Creation of a Low Complexity Region Evolution Simulator for use in an Approximate Bayesian Computation

Alexander Turco

April 3, 2023

[1] Department of Biology, McMaster University, Hamilton, ON, Canada

Contents

Abstract

It has been shown that among eukaryotic proteomes, the most commonly shared peptide sequences tend to lack stable three-dimensional structures and contain low information and entropy content. These sequences, now known as low-complexity regions, lack diversity in amino acid composition, but recent research has demonstrated an important role played by these regions in a variety of cellular functions. Model-based analysis of amino acid sequences is a common approach utilized to study molecular evolution, however, due to the increasing complexity of available data, many model-based approaches have become intractable due to difficulties in calculating the likelihood function. To overcome the issue of intractability, this study employs an approximate Bayesian computation Markov chain Monte Carlo algorithm (ABC-MCMC), which utilizes a simulation step in place of calculating the likelihood function. Through the simulation of mutated protein sequences, this study aims to investigate the evolution of protein low-complexity regions by estimating parameters such as mutation and insertion/deletion rates. This work provides insight into the formation of low-complexity regions, and demonstrates the potential of using approximate Bayesian computation methods for studying evolutionary genetics.

Literature Review/Proposal

What are Low Complexity Regions?

For decades, it was believed that peptide sequences which lack the ability to form stable three-dimensional structures also lack specific biological function (Haerty and Golding 2010b). Interestingly, among eukaryotic proteomes, the most commonly shared peptide sequences are found to be sequences with a low information content which lack a stable three-dimensional
Order ref's by year or by author names
structure (Haerty and Golding 2010b; Marcotte et al. 1999; Bannen et al. 2007). These sequences have been termed 'low-complexity regions' (LCRs) due to their low information content and entropy, as well as their lack of diversity in amino acid composition (Wootton and Federhen 1993; Coletta et al. 2010). LCRs are found in DNA as well as protein sequences and can present in a variety of ways, all of which skew the composition of amino acids in a different manner (Wootton and Federhen 1993; Mier et al. 2020). Homorepeats, direpeats, tandem repeats, and imperfect repeats are common definitions of LCRs based on the periodicity of amino acids in a given sequence, but not every LCR is defined by a specific pattern (Mier et al. 2020).Most of the time, these patterns are found to occur in non-coding regions and evolve with minimal selective pressure (Kruglyak et al. 2000). Further research is being done in order to uncover the function of LCRs in protein coding regions, as well as theevolutionary background of these repetitive regions (Huntley and Golding 2006). To investigate the process of LCR evolution,this study proposes an Approximate Bayesian Computation (ABC) approach which will enable the prediction of evolutionary parameters such as mutation rates and insertion/deletion rates.

Characteristics and Types of LCRs

Algorithms to detect the presence of low complexity regions in a sequence are available, and continue to be improved with further research into LCRs. Wootton and Federhen (1993) first developed an algorithm called SEG to find low complexity regions in protein sequences using information content (Huntley and Golding 2002). Information content is a common characteristic used to identify low complexity regions and in order to calculate the amount of information within a segment, the SEG algorithm implements Shannon's entropy (Battistuzzi et al. 2016; Wootton and Federhen 1993). Shannon's entropy (Shannon 1948) has been commonly used as a measure of complexity of a string of characters (Battistuzzi et al. 2016; Coletta et al. 2010; Wootton and Federhen 1993). This study will use the SEG algorithm and therefore Shannon's entropy to assess the complexity of protein sequences. The less complex a sequence is (low variety of residues), the lower the entropy/information content of the sequence. Although LCRs are defined by their low information content, these regions have also been found to be hyper-mutable, and it is thought that throughout evolutionary history, they frequently gained and lost repeats (Marcotte et al. 1999; Kruglyak et al. 1998). In studying the Drosophila melanogaster gene mastermind, which encodes a highly repetitive nuclear
italics
protein, Newfeld et al. (1991) identified different patterns of evolutionary change between regions of high and low complexity. Repetitive regions were found to have a much higher rate of amino acid replacement, therefore the rate of evolution within these regions is higher than outside (Newfeld et al. 1991; Huntley and Golding 2000).

LCRs all share an overall low diversity of residues but present in unique ways as periodic or aperiodic repeats, which take on the form of homopolymers and heteropolymers (Wootton and Federhen 1993; Battistuzzi et al. 2016). Homopolymers/homorepeats are consecutive iterations of a single amino acid residue, and heteropolymers (direpeats, tandem repeats) are consective iterations of more than one residue that can be found in a variety of different patterns based on periodicity (Battistuzzi et al. 2016; Mier et al. 2020). Microsatellites, one of the best studied types of LCRs, commonly describe regions composed of tandem repeats that are typically made from anything between one to six nucleotides (Ellegren 2004). Although microsatellites normally refer to DNA sequences, it has been found that LCRs in proteins are comparable to microsatellites (DePristo et al. 2006). The molecular mechanisms involved in the process of evolution including slippage and unequal recombination are important for microsatellites and therefore protein LCRs as well (DePristo et al. 2006). A class of proteins which are related to, but slightly differ from LCRs are intrinsically disordered proteins (IDPs). IDPs are unable to form stable three dimensional structures and are characterized by low sequence complexity, biased amino acid composition, and high proportions of charged and hydrophilic residues (Wright and Dyson 2015). IDPs are composed of intrinsically disordered regions ~~which are not necessarily defined by a low~~ information content as LCRs are (Haerty and Golding 2010b; Dunker et al. 2002) ~~(Haerty and Golding 2010b; Dunker et al. 2002)~~. In this study, we propose a focus on low complexity regions.

Why care about LCRs?

Proteins continue to be a large area of research due to their involvement in vital cellular processes and many human diseases. In protein sequence databases such as Swiss-Prot, the increase in the number of sequences and organisms represented has subsequently led to a decrease in the proportion of proteins containing LCRs (Coletta et al. 2010). On top of this, there is a lack of representation of LCRs in the protein data bank (Huntley and Golding 2002). Despite this underrepresentation of LCRs, they are known to be associated with several human neurodegenerative diseases and are thought to serve important biological functions (Coletta et al. 2010; Huntley and Golding 2006). It has also been found that the proteins of Plasmodium falciparum (the human malaria parasite) contain a high incidence of LCRs which has further highlighted the importance of both the evolution and function of LCRs (Gardner et al. 2002; DePristo et al. 2006). LCRs can appear as trinucleotide repeats which form repeated units of three nucleotides and are a well known form of deleterious mutation in humans (Ross et al. 1993). These are found to be the cause of diseases including fragile X syndrome, myotonic dystrophy, spinal atrophy, muscular atrophy, and Huntington's disease (Ross et al. 1993). These diseases can be broadly classified into two distinct groups, translated polyglutamine triplet repeat diseases and untranslated triplet repeat diseases (Everett and Wood 2004). Polyglutamine triplet repeat diseases, such as Huntington's disease, result in the formation of protein aggregates in the cell and occur due to expanded repeats being translated into expanded polyglutamine residues (Everett and Wood 2004). Untranslated triplet repeat diseases such as myotonic dystrophy and fragile X syndrome differ from polyglutamine repeat diseases as they contain trinucleotide repeats which are not translated into expansion within a mutant protein (Everett and Wood 2004).

~~The persistence of LCRs within genomes provides good evidence of their beneficial functions Verstrepen et al. 2005~~.

Not all things that persist are beneficial

LCRs have been associated with important biological processes such as genetic recombination, antigen diversification, and protein-protein interactions (Karlin et al. 2002; Verstrepen et al. 2005; Kumari et al. 2015). The repetitve regions are thought to drive recombination events which alter genes and result in phenotypic variation (Verstrepen et al. 2005). In the genomes of Haemophilus influenzae and Neisseria meningitidis, LCRs are abundant and cause phase variation which gives the bacteria the ability to change their adherance patterns to host cells (Bayliss et al. 2001). This ultimately increases the fitness of the population and allows the bacteria to evade the host response (Bayliss et al. 2001). It was previously believed, based on structural evidence,that these hypermutable LCRs did not form stable structures but instead existed as solvent-exposed disordered coils (Wootton and Federhen 1993; Huntley and Golding 2002; DePristo et al. 2006). Using proteins from a non-redundant Protein Data Bank (PDB) dataset, Kumari et al. (2015) analyzed secondary structure content and surface accessibility and discovered that LCRs can form secondary structures within proteins. More specifically, in a large majority of identified LCRs, the analysis revealed the presence of more than one secondary structure, indicating that LCRs are found in regions where structure transition occurs (Kumari et al. 2015). Although more work is necessary to further understand the functions of LCRs, their role in genetic recombination, protein structure and function, and antigen diversity, highlight the importance of LCR research.

How do LCRs Evolve?

Although research surrounding the evolution of LCRs is lacking, there are two proposed mechanisms of microsatellite evolution, which can be applied to many forms of LCRs. The first, polymerase slippage or slipped strand mispairing, involves loops being formed in either the coding or template strand, which causes a misalignment of strands and results in either the insertion or deletion of repetitive motifs (Levinson and Gutman 1987; Ellegren 2004). It is believed that slipped strand mispairing is the predominant mode of mutation of LCRs, specifically in homopolymer sequences (Levinson and Gutman 1987). The second mechanism, unequal recombination, occurs when repetitive regions in homologous chromosomes do not align properly during meiosis, which results in the repetitive region being expanded in one chromosome and contracted in the other (Warren et al. 1997; Mirkin 2007). In order to gain more insight into the evolutionary background of a variety of organisms, researchers have created models of events such as slippage in order to estimate mutation rate and other evolutionary parameters (Kruglyak et al. 2000). In a study of 10,844 parent/child allele transfers at nine short tandem repeat loci, Brinkmann et al. (1998) discovered 23 mutations, all of which were either gains or losses of repeats. Of the 23 mutations, 22 were due to single repeat mutations, which is why it has been common to use the stepwise mutation model of Ohta and Kimura (1973),that assumes repetitive regions expand or contract by 1 unit at a specific mutation rate (Kruglyak et al. 2000; Brinkmann et al. 1998). There are however, major drawbacks of the stepwise mutation model including that lengths can become negative, and the collection of repeat lengths in a sample will not have a stationary distribution (Kruglyak et al. 2000). It was thought that more complex models of LCR evolution were necessary to gain more accurate results, thus Kruglyak et al. (1998) proposed a model that incorporated length dependent slippage events. This differed from the stepwise mutation model in that the balance between slippage events and point mutations produced an equilibrium distribution of repeats (Kruglyak et al. 1998).

Models of LCR formation including replication slippage support the historical belief that LCRs evolve neutrally. More recently, there has been increasing evidence suggesting that LCRs are also acted upon by selective pressure (Haerty and Golding 2010b). Kimura (1983) proposed the neutral theory of molecular evolution which suggests that selection does not play a role in the genetic diversity within and between species, rather genetic diversity is neutral (Nevo 2001). Evidence for the neutral evolution of LCRs relies on a large number of factors including both their lack of stable structure and function (Dunker et al. 2002; Haerty and Golding 2010b), and ability to frequently gain or lose repeats through replication slippage (Marcotte et al. 1999; Kruglyak et al. 1998; Huntley and Golding 2000). Support for a selective model of LCR evolution comes from the non-random patterns of changes within LCRs, the deleterious effect of their expansion in humans (Karlin et al. 2002), and their enrichment in proteins involved in transcription, DNA, protein binding, reproduction, and development (Huntley and Clark 2007; Haerty and Golding 2010a; Battistuzzi et al. 2016). In a study of orthologous mouse and human genes, Mularoni et al. (2007) found a significant negative correlation between repeat number and gene nonsynonomous substitution rate, indicating that proteins acted upon by strong selective pressure contain a large number of repeats conserved between the two species (Mularoni et al. 2007). Interestingly, the study also reported a significant positive correlation between repeat size difference and protein nonsynonymous substitution rate, demonstrating that events such as slippage and substitutions occur in proteins which undergo neutral evolution (Mularoni et al. 2007). It was later revealed in a study by Battistuzzi et al. (2016) in which 11 representative Apicomplexa genomes were analyzed, that neutral mechanisms were found to act on highly repetitive LCRs (homopolymers) whereas selective pressures were influenced by the heterogeneity and length of the LCR (Battistuzzi et al. 2016). This work only begins to unravel the complexities of the evolutionary patterns associated with LCRs.

What is an Approximate Bayesian Computation Markov chain Monte Carlo algorithm?

When studying molecular evolution, a common practice is to use model-based analyses of sets of DNA and amino acid sequences (Laurin-Lemay et al. 2022). This approach allows for the estimation of evolutionary genetic parameters such as mutation rates and insertion/deletion rates (Wu and Rodrigo 2015). Model-based statistical inference generally revolves around calculating the likelihood function, which represents the probability of the observed data under a chosen model (Sunnåker et al. 2013). The likelihood function therefore quantifies how well the data supports both the parameter values as well as the model (Sunnåker et al. 2013). However, due to an increase in the complexity and magnitude of available data, many current model-based analyses have become intractable by virtue of the likelihood function being difficult to calculate (Marjoram 2013). Approximate Bayesian computation (ABC) methods are rooted in Bayesian statistics and have been gaining popularity in areas such as genetics, as they bypass the calculation of the likelihood function (Sunnåker et al. 2013). The way in which they do this is by utilizing a simulation step in place of the calculation as a way to provide an estimate of the likelihood function (Marjoram 2013). Since there are many ways to approach a simulation, there are many different forms of ABCs. The more popular forms include ABC rejection methods, ABC Markov chain Monte Carlo methods (ABC-MCMC), and Sequential Monte Carlo ABC methods (ABC-SMC) (Marjoram 2013). This study proposes the use of an ABC-MCMC algorithm in order to estimate evolutionary parameters such as mutation and indel rates, and provide insight into the formation and evolution of protein LCRs.

The reason for proposing an ABC-MCMC in this study stems from the lack of a pre-existing model which explains how insertions and deletions work. Insertions and deletions alter the landscape of a sequence, making the likelihood calculation extremely challenging. Marjoram et al. (2003) originally proposed the algorithm for a MCMC method without the use of likelihoods. The algorithm first starts from a selected parameter value and proposes a move to a new parameter value based on a proposal distribution (Marjoram et al. 2003). Using this new parameter value, a dataset is then simulated and summary statistics are calculated, which makes it possible to quantitatively compare differences between the simulated dataset and the observed dataset (Marjoram et al. 2003). If the difference between summary statistics is small, the Hastings Ratio is calculated and the proposed parameter value can be accepted with a certain probability, then a new value is proposed and the process begins again (Marjoram et al. 2003). On the other hand if the difference in summary statistics between the observed and simulated data is very large, we propose a new parameter value and begin the algorithm again (Marjoram et al. 2003; Marjoram 2013). The use of this algorithm has enabled the analysis of complex problems which tend to arise in the areas of population genetics, ecology, epidemiology, and systems biology (Sunnåker et al. 2013). The group of Liepe et al. (2010) have been leaders in the use of ABCs for inference of genetic networks. This is evident through the creation of a software package they created called ABC SysBio which can implement ABC algorithms in a straightforward manner (Marjoram 2013; Liepe et al. 2010). Prior to the year 2000, there were essentially no papers published on ABCs (Marjoram 2013). As we enter into an era where larger and more complex data can be collected, the need for improved models is necessary, hence the large increase over the last decade in papers which mention ABC methods (Marjoram 2013).

How will we use an ABC-MCMC ~~- Oct 28 First draft~~

Using the algorithm mentioned above for an ABC-MCMC, this study aims to better understand the evolutionary background/formation of protein LCRs. An ABC-MCMC will enable the prediction of two important evolutionary parameters, mutation rate and indel rate. There is possibilty for the estimation of other parameters which will be explored upon investigating the first two. We will utilize amino acid sequences in this study, one being the SRP40 protein found in Saccharomyces cerevisiae, which is extremely biased in composition. This protein sequence will act as our observed data and we will compare this observed data to our simulated data.

In terms of a simulation, we will use C++ to first generate a random amino acid sequence of a certain length. This randomly generated sequence will then be mutated over a number of generations in a two-step process. The first process is

Poisson should be capitalized

to choose a random poisson deviate with a mean that is equal to the mutation rate multiplied by the total number of sites. A poisson distribution is used here because mutation is a rare event and rare events can be modelled using this distribution. The value of the poisson deviate yields the total number of sites in the simulated sequence which should be mutated at random. The second mutation process deals with amino acid expansion and in this case we iterate through each residue in the simulated protein sequence and scan for repeats. If a residue is part of a repeat, we take the total length of the repeat, multiply it by the mutation rate and use this value as the mean of a random exponential deviate. We use the exponential distribution as it models waiting times between events. Based on the random exponential deviates assigned, we select the lowest value which represents the residue that will change fastest, and we alter that residue to either delete or insert a repeat at that position.

Once we simulate a protein sequence for a number of generations under certain parameter values, we need to obtain a set of summary statistics and compare the summary statistics of the observed and simulated data. We have proposed summary statistics based off notable characteristics such as protein length, number of LCRs, and the average entropy of the LCRs. There is the possibility for additional summary statistic characteristics upon exploration of the initially proposed characteristics. To quantitatively compare the differences between the observed and simulated data, we propose using a distance measure between the two vectors of summary statistics. This distance is just the norm of the vector observed-simulated. Along with this, we also propose the use of a threshold as a way to assess how close the two datasets are. If the distance between the two vectors of summary statistics is larger than this threshold, we can not accept the proposed parameter value and the algorithm begins again. On the other hand, if the distance is very small, we can move forward in the algorithm and potentially accept the new parameter value.

We intend to run the simulation under the same parameters many times and take the average of the produced summary statistic vectors before calculating the distance between observed and simulated data. It is also worth noting that each time we begin the algorithm again, new parameter proposals will be selected using random normal deviates. We hope to see the distance between summary statistics being minimized upon every iteration of the algorithm as this means the simulated protein closely resembles the observed protein under specific parameters.

Materials and Methods (mid-year stuff Jan 20)

Custom scripts and commands utilized in this analysis can be found on `GitHub` at https://github.com/opticrom/abcmcmc-thesis4c12.

ABC-MCMC: The Algorithm

This study utilized the ABC Markov chain Monte Carlo algorithm, originally proposed by Marjoram et al. (2003). The algorithm begins from a randomly selected parameter value and follows the steps below.

1. If now at θ, propose a move to θ' according to a proposal distribution $q(\theta, \theta')$.
2. Simulate a dataset, D' using θ'.
3. If $D' \approx D$ proceed to step 4; else, output θ and return to 1.
4. Calculate the Hastings Ratio.

5. Accept, and output, the new θ' with probability h. Else return to, and output, θ, Go back to 1.

Normal is not a person's name and hence is not capitalized

A custom C++ script was written to iterate through the algorithm for a desired number of simulations. The Normal distribution was used to control how new parameter values were proposed. We simulated a dataset under the newly proposed parameter value, which consisted of a randomly generated protein sequence. The simulated protein was compared to a protein of known low complexity called SRP40, which is found in the model organism *Saccharomyces cerevisiae*. The protein sequence for SRP40 was obtained from the NCBI database. To quantitatively compare similarities between simulated and observed protein sequences, vectors of summary statistics (characteristics that describe the sequences) were created for each sequence.

The Euclidean distance between the observed and simulated protein vectors was calculated in order to determine if the newly proposed parameter value could be accepted. If the Euclidean distance between the SRP40 vector and the vector

and the distance calculated using the current

produced using the newly proposed parameter values was smaller than the distance between the SRP40 vector and the current parameter values, the newly proposed parameter values were accepted. If this distance was larger, a one sample t-test was employed to determine the probability of accepting the newly proposed parameter values.

Parameters and Summary Statistics

Two parameters were estimated in this study, mutation rate and insertion/deletion (indel) rate. Mutation rate referred to the rate at which a single amino acid in a protein sequence changed into a different amino acid. Indel rate referred to the rate at which an amino acid was deleted or inserted from a protein sequence. For the purpose of this study, to determine if the length of a repetitive region played a role in insertions, any amino acid that was inserted into the simulated protein sequence was the same unit as the previous amino acid in the sequence.

Summary statistics were utilized to capture important information about simulated and observed protein sequences in order to assess how similar the sequences were. Three summary statistics were used which included, the length of the protein sequence, the number of LCRs in the sequence, and the average entropy of the LCRs. To identify LCRs and their corresponding entropies, the Seg algorithm was implemented (Wootton and Federhen 1993). The following Seg parameters were utilized to search for LCRs in proteins; a window length of 15, a trigger segment complexity of 1.9, and an extension segment complexity of 2.2. We selected these due to previous research which demonstrated that these parameter values would better detect regions of low complexity in eukaryotes which are typically longer and contain more repetitive repeats (Huntley and Golding 2000). Summary statistics were stored in vectors and normalized in order to prevent large values (for example the length of the protein sequence) from dominating when calculating the Euclidean distance.

Simulation Step: Creation and Mutation of Protein Sequences

To bypass calculation of the likelihood function, a custom C++ script was written to simulate the generation and mutation of protein sequences over numerous generations. Two random proteins of desired length were generated first, one that would be mutated based on the current simulation parameters, and one mutated based on the proposed simulation parameters. We generated two protein sequences 400 amino acids in length, similar to the length of the SRP40 protein in *S. cerevisiae*. This simulated protein sequence was then mutated in the following ways.

We iterated over the simulated protein sequence and assigned exponential deviates to each amino acid. We utilized the exponential distribution because it is commonly used to model waiting times between events. Mutation and indel rates served to act as the scale parameter (β), or mean of the distribution, indicative of the mean time until mutation occurs. In the case of mutation rate, the same rate was utilized across all sites, with the assumption made that repeats do not play a role in point mutation. In the case of the indel rate, we scanned for repeats, and if found, we multiplied the length of the repetitive segment by the indel rate, and utilized this new value as the scale parameter (β) for the exponential distribution. THINK ABOUT WHAT THIS MEANS IN TERMS OF THE EXPONENTIAL DISTRIBUTION, THE LONGER THE LENGTH OF REPEAT, HOW DOES THIS AFFECT THE EXPONENTIAL DISTRIBUTION, DOES IT MAKE SENSE?. Yes should be good

Exponential deviates were stored in two vectors, one for deviates generated using mutation rate, and the other for deviates generated using the indel rate. We then identified a single deviate with the lowest value (based on both vectors), which represented the residue that mutated quickest. Depending on which vector the lowest deviate came from, we either altered the corresponding amino acid or inserted/deleted a repetitive amino acid. Upon mutating, deleting, or inserting an amino acid, we scanned the sequence to see if the mutation altered the landscape (inturrupted a repeat, created a repeat), and subsequently generated new deviates for the amino acids that were affected.

We ran the simulation ten times for each newly proposed parameter value and took the average of all ten vectors of summary statistics prior to calculating the distance. On each simulation, we mutated the protein sequence for 50 generations before obtaining summary statistics. We plan on testing the program in the future with various numbers of iterations.

Normalization and Euclidean Distance Calculation

To determine the similarity between simulated and observed protein sequences, the distance between the vectors of summary statistics was calculated by employing a Euclidean distance measure (4). Before calculating this distance, vectors of summary statistics were normalized to prevent large values (such as protein length) from dominating the distance measure.

In order to normalize all elements of the vector to be between 0 and 1, maximum and minimum values for each summary statistic were required. For the length of the simulated protein, an upper limit of 1.5x the length of the SRP40 protein, and a lower limit of 0.5x the length of the SRP40 protein were set. Due to the length of the SRP40 protein being 406 amino acids long, the upper limit was equal to 609 amino acids, and the lower limit was equal to 203 amino acids (1). In terms of the number of LCRs, the minimum value was set to 0 (no LCRs) and the maximum value was chosen based off the shortest LCR length. We selected 5 amino acids to be the length of the shortest LCR, resulting in the maximum number of LCRs being 81.2 (406/5) (2). Finally, for the average entropy of the LCRs, the minimum value was set to 0, and the maximum value was set to 4.3 (maximum entropy for amino acids). Normalization calculations are shown below.

$$normalized\_length = \frac{(simulated\_protein\_length - SRP40\_length + min\_length)}{SRP40\_length} \tag{1}$$

$$normalized\_number\_LCRs = \frac{number\_LCRs}{max\_number\_LCRs} \tag{2}$$

$$normalized\_average\_entropy = \frac{average\_entropy}{max\_entropy} \tag{3}$$

$$d(P1, P2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{4}$$

One Sample T-Test for Probability of Acceptance

A one sample t-test was employed in order to assess the probability of accepting newly proposed parameter values in cases where the Euclidean distance between vectors produced using the newly proposed parameters was larger than the distance between vectors produced using the current parameter values. For each set of newly proposed parameter values, 10 vectors of summary statistics were produced. From these 10 vectors, 10 distances were found, and the mean and standard deviation were calculated and subsequently used in the calculation of the t-statistic (5). Vectors of t-statistics and associated probabilities were stored, and the calculated t-statistic was used to estimate a p-value by linear interpolation.

$$t = \frac{\bar{x} - \mu}{\frac{S}{\sqrt{n}}} \tag{5}$$

You might want to add the rational for using the t-test.
You are testing if the new parameters are or are not
samples of data from the same distribution.

Results

Assessing the Evolution Simulator

To be able to utilize the evolution simulator within an approximate bayesian computation, tests had to be done to ensure it was working properly. The first step in this study involved running simulations of LCR evolution for various mutation and indel rates. Simulations were run for 1000 iterations each, and on each iteration, the simulator would return the average entropy of the LCRs in the sequence, the total number of LCRs in the sequence, and the overall sequence entropy. To test the effects No parentheses of each individual rate parameter, only 1 rate was adjusted while the other remained a fixed constant. (Figure 1) and (Figure 2) are the result of holding the mutation rate constant at 1, and only changing the insertion/deletion rate. It is expected that a low indel rate will lead to the formation of less LCRs and hence a higher sequence entropy. The results in (Figure 1) match this expectation, showing the formation of only one to two low complexity regions that are quickly removed by mutations, and a relatively unaffected maximal protein sequence entropy.

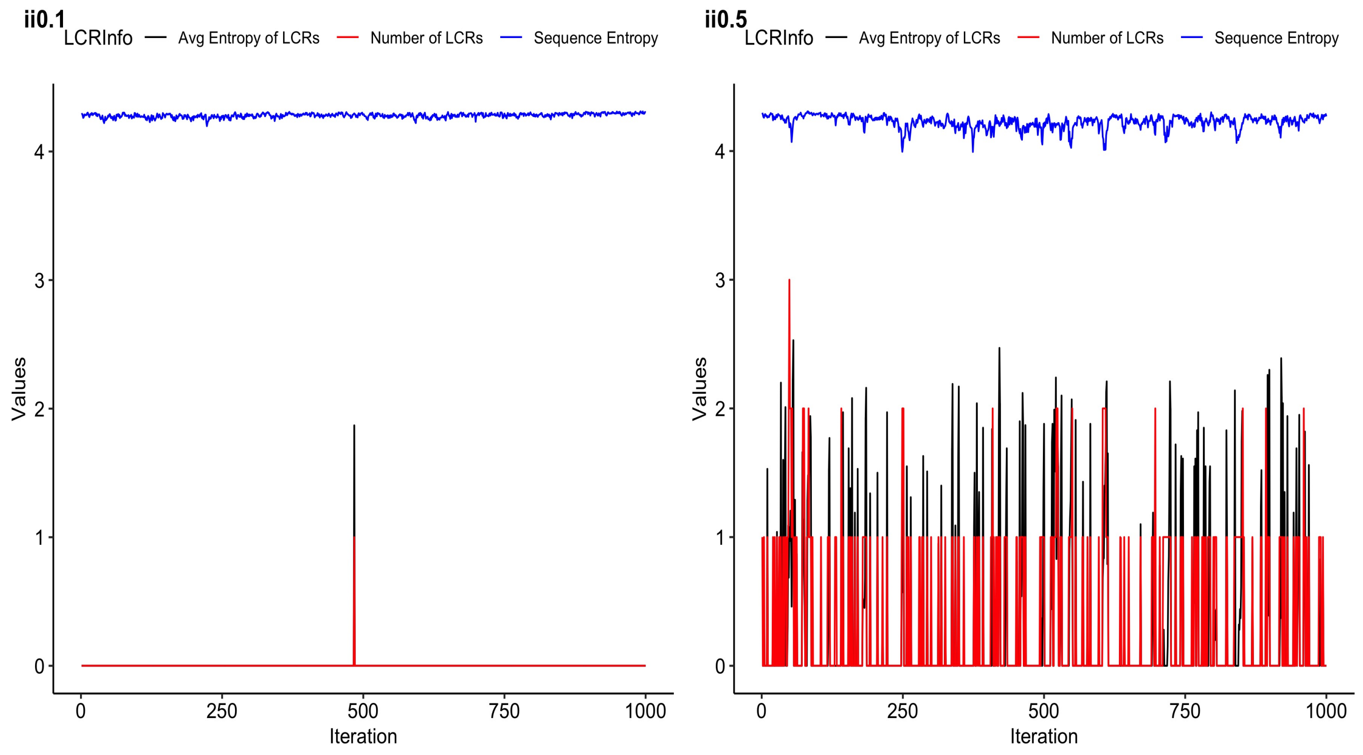

Figure 1: Entropy and number of LCR comparisons in a randomly generated protein sequence that has undergone simulated evolution. (Left) An indel rate of 0.1 was utilized. (RIGHT) An indel rate of 0.5 was utilized. A mutation rate of 1 was utilized for both.

As the indel rate is increased, it is expected that there should be more low complexity regions and hence, a lower average sequence entropy. Results in (Figure 2) show the formation of many more LCRs that remain in the protein sequence, possibly because mutations don't have as much of a chance swamp out the formation of low complexity regions. These results also show much more of a fluctuation in average sequence entropy compared to (Figure 1), with decreases in entropy occuring where new LCRs are being formed.

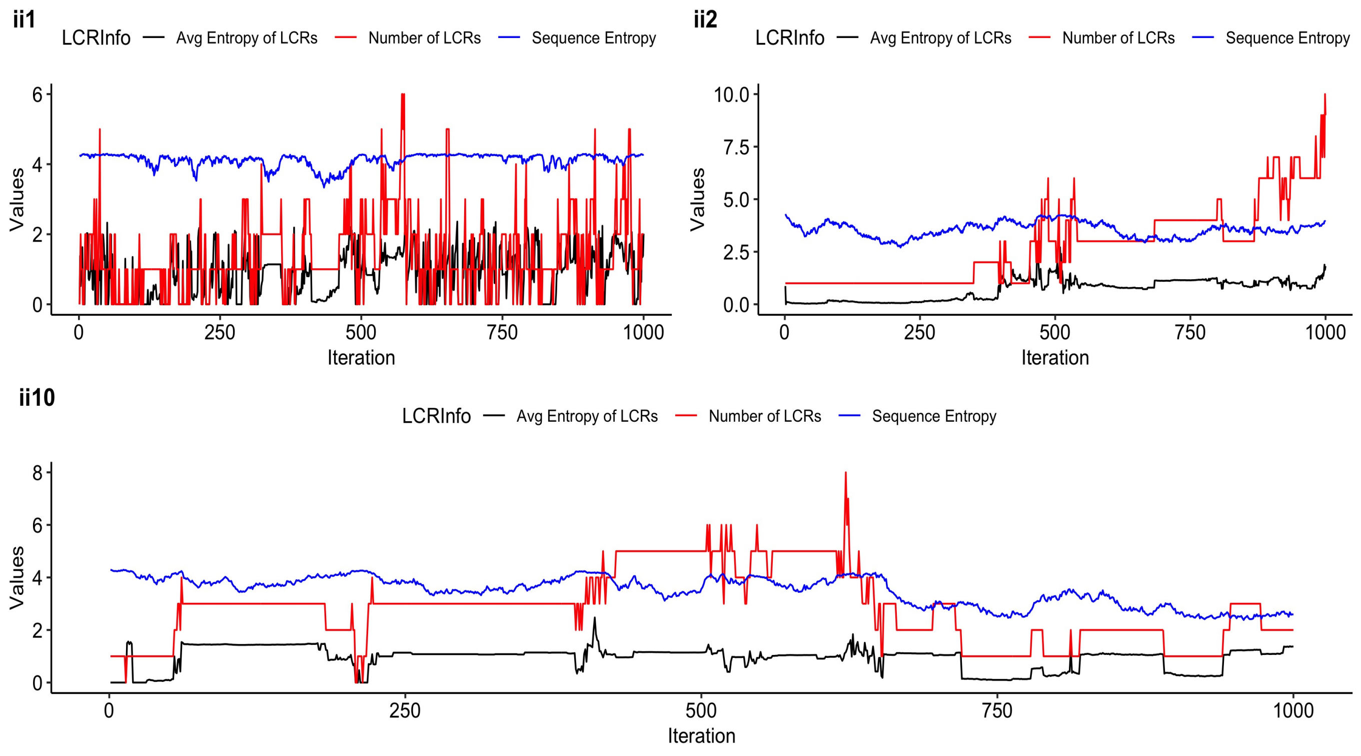

Figure 2: Entropy and number of LCR comparisons in a randomly generated protein sequence that has undergone simulated evolution. (TOP LEFT) An indel rate of 1 was utilized. (TOP RIGHT) An indel rate of 2 was utilized. (BOTTOM) An indel rate of 10 was utilized. A mutation rate of 1 was used for all.

To explore the effects of mutation on LCR formation, (Figure 3) and (Figure 4) show the average sequence entropy, the entropy of the LCRs in the sequence, and the total number of LCRs found in a randomly generated protein sequence that has undergone simulated mutation for 1000 generations. It is expected that under higher rates of mutation, indels will be difficult to form as the large number of mutations destroy repetitive segments. (Figure 3) utilized mutation rates of 0.01, 0.1 and 0.5, all of which are less than the fixed indel rate of 1. In all three cases, the average sequence entropy fluctuates as the number of LCRs remains fairly large. In (Figure 4), mutation rates are increased to 1, 2, and 10, much larger than the fixed indel rate. The average sequence entropy fluctuates much less in these cases and the number of LCRs in the sequence decreases in comparison to (Figure 3). These results match the expectation that a larger mutation rate will have major effects of insertions and deletions. In the case where we set the mutation rate to be 10x the indel rate, only three LCRs are formed in total and are quickly removed due to mutation. In this case the entropy is also relatively unaffected, meaning that repetitve regions are not being formed.

It should also be noted that the average entropy of the low complexity regions being formed is significantly less than the average entropy of the entire sequence, and this is expected.

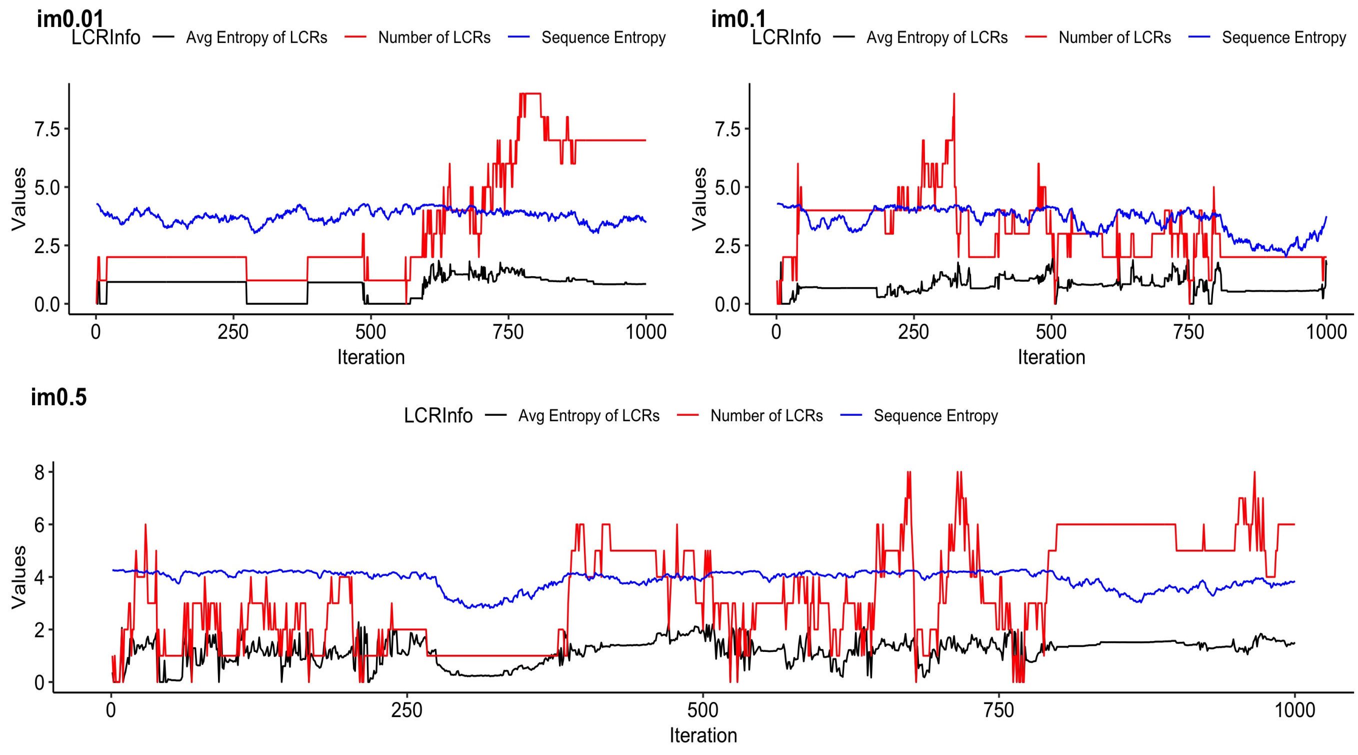

Figure 3: Entropy and number of LCR comparisons in a randomly generated protein sequence that has undergone simulated evolution. (TOP LEFT) A mutation rate of 0.01 was utilized. (TOP RIGHT) A mutation rate of 0.1 was utilized. (BOTTOM) A mutation rate of 0.5 was utilized. An indel rate of 1 was used for all.

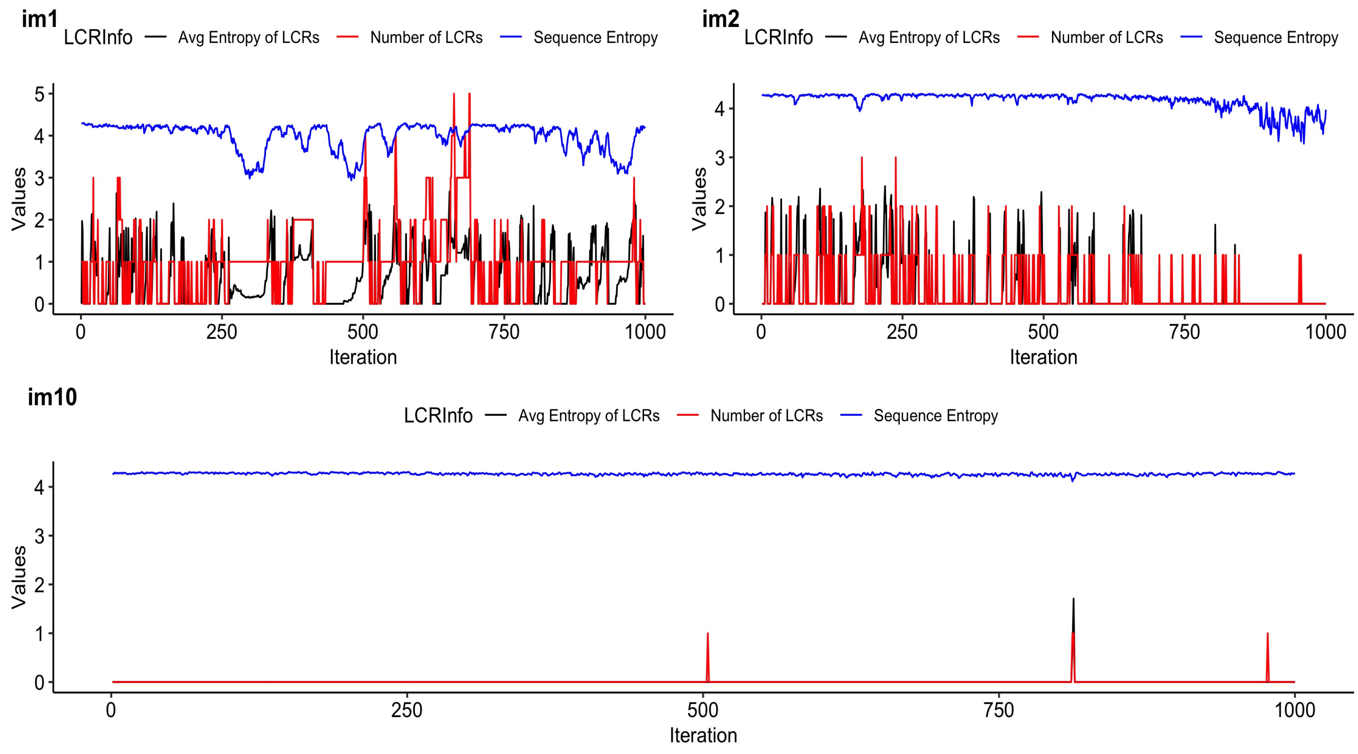

Figure 4: Entropy and number of LCR comparisons in a randomly generated protein sequence that has undergone simulated evolution. (TOP LEFT) A mutation rate of 1 was utilized. (TOP RIGHT) A mutation rate of 2 was utilized. (BOTTOM) A mutation rate of 0.5 was utilized. An indel rate of 1 was used for all.

Applying the Simulator to an ABC-MCMC

The second part of this study attempted to create an ABC-MCMC, using the LCR evolution simulator as a step in the algorithm. Time constraints in programming the ABC did not allow for the visualization of posterior distributions. We did however establish that the evolution simulator must be run for a minimum 1000 iterates for each newly proposed parameter value, in order to reach an almost "equilibrium" state. (Figure 5) shows the euclidean distance plotted over each iteration of the simulation. The euclidean distance was calculated using vectors of summary statistics for both the simulated protein sequence and the observed protein sequence (SRP40 *S. cerevisiae*). Essentially the larger the distance, the more differences there are between the simulated protein sequence and the SRP40 protein sequence. In the plot on the left, mutation rate is set to 0 and indel rate is set to 1. In this case, we see the distance between the mutated protein and SRP40 increasing quite rapidly as more LCRs are being formed, and then start to reach a more steady state around 0.5. In the plot on the left, mutation and indel rates are both set to 1. In this case, since mutations are involved as well now, we the distance rapidly increase and start to decrease more steadily at around 0.85.

For each newly proposed parameter value in the ABC-MCMC, we run through the evolution simulator 1000 times and take the average of all 1000 vectors produced. This is why we need to reach some sort of equilibrium in distance to ensure that we reach a level status of both parameters being proposed.

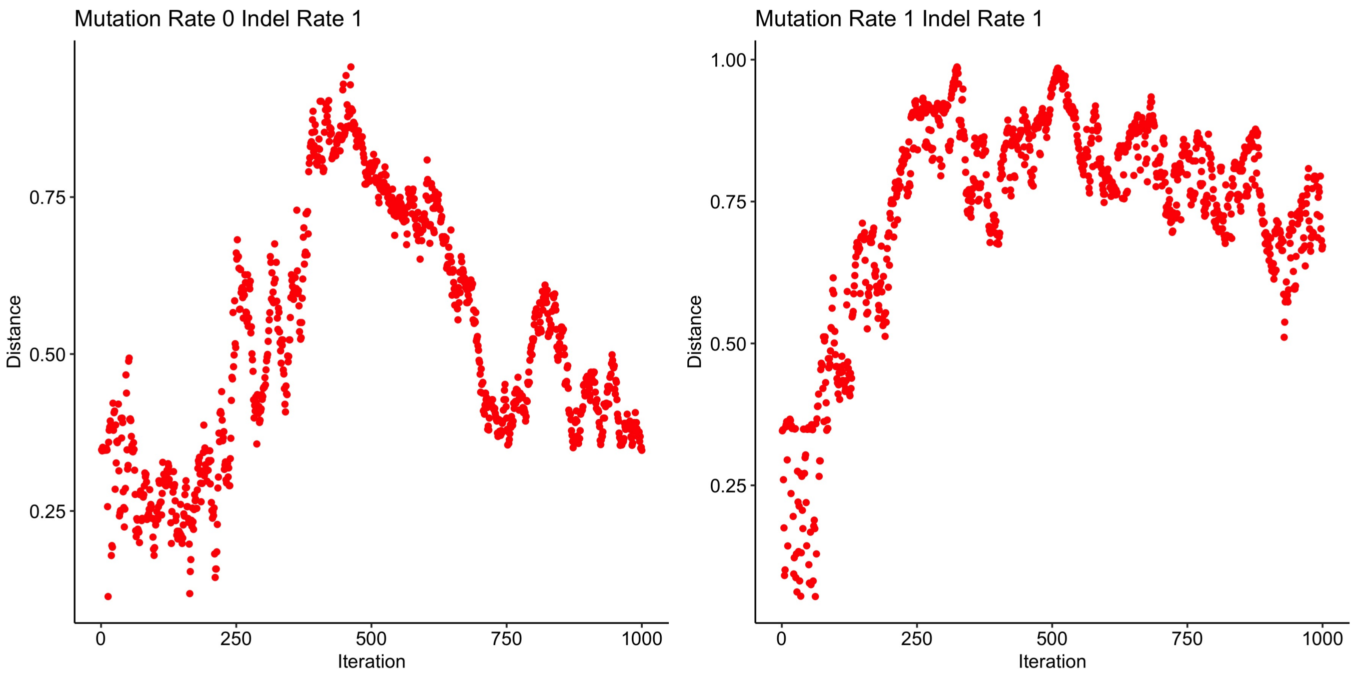

Figure 5: Calculated euclidean distances between mutated protein and SRP40 protein over 1000 evolution simulator iterates. (LEFT) mutation rate 0, indel rate 1. (RIGHT) mutation rate 1, indel rate 1.

Discussion

This study has developed a method of LCR simulation, which can be used in combination with an approximate bayesian computation markov chain monte carlo method. Due to timing, this study lacks estimates of parameter values that correspond to mutation and indel rates. However, this study does show the benefits of utilizing an approach that does not require calculation of the likelihood function. In the case of this study, mutations, insertions, and deletions alter the landscape of a protein sequence, making it extremely challenging to calculate the likelihood function. In the context of population genetics, ecology, epidemiology and systems biology, ABC approaches have been rapidly gaining popularity due to the massive increase in both the quantity and complexity of biological data (Sunnåker et al. 2013). With the advent of technology and data collection, we are at a point in time where new, tractable approaches must be developed and tested.

Mutations Have the Ability to Destroy LCR Formation

Results obtained from the evolution simulator are heavily dependent on the selected mutation and indel rates. Although both are forms of mutation, each rate parameter corresponds to a different process underneath the hood. Mutation rate in this case refers to the rate at which a base substitution occurs (changing one amino acid into another), while the indel rate refers to the rate at which an insertion or deletion takes place (adding or removing an amino acid residue). By altering mutation and indel rates, we are able to prove the simulation is functioning, and qualitatively assess the effects of having more point mutations on indel formation and vice versa.

With indel rates of 0.1 and 0.5 and a mutation rate of 1, very few low complexity regions are formed, and tend to be destroyed very easily. This is potentially due to the mutation rate being larger, as point mutations have the ability to destroy repetitive regions. As the indel rate is increased to 1,2, and 10, while still holding the mutation rate constant at 1, LCR's are formed in much higher quantities and remain in the sequence for much longer. The overall sequence entropy also decreases with these rates more than the previous rates, due to the increase of low complexity regions contributing to the decrease in sequence entropy. It is interesting to observe that an indel rate 10x larger than the mutation rate does not seem to produce more repetitive, low complexity regions, that swamp out the mutations being created. This could be due in part to the fact that since deletions are also simultaneously occuring, this process could affect how many LCRs are created.

This is not the case however when mutation rates are adjusted and indel rates are held constant at 1. With mutation rates of 0.01, 0.1, and 0.5, many low complexity regions are being formed and the overall sequence entropy tends to fluctuate rapidly. Since the indel rate is higher than the mutation rate in this case, LCRs will be formed and remain stable throughout the entire simulation. When mutation rates are increased to values larger than the indel rates, very few LCRs will be formed and the average sequence entropy will remain close to maximum at about 4.3. With a mutation rate 10x larger than the indel rate, we see the formation of only 3 LCRs which are destroyed in the next iteration, and the average sequence entropy does not fluctuate. This suggests that when the mutation rate is much larger than the indel rate, we are very rarely forming repeats, and when we do, they seem to be immediately destroyed by mutation.

Does the Length of a Repeat Play a Role in Insertions/Deletions

Insertions and deletions (indels) are a very important type of spontaneous mutation, as they make up the majority of the divergence among species (Britten 2002; Anzai et al. 2003). Indels however have been studied much less extensively compared to point mutations (Cartwright 2009). In this study, deviates were drawn from the exponential distribution and assigned to each individual amino acid residue. When deviates were assigned to residues that were part of a repetitive unit, the length of the repeat was multiplied by the indel rate, and this value was used as the scale parameter (β) of the exponential distribution. The reason for doing this is because it has been repeatedly observed in both DNA and proteins that the occurrence of indels declines monotonically as a function of their length (Pascarella and Argos 1992; Benner et al. 1993; Qian and Goldstein 2001; Loewenthal et al. 2021). As the length of a repeat increases, so does the scale parameter we use to draw deviates from the exponential distribution. With a larger scale parameter, the mean of the distribution also increases and there is a greater chance we draw deviates with larger values. This subsequently means that the longer the repeat is, the less likely it is to continue to gain repeats and hence we see this monotonic decrease in the occurrence of indels.

This monotonic decrease could also explain why an indel rate 10x larger than the mutation rate does not produce a protein chalked full of repetitive regions. As the indel rate is increased, the number of LCRs increases, but as these LCRs continue to increase in length, it becomes increasingly rare to continue to add repeats in that region. An ideal extension of this research would be to explore alternative relationships which describe the role of the length of a repetitve region in its subsequent mutation. This could be done through assigning deviates for repeats based on different mathematical functions. Currently deviates are assigned to repeated amino acids by multiplying the length of the repeat by the indel rate, but this could be potentially changed to multiplying the indel rate by an exponential function, e^{length}. In this case, as the length of the repetitive region increases, the exponential part of the function would also increase exponentially. This means that when we have longer repeats, it will be more difficult to keep adding repeats due to the extremely large scale parameter that would be used to draw deviates from the exponential distribution.

Why Parameters Could not be Estimated

As previously stated, the ABC-MCMC did not produce posterior distributions to estimate parameter values for mutation and indel rates. Overall, the process behind an ABC-MCMC is very complex, but in this study we simplified many of the steps in the algorithm, which could potentially be the reason posterior distributions were not produced.

An area of major concern in this study stemmed from the chosen summary statistics. Summary statistics are vital in order to quantitatively capture relevent information about the data under analysis (Sunnåker et al. 2013). One potential issue with the summary statistics is that they could not be explaining the characteristics we want in the best way. In this study, we utilized the average number of LCRs, the average entropy of those LCRs, and the length of the entire protein sequence as characteristics that explained the protein sequence. These summary statistics may be too general to accurately capture the differences between a protein of known low complexity (SRP40 *S. cerevisiae*), and a randomly simulated protein sequence. The goal of the ABC-MCMC was to essentially simulate a protein similar to a known protein sequence using various mutation and indel rates, in order to estimate the rates that produce the known sequence. The way this simulated sequence is compared to the known sequence is only through these summary statistics, not by the composition of the sequence itself. This means for example that a simulated protein sequence could have 3 LCRs (Same as in SRP40), but these LCRs could be in different locations and made up of different amino acids, which means the simulated protein is still very different from the known protein, but we are inccurately saying they are similar based on this summary statistic.

It could also be the case however that the summary statistics used in this study required specific weightings to minimize certain statistics from dominating the distance calculation. The length summary statistic tended to be a dominating force when running the ABC, which affected how often we accepted and rejected parameter values. Due to the fact that we attempted to estimate two parameters with the ABC, it is often the case that while a statistic may estimate one of these parameters well, it may not do the same for the other (Hamilton et al. 2005). A potential workaround for this is to implement a weighting scheme that essentially gives greater weights to statistics carrying more information on the parameter of interest (Hamilton et al. 2005). The way this could be done is to assess relationships between parameters and statistics using local regressions, and subsequently implement a weighted Euclidean distance using the results from the local regression. (Hamilton et al. 2005). Through multiplying summary statistics by a certain weight, it could potentially allow for the acceptance of more parameters due to statistics like length not being the dominating factor in the Euclidean distance calculation.

Excellent will have to go read Hamilton's paper

Conclusion

In summary, the creation of a program that simulates the evolution of low complexity regions can enable the exploration of new model based approaches such as approximate bayesian computations. This can subsequently allow for predictions to be made regarding the evolutionary history of protein or DNA sequences. This research also contributes to the lack of knowledge there is surrounding indels and the way in which they evolve. Through adjusting models of indel evolution using the simulator, the way in which indels are formed can be better understood. This research provides the ground work that is required for utilizing approximate bayesian computation approaches in the context of low complexity regions. The evolution of low complexity regions is still a growing area of research, as the complexity behind how they evolve makes it very difficult to analyze the way they evolve. This research can help pave the way for future work into using model based analyses for exploring the evolutionary dynamics of low complexity regions in proteins.

Please also provide the directory locations for the suppl files.

References

Anzai T, Shiina T, Kimura N, Yanagiya K, Kohara S, Shigenari A, Yamagata T, Kulski J K, Naruse T K, Fujimori Y, et al. (2003). Comparative sequencing of human and chimpanzee MHC class I regions unveils insertions/deletions as the major path to genomic divergence. Proceedings of the National Academy of Sciences 100(13), 7708–7713.

Bannen R M, Bingman C A, and Phillips G N (2007). Effect of low-complexity regions on protein structure determination. Journal of Structural and Functional Genomics 8(4), 217–226.

Battistuzzi F U, Schneider K A, Spencer M K, Fisher D, Chaudhry S, and Escalante A A (2016). Profiles of low complexity regions in Apicomplexa. BMC evolutionary biology 16(1), 1–12.

Bayliss C D, Field D, Moxon E R, et al. (2001). The simple sequence contingency loci of Haemophilus influenzae and Neisseria meningitidis. The Journal of clinical investigation 107(6), 657–666.

Benner S A, Cohen M A, and Gonnet G H (1993). Empirical and structural models for insertions and deletions in the divergent evolution of proteins. Journal of molecular biology 229(4), 1065–1082.

Brinkmann B, Klintschar M, Neuhuber F, Hühne J, and Rolf B (1998). Mutation rate in human microsatellites: influence of the structure and length of the tandem repeat. The American Journal of Human Genetics 62(6), 1408–1415.

Britten R J (2002). Divergence between samples of chimpanzee and human DNA sequences is 5%, counting indels. Proceedings of the National Academy of Sciences 99(21), 13633–13635.

Cartwright R A (2009). Problems and solutions for estimating indel rates and length distributions. Molecular biology and evolution 26(2), 473–480.

Coletta A, Pinney J W, Solıés D Y W, Marsh J, Pettifer S R, and Attwood T K (2010). Low-complexity regions within protein sequences have position-dependent roles. BMC systems biology 4(1), 1–13.

DePristo M A, Zilversmit M M, and Hartl D L (2006). On the abundance, amino acid composition, and evolutionary dynamics of low-complexity regions in proteins. Gene 378, 19–30.

Dunker A K, Brown C J, Lawson J D, Iakoucheva L M, and Obradović Z (2002). Intrinsic disorder and protein function. Biochemistry 41(21), 6573–6582.

Ellegren H (2004). Microsatellites: simple sequences with complex evolution. Nature reviews genetics 5(6), 435–445.

Everett C and Wood N (2004). Trinucleotide repeats and neurodegenerative disease. Brain 127(11), 2385–2405.

Gardner M J, Hall N, Fung E, White O, Berriman M, Hyman R W, Carlton J M, Pain A, Nelson K E, Bowman S, et al. (2002). Genome sequence of the human malaria parasite Plasmodium falciparum. Nature 419(6906), 498–511.

Haerty W and Golding G B (2010a). Genome-wide evidence for selection acting on single amino acid repeats. Genome research 20(6), 755–760.

Haerty W and Golding G B (2010b). Low-complexity sequences and single amino acid repeats: not just “junk” peptide sequences. Genome 53(10), 753–762.

Hamilton G, Currat M, Ray N, Heckel G, Beaumont M, and Excoffier L (2005). Bayesian estimation of recent migration rates after a spatial expansion. Genetics 170(1), 409–417.

Huntley M and Golding G B (2000). Evolution of simple sequence in proteins. Journal of molecular evolution 51(2), 131–140.

Huntley M A and Clark A G (2007). Evolutionary analysis of amino acid repeats across the genomes of 12 Drosophila species. Molecular biology and evolution 24(12), 2598–2609.

Huntley M A and Golding G B (2002). Simple sequences are rare in the Protein Data Bank. Proteins: Structure, Function, and Bioinformatics 48(1), 134–140.

Huntley M A and Golding G B (2006). Selection and slippage creating serine homopolymers. Molecular biology and evolution 23(11), 2017–2025.

Karlin S, Brocchieri L, Bergman A, Mrázek J, and Gentles A J (2002). Amino acid runs in eukaryotic proteomes and disease associations. Proceedings of the National Academy of Sciences 99(1), 333–338.

Kimura M (1983). *The neutral theory of molecular evolution*. Cambridge University Press.

Kruglyak S, Durrett R, Schug M D, and Aquadro C F (2000). Distribution and abundance of microsatellites in the yeast genome can be explained by a balance between slippage events and point mutations. Molecular Biology and Evolution 17(8), 1210–1219.

Kruglyak S, Durrett R T, Schug M D, and Aquadro C F (1998). Equilibrium distributions of microsatellite repeat length resulting from a balance between slippage events and point mutations. Proceedings of the National Academy of Sciences 95(18), 10774–10778.

Kumari B, Kumar R, and Kumar M (2015). Low complexity and disordered regions of proteins have different structural and amino acid preferences. Molecular BioSystems 11(2), 585–594.

Laurin-Lemay S, Dickson K, and Rodrigue N (2022). Jump-Chain Simulation of Markov Substitution Processes Over Phylogenies. Journal of Molecular Evolution, 1–5.

Levinson G and Gutman G A (1987). Slipped-strand mispairing: a major mechanism for DNA sequence evolution. Molecular biology and evolution 4(3), 203–221.

Liepe J, Barnes C, Cule E, Erguler K, Kirk P, Toni T, and Stumpf M P (2010). ABC-SysBio—approximate Bayesian computation in Python with GPU support. Bioinformatics 26(14), 1797–1799.

Loewenthal G, Rapoport D, Avram O, Moshe A, Wygoda E, Itzkovitch A, Israeli O, Azouri D, Cartwright R A, Mayrose I, et al. (2021). A probabilistic model for indel evolution: differentiating insertions from deletions. Molecular biology and evolution 38(12), 5769–5781.

Marcotte E M, Pellegrini M, Yeates T O, and Eisenberg D (1999). A census of protein repeats. Journal of molecular biology 293(1), 151–160.

Marjoram P (2013). Approximation bayesian computation. OA genetics 1(3), 853.

Marjoram P, Molitor J, Plagnol V, and Tavaré S (2003). Markov chain Monte Carlo without likelihoods. Proceedings of the National Academy of Sciences 100(26), 15324–15328.

Mier P, Paladin L, Tamana S, Petrosian S, Hajdu-Soltész B, Urbanek A, Gruca A, Plewczynski D, Grynberg M, Bernadó P, et al. (2020). Disentangling the complexity of low complexity proteins. Briefings in Bioinformatics 21(2), 458–472.

Mirkin S M (2007). Expandable DNA repeats and human disease. Nature 447(7147), 932–940.

Mularoni L, Veitia R A, and Albà M M (2007). Highly constrained proteins contain an unexpectedly large number of amino acid tandem repeats. Genomics 89(3), 316–325.

Nevo E (2001). Genetic diversity.

Newfeld S J, Smoller D A, and Yedvobnick B (1991). Interspecific comparison of the unusually repetitiveDrosophila locus-mastermind. Journal of molecular evolution 32(5), 415–420.

Newfeld S J, Tachida H, and Yedvobnick B (1994). Drive-selection equilibrium: homopolymer evolution in the Drosophila gene mastermind. Journal of molecular evolution 38(6), 637–641.

Ohta T and Kimura M (1973). A model of mutation appropriate to estimate the number of electrophoretically detectable alleles in a finite population. Genetics Research 22(2), 201–204.

Pascarella S and Argos P (1992). Analysis of insertions/deletions in protein structures. Journal of molecular biology 224(2), 461–471.

Qian B and Goldstein R A (2001). Distribution of indel lengths. Proteins: Structure, Function, and Bioinformatics 45(1), 102–104.

Ross C A, McInnis M G, Margolis R L, and Li S.-H (1993). Genes with triplet repeats: candidate mediators of neuropsychiatric disorders. Trends in neurosciences 16(7), 254–260.

Shannon C E (1948). A mathematical theory of communication. The Bell system technical journal 27(3), 379–423.

Sunnåker M, Busetto A G, Numminen E, Corander J, Foll M, and Dessimoz C (2013). Approximate bayesian computation. PLoS computational biology 9(1), e1002803.

Verstrepen K J, Jansen A, Lewitter F, and Fink G R (2005). Intragenic tandem repeats generate functional variability. Nature genetics 37(9), 986–990.

Warren S T, Muragaki Y, Mundlos S, Upton J, and Olsen B R (1997). Polyalanine expansion in synpolydactyly might result from unequal crossing-over of HOXD13. Science 275(5298), 408–409.

Wootton J C and Federhen S (1993). Statistics of local complexity in amino acid sequences and sequence databases. Computers & chemistry 17(2), 149–163.

Wright P E and Dyson H J (2015). Intrinsically disordered proteins in cellular signalling and regulation. Nature reviews Molecular cell biology 16(1), 18–29.

Wu S H and Rodrigo A G (2015). Estimation of evolutionary parameters using short, random and partial sequences from mixed samples of anonymous individuals. BMC bioinformatics 16(1), 1–12.

Appendix 1: Low Complexity Region Evolution Simulator

```cpp
#include "functions.cpp"
#include "getindex.cpp"
//#include "simulated_protein.cpp"
#include <bits/stdc++.h>
#include <iostream>
#include <vector>
#include <string>
#include <random>
#include <ctime>
#include <cmath>
#include <map>
#include <algorithm> //this is to get min element stuff, cool library
#define numAA 20
using namespace std;

Ran myran(time(NULL)); //We will use 21 as the random seed right now, used in Poissondev too

////////////////////////////////////////////////////////////////
// FIRST FUNCTION TAKES AN INTEGER VALUE AND GENERATES A RANDOM/
// AMINO ACID SEQUENCE OF THAT LENGTH                          /
////////////////////////////////////////////////////////////////
std::string createSeq(int n){

    char aminoAcids[numAA] = { 'G', 'A', 'L', 'M', 'F', 'W', 'K', 'Q', 'E', 'S', 'P', 'V',
                                   'I', 'C', 'Y', 'H', 'R', 'N', 'D', 'T' };

    std::string protein = "";
    for (int i = 0; i < n; i++){
        protein += aminoAcids[myran.int64() % numAA];} //this rand() % 20 means in the range 0-19

    //std::cout << protein << "\n" << "\n" ;
    return protein;
}

///////////////////////////////////////////////////
/////// FUNCTION TO GENERATE NORMAL DEVIATES //////
///////////////////////////////////////////////////
double getNormalDev(double mu, double stdev) {
    Normaldev mynorm(mu, stdev, myran.int64());
    double dev = mynorm.dev();
    //std::cout << dev << "\n";
    return dev;
}

/////Trying different function to generate random deviates
double getNormalDev2(double mu, double stdev) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::normal_distribution<double> dist(mu, stdev);

    double random_num = dist(gen);
    return random_num;
}

/////using uniform real distribution to generate numbers between
//0.0005 and 0.5 for the ttest pvalue

double random_num_zero_half() {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<double> dist(0.0,0.5);

    double ran_num = dist(gen);
    return ran_num;
}

/////FUNCTION TO CALUCLATE ENTROPY OF PROTEIN SEQUENCE
double calc_entropy(string protein_seq) {
    map<char, int> freq_map;
    for (char c : protein_seq) {
        freq_map[c]++;
    }
    double entropy = 0.0;
    int seq_len = protein_seq.length();
    for (auto const& pair : freq_map) {
        double prob = (double) pair.second / seq_len;
        entropy -= prob * log2(prob);
    }
    return entropy;
}

//
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```cpp
////////// Changed here down - Brian - Feb 12 Xander made Changes, trying to understand the new stuff ////////////
// ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
std::string mutateSeqExpBG(std::string simulated_protein, double mutation_rate, double indel_rate){ // took out iterates as a parameter

    //Setting up the vectors
    std::vector<double> mut_dev;
    std::vector<double> ind_dev;

    char aminoAcids[20] = { 'G', 'A', 'L', 'M', 'F', 'W', 'K', 'Q', 'E',
        'S', 'P', 'V', 'I', 'C', 'Y', 'H', 'R', 'N', 'D', 'T' };

    //std::cout << "before mutateseqEXP:\t" << simulated_protein << "\n"; // Initially printing the non-mutated strin.
    int len = simulated_protein.length();

    //ASSIGN DEVIATES FOR MUTATION
    for (int i = 0; i < len; i++) {
        float beta1 = mutation_rate ;
        Expondev myexp(beta1,myran.int64());
        double deviate = myexp.dev(); // choose exp_deviate(mean of beta)
        mut_dev.push_back(deviate);
    }
    // Traverse the string and generate deviates FOR INDELS - this seems to be
    // working, counter is showing correct values for repeats -Feb 15
    for (int i = 0; i < len; i++) {
        int counter = 1 ;
        //Code to scan back and forth to find repeats - this i-1 will give -1 on the first iteration though - check this in while loop too for scanning backwards
        if (simulated_protein[i] != simulated_protein[i+1] && simulated_protein[i] != simulated_protein[i-1]) {
            float beta2 = indel_rate ; // the length if no repeats is 1
            Expondev myexp(beta2,myran.int64());
            double deviate = myexp.dev();
            ind_dev.push_back(deviate);
        } else {
            int x = 1 ;
            int y = 1 ;
            //Be careful in these while loops, for i-y, when i is 0
            //and y is 1, how does it not throw error
            //Looking forward for repeats
            while (simulated_protein[i] == simulated_protein[i + x]) {
                counter += 1 ;
                x++;
            }
            //Looking backwards for repeats
            while (simulated_protein[i] == simulated_protein[i - y]) {
                counter += 1 ;
                y++;
            }
            float beta3 = indel_rate * counter ; // trying to see if the # of repeats plays a role, multiply by counter
            Expondev myexp(beta3,myran.int64());
            double deviate = myexp.dev();
            ind_dev.push_back(deviate);
        }
        //std::cout << counter << "\n"; //Checking if counter is finding repeats
    }

    // Working with the vector of structs, why we doing 0.5*len
    // FEB 15 - WORKING WITH VECTORS NOW
    for(int iter=0; iter<0.5*len; iter++) { // throw down 0.5 mut/site
        //selecting the lowest deviate from both vectors
        double minExpDev=mut_dev[0];
        int minPosition=0;
        int minType=0; // type=0 for mut, 1 for ins, 2 for del
        for (int i = 0; i < len; i++) {
            if(minExpDev > mut_dev[i]) {
                minExpDev=mut_dev[i];
                minPosition=i; minType=0;
            }
            if(minExpDev > ind_dev[i]) {
                minExpDev=ind_dev[i];
                minPosition=i; minType=1;
                if(myran.doub() < 0.5) { minType=2; }
            }
        }
        //std::cout << "minpos" << minPosition << "\n";
        //std::cout << "mindev" << minExpDev << "\n";
        //std::cout << "mintype" << minType << "\n";
        //std::cout << "iter" << iter << "\n";

        //This is where base changes, insertions, deletions occur
        if(minType==0) { // put in a base change
            simulated_protein[minPosition]=aminoAcids[myran.int64() % numAA];
            float beta1 = mutation_rate ;
            Expondev myexp(beta1,myran.int64());
            double deviate = myexp.dev();
            mut_dev[minPosition]=deviate;//swap deviate for base change
            //Printing to see mutated protein afer base change//
            //std::cout << "\n" << "after basechange:\t" << simulated_protein << "\n" << "\n" ;
            // THIS IS JUST TO PRINT THE VECTOR OF DEVIATES FOR MUTATION TO SEE IF IT CHANGED
            /*for (int x = 0; x < mut_dev.size(); x++) {
                std::cout << mut_dev[x] << ' ';
            }
            for (int x = 0; x < ind_dev.size(); x++) {
                std::cout << ind_dev[x] << ' ';
            }*/
        } else { // put in an indel
            if(minType==1) { // use insertion
                if(len < 1.5*len) {
                    len++;
                    simulated_protein.insert(minPosition+1, 1, simulated_protein[minPosition]); // insert
     repeat beside minPosition
                    float beta1 = mutation_rate ;
                    Expondev myexp(beta1,myran.int64());
                    double deviate = myexp.dev();
                    mut_dev.insert(mut_dev.begin() + minPosition+1, deviate); //Adding a new deviate to
    mutation vector
                    ind_dev.insert(ind_dev.begin() + minPosition+1, deviate); //Adding new deviate to
    indel vector as placeholder
                    //Printing to see mutated protein after insertion//
                    //std::cout << "\n" << "after insertion:\t" << simulated_protein << "\n" << "\n" ;
                    // THIS IS JUST TO PRINT THE VECTOR OF DEVIATES FOR MUTATION TO SEE IF IT CHANGED
                    /*for (int x = 0; x < mut_dev.size(); x++) {
                        std::cout << mut_dev[x] << ' ';
                    }
                    for (int x = 0; x < ind_dev.size(); x++) {
                        std::cout << ind_dev[x] << ' ';
                    }*/
                }
            } else { // use deletion
                if(len > 0.5*len) {
                    len--;
                    simulated_protein.erase(minPosition, 1); //delete amino acid at minPosition
                    mut_dev.erase(mut_dev.begin() + minPosition); //Deleting deviate in the mutation vector
                    ind_dev.erase(ind_dev.begin() + minPosition); //Deleting deviate in the indel vector
                    //Printing to see mutated protein after deletion//
                    //std::cout << "\n" << "after deletion:\t" << simulated_protein << "\n" << "\n" ;
                    // THIS IS JUST TO PRINT THE VECTOR OF DEVIATES FOR MUTATION TO SEE IF IT CHANGED
                    /*for (int x = 0; x < mut_dev.size(); x++) {
                        std::cout << mut_dev[x] << ' ';
                    }
                    for (int x = 0; x < ind_dev.size(); x++) {
                        std::cout << ind_dev[x] << ' ';
                    }*/
                }
            }
        }

        // THIS CHUNK OF CODE CAUSING MEMORY ISSUES, INDEXING SHIT - FEB 17
        // check what is around the change
        // need to look down one, the site and up one
        // So here were checking if the insertion,
        // or deletion affected the landscape of the DNA sequence.
        // Did it create a repeat, inturrupt a repeat, etc.

        //MUTATION AT START OF SEQUENCE
        if (minPosition == 0) {
            if (simulated_protein[minPosition] == simulated_protein[minPosition+1]) {
                int counter = 1;
                int x = 1;
                while (simulated_protein[minPosition] == simulated_protein[minPosition+x]) {
                    counter += 1;
                    x++;
                }
                int start = 0;
                int end = x-1;
                //std::cout << "start" << start << "\n";
                //std::cout << "end" << end << "\n";
                float beta3 = indel_rate * counter ;
                for(int j=start; j<=end; j++) {
                    Expondev myexp(beta3,myran.int64());
                    double deviate = myexp.dev();
                    ind_dev[j]=deviate;
                }
            } else {//NOT PART OF REPEAT
```

```
                float beta3 = indel_rate;
                Expondev myexp(beta3,myran.int64());
                double deviate = myexp.dev();
                ind_dev[minPosition]=deviate;
            }
        } else if (minPosition == simulated_protein.length()) {//Mutation at end of sequence
            if (simulated_protein[minPosition] == simulated_protein[minPosition-1]){
                int counter = 1;
                int y = 1;
                while (simulated_protein[minPosition] == simulated_protein[minPosition-y]) {
                    counter += 1;
                    y++;
                }
                int start = minPosition - y;
                int end = simulated_protein.length();
                //std::cout << "start" << start << "\n";
                //std::cout << "end" << end << "\n";
                float beta3 = indel_rate * counter;
                for(int j=start; j<=end; j++) {
                    Expondev myexp(beta3, myran.int64());
                    double deviate = myexp.dev();
                    ind_dev[j]=deviate;
                }
            }  else {//NOT PART OF REPEAT
                float beta3 = indel_rate;
                Expondev myexp(beta3, myran.int64());
                double deviate = myexp.dev();
                ind_dev[minPosition]=deviate;
            }
        } else {//mutation in middle of sequence
            if (simulated_protein[minPosition] == simulated_protein[minPosition+1] || simulated_protein[
    minPosition] == simulated_protein[minPosition-1]) { // part of a repeat
            int counter = 1;
            int x = 1;
            int y = 1;
            //Look backward
            while (simulated_protein[minPosition] == simulated_protein[minPosition-y]) {
                counter += 1;
                y++;
            }
            int start = minPosition - y; //Getting start position of repeats
            //std::cout << "start" << start+1 << "\n";
            //count forward
            while (simulated_protein[minPosition] == simulated_protein[minPosition + x]) {
                counter += 1;
                x++;
            }
            int end = minPosition + x; //Getting end position of repeats
            //std::cout << "end" << end-1 << "\n";
            float beta3 = indel_rate * counter ;
            for(int j=start+1; j<=end-1; j++) {
                Expondev myexp(beta3,myran.int64());
                double deviate = myexp.dev();
                ind_dev[j]=deviate;
            }
        }   else { // not part of a repeat
                float beta3 = indel_rate;
                Expondev myexp(beta3,myran.int64());
                double deviate = myexp.dev();
                ind_dev[minPosition]=deviate;
            }
        }
    }
    //std::cout << "\n" << "after mutateSeqEXP:\t" << simulated_protein << "\n" << "\n" ;
    return simulated_protein;
}

///// TRYING TO DEBUG THIS FILE WITH THIS MAIN FUNCTION /////
/*int main() {
    std::string simulated_protein = createSeq(10);
    for (int j = 0; j < 5; j++){
       // std::string simulated_protein = createSeq(10);
        double mutation_rate = 0.14;
        double indel_rate = 0.14;
        std::cout << calc_entropy(simulated_protein) << "\n";
        std::string mutated_protein = mutateSeqExpBG(simulated_protein, mutation_rate, indel_rate);
        simulated_protein = mutated_protein;
        sim_protein(simulated_protein);
    }
}*/
```

Appendix 2: ABC-MCMC in C++

```
#include "mutations_2_BG_vecs.cpp"
#include "observed_protein.cpp"
#include "simulated_protein.cpp"
#include "distance2.cpp"
#include "vecAvg.cpp"
#include "ttest.cpp"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <bits/stdc++.h>
#include <fstream> //The fstream library allows us to work with files
#include <vector> // Working with vectors is nicer than arrays, more c++

//////////////////////////////////////////////
// THE MAIN FUNCTION - WHERE EXECUTION BEGINS/
//////////////////////////////////////////////

int main() {

    // Setting initial parameters
    double mutation_rate = 10;
    double indel_rate = 1;
    int num_simulations = 1;
    int num_mutations = 1;
    double mean_proposal = 0.0;
    double stddev_proposal = 1.0;

    double mut_rate_arr[10000]; //for accepted mutation rates
    double ind_rate_arr[10000]; //for accepted indel rates
    double index[10000]; // simulation iterations
    double distance_array[10000]; //for accepted distances

    double current_mut_rate_arr[10000]; //for current mutation rates
    double proposed_mut_rate_arr[10000]; //for proposed mutation rates
    double current_ind_rate_arr[10000]; //for current indel rates
    double proposed_ind_rate_arr[10000]; //for proposed indel rates
    double current_distances_arr[10000]; //for current distances avg of 10
    double proposed_distances_arr[10000]; //for proposed distances avg of 10
    double distances_ttest[10000]; //for each of the 10 distances for proposed parameters

    double acc_rej_rate[10000]; //acceptance-rejection rate for mutation

    double mean_ttest_arr[10000]; //storing means from ttest of 10 vectors
    double sttdev_ttest_arr[10000]; //storing stdev from 10 vectors
    double tstat_array[10000]; //storing test statistics
    double probability_ttest[10000]; //storing pvalues for ttest
    double accepted_rejected[10000]; //1 for accepted, 0 for rejected
    double entropy_vec[10000];
    std::vector<vector<double>> vec_of_vecs3; //for getting vectors of 10 distances
    std::vector<vector<double>> vec_of_vecs4; //summary statistics for each of 10

    std::vector<vector<double>> vec_of_vecs_length;
    std::vector<vector<double>> vec_of_vecs_avgnum;
    std::vector<vector<double>> vec_of_vecs_entropy;
    std::vector<double> distance_current_vec;

    double new_mut_rate = 0;
    double new_ind_rate = 0;
    double acceptance_counter = 0;

    //importing the vector of summary statistic for observed protein SRP40 (Saccharomyces)
    //AND NORMALIZING IT, IT DOES NOT CHANGE
    std::vector<double> obs_prot_vtr = og_protein();
    obs_prot_vtr = normalize_vector(obs_prot_vtr);

    //Generating current state protein sequence outside loop for
    //first time
    std::string simulated_protein = createSeq(1000); //For the current state

    // GETTING THE FIRST CURRENT DISTANCE OUTSIDE THE MAIN LOOP
    // TRYING TO GET MUTATED SEQUENCE HERE SO WE CAN KEEP MUTATING
    // THE SAME SEQUENCE IN THE ALGORITHM

    for (int a = 0; a < 1000; a++) {
        for (int j = 0; j < num_mutations; j++){
            std::string mutated_protein = mutateSeqExpBG(simulated_protein, mutation_rate, indel_rate);
            simulated_protein = mutated_protein;
        }

        std::vector<double> sim_prot_vtr = sim_protein(simulated_protein);
        sim_prot_vtr = normalize_vector(sim_prot_vtr);
        double distance_current = vectors_distance2(sim_prot_vtr, obs_prot_vtr);
        distance_current_vec.push_back(distance_current);
    }

    double distance_current = Mean(distance_current_vec);

    std::cout << distance_current << '\n';

    //std::string simulated_protein2 = createSeq(400); //for proposed state - going to use the same
    sequence and keep mutating it for all 10 iterates below - keep mutatung same sequence the whole
    simulation

    //First for loop is for the number of simulations
    for (int i = 0; i < num_simulations; i++) {

        //setting this in the loop
        std::vector<vector<double>> vec_of_vecs; //for the current state
        std::vector<vector<double>> vec_of_vecs2; //for proposed state
        std::vector<double> distances; //holding vector of 10 distances per simulation iteration
        std::vector<double> length_vec; //holding length summary statistic
        std::vector<double> length_vec2; //holding unnormalized length
        std::vector<double> num_lcrs_vec; //holding number of LCRs summary statistic
        std::vector<double> avg_entropy_vec; //holding average entropy summary statistic

        current_mut_rate_arr[i] = mutation_rate; // adding current mut rate to array
        current_ind_rate_arr[i] = indel_rate; // adding current ind rate to array

        // 0. Proposing new parameter values - this if statement is
        // to hold one of the parameters constant on every other
        // iteration... in other words, holding one parameter
        // constant and changing the other on every other
        // iteration. - I changed this back to just generating new
        // ones each time.
        new_mut_rate = getNormalDev2(mean_proposal, stddev_proposal) + mutation_rate; //+ mutation_rate;
        std::cout << "Mutation rate: " << new_mut_rate << "\n";
        new_ind_rate = getNormalDev2(mean_proposal, stddev_proposal) + indel_rate; //+ indel_rate;i
        std::cout << "Indel Rate: " << new_ind_rate << "\n";

        if (i == 0) {
            new_mut_rate = mutation_rate;
            new_ind_rate = indel_rate;
        }

        proposed_mut_rate_arr[i] = new_mut_rate; // adding the proposed mut rate to array
        proposed_ind_rate_arr[i] = new_ind_rate; // adding the proposed ind rate to array

        std::string simulated_protein2 = createSeq(400); //for proposed state - going to use the same sequence and keep mutating it for all 10 iterates below

        // 1. We need to generate the random protein sequence
        // For loop here is to generate 10 vectors of summary
        // statistics for each parameter and get the average of all
        for (int k = 0; k<1000; k++){
            // 2. Next we need to mutate the simulated protein
            // Going to try and mutate over 2 gens
            // this for loop is just to mutate the protein
            // Only going thru 1 mutation process right now - Feb 17
            // Might need to stick with a lower number of mutations,
            // around 200, 1000 giving me errors and idk why...
            // Also mutating a protein under the newly proposed
            // parameter values with this loop
            for (int h = 0; h < num_mutations; h++) {
                std::string mutated_protein2 = mutateSeqExpBG(simulated_protein2, new_mut_rate, new_ind_rate);
                simulated_protein2 = mutated_protein2;
            }

            //Adding entropy of sequence to vector for plotting
            double entropy = calc_entropy(simulated_protein2);
            entropy_vec[k] = entropy;
            // 3. Next we will get the average of 10 vectors of summary statistics
            // For newly proposed parameter values

            std::vector<double> sim_prot_vtr_2 = sim_protein(simulated_protein2);
            vec_of_vecs2.push_back(sim_prot_vtr_2);

            // Getting the 10 distances in case we need t-test
            std::vector<double> vec_normal = normalize_vector(sim_prot_vtr_2);
            double length = sim_prot_vtr_2[0];
            double numlcr = sim_prot_vtr_2[1];
            double avgent = sim_prot_vtr_2[2];
            double length2 = sim_prot_vtr_2[0];
            length_vec2.push_back(length2);
            length_vec.push_back(length);
            num_lcrs_vec.push_back(numlcr);
            avg_entropy_vec.push_back(avgent);
            double dist = vectors_distance2(vec_normal, obs_prot_vtr);
            distances.push_back(dist); //storing the 10 distances to get mean and stdev
```

```
            distances_ttest[k] = dist; //storing each of the 10 distances for output

        }

        double expected_mutations = Mean(length_vec2)*mutation_rate; //getting the # of expected
    mutations
        std::cout << expected_mutations << "\n";
        vec_of_vecs3.push_back(distances); //pushing back all 10 distances
        vec_of_vecs_length.push_back(length_vec); //pushing back lengths
        vec_of_vecs_avgnum.push_back(num_lcrs_vec);
        vec_of_vecs_entropy.push_back(avg_entropy_vec);

        // TO TEST THE OUTPUT/PRINT THE VECTOR
        //for (int x = 0; x < vec_of_vecs.size(); x++) {
        //    for (int y = 0; y < vec_of_vecs[i].size(); y++){
        //        std::cout << vec_of_vecs[x][y] << " " << "\n";
        //    }
        //}

        // Getting the average vector of the 10 simulated vectors
        // for newly proposed parameters
        std::vector<double> sim_prot_vtravg2 = vectors_average(vec_of_vecs2);

        // ADDED STEP - LETS NORMALIZE THE VECTOR
        sim_prot_vtravg2 = normalize_vector(sim_prot_vtravg2);

        // 5. Lets try finding the distance between the vectors
        double distance_new = vectors_distance2(sim_prot_vtravg2, obs_prot_vtr);

        current_distances_arr[i] = distance_current; //storing current distances
        proposed_distances_arr[i] = distance_new; //storing proposed new distances

        // 6. Does the distance satisfy the conditions?
        // If it does, add new parameter to corresponding list
        // otherwise stay at same value
        // Updated the code now to calculate the current state of
        // the model and the proposed state of the model to compare
        // those distances.

        if (distance_new < distance_current && new_mut_rate > 0 && new_ind_rate > 0) {
            mutation_rate = new_mut_rate;
            indel_rate = new_ind_rate;
            mut_rate_arr[i] = mutation_rate;
            ind_rate_arr[i] = indel_rate;
            distance_array[i] = distance_new;
            index[i] = i;
            accepted_rejected[i] = 1;
            acceptance_counter = acceptance_counter + 1;
            acc_rej_rate[i] = acceptance_counter/i;
            distance_current = distance_new; //setting the new current distance to the new distance so we
     can compare a new distance to this accepted one
            std::cout << "ACCPETED" << "\n" << "\n";
        } else {//Do a ttest
            double standard_dev = stdDev(distances);
            double mean = Mean(distances);
            double tstat = abs((mean - distance_current) / (standard_dev / sqrt(distances.size())));
            double p_value = tTest(distances, distance_current);

            //Adding ttest values to output
            mean_ttest_arr[i] = mean; //storing means from ttest of 10 vectors
            sttdev_ttest_arr[i] = standard_dev; //storing stdev from 10 vectors
            tstat_array[i] = tstat; //storing test statistics
            probability_ttest[i] = p_value; //storing pvalues for ttest

            double random_number = myran.doub(); //Random number 0-1
            if (random_number < p_value && new_mut_rate > 0 && new_ind_rate > 0) {
                std::cout << "ACCEPTED pval" << "\n" << "\n";
                mutation_rate = new_mut_rate;
                indel_rate = new_ind_rate;
                mut_rate_arr[i] = mutation_rate;
                ind_rate_arr[i] = indel_rate;
                distance_array[i] = distance_new;
                index[i] = i;
                accepted_rejected[i] = 1;
                acceptance_counter = acceptance_counter + 1;
                acc_rej_rate[i] = acceptance_counter/i;
                distance_current = distance_new;
            } else {
                std::cout << p_value << "\n" << random_number << "\n";
                std::cout << "NOT ACCEPTED pval" << "\n" << "\n";
                std::cout << i << "\n";
                mut_rate_arr[i] = mutation_rate;
                ind_rate_arr[i] = indel_rate;
                distance_array[i] = distance_current;
                index[i] = i;
                accepted_rejected[i] = 0;
                acc_rej_rate[i] = acceptance_counter/i;
                continue;
            }
        }
  }

    std::ofstream myfile("accepted_parameters.txt"); //Create and open txt file
    // Printing the arrays of parameter values
    for (int b = 0; b<num_simulations; b++) {
        myfile << index[b] << '\t' << mut_rate_arr[b] << '\t' << ind_rate_arr[b] << '\t' <<
    distance_array[b] << '\t' << '\t' <<  current_mut_rate_arr[b] << '\t' << current_ind_rate_arr[b] << '
    \t' <<
            proposed_mut_rate_arr[b] << '\t' << proposed_ind_rate_arr[b] << '\t' << '\t' <<
    mean_ttest_arr[b] << '\t' <<  sttdev_ttest_arr[b] << '\t' <<  tstat_array[b] << '\t' <<
    probability_ttest[b] << '\t'
            << accepted_rejected[b] << '\t' << '\t' << acc_rej_rate[b] << '\n';
    }

    std::ofstream myfile2("10_distances.txt");
    for (int x = 0; x<vec_of_vecs3.size(); x++) {
        for (int y = 0; y<vec_of_vecs3[x].size(); y++) {
            myfile2 << index[x] << '\t' << '\t' << proposed_mut_rate_arr[x] << '\t' <<
    proposed_ind_rate_arr[x] << '\t' << vec_of_vecs3[x][y] << '\t' << vec_of_vecs_length[x][y]
                << '\t' << vec_of_vecs_avgnum[x][y] << '\t' << vec_of_vecs_entropy[x][y] << '\t' <<
    entropy_vec[y] << '\n';
        }
    }
}
```

Appendix 3: Simulated Protein Summary Statistics

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>

//////////////////////////////////////////////////////////////////////
// SECOND FUNCTION READS IN SRP40 PROTEIN AND CREATES A VECTOR WITH/
// CHARACTERISTICS - THIS IS EITHER S or D - THE ORIGINAL DATA     /
//////////////////////////////////////////////////////////////////////

std::vector<double> sim_protein(std::string sim_prot){

    std::vector<double> sim_ss_vec; //Making the simulated summary statistic vector

    // Finding the LENGTH of protein sequence
    int length = sim_prot.length();
    sim_ss_vec.push_back(length);

    // Creating fasta file of simulated protein string to use in segA
    std::ofstream MyFile("simulated_protein.fasta"); //Create and open a fasta file
    MyFile << ">simulation\n" << sim_prot; //Write to the file
    MyFile.close(); //Close the file

    // Finding the number of LCRs
    system("segA simulated_protein.fasta 15 1.9 2.2 -l | grep '>' | wc -l >> numlcr_sim.txt"); //Write
    the results of counting the LCRs to a txt file
    std::ifstream myreadfile3("numlcr_sim.txt"); //Declaring fstream variable and connecting it to a
    stream object by opening the file
    double numlcr_sim; //Declare int variable which holds the contents of the 'numlcr.txt' file
    myreadfile3 >> numlcr_sim; //Pipe file's content into stream
    sim_ss_vec.push_back(numlcr_sim); // using push_back to append to vector

    //This if statement is to remove the division by zero error in
    //the awk command below
    if (numlcr_sim == 0) {
        sim_ss_vec.push_back(0);
        //Removing files so they are empty after
        system("rm numlcr_sim.txt");
        system("rm simulated_protein.fasta");
    } else {

```

```
    // Finding the average entropy of the LCRs dont know why I have
    // to use egrep for the extraction of numbers part in the system
    // call
    system("segA simulated_protein.fasta 15 1.9 2.2 -l | grep 'complexity' | awk '{print $2}' | egrep -o
    '([0.0-9.0]+)' | awk '{sum += $1} END {print mean=sum/NR}' >> avg_entropy_lcrs_sim.txt");
    std::ifstream myreadfile4("avg_entropy_lcrs_sim.txt");
    double avgentropylcrs_sim;
    myreadfile4 >> avgentropylcrs_sim;
    sim_ss_vec.push_back(avgentropylcrs_sim);
    //Removing files so they are empty after
    system("rm avg_entropy_lcrs_sim.txt");
    system("rm numlcr_sim.txt");
    system("rm simulated_protein.fasta");
    }

    // TO TEST THE OUTPUT/PRINT THE VECTOR
    /*for (int x = 0; x < sim_ss_vec.size(); x++) {
        std::cout << sim_ss_vec[x] << ' ' << "\n";
    }*/

    return sim_ss_vec;
}

//int main() {

    //sim_protein("
    MASKKIKVDEVPKLSVKEKEIEEKSSSSSSSSSSSSSSSSSSSSSSSSSGESSSSSSSSSSSSSSDSSDSSDSESSSSSSSSSSSSSSSSDSESSSESDSSSSGSSSSSSSS
    ");
//    sim_protein(createSeq(400));
//    return 0;
//}
```

Appendix 4: Euclidean Distance and Normlization of Vectors

```cpp
#include <iostream>
#include <string>
#include <iterator>
#include <algorithm>
#include <vector>
#include <functional>
#include <cmath>
#include <numeric>

//this function I made to normalize vectors
//FEB 23 - THIS IS NOT HOW IM GOING TO NORMALIZE
/*std::vector<double> normalize_vector(std::vector<double> vec1) {
    std::vector<double> normalized_vec;
    double normalize_value = 0.00;
    int vecSize = vec1.size();
    for (int i = 0; i < vecSize; i++) {
        normalize_value += vec1[i] * vec1[i];
    }
    for (int j = 0; j < vecSize; j++) {
        normalized_vec.push_back(vec1[j]/sqrt(normalize_value));
    }

    // TO TEST THE OUTPUT/PRINT THE VECTOR
    for (int x = 0; x < normalized_vec.size(); x++) {
        std::cout << normalized_vec[x] << ' ' << "\n";
    }


    return normalized_vec;
}*/

//Feb 23 - New function for normalizing
std::vector<double> normalize_vector(std::vector<double> vec1) {
    std::vector<double> normalized_vec;
    int srp40_length = 406;
    int vecSize = vec1.size();
    float length = vec1[0];
    float num_lcrs = vec1[1];
    double avg_entropy = vec1[2];

    //setting the max and mins
    float max_length = srp40_length*1.5;
    float min_length = srp40_length*0.5;

    double max_lcrs = srp40_length/8; //8 is the hypothetical length of the shortest LCR, this will give
    you a maximal total number of LCRs
    double max_entropy = 4.3;

    //pushing back the normalized values
    normalized_vec.push_back((length - srp40_length + min_length)/srp40_length);
    normalized_vec.push_back(num_lcrs/max_lcrs);
    normalized_vec.push_back(avg_entropy/max_entropy);

    // TO TEST THE OUTPUT/PRINT THE VECTOR
    /*for (int x = 0; x < normalized_vec.size(); x++) {
        std::cout << normalized_vec[x] << ' ' << "\n";
    }*/

    return normalized_vec;

}

//this is my function I made to solve euclidean distance
double vectors_distance2(std::vector<double> a, std::vector<double> b) {
    int vecSize = a.size();
    double dist = 0.0;
    for (int i = 0; i < vecSize; i++){
        dist += pow((a[i] - b[i]), 2);
    }

    //std::cout << sqrt(dist) << "\n";
    return sqrt(dist);
}

//FOR TESTING THIS FILE
/*int main() {
    std::vector<double> v1 = {500,6,2.678};
    normalize_vector(v1);
}*/
```

Appendix 5: One Sample t-test

```cpp
#include <iostream>
#include <bits/stdc++.h>
#include <vector>
#include <iomanip>
using namespace std;

//Function to find Mean
double Mean(std::vector<double> vec) {
    double sum = 0;
    for (int i=0; i < vec.size(); i++) {
        sum = sum + vec[i];
    }

    return sum/vec.size();
}

//Function to caluclate Standard deviation of given vector
double stdDev(std::vector<double> vec) {
    double sum = 0;
    for (int i = 0; i < vec.size(); i++) {
        sum = sum + (vec[i] - Mean(vec)) * (vec[i] - Mean(vec));
    }

    return sqrt(sum / (vec.size() - 1));
}

//Function to get the equation of a line and estimate the
//probability based on the t-statistic
double eqn_line(double y2,double y1,double x2,double x1,double tstat) {
    double m = (y2 - y1) / (x2 - x1);
    double b = y2 - (m*x2);
    double x = (tstat - b) / m;

    return x;
}

//ReWriting function to obtain estimate of pvalue from linear
//interpolation
double tTest(std::vector<double> vec1, double prev_dist) {
    double mean1 = Mean(vec1);
    double sd1 = stdDev(vec1);

    //Equation to get t-statistic for 1 sample
    double t_statistic = abs((mean1 - prev_dist) / (sd1 / sqrt(vec1.size())));

    std::vector<double> areas = {0.5, 0.25, 0.2, 0.15, 0.10, 0.05, 0.025, 0.01, 0.005, 0.001, 0.0005}; //
    this is like y
    //std::vector<double> crit_t_values = {0.000, 0.703, 0.883, 1.100, 1.383, 1.833, 2.262, 2.821, 3.250,
     4.297, 4.781}; //critical t values 9 degrees of freedom - this is like x
    //std::vector<double> crit_t_values = {0.000, 0.677, 0.845, 1.042, 1.290, 1.660, 1.984, 2.364, 2.626,
     3.174, 3.390}; //critical t values 99 degrees of freedom
    std::vector<double>crit_t_values = {0.000, 0.675, 0.842, 1.037, 1.282, 1.646, 1.962, 2.330, 2.581,
    3.098, 3.300}; //critial t values 1000 degrees of freedom

    int minPosition=0;
    for (int i = 0; i < areas.size(); i++) {
        if (t_statistic > crit_t_values[i]) {
            minPosition = i;
        }
    }
    int maxPosition = minPosition + 1;

    double pval_est = eqn_line(crit_t_values[minPosition], crit_t_values[maxPosition], areas[minPosition
    ], areas[maxPosition], t_statistic);

    return pval_est;
    return minPosition;
    return abs(t_statistic);
}


//Function to find p value from table above
//Since im doing average of 10, the degrees of freedom would be 10-1
//which is 9 - Using 9 degrees of freedom all the time so crit
//values will be the same all the time
//Larger P value means not statically significant.

//To test the functions
/*int main() {
    std::vector<double> testvec1 = {10,20,30,40,50,60,70,80,90,100};
    double old_dist = 27.00;

    std::cout << tTest(testvec1, old_dist);

    return 0;
}*/
```