Polymer Simulations and DNA Topology

A Thesis presented
by

Ashok Cutkosky

to
The Department of Mathematics
in partial fulfillment of the honors requirements
for a Bachelors degree

Advisor: Erez Lieberman Aiden

Harvard University
Cambridge, Massachusetts
March 2013

(573)-529-6160
acutkosky@college.harvard.edu

Abstract

In 2009 the Hi-C procedure provided a new way to query the structure of the genome through genome-wide contact maps. A contact map indicates the probability that any two locations in the genome are in physical contact in the nucleus. This information was used to determine that chromatin, which is the composite polymer of DNA and protein that resides in the nucleus, has a fractal globule structure. The fractal globule is an unknotted yet densely packed polymer conformation model that is locally expandable. Hi-C data also provides information about higher-level structures present in the genome, such as the presence of open and closed compartments, which are regions of lower and higher density. All of this data represents an ensemble average of genomes. We develop and utilize computational tools to simulate chromatin dynamics and provide methods to reconstruct polymer conformations that reproduce these observed higher-level structures. This provides a way to use average Hi-C data to produce candidate individual genome conformations. Several video clips of data obtained from this simulation procedure can be found online at http://bit.ly/107Qlo4. We also give some detail on the algorithmic theory of molecular dynamics and polymer simulations as well as a discussion of a practical implementation of a polymer simulator on graphics processing units.

Next we take up the problem of double strand breaks and mutation. Double strand breaks are a form of DNA damage in which the DNA polymer is broken at one point into two pieces, and are a type of damage known to be highly carcinogenic. One common explanation for this is that double strand breaks cause chromosomal rearrangements or translocations, both of which modify the sequence of nucleotides. We suggest the possibility of a new mechanism for double strand break toxicity, the *topological mutation*. Instead of altering the DNA sequence, a double strand break allows topology violating strand passes to occur. These can destroy the special dynamical properties of the fractal globule, such as topological unknottedness and the consequent local expandability. This is a property of fractal globules in which any small subsection of the polymer can be unfolded from the dense globule packing without disturbing the rest of the structure. This phenomenon could have a role in facilitating access to specific parts of the genome, for example for gene expression. We demonstrate that after simulated double strand breaks, chromatin loses this local expandability property, thus providing evidence for the existence of topological mutations.

Contents

1 Introduction

1.1 Overview of Hi-C

The Hi-C protocol [5] is a technique that provides information about the frequency of contacts between different locations of the genome. The data from Hi-C can be used to provide deep insight into the structure of chromatin, which is the amalgamation of DNA and associated proteins in the nucleus. Before describing some of the properties of chromatin that can be inferred by Hi-C we will give a brief description of the procedure. As the first step, the genome is treated with formaldehyde in order to cause cross-linking - that is, chemical bonds form between locations of the genome that are physically adjacent at the time of treatment. This effectively "locks" the genome in its current conformation. Then the genome is treated with a *restriction enzyme*. This is a type of protein that breaks DNA at any point in which the nucleotide sequence matches a specific pattern. These points are called *restriction sites*, and the pattern of nucleotides that mark a restriction site varies from one restriction enzyme to another. The spacing between restriction sites is typically on the order of a kilobase (1,000 base pairs) so that the result of this procedure is a large number of kilobase-long fragments of DNA. An important point here is that the restriction enzyme cuts the DNA up in this very regular manner. Since we know the original sequence of the genome, we can see where all the restriction sites will be and so we know the nucleotide sequences for all the fragments. Crucially, the restriction enzyme does not affect the cross-linking created with formaldehyde. Thus in the Hi-C protocol the result of applying the restriction enzyme is a large number of pairs of fragments of DNA, where each linked pair of fragments was originally in physical contact in the genome.

The remainder of the protocol is a procedure for isolating these pairs and identifying their original locations in the genome. First, a few new nucleotides are added to the loose ends of the pairs, and one of the new nucleotides is marked with the chemical biotin. The current situation after this step is a set of pairs of fragments whose ends are marked with biotin. Then these fragments are ligated, which means they are treated with an enzyme that is effectively the inverse of a restriction enzyme, fusing broken ends of DNA instead of creating them. The ligation procedure causes the pairs of fragments to become loops. These loops are then *sheared*, which means effectively broken into small pieces so that now we have several small fragments of DNA, a few of which are marked with biotin. The fragments with biotin are isolated and sequenced. The sequence of a fragment marked with biotin (one that came from a loop), should contain part of the sequence of one fragment, followed by part of the sequence of a different fragment. By searching for these sequences in the known sequence of nucleotides for the entire genome, it is possible to uniquely identify where in the genome the two fragments came from.

When this procedure is repeated over a large set of genomes, one can obtain a probability that two fragments are in contact for all pairs of fragments. We can assign each fragment a number label based upon its position in the genome, and so we have a matrix (p_{ij}) where p_{ij} is the probability that fragment i is in contact with fragment j. This matrix is called a *contact map*. A nice way to visualize the contact map is as a heat map, where darker colors correspond to higher probabilities:

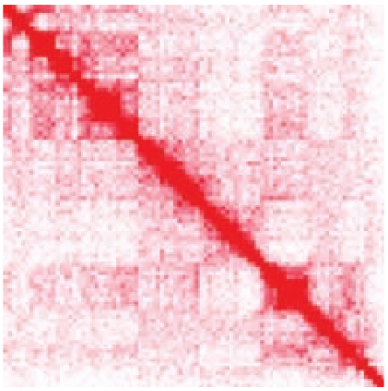


Figure 1: A Hi-C contact map, representing the probability that a pair of locations in the genome are in contact. Darker colors correspond to higher probabilities.

An important point to note about contact maps is that one expects the value of p_{ij} to increase as i approaches j since locations that are closer together along the linear path of the DNA polymer are expected to be closer together and so have a higher probability of contact than locations that are farther apart. This suggests one important quantity that can be computed from a contact map, namely the *contact probability* of chromatin. The contact probability $\chi(n)$ of a polymer is simply the probability that two points separated by a distance of n along the polymer contour are in contact. In the case of chromatin, a convenient unit of distance along the polymer is the base-pair, and that is the unit we will use in discussing the contact probability of chromatin. One expects $\chi(0) = 1$, and $\lim_{n\to\infty} \chi(n) = 0$. χ is a characteristic property of a polymer, and so can be used to classify different conformational models. In particular, two models of chromatin structure are the random-walk or *equilibrium globule* structure and the *fractal globule* structure. The equilibrium globule is simply the idea that the chromatin polymer describes a random walk in space. We can compute the contact probability of a random walk, and see that it follows the rule $\chi(n) \propto n^{-3/2}$. However, Hi-C data showed that for n in the 500 kilobase to 7 megabase range, the measured value of $\chi(n)$ was proportional to n^{-1}. This is not consistent with the equilibrium globule structure, but is consistent with the fractal globule structure.

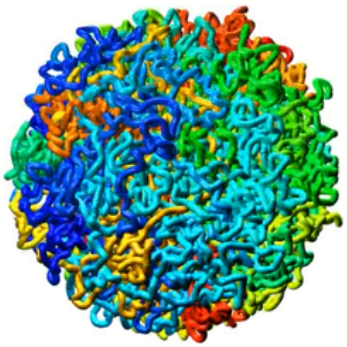
1.2 Description of the fractal globule

A fractal globule is a polymer conformation model that is unknotted and densely packed. In an equilibrium globule, a segment of length s is expected to take up a volume proportional to $s^{1/2}$ because its local behavior is equivalent to that of a random walk, while in a fractal globule a segment of length s is expected to take up volume proportional to $s^{1/3}$ [1]. They obtain this scaling through a hierarchical packing structure. The intuition behind fractal globule formation is that a long polymer chain collapses *locally*. Suppose the monomers in our polymer are given by $A[0], \cdots, A[n-1]$. If the polymer is placed into a bad solvent that causes it to collapse into fractal globule conformation, then we can observe this process happening on individual segments of the polymer. For example, $A[0-10]$ may quickly collapse into a dense conformation without interacting with the rest of the polymer. After a brief time the polymer should look like a kind of meta-polymer whose monomers consist of small compacted regions of the original polymer. This meta polymer will collapse in the same way, leading to a hierarchical (read: self-similar) structure in the final conformation that gives the fractal globule its density as well as its name.

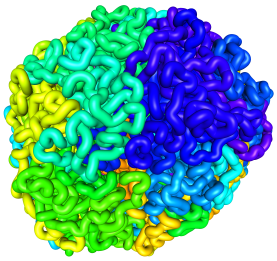
Throughout this thesis we will make use of a false-color depiction of a polymer contour in which color indicates position along the polymer:

Figure 2: We use color to indicate position along the polymer contour. Intuitively, if the polymer were stretched straight out, it would look like this.

The use of this convention is seen in that it makes equilibrium globules and fractal globules immediately distinguishable:



(a) An equilibrium globule. Note the heterogeneity of colors, indicating a well-mixed polymer.



(b) A fractal globule. Note the distinctive chromatic blocks, a consequence of the hierarchical structure of the fractal globule.

Figure 3: Equilibrium vs. Fractal globules

The Hi-C protocol provides interesting information about the topology and geometry of the genome. In order to investigate some further consequences of these data we can make use of computer simulations, specifically simulations of DNA-like polymers. Polymer simulations are a sub-field of molecular dynamics, which is the process of simulating general molecular systems. Molecular dynamics is a very well-studied area and there are a number of high-quality (and often open-source) simulators available - such as Gromacs, LAMMPS, AMBER and CHARMM [2][13]. Molecular dynamics is an important tool because it provides (modulo correctness of the physical models involved) a very precise depiction of the state of a system - which cannot be obtained via normal experimental methods. In this case while Hi-C provides a very large amount of specific data about the conformation of the genome, by supplementing this data with simulations we can attempt to acquire an even more in-depth picture of its dynamics and geometry.

1.3 We can use simulations to address problems related to chromatin structure

One issue in the Hi-C data is that the contact map represents an average of a large ensemble of genomes. A natural problem then is to computationally sample from this ensemble. That is, one would like to be able to generate a random contour in such a way that generating a contact map from a large set of these random polymers will result in the same contact map as observed in the Hi-C protocol. In essence, Hi-C provides some high-level information about chromatin structure, and we would like to use this information to infer a specific value for that structure. Molecular dynamics offers a natural way to approach this task, by simulating polymers using Hi-C data to inform the force calculations in the simulation.

These simulations can also be used to investigate the topological aspects of a crumpled polymer like chromatin, and how these aspects change under various types of perturbations. One such per-

turbation is the *double strand break*. This is a phenomenon in which the DNA polymer completely breaks into two pieces in some area. Normally DNA repair machinery inside the cell finds and restores these breaks. It is known that double strand breaks are a significant cause of cancer [10]. One commonly believed mechanism for this association is that DSBs can introduce alterations to the sequence of nucleotides, through chromosomal rearrangements or translocations. However, even in the case that the sequence of DNA base-pairs survives unscathed, it is possible that the geometry and topology of the chromatin is altered by this process in a way that is not correctable by the repair machinery. For example, in addition to being unknotted, fractal globules (and hence chromatin polymers) are also locally expandable. Intuitively, this means that any small segment can be pulled free of the globule without significantly disturbing the rest of the polymer.

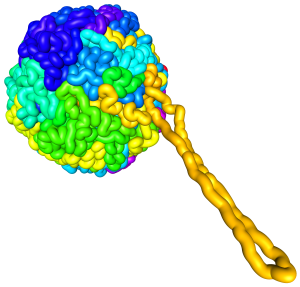


Figure 4: Fractal globules are locally expandable: we can pull out a section of the structure without disturbing the rest of the globule.

This property could have important ramifications for cellular processes such as gene expression in which transcription machinery proteins need to access a specific segment of the genome that holds a gene. Since the fractal globule is normally very densely packed, any given region of the genome may be inaccessible when it is needed. Therefore the cell would have to expand the structure in order to gain access to the desired region. Clearly, the property of local expandability makes this process significantly easier, since this allows the cell to gain access to (locally expand) a particular region while keeping the rest of the chromatin tightly packed. The geometry and topology of chromatin inform its dynamics, and the dynamical properties (such as local expandability) of chromatin could have a significant impact on its ability to carry out its various functions - e.g. coding for proteins, microRNAs. During a double strand break the topology of the chromatin may not be preserved, and so dynamical properties such as this local expandability might be lost. This suggests that double-strand breaks might be a source of a new form of mutation - what might be called *topological mutation*.

It turns out that many of the readily available simulation packages were designed for simulating biological molecules. Since we are interested in simulating chromatin, which is certainly a polymer of biological molecules, this seems ideal at first. However, a full-scale model of chromatin is very complex, involving a very high density of many types of molecules. This level of complexity means that it is not feasible to obtain simulation data spanning any reasonable length of time. Instead, what is wanted is a simplified generic polymer model that is computationally tractable. To this end, we constructed a custom-built molecular dynamics system designed specifically to simulate such generic polymers over relatively long time intervals. Our simulator operates on fairly naive algorithmic principles (in contrast to some more sophisticated molecular dynamics algorithms that we will also discuss), but achieves very fast runtimes through the use of graphics processing units. The graphics processing unit (GPU) is a piece of hardware that is now almost ubiquitous in the computer market. It is optimized for pipelined vector processing used to drive videos. What this means for scientific computation is that the GPU is an extremely parallel environment, and we will leverage this parallelism in our simulator.

We used this simulator to reproduce the chromatin ensemble. This was accomplished by using Hi-C contact map data to inform the forces used in the simulation. In particular, there are two salient features of the the contact map that we attempt to replicate - open and closed compartments and high-probability contacts. In the cell, chromatin is segregated into the open compartment - which are regions of less-densely packed, more expressible genes, and the closed compartment, which are more tightly packed, less accessible regions. Using the contact map, one can classify different regions of the genome as belonging to one compartment or the other. This is achieved essentially by noting that that in the contact maps from Hi-C, open regions are more likely to interact with open regions and closed regions are more likely to interact with closed regions. Then by using principle component analysis one can separate the two compartments. Another, perhaps more intuitive way of thinking of this, is to view the contact map as an adjacency matrix for a graph whose vertices are locations in the genome and the edge weight between two vertices as the probability of contact between the corresponding locations. Then one can apply some graph clustering techniques to separate the chromatin into two clusters, which correspond to the compartments. High-probability contacts are simply specific entries in the contact map that are significantly higher than would otherwise be expected from effects like distance along the chromatin polymer, or compartment identity.

Finally, we used the simulator to investigate the effects of double-strand breaks on the dynamics of chromatin. We will describe a procedure to simulate a double-strand break and repair in our polymers, and how this algorithm was used to simulate double-strand breaks in a number of simulated genomes. After simulating the double-strand breaks, we will then look at the property of local expandability. We can measure local expandability using our simulator. This process involves using the simulator to pull on a small segment of the polymer, and then measuring how far it travels away from the rest of the globule in a short time span. Our simulations show that double-strand breaks can negatively affect this property of a polymer, and so suggest that topological mutations occur in the genome.

2 Molecular Dynamics

Molecular dynamics is the process of simulating molecules via numerical integration of Newton's second law. This is a fundamental tool in modern biophysics because it allows one to acquire a very detailed picture of the dynamics of systems that are too small or chaotic to probe through direct experiment. In its most basic form, the computer is given a function that computes the forces on every component of a system in any given configuration. The algorithm will then update the position of each component based upon these forces, and repeat the process. That is, a bare-bones molecular dynamics simulator will call this function MD-UPDATE t/Δ times, where t is the total time to be simulated. Here U is some data structure encoding the system to be simulated.

function MD-UPDATE
  for each component M in the system U **do**
    $F \leftarrow$ Force on M due to the components of U.
    M.newvelocity = M.oldvelocity+$F \cdot \Delta$.
    M.newposition = M.oldposition+$\Delta \cdot M$.newvelocity
  end for
  for each component M in the system **do**
    M.oldvelocity = M.newvelocity.
    M.oldposition = M.newposition.
  end for
end function

This is in contrast to the use of Monte Carlo simulation methods. Essentially, the Monte-Carl method simulates molecules by choosing a cost (potential) function on the space of possible configurations of the system, and then performing a hill climb on this cost function by randomly mutating the current configuration to locate a state of minimal potential. Monte Carlo simulations of molecules attempt to reproduce some distribution by following a Markov Chain, where in general the probability of moving from some given state to another state is higher as the new state decreases in energy. A Monte Carlo simulation could operate by calling this function MC-UPDATE either some set number of times, or until the energy of the system reached some sort of defined terminal behavior - say leveling off, or falling below some threshold. Again, U is a data structure encoding the system.

function MC-UPDATE
  $U' \leftarrow$ a random perturbation of U.
  $U \leftarrow U'$ with probability $p(U, U')$.
end function

Monte Carlo simulations have the advantage that these random mutations need not resemble the actual evolution of the system in any way, so that the number of times the update procedure is run may have very little to do with the actual time it takes for the system to reach a final configuration. Further, the loops in the molecular dynamics algorithm can become quite time-consuming for large systems, while the scoring function may not be so computationally intensive. Thus Monte Carlo simulations tend to run significantly faster than molecular dynamics simulations. However, the fact that the perturbations in a Monte Carlo simulation are unrelated to the actual dynamics of the system is also the major flaw in Monte Carlo methods. While the simulation may very quickly identify the final configuration of some system, it provides little to no information about how the system achieved that configuration. With molecular dynamics, on the other hand, one can simply save the state of the system after every update and so generate a movie detailing how it evolves with time.

Unfortunately, when modeling large systems, molecular dynamics can become very slow. In particular, consider a system with N particles, each of which exerts some force on all the other particles. In order to calculate all of the forces exerted on all of the particles in each update step of molecular dynamics, one must make $O(N^2)$ computations since there are $\frac{N(N-1)}{2}$ pairs of particles. In order to simulate a system for time t, we will be computing for $O(tN^2)$ time. For large systems (say N in the order of several thousand), this can quickly become too time-consuming. There

are two primary ways to address this problem. The first is to use parallelism to speed up the computations - effectively making the constant factor in the big-Oh notation small enough that $O(tN^2)$ becomes feasible for larger systems. The second is to sacrifice accuracy in the simulation, for example by computing some approximation of the force on each component which is easier to obtain.

2.1 Speed/Accuracy Trade-offs

Before outlining the major optimization that trades accuracy for speed, we can improve the basic numerical integration scheme given above by using Verlet integration to improve accuracy without sacrificing any speed. This is simply a trick with power series. Assuming that the position of a component at time t is given by $r(t)$, velocity by $v(t)$ and acceleration by $a(t)$, we have (assuming that all functions involved are real analytic):

$$\begin{aligned} r(t+\delta) =& r(t) + v(t)\delta + a(t)\frac{\delta^2}{2} + b\delta^3 + O(\delta^4) \\ r(t-\delta) =& r(t) - v(t)\delta + a(t)\frac{\delta^2}{2} + b\delta^3 + O(\delta^4) \\ r(t+\delta) =& -r(t-\delta) + 2r(t) + a(t)\delta^2 + O(\delta^4) \end{aligned}$$

Thus by keeping track of the previous position instead of the velocity at each time point, we can use this formula to calculate $r(t+\delta)$ with error $O(\delta^4)$, rather than the error $O(\delta^3)$ given by the method outlined in the previous update function:

function UPDATE(M, F)
  $a \leftarrow F/M.mass$
  $r \leftarrow 2M.position - M.oldposition + a\delta^2$
  $M.oldposition \leftarrow M.position$
  $M.position \leftarrow r$.
end function

This allows us to speed up the computation time by increasing δ so we need fewer update steps to simulate the same time frame with the same accuracy.

This still does not help with the $O(N^2)$ component of the problem however. To address this we observe that for most systems, and indeed for the systems we will be simulating here, the force between two components of the system decreases quite rapidly as the distance between them increases. Once the distance between two components is sufficiently large, it is reasonable to approximate the force between them as 0. Thus we would like to simply not consider pairs of components whose distance from each other is too large when calculating the forces in the system. This involves solving a *nearest-neighbors problem*, which is, roughly speaking, the following question:

Question 2.1. *Given a set of N points in space and a particular point p, which points in the set are within some fixed radius r of p?*

Of course, it is not sufficient simply to check the distance between each pair since that still requires $O(N^2)$ operations. Instead, we will utilize a probabilistic method based on hash tables for keeping track of nearby components.

We tile space with a three dimensional lattice spanned by $(0, 0, d)$, $(0, d, 0)$ and $(d, 0, 0)$ for some given value d, thus assigning every point in space to a particular cube in this lattice - a "voxel". Since the coordinates of this voxel are just integral multiples of d, we can insert components of the system into a hash table keyed by the voxel the component currently inhabits. A hash table is a data structure that stores $(key, value)$ pairs such that given a particular key, one can look up all $value$s that were stored with that key in near-constant time. This is accomplished by use of a *hash-function*. A hash function is a random function $h : K \to \{0, 1, \cdots, N-1\}$ where K is the set of possible keys and N is called the size of the hash table. By a random function we

mean, for example, a function chosen uniformly at random from the set of all functions from K to $\{0, \cdots, N-1\}$. A hash table operates by inserting the pair $(key, value)$ into the $h(key)$th cell of an array A of size N. To recover the *value* associated with a *key*, simply look a the $h(key)$th cell of A.

When calculating the forces acting on a component, we can use a hash table to extract all the components occupying voxels that are the same as or neighbors to the voxel occupied by the first component.

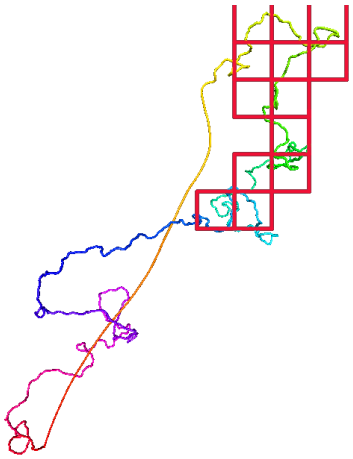


Figure 5: By dividing space up into a lattice of boxes ("voxels"), we can vastly speed up nearest-neighbor searches, significantly improving the runtime of the force-calculation algorithm.

For suitable choice of d, any components in other voxels will be far enough away that the forces they exert are negligible. Specifically, the function to calculate the force on a given component M with a set of voxels T is as follows:

function FORCE(M, T)
    Let V be the voxel M occupies in the set of voxels T
    Let L be a list of V and its neighboring voxels.
    $f \leftarrow 0$
    Let H be the hash table containing all components keyed on the voxels they are in.
    for Voxel v in L **do**
        Let C be the set of components in H with key v
        for Component c in C **do**
            Add the force on M from c to f
        end for
    end for
    return f
end function

In theory the complexity of this function could still be $O(N)$, where N is the number of components in the system; it might be that all components occupy the same voxel, or perhaps all voxels hash to the same value. However, in practice we expect some maximum density of components so that the first statement does not hold, and the second problem will happen with vanishingly small probability so that the average complexity of this calculation is actually constant, reducing the average complexity of the entire update function to $O(N)$.

2.2 Parallel Molecular Dynamics and GPUs

Parallelizing this computation is relatively straightforward from an algorithmic standpoint - molecular dynamics belongs to the class of so-called "embarrassingly parallel" problems. In particular,

each force calculation in the update function can be calculated independently of the others, which suggests that if a computer has k processors, it is easy to speed up the update function by a factor of k by assigning every kth iteration of each **for** loop to a different processor. Unfortunately, for most computers, k is somewhat modest. The current commercial desktop computer will likely not exceed $k = 12$, and even a very expensive system may only have a few hundred processors. However, an increasingly popular trend in simulations and scientific computing in general is to turn from clusters of CPUs to a few graphics processing units - GPUs. Essentially, a GPU consists of a very large number (several thousand) very weak processors that all operate in parallel. They were designed to handle the often highly parallel jobs involved in computer graphics. GPU development has proceeded at a breakneck pace, driven in large part by the video game industry and its increasing focus on high frame-rate photorealistic graphics [11]. The processors in GPUs are significantly more limited in their independence than the individual cores in a multi-core CPU however, and so some care must be taken to properly parallelize the molecular dynamics algorithm - in particular with reference to the hash tables involved in the voxel optimization.

The simulator used in this thesis was implemented on NVIDIA Tesla GPUs and coded using the CUDA programming language. We achieve a significant practical speedup over previously tested CPU-based implementations (simulations running on the order of days are reduced to hours). CUDA compliant GPUs cluster the individual processing units on a GPU into a hierarchy of groups. Small sets of 16 processors form *warps*. A potentially large number of warps (the exact number is a variable that may be adjusted by the programmer) are grouped into *blocks*. The set of blocks working on a particular job form a *kernel*. These hierarchies outline the degree of independence of the processors. Processors running for different kernels will be performing completely unrelated tasks while processors in the same warp are performing essentially exactly the same operations at the same time.

The simplest way to model a GPU is as a large array of processors each possessing a unique identification number. All of these processors run the same program. Differences in execution are obtained by using the identification number. For example, the following psuedo-code will add two large vector arrays A and B and store the result in another large array C. Id is the identification number of a processor.

function AddArrays(A, B, C)
  $C[Id] \leftarrow A[Id] + B[Id]$
end function

The general GPU algorithm for the simulator consists of these two functions, where M is a list of components in the system, F is a list of force vectors, and T is a table of voxels:

function GetForces(M, T)
  $Monomer \leftarrow M[Id]$
  $F[Id] \leftarrow$ Force($Monomer, T$) **return** F
end function
function Update(M, F, T)
  Update($M[Id], F[Id]$)
  Insert $M[Id]$ into an appropriate voxel in T.
end function

A simulation simply consists of many alternating calls to these functions.

In any multi-threaded program one must worry about synchronization issues. In this case, we can run into trouble when inserting components into the table of voxels. The voxel table is implemented as a hash table on the GPU. Thus we need to synchronize access to cells in the hash table. This is complicated by the peculiarities of GPU architecture. All sixteen threads in a warp not only all run the same code, they in fact operate in lock-step. That is, consider the following function:

function EvenOdd(A, B, C)
  if Id is even **then**
    $C[Id] \leftarrow A[Id] + B[Id]$
  else

$C[Id] \leftarrow A[Id] - B[Id]$
end if
end function

If 16 threads in a warp are executing this function, half of them will take one branch of the conditional statement and half will take the other. However, instead of jumping directly to the appropriate branch of the conditional, the warp as a whole will run through every statement in the program, and simply turn off the threads for which a particular statement does not apply. Therefore during the execution of this function, half of the threads in the warp will be idle at any given point. This branching behavior causes some difficulties in implementing the hash table used in the simulation algorithm. In the use of this hash table, many different threads may be simultaneously attempting to insert items into the table. Occasionally, this will result in a race condition in which two threads attempt to insert items into the same memory location. In traditional CPU multi-threaded applications, this problem can be solved via the use of *locks*, which are essentially memory objects that only one processor is allowed to access at a time. CUDA provides programmers with a few *atomic* functions - these are functions that may perform multiple read and write operations on memory, but are guaranteed to complete without any other GPU thread accessing the relevant memory locations. One of these functions is the ATOMICCAS(p, a, b), which compares memory location p to value a, rewrites the value stored at p to be c if the old value was a, and returns the old value of the data stored at p. On a CPU, this function could be used to implement a lock as follows (here L is some integer):

function LOCK(L)
  while ATOMICCAS$(L, unlocked, locked) \neq unlocked$ **do**
    PASS
  end while
end function
function UNLOCK(L)
  $L \leftarrow unlocked$
end function

This is a simple mechanism in which a variable L is used as a lock. When a thread calls LOCK, it will attempt to set L to the value *locked*, and will only leave the while loop and return when L is set to *locked* and was previously *unlocked*. On a CPU this is a reasonable way to perform locking because threads that are waiting on the lock may simply not be run in favor of threads that actually have tasks that they can perform. However, on a GPU this procedure is problematic because of the lock-step way in which the processor operates. If multiple GPU threads call the LOCK function at once, then only one of them will get the lock. The others will all be stuck in the while loop. However, since the GPU executes programs by having all processors execute the same instructions at the same time while turning off processors for whom the instructions do not apply, the GPU may turn off the processor that acquired the lock while all the processors execute the loop. Since the loop cannot end until the processor that acquired the lock is reactivated, in a simple GPU architecture the GPU may enter a condition in which it cannot proceed forward in execution - a deadlock. In more complicated architectures, the GPU can explore many different branches of conditional statements a bit at a time so that instead of reaching a deadlock the operation is simply made significantly slower, which is still an undesirable outcome. Thus synchronizing the hash table operations will require some care.

The hash table implementation used in these simulations makes use of *linear probing*. This is a method of resolving hash collisions that is particularly memory-cache friendly. Since memory transfers are typically a large part of the cost of any GPU program, linear probing is especially suited for a hash table on a GPU. In linear probing, (key,value) pairs are inserted into an array H of size n which is initially empty according to the following function, which makes use of a random hash function h:

function INSERT$(key, value)$
  $x \leftarrow h(key)$
  while $H[x] \neq (key, value)$ **do**

```
        x ← x + 1  mod n
    end while
    H[x] ← (key, value)
end function
```

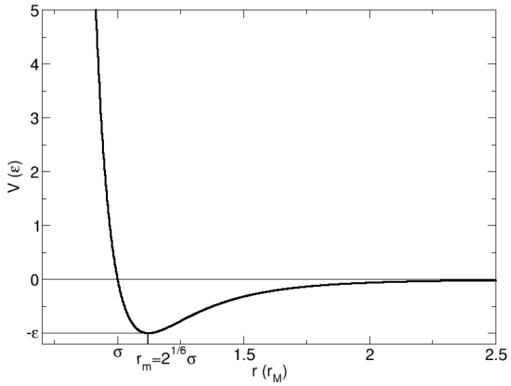
In our application, many components may be inserted into the table with the same key (since many components might occupy the same voxel). The table must maintain a list in each cell corresponding to a hash value and insert any components with key corresponding to a given cell into the list stored at that cell. To insert components into our hash table elements on a GPU, one must adopt a more complicated synchronization scheme, such as the following functions which serialize access to the entry in a hash table H corresponding to the key k:

```
function INSERT(H, key, component)
    x ← h(key)
    loop
        if ACQUIRE (H, x, key) then
            ADDCOMPONENT(H[x], component)
        end if
        x ← x + 1  mod n
    end loop
end function
function ACQUIRE(H, x, key)
    while H[x] = UNUSED do
        ATOMICCAS(H[x].key, UNUSED, key) = UNUSED
        if H[x].key = key then
            return True
        end if
    end while
    return False
end function
function ADDCOMPONENT(cell, component)
    for i = 0 to i = VOXELSIZE do
        if ATOMICCAS(cell[i], EMPTY, component) = EMPTY then
            return
        end if
    end for
end function
```

Here were are implementing the cells in the hash table as lists of size $VOXELSIZE$, which corresponds to some maximum number of components we expect to occupy any given voxel at any time during the simulation. We assume $VOXELSIZE$ is small enough that looking through the entire list of components in a voxel is relatively cheap.

2.3 Simulation Details and Initial Configurations

The simulations we will be discussing are all simulations of polymers. Specifically, the system consists of a long sequence of monomers such that each monomer in the chain exerts some bond force on the two adjacent monomers in the chain as well as a weak attractive force towards every other monomer in the polymer. Monomers also exert a short-range repulsion towards other monomers so that our simulated monomers (which are just points in space) cannot come too close to each other (since real-life monomers have real radii and cannot intersect each other). Specifically, we use the *Lennard-Jones* potential between pairs of monomers, and the *Finitely Extensible Non-linear Elastic* (FENE) potential for bond forces. The Lennard-Jones potential gives a computationally simple model for the van der Waals attraction between two monomers:



$$V_{LJ}(r) = \frac{C_{12}}{r^{12}} - \frac{C_6}{r^6}$$

where r is the distance between two monomers and C_{12} and C_6 are constants dictated by the physical properties of the monomers (the subscripts are simply a mnemonic for the fact that C_i is divided by r^i). The $\frac{C_{12}}{r^{12}}$ term generates the repulsive force between the monomers. In addition to preventing monomers from "intersecting", this repulsive force also interacts with the FENE potential to dictate the bond length of the polymer. The FENE potential is [2]:

$$V_{FENE}(r) = -\frac{1}{2}kb^2 \log\left(1 - \frac{r^2}{b^2}\right)$$

We can differentiate to get the FENE force, which has a much simpler form:

$$F_{FENE}(\vec{r}) = -k\left(1 - \frac{r^2}{b^2}\right)\vec{r}$$

Note that this force is always attractive. A stable bond between adjacent monomers is created by the addition of the Lennard-Jones repulsion to the FENE attraction, which pulls monomers adjacent on the linear chain of the polymer closer together than they would be normally be due to the ordinary Lennard-Jones attractive force. The repulsive force around monomers also serves to preserve the topology of the polymer - a true polymer can never intersect itself, and bonds cannot pass through each other. By making the FENE force strong enough, the bonds in the simulation will become small enough that the repulsive Lennard-Jones force will prevent non-bonded monomers from coming close enough to each other to allow one bond to pass through another. We use FENE and Lennard-Jones parameters such that the bond-length is 0.34 and the lowest-energy distance between non-bonded monomers is 0.5. The units used in the simulation are somewhat arbitrary as we are simulating an idealized version of the chromatin polymer. We will refer to units of distance and time as "simulation units". In general, the important values are the ratios between various values of simulation units. Here we note that since non-bonded monomers do not come closer than 0.5 simulation units to each other, it is impossible for one bond to pass through another.

Remarkably, these simple forces will be sufficient to allow a polymer to collapse into a fractal globule state. However, in order to observe such a collapse we must be able to generate an initial condition of the polymer that can be simulated. A reasonable thing to try for this is to simply use a random walk as an initial condition. Unfortunately, a polymer whose monomers are located at the steps of a random walk will typically have monomers located much closer to each other than the Lennard-Jones repulsion would allow. These are therefore unphysical configurations, and attempting to simulate one will cause the computer to calculate excessively large forces acting on the monomers that are too close to each other.

In order to fix this problem we apply a process called *energy minimization*, which simply means that we perturb an undesirable initial configuration until it becomes more stable. This can be accomplished by essentially running a heavily damped version of the molecular dynamics simulator. That is, in the update function, we update the velocity by simply setting the position

equal to the current position plus some small multiple of the calculated acceleration. This has the effect of performing gradient descent on the potential energy of the polymer configuration and so will converge to a more stable configuration eventually. However running the simulator is a slow process and we would prefer something more effective. In fact, it is possible to perform energy minimization quickly on an ordinary CPU even for polymers consisting of thousands of monomers.

2.4 Energy Minimization

The algorithm makes use of a data structure called a *heap*. A heap is a data structure that holds a set of elements with associated keys. The heap is designed so that at any given time one has quick access to the element in the heap with the smallest key. It supports the operations INSERT, DELETE, FIND-MIN, DELETE-MIN and DECREASE-KEY. These operations add an element to the heap, remove an element, return the element with minimum key, delete the element with minimum key (and return it), and decrease the key of an element in the heap.

There are many ways to implement heaps. One of the simplest (and still quite effective) is the *binary heap*, which maintains the elements in the heap in a balanced binary tree with the root node being the element with smallest key. In a binary heap with n elements, all of the operations take $O(\log(n))$ time, with the exception of FIND-MIN, which is constant time. A more sophisticated heap structure is the Fibonacci heap, in which DELETE and DELETE-MIN take (amortized) $O(\log(n))$ time and *all other operations* take constant (amortized) time. In an application in which many calls to the various heap functions are to be made on a heap that holds a large number of items the Fibonacci heap can provide some savings. This is the heap structure that we will use in our energy minimization scheme.

The algorithm for energy minimization is quite simple - for every pair of monomers we insert an element into the heap whose key is the distance between the pair. Then we take the minimum element out of the heap - this is the pair of monomers that are closest to each other and so represents the worst point of the current configuration. We move one of the monomers so that the distance between this pair is acceptable and insert the pair back into the heap. After repeating this operation some fixed number of times, we iterate over all the pairs of monomers and call DECREASE-KEY on every pair whose actual distance is now less than the distance indicated by the key value. Here we present a pseudo-code version of the algorithm. The input is a list of monomers $A[0], \cdots, A[n]$. $BONDLENGTH$ is the equilibrium distance between bonded monomers. $nb_{threshold}$ is a minimum acceptable distance between non-bonded monomers. $b_{threshold}$ is a minimum acceptable deviation of bondlength from $BONDLENGTH$. $REFRESH$ is a parameter that tells the algorithm how often to iterate DECREASE-KEY over all pairs.

function ENERGYMINIMIZE(A)
  Let H_{nb} be a Fibonacci heap, to be used for non-bonded interactions.
  let H_b be a Fibonacci heap, to be used for bonded interactions
  for $i = 0$ to $n - 2$ **do**
    for $j = i + 1$ to $n - 1$ **do**
      Insert (i, j) into H_{nb} with key $|A[i] - A[j]|$.
    end for
  end for
  for $i = 0$ to $n - 2$ **do**
    Insert $(i, i + 1)$ into H_b with key $-||A[i] - A[i + 1]| - BONDLENGTH|$.
  end for
  $counter \leftarrow 0$
  loop
    $counter \leftarrow counter + 1$.
    $(i, j) \leftarrow H_{nb}.extractmin()$.
    if $|A[i] - A[j]| < nb_{threshold}$ **then**
      Move $A[i]$ and $A[j]$ to acceptable positions.
    end if
    Insert (i, j) back into H_{nb}.

```
        (i, j) ← H_b.extractmin().
        if ||A[i] − A[j]| − BONDLENGTH| > b_threshold then
            Move A[i] and A[j] to acceptable positions.
        end if
        Insert (i, j) back into H_b.
        if counter == REFRESH then
            counter ← 0.
            for i = 0 to n − 2 do
                for j = i + 1 to n − 1 do
                    Let k be the key with which (i, j) was inserted into H_nb.
                    if k > |A[i] − A[j]| then
                        call DECREASEKEY to change the key of (i, j) to |A[i] − A[j]|.
                    end if
                end for
            end for
        end if
    end loop
end function
```

Although this algorithm does have $O(N^2)$ asymptotic complexity, since we can tune the constant factor by having a large $REFRESH$ value, it runs quite quickly in practice on polymers of size up to 4096 monomers and so is sufficient for the purposes of our simulations.

Thus our procedure for generating an initial configuration of N monomers is:

1: Form a random-walk of length N

2: Run the energy-minimization software on the random walk.

In general we will form two types of initial configurations - *closed* and *open*. A closed initial configuration is one that is a closed loop - so that the polymer is a ring rather than a chain with two ends. Simulating these loops will be useful when investigating the phenomenon of topological mutation since a loop necessarily cannot change its topology. An open configuration is simply one that is not a closed loop. Note that this use of open and closed is very different from the meaning of open and closed compartments of chromatin which will be discussed later.

2.5 The N Body Problem

The process of molecular dynamics can be viewed as numerically solving the so-called N *body problem*, which, roughly speaking, can be stated as:

Question 2.2. *Given N particles in space with initial positions and momenta, where will the particles be after some time t?*

Classically, this is meant to be a problem in celestial mechanics, where the particles are point masses and the only relevant force is that of gravity. The $N = 2$ case for gravitational attraction has the well-known solution described by Kepler in which the particles trace out a conic section. This can be derived from elementary calculus. In particular, we note that there are $2 \times 3 = 6$ functions we wish to solve for - one for each coordinate of each particle's position and momentum. The system as a whole has 10 values that must be conserved - the total energy, the three components of angular momentum, and six equations describing the uniform motion of the center of mass. Since $6 < 10$, one would intuitively expect to be able to use these constraints to solve for the motions of the particles and indeed this is possible. Surprisingly, however, the $N = 3$ case is significantly harder. In fact, it is impossible to solve the differential equations for $N = 3$ using a similar method to the one described for $N = 2$.

The motion of N particles in 3 dimensions is described by a path in $R^M = \mathbb{R}^{3N} \times \mathbb{R}^{3N}$ (called *phase space*) where the first $3N$ coordinates give the positions of the particles and the last $3N$

coordinates give the momentum. We are interested in solving some differential system

$$S : \frac{dx_i}{dt} = F_i(x)$$

Where F_i is given by an element of some algebraic extension of $\mathbb{R}(x_1, \cdots, x_M)$. A function $I : \mathbb{R}^M \to \mathbb{R}$ is called a *first integral* of the system S if for any path $r(t)$ that satisfies S, $I(r(t))$ is constant. Note that each of the conserved quantities used to solve the $N = 2$ case are first integrals. It turns out that those 10 equations are in some sense the only first integrals and so the method of first integrals is insufficient to solve the three-body problem.

We can re-write the condition $I(r(t)) = c$ as

$$\sum_{i=0}^{M} \frac{\partial I}{\partial x_i} F_i(x) = 0$$

Consider the field $K = \mathbb{R}(x_1, \cdots, x_m, t)$. Suppose $I_1, \cdots, I_m$ are algebraically independent first integrals of S, and $r(t)$ is a solution to S, all residing in some algebraic extension of K, L. Then for given constants $a_1, \cdots, a_m$, we have $I_i(r(t)) = a_i$. So that the I_i give m constraints on r. Now since L is algebraic (and hence finite) over K and the transcendence degree of K over $\mathbb{R}$ is $m + 1$, the transcendence degree of L over $\mathbb{R}$ is also $m + 1$. Thus with our m constraints we (in theory) recover $r(t)$. The main theorem about the three-body problem is that all first integrals algebraic over K are in fact algebraic over the 10 first integrals already described and so there are only 10 algebraically independent first integrals. This was first proven by Brun's in 1887. In fact, this holds in much greater generality [4]:

Theorem 2.3. *If S is the differential system describing n particles in $\mathbb{R}^p$, for $p \leq n$, then S has at most $(p + 1)(p + 2)/2$ algebraically independent first integrals.*

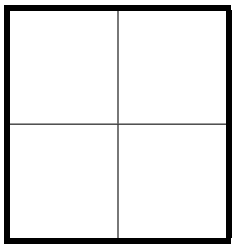
For the case $n = p = 3$, this leaves us with the 10 classical first integrals.

2.6 Barnes-Hut Algorithm

Even though there is no nice closed solution to the n-body problem, it is still important to be able to approximate the solution. The algorithm outlined previously does a good job of this on GPU architecture for the applications it was needed for, but it is hardly the only solution. Indeed the basic algorithm described is one of the simplest - the complexity (and speed) comes from the randomized hash table usage. In this section and the next we will outline two deterministic algorithms that achieve respectively $O(N \log N)$ and $O(N)$ update times. Both of these algorithms rely on the observation that the force due to a large group of particles can be well-approximated far away from the particles by a simpler equation. We describe the algorithms in two dimensions for ease of exposition. The three-dimensional situation is in all cases analogous.

The first algorithm we will discuss was described in 1986 by Barnes and Hut [7]. It belongs to a class of n-body algorithms known as tree-codes because it divides space up into a tree structure. The Barnes-Hut algorithm makes use of the basic approximation that a group of particles of total mass M can generally be approximated from far away as one particle of mass M located at the center of mass of the group. In broad strokes, for every particle the Barnes-Hut algorithm finds ways to efficiently partition the remaining particles into groups that are far enough away to use this approximation, allowing the force on the particle to be calculated without considering every other particle individually.

The fundamental operation in the Barnes-Hut algorithm is the division of space into quadrants (octants in the three-dimensional case). That is, given a "computational box" (an area of space in which particles to be simulated reside, in our case the smallest square containing the particles), we divide the box into four equally sized subboxes:



This leaves us with four sub-boxes. For each of these sub-boxes there are three possibilities: a sub-box may contain zero particles, one particle, or many particles. In the first two cases we will leave the sub-box alone. In the latter case we will subdivide the sub-box into four more sub-boxes. We repeat this procedure until all sub-boxes contain at most one particle. The sub-boxes created in this process naturally form a tree structure whose nodes are sub-boxes of the computational box and whose leaves are sub-boxes that contain at most one particle. This process can be formalized in the following algorithm, which takes a current node (sub-box) in the tree B as an argument and returns the tree structure for the system. For the initial call to the function, B should be the entire computational box.

function MAKE-TREE(B)
  if B contains at most one particle **then**
    return B
  end if
  Subdivide B into four quadrants, B_0, B_1, B_2, B_3.
  for i in $0, 1, 2, 3$ **do**
    B. ADDCHILD (MAKE-TREE (B_i))
  end for
  return B
end function

In order to use this tree to compute forces we will need to know the total mass and center of mass of the particles contained within every sub-box in the tree. This can be efficiently computed by depth-first search the tree, as in the following algorithm which takes a tree of boxes T as an argument:

function FIND-MCM(T)
  Let B be the root of T.
  if B has no children **then**
    Let M be the mass of the particle in B, or 0 if B is empty.
    let C be the location of the particle in B, or NULL if B is empty.
    $B.mass \leftarrow M$.
    $B.CenterofMass \leftarrow C$.
    return
  end if
  B_0, B_1, B_2, B_3 be the children of B.
  $T[B_i]$ be the subtree of T with root B_i.
  FIND-MCM($T[B_i]$) for $i \in \{0, 1, 2, 3\}$.
  $M_i \leftarrow B_i.mass$ for i in $\{0, 1, 2, 3\}$
  $C_i \leftarrow B_i.CenterofMass$ for i in $\{0, 1, 2, 3\}$.
  $M \leftarrow \sum_{i=0}^{3} M_i$.
  $C \leftarrow \frac{1}{M} \sum_{i=0}^{3} M_i C_i$.
  $B.mass \leftarrow M$
  $B.CenterofMass \leftarrow C$.
  return
end function

This concludes the pre-processing step of the Barnes-Hut algorithm. Then the runtime of the preprocessing is bounded by the following lemma:

Lemma 2.4. *Let b be the machine-precision of the simulator (measured in bits). Then* Make-Tree *and* Find-MCM *take $O(Nb)$ time where N is the number of particles. If the particles are uniformly distributed - that is if the tree is balanced - then the algorithms take $O(N\log(N))$ time.*

Proof. The cost to insert a particle into the tree in Make-Tree is simply the depth of the particle in the tree. With each subdivision in the Make-Tree algorithm, we specify at least one bit of precision in the location of the particle so that the cost-per-particle is at most b, which gives us our bound. Similarly, Find-MCM performs a depth-first-search on the tree, which therefore takes time $O(Nb)$ since there are $O(Nb)$ vertices in our tree.

If the tree is balanced, then it has depth $\log(N)$ and so our bounds reduce to $O(N\log(N))$. Note that $\log(N) \leq b$ so that this is a special case of the previous bound. □

Now we come to the actual calculation section of the algorithm. This is fairly straightforward: for each particle in our system, we traverse the tree starting at the root. For each node we compute the ratio $\frac{d}{r}$, where d is the size of the sub-box at that node and r is the distance from our particle of interest to the center of mass of that sub-box. If $\frac{d}{r}$ is less than some user-supplied accuracy parameter θ, then we approximate the force of all the particles contained in that sub-box on our particle by considering the particles in that sub-box to be equivalent to one meta-particle located at the center of mass of the sub-box. In precise terms, the procedure is given by the following algorithm, which takes a particle P to calculate forces on, a root-node sub-box B and the parameter θ:

function Calculate-Force(P, B, θ)
  $F \leftarrow 0$.
  if B contains only one particle **then**
    $F \leftarrow$ Force on P due to the particle in B.
    return F.
  end if
  if B is empty **then**
    return F.
  end if
  $d \leftarrow$ side-length of box B.
  $r \leftarrow$ distance from P to $B.CenterofMass$.
  if $d/r < \theta$ **then**
    $F \leftarrow$ Force on P due to a particle of mass $B.mass$ located at $B.CenterofMass$.
    return F.
  else
    Let B_0, B_1, B_2, B_3 be the children of B.
    $F \leftarrow \sum_{i=0}^{3}$ Calculate-Force(P, B_i, θ)
    return F.
  end if
end function

The run-time of this step is expected to be $O(\log(N))$. This is only heuristically known in practice, but the intuition is established by the following lemma:

Theorem 2.5. *We say that a box B is a* straight-descendant *of a box B' if every descendant in the chain from B' to B is the same "type" of child (either lower-left,lower-right,upper-left,upper-right). For a constant $\theta > 0$, for all particles ,* Calculate-Force *runs in time $O(b)$ (or $O(\log(N))$ for a balanced tree) for any particle whose box is a straight descendant of the root node.*

Proof. Let B be a straight-descendant of B'. Let L be the side-length of B. Let M be a descendant of B', and let c be the distance between B and the first common ancestor of M and B. Then if b is a particle in B and m is the center of mass of M, then the distance between b and m is at least $(2^{c-1}-1)L$. Let r be the distance between M and the first common ancestor of M and B. Then the size of box M is $2^{c-d}L$. Thus b is far enough away from M to consider all particles inside M as one unit if $2^{c-d}/(2^{c-1}-1) < \theta$. For sufficiently large c, this condition is approximately $2^{1-d} < \theta$, which

is true for $d > k$ for some constant k. So for every given ancestor of B between B and B', there are at most a constant number descendants of that ancestor for which the CALCULATE-FORCE function will not recurse. Therefore the work to run CALCULATE-FORCE on b is proportional to the depth of B, which is $O(b)$.

□

2.7 Fast Multipole Method

Here we present a somewhat more advanced method called the fast multipole method [3]. For a fixed error rate p the fast multipole method achieves a deterministic update time of $O(N)$ for a system with N particles. We will describe the method for use in a two-dimensional space here.

First we'll describe the broad strokes of the algorithm, and then give some more detail. The fast multipole method works by splitting space into a hierarchical mesh structure similar to the voxel structure used in the previously described simulation scheme. Suppose the entire simulation space is contained within some large box - call it $m[0,0]$. Divide $m[0,0]$ into four equally sized sub-boxes, $m[0,1], m[1,1], m[2,1], m[3,1]$. These form the level 1 mesh. Now divide each box in the level 1 mesh into four boxes, to obtain $m[0,2], \cdots, m[15,2]$. These are called the level 2 mesh. We continue to recursively create finer and finer meshes until the size of the boxes in a mesh reaches some user-defined value, generally such that the number of successive layers is $O(\log(N))$. We organize these meshes into a tree structure so that if $m[i,l]$ was one of the four boxes formed by subdividing $m[j,l-1]$, then $m[i,l]$ is considered a child of $m[j,l-1]$.

The major workhorse of the fast multipole method is an expression that allows one to approximate the influence of a large number of particles from some reasonable distance away. The algorithm uses the lowest-level mesh and this method to approximate motion (the tree structure is used to help with calculations, as will be seen). For every particle, one first calculates the force on that particle due to every other particle in its same lowest-level box and all the neighbors of that box. Then one obtains an approximation for the force exerted by all the other particles - which are separated from the particle of interest by at least one box-length - and adds that to the force on the given particle as well.

We will give an explicit construction of the algorithm after detailing the method for approximating forces.

The electrostatic potential created by a particle of charge q at the origin in $\mathbb{R}^2 \cong \mathbb{C}$ takes the form

$$\phi(z) = q\log(|z|) = q\mathrm{Re}(\log(z))$$

for whichever branch of the logarithm one cares to choose. We will play loose with the term "analytic" and take it to mean "a function that can be defined to be analytic on any simply-connected subset of $\mathbb{C}$, and whose real part is harmonic on $\mathbb{C}$ minus some finite set of points". The fast multipole algorithm will construct several terms of a series formula for an analytic (in this new sense) function f whose real part gives the total potential of the system at a point. We'll call this function f the potential, although it is different from the ordinary potential function in physics. Finding such an f is useful because the gradient of the real part of f (or the force field at a point), by the Cauchy Riemann equations, is $(\mathrm{Re}(f'), -\mathrm{Im}(f'))$.

Now the potential due to a single particle of charge q at location z_0 is

$$\phi_{z_0}(z) = q\log(z - z_0)$$

For $|z| > |z_0|$ we have

$$\phi_{z_0}(z) = q(\log(z) + \log(1 - z_0/z)) = q\left(\log(z) - \sum_{i=1}^{\infty} \frac{{z_0}^i}{iz^i}\right)$$

The core tool of the fast multipole method is the following fact:

Lemma 2.6. *Now if we have n particles of charges $q_1, \cdots, q_n$ at locations $z_1, \cdots, z_n$, then for $|z| > r = \max |z_i|$, we have*

$$\Phi(z) = Q \log(z) + \sum_{i=1}^{\infty} \frac{a_i}{z^i}$$

where $Q = \sum q_i$ is the total charge and $a_i = -\sum_{j=1}^{m} \frac{q_j {z_j}^i}{i}$.

Proof. We have

$$\Phi(z) = \sum_{i=1}^{n} \phi_{z_i}(z) = \sum_{i=1}^{n} q_i \left(\log(z) - \sum_{j=1}^{\infty} \frac{{z_i}^j}{j z^j} \right)$$

and so grouping terms of like power of z, we recover the expression for $\Phi(z)$. □

This already gives us some idea of what the final potential will look like. However we are interested in approximating this potential, so we need an error bound on this series evaluated to p terms (we'll assume our computer can approximate log to higher precision than we need it to).

Lemma 2.7. *Suppose we have n particles of charges $q_1, \cdots, q_n$ located at $z_1, \cdots, z_n$ with $r = \max |z_i|$. Then for $|r/z| > 2$,*

$$\Phi_p(z) = Q \log(z) + \sum_{i=1}^{p} \frac{a_i}{z^i}$$

Approximates $\Phi(z)$ to within $\left(\frac{1}{2}\right)^p \sum_{i=1}^{n} |q_i|$.

Proof. We want to compute

$$|\Phi(z) - Q \log(z) - \sum_{i=1}^{p} \frac{a_i}{z^i}| = |\sum_{i=p+1}^{\infty} \frac{a_i}{z^i}|$$

We have $|a_i| \leq (\sum |q_i|)\, r^i$. Set $W = \sum |q_i|$. Then we can write

$$\begin{aligned} |\sum_{i=p+1}^{\infty} \frac{a_i}{z^i}| \leq & W \sum_{i=p+1}^{\infty} \frac{r^i}{|z|^i} \\ = & W \frac{|r/z|^{p+1}}{1 - |z/r|} \end{aligned}$$

For $|r/z| > 2$, this gives an error of less than $W(1/2)^p$ as desired. □

Φ_p is the function f we are seeking. We can use it to closely approximate the force one group of particles exerts on another. For example, suppose particles $x_1, \cdots, x_n$ are contained within some ball A of radius r and particles $y_1, \cdots, y_m$ are similarly contained within some other ball B of radius r such that the minimum distance between an x_i and a y_j is at least r. Then if $\Phi(z)$ represents the potential due to the $x_i's$ centered at the center of A, then by the above error bound, the first p terms of $\Phi(y_i)$ give a very good estimate of $\Phi(y_i)$ for all i. Similarly, we can find a good approximation for the potential function due to the $y_i's$ near the $x_i's$. We call the first p terms of Φ the p-term multipole expansion.

Recall the tree-structure of boxes described previously. Let's define $\Phi_{i,l}$ to be the p-term multipole expansion about the center of the box of the potential field created by the particles in box $m[i,l]$. Let $\Psi_{i,l}$ be the p-term multipole expansion about the center of the box of the potential field created by all the particles outside of $m[i,l]$ and its eight immediate neighbors. $\Psi_{i,l}$ is the expression we will ultimately be interested in computing, for every box in the finest mesh layer. Let $\Theta_{i,l}$ be the p-term multipole expansion about the center of the box of the potential field created by all the particles outside of $m[i,l]$'s *parent* box and each of the parent box's neighbors. Θ will be a useful computational intermediate allowing us to efficiently find Ψ.

We'll say that two boxes in the same layer are neighbors if they share an edge or corner. Otherwise they are separated. For a given box $m[i,l]$, the interaction list of the box is the set of boxes that are children of the parent of $m[i,l]$ and the neighbors of the parent of $m[i,l]$, but are also separated from $m[i,l]$. The interaction list represents the difference between $\Psi_{i,l}$ and $\Theta_{i,l}$:

$$\Psi_{i,l} = \Theta_{i,l} + \sum_{m[j,l] \in \text{Interaction List}} \Phi_{j,l}$$

Now we give the full algorithm.

1: (Calculate Φ's at the finest layer)

Form the tree structure. For every box $m[i,l]$ in the finest layer, calculate $\Phi_{i,l}$.

2: (Calculate $\Phi_{i,l}$ for all boxes in the tree)

Suppose the tree has depth d. Then from step one we already have $\Phi_{i,d}$ for all i. Given $\Phi_{i,l}$ for all children of a box $m[j,l-1]$, we obtain $\Phi_{j,l-1}$ by translating the expansions for each child to be centered at the center of box $m[j,l-1]$ and adding them up. Thus for each box $m[i,d-1]$ in layer $d-1$, we can calculate $\Phi_{i,d-1}$. This allows us to calculate $\Phi_{i,d-2}$ for all i and so on for all $\Phi_{i,l}$.

3: (Calculate Θ's and Ψ's) Set $\Theta_{i,1} = 0$ for $i \in \{0,1,2,3\}$.

for $l = 2$ to d **do**
for $i = 0$ to $4^l - 1$ **do**
Calculate $\Psi_{i,l}$ by adding $\Phi_{j,l}$ for each $m[j,l]$ in the interaction list of $m[i,l]$ to $\Theta_{i,l}$.
Calculate $\Theta_{j,l+1}$ for each child of $m[i,l]$ by translating $\Psi_{i,l}$ to the center of each child box.
end for
end for

4: (Compute Forces) For every box $m[i,d]$ in the finest mesh layer, for every particle q_j in $m[i,d]$, calculate explicitly the interaction between q_j and every particle in $m[i,d]$ and its neighbors. Then calculate an approximation for the interaction between q_j and all the remaining particles using $\Psi_{i,d}$ and add these values together to get the the value of the force field at q_j.

Finally, we can analyze the run-time of the fast multipole method:

Theorem 2.8. *Suppose d is $O(\log(N))$, where d is the depth of the tree. Then the fast multipole method takes $O(N)$ time per update.*

Proof. Since d is $O(\log(N))$, steps 1 and 2 takes $O(N)$ time because calculating the multipole expansion for a given particle takes constant time for fixed p and the total number of boxes is $O(N)$. For step three each item in the **for** loop takes constant time. Since there are $O(N)$ boxes, this is again total time $O(N)$. Finally step four is a constant amount of work for each particle and so is $O(N)$. This gives the overall runtime of $O(N)$. □

3 Generating Structures Consistent with Contact Maps

One motivation for performing polymer simulations is that it allows one to collect much more detailed data (modulo accuracy of the model) than can be acquired from a real-life experiment. However, since there are experimental procedures that do provide some data about the conformation of polymers it is reasonable to compare these experiments with simulated data. An example is the Hi-C procedure, which measures the frequency of physical contact between all pairs of positions along the genome. Briefly, Hi-C accomplishes this by first treating the genome with formaldehyde, which forms bonds between physically proximate regions of the genome, thus freezing these contacts in place. Then the genome is cut into tens of thousands of fragments each on the order of a thousand base pairs long. The fragments that were in contact with each other in the original structure are linked together by the formaldehyde treatment and so remain close together. Next, the ends of the fragments in this soup are all tagged with a marker called biotin. Then the fragment pairs are treated with a protein that causes them to bind together, creating loops from fragments that were in contact. These loops are then re-broken to produce small segments of DNA marked with biotin. The biotin markers are isolated and the attached segments are sequenced. Using a reference full genome sequence, these sequences are matched back to their original locations in the genome thus recovering contact data between different fragments.

The data provided by a Hi-C experiment is a series of *reads* indicating an observed contact between two segments of the genome, from which one can calculate a *contact map* - a set of values $p_{i,j}$ indicating the probability that the ith and jth segments are in contact in the cell. The contact map can be visualized as a matrix whose i, j entry is $p_{i,j}$. Typically this data is presented as a heatmap, where darker colors indicate higher values of $p_{i,j}$.

From a gross perspective, this permits us to separate segments of the genome into so-call *open* and *closed* chromatin, such that contacts between segments of the same type are more likely than contacts between segments of differing types. Open chromatin represents less dense regions that may be transcriptionally active, while closed chromatin consists of more densely packed, less accessible regions of the genome. On top of this overall perspective, one can also identify particular pairs (i, j) such that $p_{i,j}$ is significantly elevated above and beyond other considerations such as linear distance and open/closed-ness. These represent specific segments that appear to have some distinct affinity for each other in the cell. All these data give some rough information about the geometric structure of the chromatin in the cell.

3.1 Evidence for the Fractal Globule

One notable statistic that can be derived from this data is the *contact probability* of the genome. This is a function $\chi(n)$ that gives the probability that two regions of the genome separated by a distance of n along the linear chain of the polymer will be in close proximity to each other. The contact probability functions derived from Hi-C data show a scaling of $\chi(n) \propto n^{-1}$ on the one to ten megabase scale. This supports the hypothesis that the chromatin is organized in a *fractal globule* structure on at this scale. A fractal globule is a polymer conformation model that is unknotted and densely packed while maintaining the ability to "locally unravel" any small segment of the globule without disturbing the rest of the structure. One can calculate that a fractal globule structure shows a contact probability scaling of $\chi(n) \propto n^{-1}$. The other prominent model for chromatin structure when the Hi-C method was published was the *equilibrium globule*. This is a model in which a polymer is viewed as a random-walk. It can be shown that the contact probability of a random walk scales as $\chi(n) \propto n^{-3/2}$, so that the contact probability gives evidence for the fractal globule over the equilibrium globule.

In addition to this Hi-C data, there is also data that can be obtained from 3D-FISH, which is a protocol that can give information about distances between a few specific locations on the genome. Using this procedure, one can get an idea for how the 3-D distance $D(s)$ between locations scales as a function of the distance s between the locations along the polymer. This scaling is measured to follow a power law as well - $D(s) \propto s^{1/3}$. This is the the same scaling calculated for a fractal globule. In contrast, for an equilibrium globule the scaling is $D(s) \propto s^{1/2}$.

A fractal globule is expected to form in a hierarchical collapsing procedure. In this process, small subchains of the polymer will first collapse into small globules, turning the polymer into a "polymer of globules". Sub-chains of this new meta-polymer will again collapse into globules, ideally with the same statistical properties of the lower-order globules. This process repeats until the entire polymer has collapsed. Since at each stage the formation of globules follows the same pattern the overall structure should exhibit some "self-similarity" properties, hence giving the fractal globule its name. The fractal globule is a *meta-stable* state. What this means is that a fractal globule, left to its own devices for an arbitrarily long time, will not maintain its fractal globule structure. Over time, the globule will come to resemble the random-walk, or equilibrium globule.

This process of equilibration is mediated by changes in the topology of the globule. We have mentioned several times that the fractal globule is an unknotted structure. Of course, polymers are not in general closed loops and so the mathematical definition of unknotted does not technically apply. To get some intuition of what we mean by unknotted, consider the case of an infinitely long polymer. This polymer can be thought of as a closed loop in the compactification of $\mathbb{R}^3$, S^3 and so is unknotted in the precise sense. The infinitely long polymer should collapse in essentially the same way as the finite polymer - first very local interactions will cause local collapses that convert the polymer into an infinitely long polymer of globules, which will in turn collapse into a polymer of meta-globules and so on. After some time, the result will be an infinite polymer of fractal globules. Since the motion of a polymer in space must describe an isotopy, the resulting polymer must be unknotted. In this sense, if we were to select fractal globules whose end-points were on the surface of the globule, we would find with very high probability that by extending the end points to infinity we would obtain an unknot. The process of conversion from a fractal globule to an equilibrium globule is precisely the process of the ends of the polymer traveling through and penetrating the globule structure due to random motion, thus generating knots in the globule and destroying the fractal structure. This equilibration process takes a very long time - order the length of the polymer. For large polymers this makes the fractal globule the dominant state for a significant time frame (hence meta-stable).

3.2 Unmodified Simulations Produce Fractal Globules

One sanity check on the validity of our polymer simulations is to see how well the naive simulation can approximate the overall statistics of the genome, as approximated by the fractal globule. We can perform the following simple experiment:

1. Generate an initial condition using the procedure described in section two.
2. Simulate this polymer until it collapses into a globule structure (or fails to collapse for some period of time).
3. Compare the statistics of this simulated globule with the known statistics of the fractal globule.

A sequence of snapshots from a simulated collapse illustrates this principle of hierarchical globule formation, as in Figure 6.

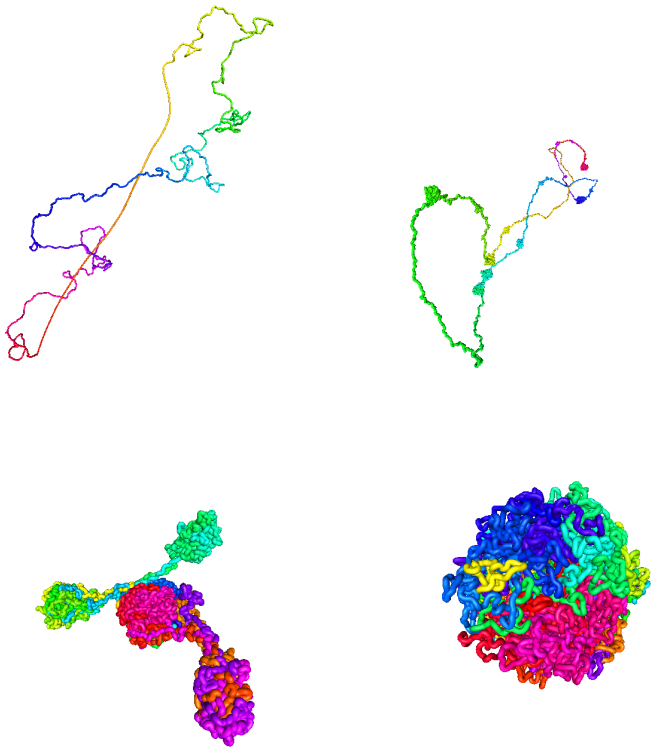


Figure 6: Four still images over the course of a simulation. Over time the simulated polymer forms larger and larger globules, eventually becoming a fractal globule.

This provides some evidence that the simulated collapses are generating fractal globule structures. We can also look at the contact probability of the simulated polymer to get a more quantitative measure. A fractal globule should have a contact probability that follows a power law: $\chi(n) \propto n^{-1}$. Plotting the simulated contact probability on logarithmic axes shows that the simulations indeed achieve this property:

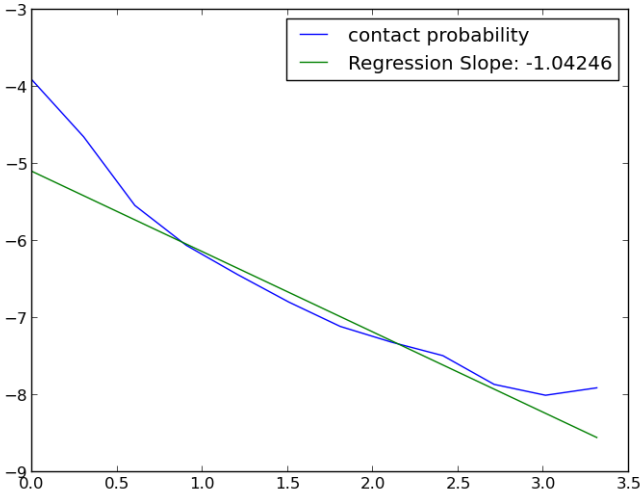


Figure 7: As seen from this log-scale plot, the contact probability calculated from simulated globules reproduces the fractal globule scaling seen in chromatin from Hi-C data.

Thus the simulated polymers can recover the gross statistical structure of chromatin measured by Hi-C experiments. We can also calculate a contact map from this simulated fractal globule. To do this, we imitate a Hi-C experiment. The monomers of the polymer are partitioned into several fragments, and the frequency with which these fragments are in contact is measured over 100 different fractal globules generated from different initial conditions. The resulting contact map is:

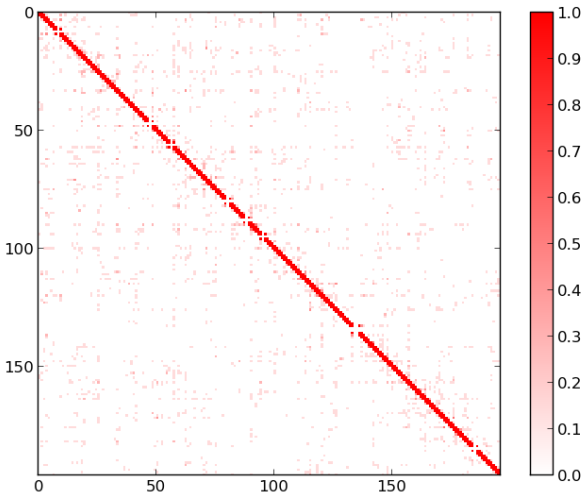


Figure 8: The contact map generated from a simulated fractal globule. Note the general featureless-ness of the map. Although the basic simulation can capture the overall statistics of chromatin, it cannot reproduce higher-level structures without more input data.

3.3 Reproducing High-level Structure by Using Hi-C to Inform Forces

We would like to go further. Since the overall statistics of chromatin can be generated with just a very simplistic physical model and without any input from Hi-C, we can hope to create simulated polymers that recover the higher-level structures of chromatin by incorporating Hi-C data into our simulation. This would allow us to generate simulated genome conformations that sample from the same distribution as the actual genome, which could provide insight into actual genomic structures rather than overall averages.

To accomplish this, we make some slight modifications to the polymer simulator. The inputs to the simulation are now an initial configuration as well as data assigning each monomer to be either part of an open or closed region of the polymer as well as a few pairs of segments that have elevated contact probabilities. The simulation program was modified to increase the attractive force between monomers in the same compartment by a small factor (1.1), and to increase force between monomers with elevated contact probabilities by a large factor (2). Further, the forces between these elevated contacts were calculated and included in the simulation regardless of whether the monomers in question were in the same voxel. This allows these elevated contacts to be relevant to the simulation; otherwise the elevated forces between these monomers would only be calculated if the monomers happened to come somewhat close together due to random fluctuation. Using these modifications, collapses of 100 different initial configurations were simulated and contact maps were generated from the final globular state.

3.4 Simulation Methods

A set of eight pairs of fragments on chromosome 14 that are expected from Hi-C data to be likely in contact with each other were selected to be simulated. We list these pairs in the table below (the ranges give the indices of the base pairs in the fragment):

Fragment One	Fragment Two
21086803-21087553	20892876-20896699
21086803-21087553	20865751-20867018
21086803-21087553	20808511-20816149
21086803-21087553	20931341-20938681
21086803-21087553	21154236-21168184
20931341-20938681	21154236-21168184
21154236-21168184	20824330-20827615
21154236-21168184	21090824-21092143

Note that these fragments all fall within the range 20603372-21350867.

Using the method described in section two, 100 open initial configurations were generated to represent this range of base pairs in chromosome 14, with each monomer representing 450 base pairs. The molecular dynamics system was used to simulate these initial configurations using the list of eight desired contacts and data obtained from Hi-C assigning each monomer in this range to the open or closed compartment of chromatin.

This input data can be summarized in a contact map, in which we use darker colors to indicate monomers between which the forces were amplified.

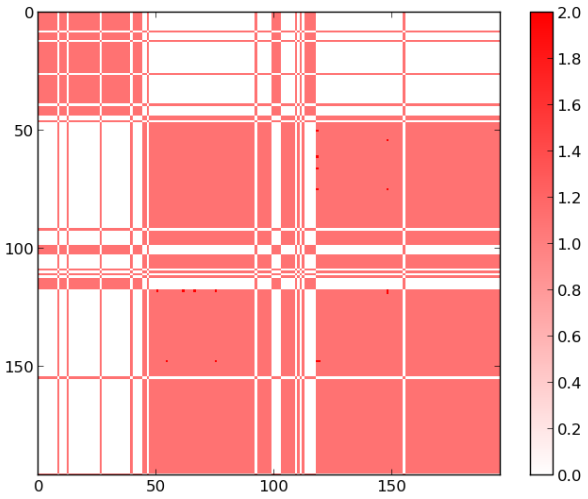


Figure 9: The simulator uses modified forces as illustrated in this force map.

Contact maps were calculated from the 100 globules obtained from the simulation procedure by simulating the action of a Hi-C experiment. Using a list of restriction sites of chromosome 14, each monomer in the simulated polymer was assigned a "fragment" into which it would be broken by our virtual Hi-C experiment based upon the fact that the simulated polymer represents base-pairs 20603372 to 21350867 in chromosome 14 and each monomer represents 450 base pairs.

Then for each pair of fragments, we calculated the probability that the pair was in contact in the 100 globules, by simply taking the fraction of the 100 globules for which the pair was in contact. Here two fragments A and B are considered to be in contact if any monomer of A comes within two bondlengths of any monomer of B. These probabilities were then organized into contact maps.

3.5 Results

As can be seen, although there is some visual evidence of structure in the contact map, it is hard to see in contrast to the very strong diagonal. In order to enhance contrast and readability on these contact maps, we treated them to two different filtering techniques. First is the *threshold* filter. This is a filter intended to bring out the low-probability features of the contact map. Since most of the entries of the contact map are very close to 0 (except on the diagonal, where they are generally very close to 1), it is hard to see much of the overall structure of the contact map. Thus we simply round every value greater than 0.1 down to 0.1. This has the effect of bringing the diagonal values down to the same level as the rest of the contact map, and so brings out the off-diagonal features. Using this filter we can check for the compartment structure we added in the input contact map.

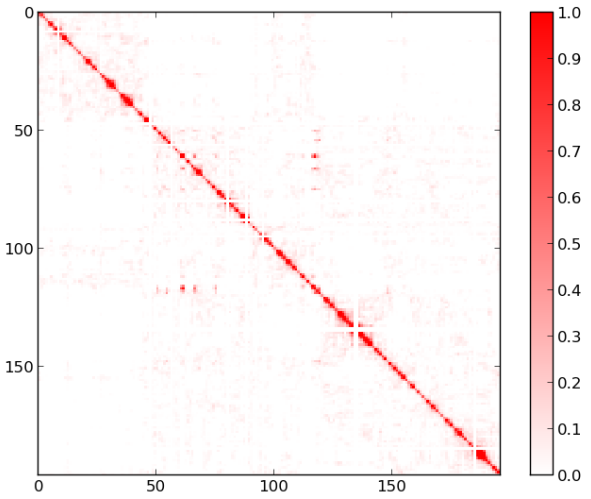


Figure 10: The raw contact map generated from simulating the input forces described in Figure 9. Now, in contrast to Figure 8, we see some higher-level structure in the contact map

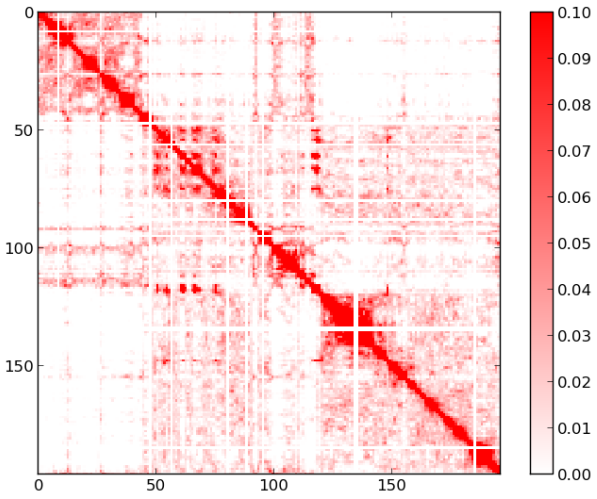


Figure 11: Here we filtered Figure 10 to make off-diagonal features more visible. This makes the higher-level structure much more apparent. We see a plaid pattern that represents the open and closed compartments.

The second filter is the *high probability* filter. This is a filter designed to bring out the features of the contact map that are elevated above the average, but may still be hard to see in contrast to the extremely high-probabilities on the diagonal. To achieve this filtering we choose a lower-bound of 0.08 and set all entries with values lower than the lower bound to 0 and all entries above the lower bound to 2, so that there is no contrast anymore but we see the entries with high probabilities. This allows us to check for the presence of the elevated probability contacts introduced in the input force map.

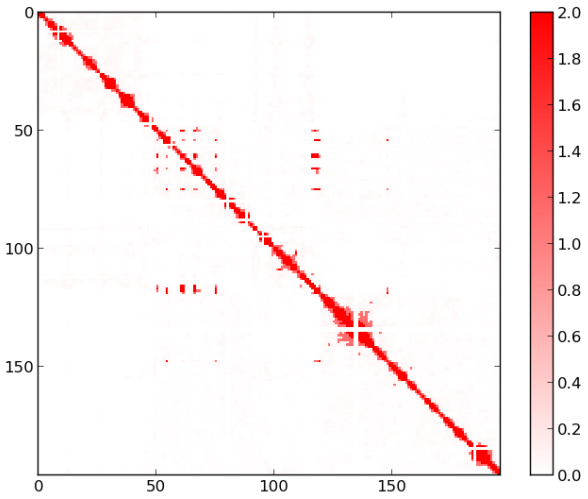


Figure 12: Here we filtered Figure 10 to make high-probability entries more visible. This destroys all contrast but allows us to see that there are some localized areas of high probability.

In order to see the correspondence between the output contact maps and the input contact maps, we show them here for side-by-side comparison. We have also applied the high-probability filter to the input force map. This results in a map that shows where the desired high-probability contacts were located.

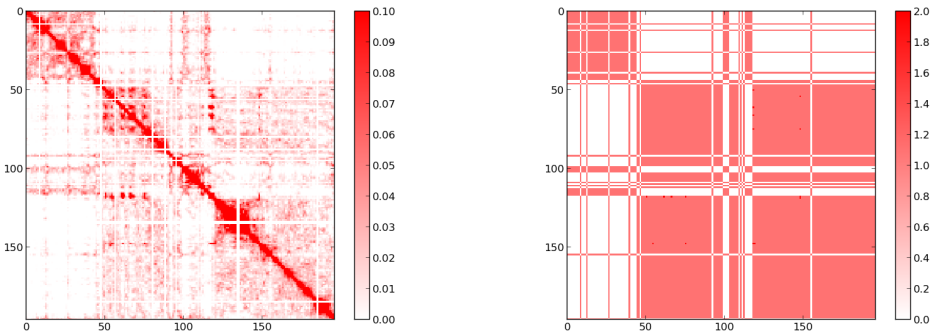


Figure 13: Threshold filter contact map with input force map.

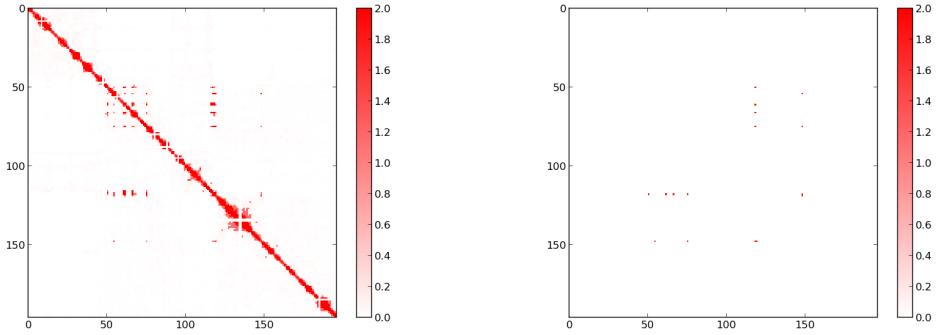


Figure 14: High probability filter applied to the contact map as well as the input map, showing where elevated contacts appeared in the simulation, and where they were supposed to appear. The diagonal is absent in the input map, but is also expected in the simulation.

One can see from visual inspection that in broad strokes this simulation data reproduces the desired contact matrix, which suggests that the simulated ensemble replicates at some of the properties of the real ensemble. One interesting note is that in the high-probability contact map we see more elevated contact probabilities than the input forces specified. This is likely due to a *transitive* effect of the input forces. For example, suppose that we wish to run a simulation in which monomer A has a high probability of contacting monomer B, and monomer B has a high probability of contacting monomer C. Then it is quite reasonable to suppose that monomer A will have a high probability of contacting monomer C. This is precisely the scenario we see in the extra high probability contacts that appear in the simulated contact map. This heuristic argument suggests that although we can get a good approximation by increasing forces between monomers that are more likely to be in contact, a more complicated modification of the simulation parameters is required in order to more closely replicate the genome ensemble.

4 Double Strand Breaks can Introduce Topological Mutations

In its healthy state, a chromosome in a eukaryotic cell consists of a single unbroken polymer chain of nucleotides. The sequence of these bases famously describes the heritable information required to reconstruct an organism. As is well-known, DNA actually has the structure of a double helix, with two intertwined chains of bases. These chains are complementary to each other in the sense that the sequence of nucleotides on one chain determines the sequence on the other.

However, the chromosome is often subjected to various types of damage that can alter this information, introducing mutations. Common sources of damage include ionizing agents, or stray chemicals in the cell. These influences can break one or both of the two chains of nitrogen bases that make up DNA, called respectively single-strand breaks (SSBs) and double-strand breaks (DSBs). When a SSB occurs, the broken chain can often be repaired using the whole chain as a scaffold. However, sometimes an SSB can become a DSB. For example, if a SSB occurs during the process of DNA replication, the replication procedure will result in a normal replicate and a replicate with a DSB. However they arise, DSBs are generally repaired either through a process called homologous recombination, or a process called non-homologous end-joining. Homologous recombination takes advantage of the fact that human cells possess duplicate copies of every chromosome. The duplicate chromosome is used as a template to align the broken ends of the DSB for repair. Non-homologous end-joining on the other hand directly finds and joins the two broken ends without need for a template chromosome [9].

DSBs are known to be a leading factor in carcinogenesis [10]. This is somewhat mysterious in that if the DNA repair machinery is operating properly no apparent damage actually occurs after a DSB. One possible (and indeed the common) explanation for this is that double strand breaks cause alterations to the DNA sequence through translocations or chromosomal rearrangement. However, in addition to the DNA sequence, there is another property of the chromosome that might be affected by a DSB: the topology. Normally the dynamics of a polymer such as DNA in space describe an isotopy, and so the topology and knot type of the chromosome are constant. If the chromosome experiences a DSB that is later repaired, it is possible that some other section of the chromosome will pass through the area of the break before the repair is completed. This would have no effect on the sequence whatsoever, but it would have an effect on the topology of the chromosome - in particular, it might introduce a knot. This is a possibility unique to the double strand break since other types of damage do not allow the chromatin polymer to undergo motions that do not describe an isotopy. As previously described, we know from Hi-C data that chromatin generally has the structure of a fractal globule, which is an unknotted conformation. Over time, fractal globules very slowly convert into equilibrium globules through the process of the loose ends of the polymer moving through the globule and forming knots. A double-strand break can essentially accelerate this process by creating additional loose ends in the middle of the globule that can create more knots.

Such a knot could have an effect on the dynamics of chromatin, which in turn could have significant effects on cell function. Thus here we introduce the idea of a *topological mutation* in DNA - a mutation that is an alteration not of the actual sequence of bases, but of the topology and dynamical properties of the genome.

One dynamical property of interest in chromatin is that of *local expandability*. Heuristically this means that although chromatin may adopt a very densely-packed conformation, if one were to exert some small force on any given localized segment of the polymer, it would readily separate from the rest of the structure without significantly disturbing the packing. Intuitively, this property goes hand-in-hand with unknottedness since one would expect a knot to introduce some constraint that would prevent this separation. Therefore, local expandability is an ideal example of a dynamical property of chromatin that might be altered by a topological mutation. Further, local expandability is an important property in that gene expression requires genes, which are local segments of chromatin, to become accessible to transcriptional machinery, and this may require some local unpacking of the chromatin.

In order to study what sort of effect such a break can have, and whether topological mutations occur, we can use the molecular dynamics program to simulate such a break and repair. This is done by introducing the concept of a "phantom monomer". A phantom monomer is just a monomer that does not interact with monomers other than its immediate neighbors in the linear chain of the polymer. In order to simulate a break in the polymer, we turn B consecutive monomers into phantom monomers. We call B the "break size". The polymer is simulated for some time with these B phantom monomers. After this, the phantom monomers are restored to non-phantom status. This process is somewhat delicate as phantom monomers may have strayed very close to other non-phantom monomers, resulting in an extremely high-energy configuration of the polymer when the phantom monomers are restored. To correct for this, the polymer is subjected to a gradual energy minimization procedure in order to relax it into a more stable configuration. We achieve this energy minimization with the same procedure used to relax the initial polymer condition into a stable configuration.

In precise terms if the kth monomer in our polymer is $A[k]$, then the force acting on a phantom monomer $A[n]$ is

$$F_{ph}(n) = F(n, n-1) + F(n, n-2) + \sum_{A[k] \text{ is phantom}, k \neq n \pm 1} F(n, k)$$

Where $F(n, k)$ is the force that would be exerted on monomer $A[n]$ by monomer $A[k]$ in the unmodified simulation. The $F(n, n \pm 1)$ terms are needed to ensure that the phantom monomers remain in locations such that when the phantom status is revoked the monomers are still one bondlength away from their neighbors. This ensures that any instability introduced by turning phantom monomers back to real monomers will be caused by close contacts. These type of instabilities are very local in the sense that a small perturbation of the conformation (i.e. pushing monomers that are too close slightly away from each other) of the polymer will remove the instability. In contrast, if phantom monomers were allowed to migrate away from their immediate neighbors, then when the phantom monomers are turned back into normal monomers it might be necessary to make a very large change to the polymer conformation in order to return it to a stable state. The summation term is used to prevent knotting and strand passing occurring in the phantom monomers themselves. We are only interested in the case of a section of "real" monomers passing through our segment of phantom monomers, so by causing the phantom monomers to interact with each other normally we avoid the scenario in which a segment of phantom monomers passes through itself.

4.1 Expandability Measurement

We can measure local expandability of a globule by literally measuring how resistant the globule is to having an arbitrary segment pulled out of it. That is, we simulate a given globule with the addition of a force acting on some small segment that pulls the segment away from the center of the globule. By taking the ratio of the maximum distance of a monomer from the center of the globule after the expansion force was applied to the maximum distance before the force was applied, we get a measure of how far the segment was pulled out of the polymer and so a measure of expandability. Specifically, let $A_0[0], \cdots, A_0[n-1]$ be the positions of the monomers in the globule before the expansion experiment, and let $A_1[0], \cdots, A_1[n-1]$ be the positions after. Apply a translation so that the center-of-mass of the globule before the expansion is at the origin. Then we define the expandability to be

$$\max_i\{|A_1[i]|\} / \max_i\{|A_0[i]|\}$$

Here the denominator, $\max_i\{|A_1[i]|\}$ will be the radius of the globule in its compacted state before the local expansion experiment. $\max_i\{|A_1[i]|\}$ is the total distance away from the center of the globule the segment that was being pulled upon reached.

Note that this definition is dependent on the magnitude of the force used to pull on the small segment, the length of time the force was applied for, and the specific segment the force was applied

on. We will be particularly interested in this last point, so that we will say that the expandability measured by pulling on a specific segment is the expandability of that segment.

In detail, to measure the expandability of a segment $A[m]$ to $A[m+k]$, one uses the following procedure:

1. Center the globule at the origin, save its coordinates as $A_0[0], \cdots, A_0[n-1]$.
2. Simulate the globule for some time frame T using the molecular dynamics program, with the modification that a constant force of magnitude F pointing radially away from the origin is applied to each monomer $A[m], A[m+1], \cdots, A[m+k]$.
3. Save the positions of the monomers after time T as $A_1[0], \cdots, A_1[n-1]$.
4. Calculate $R = \max\{|A_0[i]|\}$ and $E = \max\{A_1[i]\}$
5. Expandability is E/R.

Here T and F are explicitly parameters of the expandability measurement that should be supplied and kept constant throughout all comparable measurements.

4.2 Simulation Procedure

We first generated a library of 50 closed initial configurations of size 4096 using the methods described in chapter 1 (molecular dynamics). These fifty initial configurations were then simulated using the molecular dynamics system until they collapsed into a fractal globule state (160000 simulation time units), yielding a library of 50 fractal globules each of size 4096 monomers.

Now for each of these 50 globules, for each value B in $\{0, 30, 60, 90, 120\}$, we turn monomers $A[1000 - \frac{B}{2}]$ to $A[1000 + \frac{B}{2}]$ into phantom monomers and simulate for 2.5% of the total time taken for collapse (4000 simulation time units). This leaves us with $50 \times 5 = 250$ globules. The 50 of these for which $B = 0$ are "control globules" in that no phantom beads were used and so no topological alterations should have occurred.

Now for each of these 250 globules we perform two expandability measurements: one on monomers $A[990]$ to $A[1010]$ and another on monomers $A[3086]$ to $A[3106]$, both using $F = 0.0002$ simulation force units and $T = 800$ simulation time units. This results in $250 \times 2 = 500$ total measurements. We will call the expandability measured at monomers $A[990]$ to $A[1010]$ the "local expandability" (since it is local to the area in which a break was simulated), and the expandability measured at monomers $A[3086]$ to $A[3106]$ the "non-local expandability".

4.3 Results

Visual inspection of the data reveals that while simulating a break does not always have an effect, on average there is a marked decrease in expandability after simulating a break. Below we present some images of the results of expansion experiments. The left hand column shows control globules where no break was simulated and the right hand column shows globules where breaks were simulated. The two images in each row were generated from the same initial configuration. First we have images from local expansion experiments, where we see the average decrease in expandability.

Local Expansion Control

Local Expansion After Break and Repair

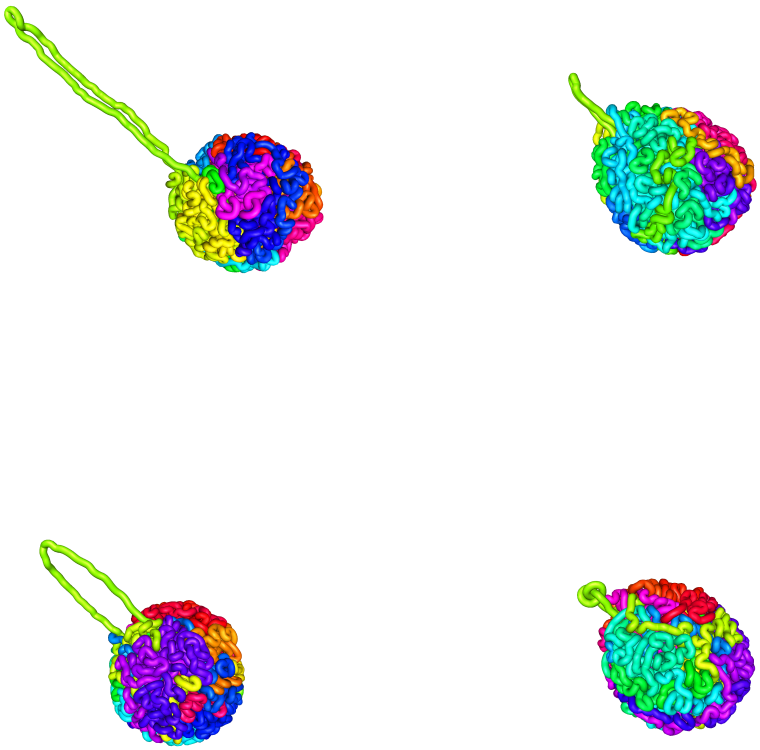


Figure 15: In most globules, there was noticeably less expansion after a simulated double-strand break

Local Expansion Control

Local Expansion After Break and Repair

Figure 16: In a few globules there was little observable effect of simulating a double-strand break

Local Expansion Control

Local Expansion After Break and Repair

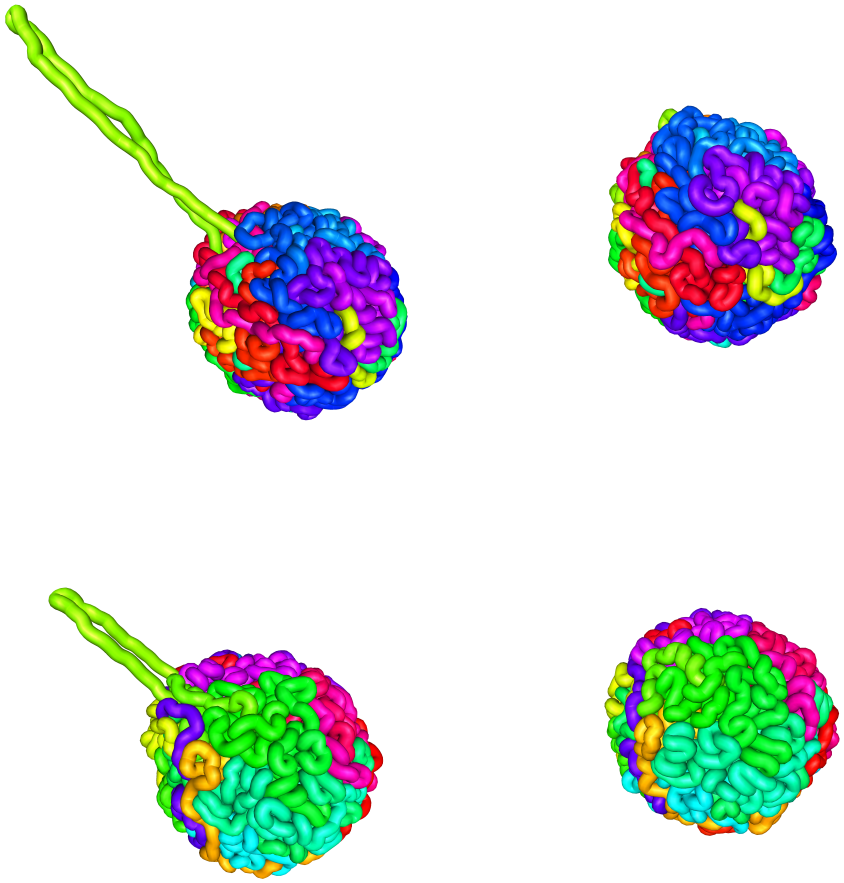


Figure 17: And in some globules, there was essentially no expansion after a simulated double-strand break

Next, we have images from non-local expansion experiment. Here there is relatively little difference in expandability between the control and the simulated break, suggesting that any obstruction to expandability created by a DSB is a "local" phenomenon.

Non-Local Expansion Control

Non-Local Expansion After Break and Repair

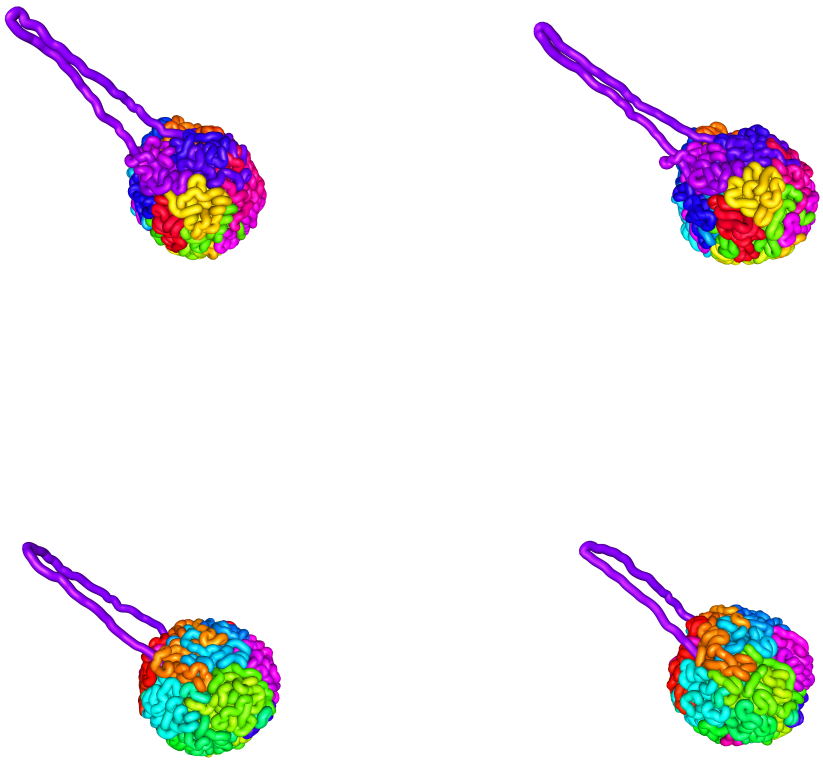


Figure 18: Here we see that the loss of expandability is a local phenomenon - measuring expansion at a different site from the simulated break shows little effect on the local expandability

This is summarized in the following charts, which show average local and non-local expandability measurements in comparison with control data.

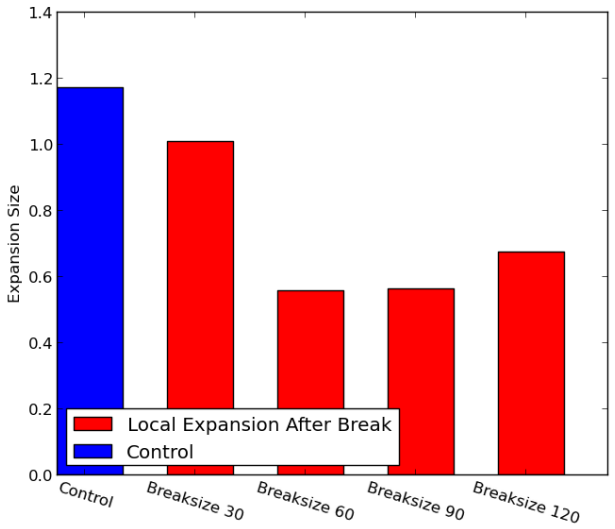


Figure 19: A plot of local expandability at the site of the simulated break for various breaksizes. The expandability is markedly less than the control globules which had no simulated break

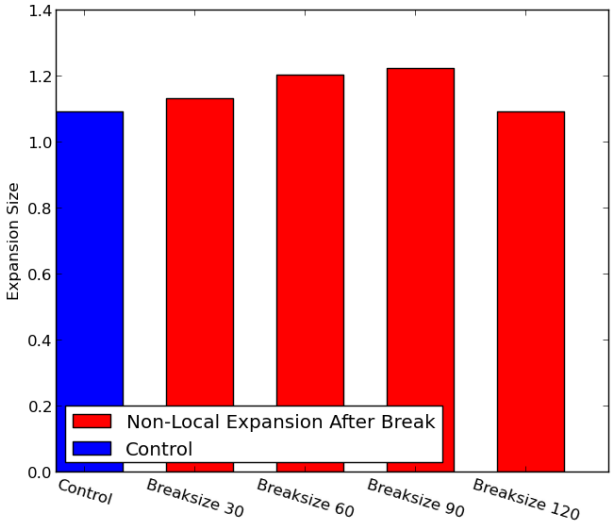


Figure 20: A plot of local expandability at a different site than the simulated break for various breaksizes. Note that there was no decrease in expandability after a simulated break, suggesting that the break has caused only a local perturbation.

Here we see a significant decrease in expandability in the area of the simulated break, while expandability in the non-break area is essentially unchanged.

We can also measure the rate at which this difference develops. That is, we can measure how much expansion has been demonstrated at many time points while exerting force on a segment to pull it out of the globule for an expandability measure rather than just at one. Here we perform this analysis for 19 time points over the course of the expansion experiment. and plot the ratio of the far and local expandability in the experiment to the control.

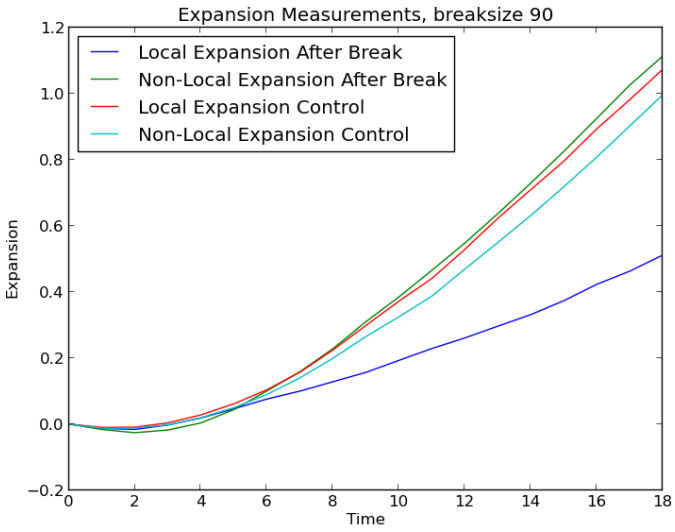


Figure 21: Amount of expansion plotted over time. Notice that the globules that have experienced a break expand significantly more slowly.

4.4 Local Knots

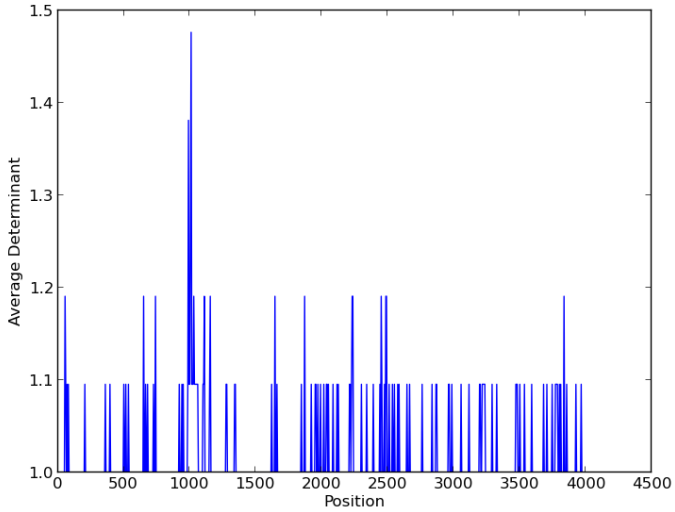
Intuitively, the proximate cause for this decreased expandability is some sort of knotting behavior caused by the breaking procedure. However, since the expandability was only altered in a neighborhood of the simulated break, this knotting phenomenon is in some sense "local". This locality clearly is a function not only of the topology of the polymer but also the geometry and we would like to have some way of quantifying the degree to which the polymer is locally knotted. We will adopt the "knot determinant", which is given by the value of the Alexander polynomial [6], A, of the knot at -1, as a proxy for knot complexity. The Alexander polynomial is a knot invariant that can be computed from an oriented diagram of a knot. It has the property that the Alexander polynomial of the connect-sum of two knots is the product of the polynomials for the original knots. Further, if $A(x)$ is the Alexander polynomial of a knot then $A \in \mathbb{Z}[x]$ and $A(-1) \geq 1$ with $A(-1) = 1$ for the unknot. Thus we see that if a knot is a sum of two other knots, its determinant is the product of the determinants of the summands so that the knot determinant gives some heuristic for the "complexity" of a knot.

Now we describe the procedure to compute local knottedness. Given any subchain of our polymer we consider the subchain as specifying a closed loop by simply drawing a straight line between the end points. Then by taking the determinant of this knot we have a way to calculate knot complexity of subchains. Let us write $D(i, j)$ for the determinant of the subchain given by the ith to jth monomers in our polymer. Then we define the local knottedness of the kth monomer to be

$$\frac{1}{a-b}\sum_{i=a}^{b} D(k-i, k+j)$$

where a and b are some chosen thresholds that reflect how sensitive this measure is to highly localized knots. Note that this definition is somewhat arbitrary - there is no canonical way to define $D(i, j)$ and so we make use of this heuristic instead. As some justification, we observe that in the case that the knots formed in evaluating $D(i, j)$ and $D(j, i)$ are not linked, then the entire polymer is a connect-sum of these two knots and so $D(i, j)D(j, i) = D$ where D is the determinant of the entire polymer. This is in fact the sort of scenario we are looking for in the case that the knot in the polymer is "localized" to the subchain between the ith and jth monomers.

We can use the program knotplot [12] to calculation knot determinants and visualize knots. Running this calculation with $a = 10$, $b = 50$ on the globules obtained after simulating a break we have the following chart of local knot complexity vs linear position on the polymer:



Here we see some bias towards local knotting around monomer 1000, which was the center of the simulated breaks.

In particular, we can visualize the knot localization in the globule:

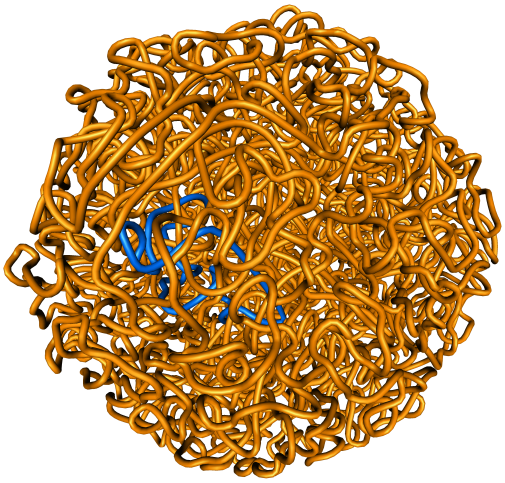


Figure 22: A visualization of knot localization after a simulated break: The area of the globule with average knot complexity above 1.2 is colored in blue.

This can be contrasted with the area in which we simulated a break:

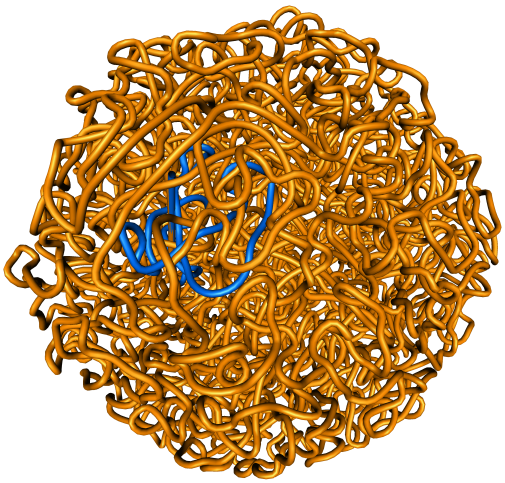


Figure 23: A visualization of the phantom beads: The segment of phantom beads is colored blue.

If we zoom in on these areas, we see that there is significant intersection.

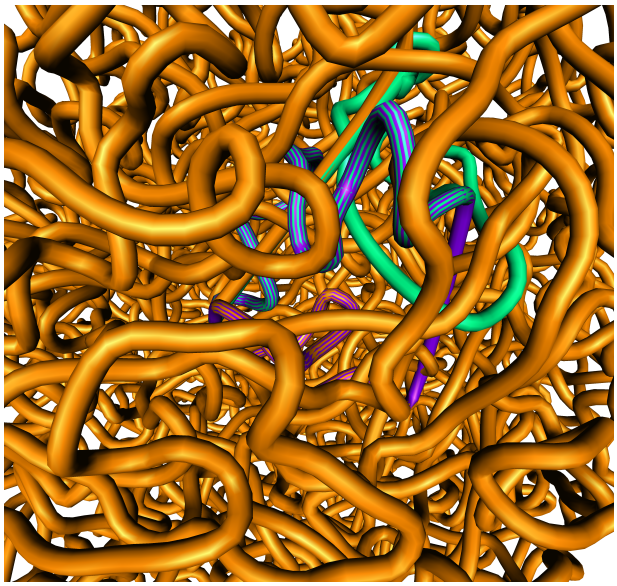


Figure 24: The intersection of the phantom beads and the area with high knot complexity is striped blue and purple.

In conclusion, these simulations give evidence for the introduction of significant alterations to the dynamics of polymers after introduction and repair of a break. This in turn suggests a new aspect to the damage caused by double-strand breaks in DNA. Although cellular repair machinery may join the broken ends and so repair the original DNA sequence without mutation, they may in the process cause a form of "topological mutation" in which a region of DNA becomes less accessible due to local knotting of the polymer.

References

[1] A. Yu. Grosberg, S.K. Nechaev, E.I. Shakhnovich. *The role of topological constraints in the kinetics of collapse of macromolecules.* J. Phys. France, Volume 49, Number 12, December 1988, Pages 2095-2100

[2] D. van der Spoel et. al. Gromacs User Manual, www.gromacs.org, 2005

[3] L. Greengard, V. Rokhlin. A Fast Algorithm for Particle Simulations. Journal of Computational Physics Volume 73, 1997

[4] E. Julliard-Tosel. Brun's Theorem: The Proof and Some Generalizations. Astronomie et Systemes Dynamiques, Institut de Mecanique Celeste, 2000

[5] E. Lieberman-Aiden, N. L. van Berkum, et al. Comprehensive mapping of long-range interactions reveals folding principles of the human genome. Science 326 (2009)

[6] J. Alexander. Topological Invariants of Knots and Links. Transactions of the American Mathematical Society. Volume 30, No. 2, 1928.

[7] J. Barnes and P. Hut. A Hierarchical $O(N \log N)$ force-calculation algorithm. Letters to Nature. Volume 324, 1986.

[8] J.A. van Meel, A. Arnold, D. Frenkel, S.F. Portegies Zwart, R.G. Belleman. *Harvesting graphics power for MD simulations.* arXiv:0709.3225

[9] J. Watson et al. Molecular Biology of the Gene, 5e. Benjamin Cummins, 2003.

[10] K. Khanna and S. Jackson. DNA Double Strand Breaks: Signaling, Repair and the Cancer Connection. Nature Genetics, Volume 27, 2001

[11] M. Harris. GPGPU: General-Purpose Computation on GPUs, Game Devloper's Conference, 2005

[12] R. Scharein. knotplot, www.knotplot.com

[13] S. Plimpton, Fast Parallel Algorithms for Short-Range Molecular Dynamics, J Comp Phys, 117, 1-19 (1995). lammps.sandia.gov