,000 iterations.

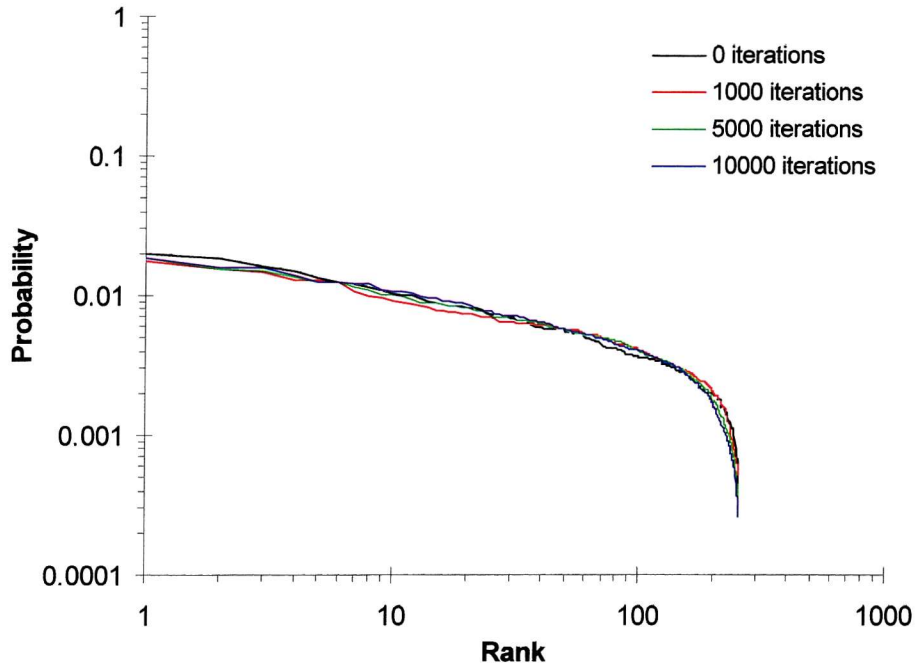


Figure 72 Zipf plots showing effect of short target sites and long copy length; insertion into HTLV II; target sites TA, TC, GG; site selection probabilities 1:1:1 respectively; copy length 20; analysed with word length 4

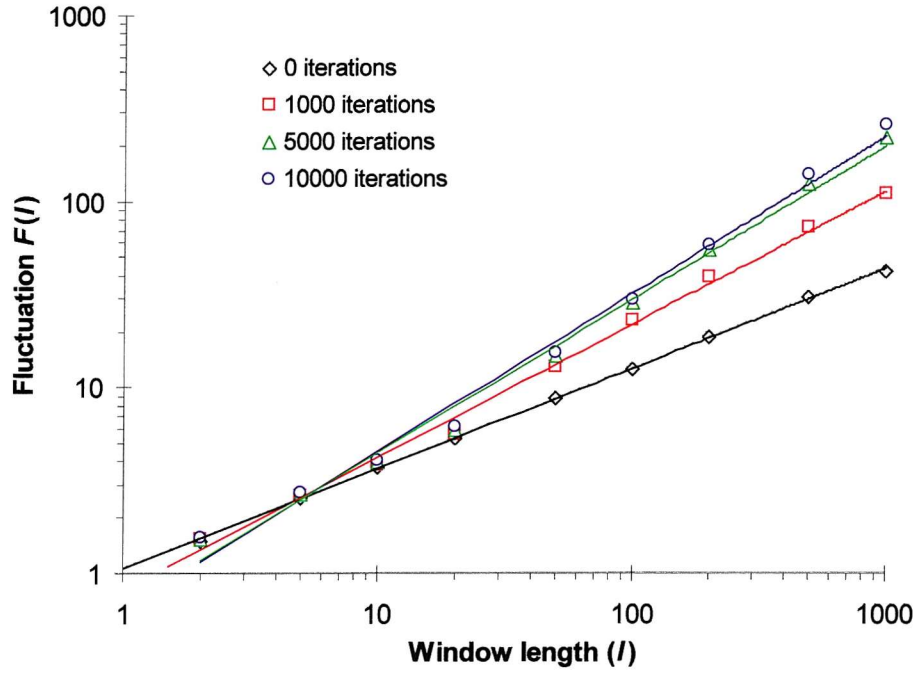


Figure 73 Fluctuation analysis showing effect of short target sites and long copy length; insertion into HTLV II; target sites TA, TC, GG; site selection probabilities 1:1:1 respectively; copy length 20

Figure 73 shows the fluctuation results for the same simulation. The plots appear very similar to those obtained using copy length 20. The depression is still present and appears centred around window length 20.

The previous results had shown that the copy length affected the fluctuation analysis. It was postulated that if a number of target sites were used, with copy lengths over a wide range, then the dip might be smoothed out to the extent that it was no longer present. The simulation was run with 4 target sites, each of length 4bp, and with copy lengths 4, 10, 33, and 100bp. Each target site was selected for duplication with the same probability as any other.

The Zipf data for the simulation using a range of copy lengths is given in Figure 74. Visually, the quality of the lines appears poor with a large plateau up to rank ≈50 and the first part of the line appearing quite curved. Taking the gradients of the lines shows the Zipf exponent to be lower for the modified sequences than for the starting sequence.

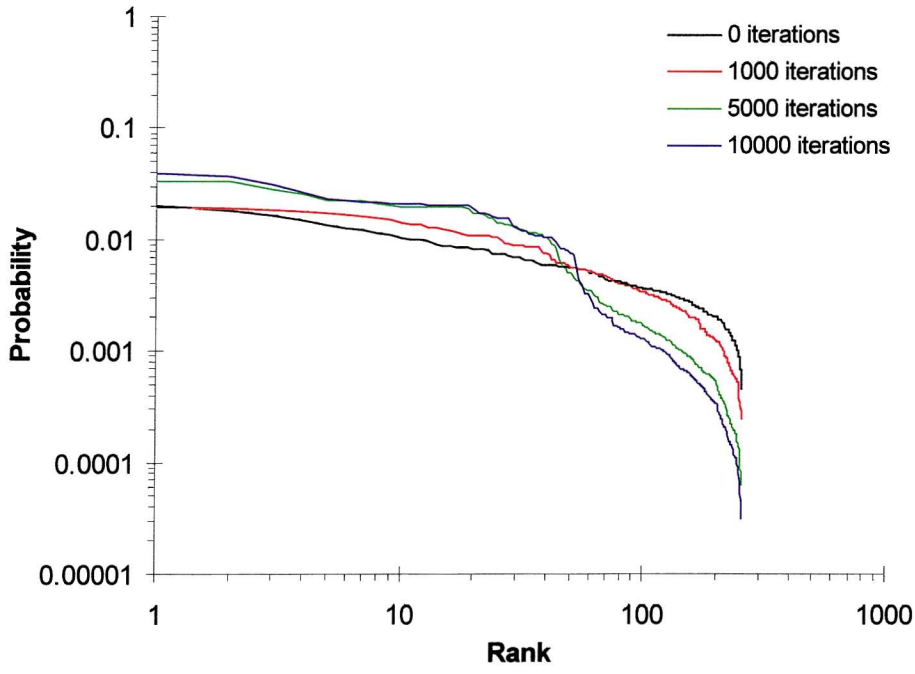


Figure 74 Zipf plots showing effect of using a wide range of copy lengths; insertion into HTLV II; target sites TATA, TCAG, GGAC, atgt; site selection probabilities 1:1:1:1 respectively; copy lengths 4, 10, 33 and 100 respectively; analysed with word length 4

The fluctuation results for the corresponding simulation are given in Figure 75. It can be seen that the anomaly in the fluctuation analysis is present over a wide range of the plot, as expected, but is still far too apparent for the points to form a straight line, particularly at longer window lengths. To try and rectify this the simulation was repeated, but with a bias introduced against the target sites with longer copy lengths, proportional to the copy length. The Zipf results of this run are shown in Figure 76 and the fluctuation results in Figure 77.

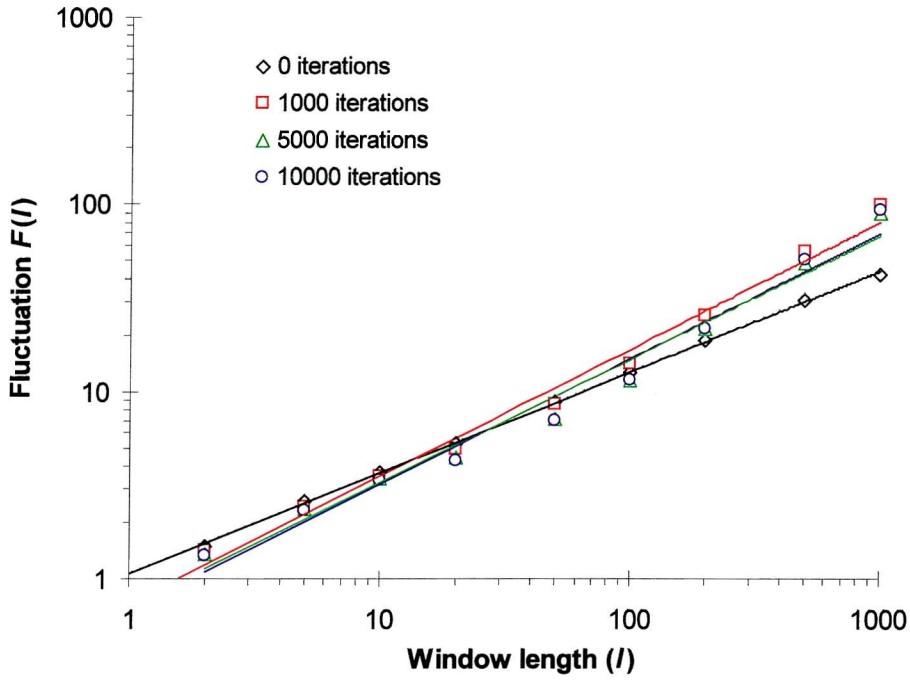


Figure 75 Fluctuation analysis showing effect of using a wide range of copy lengths; insertion into HTLV II; target sites TATA, TCAG, GGAC, ATGT; site selection probabilities 1:1:1:1 respectively; copy lengths 4, 10, 33 and 100 respectively

The Zipf results show a very large plateau formed by the two most common words. The rest of the curve closely follows that obtained with the starting sequence. Examination of the words generated shows that the two most common words are TATA and ATAT. These would be the two words formed in a repetitive block formed by insertion at a TATA site (with copy length 4 as used for the TATA target site in this simulation, the block would simply be made up of repeating TATA elements). The use of a different target site sequence as the most heavily biased (a repeating TATA sequence only produces two unique

words, another sequence could generate up to four eg. TCAG) or a slightly longer copy length would introduce more variety into the most common words obtained and should therefore reduce the plateau. Introducing a longer copy length would, however, lessen the range of copy lengths in use and changing the target site sequence would only produce two additional words which may not have a large effect on reducing the plateau.

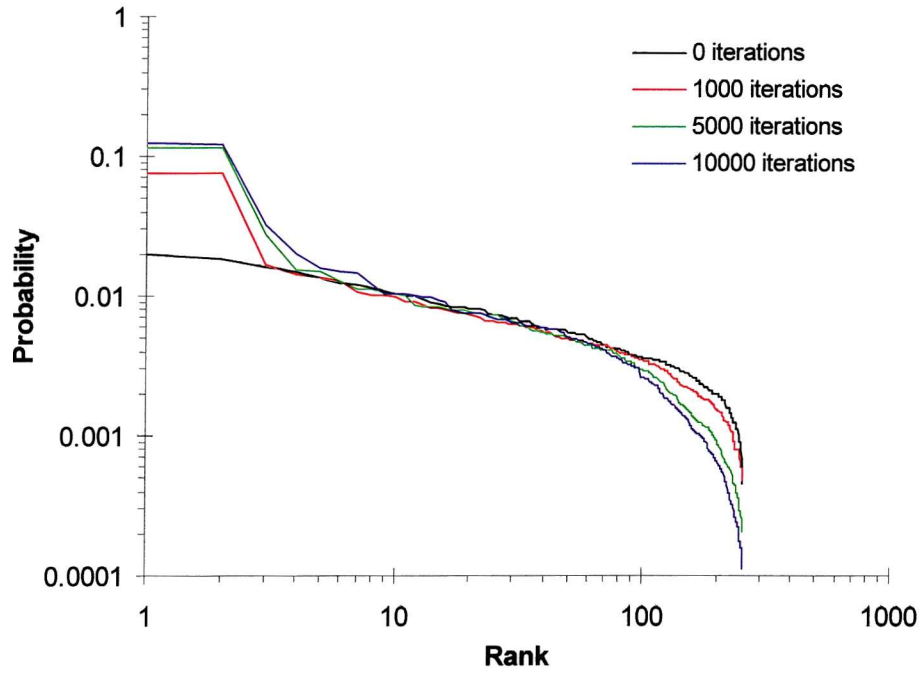


Figure 76 Zipf plots showing effect of using a wide range of copy lengths with bias against longer copy lengths; insertion into HTLV II; target sites TATA, TCAG, GGAC, ATGT; site selection probabilities 25:10:3:1 respectively; copy lengths 4, 10, 33 and 100 respectively; analysed with word length 4

The fluctuation results show that the depression in the fluctuation line has been reduced at high window lengths, but it is still clearly present around window length 10. The plot is actually quite similar to that achieved in the initial experiments using three target sites and copy length 6bp.

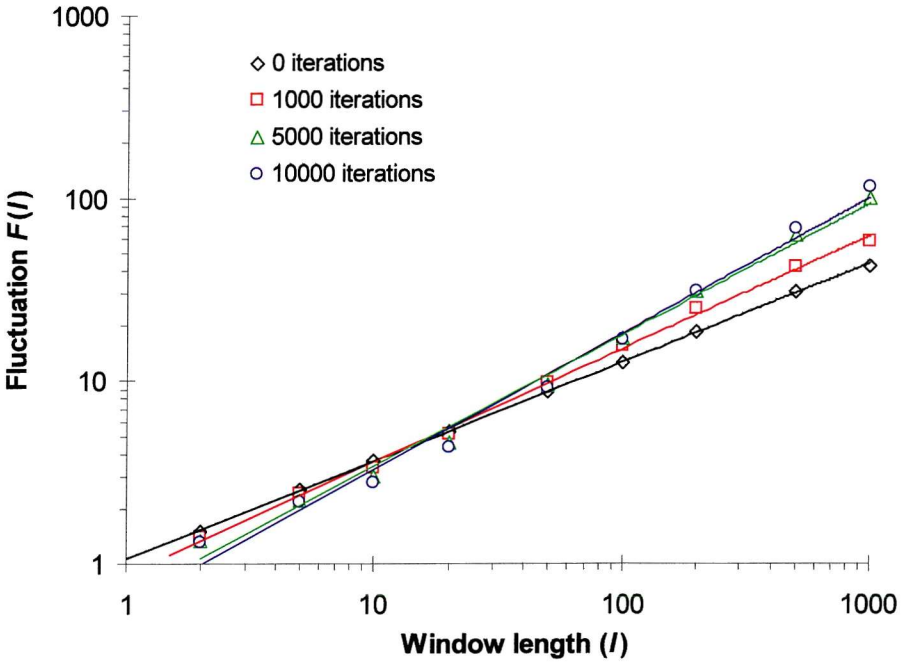


Figure 77 Fluctuation analysis showing effect of using a wide range of copy lengths with bias against longer copy lengths; insertion into HTLV II; target sites TATA, TCAG, GGAC, ATGT; site selection probabilities 25:10:3:1 respectively; copy lengths 4, 10, 33 and 100 respectively

5.8 *FURTHER MODIFICATIONS*

The results presented so far show that simply altering the copy lengths of target sites can affect the fluctuation data, but does not remove the dip in the line entirely. The copied sequences still form large blocks of perfectly repeated sequences that will always generate low values for $F(l)$ at the appropriate window length. It was felt that it would be possible to remove the dip in the fluctuation results by breaking up the perfect repeats generated by the existing model.

At this stage the model incorporated several features of transposable element insertions and excisions seen in the natural system, but still lacked some details of these events and did not include other genome reshaping processes. Two ways in which the simulation could be made more realistic are through the addition of point mutations and imperfect excision of transposable elements,

both described earlier. Consequently, with a view to making the simulation more complete and improving the quality of the results these processes were added to the simulation, and their effect on the artificial DNA produced was investigated.

6 Further Improvements to the Transposable Element Model

6.1 POINT MUTATIONS

As described earlier, point mutations involve the base at a given position being swapped for another base[7]. The mutation may either be a transition, in which a purine is swapped for another purine (or a pyrimidine for a pyrimidine); or a transversion, in which a purine is swapped for a pyrimidine (or vice versa). Transitions are more common that transversions.

The incorporation of point mutations leads to several new considerations, which had to be addressed in the program:

1) During a given iteration of the program, will a transposable element effect be modelled, or a point mutation?
2) At which base will a point mutation occur?
3) Will a point mutation be a transition or a transversion?
4) If the point mutation is a transversion, what will the new base be? (e.g. if A is to undergo a transversion, the new base could be either T or C – which should be chosen? NB if, for example, A is to undergo a transition, the new base must be G – there is no choice).

The program was modified, and the procedure is outlined below. As a result of the modifications two new parameters were added relating to point mutations that could be altered:

1) A ratio determining whether a point mutation or a transposable element effect would take place in a given iteration of the simulation.
2) A ratio determining whether each point mutation will be a transition or a transversion.

The following details were built into the structure of the program and were therefore not intended to be altered from run to run:

1) Point mutations could occur at any position in the sequence with equal probability. This is an over-simplification. In real systems there exist hot spots at which the rate of point mutation is much higher than the average.
2) When a transversion occurs, the new base will be chosen with equal probability from the two possibilities.

Following successful testing of the new program to ensure that the new features worked correctly, the simulation was run and the initial results are shown below.

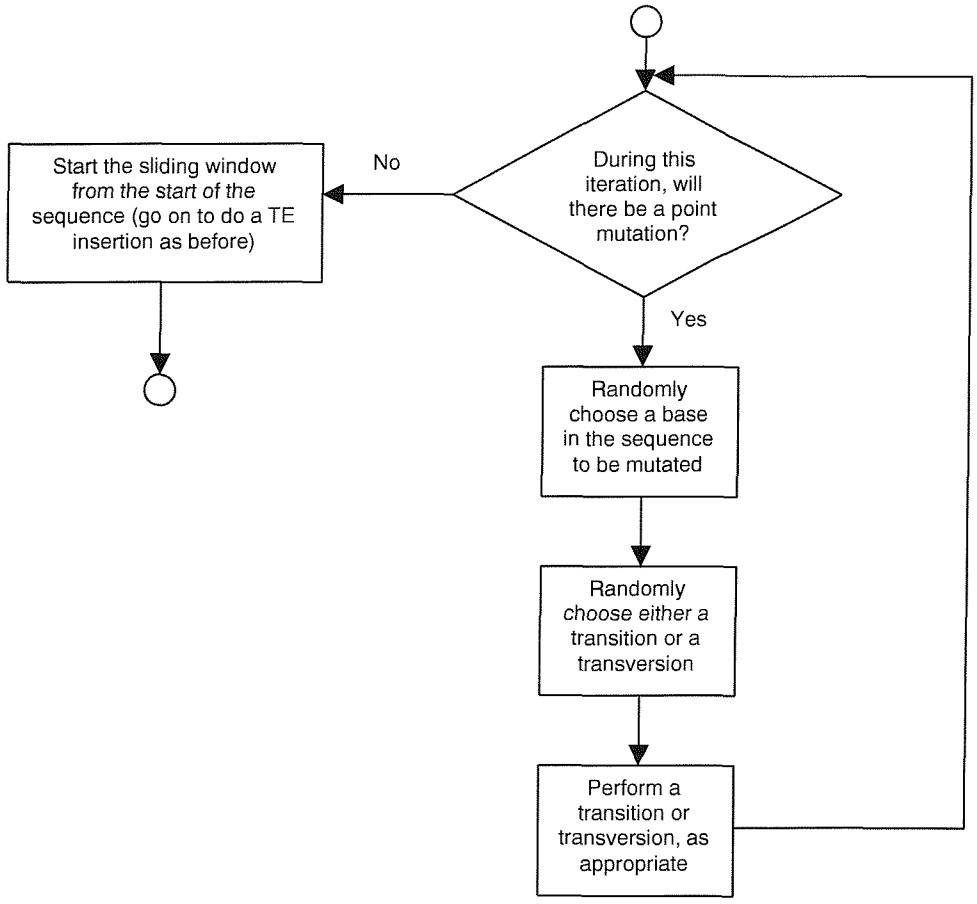


Figure 78 Flowchart illustrating the process used by the simulation incorporating point mutations. The structure of the program is the same as that presented earlier, so only the point mutation subroutine is shown here

6.2 TRANSITIONS VS TRANSVERSIONS

The first tests of the simulation were run entirely using point mutations with no transposable element insertion events modelled. The effect of altering the transition:transversion ratio was investigated.

Figure 79 shows the Zipf results for a simulation run with a transition:transversion ratio of 1:1. The curves clearly show the Zipf exponent decreasing as the number of iterations increases. The gradient after 10,000 iterations is 0.06, down from the starting gradient of 0.34. The action of the point mutations is to randomise the starting sequence, so that the bias towards the common words in the starting sequence is reduced.

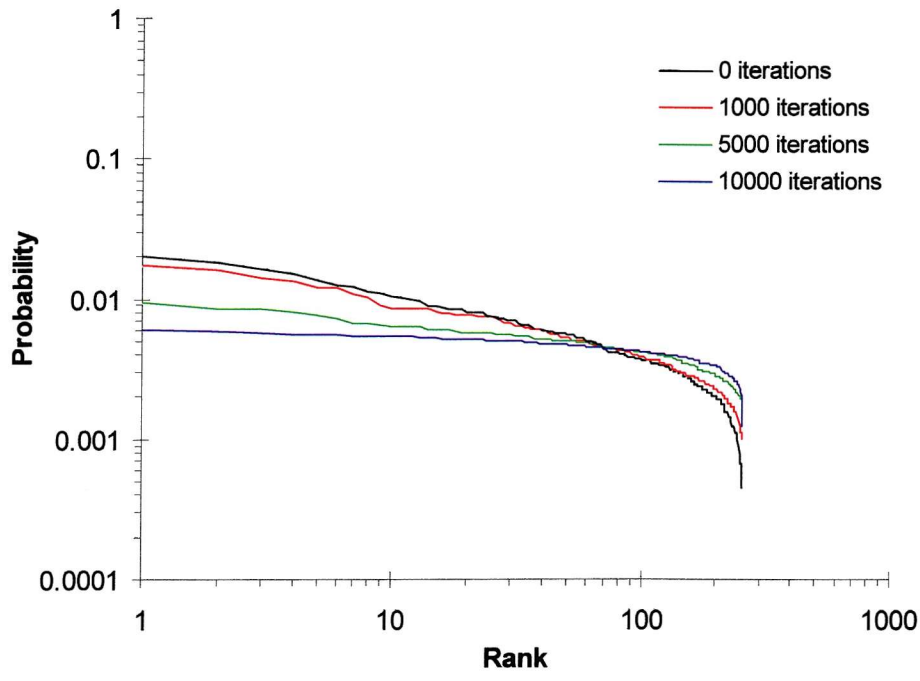


Figure 79 Zipf plots showing effect of point mutations; starting sequence HTLV II; ratio point mutations:insertion/excision events 1:0; transition:transversion ratio 1:1; analysed with word length 4

Figure 80 shows the fluctuation data obtained from the same run. Once again there is a reduction in the gradient of the fluctuation lines as the number of iterations increases. The starting sequence gives a fluctuation exponent of 0.54, while after 10,000 iterations the gradient is 0.48. Again, this can be attributed to

the randomising effect of the point mutations. The random DNA sequence used earlier gave a fluctuation exponent of 0.48.

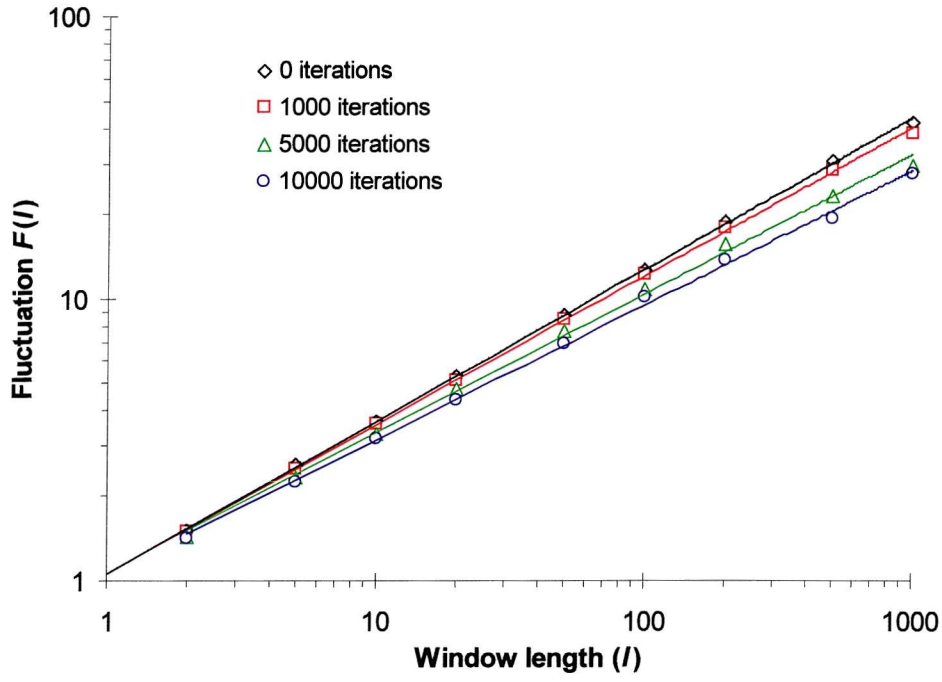


Figure 80 Fluctuation analysis showing effect of point mutations; starting sequence HTLV II; ratio point mutations:insertion/excision events 1:0; transition:transversion ratio 1:1

The effect of altering the transition:transversion ratio was investigated by running two simulations, the first with no transversions, and the second with no transitions.

The Zipf plots obtained with no transversions are given in Figure 81. It can be seen that like the previous point mutation results obtained with both transitions and transversions, the gradient decreases with iterations. The gradient after 10,000 iterations was calculated at 0.31. The reduction in gradient is small compared to that obtained when both transitions and transversions are used. This may be because when a transition is performed there is only one nucleotide that may be inserted in place of the original - with a transversion there is a choice of two possibilities. There is a greater degree of 'randomisation' that occurs when a series of transversions are performed and so they produce a sequence with a lower Zipf exponent.

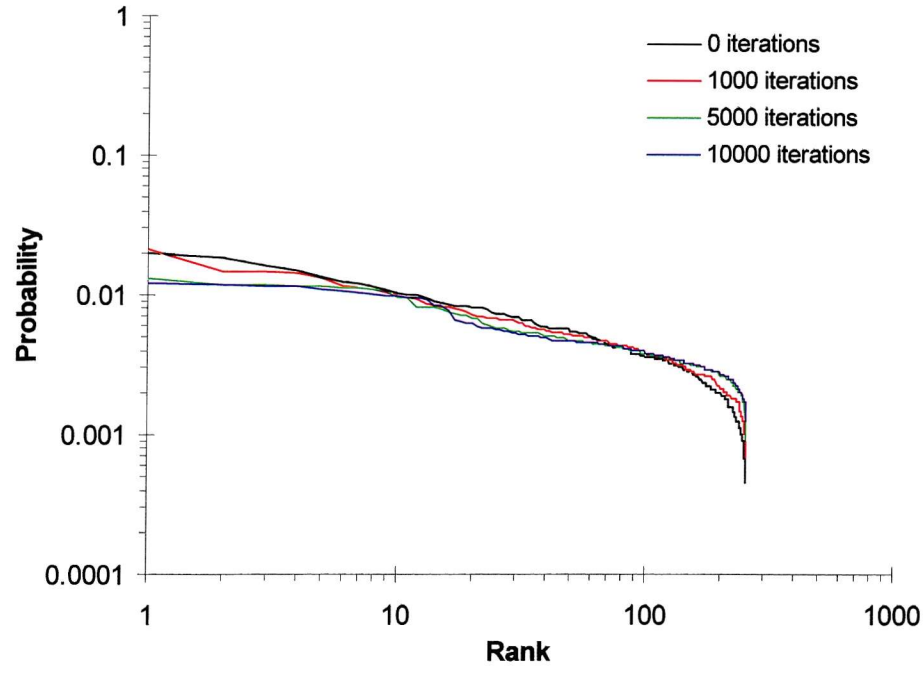


Figure 81 Zipf plots showing effect of point mutations; starting sequence HTLV II; ratio point mutations:insertion/excision events 1:0; transition:transversion ratio 1:0; analysed with word length 4

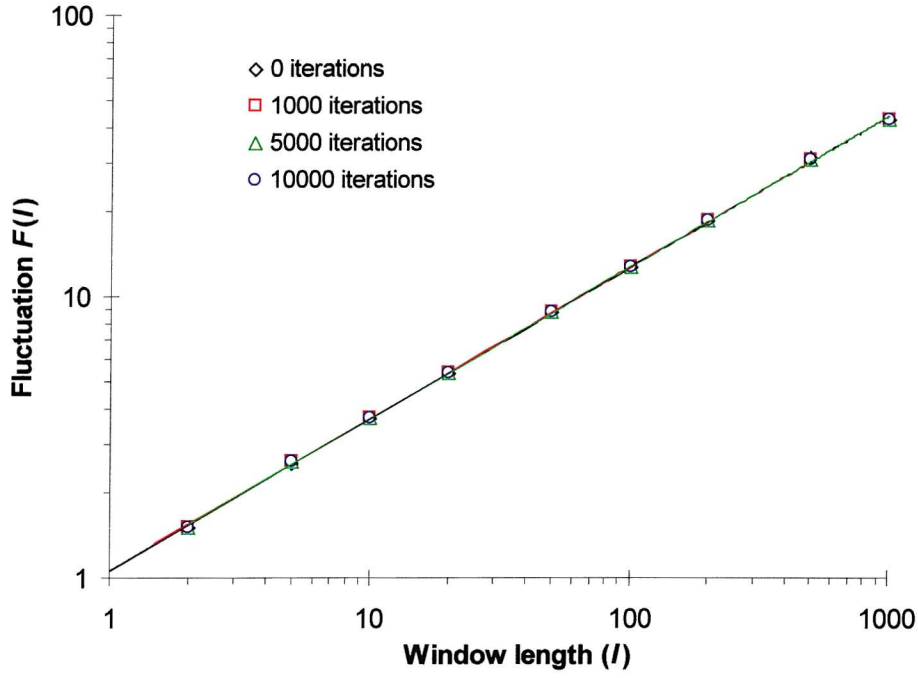


Figure 82 Fluctuation analysis showing effect of point mutations; starting sequence HTLV II; ratio point mutations:insertion/excision events 1:0; transition:transversion ratio 1:0

Figure 82 shows the fluctuation data obtained when point mutations are applied using transitions only. The gradient is unaltered – this occurs because the mechanism of the fluctuation analysis translates the DNA sequence into a numerical sequence based on whether a nucleotide is a purine or a pyrimidine. Transition point mutations do not alter the class (purine or pyrimidine) of a nucleotide, so do not change the fluctuation result from that obtained with the starting sequence.

The Zipf and fluctuation results for the simulation using transversions only are given in Figure 83 and Figure 84. The Zipf results appear very similar to those obtained when both transitions and transversions were used. The gradient after 10,000 iterations is 0.12.

The fluctuation data also show similarities to those obtained when using both transitions and transversions. The fluctuation exponent after 10,000 iterations is 0.51.

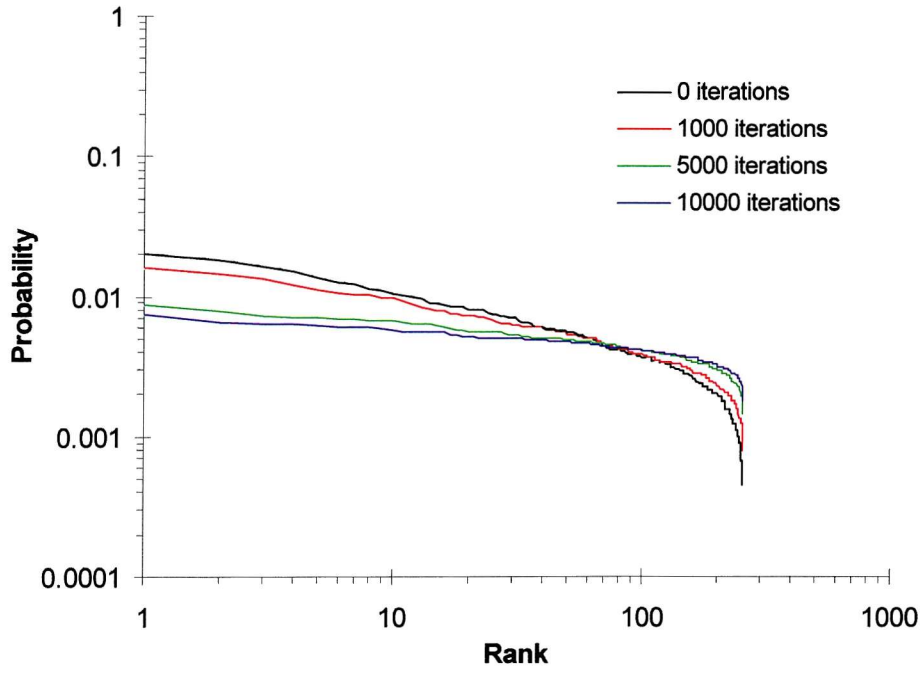


Figure 83 Zipf plots showing effect of point mutations; starting sequence HTLV II; ratio point mutations:insertion/excision events 1:0; transition:transversion ratio 0:1; analysed with word length 4

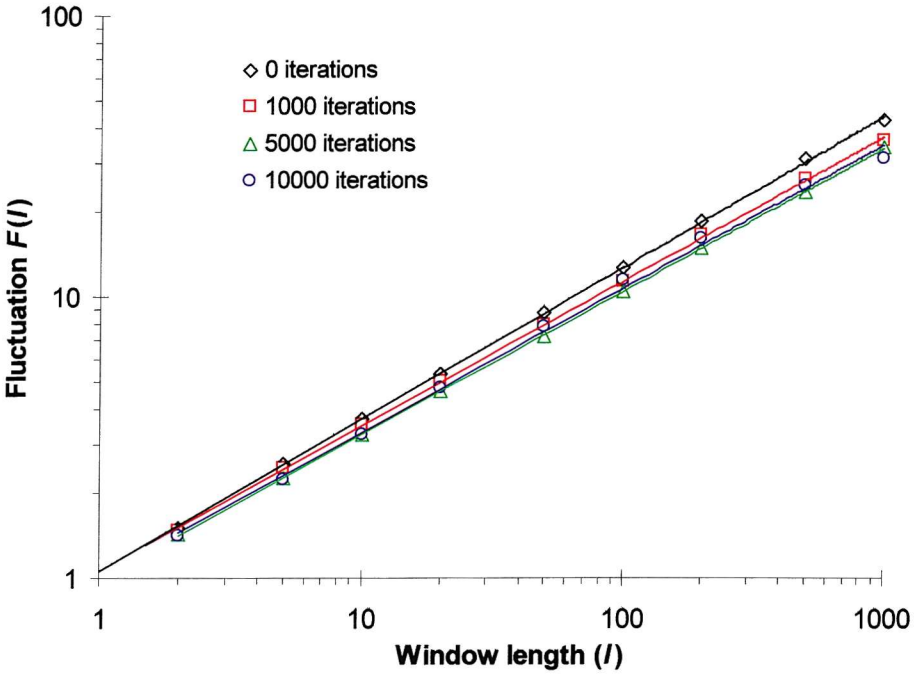


Figure 84 Fluctuation analysis showing effect of point mutations; starting sequence HTLV II; ratio point mutations:insertion/excision events 1:0; transition:transversion ratio 0:1

The results show that of the two types of point mutation, transversions seem to have the greater effect on both the Zipf and fluctuation results. Their effect is to lower both the Zipf and fluctuation exponents. Following these experiments into the results obtained by point mutations without transposable element insertion and excision, the simulation was run incorporating both systems.

6.3 *THE EFFECT OF POINT MUTATION AND TRANSPOSABLE ELEMENT INSERTIONS*

The simulation was performed incorporating both point mutations and transposable element mutations at a 1:1 ratio. The insertion/excision events were modelled using three target sites each with a copy length 6 and equal chance of selection. The point mutations were performed with transitions twice

as likely as transversions. These values correspond with the values seen in natural systems[7, 44].

It should be noted that in the new program one mutational event (transposable element insertion/excision or point mutation) occurs with each iteration. If the ratio point mutation:transposable element effect is 1:1 then the number of iterations must be doubled to get (on average) the same number of transposable element insertion/excisions as in previous runs that only simulated transposable element effects. A run of 100,000 iterations with the new program is not directly comparable to a run of 100,000 iterations with the previous program as roughly half of the iterations in the new simulation will result in point mutations rather than transposable element insertion/excisions.

Figure 85 shows the results of the Zipf analysis on sequences generated with point mutations. It can be seen that a 'step' is formed by the most common words, up to around rank 4. The remainder of the slope is very similar to that obtained with the starting sequence.

Figure 86 shows the results of the fluctuation analysis for sequences obtained using point mutations. The lines appear similar to those obtained without point mutations, although the gradient after 10,000 iterations is slightly lower with point mutations at 0.83 compared with 0.86 obtained without point mutations. This is due to the randomising effect seen earlier.

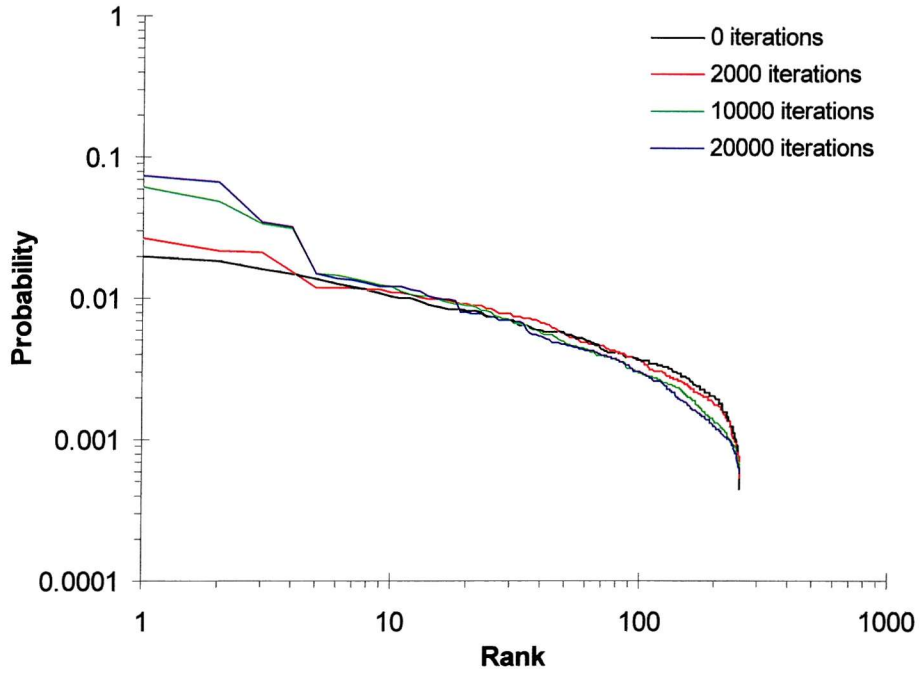


Figure 85 Zipf plots showing effect of point mutations; insertion into HTLV II; point mutations:insertion/excisions ratio 1:1; transition:transversion ratio 2:1; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; analysed with word length 4

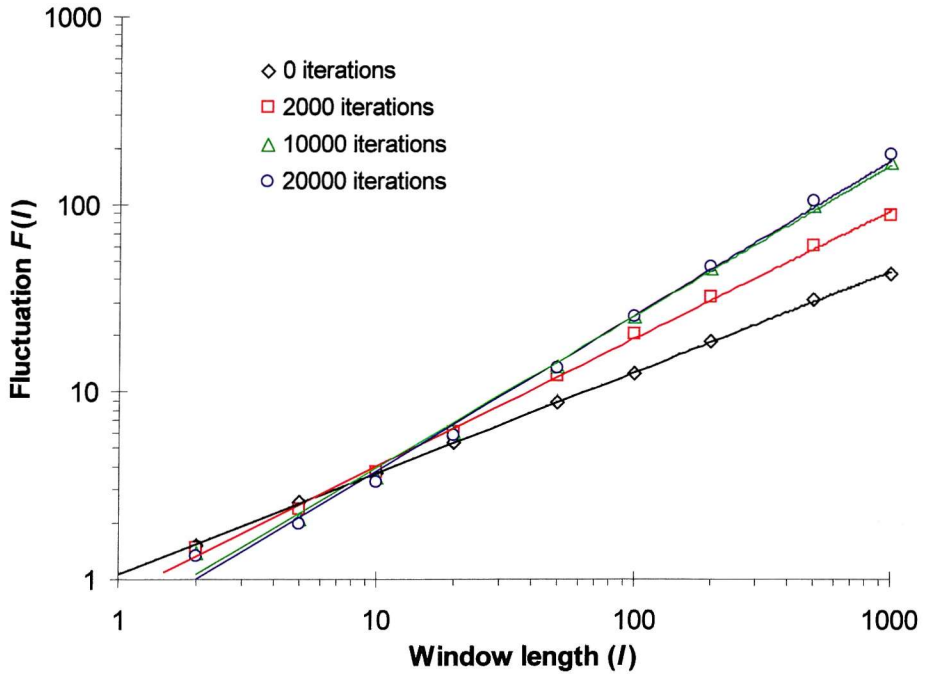


Figure 86 Fluctuation analysis showing effect of point mutations; insertion into HTLV II; point mutations:insertion/excisions ratio 1:1; transition:transversion ratio 2:1; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6

The simulations were repeated, but with an increase in the ratio of point mutations:insertion/excision events to 9:1. The Zipf results are shown in Figure 87 and the fluctuation data in Figure 88.

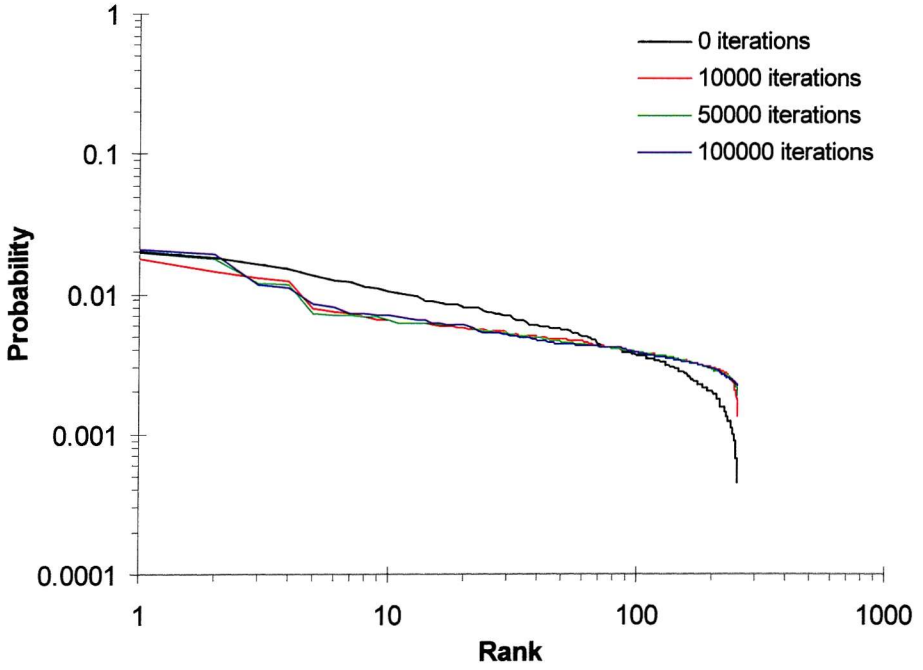


Figure 87 Zipf plots showing effect of point mutations; insertion into HTLV II; point mutations:insertion/excisions ratio 9:1; transition:transversion ratio 2:1; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; analysed with word length 4

The Zipf plots show that at a ratio of 9:1 the point mutations have a greater effect on the sequences produced than the transposable element insertion/excision events. The net result is that the lines are pushed down by the randomisation effect of the point mutations. There is still a step effect which seems to be made up of the four most common words.

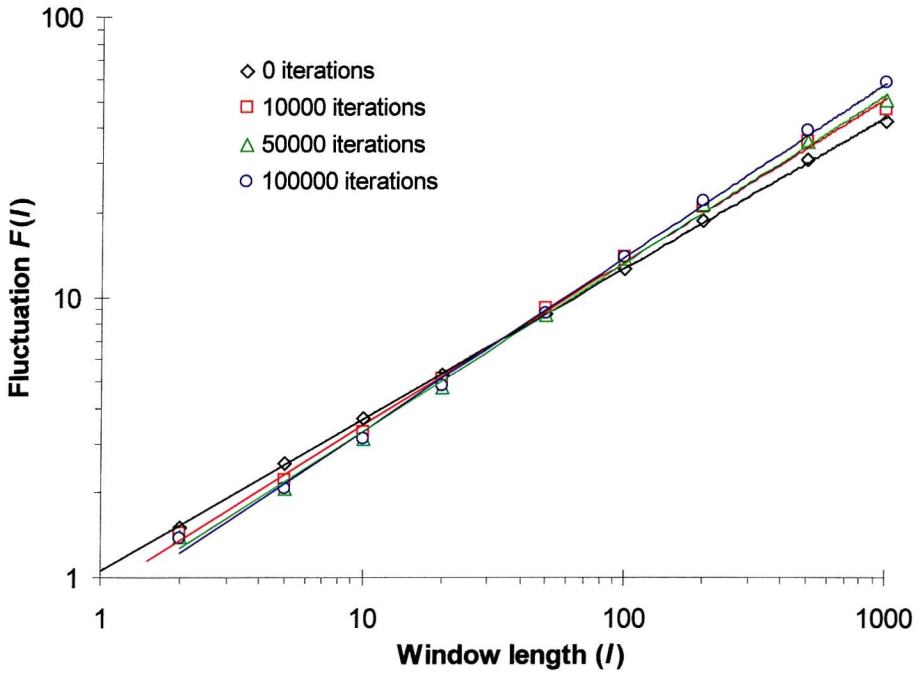


Figure 88 Fluctuation analysis showing effect of point mutations; insertion into HTLV II; point mutations:insertion/excisions ratio 9:1; transition:transversion ratio 2:1; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6

The fluctuation results show a slight increase in the fluctuation exponent with iterations. The gradient after 100,000 iterations is 0.62 (the starting sequence has a value of 0.54). In this case the action of the insertion/excision events is stronger than that of the point mutations and the net effect is to increase the fluctuation exponent, albeit at a slower rate than without point mutations.

Figure 89 and Figure 90 show the variation of Shannon entropy and fluctuation exponent with iterations for simulations run with point mutations. Note that the x-axis in both cases measures the average number of transposable element insertion/excisions that take place in the simulations – at a 1:1 point mutation:insertion/excision ratio, for example, the actual maximum number of iterations performed is 200,000 (rather than 100,000), but the number of insertion/excision events performed is approximately 100,000. The Shannon entropy chart shows how increasing the ratio of point mutations reduces the drop seen in the entropy when transposable element insertion/excision events

are modelled. Indeed, with the ratio of point mutations at its highest, the Shannon entropy actually increases slightly due to the randomisation of the sequence.

In the case of the fluctuation data, Figure 90 shows how the fluctuation exponent decreases as the ratio of point mutations is increased. Again, this effect is due to the randomising effect of the point mutations which break down long-range correlations built up by the duplication of sequences performed by the transposable element model.

The results show that point mutations performed at the realistic level of one for every transposable element insertion/excision event have a small effect on the fluctuation exponent, but a greater effect on the Shannon entropy values and the quality of the Zipf plot curves. Point mutations had no effect on the depression seen in the fluctuation lines.

Figure 91 shows a sample sequence generated with both point mutations and transposable element insertion/excisions. The sequence appears far less blocky than those seen for the model without point mutations.

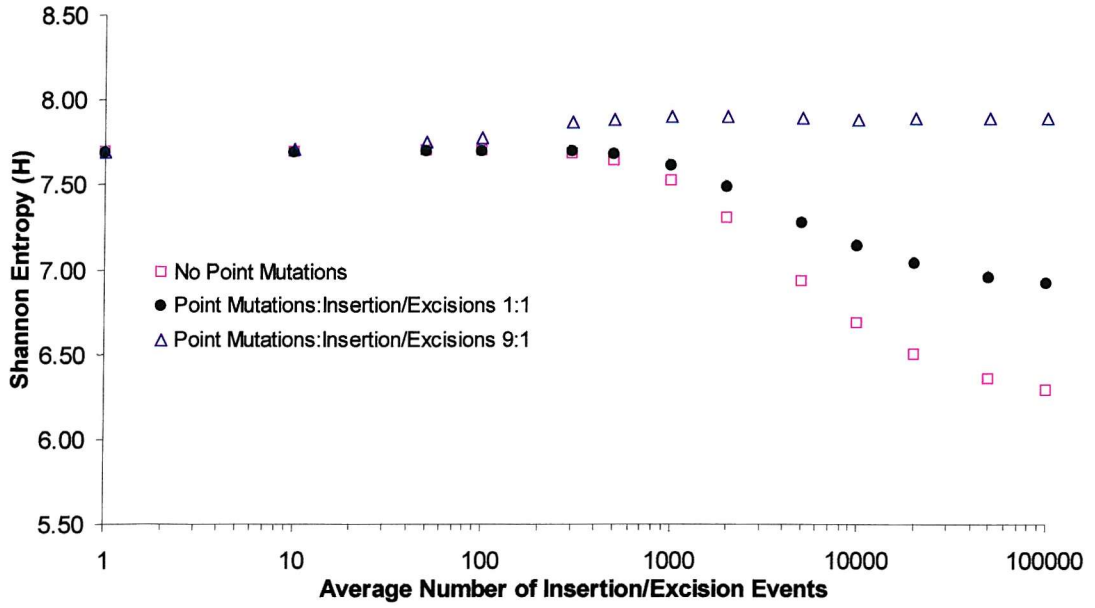


Figure 89 Variation of Shannon Entropy with iterations showing effect of point mutations; insertion into HTLV II; point mutations:insertion/excisions ratio given in figure; transition:transversion ratio 2:1; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; analysed with word length 4

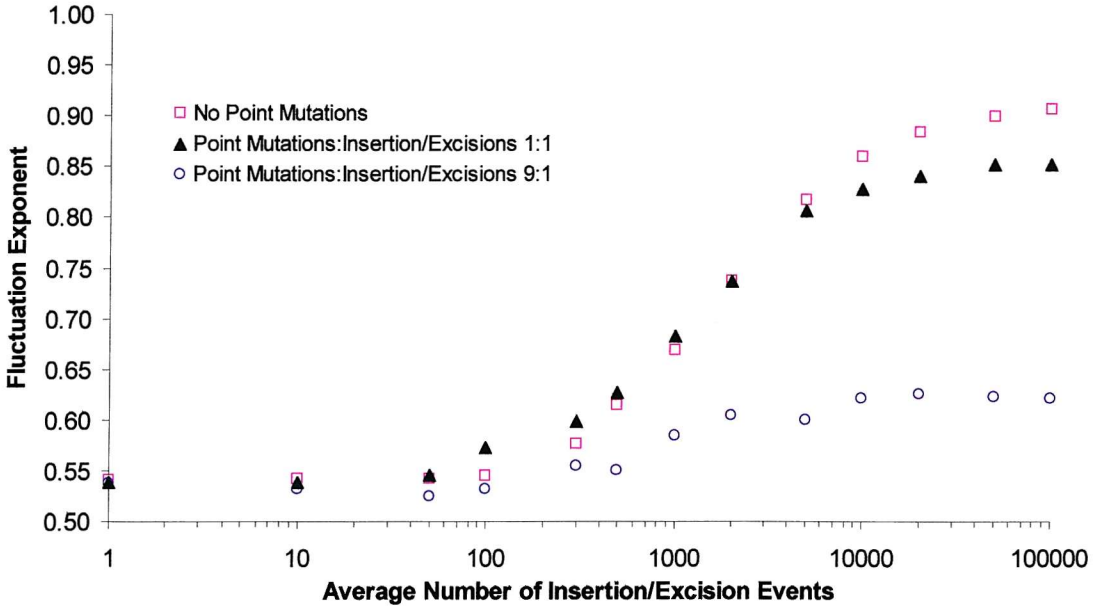


Figure 90 Variation of fluctuation exponent with iterations showing effect of point mutations; insertion into HTLV II; point mutations:insertion/excisions ratio given in figure; transition:transversion ratio 2:1; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6

TCTCAGTGCGTCTCCCGGTTACTGAACATTGTGACTTGAAGCTGTGTACAACCTCTTGGG
GTGCCACCGTGGGTATCACTATGTGTATCATTAGCAGTGTAAGTATGTGCAAACATGACT
AGAGGGACTACAGATTTCCCATAATAATAACTATAATTATAGTGGAAGCGGACGCGGACG
CGCACGCGGACGCGGACGCGGATTCGGACCTCTGCCTCAGCCCCAGCGTCAGCGTCAGCA
TTGCCGTACACATGAACTCTTTCCGTCGGACTTACCCGGACCCGGACGAGTCGACAGGGA
TAGAGACGGGAACGGGTCGGATCCGATGCCGCTAGAAAACCAGCCCGACCCTCCAGAAGG
CTCTGACTCAGACTCCGACTCTCGGTCAGGGTCAGACTCAGACGGAGAGTCATTGCATGT
GCTGAGGATATGAGTTGTACAATGTGCTAGCGAAGCTCTTTCGCTCAAGACCCGCCGATT
GCGATGGGGCTTAGGTTCCAACCTTAGGTTCGACTTTCGTCATGTTTTGACCGTGGTAGT
ATGTTGATAAAAGGTCTTACTGTGCCGATTGTTATATACCAAGTTCGGCGGAATTAACTC
ATACAGGCGACATTCGACGTACCCGTAAGTAGAGAAAAGAACAGGGACAAGGACAGGGAC
GGGGACCTAATGGGATTTACATGTGTCTAATAGGACTGGGACTGGGACTTAAGTCCGGCA
TTAGAGTGAGAGCTATGGTCAAAACGCAGACGCGCCCCTGGATATCAGGGAGGCTCGAAT
CTAAACTTCGAGCACTATATTAAATCATAACGGACATGGACGTGGACACAGATTAAGCGA
GGATTCGATTTCCGACCCAAACCTGGGGACGTAGTAGTGAGCACGAGACCGGGACTTCGA
GAGCAGCGTCAGCAGTGCCCTTGTGCTAATAACCTGAGGTTCGTCTTAAGCTTCAGCCTC
AGATTCGGACTCGGACTCGGACACGGACTTGGACTTGGACCGGGACCAGGACCCGTAATC
ATGACCTAACGTCTTAATAATAGCTAATGACAGGGTTTGAACTTGGAATTTGACCTGGAC
TCTGACTTGGATCAGGGCGCGACCGCTACTTAGAGGTAGTACTGGAACGATCCGGATTAG
TACAAGTACTAGGAAGCCATTCCGATCTCATTCGAGTGTGTTGCTTCAGGCCCGATATAT

Figure 91 Sequence generated using point mutations. Colour coding helps to highlight the sequences produced by target site duplication. The first 1200 bases of a sequence generated after 50000 iterations; starting sequence HTLV II; ratio point mutations:insertion/excision events 9:1; transition:transversion ratio 2:1; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6

Although point mutations do not improve the quality of the Zipf plots or the fluctuation lines, they do make the generated sequences appear visually more realistic. Other possible "noise generating" processes need to be looked into in order to improve the statistical properties of the sequences generated. The first of these was the simulation of imperfect excision of transposable elements.

6.4 IMPRECISE INSERTION AND EXCISION OF TRANSPOSABLE ELEMENTS

The first model of transposable element insertion and excision assumed that both of these processes were carried out without error. This is not the case in nature. As described previously, on excision some bases of the transposable element may be left in the host sequence. Alternatively, bases of the host sequence may be removed along with the transposable element. These errors would help to break up the current perfectly repeated structure.

Adding imprecision to the model required the following details to be considered:

1) How should the model determine whether to do:
   a) a perfect excision,
   b) an excision with insertion of additional bases, or
   c) an excision with deletions of host bases?
2) When performing an insertion of extra bases:
   a) How many bases should be inserted?
   b) Where should the bases be inserted?
   c) What particular bases should be inserted?
3) When performing a deletion of host sequence bases:
   a) How many bases should be deleted?
   b) From which positions should the bases be deleted?

After assessing the above points, the program was updated to simulate imperfect excision of transposable elements. A flowchart outliningthe new subroutine is given in Figure 93.

Figure 92 shows how a target site is affected by a perfect excision, an excision with insertion of extra bases, and an excision with deletion of extra bases in the updated version of the simulation. Several adjustable parameters were used in the new model:

1) A ratio used when a transposable element insertion/excision event is to be modelled, to determine whether the excision will be:
   a) perfect,
   b) imperfect with insertion of additional bases, or
   c) imperfect with deletion of bases from the host sequence.
2) In the case of insertion of additional bases:
   a) The upper limit for the number of bases to insert.
   b) The lower limit for the number of bases to insert.
3) In the case of a deletion of bases from the host sequence:
   a) The upper limit for the number of bases to delete.
   b) The lower limit for the number of bases to delete.

When selecting the number of bases to insert or delete from within a range, a value was chosen at random with each number in the range given an equal probability of selection. The bases that were inserted were chosen at random from the four possibilities, with equal probability. On deletion of an odd number of bases, the bases were deleted symmetrically (as shown in Figure 92) with the additional base being removed from a side chosen at random, with each side having an equal probability of being chosen.

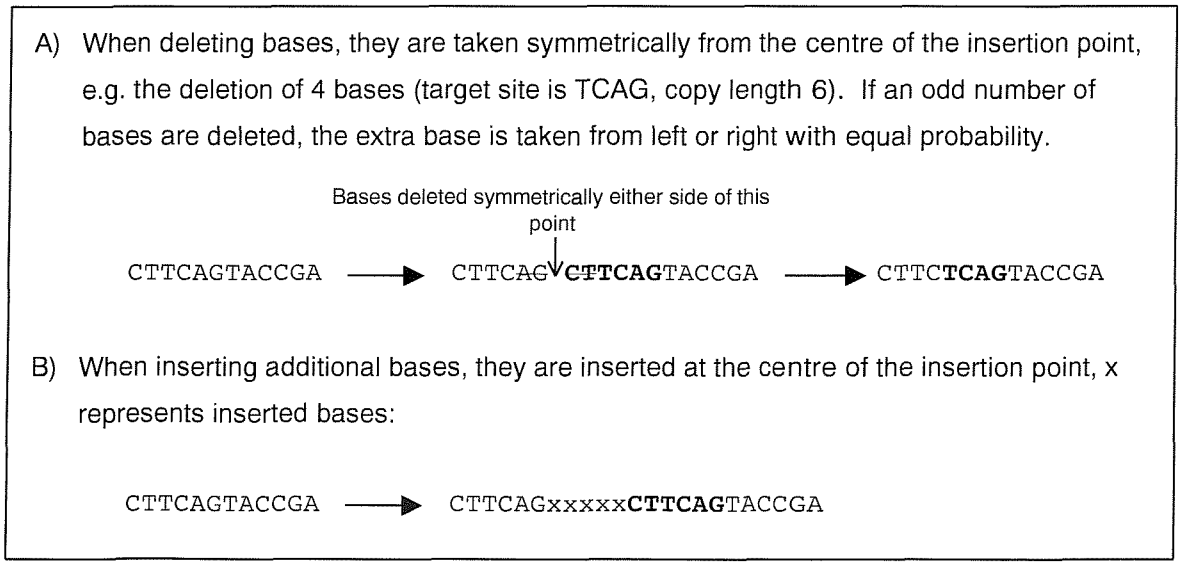


Figure 92 Details of imperfect insertion and excision showing which bases are inserted and deleted

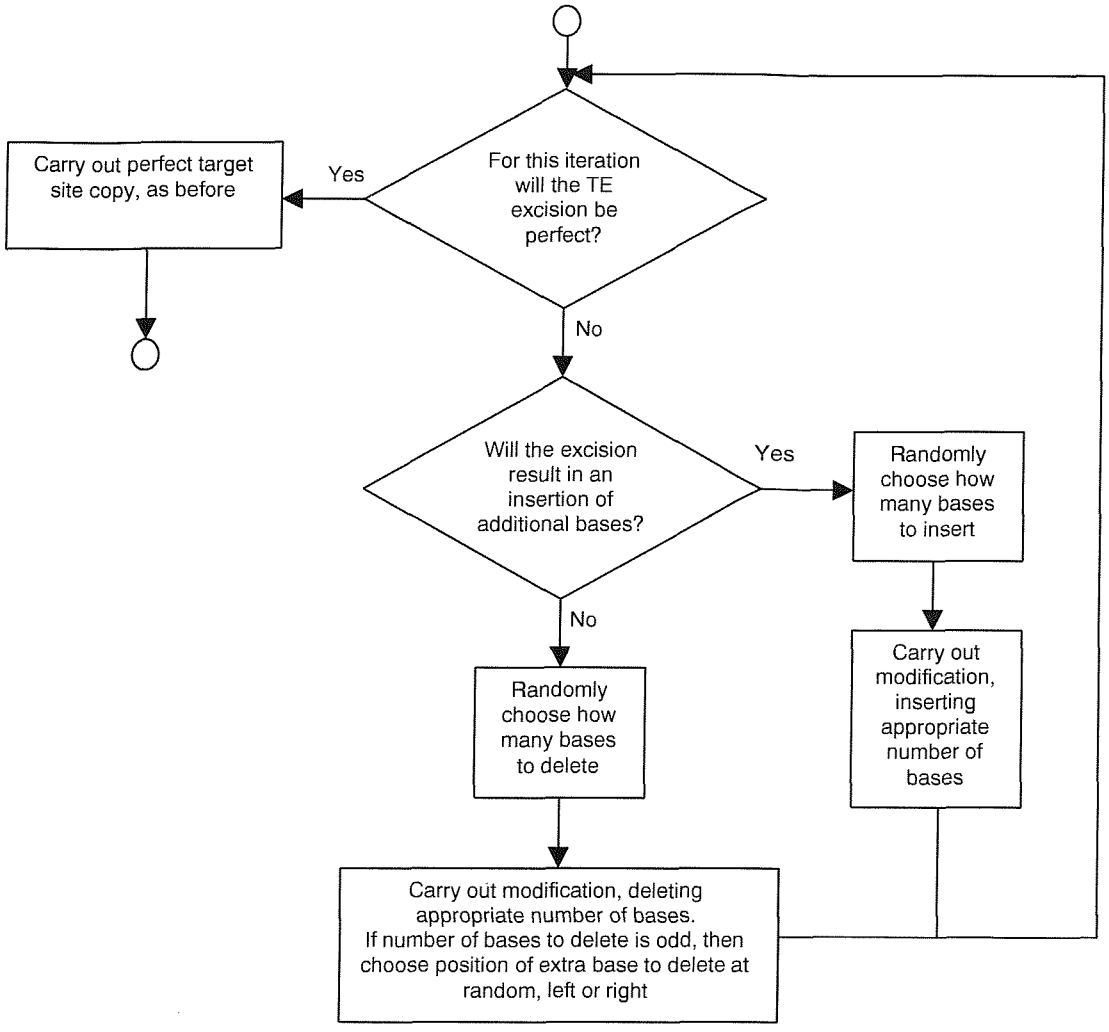


Figure 93 Flowchart depicting the imprecise insertion/excision subroutine.

6.5 *THE EFFECT OF IMPERFECT TRANSPOSABLE ELEMENT EXCISIONS*

The simulation was performed using three target sites and with a ratio of the three possible outcomes of imperfect excision (perfect excisions; imperfect excisions with insertions; imperfect excisions with deletions) of 1:1:1. Point mutations were not used. In imperfect events the number of bases inserted or deleted was set at 3. The Zipf and fluctuation results from the sequences are given in Figure 94 and Figure 95.

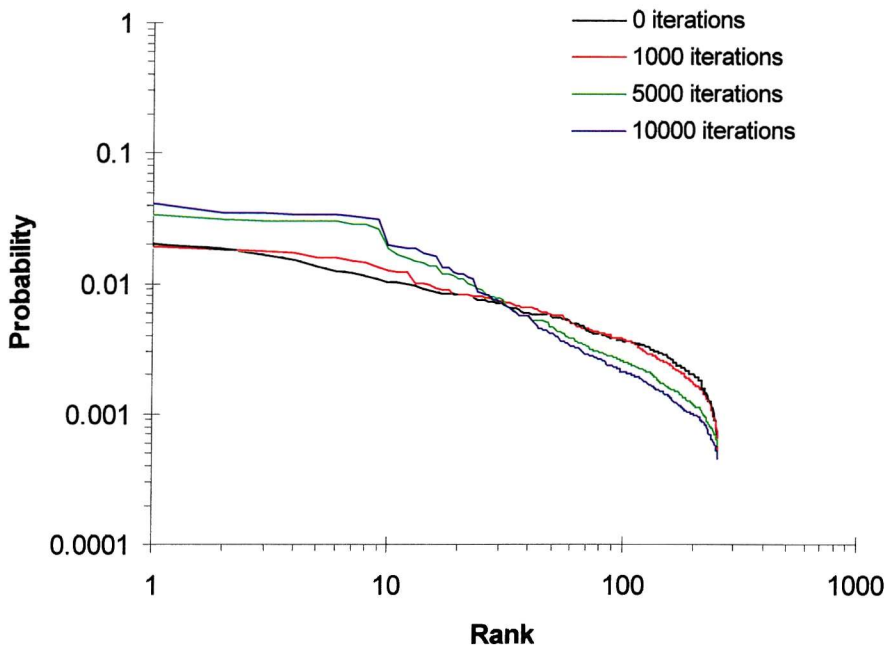


Figure 94 Zipf plots showing effect of imprecise excision; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; ratio perfect excision:imperfect with insertion:imperfect with deletion 1:1:1; 3 bases inserted in imperfect excisions; 3 bases deleted in imperfect excisions; analysed with word length 4

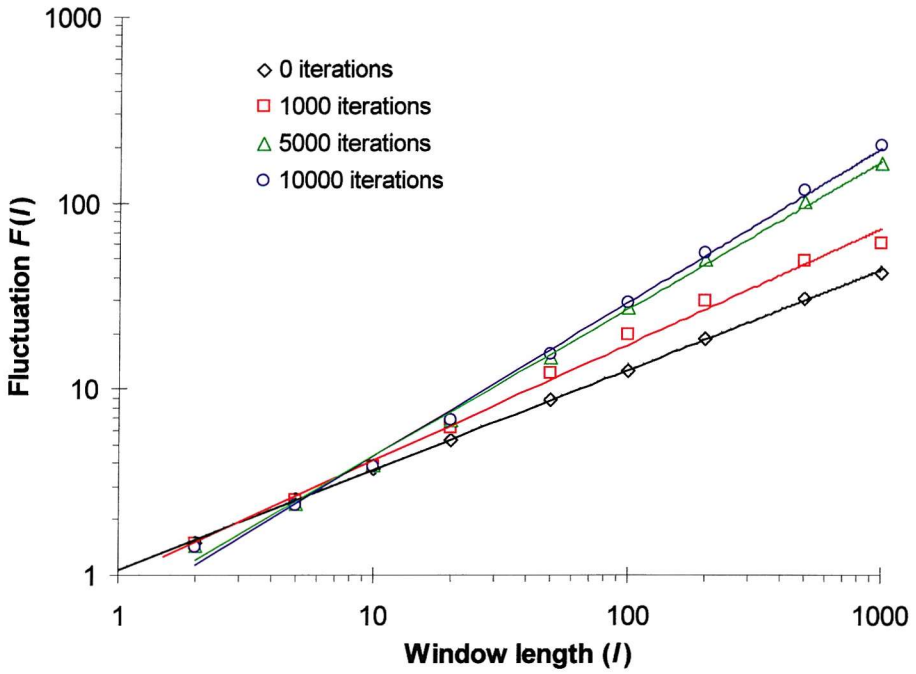


Figure 95 Fluctuation analysis showing effect of imprecise excision; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; ratio perfect excision:imperfect with insertion:imperfect with deletion 1:1:1; 3 bases inserted in imperfect excisions; 3 bases deleted in imperfect excisions

The Zipf plot shows a flat portion to the start of the curve which then becomes quite jagged in the centre. Because of the shape of the line it is not possible to measure the gradient.

The fluctuation results appear quite similar to those seen earlier with three target sites but importantly the magnitude of the depression in the line seen in earlier results appears much lower. The fluctuation exponent of the line after 10,000 iterations is 0.83, this compares with 0.86 for the basic model without imperfect transposable element excisions.

The simulations were repeated with increasing proportions of the excision events made imperfect. The fluctuation results varied little, but the Zipf results showed that the greater the proportion of imperfect insertions, the larger the step seen in the lines. Figure 96 shows a series of Zipf plots – the step at around rank 15 can be seen to increase as the proportion of imperfect insertions is increased.

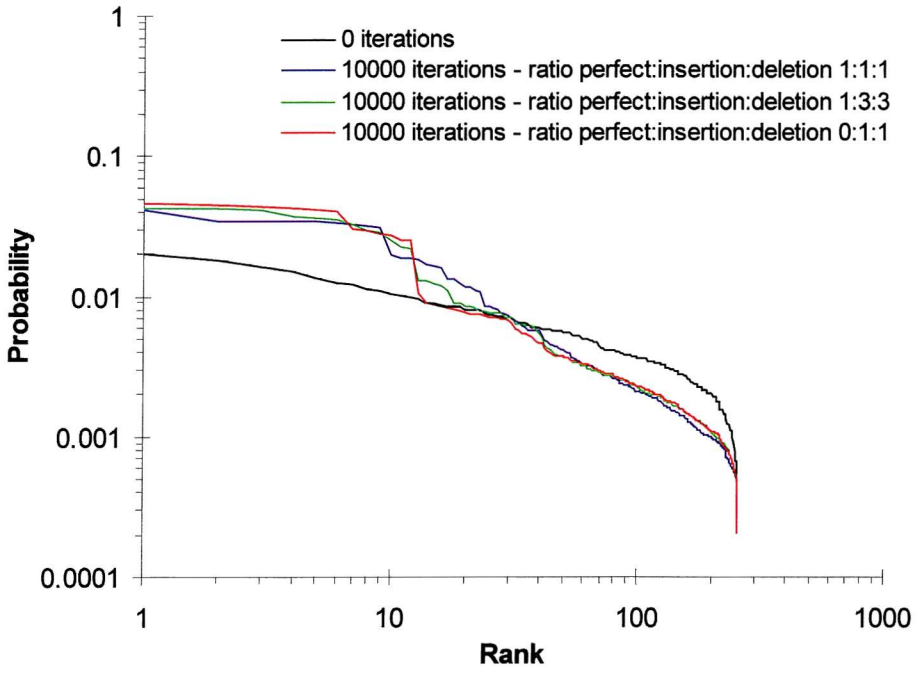


Figure 96 Zipf plots showing effect of imprecise excisions; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; ratio perfect excision:imperfect with insertion:imperfect with deletion given in figure; 3 bases inserted in imperfect insertions; 3 bases deleted in imperfect deletions; analysed with word length 4

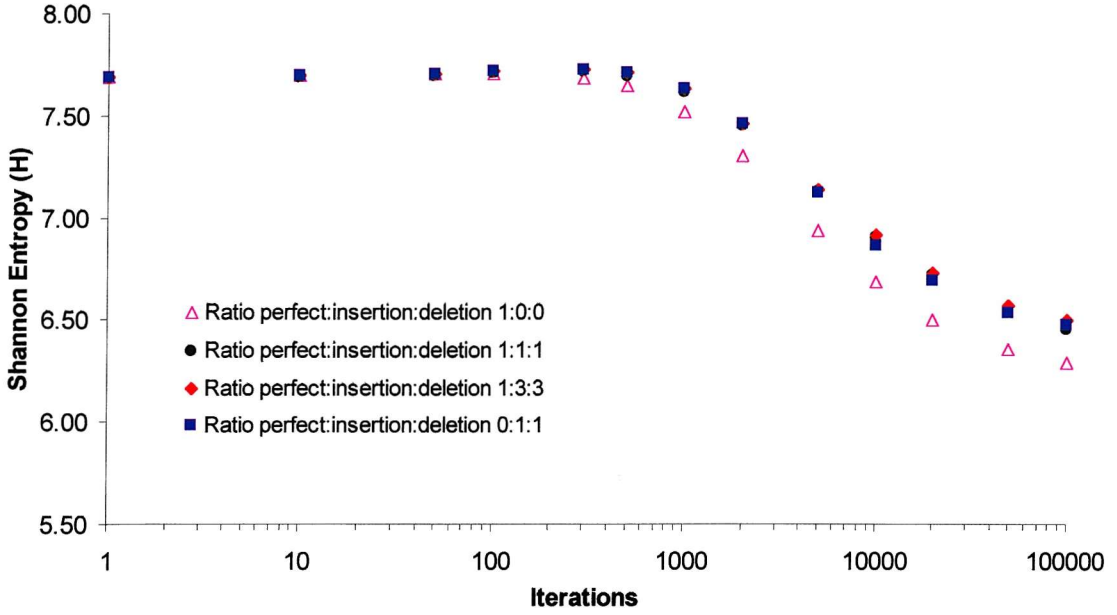


Figure 97 Variation of Shannon Entropy with iterations showing effect of imprecise excision; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; ratio perfect excision:imperfect with insertion:imperfect with deletion given in figure; 3 bases inserted in imperfect insertions; 3 bases deleted in imperfect deletions; analysed with word length 4

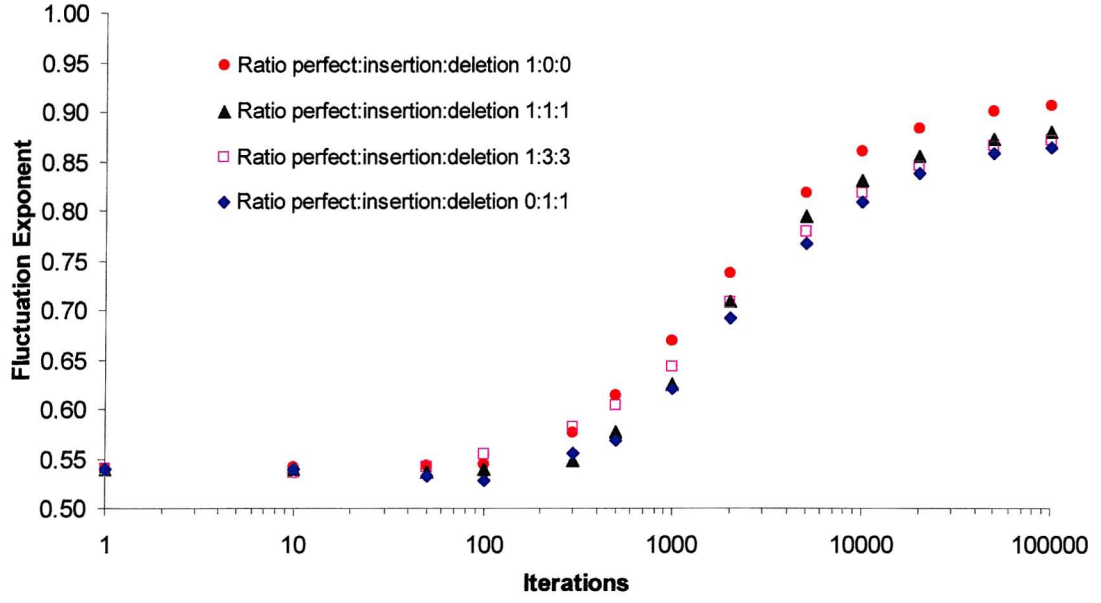


Figure 98 Variation of fluctuation exponent with iterations showing effect of imprecise excision; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; ratio perfect excision:imperfect with insertion:imperfect with deletion given in figure; 3 bases inserted in imperfect insertions; 3 bases deleted in imperfect deletions

The Shannon entropy and fluctuation results are given in Figure 97 and Figure 98. In both cases it can be seen that while there is some difference between the results obtained with all excisions perfect and those including imperfect excisions, as the proportion of imperfect excisions are increased there is little further change.

In order to investigate the effect of insertions and deletions in imperfect excisions more closely two simulations were run, the first using only imperfect excisions with insertions, and the second only imperfect excisions with deletions. The comparison between the Zipf plots is given in Figure 99, and the fluctuation results are shown in Figure 100. The two charts show that using insertions only results in a Zipf plot which, while irregular, follows a basically linear profile over most of the line, but a fluctuation result with a noticeable depression in the line. Using deletions only there is a massive step seen on the Zipf plot, but the fluctuation line shows little, if any, sign of the depression seen in other fluctuation plots.

It seems, then, that the steps seen in Zipf plots when using imperfect excision events (with both insertions and deletions) are due to the deletion of bases. Visual inspection of the final sequences shows that when only deletions are used the sequence is very blocky, and that all insertions at a particular site type (e.g. TCAG) generate the same repetitive unit (in the case of TCAG, the repeating unit is TCA) to lead to the formation of the blocks. The reason that each target site should lead to a particular repetitive sequence is not immediately obvious, and requires a closer examination of the details of the block formation.

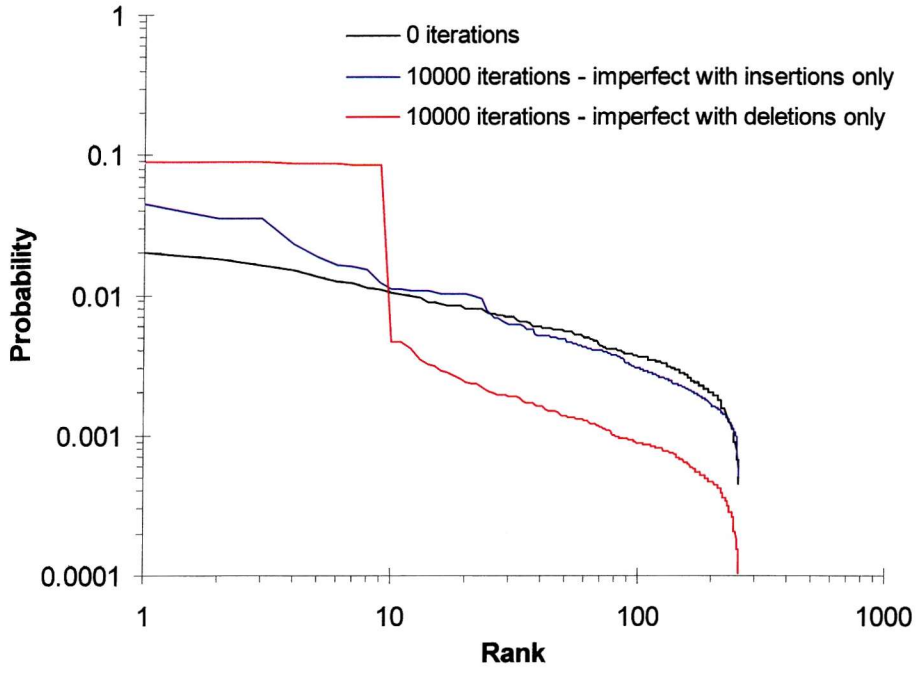


Figure 99 Zipf plots showing effect of insertions and deletions in imprecise excision; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; all insertion/excision event imperfect with insertion or deletion as given in figure; 3 bases inserted in imperfect insertions; 3 bases deleted in imperfect deletions; analysed with word length 4

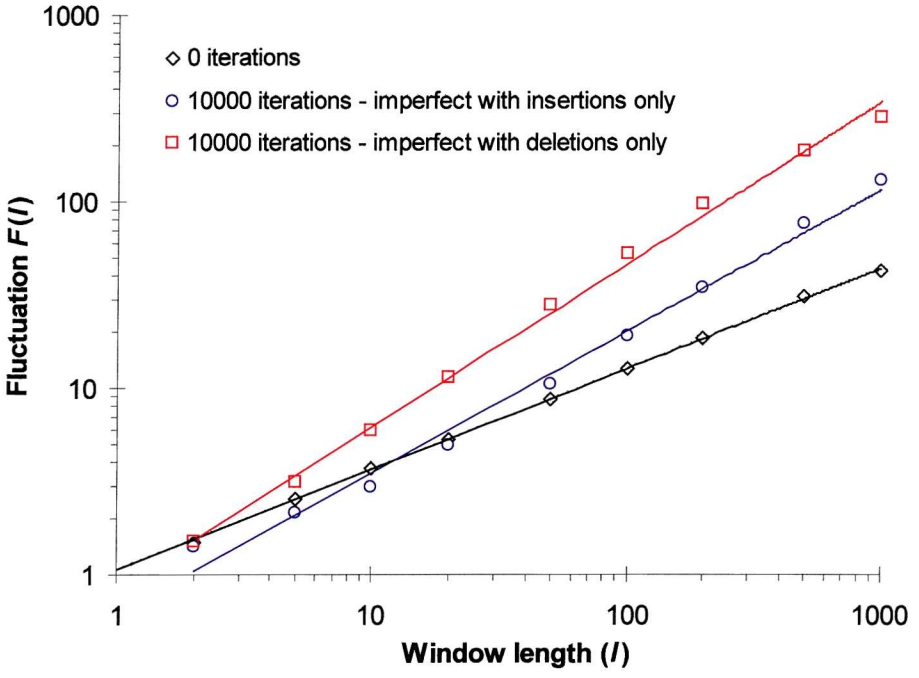


Figure 100 Fluctuation analysis showing effect of insertions and deletions in imprecise excision; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; all insertion/excision event imperfect with insertion or deletion as given in figure; 3 bases inserted in imperfect insertions; 3 bases deleted in imperfect deletions

As shown in Figure 101 the deletion of three bases during an imperfect excision can result in two possible sequences, depending on whether the odd base is taken from the left or right of the insertion point. If the bases to be deleted are removed from the right (C) then the product (E) is such that either deletion (I) or (J) from a future duplication event will lead to the formation of the same sequence ((M) and (N) are identical). Note that future deletions during copying events from sequences (M) and (N) will generate more TCA repetitive units as the duplication will proceed in the same manner as at (G) (the sequence CATCAG will be duplicated and deletions from both possible positions will lead to the same product and the formation of another TCA unit). Once the deletion process has gone via (C) (i.e. the odd base is deleted from the right) all future deletion events at that target site will lead to the same product, and the proliferation of the TCA sequence.

If the deletion proceeds initially via (D), one of the final products, (O), will lead to the formation of TCA units if a further duplication occurs at this site. (P) has the same general sequence as (F) (xxCGTCAG) so will lead to products like (O) and (P) again ((O) generates TCA; (P) generates TCG).

Inevitably at some point the deletion will result in the formation of a TCA unit and from this point all future excision events in this block lead to further TCA units. Note that in the starting sequence (A) the two Gs are irrelevant – any bases could be substituted and the final result would be the same, the formation of TCA repeats. In this way, all insertions at TCAG sites will generate blocks consisting of the TCA repeat, no matter where in the starting sequence they are located. Corresponding situations exist for insertions at GGAC sites and TATA sites.

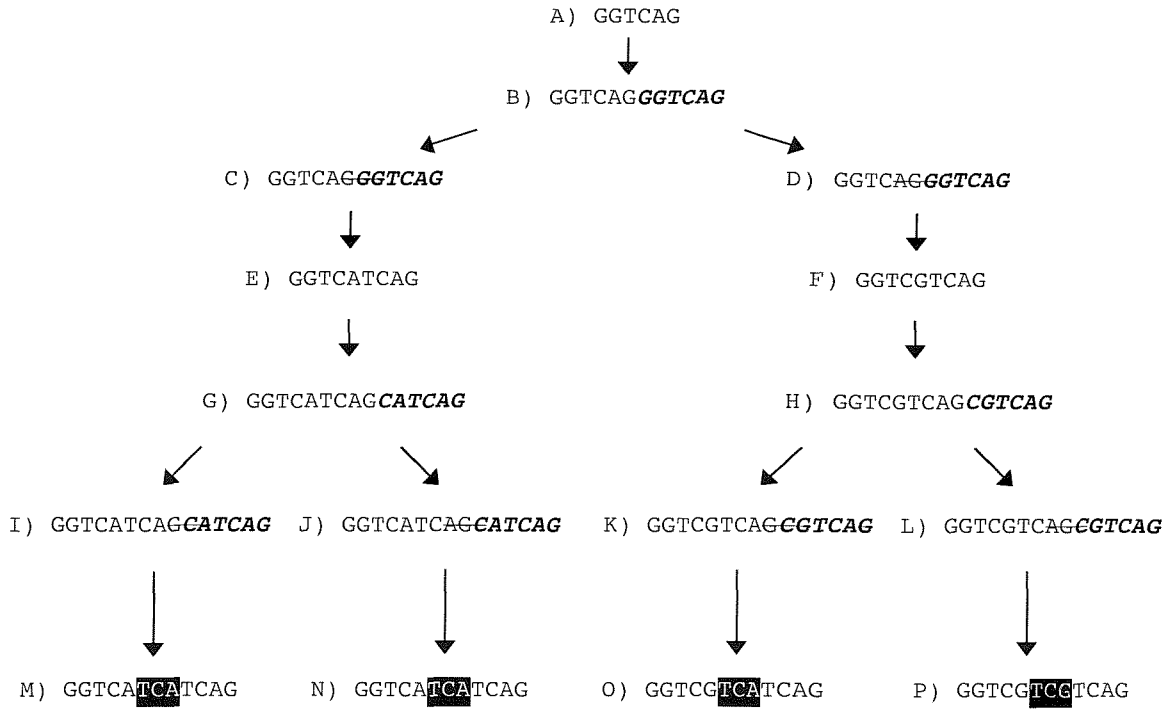


Figure 101 Formation of repetitive blocks. Duplicated sequences are shown in bold italic; bases to be deleted are shown crossed through; the repeat unit generated by the overall process is shown highlighted in black. (M), (N) and (O) have the same basic sequence as (E) (xxCATCAG) so will generate TCA repeats on future deletions.

The difference in fluctuation results between the two simulations (Figure 100) is interesting. It has been seen earlier that blocky sequences result in a large depression in the fluctuation line, and yet with the very blocky sequence generated by the use of deletions only there is no obvious depression. Running the simulation modelling only perfect excisions and using target sites TCA, TAT and GAG all with copy length 3 also generates a very blocky sequence (blocks made up of the repeating units TCA, TAT and GAG are generated, just as in the simulation using only imperfect excisions with deletions - see Figure 100) and this gives a very similar result for the fluctuation analysis. A large depression in the fluctuation line would be expected at around window length 3 - the same as

the copy length used. It seems that at a low copy length the depression is not seen.

6.6 THE EFFECT OF A RANGE OF INSERTIONS AND DELETIONS

The imperfect excision model was designed such that the number of bases inserted or deleted during an imperfect excision could be set to a range rather than a single value. The aim was that this would generate variety in the duplications made, and break up the perfectly repeated blocks seen in earlier simulations.

The simulation was run using the three target sites as in earlier experiments (copy length 6 and no bias towards any one site), and the ratio of perfect excisions:imperfect excisions with insertion:imperfect excisions with deletions set at 1:3:3. The ranges were set to 1-6 for both the number of bases to be inserted or deleted during imperfect excisions. The Zipf results for the simulation are given in Figure 102 and the fluctuation results are shown in Figure 103.

The Zipf plots show a line that is linear over a large portion of the plot, and does not show the degree of stepping seen in the equivalent plot in Figure 96. The gradient measured for 10,000 iterations is 0.60. The extra variety introduced by using a variable number of bases during imperfect excision events has resulted in a broader range of words being enhanced during the simulation. This has smoothed out the step that was seen earlier.

The fluctuation results are almost identical to those obtained when a discrete number of bases are inserted or deleted during an imperfect excision. The depression in the line is less noticeable when compared with the results achieved using perfect excisions.

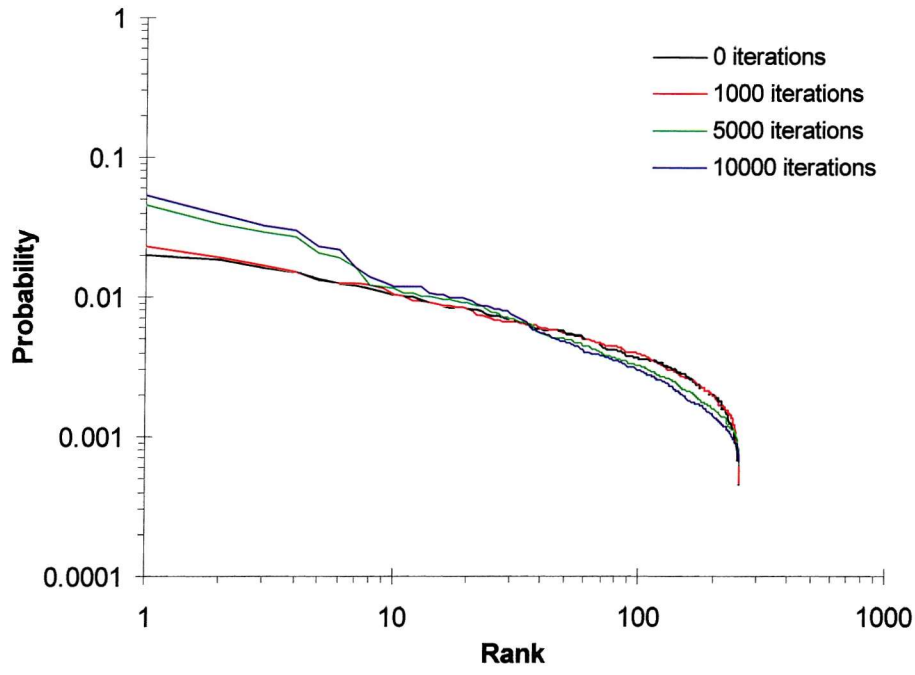


Figure 102 Zipf plots showing effect of imprecise excision using a range for number of bases to insert/delete; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; ratio perfect excision:imperfect with insertion:imperfect with deletion 1:3:3; 1-6 bases inserted in imperfect insertions; 1-6 bases deleted in imperfect deletions; analysed with word length 4

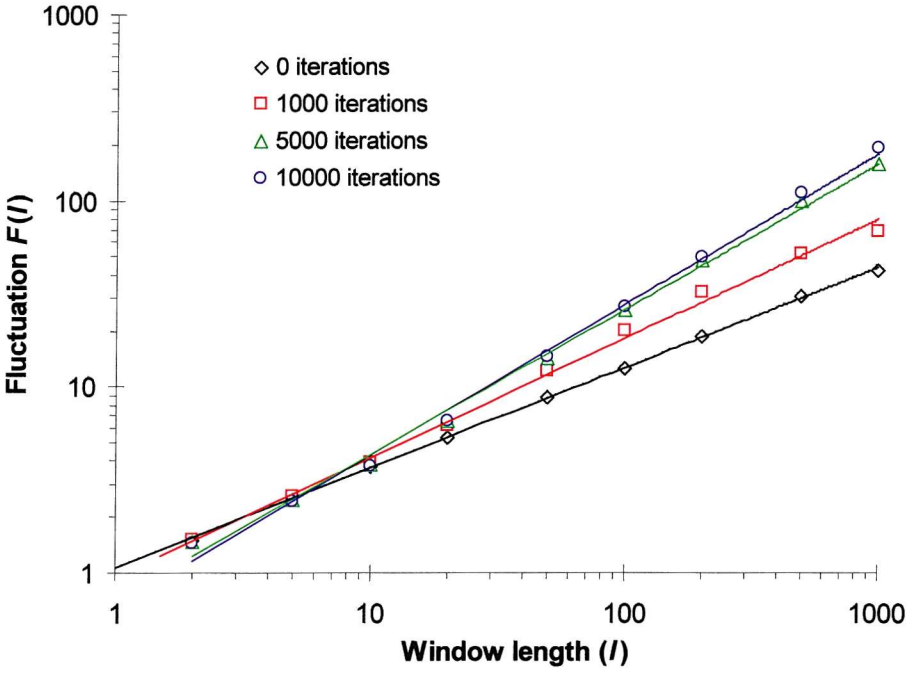


Figure 103 Fluctuation analysis showing effect of imprecise excision using a range for number of bases to insert/delete; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; ratio perfect excision:imperfect with insertion:imperfect with deletion 1:3:3; 1-6 bases inserted in imperfect insertions; 1-6 bases deleted in imperfect deletions

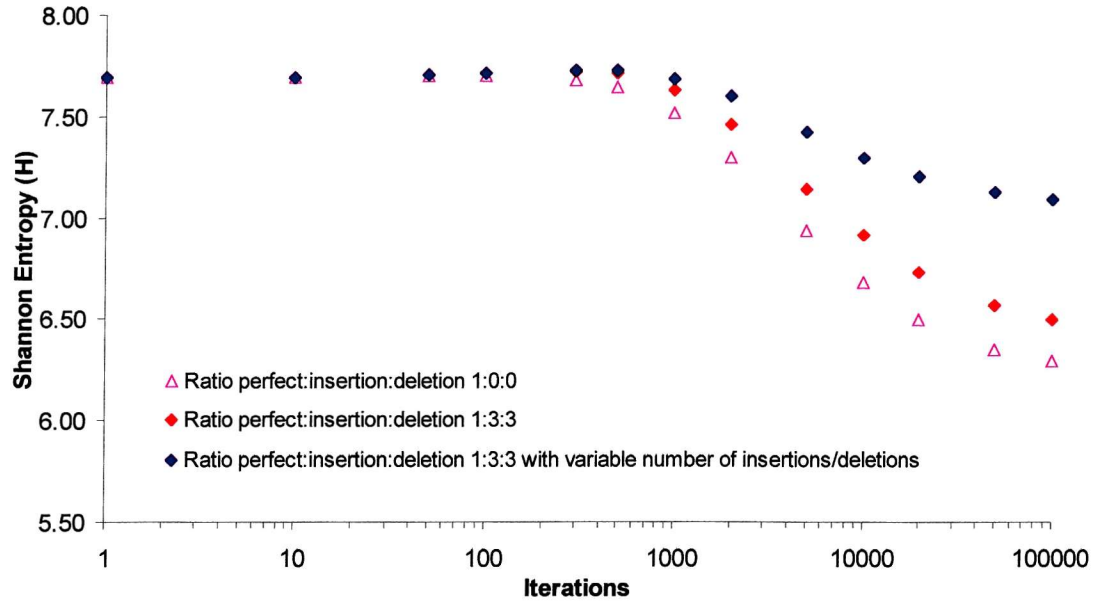


Figure 104 Variation of Shannon Entropy with iterations showing effect of imprecise excision; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; ratio perfect excision:imperfect with insertion:imperfect with deletion given in figure; 1-6 bases inserted in imperfect insertions if variable, 3 inserted otherwise; 1-6 bases deleted in imperfect deletions if variable, 3 deleted otherwise; analysed with word length 4

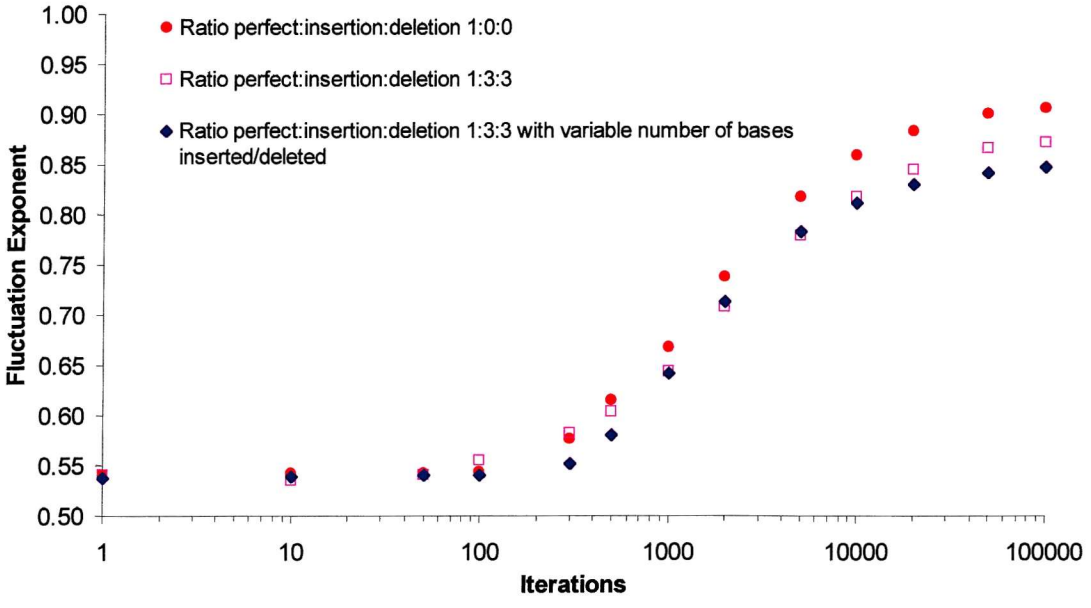


Figure 105 Variation of fluctuation exponent with iterations showing effect of imprecise excision; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; ratio perfect excision:imperfect with insertion:imperfect with deletion given in figure; 1-6 bases inserted in imperfect insertions if variable, 3 inserted otherwise; 1-6 bases deleted in imperfect deletions if variable, 3 deleted otherwise

The Shannon entropy results in Figure 104 show a much smaller decrease in entropy with iterations in comparison with results obtained using a set number of bases to insert or delete, and results achieved using perfect excisions only. Using a range of values in imperfect excisions breaks up the repetitive structures seen as a result of other simulations and therefore reduce the redundancy. This will result in a higher value for the Shannon entropy.

The fluctuation results in Figure 105 show how the fluctuation exponent varies with iterations and compares results using a variable number of bases for insertion/deletion with those using a set number of bases, and those where all excisions were perfect. There is a small difference between the results which both use imperfect excisions, such that at 100,000 iterations the fluctuation exponent is slightly higher for the sequences obtained using a set number of bases for insertion and deletion.

Figure 106 shows a section of the sequence obtained from a simulation using imperfect excisions and a range of bases for insertion/deletion during imperfect events. It can clearly be seen that the regular repeating structure seen with earlier simulations has been broken up and the sequence appears more realistic.

TGACAATGGCGACTAGCCTCCCAAGCCAGCCACCCAGGGCGAGTCATCGACCCAAAAGGT
CGTCAGATTGCGTCAGGCGTCATCAGGTCAGTCATGGTCAGATACGTGGTCAGGGTCAGG
GTCAGCGTCAGTACGGGTTTCAGTTTCAGGTTCAGTCGTTCAGTTCAGGTTCAGTTAGTT
CAGTTCAGGGTTTCAGAGGTTCATTCAGTGGTTCTTCAGCTTCAGCGTGTTCAGTCAGTG
CTCAGTCAGTCAGGCGATGAGTCAGGGGGTCATCAGATGCGGTTCTCAGGGGTTCAGATT
CGGTCCAGGGTCAGATAAGGTTCTCACTCAGCCAACTCAGACTCAGCGCGACTCAGAGGT
CAGCTCGGTCAGTCATCCAGGGTCAGTTCATTGGTCAGGTCAGGGGGGTCAGTCAGGGTC
AGTCAGACAGTCAGTTAGTCAGAGTAAGTCAGGGTCGTCAGCCCGTCAGGGCGGTCGTCA
GTTCATCAGCATCAGCGTATCATCAGCTAATTCATCAGGAGAGCATCAGCATCAGACCGT
CTCACACAAACAATCCCAAGTAAAGGCTCTGACGTCTCCCCCTTTTTTTAGGAACTGAAA
CCACGGCCCTGACGTCCCTCCCCCCTAGGAACAGGAACAGCTCTCCAGAAAAAAATAGAC
CTCACCCTTACCCACTTCCCCTAGCGCTGAAAAACAAGGCTCTGACGATTACCCCCTGCC
CATAAAATTTGCCTAGTCAAAATAAAAGATGCCGAGTCTATCTATACATTCTATTCTATA
AATCTATAGGTCCTCTATAACTCTATAGTCTATATCTATAAAAGCGCAAGGAAGGACCGC
CAAGGACACAGAAGGACGATGCAAGGACAAGGACGCCAAGGGGACGGGGGAGGGACGGGG
ACACTGTAGGGGACCCTCGAAAGGACCCAAGGACAAGGACAAGGACAAGGGAAGGGAAGG
GACTCATCCAGGGACAGGGACAGGGACATAGGGACGCGCAAGGACAAAAGGACAAGGACT
GTAACAAGGACCTCCAAGGACAAGGGGAGGGACTTCAAGGAGGACAGATTGGAGGACCAA
GGAGGACGAGGACTGGATCGAGGACTGACAAGAGGACAGGAGGACCACCAGGACCAGGAC
ATATTCAGGACAGTTCATTCAGATTCAGCATTCAGAAGCAATTCAGGTGATTCAGCGCGT

Figure 106 Sequence generated using a range of bases for insertion and deletion during imperfect excisions. Colour coding helps to highlight the sequences produced by target site duplication. The first 1200 bases of a sequence generated after 5000 iterations; starting sequence HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; ratio pefect:imperfect with insertion:imperfect with deletion 1:3:3; 1-6 bases inserted in imperfect insertions; 1-6 bases deleted in imperfect deletions

7 Final Improvements to the Transposable Element Model

The addition of point mutations and imprecise transposable element excisions to the model helped improve the realism of the simulation, and went some way to generating results comparable to those seen for real non-coding DNA sequences. Some elements of the natural system had not been incorporated, however, so three additional factors were built into the model. Firstly, the use of a variable copy length during target site duplication. Secondly, the parameters relating to imperfect excisions were explicitly assigned one set to each target site (previously one set of parameters controlled all imprecise excisions in an identical manner regardless of the target site) and finally, the ends of the transposable elements were explicitly modelled so that bases left behind as a result of imperfect excisions were chosen from specific sequences rather than being chosen at random.

7.1 THE EFFECT OF VARIABLE COPY LENGTH

Up to this point the copy lengths used in the simulations were set to a discrete value, typically 6bp. This is not realistic, as transposable element insertions do not always result in the same number of bases being duplicated, even when considering the same transposable element. To simulate this the model was altered such that each target site (representing insertions by one particular transposable element type) was assigned an upper and lower limit on the copy length to be used for duplication at that site. Each possible value within the range delimited by the upper and lower values has the same probability of selection, the exact value being chosen randomly.

The simulation was performed using three target sites, each having the same probability of insertion and each with the copy length range 2-10. The results of

the Zipf analysis are shown in Figure 107. The plots are linear over a good proportion of the line and the gradient measured after 10,000 iterations is 0.72 (this compares with 0.53 when using copy length 6).

Figure 108 shows the results of the fluctuation analysis. Immediately obvious is the large depression in the line as seen in earlier experiments. Figure 109 is a comparison of the fluctuation results obtained after 10,000 iterations with a discrete copy length 6 against the copy length range 2-10. It is apparent from this that the gradient obtained using a variable copy length is lower than that obtained with a set copy length. Measurement of the gradients shows that using copy length 2-10 the fluctuation exponent is 0.71 and for copy length 6 it is 0.86. The depression in the fluctuation line is seen in both cases, but appears slightly less obvious when the range 2-10 is used, possibly because the gradient is lower than that obtained with copy length 6.

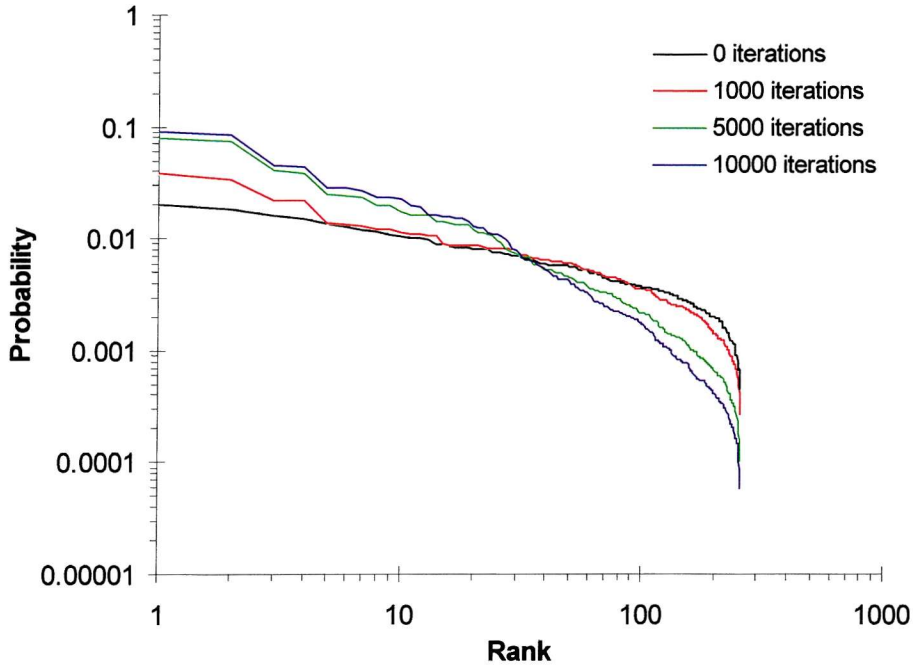


Figure 107 Zipf plots showing effect of variable copy lengths; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length range 2-10; analysed with word length 4

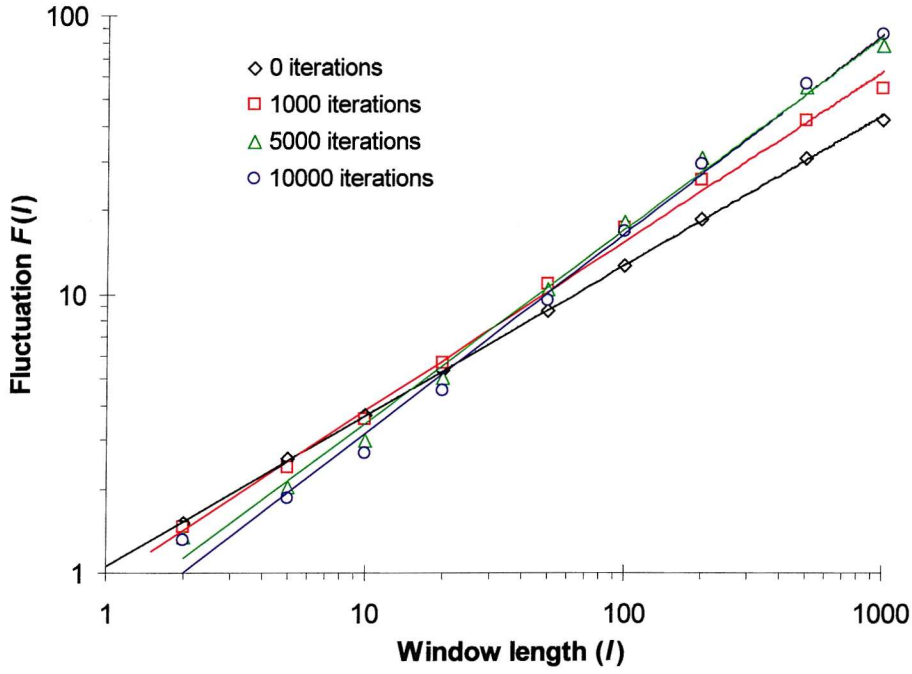


Figure 108 Fluctuation analysis showing effect of variable copy lengths; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length range 2-10

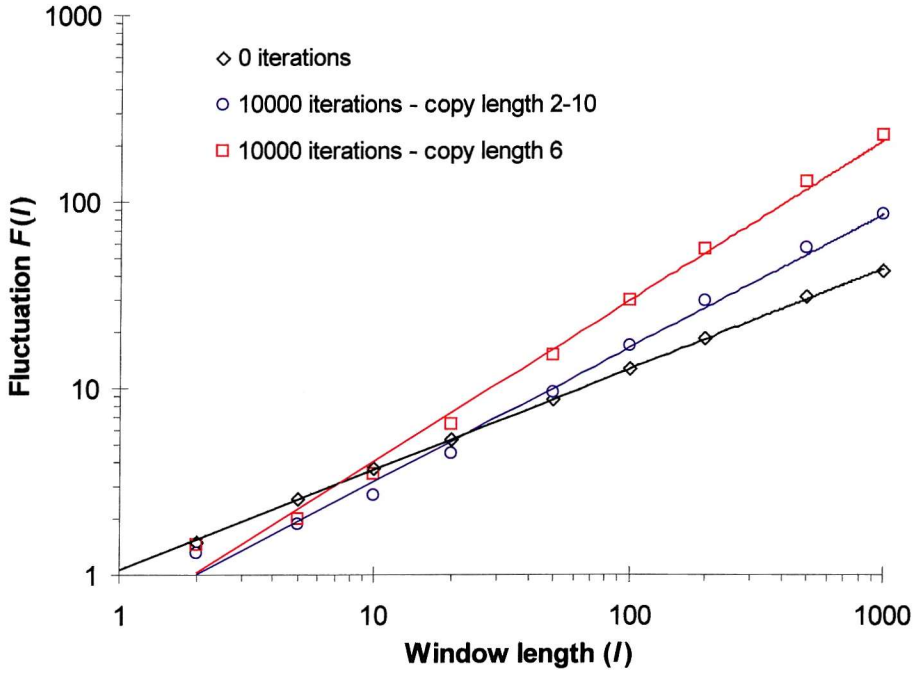


Figure 109 Fluctuation analysis showing effect of variable copy lengths; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy lengths given in figure

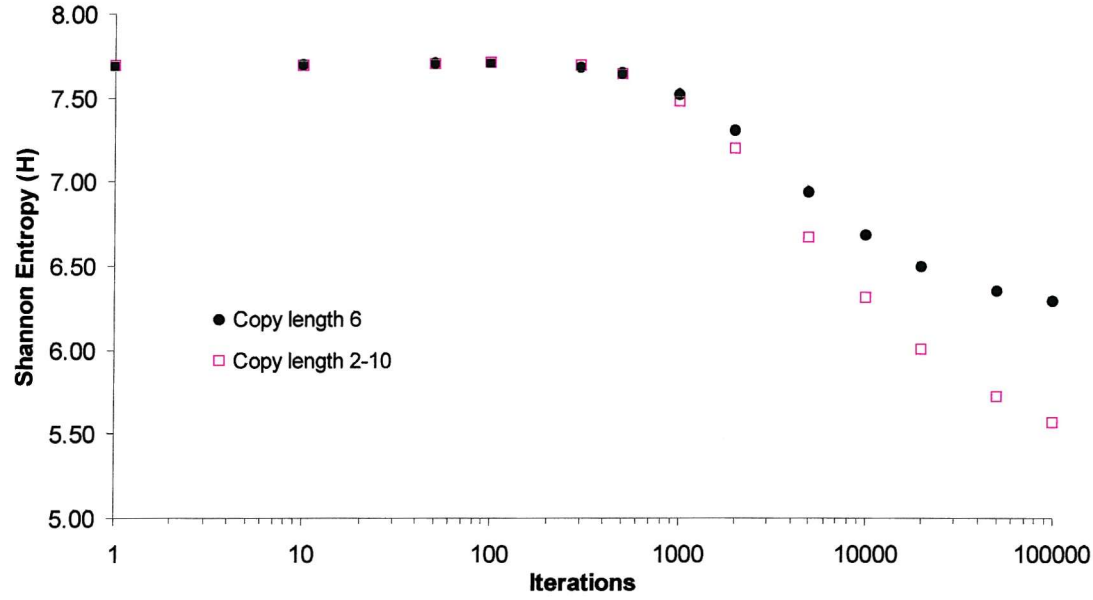


Figure 110 Variation of Shannon Entropy with iterations showing effect of variable copy lengths; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length given in figure; analysed with word length 4

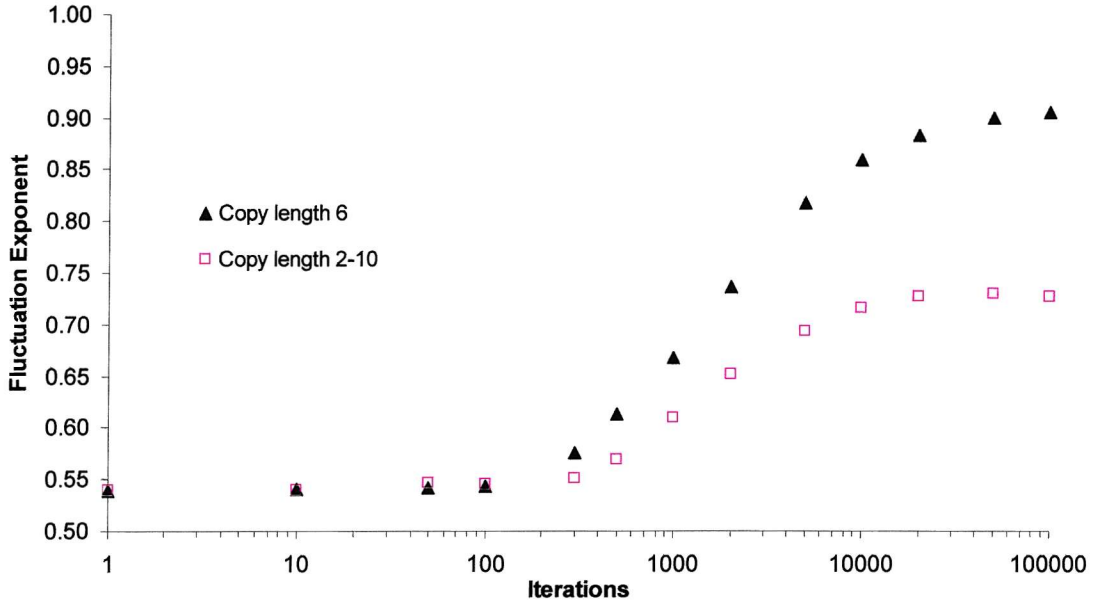


Figure 111 Variation of fluctuation exponent with iterations showing effect of variable copy lengths; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length given in figure

Figure 110 shows how the Shannon entropy is affected by a variable copy length. Figure 111 gives an indication of the effect of the variable copy length on the

fluctuation exponent. It can be seen that the Shannon entropy reaches a significantly lower value with high iterations when a variable copy length is used. This result is unusual, because it would be expected that a variable copy length would introduce more variation into the final sequence thereby giving a relatively low redundancy and high Shannon entropy value relative to a sequence obtained using a set copy length. Compare this result with that obtained when a range of bases were used in imprecise insertion/excision events (Figure 104). The extra variety in the final sequence resulting from a variable number of bases inserted or deleted in imprecise excisions lead to a lower final value for the Shannon entropy.

TGACAATGGCGACTAGCCTCCCAAGCCAGCCACCCAGGGCGAGTCATCGACCCAAAAGGT
CAGAAAGGTCAGGGTCAGAGGGTCAGGGGTCAGGGTCAGAGGGGTCAGAGGTCAGGGTCA
GGTCAGAGACCGTCTCACACAAACAATCCCAAGTAAAGGCTCTGACGTCTCCCCCTTTTT
TTAGGAACTGAAACCACGGCCCTGACGTCCCTCCCCCCTAGGAACAGGAACAGCTCTCCA
GAAAAAAATAGACCTCACCCTTACCCACTTCCCCTAGCGCTGAAAAACAAGGCTCTGACG
ATTACCCCCTGCCCATAAAATTTGCCTAGTCAAAATAAAAGATGCCGAGTCTATATCTAT
ATCTATAAGTCTATAGAGTCTATAATATCTATAATATCTATATCTATATATATATAATAT
AAAAGCGCAAGGACCAAGGACGACCAAGGACCAAGGACGACGCAAGGACGACGCAAGGAC
ACGCAAGGACGCAAGGACGGACGGACGACGGACGGACACCGGACGGACGGACCGGACGGA
CGACACGGACGGACCGGACCGGACCCGGACGACGGACGACGGACGACGGACACGACGGAC
CACGACGGACACGGACACGGACGGACCGGACAGTTCAGTCAGCAGCAGTTCAGAGTTCAG
TTCAGTCAGTCAGTCAGCAGCAGAGCAGTTCAGTTCAGTCAGAGCAGTTCAGTTCAGGTT
CAGGGTTCAGAGGACGGACTTCAGAGGACTTCAGGACTTCAGCTTCAGAGGACTCAGAGG
ACAGAGGACTCAGCAGAGGACTTCAGGGACTTCAGCAGGACTCAGAGCAGCAGGACTTCA
GGGACTTCAGAGGACACTCAGTCAGCAGAGGACGAGGACGAGGACCAGAGGACAGTTCAG
GTTCAGTCAGGTTCAGGTTCAGTTCAGTCAGTCAGCAGCAGTTCAGGTTCAGCAGTTCAG
AGTTCAGAGCAGTTCAGCAGTTCAGAGTTCAGTTCAGGAGTTCAGGAGTTCAGCAGTTCA
GAGTTCAGTCAGACAGTTCAGAGTTCAGGTTCAGAGGTTCAGTCAGAGGAGGTTCAGGAG
GTTCAGGTTCAGTCAGTCAGCAGTCAGGTCAGGTTCAGTTCAGAGAGGTTCAGTTCAGAG
CAGTTCAGTCAGTCAGTCAGCAGCAGAGGTTCAGGAGGTTCAGCAGGGTTCAGTTCAGAG

Figure 112 Sequence generated using a variable copy length. Colour coding helps to highlight the sequences produced by target site duplication. The first 1200 bases of a sequence generated after 5000 iterations; starting sequence HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 2-10

Figure 111 confirms the observation that the variable copy length results in a lower fluctuation exponent. The extra variation breaks up the long-range correlations to a certain extent, thereby lowering the fluctuation exponent.

Figure 112 shows a sample of the sequence generated after 5000 iterations of the simulation. It can be seen that the use of a variable copy length has made the sequence appear more realistic when compared with the earlier simulations, although the repetitive structure generated by the copying events is still noticeable.

7.2 THE EFFECT OF INDEPENDENT EXCISION PARAMETERS

When imprecise excisions were added to the transposable element model, the parameters controlling the excisions were included as one set controlling all insertion/excision events. This meant that all excision events had the same probability of being, for example, imperfect with deletions, regardless of which target site was involved. In natural systems each type of transposable element behaves differently with respect to factors such as whether the excision is likely to be perfect, or how many bases might be left behind after an imperfect excision[58, 59, 60]. With this in mind, the simulation was modified so that each target site had its own associated parameters for imprecise excision events. These parameters are:

1) A ratio expressing the probability that each insertion will result in either:
   a) a perfect excision
   b) an imperfect excision with additional bases left in the host sequence
   c) an imperfect excision with bases removed from the host sequence
2) The number of bases to be inserted if an imperfect excision results in bases being left in the host sequence. This may be expressed as a range in which each number in the range has an equal probability of selection.

3) The number of bases to be deleted from the host sequence if an imperfect excision results in bases being removed from the host sequence. This may be expressed as a range in which each number in the range has an equal probability of selection.

The simulation was run using three target sites with different imprecise excision parameters for each.

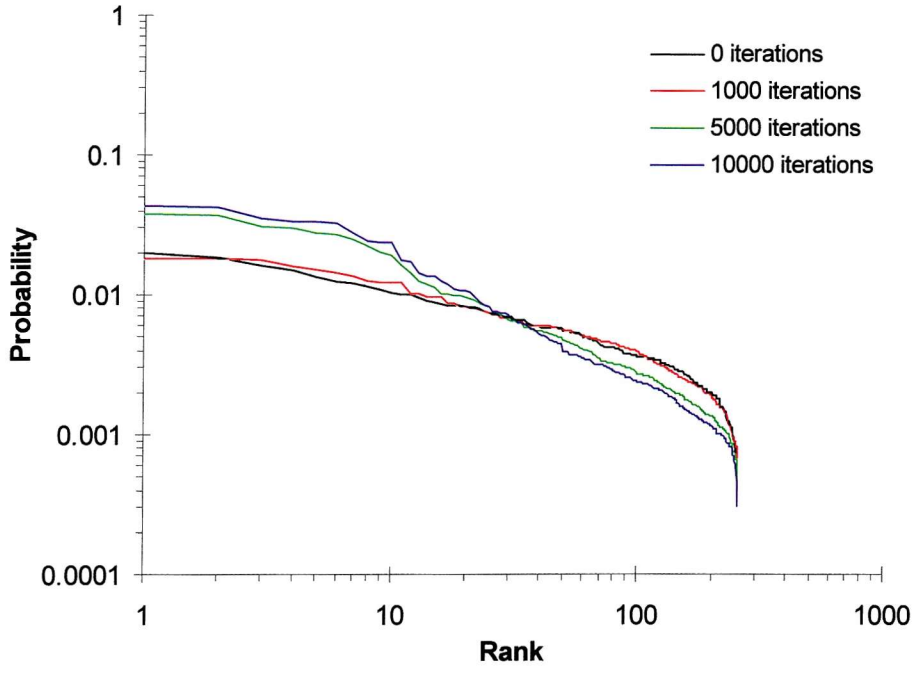


Figure 113 Zipf plots showing effect of target site specific imprecise excision parameters; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; ratios perfect:imperfect excision with insertion:imperfect excision with deletion – TATA: 1:3:3; TCAG: 1:1:1; GGAC: 0:1:1; number of bases inserted in imprecise excisions – TATA: 3; TCAG: 3; GGAC: 1-6; number of bases deleted in imprecise excisions – TATA: 3; TCAG: 3; GGAC: 1-6; analysed with word length 4

The Zipf results from the simulation are given in Figure 113. The plot shows a linear portion in the centre of the line (approximately rank 10 - rank 200) which tails off at either end. This profile of Zipf plot may have been obtained by combining the characteristics of the three Zipf plots obtained when using the parameters which are applied to each of the three different target sites in this simulation. For example, the flat portion at the start of the plot could be derived from the steps seen in Figure 96 for simulations using a perfect:imperfect with

insertions:imperfect with deletions ratio of 1:1:1 or 1:3:3 (the same parameters as used for two of the target sites, TATA and TCAG in this new simulation). The step may have been smoothed out because a variable number of bases were inserted/deleted during imperfect excisions for one of the target sites, combining the stepped Zipf results with the smooth line seen in Figure 102 (a simulation carried out using a variable number of bases inserted/deleted in imperfect excisions as used for target site GGAC in this simulation).

Figure 114 shows the results of the fluctuation analysis. Like previous results using imperfect excision, the depression in the lines is much reduced. The fluctuation results are very similar to those obtained when using a variable number of bases in imperfect excisions (Figure 103).

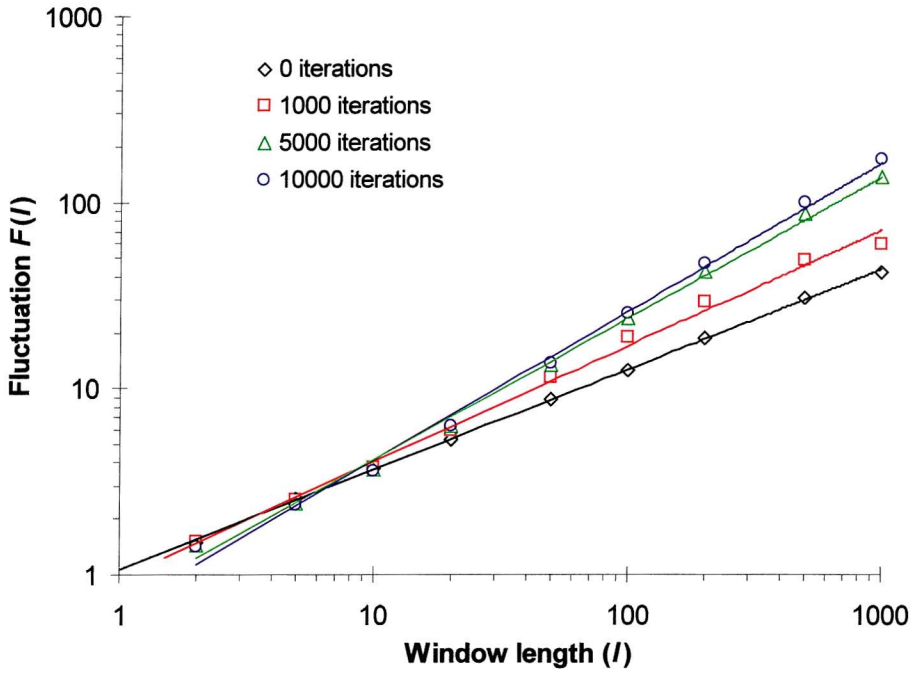


Figure 114 Fluctuation analysis showing effect of target site specific imprecise excision parameters; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; ratios perfect:imperfect excision with insertion:imperfect excision with deletion – TATA: 1:3:3; TCAG: 1:1:1; GGAC: 0:1:1; number of bases inserted in imprecise excisions – TATA: 3; TCAG: 3; GGAC: 1-6; number of bases deleted in imprecise excisions – TATA: 3; TCAG: 3; GGAC: 1-6

The variation of Shannon entropy with iterations for the simulation is shown in Figure 115. The plot shows that with different imprecise excision parameters for each site the final Shannon entropy value is lower than when the simulation is run with all sites using the same parameters. Like the Zipf plots, this could be explained by combining the results obtained from the simulations using the parameters applied to each target site. Figure 115 shows the Shannon entropy results obtained when all three target sites use the same parameters for imprecise excisions, alongside the results obtained for the new simulation. It can be seen that the results for the new simulation in which each site has its own parameters lie in between those achieved when the parameters were applied to all the target sites (the parameters are the same as those applied to the individual sites in the new simulation). The line for the new simulation appears to be the result of a combination of the results obtained using the parameters assigned for each target site. Note that Figure 97 shows that when the ratio of perfect:imperfect excision with insertion:imperfect excision with deletion is 1:1:1, the results are almost identical to those seen when the ratio is 1:3:3, so of the lines obtained using the parameters assigned to the three target sites, one lies above and two below. This would explain why the line for the new simulation appears nearer the line obtained when the ratio for perfect:imperfect excisions is 1:3:3.

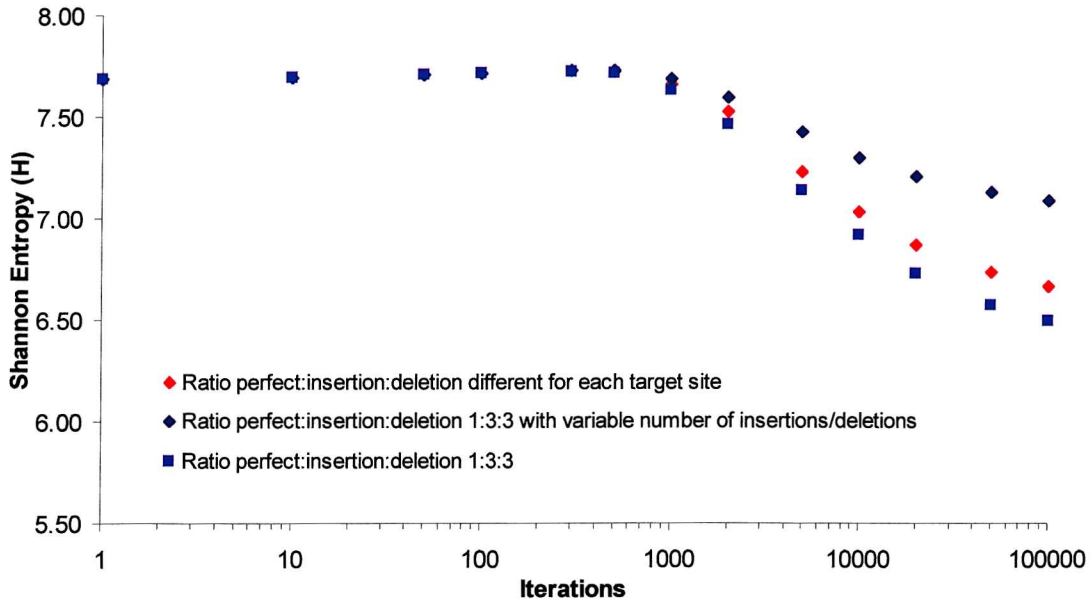


Figure 115 Variation of Shannon Entropy with iterations showing effect of target site specific imprecise excision parameters; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; A) simulation with different parameters for each site: ratios perfect:imperfect excision with insertion:imperfect excision with deletion – TATA: 1:3:3; TCAG: 1:1:1; GGAC: 0:1:1; number of bases inserted in imprecise excisions – TATA: 3; TCAG: 3; GGAC: 1-6; number of bases deleted in imprecise excisions – TATA: 3; TCAG: 3; GGAC: 1-6; B) simulation with variable number of insertions/deletions: ratios perfect:imperfect excision with insertion:imperfect excision with deletion 1:3:3; number of bases inserted in imprecise excisions 1-6; number of bases deleted in imperfect excision 1-6; C) simulation with non-variable number of insertions/deletions: ratios perfect:imperfect excision with insertion:imperfect excision with deletion 1:3:3; number of bases inserted in imprecise excisions 3; number of bases deleted in imperfect excision 3; analysed with word length 4

Figure 116 shows how the fluctuation exponent varies with iterations. The results show that there is little difference between the new simulation and the simulation in which all imperfect excision parameters are the same for each target site. Previous results show that the parameters assigned to the individual target sites, when used as the parameters for all target sites in a simulation, give fluctuation exponent vs. iterations curves that appear very similar (Figure 98 and Figure 105). With this in mind it is therefore not surprising that the two curves in Figure 116 are so similar.

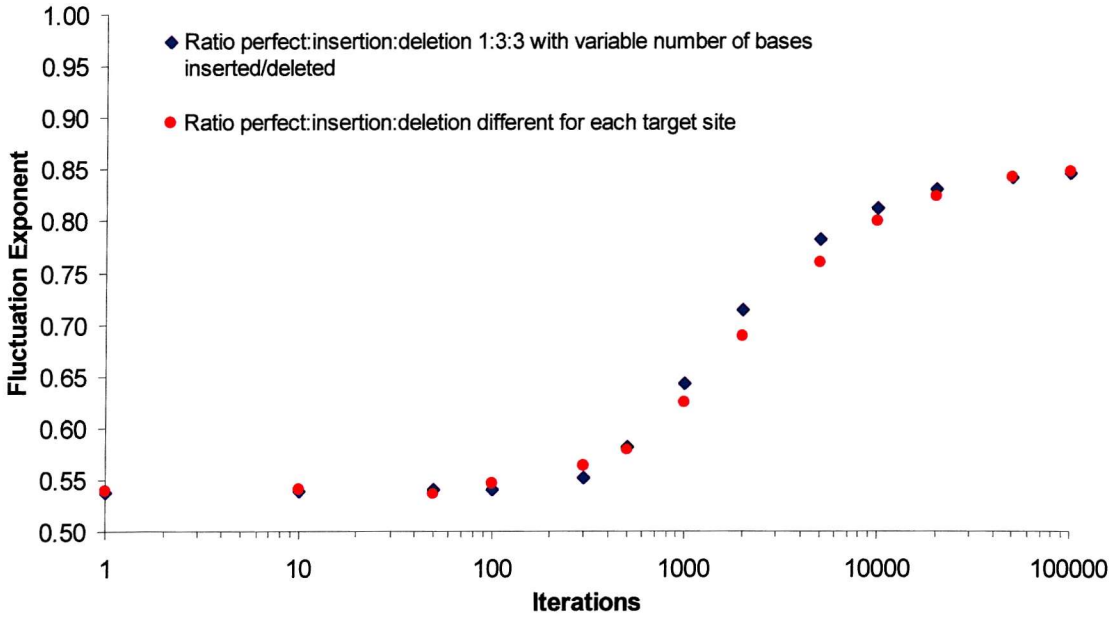


Figure 116 Variation of fluctuation exponent with iterations showing effect of target site specific imprecise excision parameters; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; A) simulation with different parameters for each site: ratios perfect:imperfect excision with insertion:imperfect excision with deletion – TATA: 1:3:3; TCAG: 1:1:1; GGAC: 0:1:1; number of bases inserted in imprecise excisions – TATA: 3; TCAG: 3; GGAC: 1-6; number of bases deleted in imprecise excisions – TATA: 3; TCAG: 3; GGAC: 1-6; B) simulation with same parameters for each site: ratios perfect:imperfect excision with insertion:imperfect excision with deletion 1:3:3; number of bases inserted in imprecise excisions 1-6; number of bases deleted in imperfect excision 1-6

7.3 THE EFFECT OF MODELLING TRANSPOSABLE ELEMENT END SEQUENCES

The details of the model dealing with imprecise excisions have been described earlier. One of the features of the model is that, when simulating an imprecise excision in which bases from the transposable element are left in the sequence, the actual bases to be inserted are chosen at random. This is not a particularly realistic solution, as the bases remaining are actually from the ends of the transposable element which has inserted at that target site. In order to make this aspect of the simulation more realistic, the program was altered to explicitly take account of the ends of the transposable elements.

The details of the modified simulation are as follows. For each target site, two sequences are required as parameters for the simulation, one representing the 5′ end of the transposable element, and another the 3′ end. When an imperfect insertion occurs in which bases from the transposable element are left in the host sequence (determined as normal, by the processes described earlier) the end of the transposable element to be left behind is determined at random (3′ end or 5′ end) each with an equal probability of being selected. From the end of the appropriate end sequence the required number of bases to be inserted are then copied and pasted into the host sequence.

The simulation was run using the transposable element end sequences (the end sequences were randomly generated before running the simulation and assigned to each target site - once assigned the sequences remain the same throughout the simulation. The sequences themselves are given in Figure 117). Three target sites were used and the perfect:imperfect excision with insertion:imperfect excision with deletion ratio was set at 1:3:3 for all sites. 1-6 bases were inserted and 1-6 bases were deleted in imperfect excisions.

The Zipf results for the simulation are given in Figure 117. The lines are linear over a large portion of the plot and do not show significant stepping. The plot appears similar to that seen in Figure 102 (the equivalent simulation, but bases inserted are random) although the line in Figure 117 is somewhat smoother and the gradients for the two plots are quite close also (0.58 for the new simulation compared with 0.60 when using random base insertions after 10,000 iterations - the basic model using perfect excisions only gave a gradient of 0.53 after 10,000 iterations).

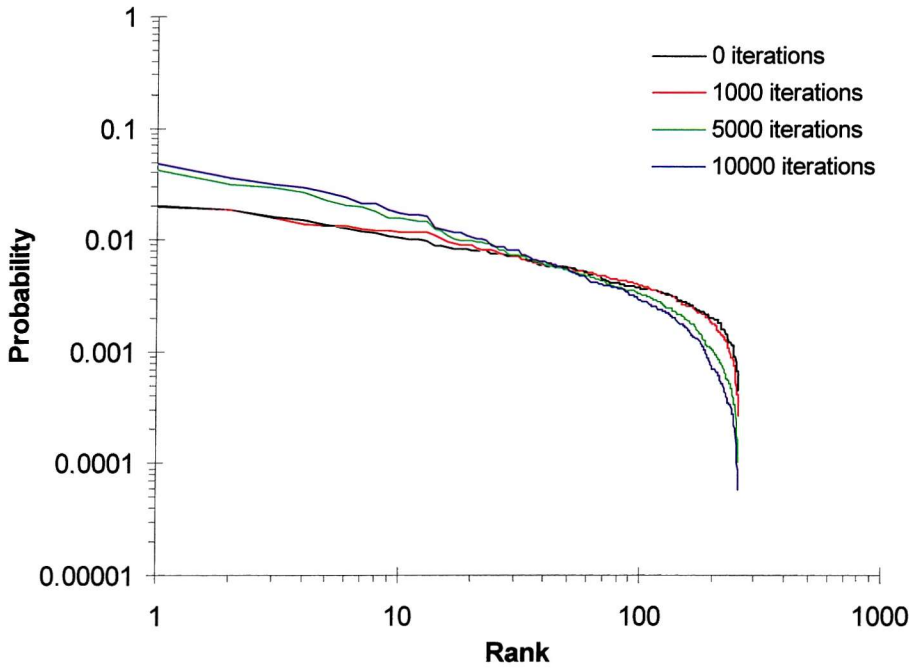


Figure 117 Zipf plots showing effect of explicit modelling of TE end sequences; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; ratio perfect excision:imperfect with insertion:imperfect with deletion 1:3:3; 1-6 bases inserted in imperfect insertions; 1-6 bases deleted in imperfect deletions; left TE sequences by site – TATA: AGAATTCGGA; TCAG: CAGGGAGATC; GGAC: CCTTGAGAGC; right TE sequences by site – TATA: CACCGTGCTT; TCAG: ACTAGCGCCA; GGAC: CCCACCGTTA; analysed with word length 4

The fluctuation results for the new simulation are given in Figure 118. As with other simulations using imperfect excisions, the depression in the fluctuation line is quite low in comparison with earlier results from simulations in which all excisions were perfect. A direct comparison between the simulation using random bases for insertion and that explicitly modelling the ends of the transposable elements is given in Figure 119. The plot shows that the gradient when modelling the transposable element sequences is slightly lower than when inserting random bases (0.75 compared with 0.81).

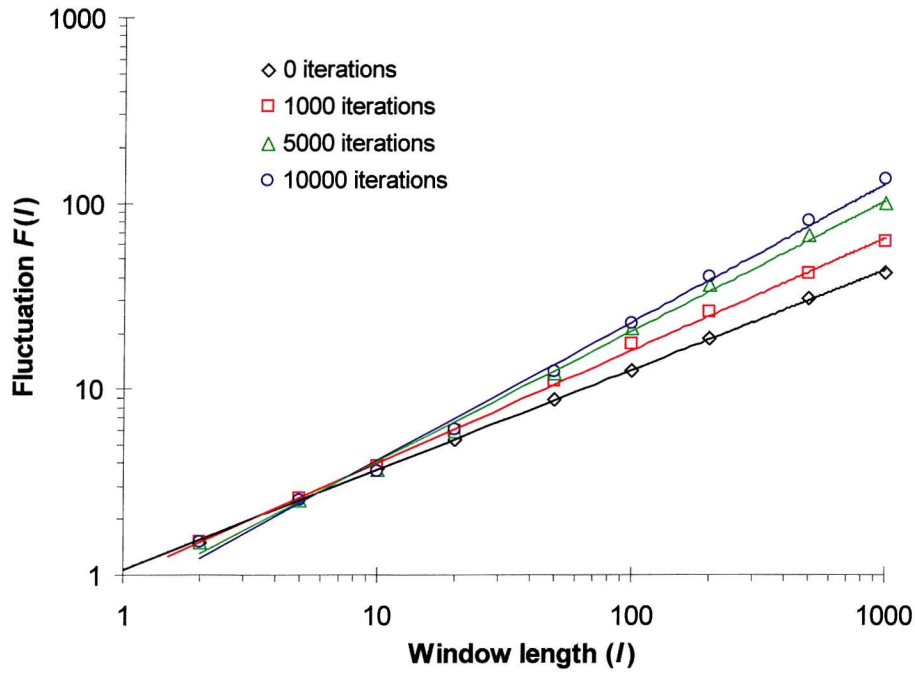


Figure 118 Fluctuation analysis showing effect of explicit modelling of TE end sequences; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; ratio perfect excision:imperfect with insertion:imperfect with deletion 1:3:3; 1-6 bases inserted in imperfect insertions; 1-6 bases deleted in imperfect deletions; left TE sequences by site – TATA: AGAATTCGGA; TCAG: CAGGGAGATC; GGAC: CCTTGAGAGC; right TE sequences by site – TATA: CACCGTGCTT; TCAG: ACTAGCGCCA; GGAC: CCCACCGTTA

The Shannon entropy vs. iterations results are shown in Figure 120. They show that with the transposable element sequences modelled, the Shannon entropy reaches a lower final value. This is due to the fact that by modelling the transposable element end sequences there is more order in the final sequence than if random bases are inserted. This order leads to a greater redundancy and therefore a lower value for the Shannon entropy.

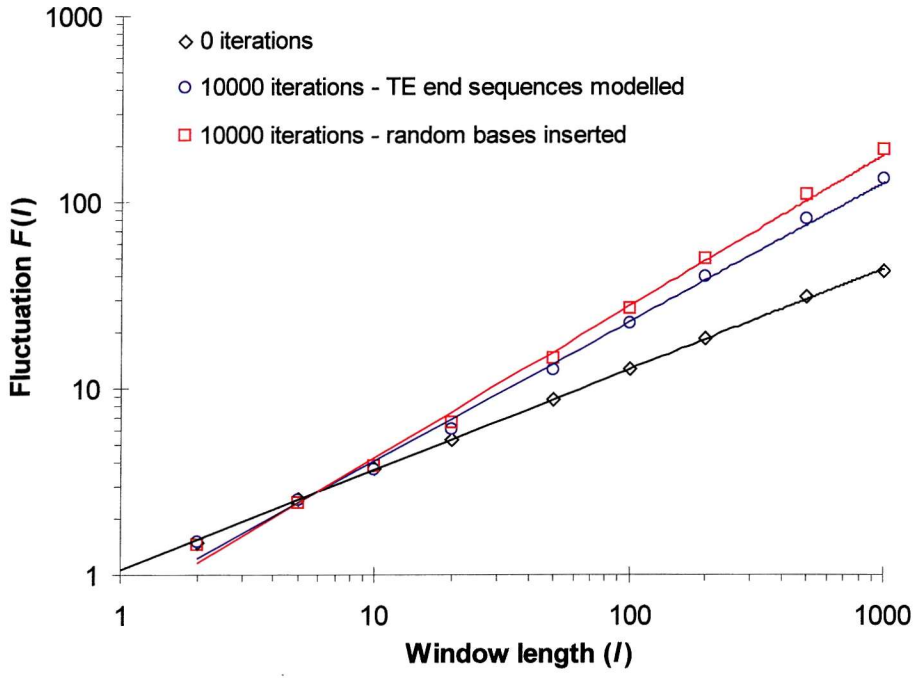


Figure 119 Fluctuation analysis showing effect of explicit modelling of TE end sequences; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; ratio perfect excision:imperfect with insertion:imperfect with deletion 1:3:3; 1-6 bases inserted in imperfect insertions; 1-6 bases deleted in imperfect deletions; for simulations with TE ends modelled – left TE sequences by site – TATA: AGAATTCGGA; TCAG: CAGGGAGATC; GGAC: CCTTGAGAGC; right TE sequences by site – TATA: CACCGTGCTT; TCAG: ACTAGCGCCA; GGAC: CCCACCGTTA

Figure 121 shows a comparison of the fluctuation exponent vs. iteration curves for sequences generated using transposable element end sequences and random base insertions during imperfect excisions. The plot confirms that earlier observation that the simulation in which the transposable element sequences are modelled results in sequences with a lower fluctuation exponent.

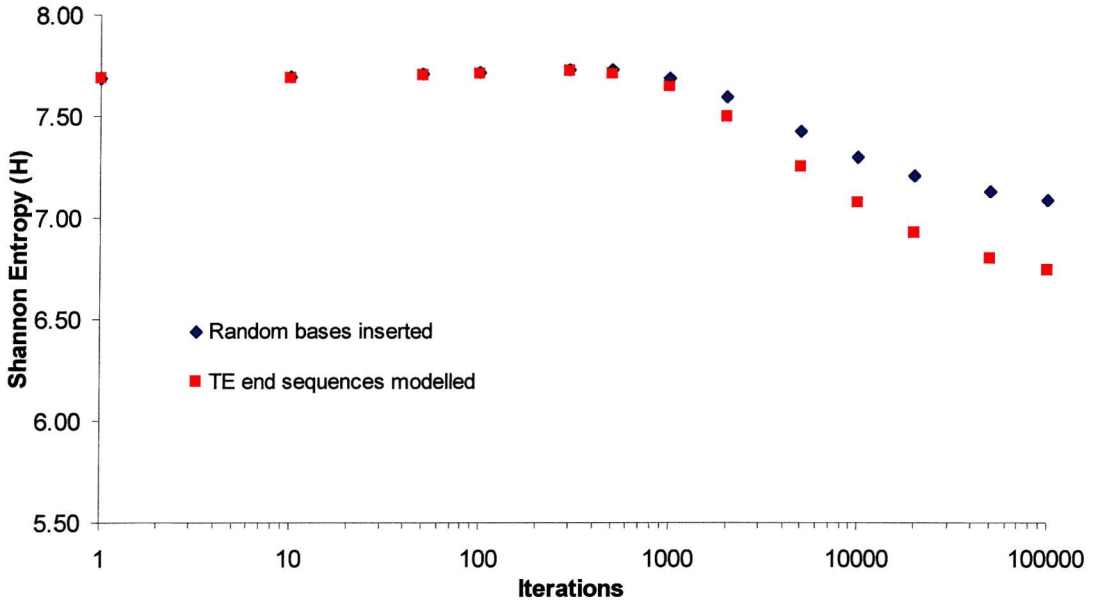


Figure 120 Variation of Shannon Entropy with iterations showing effect of explicit modelling of TE end sequences; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; ratio perfect excision:imperfect with insertion:imperfect with deletion 1:3:3; 1-6 bases inserted in imperfect insertions; 1-6 bases deleted in imperfect deletions; for simulation with TE ends modelled – left TE sequences by site – TATA: AGAATTCGGA; TCAG: CAGGGAGATC; GGAC: CCTTGAGAGC; right TE sequences by site – TATA: CACCGTGCTT; TCAG: ACTAGCGCCA; GGAC: CCCACCGTTA; analysed with word length 4

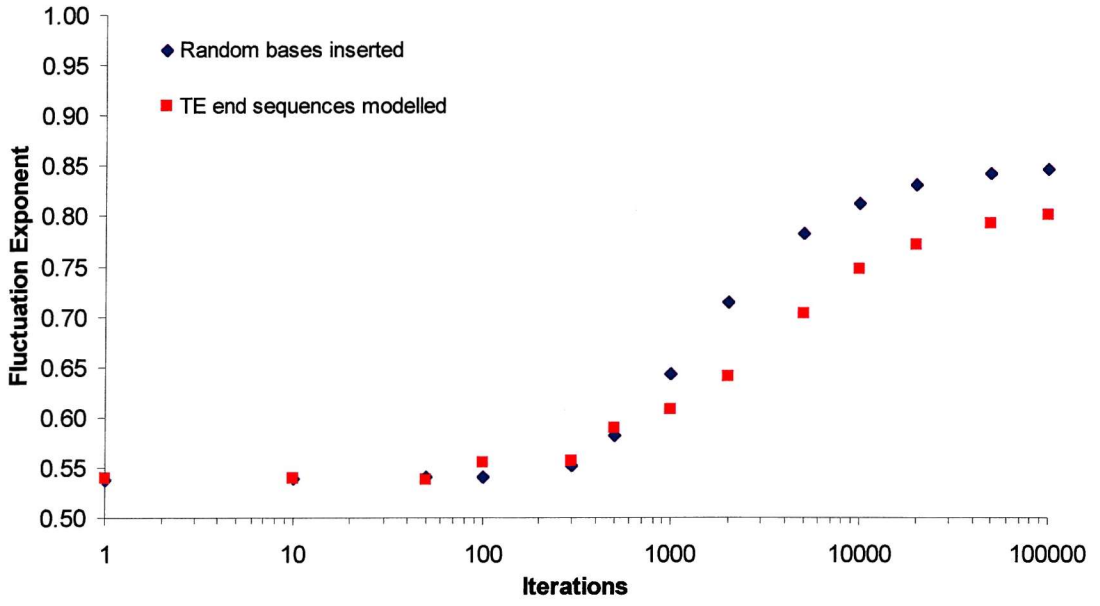


Figure 121 Variation of fluctuation exponent with iterations showing effect of explicit modelling of TE end sequences; insertion into HTLV II; target sites TATA, TCAG, GGAC; site selection probabilities 1:1:1 respectively; copy length 6; ratio perfect excision:imperfect with insertion:imperfect with deletion 1:3:3; 1-6 bases inserted in imperfect insertions; 1-6 bases deleted in imperfect deletions; for simulation with TE ends modelled – left TE sequences by site – TATA: AGAATTCGGA; TCAG: CAGGGAGATC; GGAC: CCTTGAGAGC; right TE sequences by site – TATA: CACCGTGCTT; TCAG: ACTAGCGCCA; GGAC: CCCACCGTTA

8 Final Results and Conclusions

The results shown in previous chapters have illustrated the effects of various genome reshaping processes on the language-like character of a DNA sequence, as measured using the Zipf plot, fluctuation analysis and Shannon entropy. The parameters associated with each modification of the simulation have been applied independently of each other to allow a clear analysis of the effects of each parameter. In reality, these mutational events, which have here been modelled in independent simulations, would be combined in a natural system. In this section a simulation is performed using a combination of the parameters which have been described previously, in order to accurately model the origin of non-coding DNA in a natural system.

This new combined-parameter simulation used three target sites, all of length 3bp and each using a copy length of 6bp. There was a bias towards transposable element insertion at certain sites. Imperfect insertion/excision events were modelled, with different probabilities of excision with deletion of bases or excision with insertion of bases at each site. The transposable element ends were explicitly modelled. Point mutations were included in the simulation.

Figure 122 shows the Zipf plot for the combined simulation (note that due to the use of point mutations the number of simulation iterations has been increased to remain comparable with earlier Zipf plots). The plot after 13,333 iterations shows a very smooth line which appears straight over a large portion. The gradient after 13,333 iterations is 0.49. This corresponds well with the gradients obtained from real non-coding DNA sequences as shown in Table 3. E.g. The non-coding regions of *C. elegans* and the "other invertebrates" group gave Zipf exponents of 0.537 and 0.477 respectively.

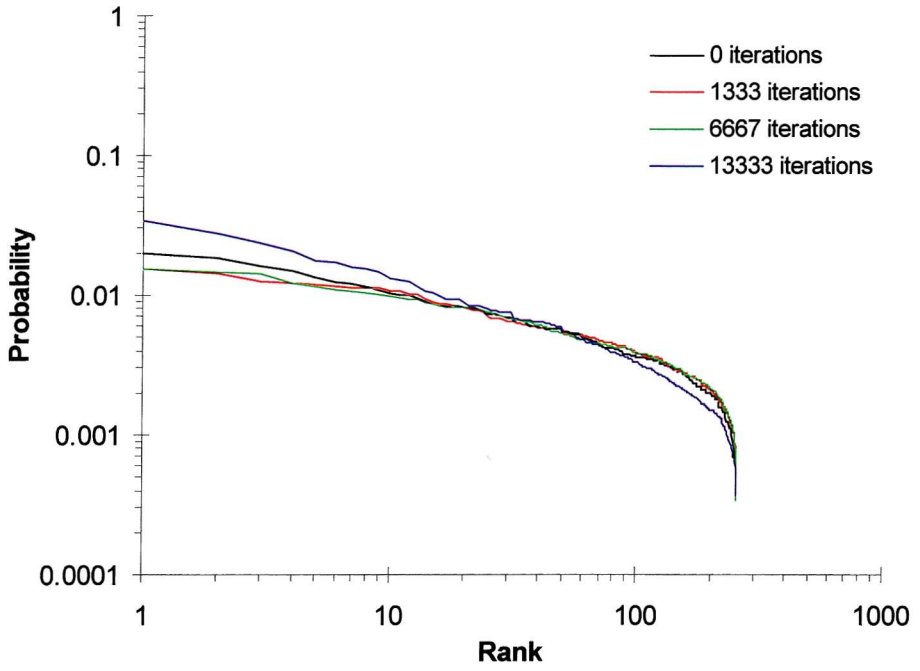


Figure 122 Zipf plots showing effect of combined insertion/excision parameters; insertion into HTLV II; point mutations:insertion/excisions ratio 1:3; transition:transversion ratio 2:1; target sites TAT, TCA, GGA; site selection probabilities 1:3:6 respectively; copy length 6; ratios perfect:imperfect excision with insertion:imperfect excision with deletion – TAT: 1:3:3; TCA: 1:1:1; GGA: 0:1:1; number of bases inserted in imprecise excisions – TAT: 2-4; TCA: 2-4; GGA: 1-6; number of bases deleted in imprecise excisions – TAT: 2-4; TCA: 2-4; GGA: 1-6; left TE sequences by site – TAT: AGAATTCGGA; TCA: CAGGGAGATC; GGA: CCTTGAGAGC; right TE sequences by site – TAT: CACCGTGCTT; TCA: ACTAGCGCCA; GGA: CCCACCGTTA; analysed with word length 4

The fluctuation results for the simulation are given in Figure 123. The lines clearly show an increase in gradient as the number of iterations are increased. The depression seen in other fluctuation results at around window length 6 is very small in this case. A comparison with a fluctuation plot obtained for real DNA (Figure 126) shows a good similarity in the quality of the plots with both lines showing slight curvature. The residuals obtained by subtracting the linear fit from the fluctuation data for both the real and simulated DNA sequences are given in Figure 124. These show that while the magnitude of the depression is similar in both cases, the depression seen using the real DNA sequence is positioned at a higher window length (around window length 40 compared with window length 10 for the simulated sequence). It may be possible to reposition the depression by using longer target sites in any future simulations.

The fluctuation gradient observed after 13,333 iterations is 0.80. The observed exponent is large compared with the data obtained from real sequences (Table 2) of which the highest gradient was 0.71 for human β-globin. The variation of the fluctuation exponent with iterations is shown in Figure 127. The fluctuation data follow the characteristic sigmoidal curve seen in earlier simulations.

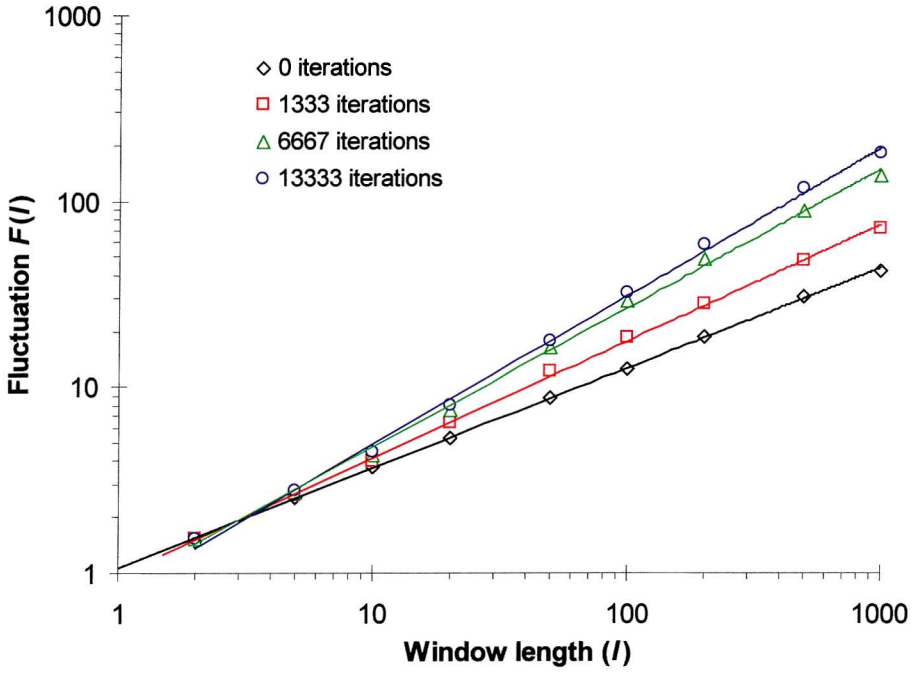


Figure 123 Fluctuation analysis showing effect of combined insertion/excision parameters; insertion into HTLV II; point mutations:insertion/excisions ratio 1:3; transition:transversion ratio 2:1; target sites TAT, TCA, GGA; site selection probabilities 1:3:6 respectively; copy length 6; ratios perfect:imperfect excision with insertion:imperfect excision with deletion – TAT: 1:3:3; TCA: 1:1:1; GGA: 0:1:1; number of bases inserted in imprecise excisions – TAT: 2-4; TCA: 2-4; GGA: 1-6; number of bases deleted in imprecise excisions – TAT: 2-4; TCA: 2-4; GGA: 1-6; left TE sequences by site – TAT: AGAATTCGGA; TCA: CAGGGAGATC; GGA: CCTTGAGAGC; right TE sequences by site – TAT: CACCGTGCTT; TCA: ACTAGCGCCA; GGA: CCCACCGTTA

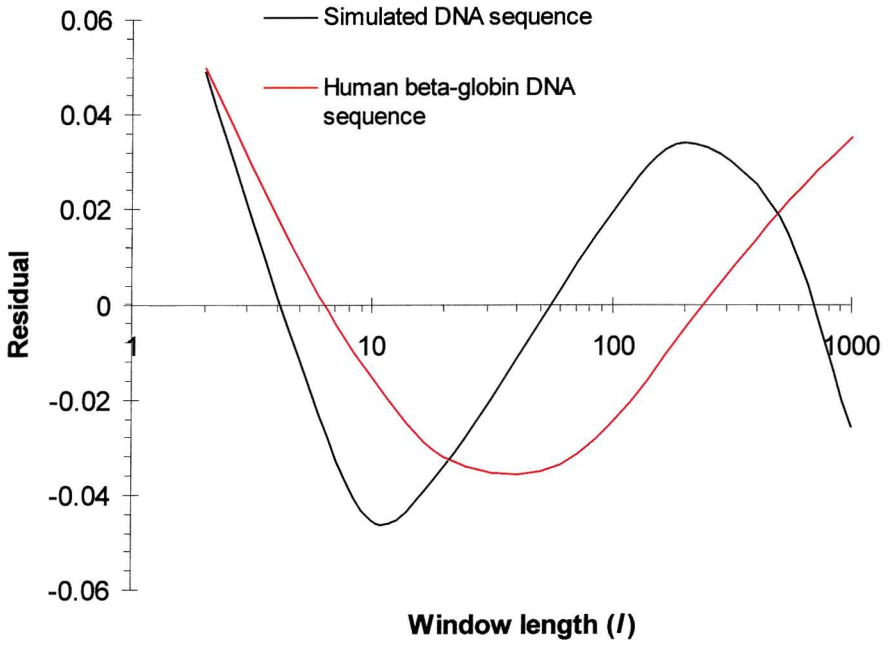


Figure 124 Comparison of the fluctuation residuals of real and simulated DNA sequences. Real sequence is the entire human beta-globin intergenomic sequence. Simulation details for artificial sequence: 13333 insertions into HTL VII; point mutations:insertion/excisions ratio 1:3; transition:transversion ratio 2:1; target sites TAT, TCA, GGA; site selection probabilities 1:3:6 respectively; copy length 6; ratios perfect:imperfect excision with insertion:imperfect excision with deletion – TAT: 1:3:3; TCA: 1:1:1; GGA: 0:1:1; number of bases inserted in imprecise excisions – TAT: 2-4; TCA: 2-4; GGAC: 1-6; number of bases deleted in imprecise excisions – TAT: 2-4; TCA: 2-4; GGA: 1-6; left TE sequences by site – TAT: AGAATTCGGA; TCA: CAGGGAGATC; GGA: CCTTGAGAGC; right TE sequences by site – TAT: CACCGTGCTT; TCA: ACTAGCGCCA; GGA: CCCACCGTTA

The variation of Shannon entropy with iterations is shown in Figure 125. The Shannon entropy plot is quite different to those seen previously. The plot follows the same general trend of a high Shannon entropy value at low iterations and a lower value at higher iterations, but the difference between the two levels is quite small. The starting value after 1 iteration is 7.69 changing to 7.28 after 100,000 insertion/excision events. This compares with the example shown in Figure 16 for the initial simulation (run with three target sites, no point mutations and all excisions perfect) in which the Shannon entropy drops from 7.69 to 6.29. The small drop obtained from the new simulation suggests that the redundancy of the sequences changes little – the sequence does not become much more ordered as the iterations increase. A similar result was seen in

previous simulations in which the sequence remains random in nature (e.g. when the target sites used were 1bp long (Figure 63) and when the copy length was 50bp (Figure 70)). The relatively high values of the Shannon entropy seen with the latest simulation are actually much closer to those seen for real non-coding DNA sequences, for example, the non-coding region of *C. elegans* has a Shannon entropy value of 7.552 - this is close to the value 7.47 obtained for the simulated DNA after 10,000 insertion/excision events.

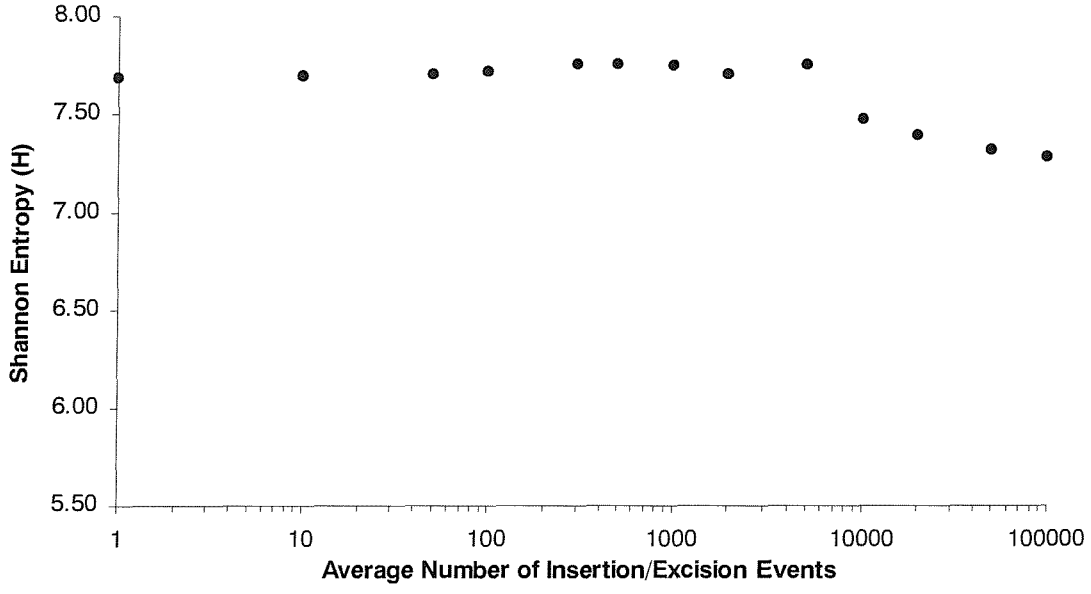


Figure 125 Variation of Shannon Entropy with iterations showing effect of combined insertion/excision parameters; insertion into HTLV II; point mutations:insertion/excisions ratio 1:3; transition:transversion ratio 2:1; target sites TAT, TCA, GGA; site selection probabilities 1:3:6 respectively; copy length 6; ratios perfect:imperfect excision with insertion:imperfect excision with deletion – TAT: 1:3:3; TCA: 1:1:1; GGA: 0:1:1; number of bases inserted in imprecise excisions – TAT: 2-4; TCA: 2-4; GGA: 1-6; number of bases deleted in imprecise excisions – TAT: 2-4; TCA: 2-4; GGA: 1-6; left TE sequences by site – TAT: AGAATTCGGA; TCA: CAGGGAGATC; GGA: CCTTGAGAGC; right TE sequences by site – TAT: CACCGTGCTT; TCA: ACTAGCGCCA; GGA: CCCACCGTTA; analysed with word length 4

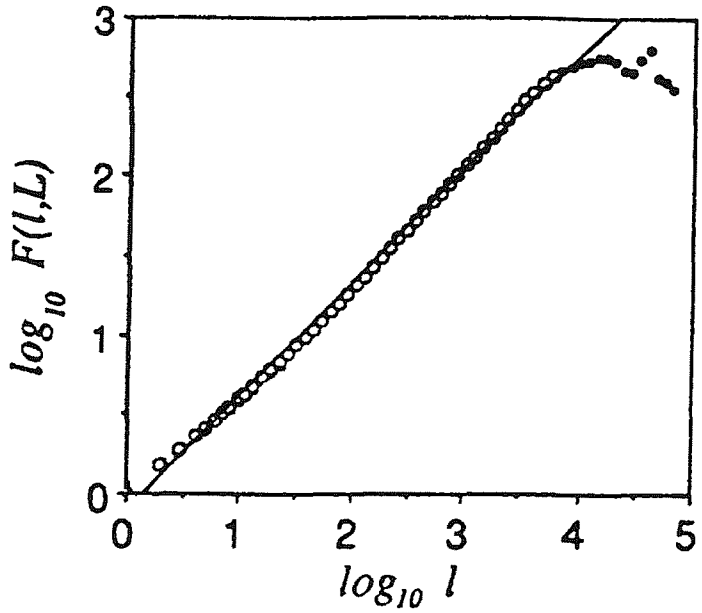


Figure 126 Fluctuation plot of a non-coding DNA sequence (adapted from reference 33). Plot of the entire human beta-globin intergenomic sequence (HUMHBB) (non-coding). The slope of the linear fit (omitting solid circles) is 0.71.

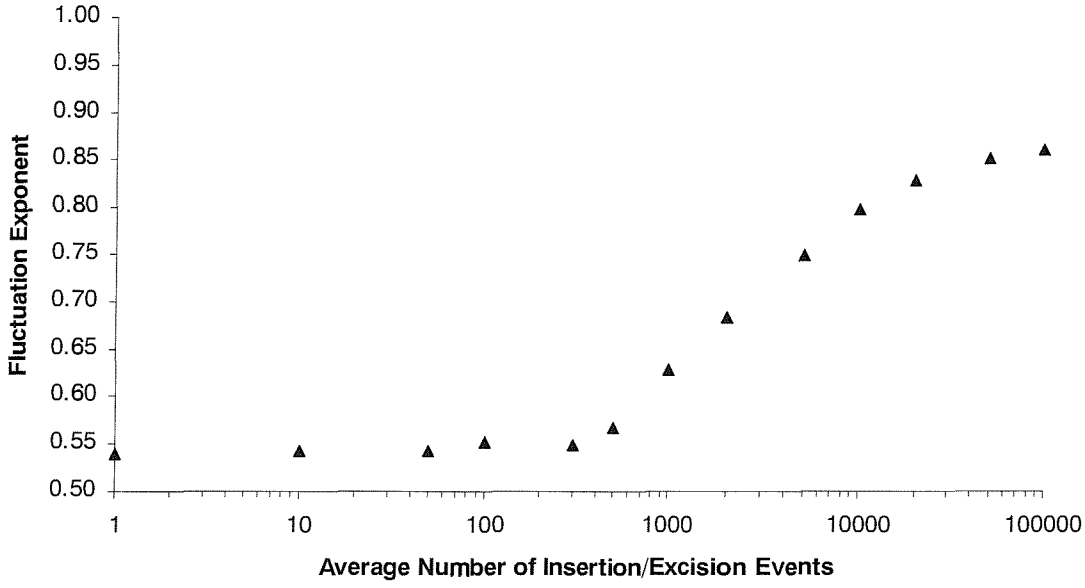


Figure 127 Variation of fluctuation exponent with iterations showing effect of combined insertion/excision parameters; insertion into HTLV II; point mutations:insertion/excisions ratio 1:3; transition:transversion ratio 2:1; target sites TAT, TCA, GGA; site selection probabilities 1:3:6 respectively; copy length 6; ratios perfect:imperfect excision with insertion:imperfect excision with deletion – TAT: 1:3:3; TCA: 1:1:1; GGA: 0:1:1; number of bases inserted in imprecise excisions – TAT: 2-4; TCA: 2-4; GGA: 1-6; number of bases deleted in imprecise excisions – TAT: 2-4; TCA: 2-4; GGA: 1-6; left TE sequences by site – TAT: AGAATTCGGA; TCA: CAGGGAGATC; GGA: CCTTGAGAGC; right TE sequences by site – TAT: CACCGTGCTT; TCA: ACTAGCGCCA; GGA: CCCACCGTTA

A section of the sequence obtained after 6,667 iterations of the simulation is given in Figure 128. The sequence is notable as the blocky nature of the sequences generated by earlier simulations is absent. There is a striking difference seen when comparing this section of sequence with that produced by the first simulation (Figure 41). A real human DNA sequence is shown in Figure 129 for comparison. With the repeat units highlighted, the real sequence shows the same kind of imperfect repetitive sequence as produced by the simulation. The sequence obtained from the new simulation is far more realistic in appearance than that presented in Figure 41.

These results show that combining the various parameters of the simulation into a realistic model produces realistic results, mimicking the data seen for real non-coding DNA sequences. The results presented both here and in previous sections, have shown that the transposable element model, simulating the repeated insertion and excision of transposable elements, can account for the formation of language-like features found in non-coding DNA sequences, starting with a sequence that shows no such features. The basic model using simple target sites and performing perfect excisions showed that the basic features of non-coding DNA could be replicated - the increase in gradient of the Zipf and fluctuation plots. This is achieved by the generation of blocks of repeating units, at various points in the host sequence. These repeating units are amplified, generating a bias towards certain sequences, or "words" - this bias is expressed in the Zipf plot as an increase in gradient when iterations of the simulation are performed. The formation of the blocks generates a high order structure within the sequence, and this gives rise to a fluctuation exponent, $\alpha > \frac{1}{2}$. The addition of detail to the simulation and the inclusion of other genome reshaping processes has refined the results, making the blocks appear more realistic and reducing the anomaly seen in the fluctuation data, for example. The results obtained by combining the different parameters of the simulation are very similar to those seen for real non-coding DNA sequences.

TGACAATGGCGACTAGCCTGCCAAGCCAAGCACCCAGGGCGAGGCAGAGTAGTCAGTCAG
TCACAGGCAGTCACCACAGTCACAGGCAGTCATCGACCCAAAAGGTGGTCAGACCGTTTC
ACAGTTTCACAGGGTTTTTCACAGTTTCAGTTTCAGTTTCACAGGGTTTCAGTTTCTTTC
ACAGTTTCAGTTTTACCAGTTTCAGTTTCAGTTTCTTTTTTCACAGGTTTCACAGTTTCA
CACAAACAATCTCAAGTAAAGGCTCCGACGGCATCCATCATCACCTACTATTACTATCTT
TAGAATAAGAGGACCTTGAGAGGACCTTTAAGAAGGAAGGAGGACCATAGGTAGTAGGAC
CTGTAGGATAGTAGGACGTAGGACTCCGAATAAGAATTAGAACGATACTAAGGACCTTGT
AAGGAATTAGTAGGACTTTTTAGGGACCTTGGGGGGGATAGGATCTTGAATAGTAGGTAG
GACCTGTAGGACCTATAGGACTTTGATTAGGGGGTGACTGAAACCACGGCCCTGGCGTCC
CTCCCCCCTAGTAGGAAGGACCATAGGAGTTACTAGAGGGGGAATTAAGAGAGGGGAGGG
ACGTGTGGACCTTGTGGACCTGTGGAGAGGGGAAAGGGGAGAGGAGGATAAGAGGACGTT
AAGAGGGAAAGAGGATTAAGAGGACCTAGAGGATAAGAGGATAAGAGGAACACGAACAGC
TCTCCGCAGAAAAATAGCTCTCCCTCCCTCACAGACCTCACAGGCCCTCACCCTCACACC
CTCAGCCACCCTCACCCTCTCACCCTTACCCACTTCCCCTAGCGCTGAAAAACAAGGCGC
TGACGATTACCCCCTGCCCATAAAATTTGCCTTGTGTCAGTGTCAGTGTCAAATAAAGG
ATGCCGAGTCTGTAAAATCGCAAGAGGACCTTAGAGGAGAGAGGACCTTGAAGAGGACAG
TACGGGGATGTTGCAGGGAGTTACAGGGATACAGGGAACCGGGGGGGGACGGGGGATAGGG
GGACCGGGGGGGGGATCGTTGTAGACGCTATTAGCGAATTAGGAGTTATGCTTAGTTAGT
TATTAGGACCTGTTAGAACGTTAGCAGAAGGCGGCTCGCTCCCTCACCCTCACAGGCCCT
CACCCTCACCGACCCTCTGGTGTTGTCCGGCGGGCGGGCGGATACGCGGCGGGACCTGCG

Figure 128 Sequence generated using combined insertion/excision parameters. Colour coding helps to highlight the sequences produced by target site duplication. The first 1200 bases of a sequence generated after 6667 iterations; starting sequence HTLV II; point mutations:insertion/excisions ratio 1:3; transition:transversion ratio 2:1; target sites TAT, TCA, GGA; site selection probabilities 1:3:6 respectively; copy length 6; ratios perfect:imperfect excision with insertion:imperfect excision with deletion – TAT: 1:3:3; TCA: 1:1:1; GGA: 0:1:1; number of bases inserted in imprecise excisions – TAT: 2-4; TCA: 2-4; GGA: 1-6; number of bases deleted in imprecise excisions – TAT: 2-4; TCA: 2-4; GGA: 1-6; left TE sequences by site – TAT: AGAATTCGGA; TCA: CAGGGAGATC; GGA: CCTTGAGAGC; right TE sequences by site – TAT: CACCGTGCTT; TCA: ACTAGCGCCA; GGA: CCCACCGTTA

TTATCCACCCATCCATCAACCCATCCATCCACCCATTCATCCACCATCCACCCATCCATT
CATCCATCCATCATCTACTCATCCATCAATCCATGCAGCTAGCCATGCATAATCCACCCA
TCTATCAATCCATTCATCATCCACCCATCCATCAATCCATCCATCCATCCATCATCCAGT
CAACCATTAATCCATCATCCCCACTCCATATATCTATCTTCCACTCATCCATCCATCATT
CACTCATCAATCAATCCATCCATTCACCCATCCATCAATCCATCTCTCCCTCATCCATCA
ATTCATCATCCACCCTTTCACCCATCCCTCCATCCATCCATCCATCCATGCATCCCTCCA
TCCATCTATTCATCCACCTATCCATTCATCCATCATCTACCCATCCATCAATCCATTTAT
CAGTCCATCCATGAAGCCATCCTCTTCCATCCATCCCTTCACCATTTGCCCACCTGCCCA
TCCCTCCATCCCTTCACCCATCCATCATTCACGCATCCACCAATCCATCTATCTATTCAT
CCCTTCACCTACCCATCCATCCCTCCATCCCTATACCCATCCATCCATCCATCCATCCAT
TTGTCCTCTACCCACCCACCCATCTGTCATCCACCCACCTATCCATCTTTTCATCCATCC
ATTCATTATCCACACACACACCTGTCTGTCATCCACCTACCCATTCATCTTTTCATCCAT
CCATCCATTGATCTATTCATCCATCCATCCAAGGGGGATTTATGGAGCTCCTGCTTCCTG
ATTGCTCTGTTGTGTACCTGAGGCCCTGCCTCTACAGCTCAGCAGTCTTGTCCTGTATCA
TGTGATCATAAAGGTAAGGGGGCAGAGGAAAGAAGGCAAGCATGGTTAGGGGATCGAGGG
TGGAGTGAGAAACTAAAACCTAGAAAGCTCAGGGGACCCCCACTTGTCACACCAGCTCTG
GCCACTCTCATTTTCAGTTATAGCAGATTAAGCTGCACACACCGGATAGACAGATACACA
GCCAGATACAGACTGACATAAAGATGGAGGTAGATGGATCCACAGATAATAGACAGATAG
ATACAGAGATCAGTGAGTAACAGATAGATAAGAGATAAATATAGATGATAAAAAGATGAT
AGATGGATGAATGATAAAGGGTTAGACAGGTGGATATGCAAACTTTTATACCTACAATTC

Figure 129 Human DNA sequence from chromosome 22 for comparison with artificial DNA. Bases 35241-36440 of GenBank sequence Z94162. Repeat unit ATCC highlighted in red.

The main stages in the development of the simulation were:

1) The original model using three target sites, each of which could be assigned a target site bias.
2) The ability to assign to each target site an individual copy length. This resulted in a higher Zipf exponent when copy lengths 2, 6 and 10 were used for the three target sites. The use of long copy lengths (e.g. 50) lead to a low Zipf exponent, the dip in the fluctuation line being moved to a higher window length value and a high final value for the Shannon entropy.
3) The ability to use more than three target sites each of a different length. The use of more target sites appeared to lower the final value for the Shannon

entropy. Long target site lengths (e.g. 6bp) lead to a large step in the Zipf plot and a large dip in the fluctuation line. Short target site lengths (e.g. 2bp) lead to low Zipf exponents but good quality fluctuation lines with a high fluctuation exponent. The final Shannon entropy value was much lower when long target site lengths were used.

4) The addition of point mutations to the model. Of the two types of mutation, transversions had the greater effect, lowering both the Zipf and fluctuation exponents. Transitions had no effect on the fluctuation data, but lowered the Zipf exponent slightly. The combined use of both types of point mutations lowered the final value of the Shannon entropy.
5) The modelling of imprecision in the excision of transposable elements. This greatly reduced the depression seen in the fluctuation analysis, but lead to a flat region at the start of the Zipf plot. The Shannon entropy was higher when imprecise excisions were modelled. It was found that deletions in imperfect excisions were responsible for the step in the Zipf plot and the improved quality of the fluctuation analysis. Using a range of values for the number of bases inserted/deleted smoothed out the step in the Zipf plot, but resulted in a lower final Shannon entropy value.
6) The ability to assign a range of copy lengths to each target site. The use of a copy length range 2-10 lead to a good quality Zipf plot with a high exponent, but a fluctuation line with low gradient and a large depression. The Shannon entropy was lowered when using a range of copy lengths.
7) The ability to assign imperfect excision parameters individually to each target site. The Zipf, fluctuation and Shannon entropy results were dependant on the individual parameters for each site, being an amalgamation of the characteristics of each set of parameters.
8) The explicit modelling of the end sequences of the transposable elements. This resulted in a smooth Zipf plot with exponent very close to that obtained with random insertion of bases. The fluctuation results gave a slightly higher exponent while the final Shannon entropy value was lower.

The final simulation incorporates, in detail, many of the reshaping mechanisms seen in real DNA sequences, as summarised above. There are two notable mutations that have not been included; these are slippage and recombination. Slippage is the mechanism by which repetitive units of DNA (in a repetitive sequence) may be duplicated or deleted thereby leading to expansion or contraction of the repetitive block. This was not incorporated into the model as the action of transposable element insertions already simulates this process - if the number of iterations of the simulation is increased or the bias towards a particular target site increased then certain repetitive units will be duplicated. Running for fewer iterations or adjusting the bias against a target site will reduce the number of repetitive units for that site. Recombination occurs between homologous sequences, usually either between sequences on sister chromosomes, or between transposable elements on the same chromosome. Recombination events generally affect long sections of the DNA while the simulation deals with relatively short sequences of DNA and does not consider other chromosomes. A recombination event involving another chromosome would simply replace all or part of the sequence with a corresponding sequence from the other chromosome; the replacement sequence would have undergone the same transposable element insertion/excisions so there is no reason to suppose the final results would differ greatly due to the recombination event. A recombination involving transposable elements on the same chromosome would lead to an inversion or a deletion of part of the sequence. Due to the scale of the mutation resulting from such a recombination, it is possible that a deletion or inversion could well affect the entire sequence. Deletion of the entire sequence is obviously not desirable and inversion of the entire sequence would not affect the statistical results. If intra-chromosomal recombination events mediated by transposable elements were to be modelled it would also become necessary to explicitly track the progress of the transposable elements as a recombination event would only occur when two transposable elements (of the same type) were present in the host sequence at the same time. In the current simulation there is no consideration given to the transposable element sequences themselves (other

than modelling their end sequences) - they are assumed to insert and immediately excise. This aspect would have to be simulated in more detail to allow for intra-chromosomal recombination events. Given the likely limited impact on the results and additional complexity required, recombination was not included in the model.

One point noted as mutational processes were added to the model was the fact that changes which improved the quality of one result might adversely affect other results. A good example of this was when the simulation was run with a short target site length (2 bp). The fluctuation results showed a good quality line with a high exponent, but the Zipf plot showed no increase in gradient and the Shannon entropy results showed a low decrease from the initial to the final value. In order for a simulation to achieve a realistic result in each statistical test it is necessary to balance the various parameters against each other. This is analogous to the situation in the natural system being modelled, in which a number of mutational processes are in action - the overall result being a combination of these processes.

8.1 FUTURE WORK

There are some features which could be added to the model. Possible modifications are:

1) Explicit modelling of transposable element insertion and excision. Transposable elements are actually inserted and removed from the host sequence. This would lead to final sequences in which transposable elements were present. The bases left behind on imperfect excision would be determined by the sequence of the transposable element.
2) Imperfect target site recognition. A given transposable element will insert at target sites that are not identical to one another. Each site will be similar to the others, but might have several bases different from the 'perfect' target site. The 'perfect' target site is known as the consensus sequence, and is

determined by 'averaging' the target sites at which insertions are observed. Generally the closer a target site is to the consensus sequence, the higher the probability of insertion. Modelling of this process would introduce additional variation into the simulation as the target sites duplicated on transposable element insertion would not be identical.

3) Slippage. This is the process by which repetitive sequences grow or shrink by addition or removal of repeats from the sequence. This occurs due to the mismatching of DNA strands during replication. Modelling of this action is potentially relevant, as the DNA sequences produced by the model contain many repeated sequences.
4) Recombination. This process leads to large-scale changes in the DNA structure. Recombination between sister chromatids during meiosis results in the exchange of large portions of the chromosomes. Intra-chromosomal exchange (mediated by transposable elements) can result in deletion or inversion of long sequences of DNA. Addition to the model would require the explicit modelling of transposable element movements (in 1) above) and the simulation of a sister chromatid.

No model can be a perfect representation of the real situation, particularly when the subject of the simulation is as complex as it is in this case. The potential benefits of additional features that might be added to the program (with each benefit being less useful than the previous as the model becomes more realistic) must be considered against the time required to implement the features and the possibility of generating unnecessary complexity. In this case the refined model appears to generate 'non-coding' DNA which is close to that seen in nature and it therefore would seem that there is little point in making any further major alterations. The reasons for leaving slippage and recombination out of the model have already been discussed. Imperfect target site recognition would add more variation into the sequence, but with the model in its present state there are already a number of features which achieve this. The impact of imperfect target site recognition is likely to be low. The explicit modelling of transposable

element movements is not practical due to the processing overhead required. Assuming insertion and excision rates are matched so there is no net build up of transposable elements, the actual number of transposable elements in the final sequence will be low. These transposable elements are likely to 'dilute' the language-like character of the sequences, rather than add or enhance any of the linguistic features seen, as their small number would not be sufficient to form a higher order structure. Again, considering the probable low impact and the additional complexity required (coupled with far longer running times for the simulation) it is reasonable not to include explicit transposable element modelling in the simulation.

Throughout the project the 'blockiness' of the simulated DNA sequences was assessed by visual comparison with real DNA. A more rigorous comparison could be obtained by mapping the DNA sequence to a numerical sequence and then applying a Fourier transform to the numerical sequence[69-71]. This analysis results in a spectrum in which periodic repeats are represented by peaks in the spectrum. The position of the peaks indicates the spacing of the repeats in the sequence, with a peak at frequency f indicating a period $1/f$. Analysis of coding sequences in this way gives rise to large peaks at frequency 0.33, corresponding to the three-base-pair periodicity of triplet codons found in such sequences. Application of this method to the DNA sequences generated by the simulation would give rise to peaks corresponding to the repeat length of the blocks. The earlier simulations which appeared 'blocky' would be expected to show strong peaks at well defined frequencies, while the later, more realistic simulations would give smaller peaks over a wider range of frequencies. The computer-generated sequences could be compared to real non-coding sequences in a less subjective manner than simple visual inspection.

Given that the current model is a realistic simulation of the natural system, the opportunity presents itself to look into the linguistic structure that underlies the sequences produced. The question has been put forward, "Does the linguistic

structure seen in non-coding DNA sequences have any biological significance?" It seems conceptually possible that the linguistic features of non-coding DNA, while not actually denoting a language, could be used as a control system of some kind by biological systems. Such a control mechanism might be used to regulate gene expression or the three-dimensional structure of DNA, for example.

Our model shows that the structure is formed by repetitive sequences in the genome, so any control structure should be based around these blocks. A control system may be built up by considering such details as the length of repetitive sequences, the period of the repeats, how close to one another the repetitive sequences are, how 'perfect' the repeats are. One could then consider setting up various rules governing the effects of these sequences on, for example, gene expression. E.g. If a repetitive sequence is within 1,000bp of the promoter for a given gene then expression at that gene is increased. The various repetitive sequences might interact with one another, possibly enhancing each others effects, or destroying them.

The investigation into the possible existence and workings of such a control structure is a logical progression of the work presented here. This is a challenging problem. The first stage would be to try and show how a control structure might function based on repetitive sequences, without trying to prove that this control structure actually exists in nature. This could be achieved by setting up a proposed system and seeing how it evolved when the simulation was used to modify the DNA. A viable system would continue to function while the mutations were applied, but should, at some point, undergo a sudden change, signifying a large-scale mutation. Depending on the outcome of this work, it may then be possible to look into a more realistic approach to the problem.

Appendix: Program listings

The simulation consists of two main files, *dnasim2.c* and *dnasim2cfg.h*. The parameters affecting the simulation from run to run are contained in *dnasim2cfg.h* (along with parameters affecting the running of data analysis programs - listings of which are not included) while *dnasim2.c* is the main source file. To run the simulation, the required parameters are adjusted in *dnasim2cfg.h* and *dnasim2.c* is then compiled along with a source code file, *ran2.o* containing the implementation of the ran2() random number generator. The resulting executable is then run. The file *ran2.h*, referred to in the code for *dnasim2.c* contains the header information for the ran2() random number generator. The code in *ran2.h* and *ran2.o* can be found in reference 68. The listing presented here reflects the final version of the program, capable of all the mutational processes outlined in the text.

5.1 DNASIM2.C

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "ran2.h"
#include "dnasim2cfg.h"

struct Base {
  Base *nextNode;
  Base *prevNode;
  char base;
};

// Structure to hold target site data
struct siteinfo {
  char site[ MAX_SITE_LENGTH ];
  unsigned long numbersites;
  int copy_length_upper;
  int copy_length_lower;
  int ratio;
  int perf_ratio;
  int im_ins_ratio; // Parameters for imperfect excision events from this point
  int im_del_ratio;
```

```
    int ins_upper;
    int ins_lower;
    int del_upper;
    int del_lower;
    char teleft[ INS_MAXRANGE + 1 ];
    char teright[ INS_MAXRANGE + 1 ];
};

const int NS = 0; // Initializer for NS in sitedetails data structure

void pointMutation( Base *, Base *, siteinfo [], Base *[][ NO_OF_SITES ],
  unsigned long, long * );
unsigned long insert( Base *, Base *, siteinfo [], Base *[][ NO_OF_SITES ],
  int, unsigned long, long * );
void deleteBases( Base *, int, Base **, Base **, unsigned long & );
Base *getInsertSequence( siteinfo, Base **, int, long * );
void appendBase( char, Base **, unsigned long & );
void appendFirstBase( char, Base **, Base **, unsigned long & );
Base *moveForward( Base *, unsigned long );
Base *moveBackwards( Base *, unsigned long );
unsigned long tsitesNotPresent( Base *[], Base *[], Base *[] );
int checkTs( Base *, char * );
int insertSubSeq( Base *, Base *, Base *, unsigned long, unsigned long & );
Base *copySubSeq( Base *, Base **, unsigned long );
Base *findBase( char, Base * );
Base *findBase( char, Base *, unsigned long &);
Base *findString( char *, Base * );
Base *findString( char *, Base *, unsigned long &);
void recordSites( Base *, siteinfo [], Base *[][ NO_OF_SITES ] );
unsigned long targetSiteSearch( char *, Base *, Base *[] );
void targetSiteSearchRegion( char *, Base *, Base *[], unsigned long );
int tsArrayMod( Base *[], Base *[], Base *[], int, int, unsigned long );
void tsArrayInit( Base *[], unsigned long );
int loadSequence( Base **, Base **, unsigned long &, char *);
int outSequence( Base *, char * );
void newSeq( Base **, Base **, unsigned long & );
void printSequence( Base * );
void deleteSequence( Base **, Base **, unsigned long & );
void deleteSequence( Base ** );
unsigned long getCycles( unsigned long [] );
int totalOfRatios( siteinfo[] );
int outputInfo( siteinfo st_dat[], char input_file[], unsigned long cycles,
  unsigned long out_array[]);
int checkSitesPresent( siteinfo [] );

main()
{
  char outFName[ MAX_PATH_LENGTH ];
  char inFName[ MAX_FILENAME_LENGTH ];
  Base *startPtr, *endPtr;
  Base *sitePositions[ NO_OF_SITE_TYPES ][ NO_OF_SITES ];
  Base *toRemoveArray[ NO_OF_SITE_TYPES ][ NO_OF_TMP_SITES ];
  Base *blankArray[ 1 ];
  Base *toRemoveEndCheck;
  unsigned long cyclesDone = 0, cyclesRequired;
  unsigned long length;
  unsigned long testLength;
  Base *testStartPtr, *testEndPtr;
  int ratioSum;
  int writeAtIndex = 0;
  int mutType;
  int mutTypRatioTotal;
  long *initial, seed;
  int sitesPresent;
  int range;
  int k, numToRemove;
```

```
/* The following macro defines sitedetails, and is defined in the seperate
     header file "dna_cfg.h" */

SITEDETAILS_DECLARATION;

/* The following macro defines the number of cycles at which data should be
 output and is defined in the seperate header file "dna_cfg.h" */

WRITE_AT_DECLARATION;

// Attempt to load input file
if ( !loadSequence( &startPtr, &endPtr, length, inFName ) ) {
  cout << "Input DNA file could not be opened, or file contains no DNA
    data\n";
  exit( 1 );
}

// Make a new sequence to use to test whether output files can be correctly
// opened
newSeq( &testStartPtr, &testEndPtr, testLength );
appendFirstBase( 'g', &testStartPtr, &testEndPtr, testLength );

// Check the output files can be opened (write testSeq to each) - end program
// if cannot
for ( int ckOutIndex = 0; ckOutIndex <= NO_OUTPUT_FILES - 1; ckOutIndex++ ) {
  sprintf( outFName, "%s%lu", RES_DIR, write_at[ ckOutIndex ] );
  outSequence( testStartPtr, outFName );
}

// Delete the sequence used to test the output files
deleteSequence( &testStartPtr, &testEndPtr, testLength );

// Initializations
cyclesRequired = getCycles( write_at );
ratioSum = totalOfRatios( sitedetails );

// Initialize the sitePositions array
for ( int i = 0; i <= NO_OF_SITE_TYPES - 1; i++ )

  for ( unsigned long j = 0; j <= NO_OF_SITES - 1; j++ )
    sitePositions[ i ][ j ] = 0;

mutTypRatioTotal = RATIO_PT_MT + RATIO_INSERT;

// Output the details of the simulation to a details file
if ( !outputInfo( sitedetails, inFName, cyclesRequired, write_at ) ) {
  cout << "Can't open details file\n\n";
  exit( 1 );
}

// Initialization of ran2.  Must be seeded with a -ve integer.
// The same seed is used in all calls to ran2 throughout the program.
initial = &seed;
seed = -time( NULL );

// Do initial search for target sites
recordSites( startPtr, sitedetails, sitePositions );

do
{
  // Decide whether to do an insertion or a point mutation

  mutType = static_cast< int >( floor(ran2( initial ) * mutTypRatioTotal ) +
    1 );

  /* Check to see that the sequence contains sites.  Force a
    point mutation if it doesn't.  If no mutations are enabled
```

```
      then quit the program. */
    sitesPresent = checkSitesPresent( sitedetails );

    // Record any target sites that are too close to the beginning of the
      sequence
    for ( int j = 0; j <= NO_OF_SITE_TYPES - 1; j++ ) {
      tsArrayInit( toRemoveArray[ j ], NO_OF_TMP_SITES );

      if ( ( range = sitedetails[ j ].copy_length_upper - strlen(
        sitedetails[ j ].site ) ) > 0 ) {

        targetSiteSearchRegion( sitedetails[ j ].site, startPtr,
                                toRemoveArray[ j ], range );

        // Count the number of this site to be removed
        for ( k = 0; ( toRemoveEndCheck = toRemoveArray[ j ][ k ] ) != 0; k++ )
          ;

        numToRemove = k;

        if ( numToRemove > 0 )
          sitedetails[ j ].numbersites += tsArrayMod( blankArray,
            toRemoveArray[ j ], sitePositions[ j ], 0, numToRemove,
            sitedetails[ j ].numbersites );
      }
    }

    /* If the sequence doesn't contain sites and there are to be no point
      mutations then end the program */
    if ( RATIO_PT_MT == 0 && sitesPresent == 0 ) {
        cerr << "No target sites in this sequence, and no point mutations\n\n";
        exit( 1 );
    }

    /*  Force point mutation if no insertion sites */
    if ( sitesPresent == 0 )
      mutType = RATIO_PT_MT;

    if ( mutType <= RATIO_PT_MT )
      pointMutation( startPtr, endPtr, sitedetails, sitePositions, length,
        initial );
    else
      // Insert returns the new length for use by insert in the next cycle
      length = insert( startPtr, endPtr, sitedetails, sitePositions,
                       ratioSum, length, initial );

    cyclesDone++;
    cout << "Cycles done: " << cyclesDone << "\r";

    if ( cyclesDone == write_at[ writeAtIndex ] ) {
      sprintf( outFName, "%s%lu", RES_DIR, cyclesDone );
      outSequence( startPtr, outFName );
      writeAtIndex++;
    }

    // Make sure endPtr points to the last base in the sequence
    while ( endPtr->nextNode != 0 )
      endPtr = endPtr->nextNode;

  } while ( cyclesDone <= cyclesRequired - 1 );

  deleteSequence( &startPtr, &endPtr, length );
  return 0;
}

////////////////////////////////////////////////////////////////////////////
// Function to carry out point mutations on sequence
```

```
void pointMutation( Base *startPtr, Base *endPtr, siteinfo tsinfo[], Base
                    *tsPositions[][ NO_OF_SITES ], unsigned long seqLength,
                    long *ranInit )
{
  unsigned long pmutPos;
  char baseToMut, newBase;
  int transRatioTotal, randomBase, transOrTranv;
  Base *baseForMut;
  Base *regionStartPtr;
  Base *tmpOrigTs[ NO_OF_SITE_TYPES ][ NO_OF_TMP_SITES ];
  Base *tmpNewTs[ NO_OF_SITE_TYPES ][ NO_OF_TMP_SITES ];
  Base *tmpToAddTs[ NO_OF_SITE_TYPES ][ NO_OF_TMP_SITES ];
  Base *tmpToRemoveTs[ NO_OF_SITE_TYPES ][ NO_OF_TMP_SITES ];
  int numToAdd, numToRemove;
  int numToMoveBack;

  // Sum the ratios for transition (purine -> purine) and transversion (purine
    -> pyrimidine)
  transRatioTotal = RATIO_TRANSIT + RATIO_TRANSVER;

  // Initialise the arrays used to check target sites
  for ( int l = 0; l <= NO_OF_SITE_TYPES - 1; l++ ) {
    tsArrayInit( tmpOrigTs[ l ], NO_OF_TMP_SITES );
    tsArrayInit( tmpNewTs[ l ], NO_OF_TMP_SITES );
    tsArrayInit( tmpToAddTs[ l ], NO_OF_TMP_SITES );
    tsArrayInit( tmpToRemoveTs[ l ], NO_OF_TMP_SITES );
  }

  /* Randomly select:
        a-the exact base for point mutation
        b-the new random base (eg. if a 't' (pyrimidine) goes to
          either 'a' or 'g'(purine)(a transversion), randomBase decides which)
        c-whether the modification is transition or transversion */
  pmutPos = static_cast< unsigned long >( floor( ran2( ranInit ) *
                                          seqLength ) );

  // Make baseForMut point to the base that is to be mutated
  // Start from the end of the sequence which is nearest to the base to be
    mutated
  if ( pmutPos > seqLength / 2 ) { // work from end of sequence

       baseForMut = endPtr;

       for ( unsigned long i = seqLength - 1; i >= pmutPos + 1; i-- )
             baseForMut = baseForMut->prevNode;
  }
  else { // work from start of sequence

       baseForMut = startPtr;

       for ( unsigned long j = 0; j < pmutPos; j++ )
             baseForMut = baseForMut->nextNode;
  }

  baseToMut = baseForMut->base;

  // randomBase takes either 0 or 1, with equal probability
  randomBase = static_cast< int >( floor( ran2( ranInit ) * 2 ) );

  // transOrTranv = 1 indicates a transition, 0 a transversion
  if ( static_cast< int >( floor( ran2( ranInit ) * transRatioTotal ) ) + 1 <=
       RATIO_TRANSIT ) {
    transOrTranv = 1;
  }
  else {
    transOrTranv = 0;
```

```
  }

  // Switch structure to determine the new base
  switch ( baseToMut ) {

  case 't':
    if ( transOrTranv == 1 ) {
      newBase = 'c';
    }
    else {
      if ( randomBase == 1 ) {
        newBase = 'a';
      }
      else {
        newBase = 'g';
      }
    }

    break;

 case 'c':
    if ( transOrTranv == 1 ) {
      newBase = 't';
    }
    else {
      if ( randomBase == 1 ) {
        newBase = 'a';
      }
      else {
        newBase = 'g';
      }
    }

    break;

 case 'a':
    if ( transOrTranv == 1 ) {
      newBase = 'g';
    }
    else {
      if ( randomBase == 1 ) {
        newBase = 't';
      }
      else {
        newBase = 'c';
      }
    }

    break;

 case 'g':
    if ( transOrTranv == 1 ) {
      newBase = 'a';
    }
    else {
      if ( randomBase == 1 ) {
        newBase = 't';
      }
      else {
        newBase = 'c';
      }
    }
  }

  // Record the target sites in the area where the mutation is to occur
  for ( int k = 0; k <= NO_OF_SITE_TYPES - 1; k++ ) {
    numToMoveBack = strlen( tsinfo[ k ].site ) - 1;
```

```
      regionStartPtr = baseForMut;

      // Set regionStartPtr to the first base to check
      for ( ; numToMoveBack != 0 && regionStartPtr != 0; numToMoveBack-- )
        regionStartPtr = moveBackwards( regionStartPtr, 1 );


      // Perform the check
      if ( regionStartPtr != 0 ) // If didn't find the beginning of the
                                 sequence when moving back
        targetSiteSearchRegion( tsinfo[ k ].site, regionStartPtr,
                                tmpOrigTs[ k ], strlen( tsinfo[ k ].site ) );
      else
        targetSiteSearchRegion( tsinfo[ k ].site, startPtr, tmpOrigTs[ k ],
                                strlen( tsinfo[ k ].site ) -
                                ( numToMoveBack + 1 ) );
    }

    // Perform the mutation
    baseForMut->base = newBase;

    // Re-check the area where the mutation occurred for new target sites
    for ( int m = 0; m <= NO_OF_SITE_TYPES - 1; m++ ) {
      numToMoveBack = strlen( tsinfo[ m ].site ) - 1;
      regionStartPtr = baseForMut;

      // Set regionStartPtr to the first base to check
      for ( ; numToMoveBack != 0 && regionStartPtr != 0; numToMoveBack-- )
        regionStartPtr = moveBackwards( regionStartPtr, 1 );

        // Perform the check
        if ( regionStartPtr != 0 ) // If didn't find the beginning of the
                                   sequence when moving back
          targetSiteSearchRegion( tsinfo[ m ].site, regionStartPtr,
                                  tmpNewTs[ m ], strlen( tsinfo[ m ].site ) );
        else
          targetSiteSearchRegion( tsinfo[ m ].site, startPtr, tmpNewTs[ m ],
                                  strlen( tsinfo[ m ].site ) -
                                  ( numToMoveBack + 1 ) );
    }

    // For each target site list perform any required modifications
    for ( int n = 0; n <= NO_OF_SITE_TYPES - 1; n++ ) {
      numToRemove = tsitesNotPresent( tmpOrigTs[ n ], tmpNewTs[ n ],
                                      tmpToRemoveTs[ n ] );
      numToAdd = tsitesNotPresent( tmpNewTs[ n ], tmpOrigTs[ n ],
                                   tmpToAddTs[ n ] );
      tsArrayMod( tmpToAddTs[ n ], tmpToRemoveTs[ n ], tsPositions[ n ],
                  numToAdd, numToRemove, tsinfo[ n ].numbersites );
      tsinfo[ n ].numbersites += numToAdd - numToRemove;
    }
}

////////////////////////////////////////////////////////////////////////////
// Function that carries out insertion process
// Takes pointer to start of sequence, end of sequence, array of sitedata, list
// of target sites, sum of TS ratios, the length of the host sequence and the
// ran2 initialiser.
// Returns the new length of the host sequence after the insertion

unsigned long insert( Base *start, Base *end, siteinfo siteparam[], Base
                      *sitePos[][ NO_OF_SITES ], int ratioSum,
                      unsigned long lgth, long *ran2init )
{
  Base *tmpOrigTs[ NO_OF_SITE_TYPES ][ NO_OF_TMP_SITES ];
  Base *tmpNewTs[ NO_OF_SITE_TYPES ][ NO_OF_TMP_SITES ];
  Base *tmpToAddTs[ NO_OF_SITE_TYPES ][ NO_OF_TMP_SITES ];
```

```
 Base *tmpToRemoveTs[ NO_OF_SITE_TYPES ][ NO_OF_TMP_SITES ];
 Base *regionStartPtr;
 Base *insertTSPtr;
 Base *insertPtr;
 Base *deletePtr;
 Base *copyStartPtr;
 Base *subPtr, *subEndPtr;
 Base *extraBasesPtr, *extraBasesEndPtr;
 unsigned long substringLength;
 int numToRemove, numToAdd;
 int regionStartOffset;
 int typeForInsert, sitetypeNum, runTotal;
 int  i_or_d_ratio_sum, i_or_d_rand_num;
 int num_bases_ins, num_bases_del, copyLength;
 unsigned long sitenumForInsert;

 // Initialize the temp TS arrays
 for ( int k = 0; k <= NO_OF_SITE_TYPES - 1; k++ ) {
   tsArrayInit( tmpOrigTs[ k ], NO_OF_TMP_SITES );
   tsArrayInit( tmpNewTs[ k ], NO_OF_TMP_SITES );
   tsArrayInit( tmpToAddTs[ k ], NO_OF_TMP_SITES );
   tsArrayInit( tmpToRemoveTs[ k ], NO_OF_TMP_SITES );
 }

 /* Randomly select the site sequence to insert at, continuing while
    the site type selected isn't present.

    First select a random number, in range 1 to ratio_total
    (total of all ratios used to bias site insertion).  Then taking the
    sites in turn, run past each one using site_index, adding its
    ratio, until the running total exceeds the random number, at which
    point the loop (increments site_index and then) exits.  The
    sitetype to insert at is the sitetype denoted by the site_index - 1
    when the loop exited. */

 do {
   runTotal = 0;

   sitetypeNum = static_cast< int >( floor( ran2( ran2init ) * ratioSum )
                                     + 1 );

   for ( int i = 0; i <= NO_OF_SITE_TYPES - 1 && runTotal <= sitetypeNum - 1;
         i++ )
     runTotal += siteparam[ i ].ratio;

     typeForInsert = i - 1;
 } while ( siteparam[ typeForInsert ].numbersites == 0 );


 // Randomly select one of the sites from the site sequence which was chosen
 // above
 sitenumForInsert = static_cast< unsigned long >( ( floor( ran2( ran2init ) *
                    siteparam[ typeForInsert ].numbersites ) ) );
 insertTSPtr = sitePos[ typeForInsert ][ sitenumForInsert ];

 // Once the site has been chosen, decide what kind of insert/excision to do -
 // insert extra bases, delete extra bases, or a perfect copy
 i_or_d_ratio_sum = siteparam[ typeForInsert ].im_ins_ratio +
                    siteparam[ typeForInsert ].im_del_ratio +
                    siteparam[ typeForInsert ].perf_ratio;
 i_or_d_rand_num = static_cast< int >( floor( ran2( ran2init ) *
                   i_or_d_ratio_sum ) ) + 1;

 if ( i_or_d_rand_num <= siteparam[ typeForInsert ].im_ins_ratio ) {
   // Do an insertion of extra bases
   // First determine the number of bases to insert
   num_bases_ins = static_cast< int >( floor( ran2( ran2init ) *
```

```
                ( siteparam[ typeForInsert ].ins_upper -
                siteparam[ typeForInsert ].ins_lower + 1 ) ) +
                siteparam[ typeForInsert ].ins_lower );

  // Call getInsertSequence to generate sequence to insert
  extraBasesPtr = getInsertSequence( siteparam[ typeForInsert ],
                &extraBasesEndPtr, num_bases_ins, ran2init );

  // Choose the copy length to be used
  copyLength = static_cast< int >( ( floor( ran2( ran2init ) *
              ( siteparam[ typeForInsert ].copy_length_upper -
              siteparam[ typeForInsert ].copy_length_lower + 1 ) ) ) +
              siteparam[ typeForInsert ].copy_length_lower );

  // Record the target sites in the region where the insertion is to occur
  for ( int j = 0; j <= NO_OF_SITE_TYPES - 1; j++ ) {

    // Set regionStartPtr to the first base to check
    regionStartOffset = strlen( siteparam[ typeForInsert ].site ) -
      strlen( siteparam[ j ].site ) + 1;

    if ( regionStartOffset == 0 )
      regionStartPtr = insertTSPtr;
    else
      if ( regionStartOffset < 0 )
        regionStartPtr = moveBackwards( insertTSPtr, -regionStartOffset );
      else // Must be +ve
        regionStartPtr = moveForward( insertTSPtr, regionStartOffset );

    // Perform the check
    targetSiteSearchRegion( siteparam[ j ].site, regionStartPtr,
                            tmpOrigTs[ j ], strlen( siteparam[ j ].site ) -
                            1 );
  }

  // Move the copyStartPtr to the start of the region to be copied
  // 1) Move to the end of the target site then
  // 2) Move back by the copy length
  copyStartPtr = moveForward( insertTSPtr,
                 strlen( siteparam[ typeForInsert ].site ) - 1 );
  copyStartPtr = moveBackwards( copyStartPtr, copyLength - 1 );

  // Copy the sub-sequence
  if ( !( subPtr = copySubSeq( copyStartPtr, &subEndPtr, copyLength ) ) ) {
    cout << "Copy failed\n";
    exit( 1 );
  }

  // Move the insertion pointer to the correct insertion point
  if ( ( insertPtr = moveForward( insertTSPtr,
                     strlen( siteparam[ typeForInsert ].site ) - 1 ) ) ==
                     0 ) {
    cout << "Move forward failed\n";
    deleteSequence( &extraBasesPtr, &extraBasesEndPtr, substringLength );
    deleteSequence( &subPtr, &subEndPtr, substringLength );
    exit( 1 );
  }

  // Insert the extra bases that are to be inserted (TE remains)
  if ( !insertSubSeq( insertPtr, extraBasesPtr, extraBasesEndPtr,
                      num_bases_ins, lgth ) ) {
    cout << "Insertion failed\n";
    deleteSequence( &extraBasesPtr, &extraBasesEndPtr, substringLength );
    deleteSequence( &subPtr, &subEndPtr, substringLength );
    exit( 1 );
  }
  // Move the insertion pointer to the new insertion point (after the TE
```

```
    // remains)
    if ( ( insertPtr = moveForward( insertPtr, num_bases_ins ) ) == 0 ) {
      cout << "Move forward failed\n";
      deleteSequence( &subPtr, &subEndPtr, substringLength );
      exit( 1 );
    }

    // Insert the sub-sequence
    if ( !insertSubSeq( insertPtr, subPtr, subEndPtr, copyLength, lgth ) ) {
      cout << "Insertion failed\n";
      deleteSequence( &subPtr, &subEndPtr, substringLength );
      exit( 1 );
    }

    // Recheck the target sites in the region of the insertion to see if they
    // are still present
    for ( int l = 0; l <= NO_OF_SITE_TYPES - 1; l++ ) {
      // Set regionStartPtr to the first base to check
      regionStartOffset = strlen( siteparam[ typeForInsert ].site ) -
                          strlen( siteparam[ l ].site ) + 1;

      if ( regionStartOffset == 0 )
        regionStartPtr = insertTSPtr;
      else
        if ( regionStartOffset < 0 )
          regionStartPtr = moveBackwards( insertTSPtr, -regionStartOffset );
        else // Must be +ve
          regionStartPtr = moveForward( insertTSPtr, regionStartOffset );

      // Perform the check
      targetSiteSearchRegion( siteparam[ l ].site, regionStartPtr,
                              tmpNewTs[ l ], strlen( siteparam[ l ].site ) +
                              copyLength + num_bases_ins - 1 );
    }

    // For each target site list perform any required modifications
    for ( int m = 0; m <= NO_OF_SITE_TYPES - 1; m++ ) {
      numToRemove = tsitesNotPresent( tmpOrigTs[ m ], tmpNewTs[ m ],
                                      tmpToRemoveTs[ m ] );
      numToAdd = tsitesNotPresent( tmpNewTs[ m ], tmpOrigTs[ m ],
                                   tmpToAddTs[ m ] );
      tsArrayMod( tmpToAddTs[ m ], tmpToRemoveTs[ m ], sitePos[ m ], numToAdd,
                  numToRemove, siteparam[ m ].numbersites );
      siteparam[ m ].numbersites += numToAdd - numToRemove;
    }

    return lgth;
  }
  else {

    if ( i_or_d_rand_num <= siteparam[ typeForInsert ].im_ins_ratio +
                            siteparam[ typeForInsert ].im_del_ratio ) {
    // Do a deletion of extra bases

      Base *startSearch[ NO_OF_SITE_TYPES ]; // Array to hold searchregion
                                             // start pointers
      int lhs, rhs; // Bases to be deleted from left of the centre point, and
                    // right of the centre point
      int startRangeMod; // Amount to extend start search range
      int endRangeMod; // Amount to extend end search range

      num_bases_del = static_cast< int >( floor( ran2( ran2init ) *
                      ( siteparam[ typeForInsert ].del_upper -
                      siteparam[ typeForInsert ].del_lower + 1 ) ) +
                      siteparam[ typeForInsert ].del_lower );

      // If num_bases_del is odd, take additional base from rhs or lhs with
```

```
// equal probability
if ( num_bases_del % 2 == 0 )
  lhs = rhs = num_bases_del / 2;
else {
  if ( static_cast< int >( floor( ran2( ran2init ) * 2 ) ) == 0 ) {
    lhs = num_bases_del / 2;
    rhs = lhs + 1; // Take extra base from rhs
  }
  else {
    lhs = ( num_bases_del / 2 ) + 1; // Take extra base from lhs
    rhs = lhs - 1;
  }
}

// First do a normal perfect copy

// Once the site has been chosen, determine the copy length to use
copyLength = static_cast< int >( ( floor( ran2( ran2init ) *
             ( siteparam[ typeForInsert ].copy_length_upper -
             siteparam[ typeForInsert ].copy_length_lower + 1 ) ) ) +
             siteparam[ typeForInsert ].copy_length_lower );

// If the number of bases to be deleted would remove all the bases in the
// sequence then quit the program
if ( lgth + copyLength <= static_cast< unsigned long >
                          ( num_bases_del ) ) {
  cout << "\nImprecise excision has deleted all bases in the sequence\n";
  exit( 1 );
}

// Record the target sites in the region where the insertion is to occur
for ( int j = 0; j <= NO_OF_SITE_TYPES - 1; j++ ) {

  // Set regionStartPtr to the first base to check
  regionStartOffset = strlen( siteparam[ typeForInsert ].site ) -
                      strlen( siteparam[ j ].site ) - lhs + 1;

  if ( regionStartOffset == 0 )
    regionStartPtr = insertTSPtr;
  else
    if ( regionStartOffset < 0 )
      regionStartPtr = moveBackwards( insertTSPtr, -regionStartOffset );
    else // Must be +ve
      regionStartPtr = moveForward( insertTSPtr, regionStartOffset );

  startSearch[ j ] = regionStartPtr; // Record the starting point for the
                                     // search for use in search after
                                     // the insertion+deletion events

  if ( rhs - copyLength > 0 )
    startRangeMod = rhs - copyLength;
  else
    startRangeMod = 0;

  // Perform the check
  targetSiteSearchRegion( siteparam[ j ].site, regionStartPtr,
                          tmpOrigTs[ j ], strlen( siteparam[ j ].site ) +
                          lhs + startRangeMod - 1 );
}

// Move the copyStartPtr to the start of the region to be copied
// 1) Move to the end of the target site then
// 2) Move back by the copy length
copyStartPtr = moveForward( insertTSPtr,
                            strlen( siteparam[ typeForInsert ].site ) -
                            1 );
copyStartPtr = moveBackwards( copyStartPtr, copyLength - 1 );
```

```
// Copy the sub-sequence
if ( !( subPtr = copySubSeq( copyStartPtr, &subEndPtr, copyLength ) ) ) {
  cout << "Copy failed\n";
  exit( 1 );
}

// Move the insertion pointer to the correct insertion point
if ( ( insertPtr = moveForward( insertTSPtr,
                   strlen( siteparam[ typeForInsert ].site ) - 1 ) ) ==
                   0 ) {
  cout << "Move forward failed\n";
  deleteSequence( &subPtr, &subEndPtr, substringLength );
  exit( 1 );
}

// Insert the sub-sequence
if ( !insertSubSeq( insertPtr, subPtr, subEndPtr, copyLength, lgth ) ) {
  cout << "Insertion failed\n";
  deleteSequence( &subPtr, &subEndPtr, substringLength );
  exit( 1 );
}

// Now delete the bases
if ( ( deletePtr = moveForward( insertTSPtr,
                   strlen( siteparam[ typeForInsert ].site ) - 1 ) ) ==
                   0 ) {
  cout << "Move forward failed for deletePtr\n";
  exit( 1 );
}

if ( lhs != 0 ) {

  if ( ( deletePtr = moveBackwards( deletePtr, lhs - 1 ) ) == 0 ) {
    cout << "Move backwards failed for deletePtr\n";
    exit( 1 );
  }
}
else {

  if ( ( deletePtr = moveForward( deletePtr, 1 ) ) == 0 ) {
    cout << "Move forwards (when lhs == 0) failed for deletePtr\n";
    exit( 1 );
  }
}

deleteBases( deletePtr, num_bases_del, &start, &end, lgth );

// Recheck the target sites in the region of the insertion to see if they
// are still present
for ( int l = 0; l <= NO_OF_SITE_TYPES - 1; l++ ) {
  // Set regionStartPtr to the first base to check - uses the same start
  // point as in the 'before' check
  regionStartPtr = startSearch[ l ];

  // endRangeMod takes the smaller of rhs or copyLength
  if ( copyLength < rhs )
    endRangeMod = copyLength;
  else
    endRangeMod = rhs;

  // Perform the check
  targetSiteSearchRegion( siteparam[ l ].site, regionStartPtr,
                          tmpNewTs[ l ], strlen( siteparam[ l ].site ) +
                          copyLength - endRangeMod - 1 );
}
```

```
    // For each target site list perform any required modifications
    for ( int m = 0; m <= NO_OF_SITE_TYPES - 1; m++ ) {
      numToRemove = tsitesNotPresent( tmpOrigTs[ m ], tmpNewTs[ m ],
                    tmpToRemoveTs[ m ] );
      numToAdd = tsitesNotPresent( tmpNewTs[ m ], tmpOrigTs[ m ],
                 tmpToAddTs[ m ] );
      tsArrayMod( tmpToAddTs[ m ], tmpToRemoveTs[ m ], sitePos[ m ],
                  numToAdd, numToRemove, siteparam[ m ].numbersites );
      siteparam[ m ].numbersites += numToAdd - numToRemove;
    }

    return lgth;
  }
  else {
    // Do a perfect copy
    // Once the site has been chosen, determine the copy length to use
    copyLength = static_cast< int >( ( floor( ran2( ran2init ) *
                 ( siteparam[ typeForInsert ].copy_length_upper -
                 siteparam[ typeForInsert ].copy_length_lower + 1 ) ) ) +
                 siteparam[ typeForInsert ].copy_length_lower );

    // Record the target sites in the region where the insertion is to occur
    for ( int j = 0; j <= NO_OF_SITE_TYPES - 1; j++ ) {
       // Set regionStartPtr to the first base to check
       regionStartOffset = strlen( siteparam[ typeForInsert ].site ) -
                           strlen( siteparam[ j ].site ) + 1;

       if ( regionStartOffset == 0 )
         regionStartPtr = insertTSPtr;
       else
         if ( regionStartOffset < 0 )
           regionStartPtr = moveBackwards( insertTSPtr, -regionStartOffset );
         else // Must be +ve
           regionStartPtr = moveForward( insertTSPtr, regionStartOffset );

       // Perform the check
       targetSiteSearchRegion( siteparam[ j ].site, regionStartPtr,
                               tmpOrigTs[ j ],
                               strlen( siteparam[ j ].site ) - 1 );
    }

    // Move the copyStartPtr to the start of the region to be copied
    // 1) Move to the end of the target site then
    // 2) Move back by the copy length
    copyStartPtr = moveForward( insertTSPtr,
                                strlen( siteparam[ typeForInsert ].site ) -
                                1 );
    copyStartPtr = moveBackwards( copyStartPtr, copyLength - 1 );

    // Copy the sub-sequence
    if ( !( subPtr = copySubSeq( copyStartPtr, &subEndPtr, copyLength ) ) ) {
      cout << "Copy failed\n";
      exit( 1 );
    }

    // Move the insertion pointer to the correct insertion point
    if ( ( insertPtr = moveForward( insertTSPtr,
                       strlen( siteparam[ typeForInsert ].site ) - 1 ) ) ==
                       0 ) {
      cout << "Move forward failed\n";
      deleteSequence( &subPtr, &subEndPtr, substringLength );
      exit( 1 );
    }

    // Insert the sub-sequence
    if ( !insertSubSeq( insertPtr, subPtr, subEndPtr, copyLength, lgth ) ) {
      cout << "Insertion failed\n";
```

```
        deleteSequence( &subPtr, &subEndPtr, substringLength );
        exit( 1 );
      }

      // Recheck the target sites in the region of the insertion to see if they
      // are still present
      for ( int l = 0; l <= NO_OF_SITE_TYPES - 1; l++ ) {
        // Set regionStartPtr to the first base to check
        regionStartOffset = strlen( siteparam[ typeForInsert ].site ) -
                            strlen( siteparam[ l ].site ) + 1;

        if ( regionStartOffset == 0 )
          regionStartPtr = insertTSPtr;
        else
          if ( regionStartOffset < 0 )
            regionStartPtr = moveBackwards( insertTSPtr, -regionStartOffset );
          else // Must be +ve
            regionStartPtr = moveForward( insertTSPtr, regionStartOffset );

        // Perform the check
        targetSiteSearchRegion( siteparam[ l ].site, regionStartPtr,
                                tmpNewTs[ l ], strlen( siteparam[ l ].site ) +
                                copyLength - 1 );
      }

      // For each target site list perform any required modifications
      for ( int m = 0; m <= NO_OF_SITE_TYPES - 1; m++ ) {
        numToRemove = tsitesNotPresent( tmpOrigTs[ m ], tmpNewTs[ m ],
                                        tmpToRemoveTs[ m ] );
        numToAdd = tsitesNotPresent( tmpNewTs[ m ], tmpOrigTs[ m ],
                                     tmpToAddTs[ m ] );
        tsArrayMod( tmpToAddTs[ m ], tmpToRemoveTs[ m ], sitePos[ m ],
                    numToAdd, numToRemove, siteparam[ m ].numbersites );
        siteparam[ m ].numbersites += numToAdd - numToRemove;
      }

    return lgth;
    }
  }
}


////////////////////////////////////////////////////////////////////////////
// Function to delete bases from the middle of a sequence for
// insertion/deletion events
// Takes pointer to start of sequence to be deleted, and number of bases to be
// deleted (including first pointed to) and the length of the full sequence.
// The length is modified to the new length after deletions.

void deleteBases( Base *delPtr, int numBasesDel, Base **seqStartPtr,
                  Base **seqEndPtr, unsigned long &seqLength )
{
  Base *lEndPtr, *rEndPtr, *tmpPtr;

  if ( delPtr->prevNode != 0 ) // If there are bases to the left of those to be
                               // deleted
    lEndPtr = delPtr->prevNode;
  else
    lEndPtr = 0;

  if ( ( tmpPtr = moveForward( delPtr, numBasesDel - 1 ) ) == 0 ) {
    cout << "Move Forward failed in deleteBases\n";
    exit( 1 );
  }

  if ( tmpPtr->nextNode != 0 ) // If there are bases to the right of those to
                               // be deleted
```

```
    rEndPtr = tmpPtr->nextNode;
  else
    rEndPtr = 0;

  if ( rEndPtr == 0 && lEndPtr == 0 ) {
    cout << "Error in deleteBases - entire sequence to be deleted\n";
    exit( 1 );
  }
  else

    if ( lEndPtr == 0 ) { // The bases to be deleted are exactly at the
                          // beginning of the sequence
      *seqStartPtr = rEndPtr;
      rEndPtr->prevNode->nextNode = 0;
      rEndPtr->prevNode = 0;
      deleteSequence( &tmpPtr );
    }
    else

      if ( rEndPtr == 0 ) { // The bases to be deleted are exactly at the end
                            // of the sequence
        *seqEndPtr = lEndPtr;
        lEndPtr->nextNode = 0;
        delPtr->prevNode = 0;
        deleteSequence( &tmpPtr );
      }
      else { // For a 'normal' case where there are bases on both sides of
             // those to be deleted
        rEndPtr->prevNode->nextNode = 0;
        rEndPtr->prevNode = lEndPtr;
        lEndPtr->nextNode->prevNode = 0;
        lEndPtr->nextNode = rEndPtr;
        deleteSequence( &tmpPtr );
      }

  seqLength -= numBasesDel;
}

////////////////////////////////////////////////////////////////////////////
// Function to generate the additional bases to be inserted in imprecise
// insertion/excsion.
// Takes the sitedetails structure for the site concerned, the number of
// additional bases to be inserted, and the ran2() initializer
// Returns a pointer to a sequence of bases (linked list) to insert

Base *getInsertSequence( siteinfo sitedta, Base **insertSeqEndPtr,
                         int numberBases, long *rndInit )
{
  int l_or_r, r_te_length;
  Base *insertSeqStartPtr;
  unsigned long insertSeqLgth;
  char randomBase;
  int randomBaseNum;

  // If random bases are to be used then generate sequence
  if ( INSERT_RANDOM == 1 ) {

    if ( numberBases == 0 )
      return 0;

    newSeq( &insertSeqStartPtr, insertSeqEndPtr, insertSeqLgth );
    int k;
    randomBaseNum = static_cast< int >( floor( ran2( rndInit ) * 4 ) );

    switch( randomBaseNum ) {

    case 0:
```

```
      randomBase = 't';
      break;

    case 1:
      randomBase = 'c';
      break;

    case 2:
      randomBase = 'a';
      break;

    case 3:
      randomBase = 'g';
      break;
    }

    appendFirstBase( randomBase, &insertSeqStartPtr, insertSeqEndPtr,
                     insertSeqLgth );

    for ( k = 1; k <= numberBases - 1; k++ ) {
      randomBaseNum = static_cast< int >( floor( ran2( rndInit ) * 4 ) );

      switch( randomBaseNum ) {

      case 0:
        randomBase = 't';
        break;

      case 1:
        randomBase = 'c';
        break;

      case 2:
        randomBase = 'a';
        break;

      case 3:
        randomBase = 'g';
        break;
      }

      appendBase( randomBase, insertSeqEndPtr, insertSeqLgth );
    }

    return insertSeqStartPtr;
}


/* First determine whether the sequence should be taken from the left or
   right hand end of the TE - chosen at random with equal probability.
   For variable l_or_r 0 is left and 1 is right. */

l_or_r = static_cast< int >( floor( ran2( rndInit ) * NUM_TE_SEQUENCES ) );

/* Transfer the sequence (up to the number of bases chosen to be inserted)
   from the appropriate (left or right) sequence in sitedta.

   If the right sequence is being used then bases are taken first from the
   right hand end of the TE sequence, and placed in the rightmost position
   of the sequence for insertion (variable sequence) */

if ( numberBases == 0 )
  return 0;

newSeq( &insertSeqStartPtr, insertSeqEndPtr, insertSeqLgth );
int i, j;
```

```
    switch ( l_or_r ) {

      case 0:
        appendFirstBase( sitedta.teleft[ 0 ], &insertSeqStartPtr,
                         insertSeqEndPtr, insertSeqLgth );

          for ( i = 1; i <= numberBases - 1; i++ )
            appendBase( sitedta.teleft[ i ], insertSeqEndPtr, insertSeqLgth );

        break;

      case 1:
        r_te_length = strlen( sitedta.teright );
        appendFirstBase( sitedta.teright[ r_te_length - numberBases ],
          &insertSeqStartPtr, insertSeqEndPtr, insertSeqLgth );

        for ( j = r_te_length - numberBases + 1; j <= r_te_length - 1; j++ )
          appendBase( sitedta.teright[ j ], insertSeqEndPtr, insertSeqLgth );

        break;
    }

    return insertSeqStartPtr;
}

////////////////////////////////////////////////////////////////////////////
// Function that inserts a sub-sequence into a DNA sequence
// Takes a pointer to the insertion location (insertion occurs immediately
// after the pointed-to base) and a pointer to the start of the sub-sequence.
// Also takes reference to the old sequence length (modifies this to new
// length).  Returns 1 if successful, 0 if not

int insertSubSeq( Base *insertionPoint, Base *subSeqSt, Base *subSeqEnd,
                  unsigned long subSeqLength, unsigned long &oldSeqLength )
{
  if ( insertionPoint == 0 || subSeqSt == 0 || subSeqEnd == 0 )
    return 0;
  else {
    if ( insertionPoint->nextNode != 0 ) { // If the insertion point is not the
                                           // last base of the sequence
      insertionPoint->nextNode->prevNode = subSeqEnd;
      subSeqEnd->nextNode = insertionPoint->nextNode;
    }

    subSeqSt->prevNode = insertionPoint;
    insertionPoint->nextNode = subSeqSt;
    oldSeqLength += subSeqLength;
    return 1;
  }
}

////////////////////////////////////////////////////////////////////////////
// Function that moves a pointer forwards in a DNA sequence
// Takes a pointer to the sequence (any part of the sequence) and the number of
// bases to move.  Returns the new pointer if successful - if try to move off
// end of sequence then returns 0

Base *moveForward( Base *startPoint, unsigned long bases )
{
  if ( bases != 0 )

  for ( unsigned long i = 0; i <= bases - 1 && startPoint != 0; i++ )
    startPoint = startPoint->nextNode;

  if ( startPoint == 0 )
    return 0;
  else
```

```
    return startPoint;
}

////////////////////////////////////////////////////////////////////////////
// Function that moves a pointer backwards in a DNA sequence
// Takes a pointer to the sequence (any part of the sequence) and the number of
// base to move.  Returns the new pointer if successful - if try to move off
// end of sequence then returns 0

Base *moveBackwards( Base *startPoint, unsigned long bases )
{
  if ( bases != 0 )

    for ( unsigned long i = 0; i <= bases - 1 && startPoint != 0; i++ )
      startPoint = startPoint->prevNode;

  if ( startPoint == 0 )
    return 0;
  else
    return startPoint;
}

////////////////////////////////////////////////////////////////////////////
// Function that takes two arrays of target sites (one before insertion, one
// after) and compares the two to find which sites to add or remove from the
// overall list of target sites. Takes 3 arrays - original, search, and
// sitesNotPresent which stores the list of sites from original that are not in
// search. Returns the number of sites in sitesNotPresent

unsigned long tsitesNotPresent( Base *original[], Base *search[], Base
                                *sitesNotPresent[] )
{
  int siteMatch;
  unsigned long k = 0;
  unsigned long sitesToRemove = 0;

  for ( unsigned long i = 0; original[ i ] != 0; i++ ) {
    siteMatch = 0;

    for ( unsigned long j = 0; siteMatch == 0 && search[ j ] != 0; j++ )

      if ( original[ i ] == search[ j ] )
        siteMatch = 1;

    if ( siteMatch == 0 ) {
      sitesNotPresent[ k ] = original[ i ];
      sitesToRemove++;
      k++;
    }
  }

  return sitesToRemove;
}

////////////////////////////////////////////////////////////////////////////
// Function to modify a target site array (add new target sites, and remove
// those that have been destroyed). Takes array of sites to add, array of sites
// to remove, original target site array, number of TS to add and number of TS
// to remove

int tsArrayMod( Base *addArray[], Base *removeArray[], Base *tsArray[],
                int numToAdd, int numToRemove, unsigned long numTS )
{
  unsigned long siteIndex = 0;
  int offset = 0;
  int siteFound = 0;
```

```
int removeDecrement = 0;
int found;
int returnValue;

if ( numToRemove != 0 ) {

  for ( int a = 0; a <= numToRemove - 1; ) {
    found = 0;

    if ( numTS != 0 )

     for ( unsigned long b = 0; b <= numTS - 1 && found != 1; b++ )

       if ( removeArray[ a ] == tsArray[ b ] )
         found = 1;

     if ( found == 0 ) {

       for ( int c = a; removeArray[ c ] != 0; c++ )
         removeArray[ c ] = removeArray[ c + 1 ];

     numToRemove--;
     }
     else
       a++;
  }
}

// If there are no sites to add or remove then return
if ( numToRemove == 0 && numToAdd == 0)
  return 0;

// If there are only target sites to add
if ( numToRemove == 0 ) {

  for ( int k = 0; k <= numToAdd - 1; k++ ) // Run through the addArray and
                                            // append to the end of the
    tsArray[ numTS + k ] = addArray[ k ];   // tsArray

  return numToAdd;
}

// If there are only target sites to remove
if ( numToAdd == 0 ) {

  returnValue = -numToRemove;

  // Move through all the Tsites in the full array
  for ( unsigned long i = 0; tsArray[ i ] != 0; ) {
    siteFound = 0;

  // Cycle through all the TS to remove
  for ( int j = 0; j <= numToRemove - 1 && siteFound == 0; j++ )

    if ( tsArray[ i ] == removeArray[ j ] ) { // If one matches the current
                                              // TS in the full array
      siteFound = 1;                          // set siteFound to 1
      numToRemove--;

      while ( removeArray[ j ] != 0 ) { // Delete the TS just found from the
                                        // removeArray
        removeArray[ j ] = removeArray[ j + 1 ];
        j++;
      }

      for ( unsigned long x = i; tsArray[ x ] != 0; x++ ) // Delete the TS
                                                          // from the tsArray
```

```
            tsArray[ x ] = tsArray[ x + 1 ];
        }
        if ( siteFound == 0 ) // If a site wasn't found this cycle
          i++;
      }

      return returnValue;
    }

    // If the number of sites to remove equals the number of sites to add
    if ( numToRemove == numToAdd ) {
      unsigned long n = 0;

      for ( unsigned long l = 0; tsArray[ l ] != 0; l++ ) { // Move through all
                                                  // the TSites in the full array
        siteFound = 0;

        for ( int m = 0; m <= numToRemove - 1 && siteFound == 0; m++ ) // Cycle
                                                        // through all the Tsites
                                                        // to remove
          if ( tsArray[ l ] == removeArray[ m ] ) { // If one matches the current
                                                    // TS in the full array
            tsArray[ l ] = addArray[ n++ ];         // Copy the site to add over
                                                    // the site to remove
            siteFound = 1;
            numToRemove--;

            while ( removeArray[ m ] != 0 ) { // Delete the TS just found
                                              // from the removeArray
              removeArray[ m ] = removeArray[ m + 1 ];
              m++;
            }
          }
      }

      return 0;
    }

    // If there are more sites to add than there are to remove
    if ( numToRemove < numToAdd ) {
      unsigned long p = 0;
      returnValue = numToAdd - numToRemove;

      for ( unsigned long q = 0; tsArray[ q ] != 0; q++ ) { // Move through all
                                                  // the TSites in the full array
        siteFound = 0;

        // Cycle through all the TSites to remove
        for ( int r = 0; r <= numToRemove - 1 && siteFound == 0; r++ )

          if ( tsArray[ q ] == removeArray[ r ] ) { // If one matches the current
                                                    // TS in the full array
            tsArray[ q ] = addArray[ p++ ];         // Copy the site to add over
            siteFound = 1;                          // the site to remove
            numToRemove--;

            while ( removeArray[ r ] != 0 ) { // Delete the TS just found from
                                              // the removeArray
              removeArray[ r ] = removeArray[ r + 1 ];
              r++;
            }
          }
      }

      // Add the remainder of the sites to the end of the TS array
      for ( int s = p; s <= numToAdd - 1; s++ )
        tsArray[ numTS + s - p ] = addArray[ s ];
```

```
    return returnValue;
  }

  // If there are more sites to remove than there are to add
  if ( numToRemove > numToAdd ) {
    int t = 0;
    returnValue = numToAdd - numToRemove;

    for ( unsigned long u = 0; tsArray[ u ] != 0; ) { // Move through all the
                                                      // TSites in the full array
      siteFound = 0;

      // Cycle through all the TS to remove
      for ( int v = 0; v <= numToRemove - 1 && siteFound == 0; v++ )

        // If a TS to remove matches the current TS in the full array
        if ( tsArray[ u ] == removeArray[ v ] ) {
          siteFound = 1;
          numToRemove--;

          if ( t <= numToAdd - 1 ) // If there are still TS to add
            tsArray[ u++ ] = addArray[ t++ ]; // Copy the site to add over the
                                              // site to remove
          else

            for ( unsigned long y = u; tsArray[ y ] != 0; y++ ) // Delete the
                                                                // TS from the tsArray
              tsArray[ y ] = tsArray[ y + 1 ];

          while ( removeArray[ v ] != 0 ) { // Delete the TS just found from
                                            // the removeArray
            removeArray[ v ] = removeArray[ v + 1 ];
            v++;
          }
        }

      if ( siteFound == 0 ) // If a site wasn't found then increment to next TS
        u++
    }

    return returnValue;
  }

  // Should never reach this part
  cerr << "Error in tsArrayMod - have reached end of function\n";
  exit( 1 );
  return 0; // Never used but needed to stop compiler error
}


////////////////////////////////////////////////////////////////////////////
// Function that takes a pointer to a target site, and checks whether that
// target site still exists.  Returns 1 if target site is OK, 0 if not

int checkTs( Base *targetSite, char *tsSequence )
{
  int tsLength;

  tsLength = strlen( tsSequence );

  for ( int i = 0; i <= tsLength - 1 ; i++ ) {

    if ( tsSequence[ i ] == targetSite->base ) { // If the base matches the
                                                 // target site

      if ( i == tsLength - 1 ) // and if this is the last base to check then
```

```
                                    // return 1
        return 1;
      }
      else // If the base doesn't match the target site then return 0
        return 0;

      if ( !( targetSite = targetSite->nextNode ) ) // Move to next base - if try
                                                    // to run off the end of the
        return 0;                                   // sequence then return 0

    }

    cout << "Should never reach this point in checkTs\n";
    exit( 1 );
    return 0; // Will never execute this, but needed to stop compiler error
}

////////////////////////////////////////////////////////////////////////////
// Function that copies a sub-sequence from a given pointer for a given number
// of bases. Takes the start pointer, the number of bases to copy
// Returns a pointer to the new sub-sequence - returns 0 if try to move off end
// of sequence
// NB Doesn't return other sequence information eg. sequence length and pointer
// to end of sequence

Base *copySubSeq( Base *copyStart, Base **subSequenceEnd,
                  unsigned long copyBases )
{
  Base *subSeqSt, *subSeqEnd;
  unsigned long subSeqLength;

  if ( copyStart != 0 ) { // If the copyStart pointer points to a base
    newSeq( &subSeqSt, &subSeqEnd, subSeqLength );
    appendFirstBase( copyStart->base, &subSeqSt, &subSeqEnd, subSeqLength );
    copyStart = copyStart->nextNode;

    for ( unsigned long i = 0; i <= copyBases - 2 && copyStart != 0; i++ ) {
      // Until all bases copied or get to end of sequence
      appendBase( copyStart->base, &subSeqEnd, subSeqLength );
      copyStart = copyStart->nextNode;
    }

    if ( copyStart == 0 && subSeqLength < copyBases)
      return 0;  // If the loop exited because the end of the sequence was
                 // reached before all the bases were copied then return 0
    else {
      *subSequenceEnd = subSeqEnd;
      return subSeqSt; // ...otherwise return pointer to the list
    }
  }
  else
    return 0;
}

////////////////////////////////////////////////////////////////////////////
// Function that opens a DNA file and loads the contents into a sequence list
// Returns 1 if successful or 0 if failed

int loadSequence( Base **beginning, Base **end, unsigned long &sequenceLength,
                  char *inFileName )
{
  char tmpChar, tryAgain;
  ifstream inputFile;

  cout << "\n\nInput name of starting DNA file: ";
  cin >> inFileName;
  inputFile.open( inFileName, ios::in | ios::nocreate );
```

```
    while ( !inputFile ) {
      inputFile.clear( ios::goodbit );
      cout << "Input file could not be opened.  Press 'y' to try again, any other
              key to quit\n";

      // Filter out whitespace while get tryAgain value
      while ( tryAgain = cin.get(), tryAgain == ' ' || tryAgain == '\n' ||
                                    tryAgain == '\t' )
        ;

      if ( tryAgain != 'y' && tryAgain != 'Y' )
        return 0;

      cout << "\nFilename: ";
      cin >> inFileName;
      inputFile.open( inFileName, ios::in | ios::nocreate );
    }

    newSeq( beginning, end, sequenceLength );

    if ( ( inputFile >> tmpChar ) == 0) // If the first character is EOF then
                                        // return 0
      return 0;

    while ( !strchr( "atcg", tmpChar ) ) // Search through file until first valid
                                         // base found

      if ( ( inputFile >> tmpChar ) == 0 ) // If EOF is reached before a valid
                                           // base then return 0
        return 0;

    appendFirstBase( tmpChar, beginning, end, sequenceLength );

    while ( inputFile >> tmpChar )

      if ( strchr( "atcg", tmpChar) )
        appendBase( tmpChar, end, sequenceLength );

    return 1;
}

////////////////////////////////////////////////////////////////////////////
// Function that outputs a DNA sequence to a DNA file.  Takes pointer to
// sequence, and name of file to output to. Returns 1 if successful or 0 if
// failed

int outSequence( Base *outPtr, char *filename )
{
    int charCounter = 0;
    ofstream outputFile( filename, ios::out );

    if ( !outputFile )
      return 0;

    while ( outPtr != 0 ) {
      outputFile << outPtr->base;
      charCounter++;

      if ( charCounter % 60 == 0 ) {
        outputFile << endl;
        charCounter = 0;
      }

      outPtr = outPtr->nextNode;
    }
```

```
   return 1;
 }

 ////////////////////////////////////////////////////////////////////////////
 // Function that searches for a base and returns a pointer to the base -
 // returns pointer to 0 if not found

 Base *findBase( char searchBase, Base *searchPtr )
 {
   while ( searchPtr != 0 )

     if ( searchPtr->base == searchBase )
       return searchPtr; // If the required base is found then return pointer to
                         // it...
     else
       searchPtr = searchPtr->nextNode;  // ...otherwise move the searchPtr
                                         // along one node

   return 0;
 }

 ////////////////////////////////////////////////////////////////////////////
 // Function that searches for a base and returns a pointer to the base -
 // returns pointer to 0 if not found
 // Overloaded version of function that takes number of bases to search over
 // Function reduces number of bases to search as they are searched so calling
 // function knows how many bases have
 // yet to be checked if a match is found (so it can go back and check the rest)

 Base *findBase( char searchBase, Base *searchPtr,
                 unsigned long &basesToSearch )
 {
   while ( searchPtr != 0 && basesToSearch >= 1 ) {
     basesToSearch--;

     if ( searchPtr->base == searchBase )
       return searchPtr; // If the required base is found then return pointer to
                         // it...
     else
       searchPtr = searchPtr->nextNode;  // ...otherwise move the searchPtr
                                         // along one node
   }
   return 0;
 }

 ////////////////////////////////////////////////////////////////////////////
 // Searches for a sub-string within a DNA sequence, and returns a pointer to
 // the start of the sub-string. Returns 0 if sub-string not found

 Base *findString( char *searchString, Base *findPtr )
 {
   int charIndex, schStringLgth;
   int tsFound = 0;
   int tsNotPresent = 0;
   int baseNoMatch = 0;
   Base *stringFirst = 0;

   schStringLgth = strlen( searchString );

   if ( schStringLgth == 1 ) { // If the searchString is actually just one
                               // character, then call findBase instead
     stringFirst = findBase( *searchString, findPtr );

     if ( stringFirst != 0 )
       tsFound = 1;
   }
   else
```

```
    while ( tsFound == 0 && tsNotPresent == 0 ) { // Until the target site is
                                                  // found, or the target doesn't exist
      stringFirst = findBase( searchString[ 0 ], findPtr );  // Search for
                                                  // first base in sequence (index [0])

      if ( stringFirst != 0 ) { // If the first base is found
        findPtr = stringFirst->nextNode;
        baseNoMatch = 0;
        charIndex = 1;

        while ( baseNoMatch == 0 && tsFound == 0 && findPtr != 0 )

          if ( findPtr->base != searchString[ charIndex ] ) {  // If the base
                                          // pointed to by findPtr doesn't
            baseNoMatch = 1;              // match the searchString
            findPtr = stringFirst->nextNode;
          }
          else  // If the base does match the searchString

            if ( charIndex == schStringLgth - 1 )
              // If this is the last base of the target site...
              tsFound = 1;  // ...then mark the target site as found...
            else {
              findPtr = findPtr->nextNode;// ...otherwise move to the next base
              charIndex++;
            }
      }
      else // If the first base isn't found then the target site isn't present
        tsNotPresent = 1;
    }

  if ( tsFound == 1 )
    return stringFirst;
  else
    return 0;
}


////////////////////////////////////////////////////////////////////////////
// Searches for a sub-string within a DNA sequence, and returns a pointer to
// the start of the sub-string.  Returns 0 if sub-string not found
// Overloaded version takes max number of bases to search over

Base *findString( char *searchString, Base *findPtr,
                  unsigned long &numBasesToCheck )
{
  int charIndex, schStringLgth;
  int tsFound = 0;
  int tsNotPresent = 0;
  int baseNoMatch = 0;
  Base *stringFirst = 0;

  schStringLgth = strlen( searchString );

  if ( schStringLgth == 1 ) { // If the searchString is actually just one
                              // character, then call findBase instead
    stringFirst = findBase( *searchString, findPtr, numBasesToCheck );

    if ( stringFirst != 0 )
      tsFound = 1;
  }
  else

    while ( tsFound == 0 && tsNotPresent == 0 ) { // Until the target site is
                                        // found, or the target doesn't exist
      stringFirst = findBase( searchString[ 0 ], findPtr, numBasesToCheck );
```

```
      // Search for first base in sequence (index [0])

      if ( stringFirst != 0 ) { // If the first base is found
        findPtr = stringFirst->nextNode;
        baseNoMatch = 0;
        charIndex = 1;

        while ( baseNoMatch == 0 && tsFound == 0 && findPtr != 0 )

          if ( findPtr->base != searchString[ charIndex ] ) {  // If the base
                                                // pointed to by findPtr doesn't
            baseNoMatch = 1;                    // match the searchString
            findPtr = stringFirst->nextNode;
          }
          else                    // If the base does match the searchString

            if ( charIndex == schStringLgth - 1 ) // If this is the last base
                                                  // of the target site...
              tsFound = 1;       // ...then mark the target site as found...
            else {               // ...otherwise move to the next base
              findPtr = findPtr->nextNode;
              charIndex++;
            }
      }
      else // If the first base isn't found then the target site isn't present
        tsNotPresent = 1;
      }

  if ( tsFound == 1 )
    return stringFirst;
  else
    return 0;
}

////////////////////////////////////////////////////////////////////////////
// Function to record all target sites in a sequence

void recordSites( Base *startPtr, siteinfo sitedata[],
                  Base *sitePos[][ NO_OF_SITES ] )
{
  Base *firstBase; // This is the first base of the sequence that will be
                   // passed to targetSiteSearch
                   // This stops target sites being recorded when they are too
                   // near the start of the sequence
  int firstBaseOffset;

  // Cycle through each target site type and perform a targetSiteSearch
  for ( int i = 0; i <= NO_OF_SITE_TYPES - 1; i++ ) {
    tsArrayInit( sitePos[ i ], NO_OF_SITES );

    if ( ( firstBaseOffset = sitedata[ i ].copy_length_upper -
                             strlen( sitedata[ i ].site ) ) > 0 )
      firstBase = moveForward( startPtr, firstBaseOffset );
    else
      firstBase = startPtr;

    // NB Doesn't pass the start of the DNA sequence
    sitedata[ i ].numbersites = targetSiteSearch( sitedata[ i ].site,
                                                  firstBase, sitePos[ i ] );


  }
}

////////////////////////////////////////////////////////////////////////////
// Takes a target site sequence, a DNA sequence and an array of pointers to
// bases. Searches the DNA sequence for target sites and places a pointer to
```

```
// each in the array. Returns the number of target sites found

unsigned long targetSiteSearch( char *tsSequence, Base *dnaSeq,
                                Base *tsArray[] )
{
  unsigned long tsNum = 0;
  Base *searchPtr = dnaSeq;

  while ( tsArray[ tsNum ] = findString( tsSequence, searchPtr ) ) {
    searchPtr = tsArray[ tsNum ]->nextNode;
    // Reset search pointer to base after last target site
    tsNum++;
  }

  return tsNum;
}

////////////////////////////////////////////////////////////////////////////
// Searches for target sites over a given number of bases from a given point
// and loads into a temporary tsarray

void targetSiteSearchRegion( char *tsSequence, Base *searchStart, Base
  *tsTempArray[], unsigned long basesToCheck )
{
  Base *srchPtr = searchStart;
  unsigned long newTSNum= 0;

  while ( ( tsTempArray[ newTSNum ] = findString( tsSequence, srchPtr,
                                    basesToCheck ) ) &&
                                    basesToCheck >= 1 ) {
    srchPtr = tsTempArray[ newTSNum ]->nextNode;
    newTSNum++;
  }
}

////////////////////////////////////////////////////////////////////////////
// Function that initialises a target site array (all elements initialised to
// 0)

void tsArrayInit( Base *array[], unsigned long arraySize )
{
  for ( unsigned long i = 0; i <= arraySize - 1; i++ )
    array[ i ] = 0;
}

////////////////////////////////////////////////////////////////////////////
// Function to append the first base in a new sequence.
// Makes stPtr point to the first node in the list

void appendFirstBase( char firstBase, Base **stPtr, Base **finalNodePtr,
                      unsigned long &seqLgth )
{
  // Initialize the first node
  *stPtr = new Base;
  ( *stPtr )->nextNode = 0;
  ( *stPtr )->prevNode = 0;
  ( *stPtr )->base = firstBase;

  // Move the seqEndPtr to the last node (also the first node)
  *finalNodePtr = *stPtr;

  seqLgth++;
}

////////////////////////////////////////////////////////////////////////////
// Takes a char and appends it onto the end of the DNA sequence (end of the
// sequence is pointed to by seqEndPtr). Then moves seqEndPtr to point to the
```

```
// last node, and returns the new seqEndPtr
void appendBase( char newBase, Base **seqEndPtr, unsigned long &seqLength )
{
  Base *tempPtr;

  // Initialize the new node
  tempPtr = new Base;
  tempPtr->nextNode = 0;
  tempPtr->prevNode = *seqEndPtr;
  tempPtr->base = newBase;

  // If this is not the first node then add the new node to the end of the
  // sequence
  if ( *seqEndPtr != 0 )
    ( *seqEndPtr )->nextNode = tempPtr;

  // Move the seqEndPtr to the last node
  *seqEndPtr = tempPtr;

  seqLength++;
}

////////////////////////////////////////////////////////////////////////////
// Function to start a new sequence - initialises the sequence start pointer,
// the end pointer and the length

void newSeq( Base **seqStPtr, Base **seqEndPtr, unsigned long &seqLen )
{
  *seqStPtr = *seqEndPtr = 0;
  seqLen = 0;
}

////////////////////////////////////////////////////////////////////////////
// Takes a pointer to the start of a sequence and prints the sequence

void printSequence( Base *printPtr )
{
  while ( printPtr != 0 ) {
    cout << printPtr->base;
    printPtr = printPtr->nextNode;
  }
}

////////////////////////////////////////////////////////////////////////////
// Takes a pointer to the pointer to the end of a sequence and deletes the
// sequence

void deleteSequence( Base **strt, Base **end, unsigned long &len )
{
  Base *deletePtr = *end;
  unsigned long numDeleted = 0;

  while ( deletePtr->prevNode != 0 ) { // Until get to last node
    *end = deletePtr->prevNode; // Move the endPtr to penultimate node (soon to
                                // be the new end node)
    delete deletePtr; // Delete the end node
    deletePtr = *end; // Make deletePtr point to the new end node
    cout << "Deleting: " << numDeleted++ << "\r";
  }

  // When have reached the last node of the list
  delete deletePtr;
  *strt = 0;
  *end = 0;
  len = 0; // Set the sequence length to 0
}
```

```
////////////////////////////////////////////////////////////////////////////
// Takes a pointer to the pointer to the end of a sequence and deletes the
// sequence.  Overloaded version for deletions of bases within the sequence
// (length and start pointer are not important

void deleteSequence( Base **end )
{
  Base *deletePtr = *end;

  while ( deletePtr->prevNode != 0 ) { // Until get to last node
    *end = deletePtr->prevNode; // Move the endPtr to penultimate node (soon to
                                // be the new end node)
    delete deletePtr; // Delete the end node
    deletePtr = *end; // Make deletePtr point to the new end node
  }

  // When have reached the last node of the list
  delete deletePtr;
  *end = 0;
}

////////////////////////////////////////////////////////////////////////////
// Function that searches the write_at array and returns the highest value (the
// number of cycles that need to be performed)

unsigned long getCycles( unsigned long write[] )
{
  unsigned long temp = 0;

  for ( int i = 0; i <= NO_OUTPUT_FILES - 1; i++ )

    if ( write[ i ] > temp )
      temp = write[ i ];

  return temp;
}

////////////////////////////////////////////////////////////////////////////
// Function to calculate the total of all the ratios used in selecting sites.
// This is used to randomly select a site type for insertion

int totalOfRatios( siteinfo sitedat[] )
{
  int total = 0;

  for ( int ratio_index = 0; ratio_index <= NO_OF_SITE_TYPES - 1;
        ratio_index++ )
    total += sitedat[ ratio_index ].ratio;

  return total;
}

////////////////////////////////////////////////////////////////////////////
// Function to ouput information about the simulation run

int outputInfo( siteinfo st_dat[], char input_file[], unsigned long cycles,
                unsigned long out_array[] )
{
  ofstream fp_details;
  char file[ MAX_PATH_LENGTH ];

  file[ 0 ] = '\0';

  /* Make file from the path to results directory plus the name of
     detail file */
  strcat(file, RES_DIR);
  strcat(file, DETAIL_FILE);
```

```
fp_details.open( file, ios::out );

if ( !fp_details )
  return 0;

/* Print to file the details of the simulation */
fp_details << "Start sequence: " << input_file << endl;
fp_details << "Number of cycles: " << cycles << endl;
fp_details << "Output at: ";

for ( int inf_ind = 0; inf_ind <= NO_OUTPUT_FILES - 2; inf_ind++ )
  fp_details << out_array[ inf_ind ] << ", ";

fp_details << out_array[ inf_ind ] << endl;
fp_details << "Ratio of insertion/excision events: " << RATIO_INSERT << endl;
fp_details << "Ratio of mutations: " << RATIO_PT_MT << endl;

if ( RATIO_PT_MT != 0 )
  fp_details << "Transition : Transversion ratio is " << RATIO_TRANSIT <<
                ":" << RATIO_TRANSVER << endl;

fp_details << endl;

for ( int info_index = 0; info_index <= NO_OF_SITE_TYPES - 1;
      info_index++ ) {
  fp_details << "Site sequence: " << st_dat[ info_index ].site << "\t";

  if ( st_dat[ info_index ].copy_length_lower !=
       st_dat[ info_index ].copy_length_upper ) {
    fp_details << "Copy length range: " <<
                  st_dat[ info_index ].copy_length_lower << " - ";
    fp_details << st_dat[ info_index ].copy_length_upper << "\t";
  }
  else
    fp_details << "Copy length: " << st_dat[ info_index ].copy_length_lower
               << "\t";

  fp_details << "Ratio: " << st_dat[ info_index ].ratio << endl;
  fp_details << "Ratio perfect:(imperfect with insertion):
                (imperfect with deletion) is ";
  fp_details << st_dat[ info_index ].perf_ratio << ":";
  fp_details << st_dat[ info_index ].im_ins_ratio << ":";
  fp_details << st_dat[ info_index ].im_del_ratio << endl;

  if ( st_dat[ info_index ].im_ins_ratio != 0 ) {
    fp_details << "Insertion range: " << st_dat[ info_index ].ins_lower <<
                  "-";
    fp_details << st_dat[ info_index ].ins_upper << "\t";
  }

  if ( st_dat[ info_index ].im_del_ratio != 0 ) {
    fp_details << "Deletion range: " << st_dat[ info_index ].del_lower <<
                  "-";
    fp_details << st_dat[ info_index ].del_upper;
  }

  fp_details << endl;

  if ( st_dat[ info_index ].im_ins_ratio != 0 ) {
    fp_details << "Left TE: " << st_dat[ info_index ].teleft;
    fp_details << "\nRight TE: " << st_dat[ info_index ].teright << endl;
  }

  if ( st_dat[ info_index ].im_ins_ratio != 0 ||
       st_dat[ info_index ].im_del_ratio != 0 )
    fp_details << endl;
```

```
    }

    return 1;
}

////////////////////////////////////////////////////////////////////////////
// Function to check that the sequence contains target sites
// Returns 1 if sites are present, 0 if no sites are present

int checkSitesPresent( siteinfo sdata[] )
{
  // Cycle through all the site types, continuing to the next if there are no
  // sites present
  for ( int i = 0; sdata[ i ].numbersites == 0 && i <= NO_OF_SITE_TYPES - 1;
        i++ )
    ;


  /* If the loop was exited because it ran out of site types ie. all the
     site types were present, then !TRUE is returned, indicating that
     sites are present */

  if ( i == NO_OF_SITE_TYPES ) // If reached the end of the loop (ie. no sites
                               // present for each TS type
    return 0;
  else
    return 1;
}
```

5.2 *DNASIM2CFG.H*

```
/* Header file containing all the information that might be varied from one run
   of the simulation to another */

#ifndef DNA_CFG
#define DNA_CFG


/* SECTION 1 - Folder information */

#define RES_DIR "/dos/data/real5/"
#define DETAIL_FILE "details"
#define MAX_PATH_LENGTH 60   /* Length of path to files in RES_DIR,
                           including DETAIL_FILE itself */


/* SECTION 2 - Parameters regularly changed */
/* TARGET SITE INFORMATION */

#define NO_OF_SITE_TYPES 3

#define MAX_SITE_LENGTH 101    /* Length of array (includes '\0') */
#define NO_OF_TMP_SITES 20 // The maximum number of sites that can be used in
                           // the temp arrays
#define MAX_WINDOW_LENGTH 101  /* Must be at least MAX_SITE_LENGTH */
#define INS_MAXRANGE 10        /* Maximum no bases that will be inserted during
                                  imperfect insertion/excision events */

#define SN1 "tat" /*gcccttaccccaaagcagggcagccatct"*/
#define CLU1 6  /* Copy length upper limit */
#define CLL1 6   /* Copy length lower limit */
#define RT1 1    /* Ratio of insertions at this site vs others*/
#define PRT1 1   /* Ratio of insertions that are perfect */
#define IIRT1 3  /* Ratio of insertions imperfect with extra insertions */
```

```
#define IDRT1 3  /* Ratio of insertions imperfect with extra deletions */
#define IUP1 4   /* Extra insertions upper limit */
#define ILR1 2   /* Extra insertions lower limit */
#define DUP1 4   /* Extra deletions upper limit */
#define DLR1 2   /* Extra deletions lower limit */
#define TELEFT1 "agaattcgga"   /* Left end of TE (ggggggggggg-TE) */
#define TERIGHT1 "caccgtgctt"  /* Right end of TE (TE-ccccccccccc) */

#define SN2 "tca"
//"tcctacacacagggacagtcatagtcctcccggaggacgacctacccaccacaatgttccaacccgtgagggctcc
ctgtatccagactgcctggt"
#define CLU2 6
#define CLL2 6
#define RT2 3
#define PRT2 1
#define IIRT2 1
#define IDRT2 1
#define IUP2 4
#define ILR2 2
#define DUP2 4
#define DLR2 2
#define TELEFT2 "cagggagatc"
#define TERIGHT2 "actagcgcca"

#define SN3 "gga"
#define CLU3 6
#define CLL3 6
#define RT3 6
#define PRT3 0
#define IIRT3 1
#define IDRT3 1
#define IUP3 6
#define ILR3 1
#define DUP3 6
#define DLR3 1
#define TELEFT3 "ccttgagagc"
#define TERIGHT3 "cccaccgtta"
/*
#define SN4 "att"
#define CLU4 6
#define CLL4 6
#define RT4 18
#define PRT4 1
#define IIRT4 1
#define IDRT4 1
#define IUP4 4
#define ILR4 2
#define DUP4 4
#define DLR4 2
#define TELEFT4 "ttaaggaaac"
#define TERIGHT4 "ctttttatta"

#define SN5 "cgtg"
#define CLU5 6
#define CLL5 6
#define RT5 1
#define PRT5 1
#define IIRT5 0
#define IDRT5 0
#define IUP5 0
#define ILR5 0
#define DUP5 0
#define DLR5 0
#define TELEFT5 "ggggggggggg"
#define TERIGHT5 "ccccccccccc"

#define SN6 "ttaagtc"
```

```
#define CLU6 6
#define CLL6 6
#define RT6 7
#define PRT6 1
#define IIRT6 0
#define IDRT6 0
#define IUP6 0
#define ILR6 0
#define DUP6 0
#define DLR6 0
#define TELEFT6 "gggggggggg"
#define TERIGHT6 "cccccccccc"

#define SN7 "tat"
#define CLU7 6
#define CLL7 6
#define RT7 10
#define PRT7 1
#define IIRT7 0
#define IDRT7 0
#define IUP7 0
#define ILR7 0
#define DUP7 0
#define DLR7 0
#define TELEFT7 "gggggggggg"
#define TERIGHT7 "cccccccccc"

#define SN8 "attc"
#define CLU8 4
#define CLL8 4
#define RT8 2
#define PRT8 1
#define IIRT8 0
#define IDRT8 0
#define IUP8 0
#define ILR8 0
#define DUP8 0
#define DLR8 0
#define TELEFT8 "gggggggggg"
#define TERIGHT8 "cccccccccc"

#define SN9 "ttagc"
#define CLU9 6
#define CLL9 6
#define RT9 1
#define PRT9 1
#define IIRT9 0
#define IDRT9 0
#define IUP9 0
#define ILR9 0
#define DUP9 0
#define DLR9 0
#define TELEFT9 "gggggggggg"
#define TERIGHT9 "cccccccccc"

#define SN10 "tcgtgac"
#define CLU10 10
#define CLL10 10
#define RT10 2
#define PRT10 1
#define IIRT10 0
#define IDRT10 0
#define IUP10 0
#define ILR10 0
#define DUP10 0
#define DLR10 0
#define TELEFT10 "gggggggggg"
```

```
#define TERIGHT10 "ccccccccc"
*/

#define SITEDETAILS_DECLARATION siteinfo sitedetails[NO_OF_SITE_TYPES] = {{SN1,
NS, CLU1, CLL1, RT1, PRT1, IIRT1, IDRT1, IUP1, ILR1, DUP1, DLR1, TELEFT1,
TERIGHT1}, {SN2, NS, CLU2, CLL2, RT2, PRT2, IIRT2, IDRT2, IUP2, ILR2, DUP2,
DLR2, TELEFT2, TERIGHT2}, {SN3, NS, CLU3, CLL3, RT3, PRT3, IIRT3, IDRT3, IUP3,
ILR3, DUP3, DLR3, TELEFT3, TERIGHT3}/*, {SN4, NS, CLU4, CLL4, RT4, PRT4, IIRT4,
IDRT4, IUP4, ILR4, DUP4, DLR4, TELEFT4, TERIGHT4}, {SN5, NS, CLU5, CLL5, RT5,
PRT5, IIRT5, IDRT5, IUP5, ILR5, DUP5, DLR5, TELEFT5, TERIGHT5}, {SN6, NS, CLU6,
CLL6, RT6, PRT6, IIRT6, IDRT6, IUP6, ILR6, DUP6, DLR6, TELEFT6, TERIGHT6},
{SN7, NS, CLU7, CLL7, RT7, PRT7, IIRT7, IDRT7, IUP7, ILR7, DUP7, DLR7, TELEFT7,
TERIGHT7}, {SN8, NS, CLU8, CLL8, RT8, PRT8, IIRT8, IDRT8, IUP8, ILR8, DUP8,
DLR8, TELEFT8, TERIGHT8}, {SN9, NS, CLU9, CLL9,  RT9, PRT9, IIRT9, IDRT9, IUP9,
ILR9, DUP9, DLR9, TELEFT9, TERIGHT9}, {SN10, NS, CLU10, CLL10, RT10, PRT10,
IIRT10, IDRT10, IUP10, ILR10, DUP10, DLR10, TELEFT10, TERIGHT10}*/}

/* SECTION 3 - Output information */

#define NO_OUTPUT_FILES 13
#define OUT1 1 /*1*/
#define OUT2 13 /*10*/
#define OUT3 67 /*50*/
#define OUT4 133 /*100*/
#define OUT5 400 /*300*/
#define OUT6 667 /*500*/
#define OUT7 1333 /*1000*/
#define OUT8 2667 /*2000*/
#define OUT9 6667 /*5000*/
#define OUT10 13333 /*10000*/
#define OUT11 26667 /*20000*/
#define OUT12 66667 /*50000*/
#define OUT13 133333 /*100000*/


#define WRITE_AT_DECLARATION unsigned long write_at[NO_OUTPUT_FILES] = {OUT1,
OUT2, OUT3, OUT4, OUT5, OUT6, OUT7, OUT8, OUT9, OUT10, OUT11, OUT12, OUT13}


/* SECTION 4 */
/* Additional parameters - imprecise excision and point mutations */

#define RATIO_PT_MT 1
#define RATIO_INSERT 3
#define RATIO_TRANSIT 2
#define RATIO_TRANSVER 1
#define INSERT_RANDOM 0  /* If this is set to 1 then all bases inserted in
imprecise excisions are chosen at random - TE end sequences are not used */


/* SECTION 5 - Data analysis */
/* Parameters used by the supplementary programs for data analysis */

#define ZIPF_WIN 4  /* The length of the sequences produced by zipf analysis */

#define ZIPF 1
#define FLUCT 1   /* If 1 then the analysis is performed, 0 it is not */
#define DISP 0

#define ZIPF_ID 'z'   /* Appended to the end of the analysed data file */
#define FLUCT_ID 'f'  /* eg.  1000f is the fluctuation data derived from the */
#define DISP_ID 'd'   /* 1000 iterations file */


/* Fluctuation and displacement calculation parameters */
```

```
#define FLUCT_WINDOW_LENGTH_ARRAY_DEFINITION int
window_length[NUMBER_OF_WINDOWS] = {2, 5, 10, 20, 50, 100, 200, 500, 1000};

#define NUMBER_OF_WINDOWS 9
#define MAX_FLUCT_WINDOW_LGTH 1000

/* SECTION 6 - Infrequently changed parameters */

#define MAX_DNA_ARRAY_LENGTH 1100000L
#define MAX_FILENAME_LENGTH 40    /* Length of dna filename (includes '\0') */
#define MAX_NO_CYCLES 10000100L
#define NO_OF_SITES 500000L   /* The maximum instances of any one site type */
//80000

#define OPENED 1  /* Used in function to open files */

#define NUM_TE_SEQUENCES 2 /* Used in get_insert_sequence to choose left or
                           right end of TE for insertion bases */

#endif
```

References

1) M. Pagel, R. Johnstone, Proc. R. Soc. Lond. B, 249, 119-124 (1992)

2) B. Venkatesh, P. Gilligan, S. Brennar, FEBS Letters, 476, 3-7 (2000)

3) R. N. Mantegna, et al., Physical Review Letters 73, 3169-3172 (1994)

4) P. Baldi, S. Brunak, Y. Chauvin, A. Krogh, J. Mol. Biol. 263, 503-510 (1996)

5) R. Tijan, T. Maniatis, Cell, 77, 5-8 (1994)

6) S. Ogbourne, T. M. Antalis, Biochem. J. 331, 1-14 (1998)

7) B. Lewin, Genes V (Oxford University Press, Oxford, 1995)

8) T. J. Hadden, C. Ryou, R. E. Miller, Nucleic Acids Research 25, 3930-3936 (1997)

9) L. Stryer, Biochemistry 3rd ed. (W. H. Freeman and Company, New York, 1988)

10) B. Charlesworth, P. Sniegowski, W. Stephan, Nature 371, 215-220 (1994)

11) A. R. Lohe, A. J. Hilliker, P. A. Roberts, Genetics 134, 1149-1174 (1993)

12) O. Panaud, X. Chen, S. R. McCouch, Genome 38, 1170-1176 (1995)

13) P. J. Maughan, M. A. Saghai Maroof, G. R. Buss, Genome 38, 715-723 (1995)

14) C. Moran, Journal of Heredity 84, 274-280 (1993)

15) M. S. Röder, J. Plaschke, S. U. König, A. Börner, M. E. Sorrells, S. D. Tanksley, M. W. Ganal, Mol. Gen. Genetics 246, 327-333 (1995)

16) K. Wu, S. D. Tanksley, Mol. Gen. Genetics 241, 225-235 (1993)

17) P. Pasero, N. Sjakste, C. Blettry, C. Got, M. Marilley, Nucleic Acids Research 21, 4703-4710 (1993)

18) Z. Q. Ma, M. Röder, M. E. Sorrells, Genome 39, 123-130 (1996)

19) D. H. Shain, R. T. Stone, J. Y. Yoo, M. X. Zuber, Genome 39, 230-233 (1996)

20) A. J. M. Matzke, F. Varga, P. Gruendler, I. Unfried, H. Berger, B. Mayr, M. A. Matzke, Chromosoma 102, 9-14 (1992)

21) M. Jamilena, C. Ruiz Rejón, M. Ruiz Rejón, Chromosoma 102, 272-278 (1993)

22) S. Hagemann, B. Scheer, D. Schweizer, Chromosoma 102, 312-324 (1993)

23) J. A. Adegoke, Ú. Árnason, B. Widegren, Chromosoma 102, 382-388 (1993)

24) W. S. Modi, Chromosoma 102, 484-490 (1993)

25) T. Hails, M. Jobling, A. Day, Chromosoma 102, 500-507 (1993)

26) P. Broun, S. D. Tanksley, Mol. Gen. Genet. 250, 39-49 (1996)

27) A. J. Flavell, Comp. Biochem. Physiol 110B, 3-15 (1995)

28) S. P. Goff, Annu. Rev. Genetics 26, 527-544 (1992)

29) J. M. Whitcomb, S. H. Hughes, Annu. Rev. Cell Biol. 8, 275-306 (1992)

30) F. Maggini, R. D'Ovido, M. T. Gelati, M. Frediani, R. Cremonini, M. Ceccarelli, S. Minelli, P. G. Cionni, Genome 38, 1255-1261 (1995)

31) W. Li, K. Kaneko, Europhysics Letters 17, 655-660 (1992)

32) C.-K. Peng, et al., Nature 365, 168-170 (1992)

33) S. V. Buldyrev, et al., Physical Review E 47, 4514-4523 (1993)

34) S. V. Buldyrev, et al., Biophysical Journal 65, 2673-2679 (1993)

35) H. E. Stanley, et al., Journal de Physique IV Colloque C1, 3, 115-25 (1993)

36) C.-K. Peng, et al., Physical Review E 49, 1685-1689 (1994)

37) S. V. Buldyrev, et al., Physical Review E 51, 5084-5091 (1995)

38) A. Czirók, R. N. Mantegna, S. Havlin, H. E. Stanley, Physical Review E 52, 445-452 (1995)

39) R. N. Mantegna, et al., Physical Review E 52, 2939-2950 (1995)

40) W. Li, T. G. Marr, K. Kaneko, Physica D 75, 392-416 (1994)

41) G. K. Zipf, Human Behaviour and the Principle of Least Effort (Addison-Wesley Press, Cambridge MA, 1949)

42) R. Wilson et al., Nature 32, 368 (1994)

43) I. Kanter, D. A. Kessler, Physical Review Letters 74, 4556-4562 (1992)

44) S. S. Sommer, Trends in Genetics 11, 141-147 (1995)

45) G. Levinson, G. A. Gutman, Nucleic Acids Research 15, 5323-5338 (1987)

46) C. Schlötterer, D. Tautz, Nucleic Acids Research 20, 211-215 (1992)

47) P. A. Peterson, Advances in Agronomy 51, 79-124 (1993)

48) T. Giraud, P. Capy, Proc. R. Soc. Lond. B 263, 1481-1486 (1996)

49) Peterson, CRC Crit. Rev. Plant. Sci. 6, 205-208 (1987)

50) Hartings, Mol. Gen. Genet. 227, 91-96 (1991)

51) Chomet, Genetics 129, 261-270 (1991)

52) M. L. Fitzgerald, D. P. Grandgenett, Journal of Virology 68, 4314-4321 (1994)

53) S. M. Miller, R. Schmitt, D. L. Kirk, The Plant Cell 5, 1125-1138 (1993)

54) S. B. Sandmeyer, L. J. Hansen, D. L. Chalker, Annu. Rev. Genet. 24, 491-518 (1990)

55) A. D. Cresse, S. H. Hubert, W. E. Brown, J. R. Lucas, J. L. Bennetzen, Genetics 140, 315-324 (1995)

56) H. M. Robertson, D. L. Lampe, Annu. Rev. Entomol. 40, 333-357 (1995)

57) H. Ji, D. P. Moore, M. A. Blomberg, L. T. Braiterman, D. F. Voytas, G. Natsoulis, J. D. Boeke, Cell 73, 1007-1018 (1993)

58) J. J. Collins, P. Anderson, Genetics 137, 771-781 (1994)

59) H. G. A. M. van Luenen, S. D. Colloms, R. H. A. Plasterk, Cell 79, 293-301 (1994)

60) K. Ünsal, G. T. Morgan, Journal of Molecular Biology 248, 812-823 (1995)

61) A. Domínguez, J. Albornoz, Mol. Biol. Evol. 251, 130-138 (1996)

62) D-S Suh, E-H Choi, T. Yamazaki, K. Harada, Mol. Biol. Evol. 12, 748-758 (1995)

63) S. V. Nuzhdin, T. F. C. Mackay, Genet. Res. Camb. 63, 139-144 (1994)

64) J. K. Lim, M. J. Simmons, BioEssays 16, 269-275 (1994)

65) S. Tavaré, B. W. Giddings, Mathematical Methods for DNA Sequences (CRC, Boca Raton, 1989)

66) G. S. Attard, A. C. Hurworth, J. P. Jack, Europhysics Letters 36, 391-396 (1996)

67) K. Shimotohno, Y. Takahashi, N. Shimizu, T. Gojobori, D. W. Golde, I. S. Y. Chen, M. Miwa, T. Sugimura, Proc. Natl. Acad. Sci. USA 82, 3101-3105 (1985)

68) W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, Numerical Recipies in C, Second Edition (Cambridge University Press, 1994)

69) R. F. Voss, Physical Review Letters 25, 3805-3808 (1992)

70) V. R. Chechetkin, A. Y. Turygin, J. Theor. Biol. 178, 205-217 (1996)

71) G. Dodin, P. Levoir, C. Cordier, J. Theor. Biol. 183, 341-343 (1996)