The Thermo-Mechanical Dynamics of DNA Self-Assembled Nanostructures

by

Vincent C. Mao

Department of Electrical and Computer Engineering
Duke University

Date:_____

Approved:

_____
Chris Dwyer, Advisor

_____
Anne Lazarides

_____
Alvin Lebeck

_____
David Smith

_____
Tomoyuki Yoshie

Dissertation submitted in partial fulfillment of
the requirements for the degree of Doctor of Philosophy in the Department of
Electrical and Computer Engineering in the Graduate School
of Duke University

2010

ABSTRACT

The Thermo-Mechanical Dynamics of DNA Self-Assembled Nanostructures

by

Vincent C. Mao

Department of Electrical and Computer Engineering
Duke University

Date:_____
Approved:

_____
Chris Dwyer, Advisor

_____
Anne Lazarides

_____
Alvin Lebeck

_____
David Smith

_____
Tomoyuki Yoshie

An abstract of a dissertation submitted in partial
fulfillment of the requirements for the degree
of Doctor of Philosophy in the Department of
Electrical and Computer Engineering in the Graduate School
of Duke University

2010

Abstract

The manufacturing of molecular-scale computing systems requires a scalable, reliable, and economic approach to create highly interconnected, dense arrays of devices. As a candidate substrate for nanoscale logic circuits, DNA self-assembled nanostructures have the potential to fulfill these requirements. However, a number of open challenges remain, including the scalability of DNA self-assembly, long-range signal propagation, and precise patterning of functionalized components. These challenges motivate the development of theory and experimental techniques to illuminate the connections among the physical, optical, and thermodynamic properties of DNA self-assembled nanostructures.

In this thesis, three tools are developed, validated, and applied to study the thermo-mechanical properties of DNA nanostructures: 1) a method to quantitatively measure the quality of DNA grid self-assembly, 2) a spectrofluorometer capable of capturing fluorescence and absorbance data under simultaneous multi-wavelength excitation, and 3) a Monte Carlo simulator that models the ensemble response of DNA nanostructures as simple harmonic oscillators.

The broad contributions of this dissertation are as follows: 1) insight into the thermo-mechanical properties of DNA grid nanostructures, and 2) a categorization of self-assembly defects and their impact on proposed logic circuits.

The results of the work presented in this dissertation show that: 1) the quality of self-assembly of DNA grid nanostructures can be quantitatively calculated to demonstrate the impact of changes in temperature or structure, 2) the optical absorbance of complex DNA nanostructures can be modeled to capture their thermo-mechanical properties (i.e., worst case within 10% of experimental melting temperatures and 70% of experimental thermodynamic parameters), 3) the structural resilience of DNA nanostructures can be quantifiably improved by chemical cross-linking with up to 60% retaining their original structure, and 4) DNA self-assembly introduces structural defects which create new fault models with respect to conventional technologies for logic circuits.

Dedication

For Leei and Ping

Contents

List of Tables

List of Figures

Acknowledgements

I owe a tremendous amount of thanks to my brother, Jeff, who kept me fed and made sure I got enough sleep, and my parents for their unwavering love and support throughout my life. I am incredibly grateful to my advisor, Dr. Chris Dwyer, for his endless patience, knowledge, and wisdom. He taught me to be fearless in making mistakes in the name of science, though it turned out to be an expensive piece of wisdom. I must thank Victor Orlikowski and David Becker for their immense technical support, without which my research would have been much more stressful. I would also like to thank Bogdan Romanescu, Viresh Thusu, and Ashwin Wagadarikar for entertaining conversations and discussions in and out of the office that ranged far beyond the realm of research. I am deeply indebted to Dr. Costi Pistol, Randy Evans, and Vladi Ivanov for their roles as mentors, for reminding me that there was life outside of research, and for their interventions to remind me of this fact. I would like to also thank the members of the Nanoscale Self-Assembled Systems Group and the Computer Architecture Group at Duke University for keeping my intellectual curiosity going, and the ECE Department staff for making sure administrative aspects of graduate life were the least of my worries. To Lam and Stacy, Eric, and Beth, thanks for looking out for me whenever possible. Lastly, I wish to thank defMo for helping me grow personally, socially, artistically, and academically. Without the incredible support of all these people, the journey would have been much less rewarding.

1. Introduction

In the coming decades, Moore's Law will become more difficult to maintain due to 1) the physical limits of silicon and 2) the economic cost of reducing dimensions of complementary metal oxide semiconductor (CMOS) transistors. Thus, research into massively parallel fabrication methods based on self-assembly has become increasingly relevant to computing technology. DNA self-assembly can be applied to areas in which traditional computing technologies are impractical to implement due to their current scale and operational requirements (i.e., biomolecular sensing [1]). The following presents a brief overview the DNA double helical structure, related work concerning the physical and thermodynamic properties of DNA, DNA as a manufacturing technology (i.e., the methods for fabricating devices using the DNA nanostructure as a scaffold), and the DNA self-assembled grid nanostructure composed of DNA tile motifs.

1.1 The DNA Double Helix

DNA is an organic polymer composed of a sequence of nucleotides supported by a sugar-phosphate backbone. Complementary oligonucleotide strands (i.e., adenine (A) binds to thymine (T) and guanine (G) binds to cytosine (C)) hybridize anti-parallel to each other and form the energetically favorable double helix structure [2]. Figure 1 shows the typical B-form double helical DNA structure [3].

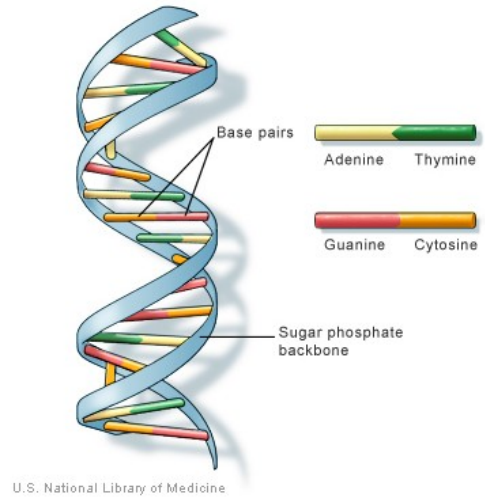


Figure 1: A simplified model of the double helical structure of DNA [3].

DNA is an ideal scaffold for nanodevices because of its chemical and physical characteristics. Its chemical properties enable the functionalization of organic and inorganic materials such as proteins and metals, potentially enabling a combination of these materials to interact. Its physical properties allow DNA to consistently self-assemble into its double helical conformation, requiring only a stable environment. The double helix form of DNA is more rigid than its single-stranded form, provided the strand is shorter than its persistence length (~53 nm or 155 base pairs [4]). This implies that materials functionalized on short DNA strands can also be precisely positioned on DNA. A combination of DNA strands can be designed to self-assemble into structures beyond the conventional double helical structure, such as the grid nanostructures in [5]. These characteristics make DNA self-assembled nanostructures an attractive alternative technology for circuit design and fabrication.

1.2 Physical Properties of DNA

Research concerning the physical properties of DNA focuses on characterizing the elastic properties of the double helix structure by extension [4, 6-11] or compression [12-13], along with a variety of nonlinear dynamics and statistical models for DNA concerning the mechanical opening of the helical structure [14]. While long dsDNA molecules (i.e., microns in length) can be consistently imaged using atomic force microscopy, studying and characterizing DNA strands that are shorter than the persistence length can be challenging. One recent approach attaches gold nanocrystals to the ends of short dsDNA strands to study the length variation of such strands using x-ray scattering interference techniques [15], presenting opportunities to further understand the physical behavior of dsDNA. The following is a brief description of two models that have been used to characterize the physical behavior of DNA.

The physical properties of DNA have been modeled using the free jointed chain model (FJC) and the wormlike chain model (WLC). The FJC model is a simple polymer model that is composed of a series of rigid rods of a fixed length and ignores monomer interactions. Thus, it only accurately captures the physical response of DNA in a low force regime (i.e., 10 - 50 femtonewtons). The WLC model, also referred to as the Kratky-Porod model, is used to describe the behavior of continuously flexible, isotropic rods that are connected together. This model matches well with experimental data over a much larger force range (i.e., pico- to nanonewton magnitudes) that causes DNA to

extend to up to twice its contour length. This model accurately matches predicted persistence lengths of DNA when subjected to torsional strain, such as those present in supercoiled structures [16]. The following presents a mathematical description of the WLC model.

Equation 1 is the approximate interpolation for applied force versus the extension of dsDNA for the WLC model [4].

$$\frac{FP_L}{k_BT} = \frac{z}{L} + \frac{1}{4\left(1-\frac{z}{L}\right)^2} - \frac{1}{4}$$

Equation 1

In the expression above, z is the DNA extension, L is the contour length, P_L is the persistence length of dsDNA, k_B is the Boltzmann constant, F is the force, and T is the temperature in Kelvin. Related work removes the problem of potentially multiple (non-unique) solutions, but is globally less accurate. Equation 2 presents an extension of the interpolation equation to include a polynomial correction term [17]:

$$F_3(z) = F_0(z) - \frac{k_BT}{P_L}\frac{3}{4}z^2 = \frac{k_BT}{4P_L}\left(\frac{1}{(1-z)^2} - 1 + 4z - 3z^2\right), 0 \le z < 1$$

Equation 2

In Equation 2, $F_0(z)$ is Equation 1 solved for the force term. The expression removes the non-uniqueness aspect of the original interpolation equation and is only dependent on the contour length and the persistence length. The force required to cause

an extension in the DNA molecule by a small amount (i.e., much less than the contour length) costs an equivalent amount of kinetic energy, *KE*:

$$\boldsymbol{KE = \frac{3k_B T \Delta x^2}{2R_0^2}}$$

Equation 3

The R_0 term is the mean squared distance of the contour length, equivalent to $\sqrt{2P_L L}$ for the WLC model for DNA. As temperature increases, the kinetic energy of the DNA increases, causing an increase in its contour length. Equation 4 takes the partial derivative of the kinetic energy with respect to the change in length of the molecule.

$$\boldsymbol{F = \frac{\delta KE}{\delta \Delta x} = \frac{3k_B T \Delta x}{2P_L L}}$$

Equation 4

Here again, the force is dependent on the temperature as well as the contour length of the DNA molecule and the extension of the strand. An increase in Δx or the temperature, T, increases the amount of force experienced by the structure. This concept is used to develop the relation between the force required to extend the length of DNA and the Brownian force in a dynamic environment in Chapter 3.

1.3 Thermodynamic Properties of DNA

Understanding the properties of DNA shorter than the persistence length is important to apply it as a scaffold for future nanoscale devices. Related work concerning the physical response of DNA in a stable environment has yielded information about its stiffness and its elastic behavior [11], and the thermodynamic

properties of DNA have been well-characterized [18-20] and extracted from van't Hoff plots using circular dichroism data in [21]. There are also a variety of programs that determine these properties based on first principles that are useful for predicting stability of oligonucleotides of varying lengths [22-24]. The thermodynamics of DNA can be used to describe how DNA is affected by the change in the environment such as temperature, pH, as well as its behavior in different solvents. DNA denatures as the temperature increases, causing an increase in the absorbance of DNA, or hyperchromicity, due to a reduction in nearest neighbor base pair interactions. This property enables the study of the DNA structure by monitoring the absorbance at 260 nm. The inflection point indicates dissociation of half of the DNA sample, and identifies the melting temperature of DNA. To identify the thermodynamic properties, a van't Hoff plot (i.e., dA/dT) can be used to find the local maximum of the plot (the melting temperature, T_M), and the full width half maximum (FWHM) of the transition which characterizes the transition enthalpy (ΔH) of the DNA sample. An example plot is shown in Figure 2 for a 13 base pair (bp) dsDNA strand with a melting temperature of 74°C and an enthalpy of 108 kcal/mol.

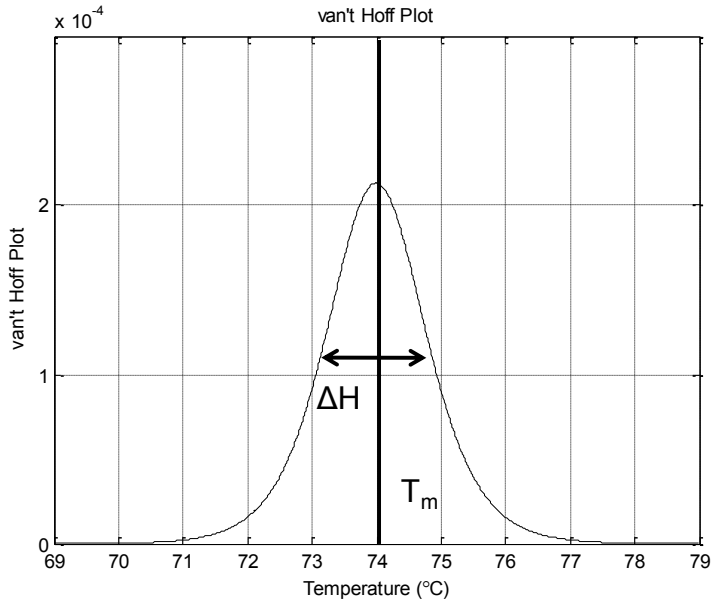


Figure 2: A van't Hoff plot of a 13bp strand with a 74°C melting temperature.

These thermodynamic properties are useful in determining other properties such as the free energy and the entropy of DNA using Gibb's free energy equation. Other examples of research on thermodynamic properties of DNA include the transition of the B-to-Z conformation of DNA [25-26] using differential scan calorimetry (DSC) to observe sharp increases in the melting curve profile as well as the effect of different intercalators, such as ethidium bromide, proflavin, or actinomycin, that inhibit this transition depending on the concentration of counter ions such as magnesium [27].

The body of work presented in the previous two sections has inspired a number of models [28-35] and simulators [22, 24, 36-39] developed for studying linear DNA in terms of both its physical response and thermodynamic properties. Understanding

these properties is an important part of the development of DNA as a manufacturing technology for future devices.

1.4 DNA as a Manufacturing Technology

As a manufacturing technology, DNA must be modified to self-assemble into structures that are more complex than the traditional double helical form, enabling the development of procedures for conjugation or functionalization of materials that will behave as active components in a logic circuit. This section presents background work concerning DNA as a template and introduces the tile motif structure as a composable unit for self-assembly of DNA grid nanostructures.

1.4.1 An Overview of DNA as a Template

DNA was first demonstrated as a scaffold for nanowire fabrication by depositing it across two gold electrodes, followed by deposition of silver ions and reduction using hydroquinone to form a DNA-templated silver nanowire[40]. Other metals have been successfully templated on DNA to form nanowires such as gold [41], copper [42], palladium [43], platinum [44-45], and cobalt [46]. DNA-templated metallization [47-50] and nanoparticle array patterning processes have also been developed [51-54].

DNA functionalized materials have applications in sensing, signal propagation, and logical output. Sensing applications have been demonstrated by using DNA to control the distances between metal nanoparticles and studying changes in their optical behavior [55]. DNA can also be used as a molecular sensor by using it as a surface for

deposition of an organic sample such as RNA and proteins [1, 56]. Signal propagation applications involve resonance energy transfer (RET), a phenomenon that takes advantage of the overlap between the emission wavelength range of one fluorescent molecule and the excitation wavelength range of another. A RET cascade is fabricated by functionalizing multiple molecules on DNA. One of the difficulties of effectively demonstrating this phenomenon is in finding a combination of fluorescent molecules that do not overlap multiple emission and excitation wavelength ranges to avoid ambiguity concerning the resulting output. Successful demonstrations of RET have been performed on DNA-based molecular wires composed of up to five different molecules [57-58]. Simple logic gate behavior applications have been demonstrated using the combination of complimentary DNA strands attached with fluorescent molecules [59-60] as well as using RET [61]. DNA has been demonstrated as a means for potential applications in switching as well [62].

The examples above describe DNA as a template for the precise positioning of components. However, to move towards the development of DNA-self-assembled nanotechnology, a more complex DNA structure capable of acting as a substrate for the patterning of organic and inorganic materials must be developed. The following section describes the DNA tile motif nanostructure.

1.4.2 The DNA Tile Motif

The tile motif is composed of nine single strand DNA (ssDNA) segments: one core, four shells, and four arms. These strands self-assemble into a tile motif using a controlled temperature annealing process. The tile motifs can self-assemble into complex grid structures facilitated by the "sticky ends" consisting of five bp ssDNA segments at the ends of each tile motif arm that are complementary to the base sequences of other tile motifs. The motifs can include modified core or shell ssDNA strands that include functionalization sites on the nanostructure. Figure 3 presents the basic tile motif, an example of a complex DNA nanostructure composed of sixteen tile motifs, and a functionalized DNA nanostructure [63].

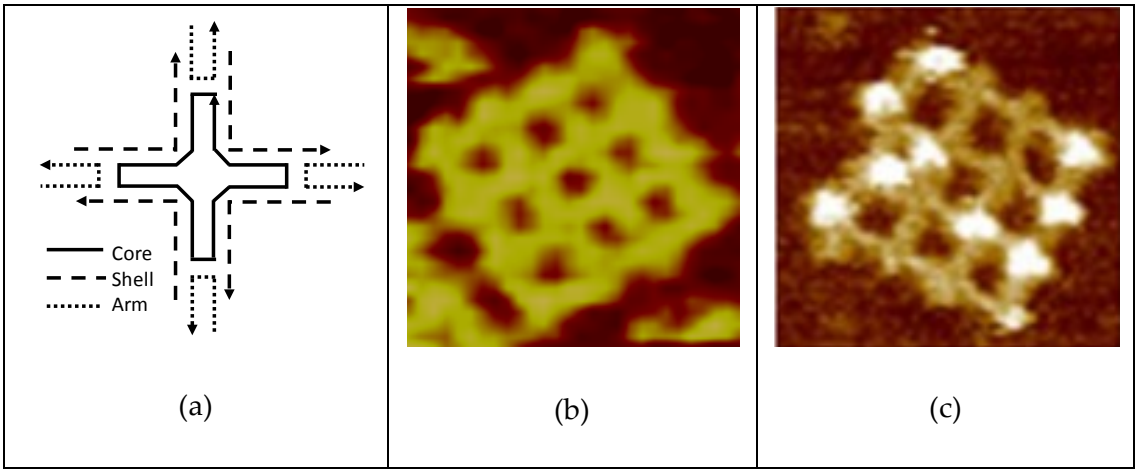


Figure 3: (a) A diagram of the DNA tile motif composed of nine individual DNA strands. (b) Two sixteen tile DNA nanostructures composed of tile motifs. (c) A sixteen DNA nanostructure with proteins functionalized in an "N" pattern [63].

In Figure 3a, the tile motif diagram indicates the number and type of strands that compose the DNA nanostructure. AFM images of both a 4 × 4 DNA nanostructure and

one that is functionalized with streptavidin protein to form the letter "N" are also presented in Figure 3b and Figure 3c, respectively, to demonstrate the addressability of the structure. The DNA tile motif is the basic composable element from which more complex DNA self-assembled nanostructures can be fabricated.

DNA self-assembled nanostructures have the potential to act as substrates for precise positioning of organic and inorganic materials. Grid nanostructures [64], triangles [65], stars [66], and the self-assembly of irregular patterns [67] are just a few examples of the many types of DNA nanostructures that have been synthesized. Metal nanoparticles [68] and proteins [69] have been individually attached, or functionalized, onto DNA nanostructures. Demonstrations using DNA to aid in the assembly of carbon nanotubes into different configurations has also been reported [70].

Although DNA self-assembled nanostructures are an attractive candidate as a substrate for future DNA-based devices, an understanding of the thermo-mechanical properties of these nanostructures must be established for rigorous design and fabrication. The potential of DNA as a manufacturing technology in the areas of engineering, medicine, biology, and computing technologies is possible because of several characteristic properties: 1) the dimensions (i.e., on the order of nanometers), 2) base pair complementarity that enables precise assembly and positioning of functionalized elements, and 3) the scalability in terms of manufacturing volume (i.e., 10^{19} structures per mL). These properties make DNA self-assembly an attractive

candidate for the fabrication of computational devices that are small, numerous, and interconnected.

This thesis focuses on the characterization of the thermo-mechanical dynamics (i.e., the interaction of physical, thermodynamic, and optical absorbance properties) of DNA grid nanostructures. Characterizing these properties is essential for making decisions about DNA-based device design and fabrication. To this end, three diagnostic and characterization tools capable of quantitatively determining the physical response, optical absorbance, and thermodynamic properties of DNA are presented: 1) an image analysis metric that evaluates the quality of the self-assembly process, 2) a fully customizable dual beam instrument that can select two distinct wavelengths for excitation of fluorescent samples over a range of temperatures, and 3) a simulator capable of modeling the transient response of a DNA nanostructure through a temperature ramp.

1.5 Research Thesis

Accurate models for the thermo-mechanical dynamics of DNA grid nanostructures enable the design and fabrication of molecular-scale computing systems.

This dissertation shows the following claims to support the thesis statement:

1. **A quality of self-assembly (QSA) metric enables the quantitative evaluation of DNA self-assembly techniques.** The metric calculates the number of "well-formed" structures present in an atomic force microscope (AFM) image. The

QSA metric is demonstrated to quantitatively evaluate the impact of two denaturation processes on a sample of 4 × 4 tile DNA nanostructures (i.e., by a temperature ramp and immersion in increasing volumes of ethanol).

2. **A Monte Carlo simulator, the Diffusive Nucleic Acid Spring TRajectory Analysis of Independent Nanostructures (DNA-STRAIN), enables predictive modeling of complex DNA structure and interactions.** The simulator is shown to match experimental data for DNA structures of increasing complexity (i.e., from dsDNA to 2 × 4 grid structures). Diffusional, optical, and thermodynamic properties have been shown to be within 1%, 15%, and 70% of the experimental data for the worst cases, respectively.
3. **DNA nanostructures can be chemically "hardened" to retain their structure, which enables functional systems based on DNA self-assembly.** The cross-linking process can be monitored to guide the degree of "hardening" in a DNA nanostructure, and the process has been shown to stabilize the DNA grid against thermal and chemical denaturation. This enables downstream functionalization in solvents other than water.
4. **Categorization of defect models enables the design of defect-tolerant systems.** The defects identified and compiled from AFM images of DNA grid nanostructures are categorized to show how these known and anticipated defects

on a DNA grid nanostructure can impact logic. This study identifies new classes of defects not seen in CMOS technologies.

1.6 Dissertation Outline

Chapter 2 covers the derivation of the QSA metric, its validation, and its implementation using the NIH ImageJ image processing software. Chapter 3 presents the derivation and validation of the DNA-STRAIN simulator, as well as the analytical tools and models used to characterize and validate the physical behavior of DNA. The application of the DNA-STRAIN simulator to several models for DNA nanostructures are also presented and discussed. Chapter 4 presents the design and validation of the DBS instrument that has two monochromators for independent selection of wavelengths. Chapter 5 presents a study in the modification of DNA nanostructures using an intercalator. The characterization tools presented in the previous chapters (i.e., the QSA metric, the DNA-STRAIN simulator, and the DBS) are used to analyze the data and compare the results to a simulated model of the cross-linked DNA nanostructure. Chapter 6 describes the defects that can occur during self-assembly and discusses their impact on logic circuits functionalized on DNA nanostructures.

2. Determining the Quality of Self-Assembly

The fabrication of DNA self-assembled nanostructures can be qualitatively analyzed by AFM images. However, a quantitative approach is applicable when assessing the impact of modifications on the self-assembly process. The Quality of Self-Assembly (QSA) metric quantitatively determines the yield of a sample of DNA grids in AFM images. The metric is determined by first calculating the ratio of the number of square cavities formed by the DNA grid structure to the total surface area that the DNA covers in the image. The resulting value is then normalized relative to an ideal case consisting of perfectly formed DNA grids that can cover the AFM image dimensions. The QSA metric is useful for comparison of DNA grid nanostructures that have been modified by functionalization or intercalation (e.g., fluorescent particles, proteins or inorganic materials), exposed to a hostile environment (e.g., changes in pH, temperature or immersion in caustic solvent), or a combination of the two processes. This chapter briefly presents related work, followed by the derivation of the QSA metric. The validation methodology is then described, followed by a demonstration of its utility as a diagnostic tool for DNA grid nanostructures in different environments.

2.1 Methodology Overview

Algorithms have been developed to analyze AFM images of organic materials, such as rotavirus molecules [71] and E. coli DNA [72], deposited on atomically flat surfaces. Studies concerning the AFM tip interaction with samples that have features

on the order of the thickness of the AFM tip [73], as well as approaches for using algorithms to aid in partially automated image processing for different types of DNA configurations (i.e., plasmids and DNA fragments [74-75]), have also been conducted in an effort to improve the image quality and subsequent analysis. Figure 4 illustrates an image processing procedure of an AFM image of a strand of DNA [72].

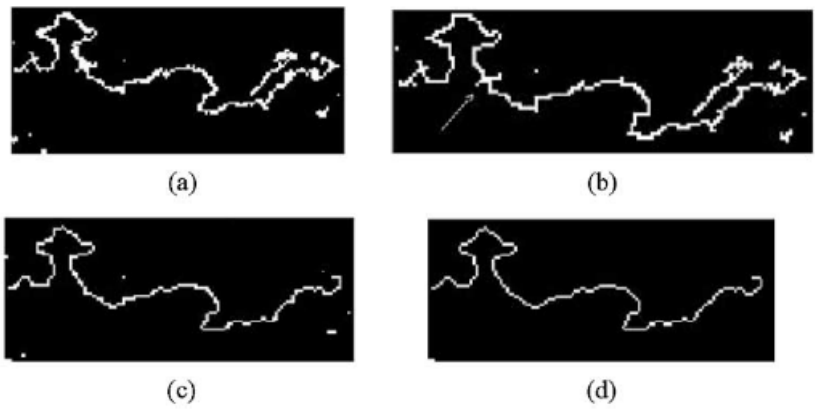


Figure 4: An (a) AFM image of a strand of DNA after threshold filtering, (b) after applying the thinning process to the image, (c) removal of extraneous shapes and (d) the final image.

First, the AFM image is processed to remove background noise using a filter. Then, the resulting image is filtered again to remove the extraneous shapes from the background. Finally, the resulting image can be processed for measurements such as DNA length and the number of particles in the image. Such information is useful for profiling a DNA sample.

The QSA metric uses a similar approach for AFM image processing, and is derived using knowledge of the dimensions of the B-form of DNA and the number of

bases that compose the DNA tile motif. AFM images of the 4×4 DNA grid nanostructures were obtained using an Agilent 5500 PicoScan AFM. The QSA metric calculation is implemented in ImageJ, an open source, Java-based image processing program inspired by a similar program, NIH Image, developed at the U.S. National Institutes of Health (NIH) [76]. The accuracy of the metric is then demonstrated by applying it to a set of training images containing well-formed DNA nanostructures. Initially, the entire metric calculation is manually executed for an ideal image to determine calibration factors required for general application to AFM images. Automation of the process is applied after testing and calibration using a set of macros. The utility of the QSA as a diagnostic tool is then demonstrated by applying it to a set of AFM images of DNA nanostructures heated to temperatures of 35°C and 55°C.

2.2 Derivation of the QSA Metric

The dimensions of the DNA tile motif are derived from knowledge of the dimensions for B-form dsDNA and the number of bases in the tile motif [63]. However, the measured diameter of DNA can be larger depending on the variation in the AFM tip width. A correction factor is included in the metric to increase the accuracy of the DNA surface area measurement in the AFM image. Figure 5 illustrates the dimensions of a DNA tile motif, a 4×4 DNA grid nanostructure, and the nominal diameter of a standard silicon nitride AFM tip.

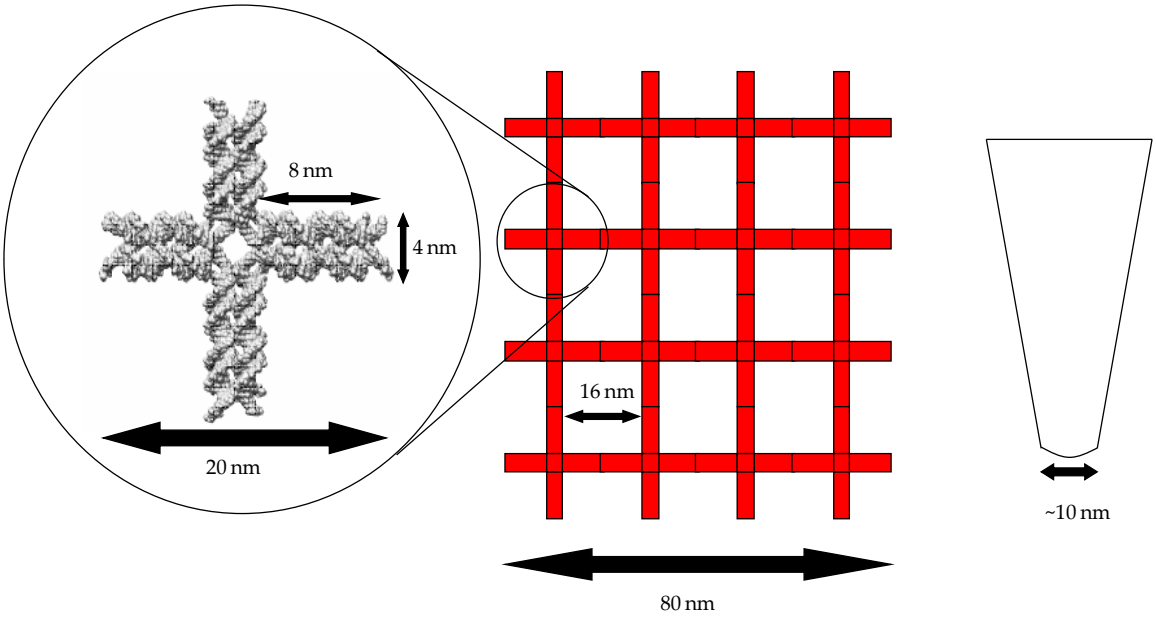


Figure 5: (a) The dimensions of a DNA tile motif, (b) 4 x 4 DNA nanostructure and (c) the AFM tip dimensions (figure not to scale).

The length of the DNA tile motif is 20 nm with each double helix having a width of 2 nm and each arm having a length of 8 nm. The QSA metric values for calculated surface areas of DNA grid nanostructures and the number of cavities per grid can be scaled for the dimensions of the sample (e.g., 2×4, 4×4, 8×8). The nine cavities in a 4×4 DNA nanostructure are approximately 16 nm on an edge, resulting in a cavity area of approximately 256 nm^2. The initial metric, M, is expressed in terms of cavities/μm^2 in Equation 5.

$$M = \frac{C_{NM}}{A} \times 10^6 \frac{nm^2}{\mu m^2}$$

Equation 5

In the expression, $C_{NM} = (N \times M) - (N + M - 1)$, calculating the number of cavities in an $N \times M$ DNA grid, and $A = A_T \times N \times M$, calculating the surface area of the DNA grid nanostructure. A_T is 144 nm^2, the surface area of a DNA tile motif. The metric value is normalized to the maximum number of possible DNA grids that fit in the space of a standard AFM image (e.g., 2×2 μm^2) as a periodic array, calculated to be 3906 cavities/μm^2 for a 4×4 DNA nanostructure. This normalization number changes depending on the dimensions of the nanostructure in the sample. The QSA metric is the normalized percentage metric in Equation 6.

$$QSA = \frac{M}{3906}$$

Equation 6

The extent of "well-formed" DNA grid nanostructures deposited on the mica surface is captured in this metric. The QSA metric ranges from 0, indicating that none of the structures in the AFM image are part of a DNA grid nanostructure, to 100, indicating that all structures deposited on the surface are part of the DNA grid.

2.2.1 Process Overview

It is important to manually determine the threshold range in order to determine the correction factor for calculations because the average diameter of an AFM tip is approximately 10 nm, as illustrated in Figure 5(c). The correction factor is applied to the metric to account for a variety of phenomena that can occur during the imaging process. Blunt AFM tips result in images with increased DNA widths and smaller, shallower

cavities, making it more difficult to determine the threshold required to distinguish the DNA from the surface. Images with large aggregates of DNA result in the creation of artificial cavities as image processing includes removal of the aggregates that are above the threshold limit, creating an outline of the aggregate itself. If the aggregate diameter is small enough, it can be counted as a cavity during image processing. In some cases, detection of the DNA at the surface is incomplete, and the software does not consider this a cavity because it is not completely closed. Conversely, several adjacent grid structures can touch to form an artificial cavity due to a high concentration of DNA grids coating the surface. Background noise due to excessive tip drive during image acquisition can also increase the DNA surface area measurement. The metric calculation can result in fewer cavities and a larger DNA surface area measurement due to the effects described, resulting in decreased cavity count by the macro and a less accurate metric. Increasing the resolution of the image is a solution to some of these effects, but requires more memory, a sharper tip, and is more time consuming. However, it can make a difference in the presence or absence of a gap in or between DNA grid structures, affecting the cavity count.

The AFM image is processed using an enhance contrast and smoothing feature in ImageJ to improve DNA detection in terms of the accuracy of the width measurements and cavity counts needed to calculate the metric. The appropriate parameters for the DNA nanostructures in the image are then determined by selecting a complete DNA

nanostructure in the image for parameter extraction. The threshold range for DNA detection and the range cavity areas are determined manually. Then, the correction factor for the DNA surface area is determined by running the initial macro for this nanostructure. Once all three parameters are known, they are applied in the metric calculation macro. Details on the calculation of the correction factor as well as using the QSA metric in the ImageJ software are described and provided in Appendix A.

2.3 QSA Validation

The macros for metric calculation are validated using two 1×1 μm^2 AFM images which were quartered. The image dimensions are scaled by a factor of five for improved accuracy by sub-pixel area analysis. The software for counting the cavities is verified by counting DNA grid nanostructure cavities by hand in each section of the image. These numbers are compared with the cavity counter in the ImageJ software to determine whether the cavity counter in the software can be used for the QSA metric. Table 1 displays the resulting cavity count comparison to the manual count, as well as the Pearson product-moment correlation (PPMC) between the numbers in the two AFM images.

Table 1: Cavity counts for two AFM images comparing cavity counts by hand to the cavity counts determined by the macro.

	Image 1			Image 2		
	Manual	**Macro**	**% Error**	**Manual**	**Macro**	**% Error**
Section 1	90	103	14.44%	56	58	3.57%
Section 2	83	93	12.05%	52	58	11.54%
Section 3	90	77	-14.44%	47	43	-8.51%
Section 4	120	120	0.00%	40	46	15.00%
RMS	96.81	99.48	2.76%	49.11	53.97	9.90%
PPMC	0.77			0.97		

The ImageJ cavity counter is verified to perform adequately for the determination of the number of cavities in an image within 15% error per section and within 10% for full image. The correlation coefficient is a statistical measure of the correlation between two variables, in this case the cavities counted manually and those counted by the macro, and ranges between -1, indicating no correlation, and 1, indicating high correlation. The results show that implementing the QSA calculations using the cavity counter in the software gives reasonable cavity counts when compaired to counting manually.

Two sets of experimental data were processed to 1) validate the QSA metric macro performance, and 2) demonstrate the utility of the macro implementation of the QSA metric. The first set of data was from a sample of 4×4 DNA grids that were immersed in steadily increasing percent volumes of ethanol to denature the structures and presumably decrease the metric value. The QSA metric was manually applied to a

set of five AFM images, one for each sample. The macro for the QSA metric was applied to the entire AFM image to compare the accuracy of the macro to the manual calculation, as presented in Figure 6.

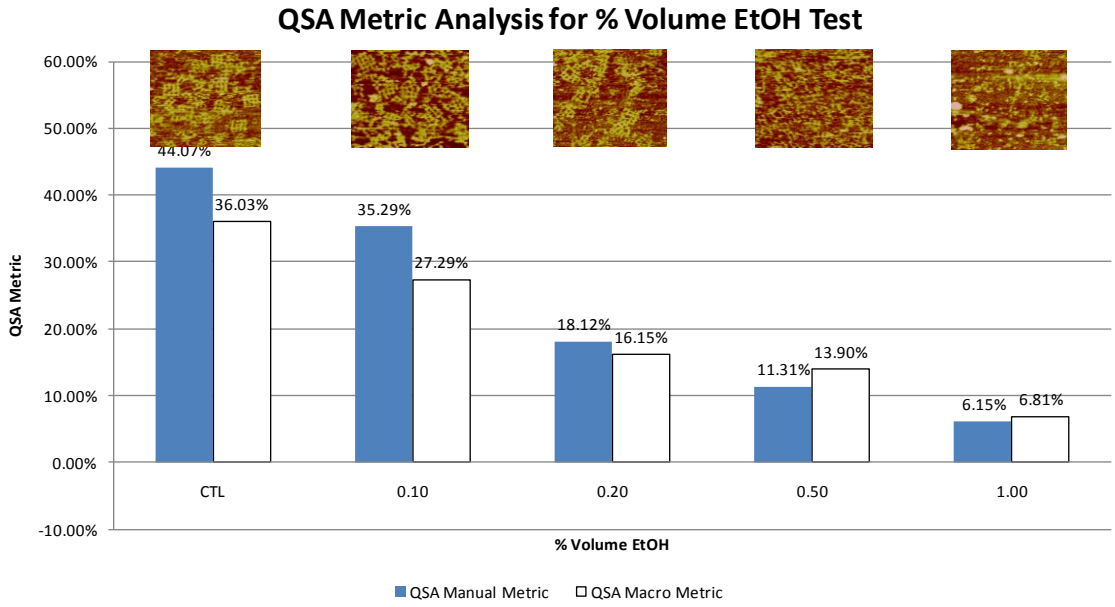


Figure 6: Bar graph for metrics from the best section of each image (blue) compared to the metrics resulting from the entire image using the macro (white).

The results indicate that the DNA structures can tolerate ethanol concentrations of up to 20% without significant denaturation. The DNA structure visibly degrades and aggregates significantly at 50% volume ethanol content in the solvent. These results agree well with a similar study concerning dsDNA immersed in increasing concentrations of ethanol [77]. The results calculated from the macro for the QSA metric agree to within 10% of the results from the manual calculation, within the 15% range of error from the cavity count macro as previously shown using the PPMC coefficient. Thus, the discrepancy between the manual and the macro QSA metric calculation is

demonstrated to be dependent on the cavity count. The QSA macro is demonstrated to give results within 15% of those expected with the manual calculation.

To make the entire procedure more accessible to users, two macros were created: 1) a macro for determining correction factor for the specific tip used for the image and 2) a macro for applying the correction factor to the image itself using the QSA metric. The first macro scales the image dimensions and sets the cavity area range to 30-300 nm^2 for more consistent measurements over a larger set of images to compensate for varying image resolution as well as to account for distortion of the cavity caused by orientation of the grid deposition. The second macro executes the calculation of the QSA metric using the correction factor from the first macro. The second set of experimental data used the ImageJ macros to automate the QSA metric calculation as a demonstration of the utility of the QSA metric.

The QSA macro is applied to a set of images of 4×4 DNA grid nanostructures. A set of three AFM images of the samples are obtained for 23°C, 35°C, and 55°C. Figure 7 presents the results of the QSA metric analysis.

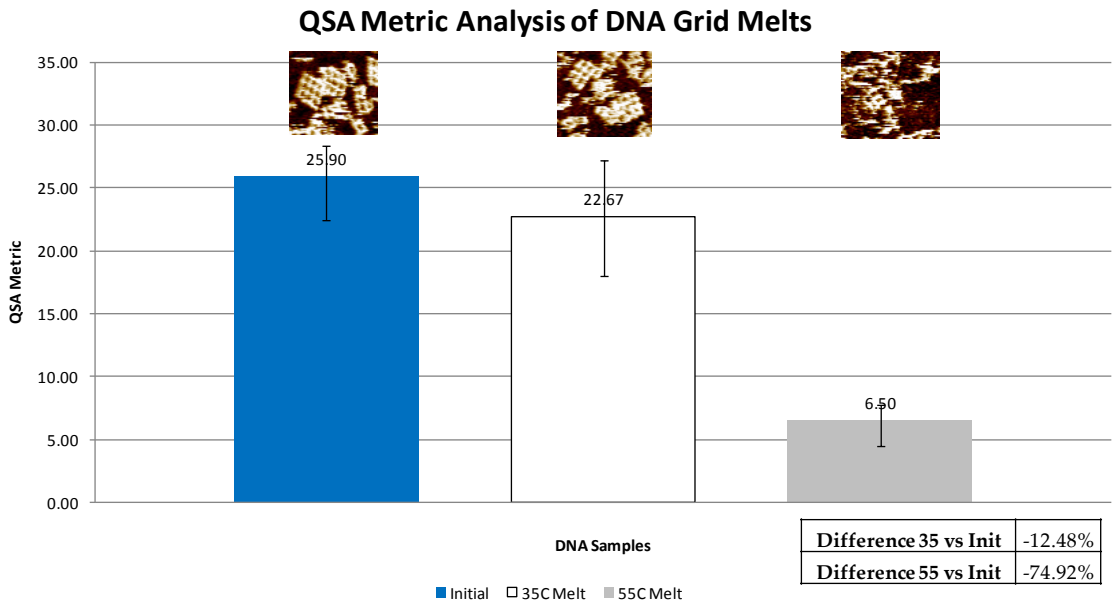


Figure 7: Metric analysis of DNA melts at 23°C, 35°C, and 55°C for functionalized DNA grid nanostructures.

The QSA metric decreases as the nanostructures are subjected to higher temperatures, indicating that the grids are still mostly well-formed in the first two temperatures, but significantly denatures in the range between the second and third DNA samples, demonstrating its utility in determining the quality of self-assembly of DNA grid nanostructures.

2.4 Limitations of the QSA Analysis Metric

The primary limitation of the metric is its dependence on image quality. The errors in the macro can originate from a variety of image-related factors such as extreme distortion of the grid nanostructures or counting artificially created cavities from adjacent nanostructures. The metric relies on initial calibration (i.e., a correction factor) for each set of images taken with an AFM tip due to variation of the AFM tip radius

from image to image. It is assumed that the AFM tip is consistent in diameter throughout the entire image acquisition process, that there is minimal noise, and that all material detected within the high and low thresholds is deposited DNA. Images that do not fulfill these criteria cannot be analyzed using the QSA metric with a significant degree of confidence, as any one of these criteria can detrimentally affect the calculation.

In conclusion, the QSA metric quantitatively determines the quality of self-assembly for a sample of DNA grid nanostructures. The theory for the metric was derived from dimensions of the B-form of dsDNA structures and the dimensions of AFM images of DNA grid nanostructures. The metric was validated using AFM images of samples of DNA nanostructures immersed in increasing volumes of ethanol to demonstrate the accuracy of the macros to automate the calculation to be within 15% of the manual calculation. The QSA metric macros were then demonstrated to quantitatively indicate the detrimental effects of increasing temperatures on a sample of DNA grid nanostructures, indicating that the denaturation process for these nanostructures began after 35°C. The denaturation of DNA self-assembled nanostructures is studied further in the following chapters.

3. Modeling the Physical Properties of DNA Nanostructures

This chapter presents the Diffusive Nucleic Acid Spring TRajectory Analysis of Independent Nanostructures (DNA-STRAIN) simulator that captures the physical, thermodynamic, and optical absorbance properties of double-stranded DNA (dsDNA) as it becomes denatured dsDNA assuming a unimolecular process. The simulator uses simple harmonic oscillators to model the elastic properties dsDNA with random force vectors applied to individual masses. The resulting force vectors calculated from the sum of the random forces and the restoring force of the spring determine the new coordinate positions of the masses. The change in length that results from the new positions determines the absorbance value of the dsDNA model. It is the relation between the change in length of the spring and the absorbance value that is novel in the DNA-STRAIN simulator. The relation enables the connection of the elastic response of the spring model to the temperature as well as to the absorbance value. A hierarchical approach is applied to develop a 3D spring network model of DNA grid nanostructures to predict their thermo-mechanical properites.

The simulator generates trajectories of mass nodes connected by a network of springs, which characterize dsDNA. The DNA-STRAIN simulator is validated first for a simple bead-spring model, then a 3D spring network that represents the dsDNA structure, using the elastic response of dsDNA as well as the thermo-mechanical properties described in Chapter 1. A DNA tile motif model composed of these structures is simulated and compared to experimental data following the validation of

the 3D dsDNA spring models. A set of theories are introduced to capture the thermodynamic interactions of these models in the tile configuration. Following the validation, the 2×2 and 2×4 tile DNA nanostructures that are composed of the tile motif model are simulated and compared to experimental data to demonstrate its accuracy. While it is used to study DNA grid structures here, the simulator can be potentially used to study DNA nanostructures such as those created in [67], as well as interactions with other biomolecules [1, 13]. The DNA-STRAIN simulator can be useful in predicting the behavior of DNA grid nanostructures in different environments such as future fabrication processes for DNA-based nanodevices.

The elastic and optical absorbance properties characterize the physical response of DNA to its environment. The absorbance properties are related to the thermodynamic properties of DNA using a van't Hoff plot to show the rate of change of the absorbance relative to the change in temperature. The relationship among the physical, optical, and thermodynamic properties of DNA is illustrated in Figure 8 using the experimentally determined properties of a 13 base pair dsDNA strand from [18].

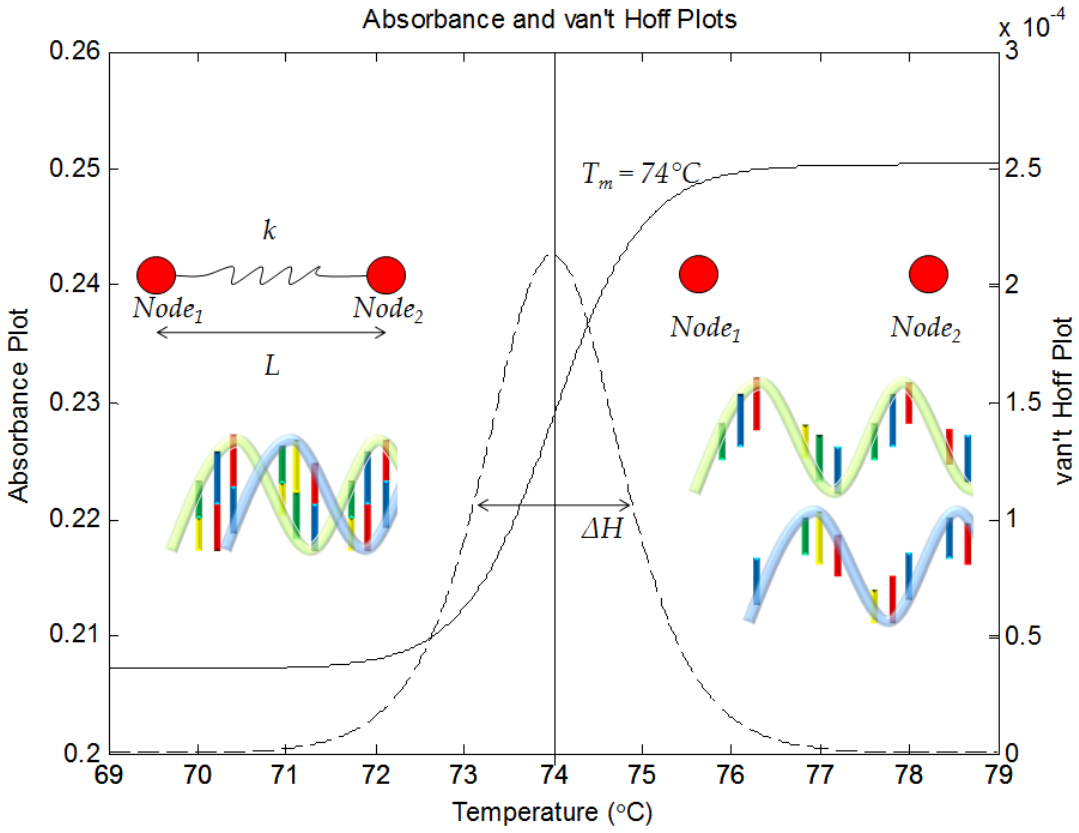


Figure 8: The relation among the physical, optical, and thermodynamic properties of a 13 base pair DNA strand.

On the left side of Figure 8, the elastic properties of DNA of a contour length, L, are simplified to that of a Hookian spring with spring constant, k, and mass nodes, $Node_1$ and $Node_2$, at either end at a stable temperature below the melting point, T_m. The elastic properties of the spring prevent the nodes from drifting apart due to Brownian forces applied to each mass. On the right side of Figure 8, the stiffness of the spring gracefully decreases as the temperature increases beyond the point that DNA can maintain its double helical structure. As a result, the mass nodes drift apart because the spring no longer applies any restoring force to the mass nodes.

Figure 8 also shows the dependence of the optical absorbance properties on the change in the elastic properties during the transition from dsDNA to ssDNA. As the structure melts, the absorbance (solid line) increases as DNA becomes less hypochromic (i.e., it absorbs more light at 260 nm). The peak of the van't Hoff plot (dotted line) is centered on the melting temperature, T_m. The full width half maximum (FWHM) of the peak relates the change in the dissociation rate constant to the change in standard enthalpy, ΔH, dependent on the number of base pairs in the structure. Thus, the larger the number of base pairs in the dsDNA structure, the sharper the transition in the absorbance, and the smaller the magnitude of the FWHM of the van't Hoff plot. The following section describes the spring model used to represent the dsDNA structure.

3.1 The DNA Spring Model

DNA has been modeled physically by many models, but principally by one of four methods: 1) a two bead-spring system [35], 2) a bead-pin complex [29-30, 32], 3) a worm-like chain model (WLC) [11, 78] and 4) a freely-jointed chain model (FJC) [4]. The most recent work that applies the WLC model uses it to study the thermomechanical properties of general polymer molecules such as protein and DNA by Monte Carlo simulations [78]. These models have been used to characterize DNA sedimentation and its low-angle light scattering properties, to study base stacking interactions, hairpin structure kinetics and melting properties, and to study the elastic properties of DNA. DNA-STRAIN simulator treats dsDNA as a building block that possesses the physical properties of the WLC model, leading to a complex composable structure that ultimately can model DNA self-assembled nanostructures. The following sections present a bead-

spring model of dsDNA, first as a simple spring with two beads on either end, then as a three-dimensional rectangular spring model. The 3D model is then extended to DNA self-assembled nanostructures as a complex spring network. Three models of increasing size and mass are simulated and analyzed with respect to their properties of diffusivity and thermodynamic properties. The limitations of the simulator are discussed in Section 3.3.4.

3.1.1 The Simple DNA Spring Model

The simple spring model consists of two beads, hereafter referred to as mass nodes, on either end of a massless spring as shown in Figure 9:

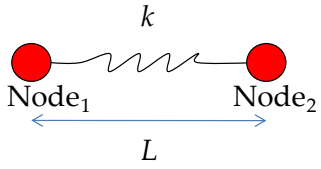


Figure 9: The simple dsDNA spring model.

Each spring represents a strand of B-form dsDNA of length nB and mass nM, where n is the number of base pairs and B is the length of a single base pair, 3.4 Å. The mass, M, is assumed to be 660 Daltons per base pair, with the average mass being approximately 330 Daltons per nucleotide. The resulting molar mass of dsDNA is divided by Avogadro's number and divided between the two mass nodes. The spring constant, k, is determined using the standard expression for dsDNA with lengths less than the persistence length (i.e., 53 nm) [9] as shown in Equation 7.

$$k^{\circ}(T) = \frac{3}{2}\frac{k_B T}{P_D L}$$

Equation 7

In the expression above, k_B is the Boltzmann constant, T is the temperature, P_D is the persistence length and L is the contour length of the DNA strand.

With this model, the physical response of DNA is limited to its linear extension or compression over a period of time for a two-dimensional depiction of the helical structure. To study how DNA self-assembled nanostructures respond as a function of temperature and time, the dsDNA model must be extended to a 3D system of springs to model the double helical structures to include self-avoidance and shear behavior. The following section presents the 3D spring model and its justification based on the elastic properties of the simple spring model.

3.1.2 The 3D DNA Spring Model

DNA is a double helical structure that, in the low force and displacement limit, extends and compresses like a spring. It is also exhibits behavior similar to that of a flexible rod or a freely-jointed chain. To capture these aspects of it physical response, a rectangular 3D spring network is used to model the double helical structure of DNA. Twenty-four springs connected by eight mass nodes compose the model. Figure 10 illustrates the model in terms of the frame structure and the support structure.

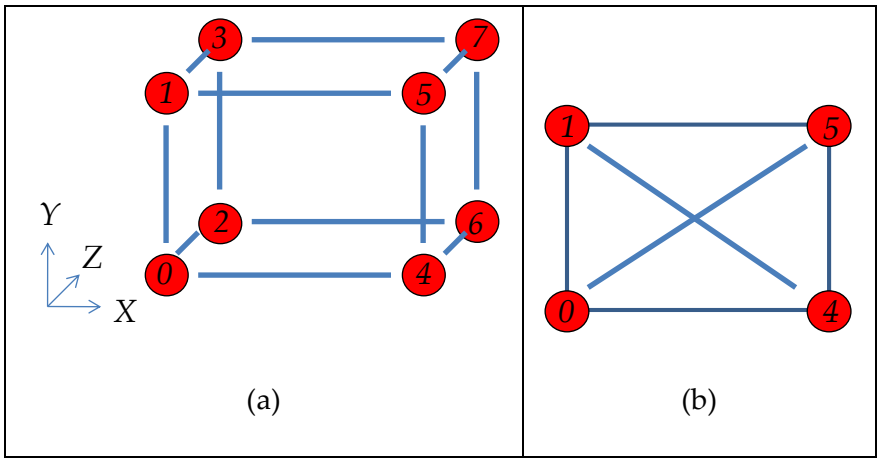


Figure 10: (a) The rectangular frame and (b) the cross braces for the 3D dsDNA spring model.

Eight springs form the six faces of the rectangular frame as shown in Figure 10(a). Twelve brace springs provide support across each face to prevent the model from collapsing into a degenerate structure, illustrated in Figure 10(b). The 3D model possesses the same physical and thermodynamic properties as the simple spring model because it represents the response of the DNA double helical structure in a volume, in contrast to the simple spring model that is limited to extension and compression of the structure. Therefore, both the mass values and the spring constants must be distributed appropriately over the spring network. The total mass of the simple model is evenly distributed over the eight mass nodes in the 3D model. The spring constants of twelve springs are calibrated so that the four horizontal springs are 20% of the value calculated for the single spring model, while the remaining eight crossing each face are 5% of the value. When subjected to Brownian forces and a changing temperature environment, each of the springs in the 3D model must replicate the elastic response of the simple model. The details of the implemented system used to achieve this response in the Monte Carlo simulator are described in Appendix B.1.

The next section establishes a relation between the physical response and the optical absorbance using the thermodynamic properties of DNA, followed by related work describing the diffusivity of dsDNA strands that will be implemented in the simulator to calibrate the Brownian force application on the model.

3.2 DNA Spring Model Theory

The connection between the elastic properties of dsDNA and its optical absorbance properties are established using thermodynamic properties that characterize the denaturation process, assumed to be unimolecular based on related work [21]. The solution of the derivation establishes the optical absorbance as the modality that will be used to analyze the physical response of dsDNA to its environment.

3.2.1 Connecting Optical Absorbance to Thermodynamic Properties

To ultimately demonstrate the relation between the spring constant and the absorbance, the relation between the fraction of denatured DNA and native DNA is first derived. It is assumed that the denaturing process is unimolecular, yielding the dissociation rate constant, K, for the transition of the native conformation, N, of dsDNA to the denatured conformation, D, of ssDNA using the following expression from [21]:

$$K(T) = \frac{D}{N} = \frac{f(T)}{1 - f(T)}$$

Equation 8

In Equation 8, the increasing fraction of ssDNA results in an increase in the dissociation rate constant. By rearranging the equation, the fraction of ssDNA, $f(T)$, can be expressed in terms of the dissociation rate constant, $K(T)$:

$$f(T) = \frac{K(T)}{1 + K(T)}$$

Equation 9

The expression defines the fraction of ssDNA as the temperature increases. As the dissociation rate constant becomes large, the fraction of ssDNA approaches one, indicating complete denaturation of the structure. The following equations are presented in terms of the fraction of ssDNA and the dissociation rate constant. Equation 10(a) and (b) relate the change in optical absorbance signal, A, and the dissociation rate constant to the thermodynamic properties of DNA.

(a)	$\frac{dA}{dT} = \frac{Bf(1-f)}{T^2}$
(b)	$K(T) = e^{\left[\frac{\Delta H}{R}\left(\frac{1}{T_m} - \frac{1}{T}\right)\right]}$

Equation 10

In these expressions, $K(T)$ is the unitless temperature-dependent dissociation rate constant, f is the fraction of ssDNA, ΔH is the van't Hoff transition enthalpy determined experimentally or by the nearest-neighbor model in [19], R is the gas constant and T_m and T are the melting temperature and the current temperature, respectively. The scaling factor, B, has $(signal)K^2$ units. These equations express the change in absorbance in terms of the fraction of ssDNA and the dissociation rate constant in terms of temperature and enthalpy. The expressions are used for the demonstration of the connection between the elastic properties and the optical absorbance properties of DNA by using the thermodynamic properties of its denaturation process. The derivation is

presented starting with the integration of Equation 10(a) with respect to T, grouping all terms dependent on temperature to the right:

$$\int \frac{dA}{dT} = \int \frac{Bf(T)[1 - f(T)]}{T^2}$$

$$\int dA = B \int \frac{f(T)[1 - f(T)]}{T^2} dT$$

Equation 11

Here, the integral is done by using a substitution of the derivative of the fraction of dsDNA. Let u be the expression for the fraction of dsDNA. Then,

$$u = \frac{1}{1 + K(T)} = \big(1 - K(T)\big)^{-1}$$

Equation 12

Taking the derivative of Equation 12 with respect to temperature results in the following expression:

$$\frac{du}{dT} = -\frac{1}{\big(1 - K(T)\big)^2} \frac{d}{dT} K(T)$$

Equation 13

The derivative of the temperature-dependent dissociation rate constant must also be taken with respect to temperature:

$$\frac{d}{dT} K(T) = \frac{\Delta H}{RT^2} K(T)$$

Equation 14

Substitution of the resulting expression into Equation 12 gives:

$$\frac{du}{dT} = -\frac{1}{\big(1 - K(T)\big)^2} \left[\frac{\Delta H}{RT^2} K(T)\right]$$

Equation 15

Rearranging Equation 15 to group like terms together results in:

$$du = -\frac{\Delta H}{RT^2}\frac{K(T)}{\left(1-K(T)\right)^2}dT$$

Equation 16

Equation 16 can be rewritten in terms of the fraction of ssDNA and dsDNA using the following equation:

$$\frac{K(T)}{(1+K(T))^2} = f(T)[1-f(T)]$$

Equation 17

The derivative of the fraction of dsDNA in the final expression is then:

$$du = -\frac{\Delta H}{RT^2}f(T)[1-f(T)]dT$$

Equation 18

Rewriting Equation 18 in terms of the integral in Equation 11 yields:

$$du = -\frac{\Delta H}{R}\frac{f(T)[1-f(T)]}{T^2}dT$$

Equation 19

The integration of Equation 11 results in:

$$dA = -\frac{BR}{\Delta H}\int du$$

$$A^{\circ}(T) = -\frac{BR}{\Delta H}u + C$$

Equation 20

To rewrite the integral in terms of the logistic sigmoid function, let A_0 be the base absorbance of the fully denatured sample and α_0 be the change in optical absorbance determined by subtracting the absorbance of ssDNA from the absorbance of dsDNA:

$$\alpha_0 = -\frac{BR}{\Delta H}, C = A_0$$

Equation 21

The substitution of these terms into Equation 20 results in the logistic sigmoid function, completing the integration.

$$A^{\circ}(T) = A_0 + \alpha_0 \frac{1}{1 + K(T)}$$

Equation 22

The absorbance of DNA is now expressed in terms of a temperature-dependent dissociation rate constant.

To demonstrate the connection between the physical properties of DNA and the optical properties of the DNA, the derivative of Equation 22 is taken with respect to temperature:

$$\frac{dA^{\circ}}{dT} = \frac{d}{dT}\left(A_0 + \alpha_0 \frac{1}{1 + K(T)}\right)$$

$$\frac{dA^{\circ}}{dT} = \frac{d}{dT}A_0 + \alpha_0 \frac{d}{dT}\frac{1}{1 + K(T)}$$

Equation 23

Substituting Equation 12 into the expression results in:

$$\frac{dA^{\circ}}{dT} = \frac{d}{dT}A_0 + \alpha_0 \frac{d}{dT}u$$

Equation 24

Taking the derivative of the term by substituting Equation 15 yields:

$$\frac{dA^{\circ}}{dT} = -\frac{\Delta H}{RT^2} f(T)[1 - f(T)]$$

Equation 25

Let $B = -\frac{\Delta H}{R}$ be a scaling factor to rewrite Equation 25:

$$\frac{dA^{\circ}}{dT} = \frac{Bf(1-f)}{T^2}$$

Equation 26

Equation 26 is the same as Equation 10(a). Rewriting Equation 22 in terms of the fraction of denatured DNA results in the following equation:

$$A^{\circ}(T) = A_0 + \alpha_0[(1 - f(T)]$$

Equation 27

Equation 27 is the postulated absorbance expression. Rewriting the fraction of dsDNA in terms of the temperature-dependent dissociation rate constant gives an equivalent expression in terms of the dissociation rate constant:

$$1 - f(T) = 1 - \frac{K(T)}{1 + K(T)}$$

$$1 - f(T) = \frac{1 + K(T)}{1 + K(T)} - \frac{K(T)}{1 + K(T)}$$

$$1 - f(T) = \frac{1}{1 + K(T)}$$

Equation 28

Rewriting Equation 22 in terms of the fraction of dsDNA by substituting Equation 28 results in Equation 27 as presented in the following:

$$A^{\circ}(T) = A_0 + \alpha_0 \frac{1}{1 + K(T)}$$

$$A^{\circ}(T) = A_0 + \alpha_0[(1 - f(T)]$$

Equation 29

Thus, the connection between the optical absorbance of DNA and its thermodynamic properties has been established by both integration and derivation. The next section presents three postulates for the absorbance, restoring force, and the spring constant to establish the connection between the physical properties and optical absorbance that are implemented in the simulator.

3.2.2 Connecting Optical Absorbance to Physical Properties of DNA

The expression for the temperature-dependent spring constant is derived starting with the contour-length dependent expression for the spring constant presented in Equation 7. While the expression is true for the temperature range in which DNA behaves like a Hookian spring [4], the expression must be modified to capture the graceful degradation of the spring constant to model the denaturation process of dsDNA as the temperature increases past the melting temperature. The inflection point of the transition is centered on the melting temperature of DNA based on the fraction of ssDNA as was shown in Figure 8 in which the absorbance of a DNA molecule was plotted with the change in absorbance with temperature. To model the desired degradation behavior, the term for the fraction of dsDNA, $1 - f(T)$, is added to Equation 7. The resulting equation, along with the following postulates, is used to make a connection among the physical, optical, and thermodynamic properties. It remains to be shown in Section 3.3.2 how these postulates yield the length, temperature, and dynamic behavior of DNA.

Postulate 1	$\langle k(T)\rangle = k^{\circ}(T)[1 - f(T)]$
Postulate 2	$\langle A(T)\rangle = A_f + \alpha_1\langle x_B\rangle[1 - f(T)]^2 \equiv A^{\circ}(T)$

Equation 30

The postulates are connected by the total force exerted on the mass nodes in the model. Postulate 1 states that the spring constant of DNA, $k^{\circ}(T)$ that is presented in Equation 7, decreases in magnitude as the probability of native dsDNA decreases with temperature, giving the spring constant the desired degradation behavior. This behavior affects the observed restoring force of the spring, $\langle F_R\rangle$, which is the product of the spring constant of DNA and one tenth the contour length based on results from [15], as shown in Equation 31.

$$\langle F_R\rangle = k^{\circ}(T)\frac{L}{10}$$

Equation 31

As the temperature increases, $k(T)$ decreases in magnitude, resulting in a lower F_R, allowing the change in length of the spring, x_B, to increase with the Brownian force. This expression also satisfies the observation from related work describing the general change in length at low temperatures. Postulate 2 states that the observed absorbance, $\langle A(T)\rangle$, is expressed as the absorbance, A_f, affected by the product of $\langle x_B\rangle$, the observed change in length, with a corrective scale factor, α_1, and the squared probability of the fraction of native dsDNA. The following uses Postulate 1 to show that the simplified expression in Postulate 2 is demonstrated to be the equivalent to the logistic sigmoid function in Section 3.2.1.

The expression for the change in length is determined by substituting Postulate 1 and the observed restoring force into the equation for the restoring force of a spring:

$$\langle F_R \rangle = \langle k(T) \rangle \langle x_B \rangle$$

$$\langle x_B \rangle = \frac{k^{\circ}(T)L}{10k^{\circ}(T)[1 - f(T)]}$$

$$\langle x_B \rangle = \frac{L}{10[1 - f(T)]}$$

Equation 32

In Equation 32, L is the contour length of the dsDNA molecule. The variation in the spring length is treated as the RMS over a period of time in the simulation due to the fraction of dsDNA being a probability of a large number of DNA strands in a sample. To this end, the magnitude of the change in the length determines the significance of the value on the overall absorbance of the DNA strand. The final expression indicates that as the temperature increases and the fraction of dsDNA approaches zero, the magnitude of the length becomes very large, as expected. Substitution of the expression into Postulate 2 shows that the scaling factor α_1 is a constant ratio of the change in the absorbance to the length of a base pair. The expression reduces to Equation 22 as presented in the following:

$$\langle A(T) \rangle = A_f + \alpha_1 \langle x_B \rangle [(1 - f(T)]^2$$

$$\langle A(T) \rangle = A_f + \alpha_1 \left(\frac{L}{10[1 - f(T)]} \right) [(1 - f(T)]^2$$

$$\langle A(T) \rangle = A_f + \alpha_1 \left(\frac{L}{10} \right) [1 - f(T)], \alpha_1 = \frac{\alpha_0 10}{L}$$

$$\langle A(T) \rangle = A_f + \alpha_0 [(1 - f(T)]$$

$$\langle A(T)\rangle = A_f + \alpha_0[(1 - f(T)], [(1 - f(T)] = \frac{1}{1 + K(T)}, A_f = A_0$$

$$\langle A(T)\rangle = A_0 + \alpha_0 \frac{1}{1 + K(T)} \equiv A°(T)$$

Equation 33

Equation 33 demonstrates that there is a relation among the thermodynamic, optical, and physical properties of DNA based on the assumptions and postulates that have been presented in this section. The postulates are implemented in the DNA-STRAIN simulator. To accurately implement the expressions, the Brownian force applied to the models must be evenly distributed over the structures in order for them to diffuse as shown in related work [79]. The next section describes the methodology for calculating the diffusivity of the dsDNA model.

3.2.3 The DNA Diffusion Model

In the DNA-STRAIN simulator, the Brownian force is dependent on the temperature and affects the diffusivity of the models. The diffusion coefficient is determined based on previous work [79-81] treating short dsDNA as a rigid rod structure using the diffusivity, D, as expressed in Equation 34.

$$D \cong \frac{k_B T}{3\pi\eta L/(ln(L/d) + \gamma)}$$

Equation 34

The expression is dependent on the viscosity of the aqueous medium, η, the contour length, L, the diameter of DNA (2 nm for dsDNA) and the correction for end effects, γ, assumed to be -0.73 from [79] for an 18 bp strand. The diffusivity of DNA is generally dependent on the number of base pairs, M. The diffusion coefficient for an 18

base pair dsDNA segment is used to extrapolate the coefficients for a range from 1 to 155 base pair segments based on the proportion expression dependent on the number of base pairs, $D \sim 1/M^{(0.68 \pm 0.03)}$, as described in [79]. Table 2 presents the resulting coefficients.

Table 2: Calculated diffusion coefficients based on work from [79].

Base Pairs	Calculated dsDNA Diffusivity (m^2/s)
1	2.21E-10
5	7.41E-11
10	4.62E-11
13	3.87E-11
18	3.10E-11
20	2.89E-11
25	2.48E-11
30	2.19E-11
35	1.97E-11
40	1.80E-11
45	1.66E-11
50	1.55E-11
155	7.17E-12

While the expressions presented in this section connect the physical, optical, and thermodynamic properties of DNA, the distribution of the Brownian force over the model must be calibrated based on the total mass of the DNA strand. The diffusivity values in the table are used for this calibration in the DNA-STRAIN simulator.

3.3 The Simulator

3.3.1 General Overview

The DNA-STRAIN simulator is a Monte Carlo simulator that applies randomly generated forces to a user-defined 3D spring network model that represents a DNA nanostructure. The change in position of each mass node in the system is dependent on

the magnitude of the forces applied and the stiffness of the springs. The properties of the springs (i.e., stiffness, diffusivity), as well as the Brownian force applied, are dependent on the temperature of the environment. The spring response is reflected in the change in absorbance of the structure. This section describes the simulation and validation process for the simulator by analyzing the elastic response, the diffusivity, and the denaturation process of 3D spring networks representing dsDNA.

The simulator is capable of simulating DNA nanostructures and, in principle, generating any observable modality concerning the physical, thermodynamic, and optical absorbance properties of DNA. The input file that contains the initial properties of the DNA model is generated using a mesh file-generator implemented in MATLAB and is provided in Appendix B.5. The simulator is implemented in C with a number of command flags that are used to define the parameters for the simulation. The input file generator is used to provide the required parameters that define the details such as the 3D model structure, the output data resolution and the concentration of the DNA sample. The output data is generated either at constant temperature or over a temperature range, and is processed for further analysis (e.g., absorbance, length, diffusivity, melting temperature). A graphical output program can use the mass coordinate data generated as a diagnostic tool for studying the physical behavior of the model over the course of the simulation. A more detailed description of these aspects of the simulator is provided in Appendix B. The next section describes the simulation process.

3.3.1.1 The Simulation Process

The input mesh file contains all the information needed to initialize the simulation environment as well as the mass node and spring properties. Once initialization is complete, the simulator generates the trajectories for the model, storing the data in arrays. Upon completion of the simulation, the data are written to output files for further analysis. The pseudocode is shown in Figure 11.

```
load command inputs
load input file
initialize simulation parameters
while(run < totalruns)
        recordData
        reset simulation parameters
        while(currentTemp <= finalTemp)
                updateSpringConstants
                while(currentTime < simulationTime)
                        while(spring <= lastSpring)
                                sumForces
                                increment spring
                        end
                        while(node <= lastNode)
                                applyForces
                                updateMassPosition
                                updateSpringProperties
                                increment node
                        end
                        updateModelPosition
                        updateData
                        increment currentTime
                        end
                increment currentTemp
                end
        increment runCount
        end
 end
outputData
```

Figure 11: The pseudocode for the DNA-STRAIN simulator.

The initialization phase of the simulation determines the temperature range, the duration of the time period, the initial mass node positions, the support spring assignments, the initial spring constants, the initial absorbance value of the sample and

the data resolution (i.e., time step and temperature increments). The trajectory generation begins following initialization.

The simulator currently does not support exact calculation of extinction coefficients for specific strands. While in principle, precise sequence-dependent constants could be used, the extinction coefficient values for ssDNA and dsDNA are derived by the nearest neighbor expression in [82] using specific ssDNA sequences. The sequence-averaged extinction coefficients are calculated to be 24156 $M^{-1}cm^{-1}$ for dsDNA and 29181 $M^{-1}cm^{-1}$ for ssDNA. Details on the calculation are described in Appendix C.

The extinction coefficients are validated by comparing the result to experimental data from [18], in which the change in absorbance on average is approximately 17% of A_0 and experimentally demonstrated for dsDNA oligonucleotide sequences. Using the calculated extinction coefficient values, the change in absorbance for the model is 18.3%, within 10% of the results in [18]. The initial and final absorbance values are calculated using these extinction coefficients along with the total number of base pairs calculated from the input mesh file.

In each cycle, the temperature is checked to see whether the final temperature has been reached. If not, the spring constants are updated according to the current temperature, and the trajectory is generated over the current time duration. A randomly generated Brownian force vector is applied to each set of four mass nodes per face in the model while below the melting temperature.

The spring lengths are calculated following the application of the Brownian force. The change in the spring length is used to calculate the restoring force vectors exerted by the springs in the model. Each mass node keeps a running sum of the total force vectors per time step, changing their coordinate each time step based on the resulting vector. The data are then stored in arrays, and the time step is incremented. This process repeats until the end of the simulated time period. Once the time period has ended, the temperature is incremented, and the time duration is reset to run the next simulated time period. Upon reaching the final temperature, the data are processed to output files. The data from previous runs are stored to be averaged with subsequent runs. When the simulation has reached its final iteration, the averaged data are written to output data files. The source code for the DNA-STRAIN simulator is provided in Appendix B.3. The next section describes the generation and analysis tools used to create the input mesh files as well as to process and to analyze the simulation data.

3.3.1.2 Post-Processing Tools

Post-processing software is used to analyze the absorbance data generated by a melting simulation. A Savitsky-Golay filter is used to smooth the absorbance data using a 3rd order polynomial regression over a 101-point window of data equivalent to ±5°C for melting simulation data. The filter preserves the features of a plot better than filters that apply conventional averaging techniques, but is not as good at removing noise that might affect analysis. The following is a description of the process for determining the optimal width of the filter window dependent on the resolution of the data.

Using a 13 bp dsDNA segment from [18], a series of simulations were completed to validate the simulator for a dsDNA model. Each run generated 1000 data points over a 100 K range with 5 ns simulated time per 0.1K temperature increment. The filter window was selected from a range of windows ranging from ±1°C (11 point window) to ±15°C (201 point window), followed by an extraction of the melting temperature and the enthalpy. These two parameters were compared to the experimental data for the 13 bp dsDNA molecule as shown in Table 3.

Table 3: Savitsky-Golay Filter results for 13 bp 3D model.

Data Window	dA/dT Window	Tm (°C)	% Error	ΔH (kcal/mol)	% Error	R
11	11	72.605	-1.89%	110.82	-5.28%	0.48324
11	21	72.605	-1.89%	110.81	-5.29%	0.72787
11	51	72.608	-1.88%	110.35	-5.68%	0.8443
11	101	72.621	-1.86%	106.41	-9.05%	0.89338
11	151	72.637	-1.84%	98.278	-16.00%	0.91527
11	201	72.695	-1.76%	87.464	-25.24%	0.91269
51	11	72.608	-1.88%	110.34	-5.69%	0.92595
51	21	72.608	-1.88%	110.33	-5.70%	0.94236
51	51	72.61	-1.88%	109.92	-6.05%	0.9523
51	101	72.621	-1.86%	106.17	-9.26%	0.96342
51	151	72.636	-1.84%	98.06	-16.19%	0.97311
51	201	72.69	-1.77%	87.012	-25.63%	0.97101
101	11	72.621	-1.86%	106.35	-9.10%	0.98977
101	21	72.621	-1.86%	106.34	-9.11%	0.99298
101	51	72.62	-1.86%	106.13	-9.29%	0.99406
101	101	72.623	-1.86%	103.6	-11.45%	0.9944
101	151	72.741	-1.70%	96.55	-17.48%	0.9935
101	201	72.688	-1.77%	85.917	-26.57%	0.99143

From the results in the table, the ±5°C window (the 101 point window) was selected because it was the largest window that preserved the shape of the absorbance plot while removing the most noise while maximizing the R (correlation) value. The resulting data is then processed further by calculating the change in absorbance over temperature to generate a van't Hoff plot. The windows for the filtering process are determined based on the number of data points as well as the variation in power from the light source used to obtain the data.

From this plot, the thermodynamic parameters are extracted using another MATLAB program based on an existing method [21]. The van't Hoff plot can provide information based on the shape of the curve that is dependent on the sharpness of the transition during the melting process. The local maximum indicates the melting temperature of DNA while the full width half maximum (FWHM) determines the enthalpy at which the melting occurs. From this data, the entropy and the free energy can be determined using the Gibbs free energy expression $\Delta G = \Delta H - T\Delta S$ at 25°C. The source code for the tools is provided in Appendix B.6 and B.7.

3.3.2 Validation Methodology

The simulator is validated by testing the physical response (i.e., oscillation, diffusivity, length variation, and deflection) of simple DNA models as well as comparing simulated absorbance and thermodynamic parameters (i.e., enthalpy, melting temperature, entropy, and free energy) to experimental data from related work. This section presents and discusses the results of these validation tests for spring behavior.

The oscillation test requires setting the initial length of a 3D model to a different value from its rest length at a temperature must be sufficiently less than the melting temperature. The test enables the validation of the elastic response simulated by the program. The diffusivity and length variation validations are dependent on the Brownian force distribution over the mass nodes based on the total mass of the DNA. The magnitude of the force experienced in each node is calibrated to the expected diffusion coefficient values described in Section 3.2.3. In the diffusivity test, the root mean square (RMS) distance travelled by the center coordinates of the model are recorded over the total time, t. The diffusivity, D, is extracted from the expression for the RMS distance, $\sqrt{6Dt}$, using Gnuplot. The extracted diffusion coefficient value is compared to the calculated diffusivity value in Table 2. The test validates the Brownian force applied over the entire model results in the diffusivity determined from related work [79]. For the length variation test, the length of a 3D model is recorded over a period of time and analyzed. According to recent work [15], the range of the lengths is within ±10% of the contour length, and the length variation test validates that the model exhibits the same behavior. The deflection test validates the physical properties for multi-segmented models that must diffuse as a stiff rod because the simulations are for models of dsDNA that are below the persistence length. The melting test validates the 3D spring model of a 13 bp model by comparing the thermodynamic properties to previous work [18].

3.3.2.1 The Oscillation Test

The undamped (i.e., vacuum) behavior of the simple and 3D spring models is validated based on the expected oscillation time period expression of a simple harmonic oscillator, $2\pi\sqrt{m/k}$, that is dependent on the mass node value, m, and the spring constant of DNA, k. For example, the time period of oscillation for a 5 base pair segment using calculated mass and spring constant values is 1.35 ns. The simulator is run for both the simple spring model and the 3D spring model for a 5 base pair segment, setting the initial positions of the mass nodes in the input file to be shorter (1.4 nm) than the rest length, for 5 ns. The length (in nm) of the segment over the time period is plotted for both models using Gnuplot as shown in Figure 12.

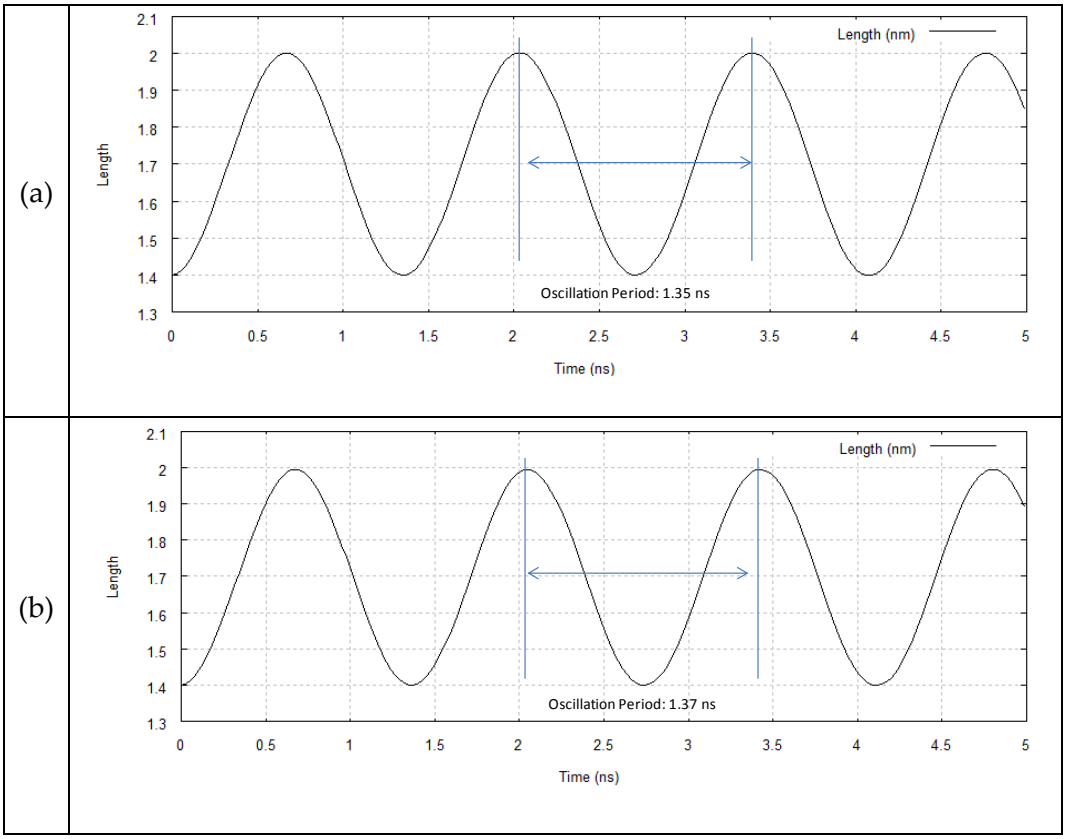


Figure 12: The length of a 5 bp (a) simple spring model and (b) 3D spring model over a 5 ns time period.

From the above plot, the simulation agrees with the expected calculations. The oscillation time period for the 3D spring model is 1.37 ns or 1.5% error. The oscillation test validates that the two models are in good agreement with each other using the calculated time period for the simple spring model as the figure of merit for the 3D spring model.

The oscillation test is used as a means for validating the elastic behavior of the models in a static, undamped environment. However, the simulations hereafter are in an aqueous environment with potentially increasing temperature. Quantum chemistry studies have shown that DNA resonance can be observed in highly structured or very dry environments [83]. It is assumed the springs are critically damped from this point forward in tests because the simulation environment is aqueous, and the dsDNA molecules simulated are much less than the persistence length. Thus, no resonance should be present in the simulation.

The calculation of the natural frequency of each DNA model is also used as a means of determining the minimum simulation time step. The minimum scaling factor for recording reasonable data is 11 steps per oscillation period, meaning that the resolution of the data at this point is still high enough to observe the oscillation period. While this scale factor is sufficient for measuring the oscillation period, the time step is scaled by a factor of 51 to avoid aliasing and to ensure good resolution of the data generated from the simulation. Increasing the factor beyond this point will increase the simulation time without significant improvement in the output data.

3.3.2.2 Brownian Force Calibration and Diffusion Validation

While the fundamental spring behavior is correct without any Brownian force applied, the diffusion behavior must be modeled using the calibration of Brownian force distributed over the mass nodes. The force applied to each mass node in the model is calibrated as a power scale dependence on the total mass value calculated for the DNA sample for a constant number of mass nodes per model. The force is also calibrated as a log scale dependence for the case of an increasing number of mass nodes per model to validate the deflection behavior of a WLC model below the persistence length of DNA. The diffusion coefficients presented in Table 2 in Section 3.2.3 are used to determine the scaling dependence for the Brownian force in terms of the total mass of the model, $\delta(M)$, and the total number of segments, $\gamma(S)$, as shown in Equation 35.

(a)	$F_B = \frac{3k_BT}{2P_D}\frac{1}{\delta(M)}\frac{1}{\gamma(S)}$
(b)	$\delta(M) = 0.0962M^{0.841}$
(c)	$\gamma(S) = -0.232\ln(S) + 1.056$

Equation 35

To determine the power scaling relation, the RMS distance traveled by the center of the 3D models of varying lengths for 1000 trajectories at 25°C over a 50 ns time period are recorded. The data are plotted over time in Gnuplot, and a diffusion coefficient is extracted from the fit to the RMS equation for diffusivity. This process is repeated five times to calculate the average diffusion coefficient. If the resulting value is within ±5% error of the calculated diffusion coefficient, the scaling factor is acceptable. Figure 13 shows the results for a 10 base pair segment.

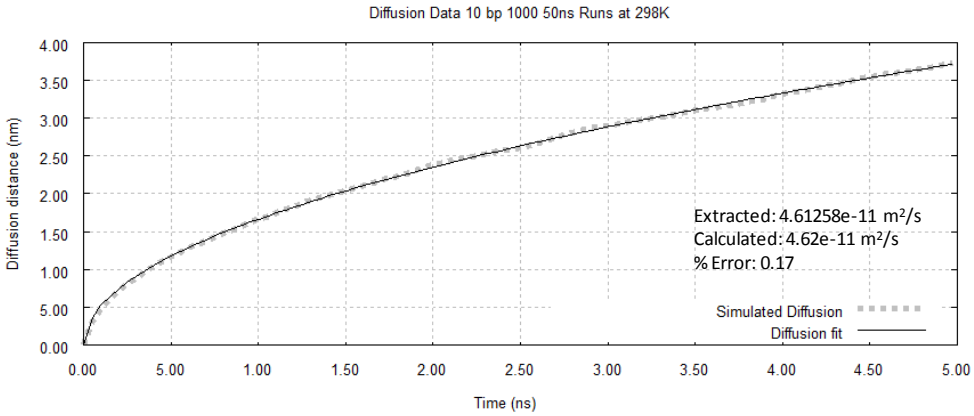


Figure 13: Diffusion fit curve for a 3D 10 bp model simulation, averaged over 1000 trajectories.

The diffusion coefficient extracted from the fit is within 5% of the expected value determined using Equation 34 assuming an aqueous medium at room temperature. The same method is repeated with a 5, 25, and 50 bp model to determine the mass-dependent power expression for scaling the Brownian force as shown in Equation 35b.

The power expression in Equation 35b used to fit the scaling factor values is validated by simulating 13, 18, 20, and 40 bp models. Each model is simulated five times over a 50 ns time period at 25°C to generate 1000 trajectories. Table 4 presents the results.

Table 4: Test results for scaled mass power equation fit for single segment models.

Scaling Factor Test for Diffusion Coefficients								
BP	**13 bp**		**18 bp**		**20 bp**		**40 bp**	
RUN	**Diff Coeff**	**% ERROR**	**Diff Coeff**	**% ERROR**	**Diff Coeff**	**% ERROR**	**Diff Coeff**	**% ERROR**
1	4.05E-11	4.79%	3.30E-11	6.42%	2.94E-11	1.91%	1.75E-11	-2.72%
2	4.04E-11	4.42%	3.07E-11	-1.15%	3.00E-11	4.00%	1.74E-11	-3.55%
3	3.98E-11	2.74%	3.13E-11	0.83%	2.91E-11	0.80%	1.82E-11	0.88%
4	3.88E-11	0.38%	3.11E-11	0.20%	2.96E-11	2.40%	1.73E-11	-4.22%
5	4.04E-11	4.50%	2.98E-11	-3.93%	2.87E-11	-0.65%	1.77E-11	-1.86%
MIN	3.88E-11	0.38%	2.98E-11	-3.93%	2.87E-11	-0.65%	1.73E-11	-4.22%
MAX	4.05E-11	4.79%	3.30E-11	6.42%	3.00E-11	4.00%	1.82E-11	0.88%
AVG	4.00E-11	3.37%	3.12E-11	0.47%	2.94E-11	1.69%	1.76E-11	-2.29%

The table shows that the mass-dependent distribution of the Brownian force satisfies the diffusive properties observed in [79]. In each model, the average diffusivity is within 5% of the calculated value for the five runs simulated.

The dsDNA models are composed of only one rectangular structure for the previous validation. However, the force distribution for larger nanostructures that are composed of multiple dsDNA models must also be validated. The same methodology is applied for these models for structures composed of 5 bp dsDNA models. With each additional segment added to the structure, the number of mass nodes increases while the mass values assigned to each node decreases. The Brownian force must be distributed over the structure so that the diffusivity of the structure is the same as the single segment model. The segment-dependent Brownian force calibration scaling factors fit the logarithmic expression in Equation 35c. Validation of the fit is done using 15, 25, 30 and 35 bp dsDNA models composed of 5 bp dsDNA structures as shown in Table 5.

Table 5: Test results for segment logarithmic equation fit for multiple segment models.

Scaling Factor Test for Diffusion Coefficients								
SEG	**3 (15 bp)**		**5 (25 bp)**		**6 (30 bp)**		**7 (35 bp)**	
RUNS	**Diff Coeff**	**% ERROR**	**Diff Coeff**	**% ERROR**	**Diff Coeff**	**% ERROR**	**Diff Coeff**	**% ERROR**
1	3.65E-11	4.05%	2.49E-11	0.53%	2.19E-11	-0.16%	1.93E-11	-2.08%
2	3.52E-11	0.34%	2.51E-11	1.06%	2.19E-11	-0.04%	2.07E-11	4.95%
3	3.60E-11	2.56%	2.51E-11	1.37%	2.22E-11	1.36%	2.00E-11	1.31%
4	3.52E-11	0.15%	2.50E-11	0.69%	2.21E-11	0.70%	1.96E-11	-0.83%
5	3.69E-11	5.22%	2.36E-11	-4.81%	2.20E-11	0.24%	1.99E-11	0.97%
MIN	3.52E-11	0.15%	2.36E-11	-4.81%	2.19E-11	-0.16%	1.93E-11	-2.08%
MAX	3.69E-11	5.22%	2.51E-11	1.37%	2.22E-11	1.36%	2.07E-11	4.95%
AVG	3.60E-11	2.47%	2.47E-11	-0.23%	2.20E-11	0.42%	1.99E-11	0.86%

The table shows that the average diffusivity for five runs of each model is in good agreement with the calculated diffusivity. From both the single and multiple segment models, the simulator generates a physical response that results in diffusion coefficients that are in good agreement with the diffusion coefficients extrapolated from theory and experimental data [23, 79] as described in Section 3.2.3. The next section validates the simulated physical behavior by analyzing the length variation and the absorbance values generated for dsDNA models.

3.3.2.3 Length Variation and Absorbance Validation

The variation in the length of several 3D spring models is observed to validate the scaled Brownian force based on the mass of each node in the model as described in the previous section. The following shows that the same relation fulfills the criterion for both the previous work on dsDNA diffusion as well as observed length variation for the single segment and multiple segment dsDNA models.

The length variation of the simulated models at a constant temperature below the melting point should agree with recent work that indicates that the variation due to thermal fluctuations are generally within ±10% of the contour length of the dsDNA molecule [15]. The simulations were run over a 1 μs time period, and every tenth time step was recorded for 3D spring models ranging from 5 bp to 155 bp below the melting points for each model. The melting temperatures were determined using random sequences by *OligoCalc* [23], an online calculator for oligonucleotide properties. The only model that required a simulated temperature below 25°C was the 5 bp dsDNA

model. The percentage of lengths recorded that fall within 10% of the contour length for each model is presented in Figure 14 and in Table 6.

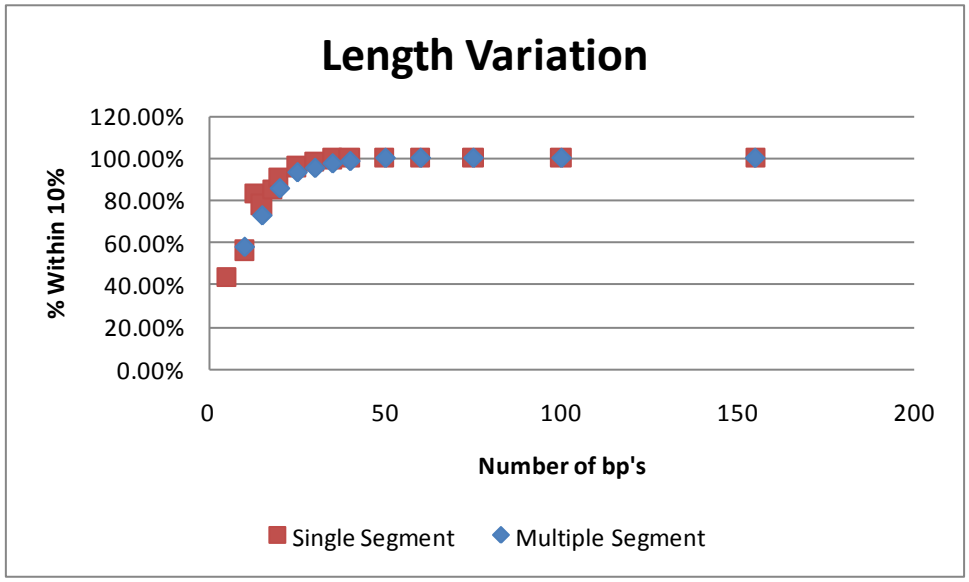


Figure 14: 3D spring models used for Brownian force test simulations.

Table 6: Percentage of lengths within 10% of the contour length for single segment and multiple segment simulations.

Model (#bp's)	Single Segment	Multiple Segment
5	44.00%	N/A
10	56.43%	57.78%
15	78.00%	72.72%
20	90.67%	85.56%
25	96.15%	93.18%
30	98.32%	95.23%
35	99.67%	97.38%
40	100.00%	98.49%
50	100.00%	100.00%
60	100.00%	100.00%
75	100.00%	100.00%
100	100.00%	100.00%
155	100.00%	100.00%

Analysis of the length variation using a MATLAB program indicates that percentage of lengths that fall within the ±10% range increases as the mass increases for

each model. The length variation is higher than expected for the shorter length models because the Brownian force calibration is dependent on the total mass of the model rather than the value at each mass node, which affects the diffusivity of the models. This presents a trade-off between the accuracy of the length variation and the accuracy of the diffusivity for models of shorter dsDNA strands. Increasing the accuracy of the length variation for shorter models would require increasing the mass value assigned to each node or changing the distribution of the Brownian force, resulting in a decrease in diffusivity accuracy. These are potential methods to correcting the discrepancy between the length variation and the diffusivity. For the current simulator, the accuracy of the diffusivity is emphasized because the results indicate a negative correlation between the length variation and the mass, i.e., as the mass increases, the diffusivity of the structure decreases, regardless of the way the mass is distributed over single or multiple segment models.

While the scaled force generates the diffusivity and absorbance that are in good agreement with related work, it also results in larger range in length variation for shorter length models. This observation can affect the calculation of the initial absorbance for nanostructures that are substantially larger in mass than the dsDNA models. The RMS value of the change in length is used in the simulation, and requires at least 5 ns worth of data to reach a value that gives the absorbance within 5% of the calculated value for each model. The absorbance value for each model is calculated using Postulate 3 with a 1 μM concentration. The percentage error from the calculated

absorbance values for the models simulated for the force calibration are plotted in Figure 15.

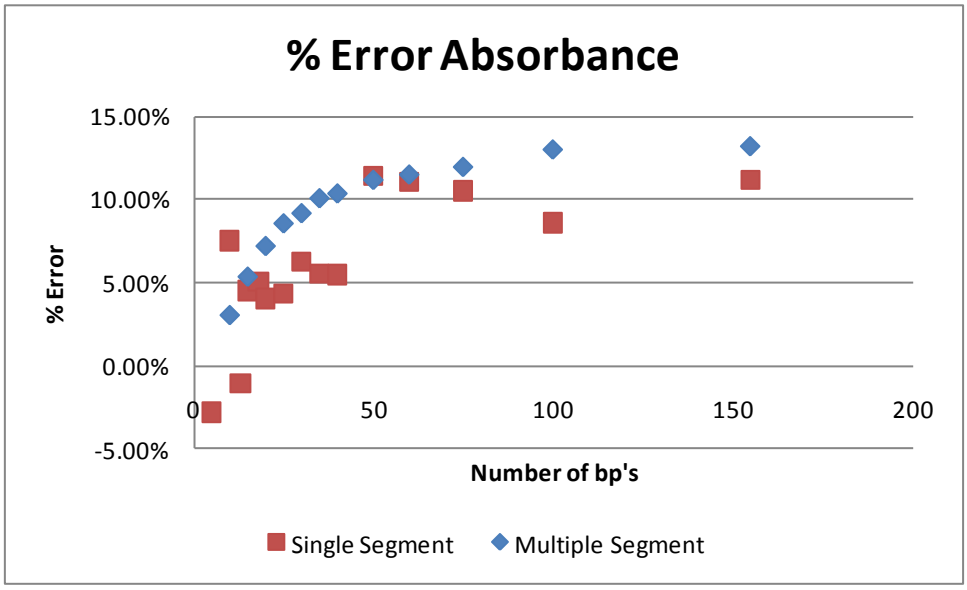


Figure 15: Percent error in simulated absorbance values.

The error in the absorbance values compared to the calculated absorbance values are within 12% over the range of both types of the simulated dsDNA models. The observation leads to the conclusion that larger or longer DNA nanostructures will have an error percentage comparable in magnitude regardless of whether they are treated as a single segment or broken into multiple composable elements. Approximately a 15% error is therefore expected for simulations of larger nanostructures when analyzing the absorbance data.

3.3.2.4 Deflection Properties of Multi-Segment DNA Models

The deflection ratio, i.e., the ratio between the dot products formed between the faces of the 3D rectangular model (details in Appendix B.4), for multiple segment 3D spring models is recorded for 1000 trajectories over a 50 ns time period using the 5 bp

3D spring model as a composable element. The simulation parameters are chosen based on the observation that the time needed per temperature increment in the simulation does not exceed 50 ns as concluded from the absorbance validation. Figure 16 shows the maximum deflection ratio for each simulated model of up to 100 base pairs.

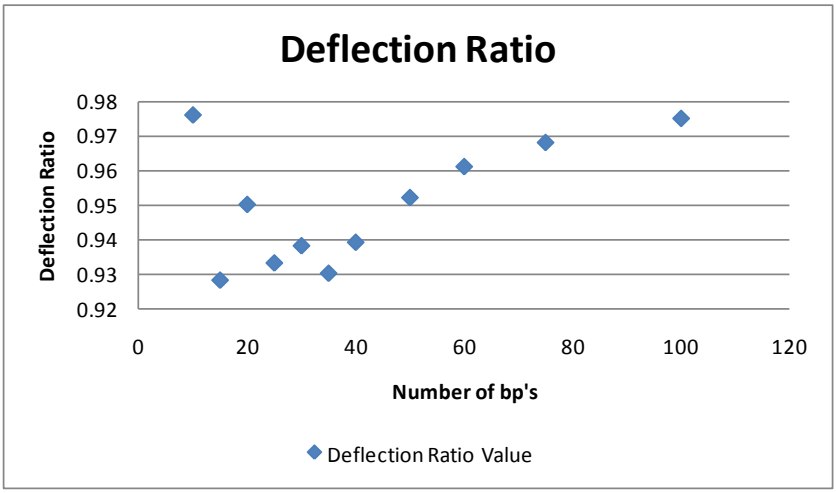


Figure 16: Deflection ratio measurements for 5 bp-composed models.

The maximum deflection ratio values are consistently around 0.94 for the simulation of models between 15 bp and 40 bp, indicating that the 3D DNA spring models retain their rod-like behavior when broken into multiple segments. However, as the length and the number of segments increases beyond this range, the deflection ratio increases, indicating that the models are more rigid in the simulation time period. As the mass becomes more evenly distributed over a larger number of mass nodes, the mass node values approach a constant. The deflection test validates the behavior of multiple segment 3D dsDNA models with contour lengths less than the persistence length based on the change in the angle between the distal ends of the dsDNA models as indicated by the calculated ratio. The spread of the results over this range of lengths

indicates that the deflection of the models is reasonable for larger nanostructures provided that the individual composable elements do not exceed this length.

3.3.2.5 Denaturation and Thermodynamic Parameters

With the 3D spring models of dsDNA validated for physical properties, the thermodynamic behavior is validated by simulating the models over a temperature range appropriate for the melting of the model. A 13 bp 3D dsDNA model is simulated from 50°C to 95°C in increments of 0.1°C with 5 ns time period for 100 trajectories. The collected data is then processed using a 101 data point window Savitsky-Golay filter with a third order polynomial regression. The van't Hoff enthalpy, the melting temperature, the free energy, and the entropy are extracted using the thermodynamic parameter MATLAB program. The extracted thermodynamic parameters and the normalized absorbance values are compared to those experimentally determined for a 13 base pair dsDNA strand studied in [18]. Figure 17 presents the resulting average length and absorbance assuming a 1 μM sample of a 13 base pair segment.

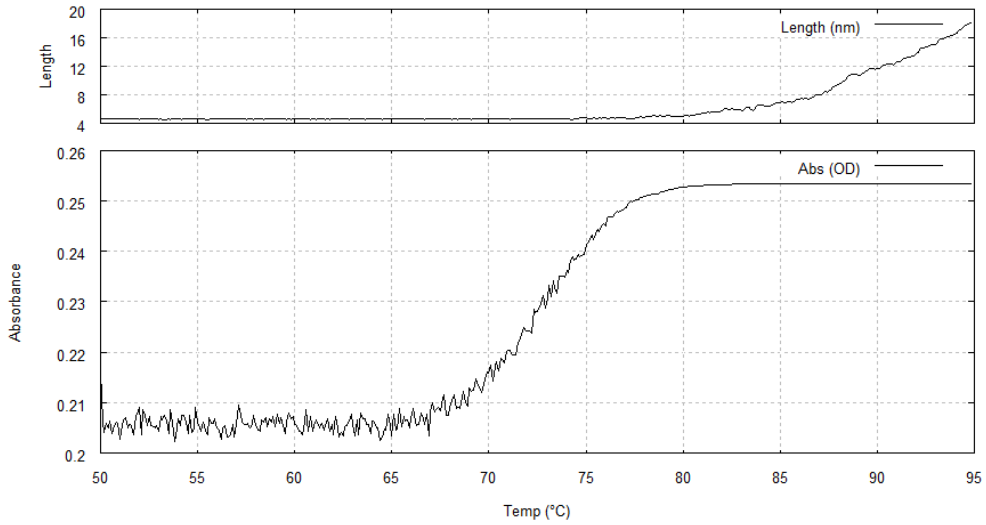


Figure 17: The absorbance and length plots for a 13 bp 3D dsDNA model for 100 trajectories.

The absorbance increases as the DNA strand denatures over the expected melting temperature to give a 22% absorbance difference, within 1% of the change in absorbance from the normalized experimental absorbance data in [18]. The initial and final absorbance values are also within 2% of the calculated values using the concentration and extinction coefficients in the input file. The length of the strand increases after the temperature has passed the melting point, indicating that the mass nodes are no longer being held by the spring. The thermodynamic properties of the 13 bp dsDNA model can be determined after filtering the data. Figure 18 shows the resulting van't Hoff plot along with the extracted thermodynamic parameters in Table 7.

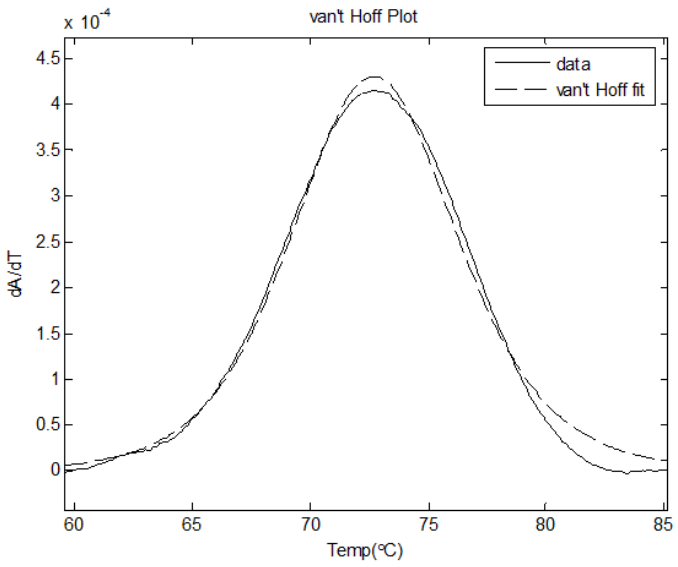


Figure 18: The van't Hoff plot for the 13 base pair 3D spring model.

Table 7: Resulting thermodynamic parameters extracted from the van't Hoff plot.

	Extracted	Expected	% Error
Tm (°C)	72.759	74	1.68%
ΔH (kcal/mol)	105.26	117	10.03%
ΔG (25°C kcal/mol)	16.869	20	15.66%
ΔS (cal/degree mol)	353.22	325.4	-8.55%

The thermodynamic parameters extracted from the plot are in fair agreement with the parameters from [18]. The error is due to the assumptions made concerning the enthalpy calculation as an average value of data from [19] is used for base pair enthalpies in the simulation. Related work concerning the base-stacking energies of dsDNA molecules could give more accurate results considering their analysis using the base sequences [84]. There are variations in thermodynamic data for different sequences depending on the techniques used (differential scanning calorimetric data, ultraviolet spectroscopy, and circular dichroism) with as much as a 10% difference between

methods measuring the same parameter [18]. The range of error is acceptable because of the observed variation in parameter values among these methods.

The DNA-STRAIN simulator was validated using simple dsDNA to generate spring trajectories that are analyzed for physical response, optical absorbance, and thermodynamic parameters. The oscillation validation test indicated that the physical response of the 3D spring network model behaved within 1.5% deviation compared to the simple spring model. The Brownian force was calibrated to distribute the forces among the mass nodes in the 3D model so that the diffusivity of the dsDNA was accurately simulated to within 5% error. The validation of the DNA-STRAIN simulator was completed by analysis of a 13 bp 3D dsDNA model and compared it against results from related work to be within 16% error for the highest deviation in the parameters. The next section describes the application of the simulator to predict these properties for DNA grid nanostructures.

3.3.3 DNA Self-Assembled Nanostructure Simulations

The simulator generates trajectories, in the form of lists of mass node coordinates, the analysis of which are within 16% of the results in related work concerning the elastic, optical, and thermodynamic properties of dsDNA as shown in the previous section. The validation of the dsDNA models enables the treatment of these models as a composable element for modeling the DNA tile motif. This section presents the tile model composed of sixteen dsDNA models, referred to as thermodynamic regions, which are connected using models that represent the crossover and core regions. The basic tile motif is analyzed using the same methodology as presented in the previous section. The initial simulations, when compared to experimental data, inspire a set of theories concerning the modification of melting temperatures of the thermodynamic regions, resulting in simulation thermodynamic values that are within 22% of the experimental data. This model is then used to construct the models for the DNA grid nanostructures described in Section 1.4.2. The simulations of these nanostructures result in thermodynamic parameters that are within 5% of experimental results for the largest denaturation transitions, but cannot fully capture the properties of the smaller transitions (up to 75% error for the worst matching transition) of these grid nanostructures.

3.3.3.1 The Tile Motif Model

The input file generator implemented in MATLAB is used to generate the mass node and spring parameters to simulate a DNA tile motif from [85-86]. The construction of the model is validated using custom graphical output software by simulating the

model for 50 ns with Brownian motion, enabling observation of the model for proper connectivity of the composable elements. Figure 19 displays screen capture images of the model with labels indicating the spring types.

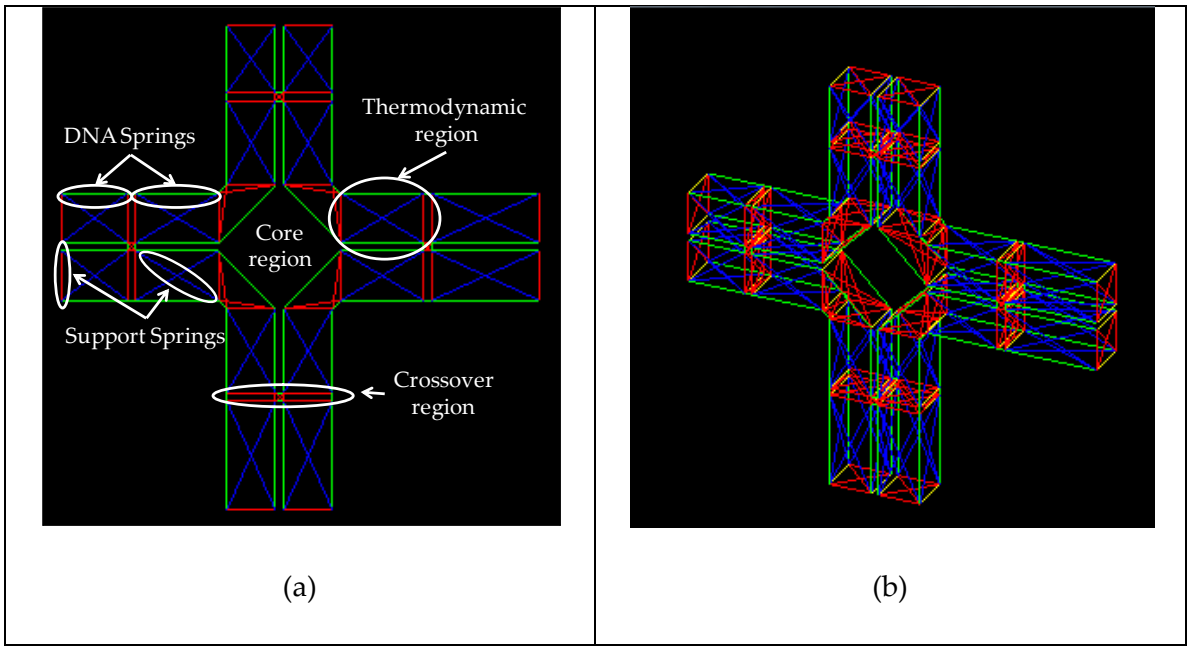


Figure 19: Screen capture images of the (a) front and (b) rotated view of the 3D DNA tile motif model.

There are 128 mass nodes and 544 springs that connect sixteen rectangular 3D spring models together to create the 3D tile motif model. Figure 19(a) labels the DNA springs as well as the support springs. The details of the generation of the tile motif model are described in Appendix B.4. Each thermodynamic region is assigned a melting temperature according to the specific oligonucleotide sequence in the region, using equations in [82]. The crossover and core regions that connect the sixteen thermodynamic regions of the tile motif are described in Appendix B.10. Figure 20 denotes the melting temperatures for each thermodynamic region, referred to as "R" and the assigned number, in the tile motif model.

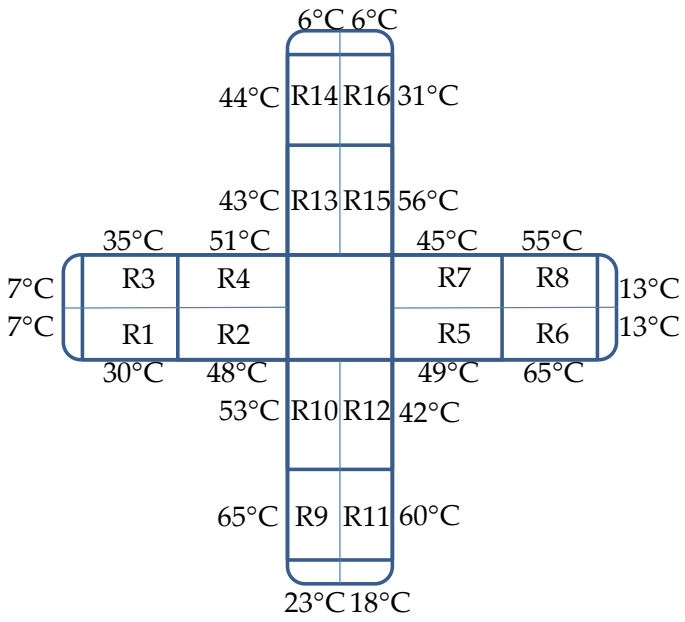


Figure 20: Thermodynamic region and melting temperature diagram for the DNA tile motif model.

The tile motif model is divided into sixteen numbered thermodynamic regions in the above figure, each labeled with their respective melting temperatures including the two 5 base pair "sticky ends" needed for self-assembly of larger structures on the end of each of the four arms. The sticky ends are assigned their individual melting temperatures, and are unique due to the design of the oligonucleotide sequences for controllable self-assembly of DNA nanostructures. However, the sticky ends are not included in the simulation unless they are used to connect multiple tile motifs as ssDNA is not modeled in the simulator below the melting temperature. While they are not included in the simulation process, they are accounted for during the calculation of the absorbance (i.e., an offset in the calculation for A_0). The following section describes the absorbance calculation compared to experimental data as well as the diffusivity of the tile model.

3.3.3.2 Tile Motif Diffusion and Absorbance

The tile motif is simulated over a 50 ns time period at a constant temperature of 298 K for 1000 trajectories to obtain data for calculating the diffusivity of the model. The simulation does not treat this structure as a multiple segment model because each thermodynamic region is separated by the crossover and core models that connect them. The diffusion coefficient is determined to be 6.66279e-12 m^2/s, within 1% of the extrapolated diffusion coefficient of a 175 base pair dsDNA molecule calculated using Equation 34. The result is fairly reasonable for this case as there are 164 base pairs in the tile motif model, and the nanostructure is not in a traditional double helix configuration. There are currently no reference values for diffusion coefficients of complex DNA self-assembled nanostructures. However, diffusivity has been studied for DNA nanostructures composed of flexible star polymers using fluorescence microscopy for drug delivery studies [87]. Results from the study suggested that the conformation, concentration, and the structural density of the DNA polymers affected the diffusivity.

At 1 μM concentration, the simulation gives an absorbance of 2.99 OD without the including the offset absorbance for the sticky ends. This value is 14.5% higher than the calculated absorbance of 2.61 OD from the extinction coefficients in the input mesh file. This discrepancy was also observed in the absorbance validation tests in Section 3.3.2.3 in which the increasing mass affected the calculation of the initial absorbance due to the affect on the Brownian force and the length variation. The tile motif model does not include the absorbance from the sticky ends, the poly-T ssDNA sections in the core or four base pairs in the core because of the design of the model. Correcting for these

omissions requires an offset of 0.99 OD for the absorbance data, giving a total calculated absorbance of 3.61 OD. A tile motif of 62 nM concentration measured using an AquaMate UV-Vis Spectrophotometer from Thermo Scientific results in a 0.222 OD measurement. Linearly scaling this value to a 1 μM concentration of DNA tile results in a 3.6 OD absorbance value, within 1% of the full calculated absorbance value. The extinction coefficient for ssDNA and dsDNA are determined using the expression from [82].

$$\varepsilon_{260} = \sum_{i,j=A,C,G,T,E} N_{ij}\, \varepsilon_{ij}^{nn}$$

Equation 36

The expression above determines the extinction coefficients of ssDNA and dsDNA based on the summation of the product of the number of nearest neighbor pairs, N_{ij}, with the extinction coefficient values associated with the nearest neighbor pairs, ε_{ij}^{nn}, reproduced in Appendix C. The subscripts denote the four bases (A, C, G, T) and the initiation (end) interactions, E, in the nucleotide sequence.

3.3.3.3 Tile Motif Melt Simulation

The tile motif is simulated over a temperature range of 100 K, 50 ns per 0.1°C temperature increment from -20°C to 80°C for 100 trajectories with Brownian forces. The total amount of time needed to run this simulation is approximately 5300 minutes or 3.6 days to simulate 5 ms. To reduce the simulation time, 100 runs are submitted to a computer cluster, reducing the time to complete the entire simulation to one hour. Approximately 15 ns of simulated time are processed in one second. The data generated

are averaged using a Python script prior to analysis. The resulting total length and the absorbance are shown in Figure 21.

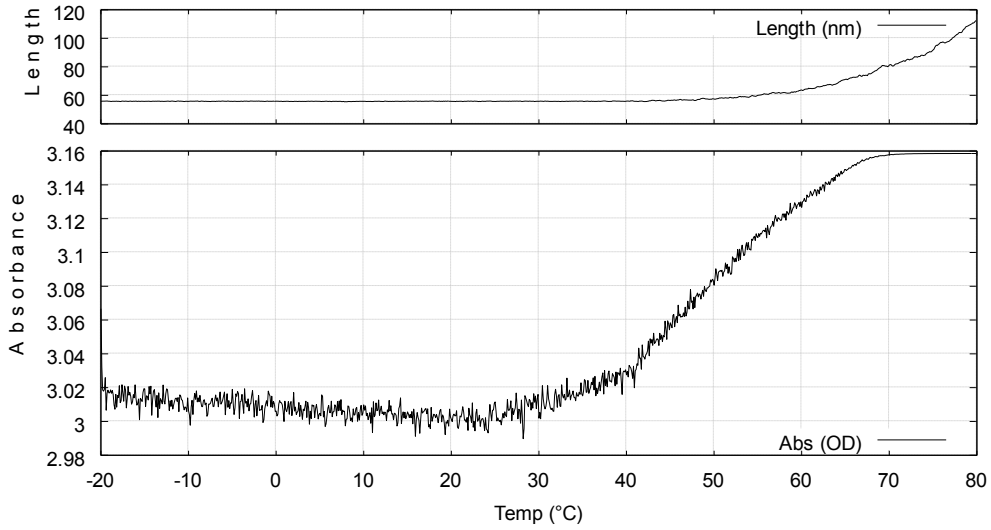


Figure 21: The total length and the optical absorbance of the 3D tile motif spring model.

The above figure indicates that the model is melting properly according to the total length recorded. Further validation is done by generating "snapshots" of the model at different points of the denaturing process using the graphical output program as shown in Figure 22.

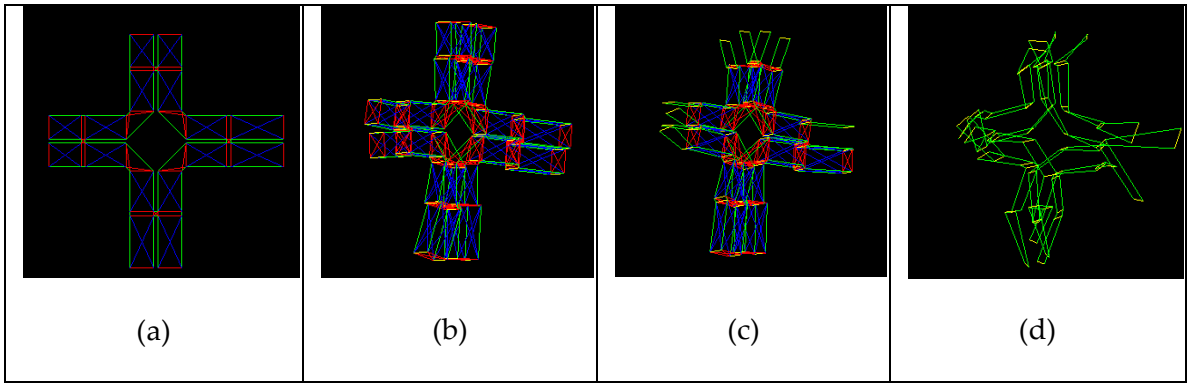


Figure 22: The tile motif simulation at (a) -20°C (b) 25°C (c) 50°C (d) 75°C.

Physical changes correspond to the changes in the optical absorbance as the tile motif melts. Figure 23 presents the normalized van't Hoff plot resulting from the simulation as well as the van't Hoff sum plot generated using a program in MATLAB. The expected transitions for each thermodynamic region are superposed assuming that each region can contribute to the plot independently based on individual melting temperatures and van't Hoff enthalpies as shown in Figure 23, labeled as the "van't Hoff sum" in the legend and normalized for comparison to the simulation. The figure also labels the transitions.

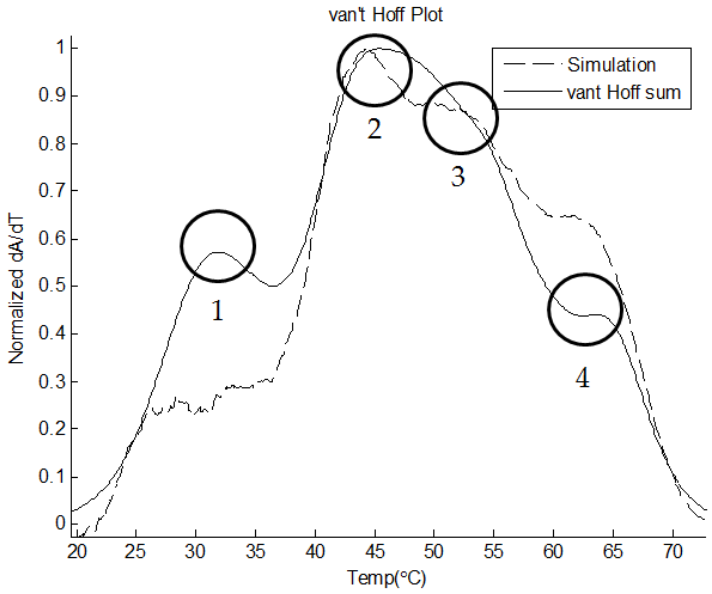


Figure 23: The van't Hoff plot for the tile motif denaturation process.

The largest optical absorbance transition indicates that the overall melting temperature of the tile motif is approximately 50°C. Using the thermodynamic parameter extraction program described in Appendix B.6, the overall transition is shown to be centered on this point in the van't Hoff plot after filtering the data using a 101 point, third order polynomial Savitsky-Golay filter. While the primary transition is

observable in the absorbance data, the van't Hoff analysis provides insight into the physical behavior of the individual segments that compose the tile motif during the melting process. The van't Hoff analysis using the thermodynamic parameter extractor program can also be applied to specific ranges in the data to determine the melting temperatures of the smaller transitions. Table 8 presents the results of this fitting process over the full range of the data.

Table 8: Extracted melting temperature transitions.

Transition	Extracted Tm (°C)	Closest Tm (°C)	%Error
1	34.586	35	1.18%
2	44.916	45	0.19%
3	49.861	51	2.23%
4	62.055	60	-3.43%

These smaller transitions indicate that it is possible to extract additional information from the van't Hoff plot for complex DNA nanostructures. Positive transitions indicate destabilization of the structure as the tile motif denatures. Negative transitions indicate stabilization occurring as parts of the tile become less strained. Transitions in simulation that are below the freezing point are not considered since the simulator does not account for phase or viscosity changes. The settings for obtaining the data used here are the same as those used for validating the single segment models.

The simulation data captures the general expected features from the total transition compiled by adding the individual melting temperatures and enthalpies for the sixteen thermodynamic regions. However, some features that are present in the simulation do not match the superposed van't Hoff plots in terms of their magnitude,

such as the transitions at ~30°C and ~60°C. While this comparison gives some indication of how the model behaves when constructed into a complex structure, it must be compared to experimental melting data. The following discusses the results of the experimental data and compares it to the model.

3.3.3.4 Experimental Tile Motif Melt

A DNA tile motif denaturation experiment was conducted four times using the DBS, the dual beam spectrofluorometer described in Chapter 4, to record data for a 30 nM sample heated from 30°C to 70°C over a two hour period. The temperature and the intensity of the beam were monitored using the reference chamber to correct for fluctuations in power from the light source.

To establish the threshold settings for the absorbance data, the noise threshold for the data due to the power fluctuations in the light source was characterized by monitoring the absorbance of a 2 × 2 DNA nanostructure motif over a four hour period at 23°C. A theoretical data set is generated using the average experimental absorbance, 0.13 OD, and randomly generated values using the standard deviation of the data, ±0.01 OD. The data set is filtered using the same order and windows as used for the experimental data to determine a ±1e-4 OD/°C, or a normalization of ±0.1 when comparing simulation data to experimental data, range as the noise threshold. The first and last sets of points in the filter window are also not considered as the filter uses these data points to begin generating the fit.

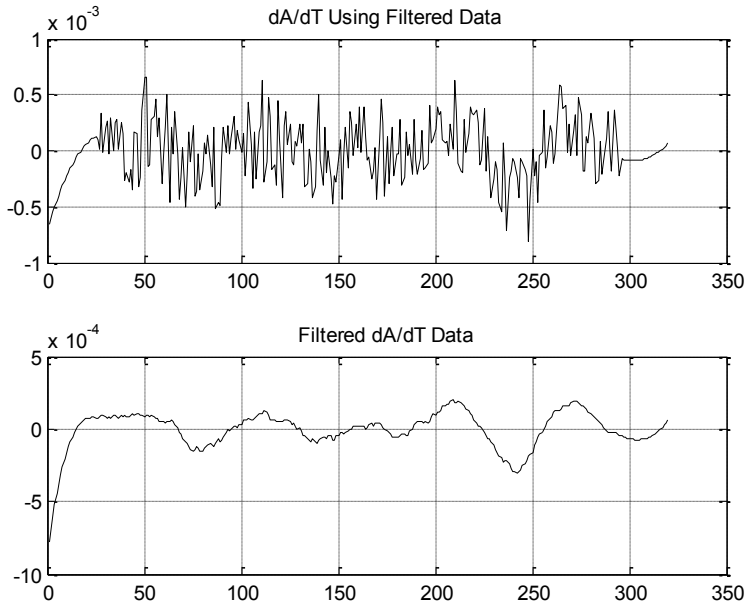


Figure 24: Noise threshold processing results from a set of simulated absorbance data.

The noise threshold is determined for the filter settings that are applied to the experimental data. A 51 point filter is used to smooth the absorbance data before taking the derivative. The derivative is then filtered again using a 51 point filter for the final van't Hoff data.

The absorbance data for the tile motif is analyzed using the thermodynamic parameter extraction MATLAB program described in Appendix B.6. Figure 25 shows the resulting data compared to the simulation data.

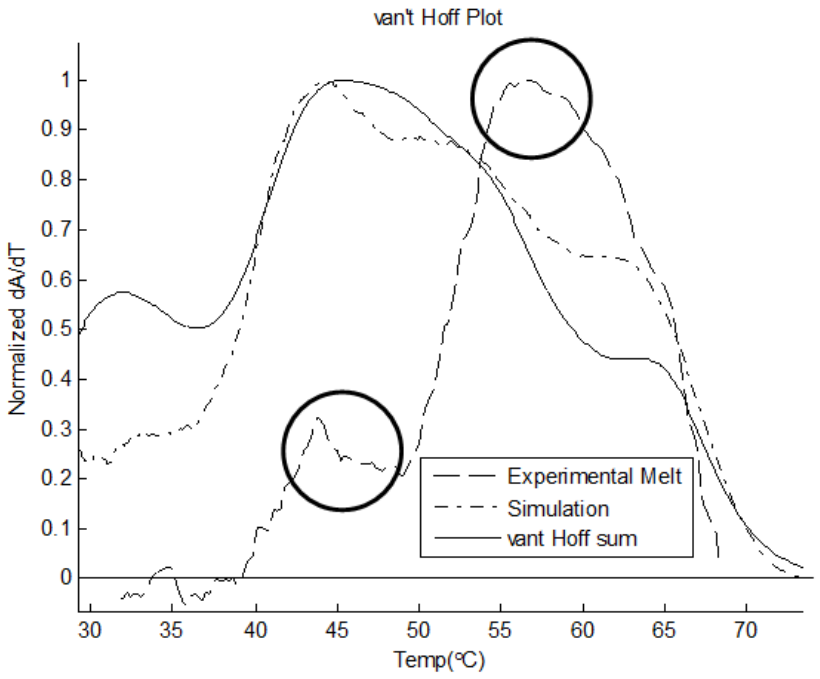


Figure 25: Simulation and experimental results for 1T denaturation.

While the experimental results indicate that the tile is stable up to 40°C, 25% higher than the lowest melting temperature in nearest neighbor model, the simulation data does not match well with the experimental data, indicating that the melting temperatures for the thermodynamic regions are not only a function of the nearest neighbor model. Figure 25 indicates the two transitions in the experimental data occur at 45°C and 57°C. While these temperatures within 5% of the melting temperatures for regions R7, R8, R13, R14, and R15 in the simulated tile motif model, the other melting temperatures are not present in the simulation. The temperatures in the experimental data indicate that a majority of the thermodynamic regions are within 40°C to 60°C, while the range in the model is from 30°C to 65°C. Melting temperatures may not appear in the van't Hoff plot because the temperatures are very close together, therefore broadening the transition without having a distinct peak. Another model must be

proposed for the tile motif with a different set of melting temperatures to more closely match the experimental data.

Two theories are used to determine melting temperatures for the thermodynamic regions to create a tile motif model that more closely matches the experimental data. The first theory, referred to as the core-shell theory, proposes that the melting temperatures for the eight regions that involve core and shell strand interaction (R2, R4, R5, R7, R10, R12, R13, and R15) are much higher than nearest neighbor model might otherwise predict. The theory is based on the hypothesis that thermodynamic regions that share a common strand with a substantially higher melting temperature (i.e., the core strand) will have the higher melting temperature than if it were an independent structure. The melting temperatures of these regions are determined using the bases in regions between each poly-T region in the core strand (i.e. the regions that interact with the shell strands). The melting temperatures of these regions are determined using the expressions in [82]. For example, because R2 and R4 share a common core strand, their melting temperatures are 60.2°C, the melting temperature of the core strand.

The second theory, referred to as the shell-arm theory, applies averages the melting temperatures of the core regions with the melting temperature of the eight remaining regions. This theory hypothesizes that the core strand affects the melting temperatures of the outer thermodynamic regions (i.e., R1, R3, R6, R8, R9, R11, R14, and R16) through the common strand that is shared – the shell strand. However, as this is a secondary connection, the melting temperatures are averaged, rather than substituted. For example, R1 and R3 are connected to the core region through their respective shell

strands. The resulting melting temperatures are 41.5°C and 47.6°C, respectively. Figure 26 presents the new melting temperatures for the tile motif model applying the core-shell and the shell-arm theories.

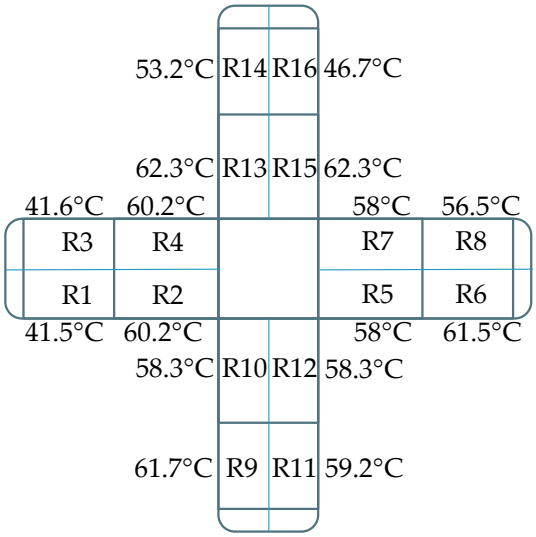


Figure 26: The melting temperature model based on core-shell and shell-arm melting temperature theories.

The resulting van't Hoff plot from the simulation using this model gives a transition that more closely matches the experimental data, as shown in Figure 27.

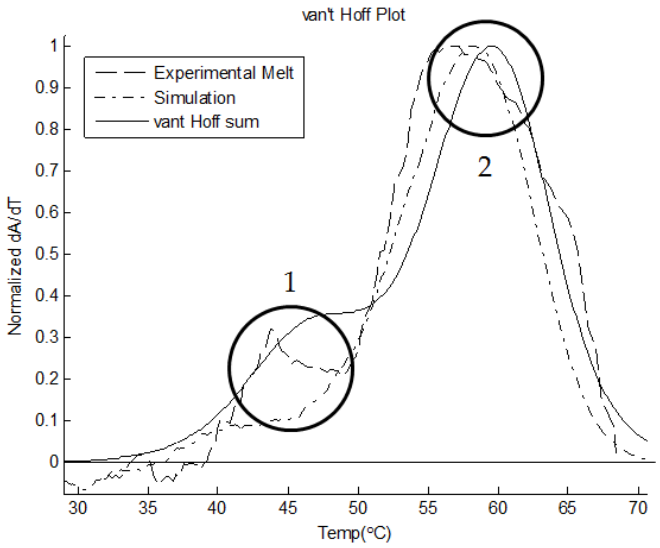


Figure 27: The new model simulation against experimental data.

The transitions below the noise threshold (i.e., 30°C - 40°C) indicate stabilization is occurring in the structure, as indicated by the negative van't Hoff values in this temperature range for both the simulation and the experimental results. The observation indicates that the simulator can be used to capture some aspects of stabilization. Furthermore, there are no melting temperatures in the modified model that are within this temperature range as shown in Figure 26. However, the analysis here focuses on the transition points for destabilization as the figures of merit. The comparison of the thermodynamic parameters extracted for the two transitions circled in Figure 27 are within 15% of each other in both cases. Table 9 presents the results of the analysis.

Table 9: Thermodynamic parameter comparison between experimental and simulation data.

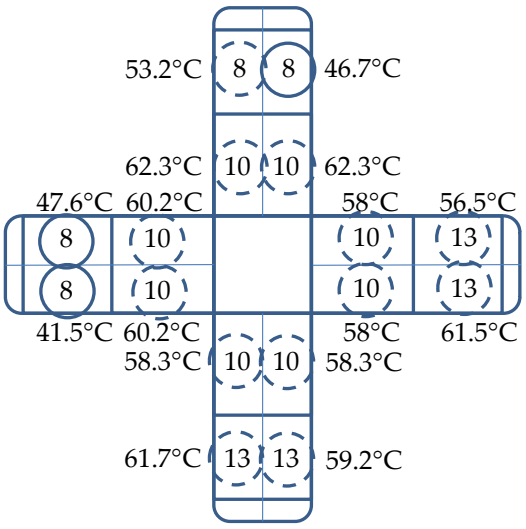
Transition 1	Extracted	Experiment	% Error
Tm (°C)	43.196	44.942	3.89%
ΔH (kcal/mol)	79.841	92.423	13.61%
ΔG (25 °C kcal/mol)	-4.8753	-6.185	21.18%
ΔS (cal/degree mol)	267.92	310.14	13.61%

Transition 2	Extracted	Experiment	% Error
Tm (°C)	57.835	56.531	-2.31%
ΔH (kcal/mol)	74.473	75.324	1.13%
ΔG (25 °C kcal/mol)	-8.2059	-7.9698	-2.96%
ΔS (cal/degree mol)	249.91	252.76	1.13%

The tile motif model is in fair agreement with the experimental data obtained using the DBS. The main melting transition thermodynamic parameters at 56°C are within 3% of the experimental data, with the thermodynamic parameters at 45°C within

22% of the experimental data. The thermodynamic parameters from the data can offer insight into the structural stability of the tile motif. High enthalpy values indicate abrupt transitions in the structure of thermodynamic regions with a high number of base pairs, implying rapid destabilization of a thermodynamic region. Low values result from a broad transition due to multiple thermodynamic regions destabilizing over a small temperature range or a single thermodynamic region with a very low number of base pairs. The enthalpy values are used to calculate the free energy and entropy values at room temperature. A lower free energy value indicates increased stability in the structure as it is more favorable. The free energy for Transition 2 in the melting data is the lower one, resulting in a lower entropy value, indicating more stability. This is a reasonable conclusion considering that the denaturation process is one that is favorable during the change in temperature, meaning the transition from dsDNA to ssDNA is the favored process. The magnitude of the transitions in the data indicate that the tile motif is more resistant to denaturing in the 40°C - 50°C range than in the 50°C - 60°C range.

The general thermodynamic behavior can be initially predicted by looking at the regions affected for each temperature transition range based on the experimental data as shown in Figure 28.



Temp (°C)	bp	# Regions
40 - 50	24	3
51 - 65	148	13

Figure 28: Tile diagram with possible cooperative regions and base pairs.

The base pairs in the figure are identified with their transition temperatures. Three regions with a solid circular border around the number of base pairs give a total of twenty-four base pairs with transition temperatures ranging from 40°C - 50°C. Thirteen regions with the dotted circular border indicate that one hundred forty-eight base pairs have temperatures ranging from 50°C - 65°C. Comparing the magnitudes of the normalized transitions in Figure 27, the ratio of the number of thermodynamic regions denaturing in the 40°C - 50°C range versus the 50°C - 65°C range is approximately 1:5 for the experimental data and 2:5 for the van't Hoff sum data. The ratio for the number of thermodynamic regions present in the tile model in Figure 28 is 1:4. In terms of the total number of thermodynamic regions needed to achieve these ratios, this implies that it is possible to predict the number of thermodynamic regions that denature over a given temperature range based on the relative magnitude of the van't Hoff peaks. However, this initial prediction only analyzes the nanostructure using the thermodynamic regions alone. The interactions and cooperativity between the

regions that are facilitated using the crossover and core models in the tile motif, as well as the sticky end models that are included in the grid nanostructure models presented in Sections 3.3.3.6 and 3.3.3.7, are not included in the analysis. The comparison between the experimental and the simulation data in this section indicates that the tile motif model can characterize DNA nanostructures using the core-shell and shell-arm melting temperature theories. A third theory concerning the interaction between the arm and sticky end regions is presented later for the DNA grid nanostructure models.

3.3.3.5 Tile Deflection Test

While the simulation data can be processed to compare the thermodynamic properties to experimental data, the deflection data obtained from the melting trajectories also indicate that neighboring thermodynamic regions affect the physical response of a region during the denaturing process. The deflection ratio of the upper right thermodynamic region in the bottom arm in the tile motif is recorded during the simulation of the tile motif. The deflection ratio is plotted against the temperature in Figure 29.

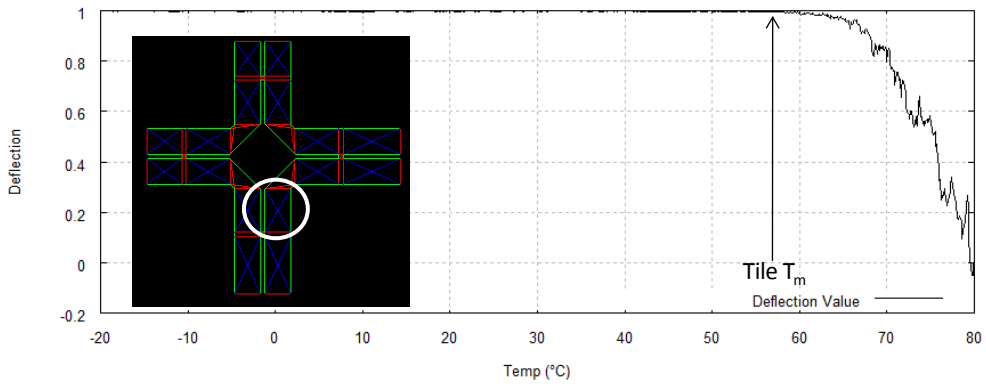


Figure 29: The deflection ratio data plotted against temperature for the upper right thermodynamic region of the bottom arm of the tile motif.

The melting temperature for the segment is 58°C, close to the melting temperature of the overall structure of the tile motif. The deflection value does not decrease before the entire structure has melted. This observation suggests that although the individual structures can denature, the physical stability, and thus the optical absorbance, is not significantly affected until a majority of the regions have denatured. This hypothesis is intuitively reasonable, but must be validated in future work.

3.3.3.6 The 2 × 2 Tile Model

The 2 × 2 tile model is composed of four tile motif models. However, the orientation of the tile motif is different in each position. Using the initial tile motif as a reference, the motif is flipped along the x-axis in the position to the right. It is flipped in along the y-axis in the position below the initial motif. Finally, the motif is flipped along both axes in the bottom right position. The sticky ends that are complementary hold the structure together, represented by four DNA or "S" springs with eight "C" springs supporting them. Here, the arm-arm theory is introduced for the DNA grid nanostructures, as it was not applicable for the tile motif. The theory is based on the association of the sticky ends with the shell through the arm strand and is only applicable to sticky ends that connect neighboring tiles together. The melting temperatures of these springs are the average of the sum of the melting temperatures of the sticky ends with the two thermodynamic regions joined by the sticky end. For example, between adjacent Tiles 1 and 2, the sticky ends connecting R8 and R6 of Tile 1 and R1 and R3 of Tile 2 are a total of 26°C. However, they are averaged with the melting temperatures for R1 and R8, and R3 and R6, respectively, resulting in melting

temperatures of 42.5°C and 45°C. The melting temperatures for the sticky ends for the 2 × 2 tile DNA nanostructure model are labeled in Figure 30a. Tiles 1 through 4 are labeled "T1" to "T4".

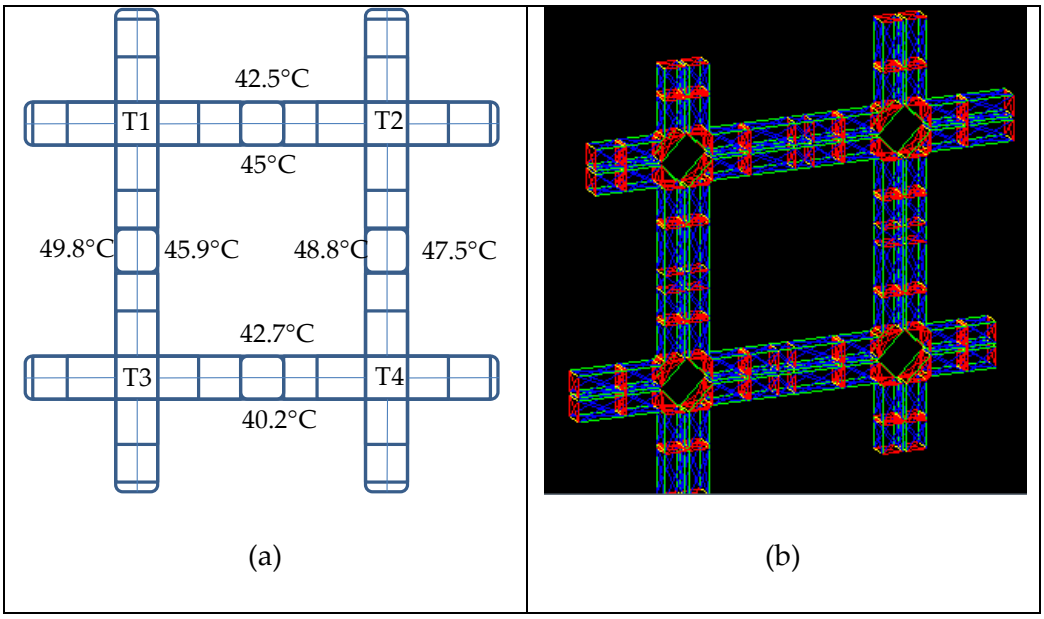


Figure 30: The (a) 2 × 2 tile model with labeled sticky end melting temperatures calculated using the arm-arm theory, and (b) a screen capture of the model using the graphical output program.

The four tile model is simulated over a 5 ns time period at a constant temperature of 298 K for 1000 trajectories. The diffusion coefficient is determined using the same methods as described in Section 3.2.3 to give a result of 1.8508e-12 m^2/s. The value is within 1% of the extrapolated diffusion coefficient of a 1138 base pair dsDNA molecule using Equation 34. There are 696 base pairs represented in this DNA nanostructure, indicating a substantial difference in this result compared with the extrapolation. The decreased diffusivity is due to the increasing number of crossover, core, and sticky end regions that are included in the nanostructure model. While there are currently no reference values for diffusion coefficients of complex DNA self-

assembled nanostructures, this result will be useful as a reference for the study of diffusive properties of complex DNA nanostructures.

The 2 × 2 tile DNA grid nanostructure is simulated over a temperature range from -20°C to 80°C at 0.1°C increments. Each temperature step is simulated for 5 ns before incrementing. The van't Hoff plot of the simulation as well as the experimental data is presented in Figure 31.

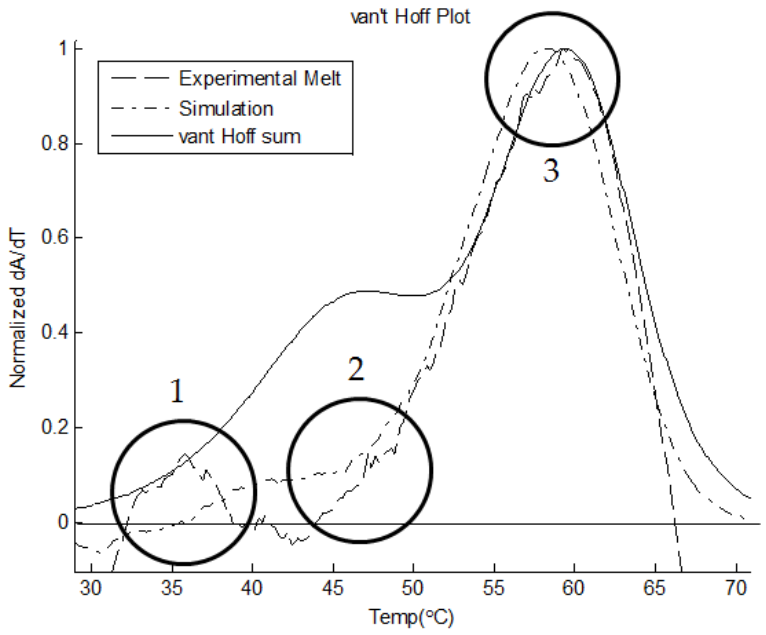


Figure 31: The van't Hoff plot of the 4 tile DNA nanostructure compared to the simulation data and the van't Hoff sum.

There are three melting transitions that can be identified in the experimental data and the simulation data. The thermodynamic parameters extracted from these transitions are presented in Table 10.

Table 10: Comparison of simulation and experimental thermodynamic parameters for a 4 tile DNA nanostructure.

Transition 1	Extracted	Experiment	% Error
Tm (°C)	40.368	35.508	-12.73%
ΔH (kcal/mol)	144.19	165.1	12.67%
ΔG (25 °C kcal/mol)	-7.4362	-5.9876	-24.19%
ΔS (cal/degree mol)	483.87	554.02	12.66%

Transition 2	Extracted	Experiment	% Error
Tm (°C)	48.901	48.648	-0.52%
ΔH (kcal/mol)	45.647	153.84	70.33%
ΔG (25 °C kcal/mol)	-3.6611	-12.208	70.01%
ΔS (cal/degree mol)	153.18	516.24	70.33%

Transition 3	Extracted	Experiment	% Error
Tm (°C)	57.955	58.561	1.03%
ΔH (kcal/mol)	76.515	76.415	-0.13%
ΔG (25 °C kcal/mol)	-8.4616	-8.6059	1.68%
ΔS (cal/degree mol)	256.76	256.43	-0.13%

The table above indicates that Transition 2 at 49°C does not match as well (70% error) as the Transitions 1 and 3 at 40°C and 57°C (25 - 26% error), respectively, due to it being a fairly broad transition that is more observable in the simulation than the experimental data. The effect that the crossover and core structures have on the DNA nanostructures can be observed when comparing the van't Hoff sum to the simulation and experimental data. The van't Hoff sum does not take into account the effect that the crossover and core models have on the stability of the overall nanostructure because it is created by using the superposition of the thermodynamic regions, including the sticky ends. The presence of the sticky ends appears to have an impact on the simulation in terms of changing the magnitude of the absorbance, thereby changing the magnitude of

the van't Hoff plot. This aspect of modeling DNA nanostructures remains to be explored.

3.3.3.7 The 2 × 4 Tile Model

The 2 × 4 tile model is composed of two 2 × 2 tile models vertically connected using two pairs of sticky ends. As with the 2 × 2 tile model, the melting temperatures for these sticky ends apply the arm-arm theory and are the average of the sum of the melting temperatures for the individual dsDNA sticky ends with the thermodynamic regions that they join as shown in Figure 32.

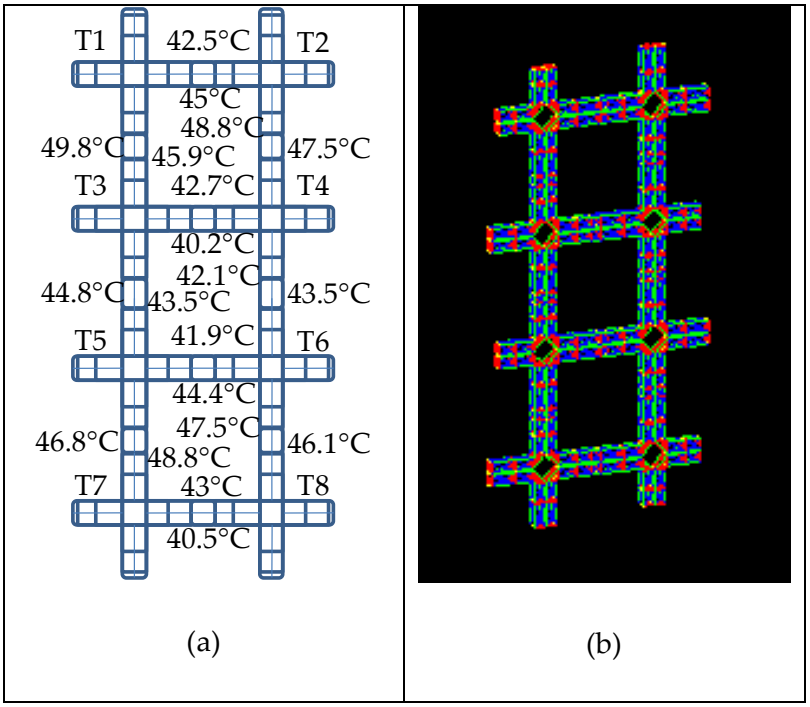


Figure 32: The 2 × 4 model.

The eight tile model is simulated over a 5 ns time period at a constant temperature of 298 K for 1000 trajectories. The diffusion coefficient is extracted using the same methods as described in Section 3.2.3 to give a result of 8.7848e-13 m^2/s. The value is within 2% of the extrapolated diffusion coefficient of a 3500 base pair dsDNA

molecule using Equation 34. Compared to the 1412 base pairs represented in this DNA nanostructure, the difference is again due to the presence of core, crossover, and sticky end structures. The trend is a reasonable observation when compared to the previous nanostructures. The ratio of the simulated diffusivity to the extrapolated diffusivity is 1.08 for the tile motif, 1.64 for the 2 x 2 nanostructure and 2.48 for the 2 x 4 nanostructure. The linear trend of the ratio, $R = 0.0012M + 0.8459$, where M is the number of base pairs in the nanostructure, indicates that as the size of the nanostructure increases, the diffusivity decreases based on the mass given a constant model density and configuration, i.e., the DNA grid nanostructure. Experimental data validating this data remains to be demonstrated. The van't Hoff sum, the simulation of the model, and the experimental data are presented in Figure 33.

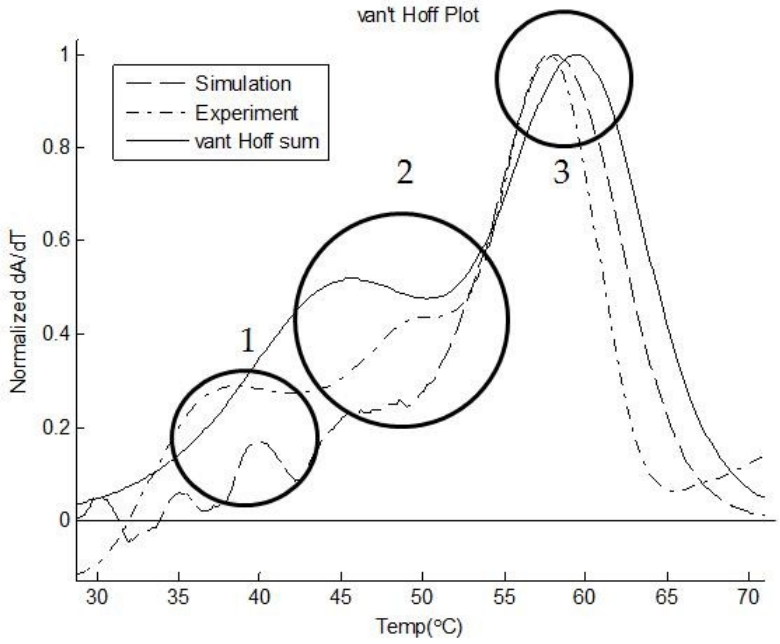


Figure 33: Comparing the 2 × 4 tile van't Hoff experimental data to simulation data.

For the case of the 2 × 4 DNA grid nanostructure, the van't Hoff sum is a better fit than the simulation data possibly due to the inclusion of the sticky ends in this structure. Compared to the previous nanostructure simulations, the sticky ends did not significantly account for the total number of thermodynamic regions in the model. In this case, there are twenty thermodynamic regions that are sticky ends; slightly more regions than there are per tile model. This trend may indicate a change the thermodynamic behavior for larger DNA grid nanostructure such as the 4 × 4 nanostructure and the 8 × 8 nanostructure, which remain to be explored.

The third transition in the simulation at 57°C matches well with the experimental data. Although the simulation does not match as well with Transitions 1 and 2 with the experimental transitions that are present at 36°C and 48°C, the thermodynamic parameters could be extracted. These parameters from the simulation are compared to the parameters extracted from the experimental data in Table 11.

Table 11: Comparison of simulation and experimental thermodynamic parameters for the simulation of the 2 × 4 tile DNA nanostructure.

Transition 1	Extracted	Experiment	% Error
Tm (°C)	40.112	35.808	-12.02%
ΔH (kcal/mol)	204.82	165.1	-24.06%
ΔG (25 °C kcal/mol)	-10.386	-5.9876	-73.46%
ΔS (cal/degree mol)	687.31	554.02	-24.06%

Transition 2	Extracted	Experiment	% Error
Tm (°C)	47.471	48.648	2.42%
ΔH (kcal/mol)	79.853	153.84	48.09%
ΔG (25 °C kcal/mol)	-6.0213	-12.208	50.68%
ΔS (cal/degree mol)	267.96	516.24	48.09%

Transition 3	Extracted	Experiment	% Error
Tm (°C)	57.944	58.561	1.05%
ΔH (kcal/mol)	80.061	76.415	-4.77%
ΔG (25 °C kcal/mol)	-8.8507	-8.6059	-2.84%
ΔS (cal/degree mol)	268.66	256.43	-4.77%

From the table above, the transition at 58°C for the simulation data is within 5% of the experimental data. At 40°C and 48°C, the parameters are within 25% and 50% of the experimental data, respectively. The errors are due to differences in the widths of the transitions in the simulation data in comparison to the experimental data. The parameters for the main transition closely agree with the experimental parameters because the enthalpy and the free energy calculations are dependent on the width of the transition. While the widths of two of the transitions are not well matched, the melting temperatures are in fair agreement with experimental data.

3.3.4 Limitations of the Simulator

The DNA-STRAIN simulator is a first attempt at characterizing DNA nanostructures by using spring dynamics to model the physical response in a changing environment. It is validated using short DNA strands that are below the persistence length of dsDNA. The simulator is not suitable for simulation of structures with a total contour length that are beyond this length. The simulation is limited in that results cannot be used for study of torsion or bending of very short strands because the four DNA springs in the 3D model function as one unit. The model cannot capture the transition from ssDNA to dsDNA due to the assumption of a unimolecular dissociation transition. The analysis in this chapter focused on destabilization due to the degradation

of the springs over the melting temperature also limits its ability to capture stabilization behavior in more complex structures. The study of the stabilization of the nanostructures for the range of temperatures below the melting transitions remains to be explored.

In conclusion, the DNA-STRAIN simulator is a Monte Carlo simulator that generates the ensemble responses of a simple harmonic oscillator representing dsDNA structures based on the WLC model. The validation of the simulator of the 3D dsDNA model is demonstrated through testing the elastic response (within 1.5% of the calculated response), the diffusivity of the model (within 5% of the values from related work), the length variation (within 10% as the mass of the model increased), the deflection of distal ends of a WLC model, and thermodynamic response (within 16% of related work). The dsDNA model was then used to construct the tile motif and simulated, resulting in thermodynamic values within 22% of experimental data for the worst parameter after implementing three thermodynamic interaction theories for the model. The tile motif model was ultimately used to construct the models of the 2×2 and 2×4 DNA grid nanostructures, both with the worst thermodynamic parameters within 75% of the experimental data. While there are a number of areas that remain to be experimentally proven and improvements to the model needed, the DNA-STRAIN simulator is a good first effort at developing a simulator that can capture the thermo-mechanical properties of DNA grid nanostructures. The next chapter focuses on the spectral analysis of these nanostructures.

4. Spectral Analysis of DNA Nanostructures

The optical absorbance properties of DNA can be determined using Beer's Law (i.e., $A = \varepsilon lc$) to calculate the absorbance, as illustrated in Figure 34.

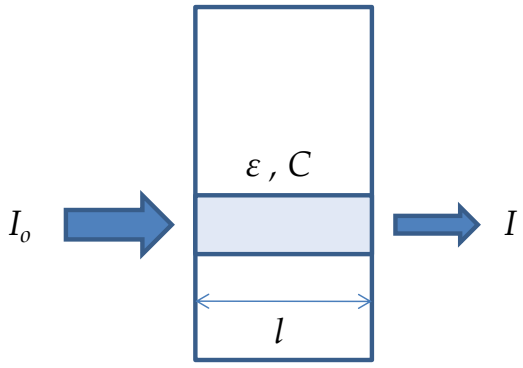


Figure 34: Diagram for illustrating Beer's Law.

The figure illustrates a cuvette of length l containing a sample of c concentration with an extinction coefficient of ε. A light source of intensity I_0 enters the cuvette from the left and leaves at an attenuated intensity, I. Beer's Law characterizes this attenuation in terms of optical density, OD. The absorbance data can be processed to extract thermodynamic parameters, identifying features such as the melting temperature and transition enthalpy of a sample, as described in Chapter 1 and implemented in Chapter 3. The utility of the DBS for modifying DNA grid nanostructures is demonstrated in Chapter 5. This chapter presents the design and validation of a custom dual-beam spectrofluorometer (DBS) instrument for studying the optical absorbance properties of DNA self-assembled grid nanostructures.

4.1 The Dual Beam Spectrofluorometer

There are several different types of instrumentation available to study the absorbance and fluorescence of molecules. Ultraviolet-visible (UV-Vis) spectrometers are useful for quantitatively determining absorbance properties, molar extinction coefficients, or the concentration of both organic and inorganic samples. The instrument is composed of a light source (e.g., a Tungsten filament, a deuterium arc lamp, a light emitting diode (LED), or a Xenon-arc lamp), a series of lenses for collimation, a prism or holographic grating for separating light into its constituent wavelengths, a sample chamber containing a transparent cell known as a cuvette, and a detector (e.g., a photodiode or a charge-coupled device (CCD)). Spectrofluorometers have the same components as UV-Vis instruments, but in a different configuration. Whereas UV-Vis spectrometers measure the light through the cuvette, spectrofluorometers measure the light that is scattered from the sample at an angle (e.g., 90°). These instruments are used to study the fluorescent properties of organic compounds (e.g., Fluorescein, Rhodamine, and Oregon Green) and their application as tracers for differentiation of tissue samples as well as tumors in biology, chemistry, and medicine. However, spectral information can be difficult to obtain in samples containing several different molecules with different physical and chemical properties.

An instrument must be capable of acquiring data over wavelengths ranging from UV to near infrared (NIR) for excitation or analysis to study the properties of DNA grid nanostructures as well as any materials functionalized to the strands. Ideally, the instrument has multiple monochromators, optical devices that reduce light into its

constituent wavelengths, converting radiation frequencies into a function of horizontal mechanical position. In this chapter, dual-beam systems refer to optical instrumentation that can select for different wavelengths independently. These systems typically consist of dual monochromator setups that include lens pairs for focusing and collimating laser light sources appropriate for the intended application. The following section presents a brief overview of some related work and applications of dual-beam monochromator systems before describing the dual-beam spectrofluorometer (DBS), a versatile custom dual-beam system.

4.1.1 Dual Beam Systems

Dual-beam monochromator systems are used for PCR reaction monitoring [88], spectral imaging [89], and trace chemical analysis [90], to name only a few examples of their utility. Figure 35 presents an example of a dual beam system as described in [89].

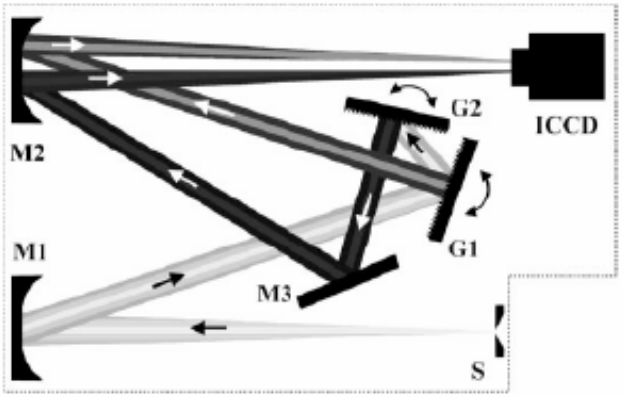


Figure 35: A dual beam system consisting of two gratings (G1 and G2) with a pair of collimating mirrors (M1 and M2) as well as an auxiliary mirror (M3). The light source originates from an entrance slit (S) and propagates through the system to an intensified CCD camera (ICCD).

The light source in the figure is reflected off a collimating mirror and diffracted by two gratings for dual wavelength selection. An auxiliary mirror reflects the second

diffracted beam to the second collimating mirror, and ultimately to a sample and detector. Different wavelengths can be selected to excite multiple donors simultaneously to study optical properties of a fluorescent molecule over a spectral range. To achieve this, the dual beam system must be capable of combining the wavelengths using beam splitters and beam combiners. This feature can also be used for noise correction originating from impurities in the solvent in real time or simultaneous excitation of two different fluorescent molecules. Previous work used such an approach by taking the difference in the signal intensity of a sample excited by two different wavelengths to effectively eliminate solvent background absorption to the signal, provided the analyte only absorbed one of the two selected wavelengths [90]. Using separate beams in a dual beam system allows independent acquisition of data from different fluorescent molecules in the same sample provided that 1) the excitation wavelengths do not significantly overlap and 2) the molecules do not physically react with each other [88]. However, overlap in excitation and emission wavelengths of different fluorescent molecules can be useful. For example, a dual-beam instrument can be useful for studying DNA scaffolds containing multiple combinations of fluorescent molecules that are placed in close proximity to facility fluorescence resonance energy transfer (FRET).

While the instruments described thus far are useful in terms of studying absorbance and fluorescence properties, they cannot be easily modified and augmented without the aid of the manufacturer. A custom-built dual beam system is more flexible in its configurability. Such an instrument, with the flexibility to select independent

wavelengths and switch between fluorescence and absorbance readings, as well as the potential to modify it for future applications, is ideal for the study of DNA self-assembled nanostructures and the impact that modifications have on their optical properties.

4.1.2 Description and Calibration of the DBS

The DBS is designed to have two monochromator systems capable of delivering a beam composed of two wavelengths to a sample and reference chamber. The instrument also features temperature control using a pair of water baths that heat and cool the two chambers. The construction, alignment process, and the schematics for the temperature control system of the instrument are provided in Appendix D. Figure 37 presents the design schematic of the DBS.

In the schematic, P_0 is the initial power measured at the collimated beam, P_{Sm} is the power measured at the beam coupler to the sample mount, and P_{Rf} is the power measured at the beam coupler to the reference mount. The attenuation of the power in the schematic is α_x, with x ranging from one to seven to indicate the possible points to measure the power along the beam path. The total power throughput to the sample and the reference mounts are expressed in Equation 37:

$$P_{Rf} = P_0\left[\alpha_2\alpha_4\alpha_{4_6} + \alpha_3\alpha_5\alpha_{5_6}\right]$$

$$P_{Sm} = P_0[\alpha_2\alpha_4\alpha_{4_7} + \alpha_3\alpha_5\alpha_{5_7}]$$

Equation 37

As the beam propagates through the instrument, Equation 37 expresses the power attenuation at each stage due to scattering, absorption, reflectance, and chromatic

aberration, a distortion which changes beam convergence as a function of wavelength. Power readings for each point in the schematic were taken for white light using a silicon photodetector. Approximately 55% of the power is lost in the system, with the holographic gratings attenuating the source power throughput the most as they split white light into its constituent wavelengths. The resulting power that reaches each leg of the system is balanced to approximately 90 μW per leg of the system.

To ensure consistent power balance from the alignment of the monochromators, a fluorescence standard can be used as a reference for calibrating the power. A sample of 90 μM Rhodamine 101 is excited using a 334 nm wavelength beam to verify that the alignment divides the power throughput evenly between the monochromators in the DBS before adjustments are applied. The spectra are acquired over a five second integration period averaged over three readings. The excitation and emission spectra for the standard are plotted in Figure 36.

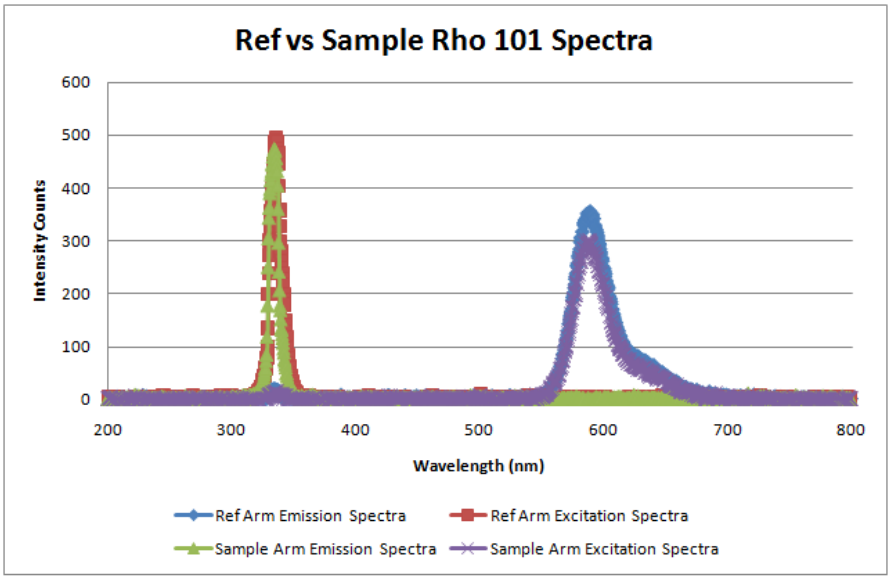
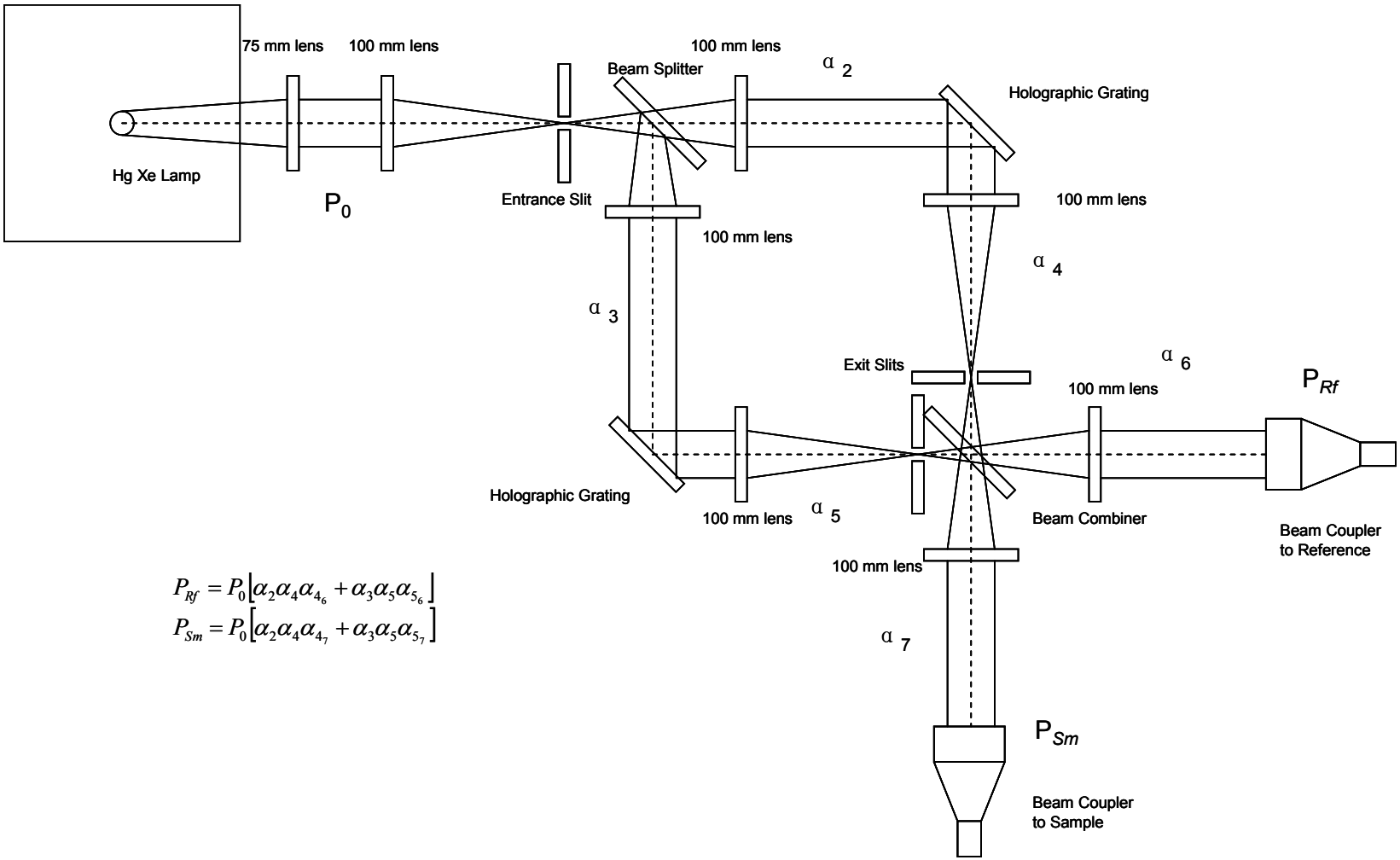


Figure 36: Excitation and emission spectra for Rhodamine 101 to measure the power throughput in the sample and reference chambers.



$$P_{Rf} = P_0\left[\alpha_2\alpha_4\alpha_{4_6} + \alpha_3\alpha_5\alpha_{5_6}\right]$$
$$P_{Sm} = P_0\left[\alpha_2\alpha_4\alpha_{4_7} + \alpha_3\alpha_5\alpha_{5_7}\right]$$

Figure 37: DBS schematic with power calibration equations.

The spectra in Figure 36 indicates that the alignment of the DBS is fairly balanced, with a 2% difference between the emission spectra measured at the 588 nm emission peaks for Rhodamine 101 in the sample and reference chambers. The results verify that the monochromators are aligned such that the power throughput is evenly distributed.

A fluorescence standard can also be used as a reference for checking the power throughput for a bandpass filter or a holographic grating. A 9 mM sample of Fluorescein is used as a standard for checking the power throughput of the DBS. The sample is excited using spectra in the range of 488 ± 10 nm using a bandpass filter, and at 365 nm and 314 nm using the holographic gratings. Acquisition of the spectra is done over a five second integration period with five periods averaged. Irises attenuate the intensity of the light through the sample chambers for maximum throughput in transmission mode while the spectral data is acquired in the fluorescence mode (i.e., at a 90° angle). Figure 38 shows the resulting output spectra normalized to the data from [91].

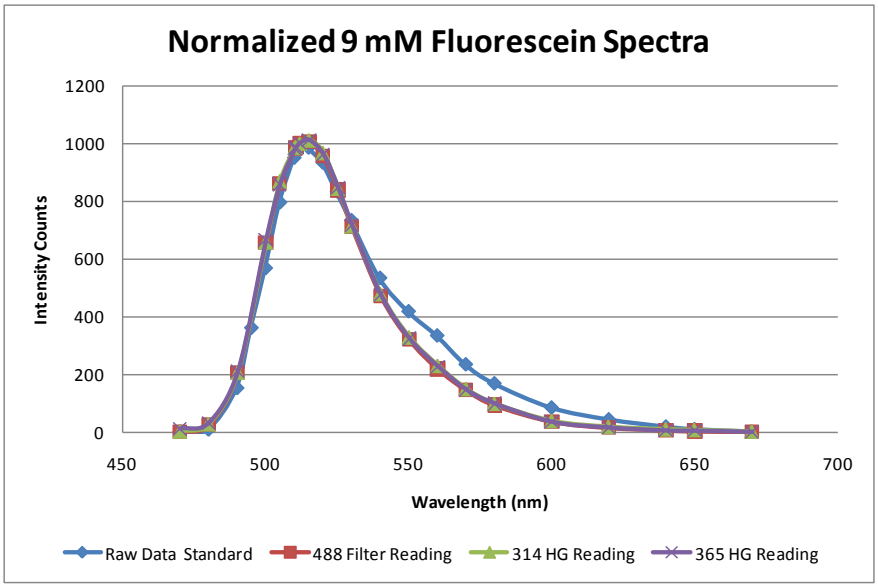


Figure 38: Normalized spectra for 9 mM Fluorescein sample from filter and holographic grating excitation at 488 nm, 314 nm, and 365 nm.

The resulting output spectra demonstrates that the DBS is capable of acquiring data for a fluorescence standard using either bandpass filters in the monochromator system, or the holographic gratings to select specific excitation wavelengths. The settings for the data acquisition can also be used to calibrate the power throughput of the DBS for diagnostic purposes.

4.1.3 DBS Capabilities and Limitations

From the initial schematic, other optical components can be used to augment or modify the instrument. Shutters, irises, neutral density filters, and filter wheels can be used to modify the beam output for each monochromator. Wavelength filters, neutral density filters, and irises can be placed towards the front of the design (near the light source) or towards the end of the beam path (near the beam couplers) for adjusting the excitation bandwidth or attenuation of the light source. Electronic shutters can be added

to select the combination of wavelengths or to isolate different monochromator legs. The configuration for the DBS can be a combination of fluorescence or transmission modes depending on the nature of the experiment whether it be FRET measurements, absorbance measurements, bleaching experiments, optical gate readings, or biological sensing on DNA scaffolding. In addition to these features, denaturation and annealing studies can be conducted.

While the DBS is a versatile instrument that can be modified for a variety of experiments, there are limitations in its operation. Beam collimation is not perfect because the lenses are slightly chromatic in shape, and this aberration propagates through the system. The mechanical inaccuracy in the rotational translation of the motorized mounts causes some inaccuracy in wavelength selection and requires manual correction. The light source generates a large amount of heat in its vicinity. Over time, the components may move by a minuscule amount due to thermal expansion, affecting the alignment. The 10 nm FWHM is the optimum width that can be achieved for the wavelengths in the monochromator after alignment. In comparison, commercial systems feature FWHM's as small as 0.2 nm using two gratings for one wavelength. However, the level of resolution required in the excitation wavelength of most fluorophores possess a broader FWHM (>50 nm) than the DBS. The temperature control is sensitive to blockage from material that deposits in the water system. A thermocouple currently monitors the sample chambers. A closed-loop temperature reading of the sample chamber during a temperature ramp, followed by an open loop ramp, is presented in Figure 39.

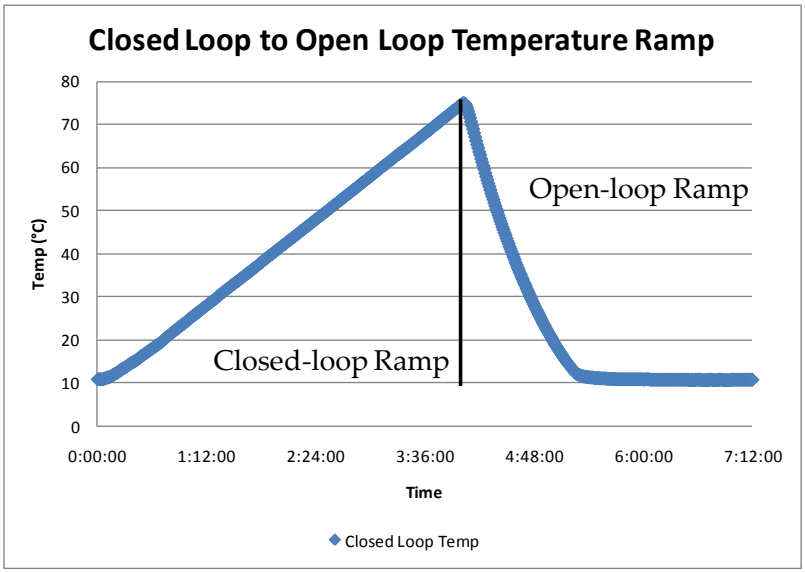


Figure 39: Closed loop temperature ramp followed by an open loop temperature ramp.

The temperature readings were taken over an eight-hour period in which the temperature control was regulated to ramp steadily at a rate of ~0.26°C/min from 11.5°C to 75°C over a four-hour period before being allowed to return to 11.5°C without regulation. The closed-loop process is evident in the linear slope from 0:00 to 4:00, while the open-loop process is evident in the curved slope from 4:00 to 5:25.

In conclusion, the DBS is a versatile dual beam monochromator system that features ease of modification and power throughput adjustment for separate legs in the optical system. It does not possess the wavelength resolution of commercial instruments due to inherent grating and lens design but is very sensitive in terms of its measurement capability. This sensitivity is sufficient to measure DNA at nanomolar concentrations. The range of power that the DBS can deliver to the cuvette sample chambers ranges from the nW to $\mu W/cm^2$, sufficient for measuring fluorescence with a short integration

time. Regular realignment as well as fluorescence and absorbance measurements using chromophores such as Fluorescein is done for calibration and maintenance of the instrument. Physical changes due to heating of the cuvette chambers can be corrected using numerical methods as described in Appendix G.

5. Chemical Cross-linking of DNA Nanostructures

DNA self-assembled nanostructures must withstand downstream fabrication procedures (i.e., functionalization of active components) without denaturing if they are to be used as scaffolds for nanodevices. A procedure for intercalating, and subsequently cross-linking, trimethylpsoralen (TMP) into the DNA grid nanostructure is presented to improve its stability. The QSA metric and the DBS, described in Chapters 2 and 4, respectively, are used to monitor the progress of the process. Spectral data from the DBS is compared to related work [92] that implements a similar procedure, and the QSA metric quantitatively determines the impact of the process on the DNA nanostructure with AFM images. The modified DNA grid sample is then subjected to multiple heating and annealing cycles in the DBS to demonstrate the improved resilience of the DNA nanostructure. Following this experiment, another denaturation experiment is conducted for multiple samples subject to a single temperature ramp process. The resulting data is compared with the simulation results of the model implemented from the DNA-STRAIN simulator, demonstrating the capability of the simulator to capture the effects of modifying the DNA grid nanostructure.

5.1 The Psoralen Intercalator

Psoralen is a DNA intercalator used in medicine to prevent dsDNA from replicating, effectively halting cellular division processes. Cross-linkers, such as psoralen derivatives [93-98] and DNA adducts [99-100], are well-documented in their chemical interactions with DNA. Psoralen cross-linkers intercalate into the helical

structure and covalently bond with nucleotides via ultraviolet (UV) light or gamma ray exposure. This section gives a general overview of the related work on DNA-psoralen cross-linking procedures.

DNA cross-linking procedures have been used in academic research for studies of the elastic response of overstretched DNA [101] as well as studies of psoralen-DNA interaction in the absence of light [94]. Thermal properties of and reactivity of psoralen cross-links with with specific oligonucleotide sequences have also been conducted to characterize the chemical reaction [96, 102]. These studies have lead to the photoligation of DNA nanostructures using such cross-linkers as deoxyuridine [103] to improve its resistance to denaturation at high temperatures. However, deoxyuridine only cross-links with neighboring bases on the same ssDNA strand, while TMP and 8-methoxypsoralen (8-MOP) are intercalators that form cross-links with bases on opposite strands. Figure 40 presents diagrams of the TMP and 8-MOP psoralen molecules [98].

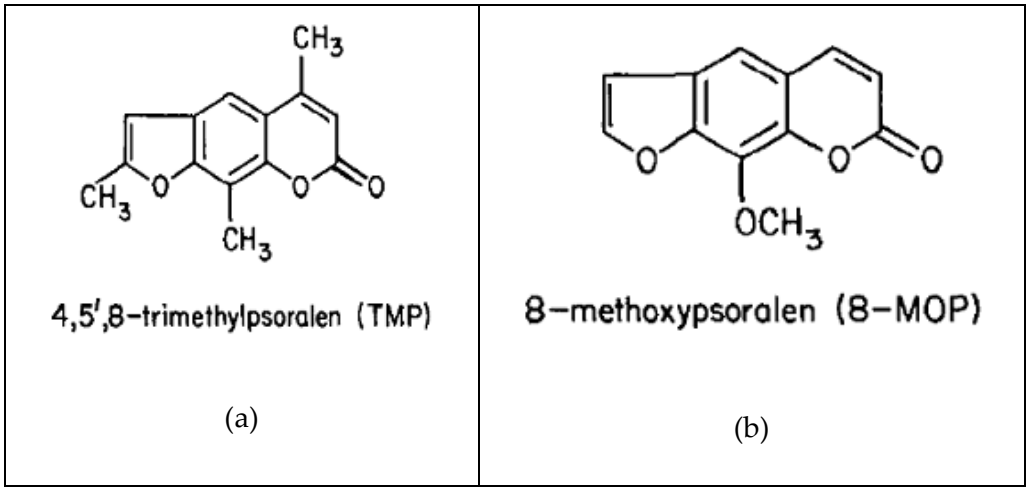


Figure 40: (a) The TMP and (b) 8-MOP psoralen molecules [98].

TMP was selected to study cross-linking of the DNA grid because the molecule is more effective at cross-linking DNA than 8-MOP [104]. The cross-linking process occurs

in a two-step reaction: 1) initial binding to a single thymine base that requires no energy input, forming the monoadduct structure; and 2) a covalent bond forming with the second thymine base on the opposite strand that requires exposure to 365 nm UV light, forming the diadduct structure. Figure 41 shows a 3D model of the diadduct structure from [105].

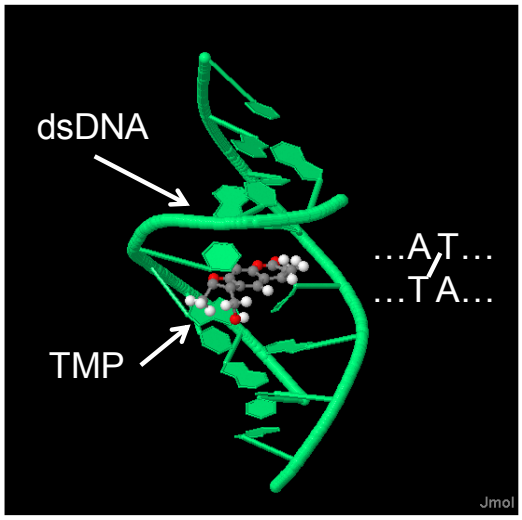


Figure 41: A 3D Model of TMP intercalation in a DNA double helix structure [105].

As shown in Figure 41, the TMP molecule only binds to alternating AT base pairs because a covalent bond must form between two thymine bases diagonally across from each other. The effectiveness of the cross-linking has been observed to increase with continuous AT base pair sequences [96]. Introduction of TMP into the DNA structure causes the helix to slightly unwind due to the size of the intercalator molecule being equivalent to an extra base in the sequence [106]. Figure 42 indicates the potential cross-link sites (i.e., AT/TA pairs) identified in the DNA tile motif nanostructure.

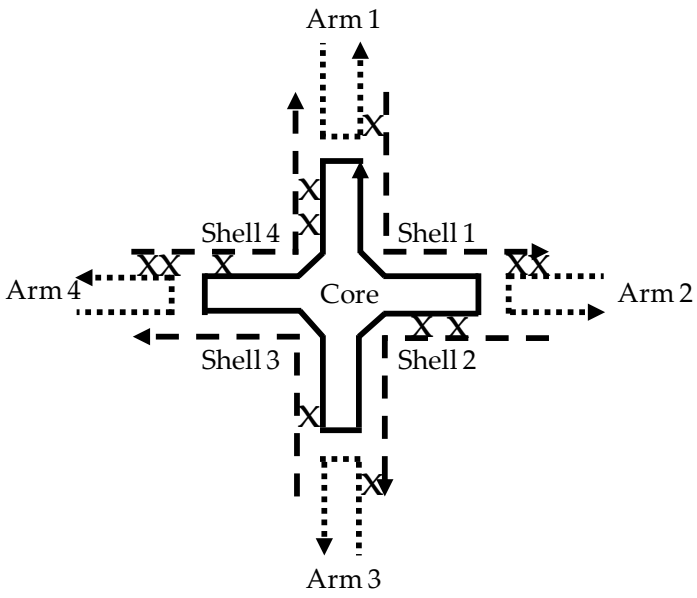


Figure 42: The DNA tile motif model with all twelve possible cross-linking sites (i.e., AT/TA sequences) identified.

There are twelve possible sites that exist in the DNA tile motif, and at least one cross-link is present among each of the three types of ssDNA oligonucleotide interactions that compose the motif (core-shell, shell-arm, arm-arm). Assuming an activity factor of 100% reactivity between TMP and all twelve available sites, justified in Appendix H, the physical configuration of the DNA nanostructure will change due to the presence of these intercalators.

The intercalation process can be performed and verified in the DBS by using the instrument to activate the cross-linking process, and to obtain spectral data of the TMP molecule before and after the reaction. The following sections describe the procedures for intercalating TMP into a 2×4 DNA nanostructure adapted from a similar procedure for cross-linking DNA nucleo-cages with the TMP intercalator [92]. Following the procedure, an experiment to study the effect of the process on the optical properties of

the DNA is conducted by heating and reannealing the structure four times in succession. In addition to the procedure, a model of the 2 × 4 DNA grid nanostructure is designed using the model generator described in Appendix B.4. The model is simulated and analyzed to compare the predicted behavior from the simulator to the experimental data. The results indicate that there is a trade-off between the increased resilience from the reaction and the physical configuration due to the number of intercalation sites present on the DNA nanostructure.

5.2 Cross-linking the 8 Tile DNA Nanostructure

5.2.1 The Cross-linking Procedure

A sample of 2 × 4 DNA nanostructures is annealed and cross-linked with TMP using a cross-linking procedure adapted from those found in biochemistry and medicine [97-98, 102, 107-110]. All cross-linking and data acquisition of the sample is done in the DBS. The details of the process are given in Appendix E. The unmodified nanostructure and the cross-linked sample were imaged using an Agilent 5100 AFM for quantitative analysis using the QSA metric.

5.2.2 Verification of Successful Cross-linking of DNA Nanostructures

Figure 43 presents the spectral data for 330 nm and 365 nm excitation over the incubation and irradiation process from the DBS and compares it to the spectra of DNA nucleo-cages from [92]. The two data sets have similar features, verifying the cross-linking of the 2 × 4 DNA nanostructure. The monoadduct structures have an emission peak at 420 nm when excited at 330 nm. After irradiation of the solution at 365 nm, a

new peak appears at 390 nm and the intensity of the 420 nm peak decreases significantly, indicating a change in the physical structure of TMP due to successful intercalation into the DNA nanostructure.

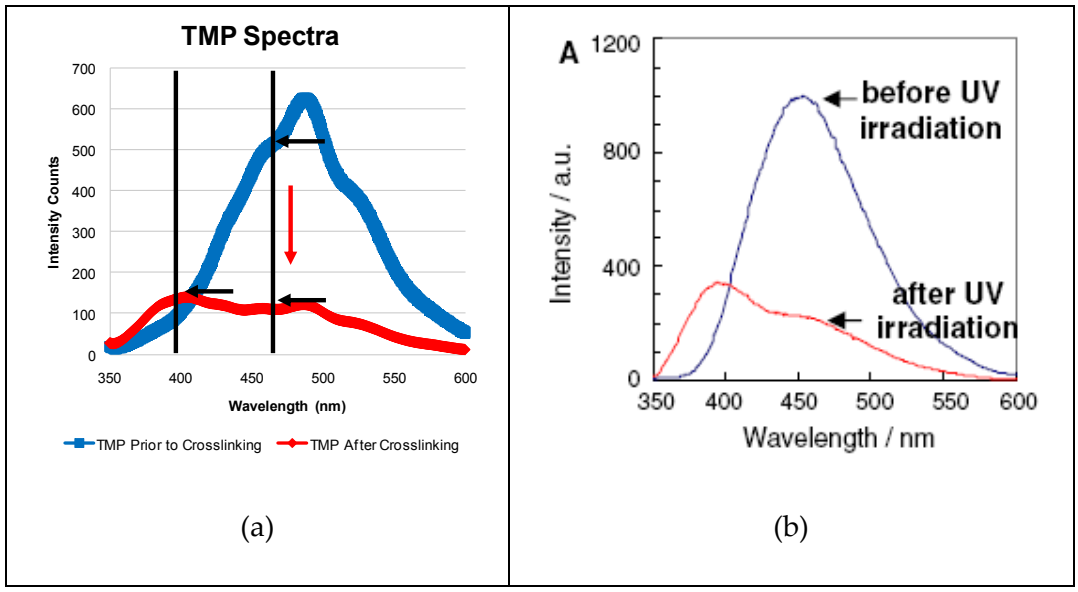


Figure 43: (a) Comparison of 450 nm excitation spectra before and after cross-linking of 10 μM TMP to the DNA nanostructure compared to (b) spectra of TMP before and after cross-linking to DNA nucleo-cages in [92].

The emission spectra in Figure 43 agree well with the emission spectra from previous work, showing an 80% attenuation in intensity counts at 420 nm for both cases as well as the appearance of the peak at 390 nm upon successful cross-linking that is proportionally larger than the peak at 420 nm. The peaks present at 490 nm and 525 nm are possibly due to the broader excitation spectrum of the DBS excitation wavelengths that could contribute to the additional fluorescence peaks detected by the spectrometer, or due to extraneous wavelengths that were not filtered from the beam. These wavelengths do not affect the overall cross-linking process, as the process can only be reversed by exposing the sample to excitation below 300 nm [111].

The success of a cross-linking procedure adapted from those used in biology and medicine has been demonstrated by using TMP to physically modify the DNA nanostructure as a possible approach for increasing the resilience of DNA in anticipation of future functionalization processes. The following sections describe the experiments to study the modified DNA nanostructure using the QSA metric and the DBS.

5.2.3 Characterizing Modified DNA Nanostructures

The QSA metric was applied to four AFM images of the DNA nanostructures before (i.e., no TMP present in the sample) and after the cross-linking process to determine the effect the process had on the physical configuration of the DNA grid nanostructure, as shown in Figure 44.

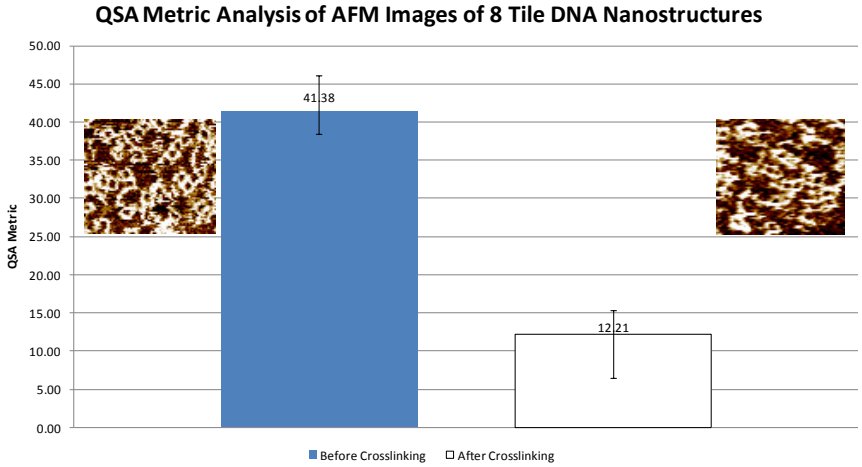


Figure 44: QSA metric analysis of uncross-linked vs. cross-linked DNA nanostructures for four images.

The results indicate that the TMP cross-linking process changes the physical configuration of the structure, resulting in a lower QSA metric. The resulting value is possibly due to the size of the TMP molecule straining the structure during the intercalation process, as successful intercalation of each molecule is the equivalent of

adding an extra base pair to the dsDNA sequence. Although the two samples are qualitatively similar in image quality (i.e., the DNA grids are observable and seemingly well-formed), the QSA metric analysis quantitatively indicates that there are physical changes affecting the DNA nanostructure.

After verifying successful cross-linking of the sample, a series of experiments were conducted to study the effects of heating the cross-linked grid structure. The absorbance of the uncross-linked DNA grid nanostructure sample was recorded in a controlled denaturing and reannealing process over a 23°C to 73°C temperature range. The cross-linked nanostructures were also given the same treatment.

The structure is hypothesized to reanneal more completely during the cooling/reannealing phase, resulting in a final absorbance that is close to the initial absorbance value because complete dissociation does not occur. The cross-linked tile motif denatures in the uncross-linked regions but retains its proximity to the complimentary strand after denaturing. The hypothetical, loosely denatured structure held together by the covalent bonds of the cross-linked sites is referred to as the "rat's nest" structure. Reannealing of the tile motif from this structure is predicted to be more complete, but not in the same configuration as the initial cross-linked structure due to the increased strain on the structure introduced by the intercalators. This hypothesis is tested in a heating procedure using a second set of control and cross-linked samples. The spectra data acquired from the DBS were processed to determine the absorbance values for the samples. Details concerning the procedure are described in Appendix F. The QSA metric is also used to analyze the AFM images of the samples.

Both the uncross-linked control and the cross-linked sample are subjected to a temperature ramp from 23°C to 90°C four times in succession to study the effects of repeatedly heating and reannealing the structures. The results of the experiments are shown in Figure 45.

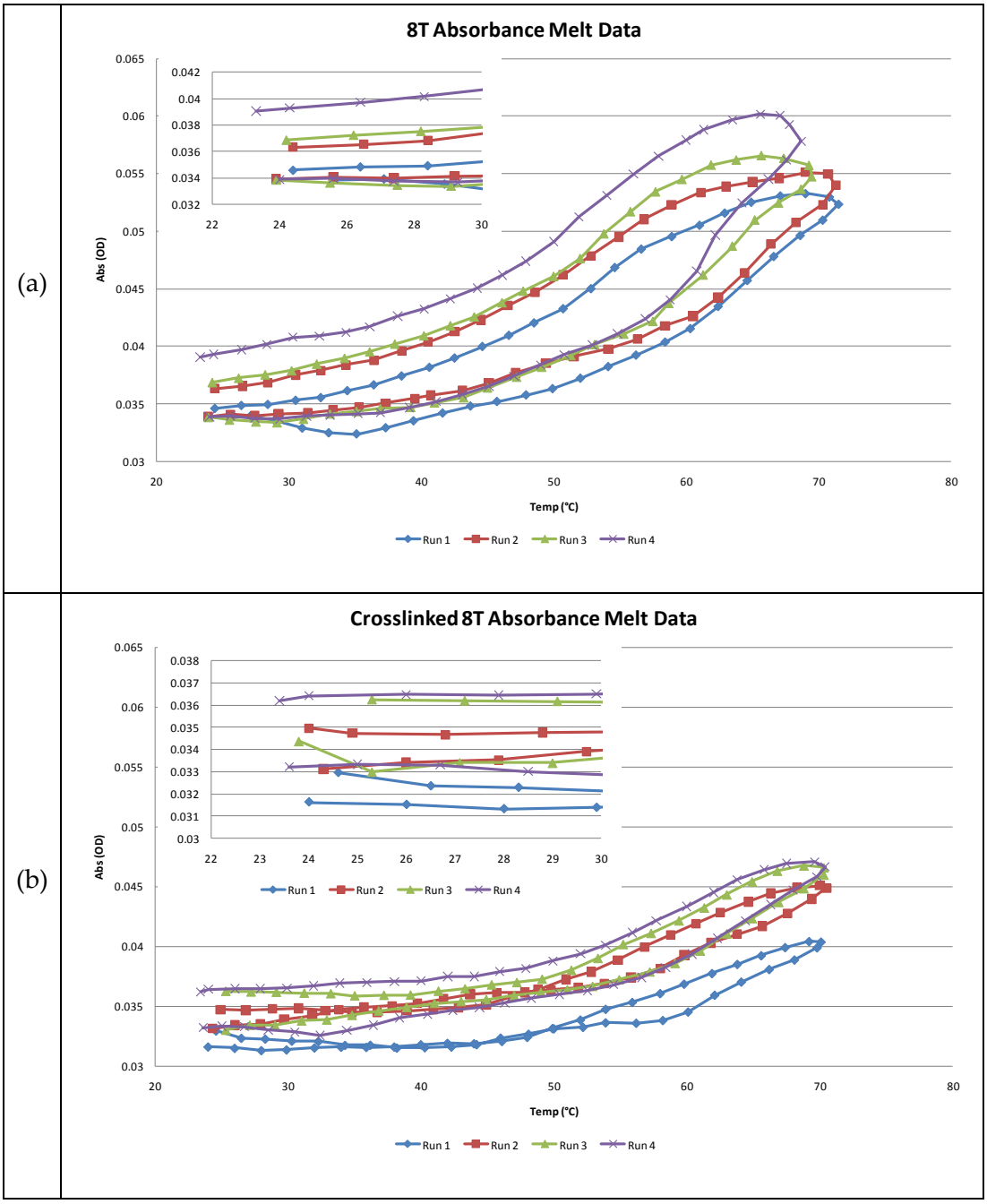


Figure 45: Hysteresis curves for (a) the control and (b) the cross-linked sample.

Evaporation of the solvent is only slowed using the paraffin as a vapor barrier over the duration of the experiment, and is corrected for in the plots by offsetting the absorbance readings for each temperature ramp to the first temperature ramp process. The insets show the differences in the control and the cross-linked samples at the beginning and the end of each temperature ramp. The uncross-linked DNA nanostructures that are subjected to the heating process are partially reannealed due to complete dissociation of the DNA scaffold nanostructures, as indicated by the 50% change in absorbance values at the maximum temperature. AFM images indicate the deposited material is composed of denatured DNA nanostructure fragments. The difference in absorbance values as the temperature returns to 23°C during the reannealing process indicates that the structure is more diffuse over this range.

The overall change in the absorbance for the cross-linked sample is lower than that of the control as it only increases by 30%, suggesting that the cross-links prevent changes in absorbance due to the covalent bonds that prevent the strands from significantly denaturing. Furthermore, the reannealing absorbance values are within 12% of the denaturation values throughout the process as opposed to 30% for the uncrosslinked sample, indicating that the cross-links prevent significant denaturing of the structure. However, the absorbance values are still offset after each successive temperature ramp with the correction for evaporation, suggesting that the structure does not return to its original configuration due to strain from the cross-links. The QSA metric is used to analyze the AFM images from the two samples, are shown in Figure 46.

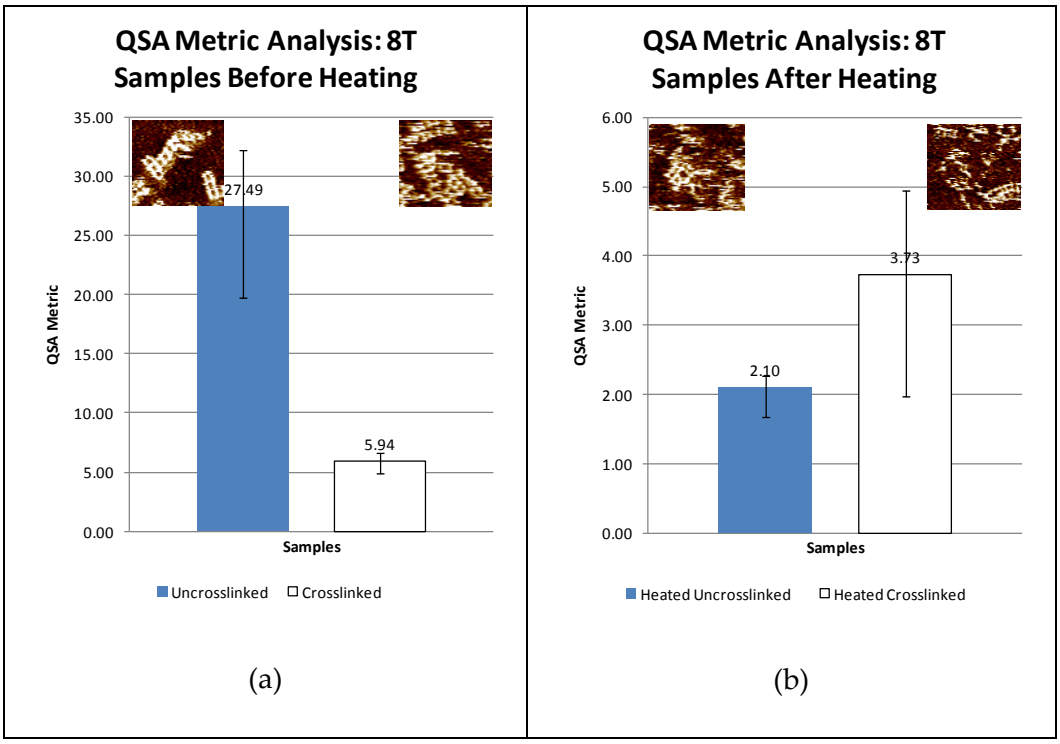


Figure 46: QSA metric analysis for (a) unheated 8T samples and (b) heated 8T samples

The QSA analysis indicates a 92% decrease in the uncross-linked control sample compared to the 37% decrease in the cross-linked control sample, indicating that the cross-linking process does have a relatively stabilizing effect on the structure. However, the QSA metric analysis alone is not a conclusive indicator of the effect of the heating process on the samples, and the magnitude of the initial and final metrics for the samples must be considered. The magnitude of the decrease in the metric for the uncross-linked control sample indicates that there are very few structures present in the AFM images after heating. Comparing the metric values for the cross-linked sample, a similar conclusion can also be made, indicating that physical changes in the structure were less dramatic than in the control.

From comparing the difference in the reannealing absorbance values to the denaturation absorbance values to the absorbance range at the minimum and maximum temperatures, cross-linking does demonstrate that it affects the optical absorbance properties, indicating significant effects on the structure. The QSA analysis agrees with the absorbance analysis. However, this is not sufficient data to determine the exact configuration of the structure during the denaturation and reannealing process.

Thermodynamic analysis of the denaturing data of the two samples was conducted using the MATLAB program described in Section 3.3.1.2. The van't Hoff plots are generated from the data after filtering them using a third order Savitsky-Golay filter with a 35-point window.

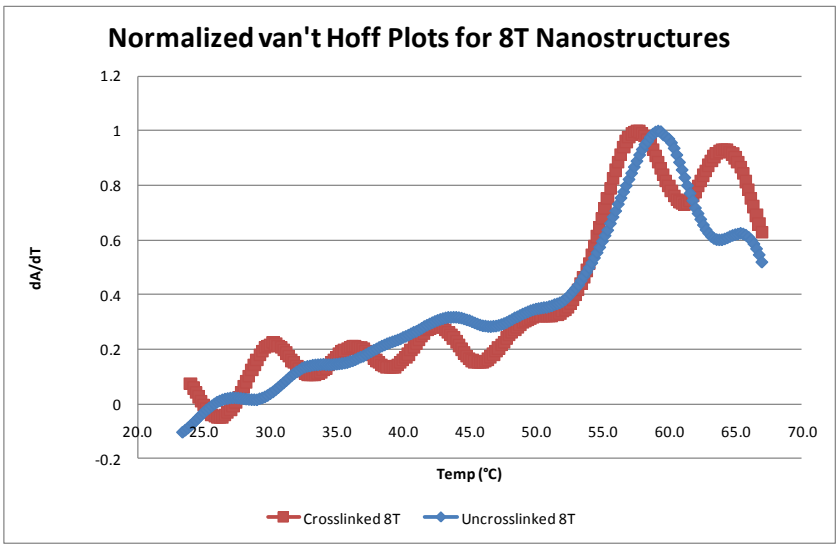


Figure 47: van't Hoff plots for the temperature ramp experiments for uncross-linked and cross-linked 8T DNA nanostructures.

The normalized van't Hoff plots for the two samples give further insight concerning how the cross-linking affects the structure. Table 12 presents the melting temperatures determined using the DNA-STRAIN thermodynamic parameter program.

Table 12: Melting temperatures for the uncross-linked and cross-linked samples from the heating experiment.

Uncross-linked T_m (°C)	Cross-linked T_m (°C)	Difference
35.821	32.482	9.32%
38.425	N/A	N/A
46.485	44.524	4.22%
54.491	53.225	2.32%
61.144	59.988	1.89%
67.11	66.001	1.65%

Thermodynamic parameters calculated indicate a shift in the melting transitions of the cross-linked sample compared to the uncross-linked sample ranging from 1% to 10%. While there are six transitions that can be qualitatively identified in the van't Hoff plots, the program could only identify five of them in the case of the uncross-linked sample due to the broad transition in the range of 30°C to 40°C. The data indicates that while the previous absorbance analysis shows that the denaturation process is incomplete for the cross-linked sample, the structure is under more strain due to the presence of the psoralen molecules in the structure, resulting in sharper transitions in the van't Hoff plot that indicate minor destabilization of the structure. In contrast, the uncross-linked DNA nanostructure has less distinct transitions without the psoralen present in the structure. Table 13 presents the enthalpy values for the two samples at each identified melting transition.

Table 13: Enthalpy values for the uncross-linked and cross-linked samples from the heating experiment.

Uncross-linked ΔH (kcal/mol)	Cross-linked ΔH (kcal/mol)	% Difference
121.36	176.05	-45.06%
111.34	N/A	N/A
42.895	120.51	-180.94%
41.828	72.399	-73.09%
70.249	86.621	-23.31%
95.327	90.404	5.16%

The data indicates that the differences between the two samples are much larger when comparing the enthalpy values, particularly in the lower temperature ranges. This result suggests that the structure is more stable in this range for the uncross-linked case, as the broad transitions indicate gradual denaturation of the structure. The cross-linked sample has much sharper transitions, narrowing the FWHM of the van't Hoff plot, resulting in enthalpy values up to three times as high as the uncross-linked case. This analysis of the two samples indicates that cross-linking is a possible strategy for improving the resilience of the structure. However, the location and number of cross-linking sites on the DNA nanostructure must be considered to strike a balance between the need for resilience in hostile environments in future fabrication processes, and the need for the structure to be stable in terms of retaining its physical configuration as a tile motif.

The experimental data indicates there is a decrease by up to 10% in the melting temperatures when the DNA grid nanostructure is modified by the intercalator molecule. The following section presents a 3D spring model representing the cross-

linked grid nanostructure that is simulated using the DNA-STRAIN simulator. The resulting analysis indicates that the simulator is potentially capable of aiding in the improvement of the design of the tile motif.

5.3 Simulation of the Cross-linked Nanostructure using DNA-STRAIN

This section presents the modifications needed for modeling the cross-linked DNA grid nanostructure. The model for the cross-linked tile motif model is first presented, followed by the full DNA grid nanostructure. The model is then simulated using the DNA-STRAIN Monte Carlo simulator, followed by an analysis of the trajectories using the thermodynamic parameter extractor described in Appendix B.6. Comparing the van't Hoff analysis to experimental data from the previous section indicates that the model can capture the effects of the intercalation process.

5.3.1 The 2×4 Cross-linked DNA Grid Model

A hierarchical approach was used to develop the 2×4 cross-linked DNA grid model. A new spring type, the "X" spring, was added in the locations where the cross-links were identified as shown earlier in Figure 42. The details of the spring are escribed in Appendix B.12. In addition to the twelve cross-linking sites in the tile motif model, there are also two sites on the sticky ends that are not included in the independent tile motif because the cross-linking process requires a dsDNA strand. The input file generator was modified to include these changes assuming a 100% activity, i.e., every site is cross-linked, during the cross-linking procedure. A screen capture of the cross-

linked tile model using the graphical output program described in Appendix B.9 is presented in Figure 48.

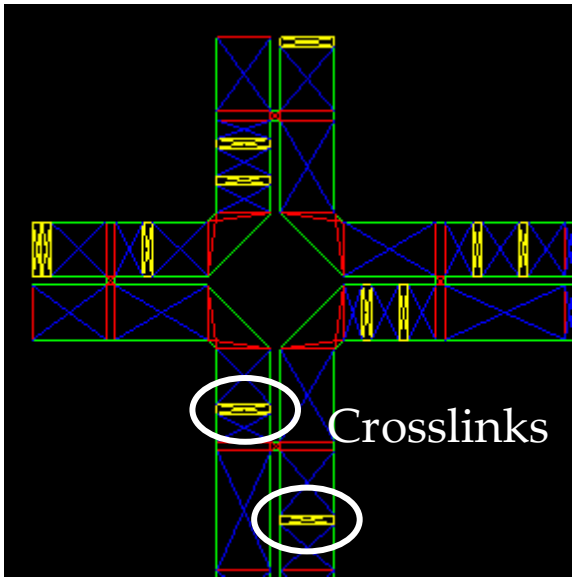


Figure 48: A screen capture of the tile motif with twelve cross-links.

The cross-linked tile motif is used to construct the 2 × 4 cross-linked DNA grid nanostructure model for simulation using the DNA-STRAIN simulator. Eight cross-linked tile motifs were connected using sticky end models. Eight cross-linking sites were identified in the oligonucleotide sequences for the sticky ends, and included in the modified DNA grid model. A screen capture of the 2 x 4 cross-linked DNA grid nanostructure model is presented in Figure 49.

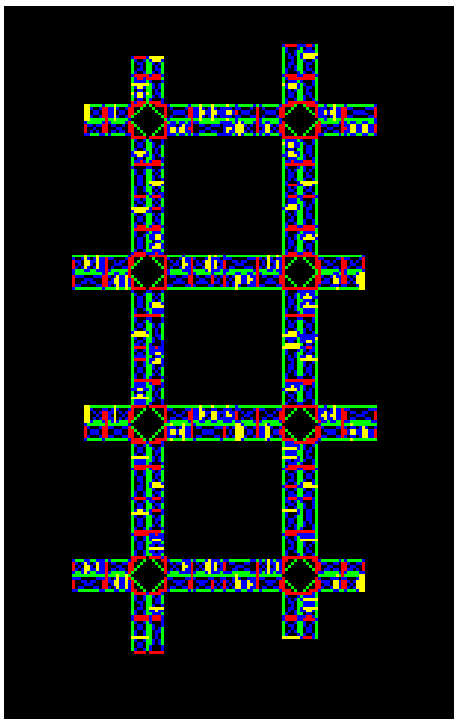


Figure 49: A screen capture of the 2 × 4 cross-linked DNA grid nanostructure.

All melting temperatures assigned to the thermodynamic regions for the model are the same as those for the uncross-linked 2 × 4 grid model. The presence of the cross-links in the nanostructure should strain the structure during the simulation to affect the thermodynamic analysis. The simulation and analysis of the modified DNA grid nanostructure is presented in the next section.

5.3.2 Simulation of the Cross-linked Tile Model

The simulations were run 100 times over a 100 °C range from -20°C to 80°C at 0.1°C increments with 5 ns per temperature step. The data was then processed by filtering the data through a third order polynomial Savitsky-Golay filter over a 101 point window. The processed data was then analyzed to determine thermodynamic parameters. Experimental data was obtained by cross-linking another sample of 2 × 4

grid nanostructures, and melting them using the same procedures as previously described in Section 5.2.1. The results were analyzed using the same procedure as the simulation data. Figure 50 presents the van't Hoff sum plot, the simulation results, and the experimental results for the uncross-linked and cross-linked samples, followed by Table 14 comparing the thermodynamic parameters for the two plots.

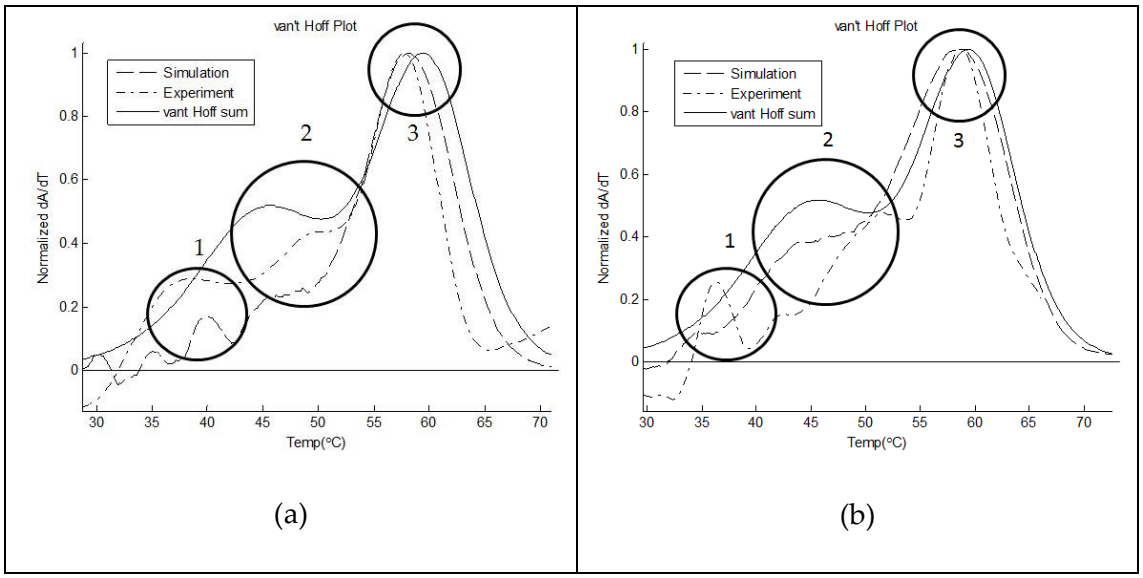


Figure 50: Data plots for (a) the uncross-linked sample and (b) the cross-linked sample of 2 × 4 DNA grid nanostructures.

Table 14: Thermodynamic parameters of the 2 × 4 cross-linked DNA grid.

Transition 1	Extracted	Experiment	% Error
Tm (°C)	34.57	36.42	5.08%
ΔH (kcal/mol)	250.47	209.48	-19.57%
ΔG (25 °C kcal/mol)	-8.0436	-8.0274	-0.20%
ΔS (cal/degree mol)	840.5	702.94	-19.57%

Transition 2	Extracted	Experiment	% Error
Tm (°C)	47.595	51.907	8.31%
ΔH (kcal/mol)	45.959	60.715	24.30%
ΔG (25 °C kcal/mol)	-3.4848	-5.482	36.43%
ΔS (cal/degree mol)	154.23	203.74	24.30%

Transition 3	Extracted	Experiment	% Error
Tm (°C)	58.081	58.672	1.01%
ΔH (kcal/mol)	68.378	105.16	34.98%
ΔG (25 °C kcal/mol)	-7.5906	-11.882	36.12%
ΔS (cal/degree mol)	229.46	352.87	34.97%

The plots indicate that the simulation parameters are overall within 37% of the experimental results for all three transitions. While the transition temperatures are within 10% of the experimental transition temperatures, the remaining thermodynamic parameters indicate that the transitions of the simulation are broader than the experimental data. This result is due to the presence of the cross-links straining the overall structure of the model. However, these models also have very stiff springs, minimizing the change in length of the overall structure, causing the absorbance to stabilize during the denaturation process. This results in a broadening of the transition in the simulation data, especially in Transition 2. The sticky end transition temperatures, based on the arm-arm theory presented in Section 3.3.3.6, fall mostly within Transition 2, with 40% of them possessing cross-link sites. As a result, this transition is larger in the than in smaller nanostructure models, again suggesting that the sticky ends additively contribute to the absorbance as the size of the structure increases. The presence of the cross-links, in both the experimental data and the simulation data, improves the resilience of the nanostructures as indicated by the broadening of the transitions quantitatively shown in the thermodynamic analysis.

Experiments that repeatedly subjected a sample to a constant temperature ramp process indicated that the cross-links gradually decreased the transition temperatures by as much as 10%, as shown in Section 5.2.3. In this experiment, three samples were subjected to only one temperature ramp for the analysis. Table 15 shows the comparison of the melting temperatures between the uncross-linked samples and the cross-linked samples for both the simulated and experimental data.

Table 15: Comparison of the simulation and experimental transition temperatures for both the uncross-linked and cross-linked 2 × 4 DNA grid nanostructures.

Uncross-linked Sample		
Transition	**Simulation Tm (°C)**	**Experimental Tm (°C)**
1	40.112	38.912
2	47.471	50.883
3	57.893	57.689

Cross-linked Sample		
Transition	**Simulation Tm (°C)**	**Experimental Tm (°C)**
1	34.57	36.42
2	47.595	51.907
3	58.081	58.672

The data in the table indicates that the single temperature ramp experiment gives a different result from the repeated temperature ramp experiment. While the first transition decreases by as much as 8% when comparing the uncross-linked to the cross-linked sample, the second and third transitions increase by up to 2%. This trend is present in the experimental data and captured by the DNA-STRAIN simulator. The shift in the transitions is possibly due to the change in the configuration of the grid

nanostructure as it denatures into the "rat's nest" cross-linked structure. Comparing the results from this experiment to the repeated ramp experiment suggests that the repeated denaturation and reannealing process gradually destabilizes the uncross-linked grid nanostructure further, eventually causing a shift in all three transitions.

In summary, the DBS and the QSA metric spectrally and quantitatively verified the successful cross-linking of the 8-tile DNA grid nanostructure using procedures adapted from previous work. A temperature test was conducted to observe the effect of cross-linking on the nanostructure in terms of its optical absorbance properties by repeatedly denaturing and reannealing the samples. Analysis of the absorbance data and the van't Hoff data from the DNA-STRAIN simulator post-processing program indicated that cross-linking the nanostructure made the nanostructures resistant to the denaturation process, preventing complete dissociation of the structure as indicated by the reduced range of the change in absorbance. However, cross-linking also destabilized the internal configuration of the nanostructure due to the introduction of strain to the tile motif during the cross-linking process, causing minor destabilization to its structural integrity in repeated temperature ramps. These conclusions are drawn from 1) the QSA analysis, indicating a reduction of the metric by 92% for the control and 37% for the cross-linked sample; 2) the thermodynamic analysis, enthalpy and melting temperature values determined from the post-processing program, indicating up to a 10% decrease in melting temperatures as well as increases in enthalpy values by as much as a factor of three; and 3) the difference in the absorbance throughout the temperature ramp, with a

30% difference in absorbance for the control as opposed to a 12% difference in absorbance for the cross-linked sample.

A second temperature ramp test with multiple samples for a single heating process was performed to compare the uncross-linked and cross-linked samples using the DNA-STRAIN simulator. A single denaturation study was performed in triplicate to obtain experimental data, and a model of the cross-linked DNA nanostructure was created and simulated. The results indicated that the cross-linking process improved the stability of the nanostructure through van't Hoff data analysis. Generally, the transition temperatures increased by as much as 2%, and broadened due to the increasing presence of sticky ends, as indicated by decreased transition enthalpy values. The DNA-STRAIN simulation of the cross-linked model captured the behavior of the experimental data in the shift of the melting transitions as well as the broadening of the transitions due to the presence of the cross-links, with the worst parameter being within 37% of the expected data.

In this chapter, the DBS, QSA metric, and the DNA-STRAIN simulator were used to study the therm-mechanical behavior of a cross-linked DNA grid nanostructure. These characterization tools can potentially aid in the future design these nanostructures by predicting their physical response in different environments. Optimization of the base sequences for the nanostructures is also possible (i.e., the number and placement of cross-link sites for optimal structural stability). These possibilities lead to the study of defects in the DNA nanostructure and their effect on future DNA-based logic circuits as presented in Chapter 6.

6. Impact of Self-Assembly Defects on Logic Circuits

DNA self-assembled nanostructures are ideal for scalable fabrication for computational applications. However, defects that occur during self-assembly can cause a device to behave incorrectly, or fail before its intended operational lifetime has expired. Identifying the source of the failure can potentially be tedious and time-consuming. This chapter describes the impact of defects in proposed DNA-based logic circuits. Defects are categorized in terms of the known defects of DNA nanostructures as well as anticipated defects from possible functionalized components. Fault models of basic logic circuits are presented to describe possible circuit-level configurations that result from these defects.

6.1 Self-Assembly Defect Categories

Figure 51 categorizes the known defects; defects imaged using the AFM for current DNA nanostructures, and anticipated defects; defects that can occur in future nanoelectronic fabrication processes on the DNA scaffold:

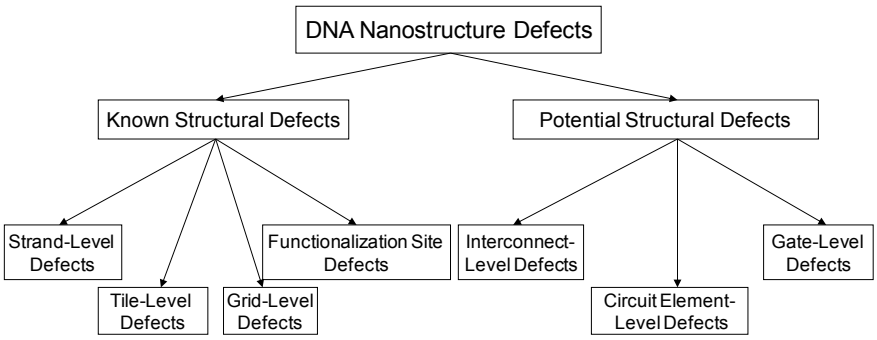


Figure 51: Categorization of physical defects.

From the diagram, strand-level defects, tile-level defects, grid-level defects, and functionalization-site defects further divide the known defect category into four levels of

increasing self-assembly complexity. Environment-related defects and functionalization defect divide the anticipated defect category into two areas. The following sections describe each defect.

6.1.1 Known Defects

Strand-level defects can occur during synthesis of the oligonucleotides used in the nanostructure. These defects can propagate to higher levels of complexity but can be eliminated by biochemical purification [112]. There are two categories of defects at this level: 1) primary-structure defects, which affect the oligonucleotide sequence, and 2) secondary-structure defects, which cause the strand to form more complex structures due to interactions in solution. Sequence design is critical for the thermodynamic stability of the nanostructure and to avoid unintentional secondary structures [69]. Defects such as deletion of a base, insertion of an extra base, or a defective base fall under the primary-structure defect class. The formation of hairpin loops by partial self-complementarity or complementarity among multiple strands fall under the secondary-structure defect class. These defects cannot be easily observed in a nanostructure but they will propagate to higher levels. Figure 52 shows a classification of various strand-level defects.

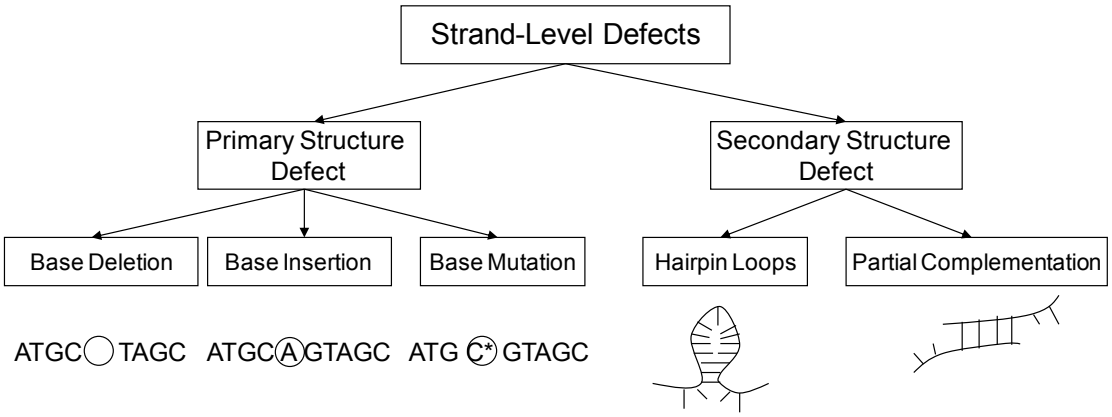


Figure 52: Strand-level defects and their categorization. X is a random base (A, T, G or C) inserted in the sequence. X* is a deteriorated version of the intended base in the sequence.

Tile-level defects occur due to propagation of defects from the strand level. The tile motif is composed of nine DNA oligonucleotide sequences that are classified into three different types based on their role in the tile: the core, the shell, and the arm, as described in [69]. Defects in the intra-tile category include partial formation of the tile due to missing DNA strands and partial complementarity of foreign strands. Inter-tile defects describe defects that involve aggregation or distortion at the tile level. The classification for defect categories at this level and corresponding illustration are presented in Figure 53.

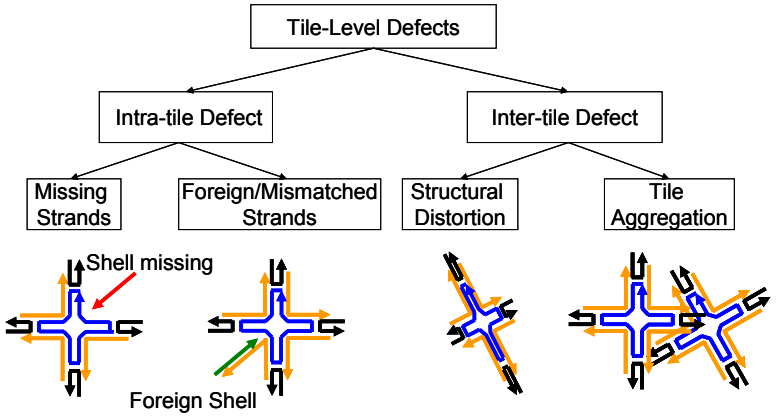


Figure 53: Classification of tile-level defects.

In the intra-tile defect category, two types of defects can occur. Missing strands on the tile due to random interactions depend on the relative concentration of the appropriate strands. These defects cause the partial or complete destabilization of a tile. The extent of the destabilization depends on whether the missing strand is a core, a shell, or an arm and defects in this class cannot be observed. Base-pair mismatches between a core and shell, or a shell and an arm, can result in a tile defect. Foreign strands are free strands that can bind to the tile structures at their sticky ends and prevent tiles from interacting correctly. Defects in this class are not easily detected but their propagation to the grid level is readily observable. For example, foreign/mismatched defects appear in an AFM image as a gap in the nanostructure from either folded or broken arms. In most cases, it is difficult to distinguish among foreign, mismatched, or aggregated tiles.

The inter-tile defect category describes possible defects that can cause tiles to interact with each other before the grid-level synthesis takes place. This defect can be attributed to strand-level defects that propagate to the tile level. Hairpin loops and partial complementarity from the strand level can further interact with partially-formed tile structures, creating large aggregates. The structure of the tile can be distorted due to missing strands or strained deposition on the imaging surface. These defects lead to a reduced number of correct tiles available for grid-level synthesis.

Periodicity defects refer to defects that disrupt the regularity of the grid structure. Missing tiles result in the partial formation of grid structures and broken tiles refer to tiles that do not completely bind to adjacent tiles. The surface defect class refers

to interactions between grids due either to defects that propagate from the tile- or strand-level interactions with the imaging substrate. Structural distortion of the grid can be caused by missing strands or tiles and can cause the grid to fold or twist. Grid aggregation can occur between different sections of the grid but can most easily occur between edges due to base stacking between free sticky ends. Figure 54 illustrates the classification of grid-level defects.

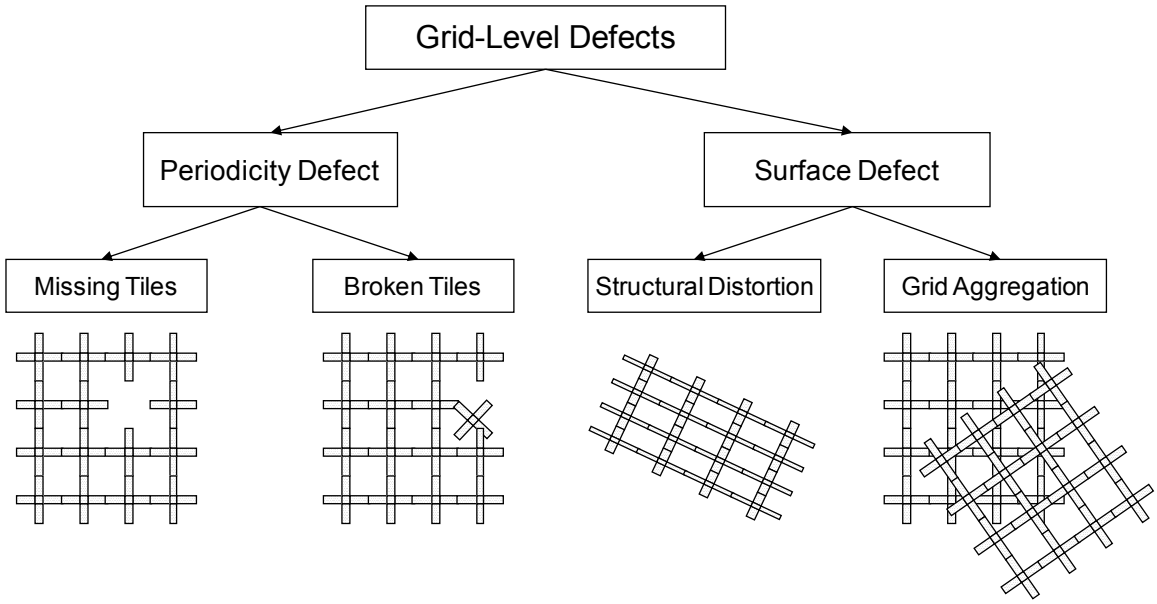


Figure 54: Categorization of grid-level defects.

Functionalization-site defects refer to defects during the preparation of the nanostructure for functionalization with a component. A proof-of-concept demonstration is presented in [64] using biotin-streptavidin conjugates functionalized to core strands in the tile. These conjugates can be laid out in predetermined configurations as templates for active components as demonstrated by prior work [113-114]. Figure 55 presents the proposed categories for classifying these defects.

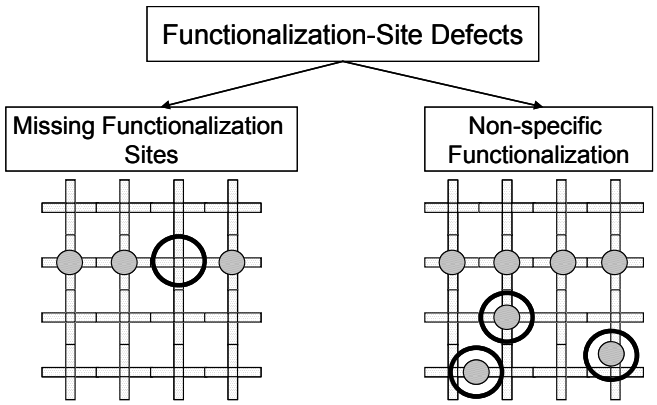


Figure 55: Functionalization-site defects in biotinylated grids. The fault free structure is a wire-like configuration with four streptavidin proteins attached across the second row of the array.

Defects at this level originate from chemical modification and environmental factors required for preparing the DNA nanostructure for functionalization. Missing functionalization sites are defects in which the streptavidin does not attach to the intended site. Non-specific functionalization defects occur when particles attach to incorrect sites.

6.1.2 Anticipated Defects

There are two types of defects in the anticipated defects category: environment-related and functionalization-related. Environment-related defects at this level arise from chemical manipulation or random interactions. Defects that occur in components can be classified as functionalization defects in interconnect, within gate components, or simply as operational defects. A brief description of each type of defect that can occur during fabrication is described in the following paragraphs.

Chemical manipulation of otherwise defect-free tiles can cause the tiles to become inactive at the device level. During the functionalization process, defects can be

introduced to the grid structure due to imbalanced pH (highly acidic or basic environments), high levels of UV irradiation (200-300 nm), or exposure to organic solvents (e.g., ethanol and methanol) which all degrade DNA. A difference in pH can make it very difficult for a grid structure to form due to structural distortion or aggregation. Irradiation by UV light can cause dimerization that destabilizes the grid structure and potentially disrupt the functionalization process. Organic solvents can cause the aggregation of grids or the dissociation of tile structures. If a functional element is loosely attached to a grid, it may dissociate during chemical manipulation. Also, non-specific functionalization can result in overcrowding of unnecessary circuit elements or distortion of the grid structure, introducing strain on the grid. The occurrence of both defects depends on the length of chemical exposure and/or the pH of the environment.

A functionalization defect is one that occurs during the attachment of active components such as metal nanowires, carbon nanotubes, or other nanoparticles. Figure 56 illustrates this category.

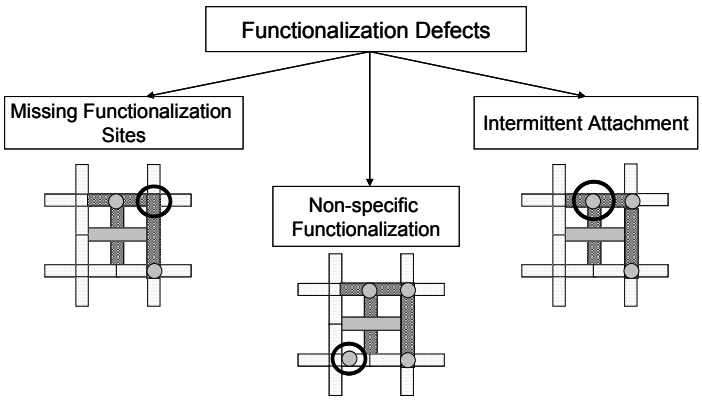


Figure 56: Functionalization defects as examples of anticipated structural defects.

The defects in this category are more complex than previous categories because they potentially depend on multiple synthesis procedures. For example, missing functionalization sites result in missing functional elements that are required for proper operation of the circuit. Non-specific functionalization of a component can cause unexpected shorts. Intermittent attachments of components cause inconsistent contacts between conducting elements.

6.1.3 Possible Fault Models

With a categorization of defects, fault models can be made using a combination of anticipated nanoelectronic layouts with the known defects that are observable in AFM images of currently synthesizable DNA nanostructures. Physical defects from the previous section can be mapped to fault models that best characterize the resulting circuit behavior. An example of a gate layout using carbon nanotube FETs (CNFETs), CNT interconnect, and metal nanoparticles is presented in Figure 57. It is assumed that the CNFET transistor can be doped either as an NFET or a PFET [115]. This functionalized DNA nanostructure was presented in [116] for a proposed DNA-based nanoarchitecture. The DNA nanostructures consist of carbon nanotube-based transistors with metal nanoparticles to serve as the contact points for two layers of interconnects. Two layers of insulation material separate the grid from possible conductive metallic planes, which allow for selective contact of V_{dd} and ground connections. Carbon nanotubes attached (by functionalization) to different parts of the DNA scaffold serve as interconnect between gates. Fault models corresponding to these defects can be derived

for wires, resistors, capacitors, and logic gate elements. Some of these possible fault models for basic circuit elements, mapped from physical defects, are illustrated in Figure 58. Some possible gate-level fault models for a NOR gate are illustrated in Figure 59.

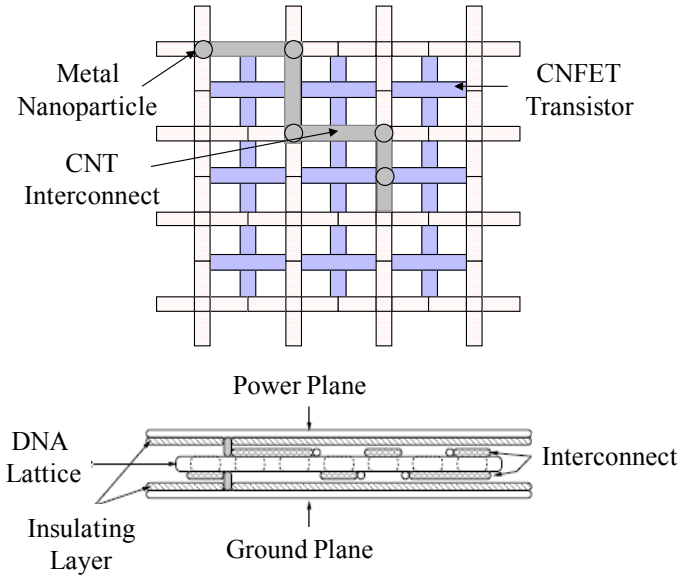
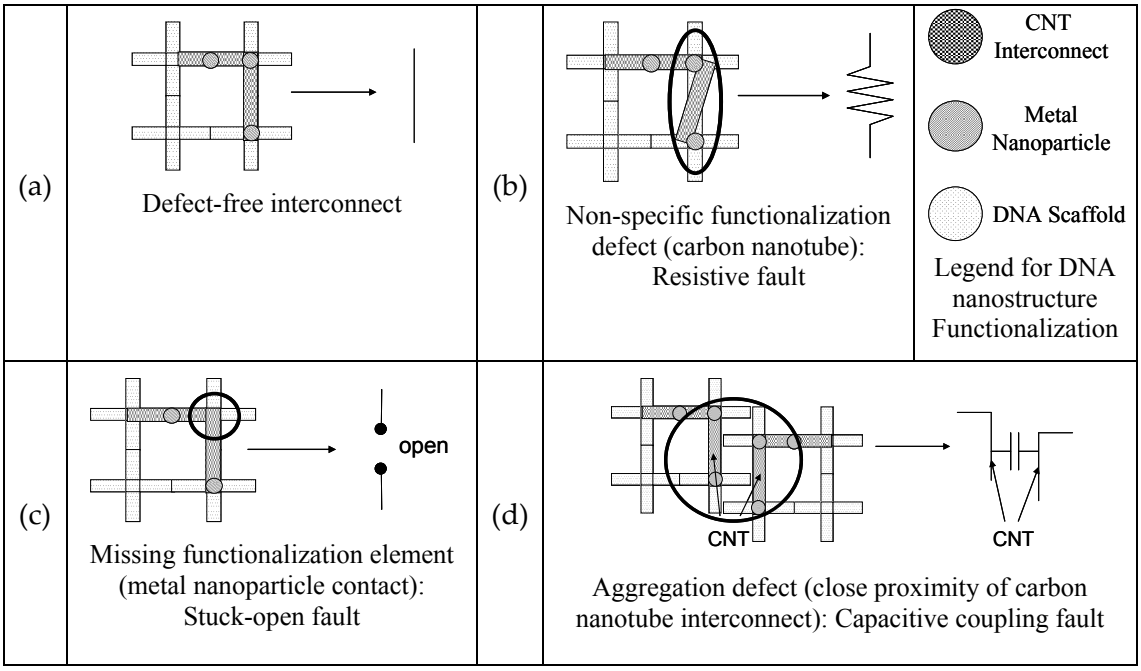


Figure 57: An example functionalized DNA self-assembled nanodevice. The cross section of the grid illustrates the proposed method for supplying power [117] .



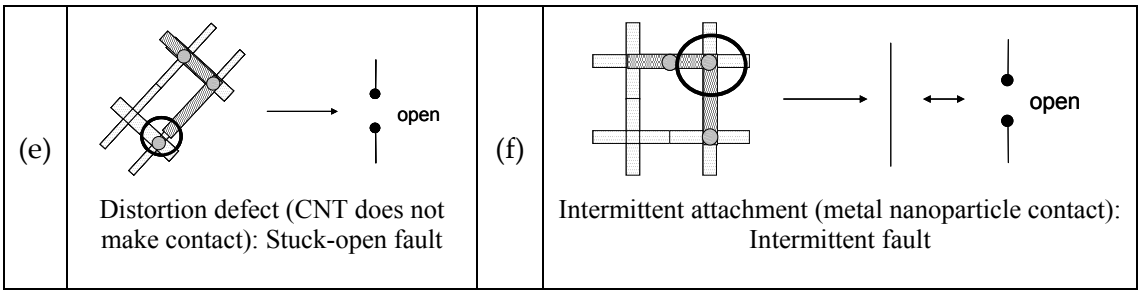


Figure 58: Physical defects mapped to possible fault models for a basic interconnect element. (a) Fault-free case, (b) a resistive fault, (c, e) stuck-open faults, (d) a capacitive coupling fault, and (f) an intermittent fault.

Figure 58(a) illustrates a defect-free interconnect made of carbon nanotubes and metal nanoparticle contacts. Figure 58(b) presents a resistive fault caused by partial contact due to non-specific functionalization of the carbon nanotube. A missing contact causes a stuck-open fault, as presented in Figure 58(c). Aggregation or close proximity between different grid structures causes capacitive coupling between different interconnects, as in Figure 58(d). Distortion of the DNA scaffold prior to functionalization causes a stuck-open fault at the carbon nanotube interconnect, as shown in Figure 58(e).

Defects in logic gates, such as the NOR gate layout presented in Figure 59(a), can give rise to other types of faults. Missing metal contacts cause stuck-open faults, e.g., on the primary output in Figure 59(b). Missing functionalization elements, such as the missing P2 transistor in Figure 59(c), can cause an output stuck-at-0 fault since the output cannot be pulled up to V_{dd} due to the defect. This is an example of a defect that is rare in CMOS technology. A missing tile on the DNA scaffold results in an unavailable functionalization site for active circuit components, presented in Figure 59(d). The resulting fault is an inactive P2 transistor for this case. The non-specific

functionalization of a carbon nanotube interconnect causes an elevated source voltage for N1. This yields a reduced gate voltage swing that can lead to deterioration in on-state drive current. Depending on the location of the defect, a delay fault can result; see Figure 59(e). As a result, N1 takes longer to switch on, adversely affecting circuit speed.

If a grid acts as a scaffold upon which conducting interconnect elements will be functionalized, the method by which these elements are fabricated or attached will determine the types of defects that can appear. If the element is composed of a series of nanoparticles, a missing tile can result in an open or degraded signal. If the element to form a wire requires a metallization growth process directly on the scaffold, it is likely that the defect will not appear since the gap can be closed by the growing metal contact. However, if the gap is large, a partial interconnect may form which yields a large resistance.

Shorts and opens can be caused by improper connections. Capacitive coupling can be caused by interference of interconnect or other circuit elements between different nanostructures due to close proximity. Inorganic functionalized materials can be improperly bound to the scaffold causing stuck-at-faults or bridging faults. The extent of the incorrect binding determines whether the defect is permanent or intermittent. An intermittent defect implies that a wire element may make contact properly at certain periods but that it is likely to lose its connection later due to an external disturbance or change in the environment.

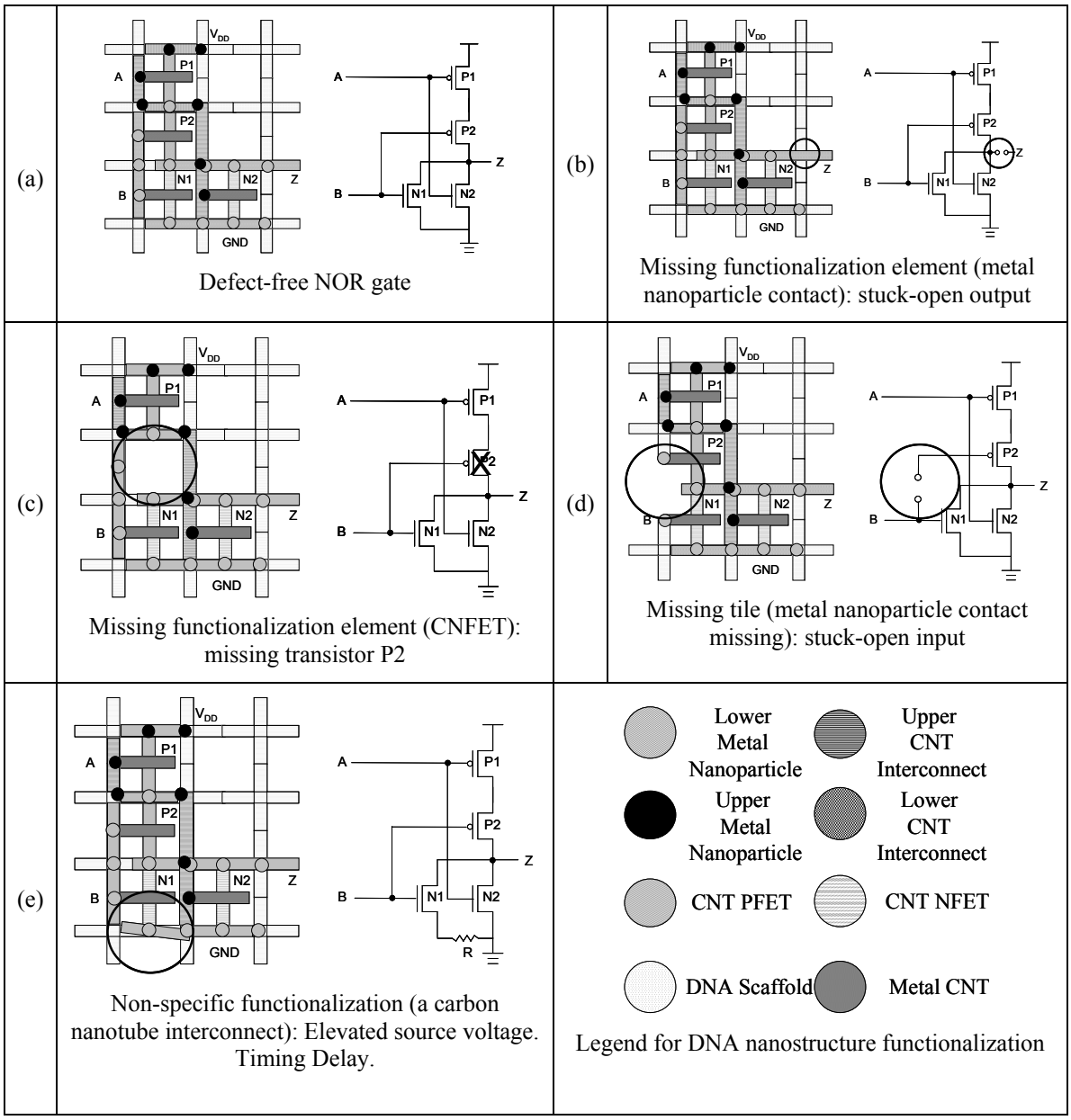


Figure 59: Physical defects mapped to possible fault models for a NOR gate. (a) Fault-free case, (b) stuck-open output, (c) inactive transistor fault, (d) stuck-open input, and (e) a resistive fault.

In conclusion, defects were identified from AFM images of DNA grid nanostructures, both unfunctionalized and functionalized, and categorized into known and anticipated defects. The impact of these defects was described with respect to logic circuits using fault models.

Appendix

A. QSA Metric

1. Calculation of the Correction Factor

To derive the calculation of the AFM tip convolution correction factor, ten randomly selected complete DNA nanostructures in an AFM image are selected for width measurements. The images are converted to a binary image after applying a threshold to remove the background before measuring the widths. For each DNA nanostructure, twenty-four pixel values are counted from the exterior of the nanostructure to the closest cavity as well as between cavities to determine the width variation in the DNA nanostructure. Figure 60 shows the histogram of the results.

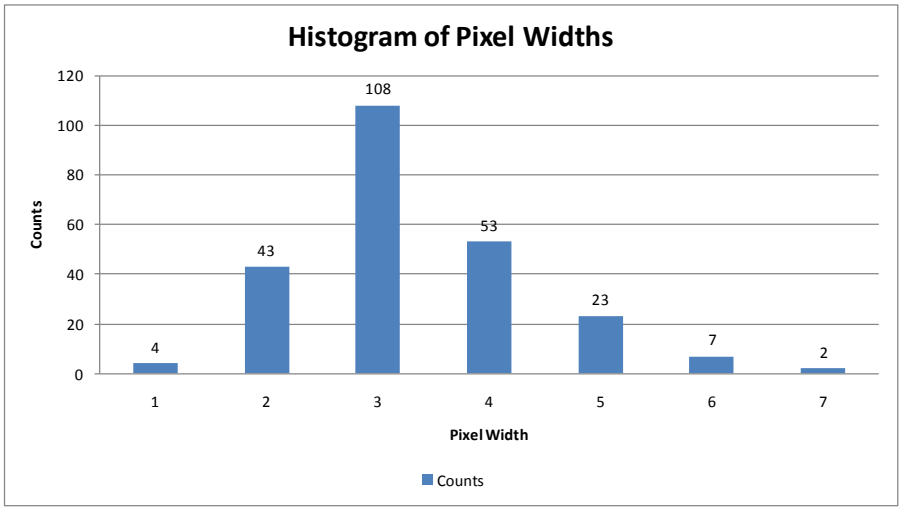


Figure 60: Histogram of 240 widths measured for ten DNA nanostructures.

The measurements are also statistically analyzed assuming a Gaussian distribution. An average width of 12.99 nm with a standard deviation of 4.19 nm is used

to determine the approximate range of the cavity area. A correction factor of 3.25 is calculated as the ratio of the average width to the accepted width of the DNA (i.e., ~2 nm). The average DNA area of the ten selected DNA nanostructures is 4730.225 nm^2, giving a correction factor of 2.05 when compared to the ideal area. The discrepancy between the correction factor determined from the apparent DNA widths and that from the area could be explained by the fact that the widths were measured along the axis-aligned horizontal distance rather than arbitrary vectors in the image. The greatest distortion in the width measurement occurs with grids that are deposited at a rotated angle.

To calculate the full range of the possible values of the cavity area, the average cavity area was determined to be approximately three standard deviations from the standard cavity area of 256 nm^2 given at rotating the grid structure in 15° increments. Let $\boldsymbol{C_{3\sigma}}$ be the cavity area determined from three standard deviations of the width. The resulting value is calculated to be 111.94 nm^2 using Equation 38.

$$C_{3\sigma} = 20 - (C_A + 3_\sigma)$$

Equation 38

2. User Manual

To use the QSA metric, the macros must be checked out from SVN to the desktop. The first macro, "DNAMacro_1.0.txt", determines the calibration value for the

image based on the type of grid nanostructure being analyzed. After increasing the resolution of the image using the "Scale" function in the "Image" menu, this macro is applied to individual nanostructures in the image, not the entire image. The user must select and "Duplicate", in the "Image" menu, the particular nanostructure and apply the macro by selecting the "Run" command in the "Plugins->Macros" menu. It is recommended that at least five to ten nanostructures be analyzed. This set of values should be recorded and averaged to get the final calibration value. A simple example is shown in Figure 61.

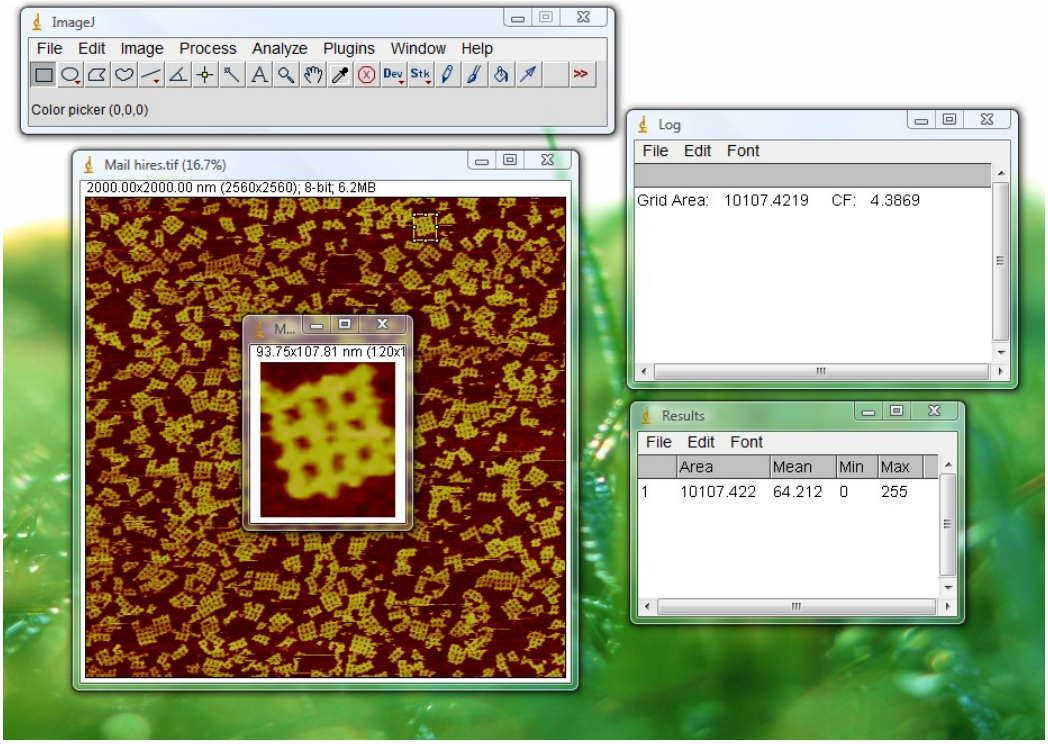


Figure 61: Running the calibration macro.

In Figure 61, the image has been scaled by a factor of five. One of the DNA nanostructures has been selected and duplicated. The "Results" window outputs the total area in square nanometers of the DNA nanostructure. The "Log" window outputs the DNA surface area as well as the calibration factor "CF". In this case, the factor is 4.3869. The calibration value is applied to the second macro.

Before applying the second macro, the threshold level of the image must be checked, as these values vary between image sessions. The "Threshold" command is selected from the "Image->Adjust" menu. Once this option is selected, a window will appear in which the lower threshold and upper threshold can be selected. These values must be modified in the macro code itself just as the calibration factor must be adjusted.

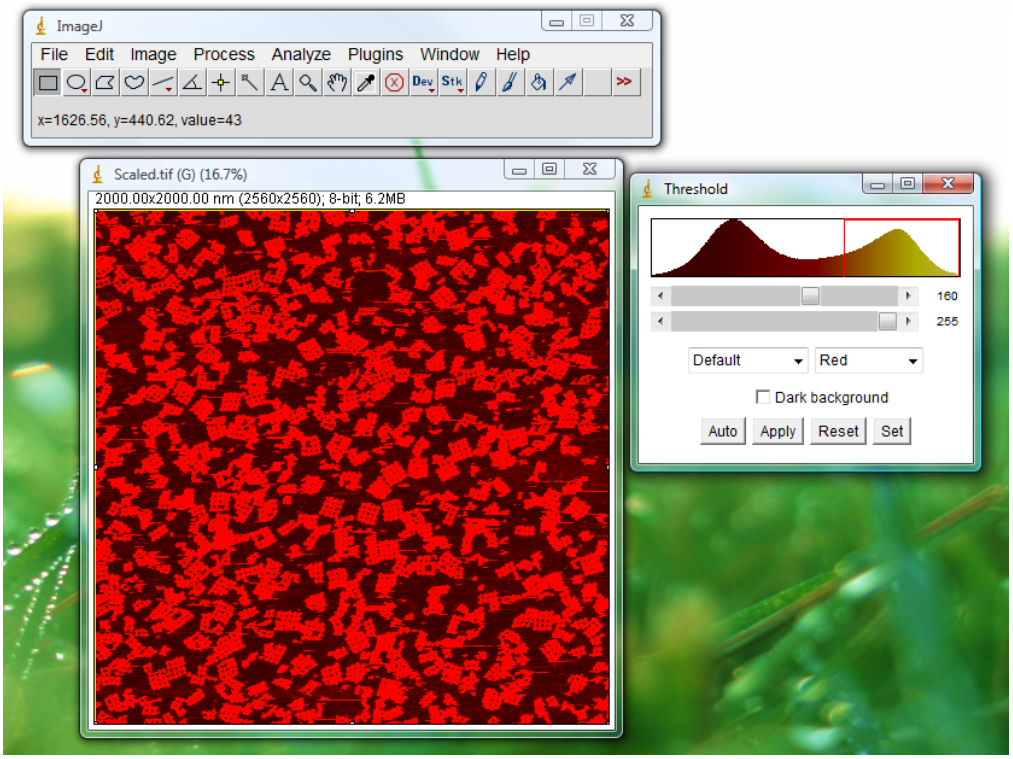


Figure 62: Identifying the threshold settings for the image.

In Figure 62, the threshold settings have been identified as being 160 and 255 for the low and high thresholds, respectively. These numbers are based on the 256 color scale. For a high quality image, the appropriate values are easy to identify as the right range results in the highlighting of nearly all the DNA in the image. These numbers, along with the calibration factor, are adjusted in the second macro before execution.

In the second macro, "MetricMacro_1.1.txt", the process is automated to generate the QSA metric once the value is applied. A screen capture of the resulting analysis for the example image is shown in Figure 63.

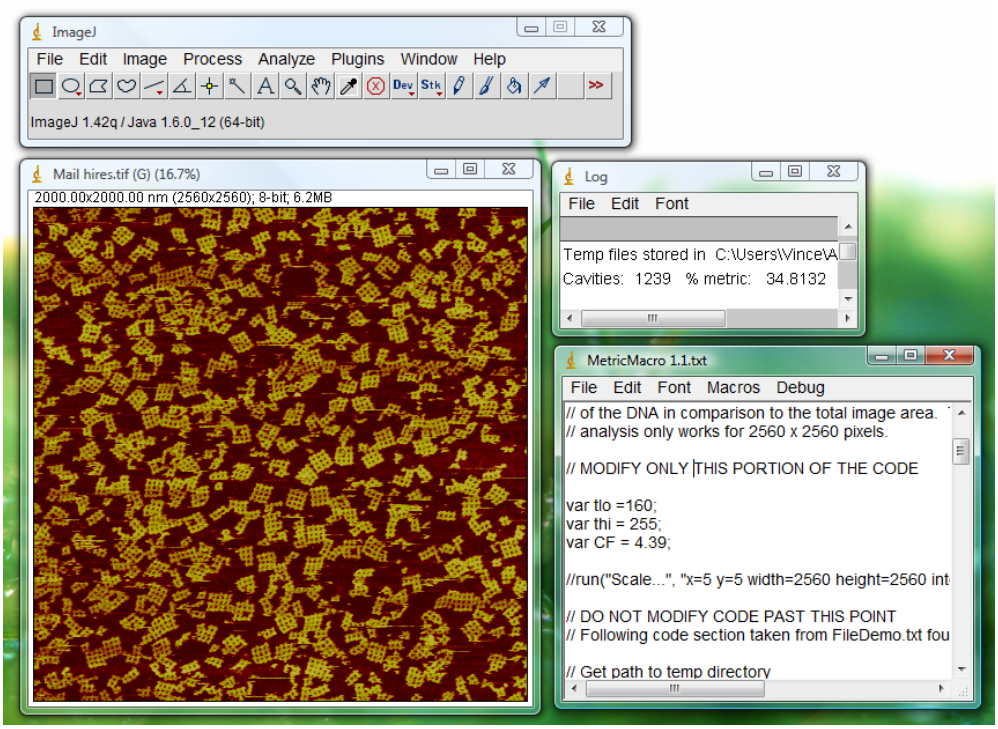


Figure 63: The processed image with the QSA result.

In the above figure, the threshold values and the calibration factor have been modified in the macro as determined earlier. The number of cavities found in the example image, 1239, is presented along with the QSA metric of 34 in the "Log" window. The result indicates that approximately 34% of the nanostructures in the image are well-formed.

3. ImageJ Macro Code

```
// DNA Area calculation macro
// Written by Vincent Mao
// Last modified: August 21, 2007

// This macro is written for AFM images of DNA lattice nanostructures.
// The macro calculates the DNA area in an image (presumably
// a complete grid is selected) and calculates the ratio between
```

```
// the calculated value and the ideal area for a 4 x 4 grid.

// MODIFY ONLY THIS PORTION OF THE CODE

var tlo =110;
var thi = 255;

run("Set Scale...", "distance=512 known=2000 pixel=1 unit=nm global");

// DO NOT MODIFY CODE PAST THIS POINT

                    run("Duplicate...", "title=[Duplicate.bmp]");
                    run("8-bit");
                    setAutoThreshold();
                    //run("Threshold...");
                    setThreshold(tlo, thi);
                    run("Convert to Mask");
                    rename("DNA.bmp");
                    run("Measure");
                    area = getResult("Area");
                    CF = area/2304;

                    print("Grid Area:  ", area, "    CF:  ", CF);
                    close();
```

```
// Image analysis macro
// Written by Vincent Mao
// Last modified: August 21, 2007

// This macro is written for AFM images of DNA lattice nanostructures.
// The macro analyzes the image by determining the number of
// cavities present in the image as well as the DNA area and
// calculates the normalized metric as well as the coverage area
// of the DNA in comparison to the total image area.  The image
// analysis only works for 2560 x 2560 pixels.

// MODIFY ONLY THIS PORTION OF THE CODE

var tlo =110;
var thi = 255;
var CF = 2.36;

run("Scale...", "x=5 y=5 width=2560 height=2560 interpolate create title=[Scaled.bmp]");

// DO NOT MODIFY CODE PAST THIS POINT

// Following code section taken from FileDemo.txt found at http://rsb.info.nih.gov/ij/macros/

// Get path to temp directory
 tmp = getDirectory("temp");
 if (tmp=="")
    exit("No temp directory available");

 // Create a directory in temp
 myDir = tmp+"Temp"+File.separator;
 File.makeDirectory(myDir);
 if (!File.exists(myDir))
    exit("Unable to create directory");
 print("");
 print("Temp files stored in ", myDir);

saveAs("Tiff", myDir+"Scaled");

// End code section taken.

var sect = 1;
var c1 = 0;
var c2 = 0;
var c3 = getWidth()/sect;
var c4= getWidth()/sect;

var j = 0;

for (i = 0; i <sect; i+=1)
{
            c1 = i* getWidth()/sect;
            for(j = 0; j<sect; j+=1)
            {
```

```
		c2 = j* getWidth()/sect;
		makeRectangle(c1, c2, c3, c4);
		open(myDir+"Scaled.tif");
		run("8-bit");
		setAutoThreshold();
		run("Set Scale...", "distance=2560 known=2000 pixel=1 unit=nm global");
		//run("Threshold...");
		setThreshold(0,tlo);
		run("Convert to Mask");
		rename("Cavities.bmp");
		run("Analyze Particles...", "size=30-300 circularity=0.40-1.00 show=Masks clear");
		cav = nResults();
		grids = cav / 9;

		open(myDir+"Scaled.tif");
		run("8-bit");
		setAutoThreshold();
		//run("Threshold...");
		setThreshold(tlo, thi);
		run("Convert to Mask");
		rename("DNA.bmp");
		run("Measure");
		area = getResult("Area");
		metric = (cav*CF)/area * 1000000;
		cover = area / (4000000) * 100;

		iarea = grids * 2304;
		pcover = area / (iarea * CF);

		pmetric = metric/3906 * 100 ;
		imetric = pmetric * pcover ;
		if(cav > 0 && isNaN(imetric) == 0)
		{
			print("Cavities: ", cav , "  % metric:  ", pmetric,"  % DNA Coverage:     ", cover);
				close();
			close();
			close();
		}
		else
		{
			print("No data for this section");
			close();
			close();
		}
}

// Clean up code taken from FileDemo.txt at http://rsb.info.nih.gov/ij/macros/

  // Display info about the files
  list = getFileList(myDir);
  print("");
  for (i=0; i<list.length; i++)
     print(list[i]+": "+File.length(myDir+list[i])+"  "+File. dateLastModified(myDir+list[i]));
```

```
// Delete the files and the directory
for (i=0; i<list.length; i++)
   File.delete(myDir+list[i]);
File.delete(myDir);
if (File.exists(myDir))
   exit("Unable to delete directory");
else
   print("Directory and files successfully deleted");

// End code section taken.
```

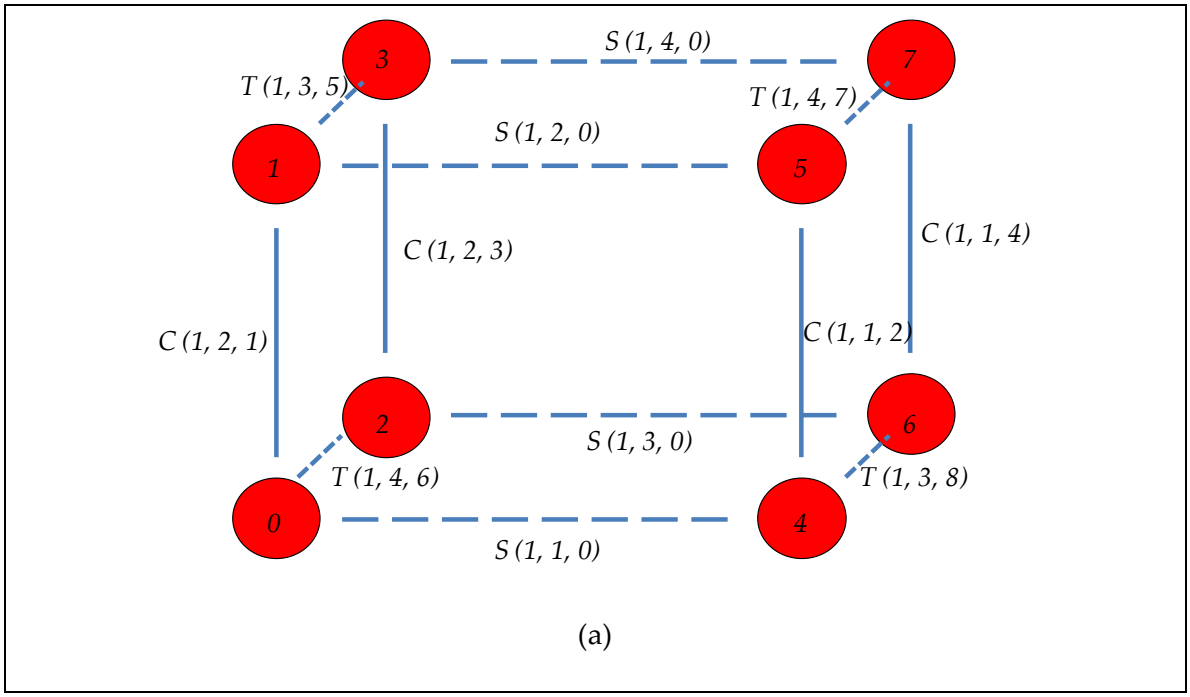
B. The DNA-STRAIN Simulator

1. 3D Spring Network Identification System

There are four spring types that are used to identify the twenty-four springs in the 3D spring model. "S" denotes the four springs that have the spring constant value of the DNA represented by the model, and are illustrated using the dashed lines running along the x-axis in Figure 64a. "T" denotes the four support springs that are stiffer than the rest of the springs in the model. These springs have a spring constant that is eight times larger than the "S" springs, and do not degrade with increasing temperature. Eight "B" springs act as braces that cross the front, back, lower and upper faces along the x-axis of the model. These springs have dynamically calculated spring constants dependent on the angle, Θ, formed between the spring and its "S" spring as shown in Figure 64b. There are eight "C" springs that have the same stiffness as the "T" springs, but degrade with temperature with the "S" springs. This degradation represents the transition from the double-stranded to the single-stranded DNA structure. A three-number identifier system is implemented in order for the support springs to follow the

DNA springs in the degradation process. Four identifiers define each spring in terms of either representing the DNA molecule itself or as a support spring.

The 3D spring model will behave the same as the simple spring model using the system described above. However, the model can potentially bend and twist in unexpected configurations that are not observed for DNA at this length range. To prevent these configurations from occurring, a numbering system assigns three of the DNA springs in the model to the fourth spring. This system also assigns the support springs to their respective DNA springs. The following describes the numbering system. The numbering system for one rectangular segment is presented in the following figure:



(a)

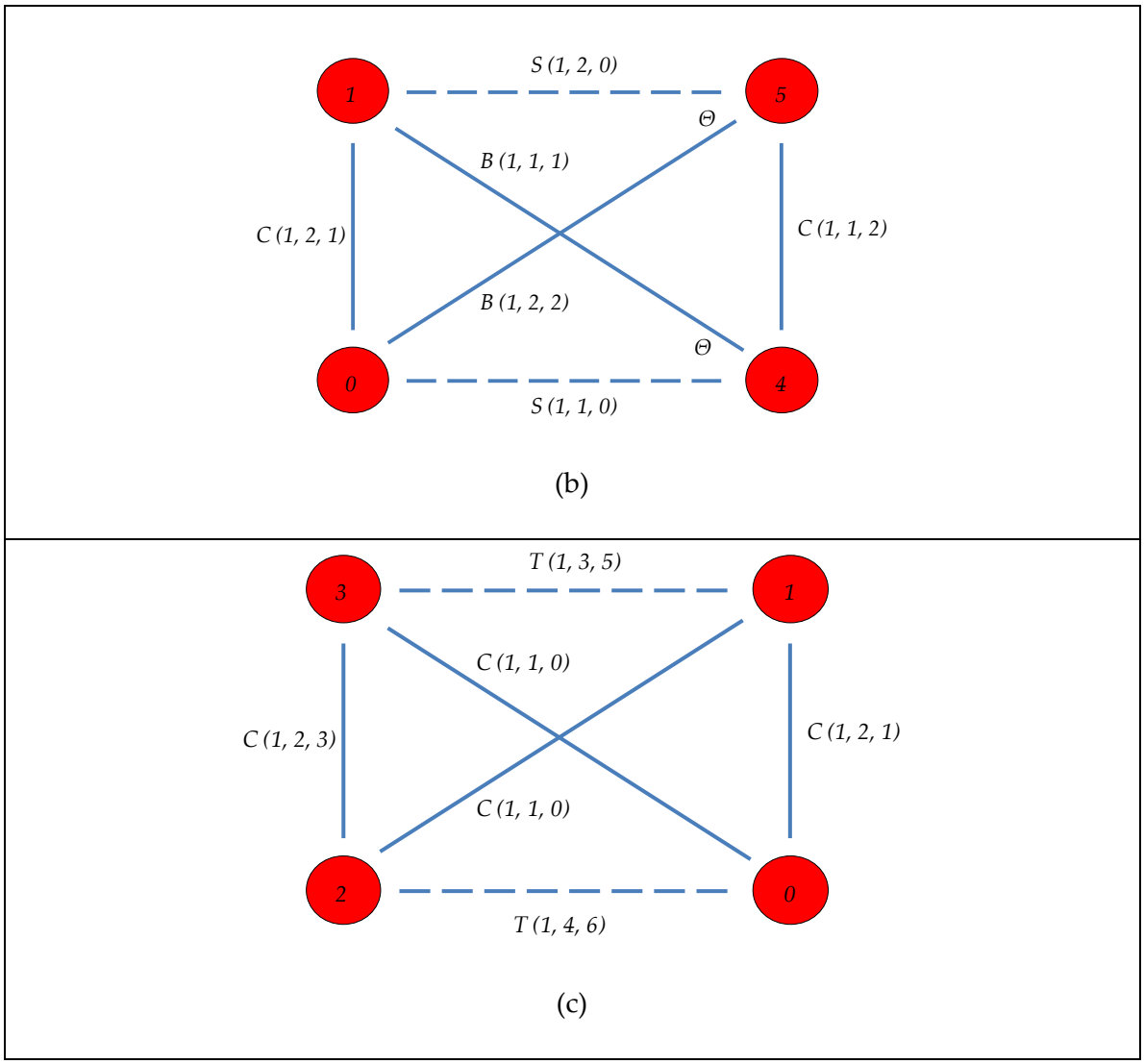


Figure 64: Identification system for the (a) rectangular frame, (b) support braces crossing the front, back, upper, and lower faces and (c) left and right faces of the 3D DNA model

The support springs ("B", "C", and "T") are assigned for the hypotenuse calculations based on a three-number identifier scheme. The first number identifies the 3D model segment to which the spring belongs to in the model if it is part of a complex nanostructure model such as a freely jointed chain. The range of this number goes from

1 to the number of 3D model segments used to compose the complex nanostructure. The second number identifies which of the four "S" springs that the support spring refers to when determining the spring constant. The angle formed between the "B" spring and its assigned "S" spring for dynamic spring constant calculations required to correctly model the physical spring response uses the first two numbers in the system. The third number in the system is used to identify which of the eight "T" springs that the "B" spring uses for the hypotenuse and angle calculations. When the model is simulated above the melting temperature, randomly generated force vectors are applied to each pair of mass nodes per face as shown in Figure 65.

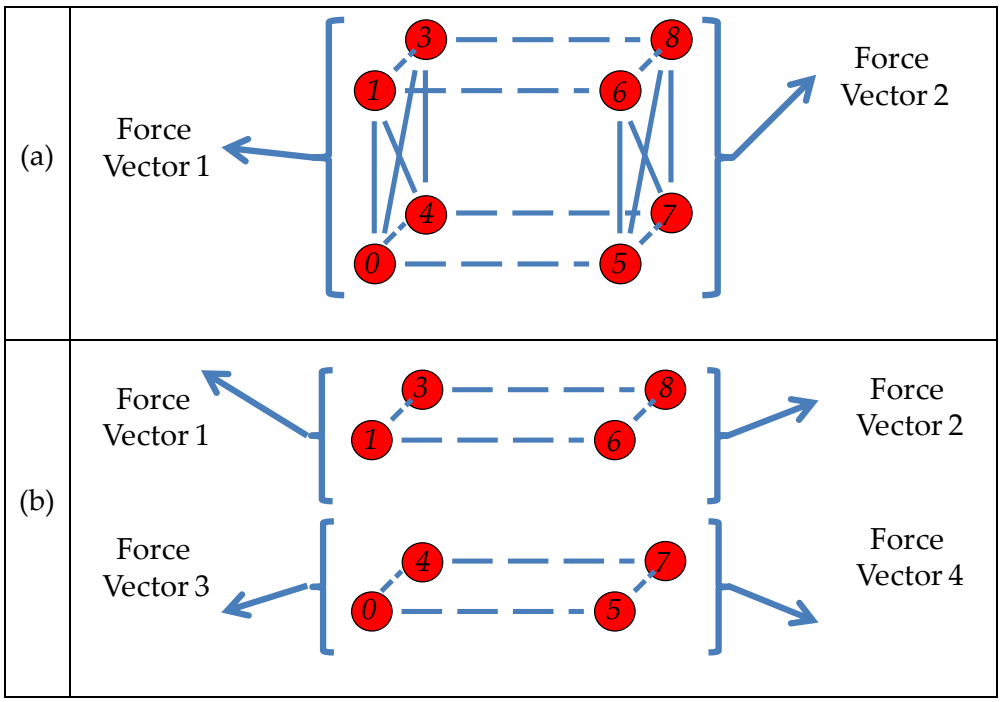


Figure 65: Brownian force vectors applied to the 3D model for temperatures (a) less than the melting temperature, and (b) greater than the melting temperature.

The above figure illustrates the extent to which the current model can represent the transition from dsDNA to ssDNA in the simulation. While the model represents a unimolecular denaturation process, the support springs that maintain the 3D structure no longer exert any restoring force. The only support springs that still have significant spring constants are the "T" springs after denaturation.

2. The Command Flags for the Simulator

The command flags for the simulator specify the settings not defined in the input file. In Table 16, the first four flags (h, t, m, o) are for diagnostic purposes, the next three flags (c, r, b) determine simulation settings and the last five flags (a, v, n, f, x) determine data collection and data output. The following paragraphs describe each flag.

Table 16: Command flags in the simulator.

Flag	Description
h	Show all possible command flags
t	List all active command flags
m	Output all mass information
o	Run an oscillation test
c	Specify the constant temperature at which to run the simulation
r	Specify the number of simulations runs
b	Apply Brownian Force
a	Sum the data (absorbance, length)
v	Export mass coordinates for constant temperature simulations
n	Export mass coordinates for denaturation simulations
f	Specify input and output files
x	Store all output data in binary files

For the diagnostic flags, the "-h" command flag displays all the command flags available for the user to select as well as any additional inputs (i.e., numbers or file names) needed. The simulator terminates if this command is present. The "-t" flag outputs all information about the active command flags detected in the command prompt at the beginning of the simulation. If the "-m" flag is present in the prompt, all information about the input file concerning the mass nodes and the springs are displayed on the screen prior to running the simulation. The "-o" flag simulates undamped oscillation behavior for the validation of the physical response of a spring model, allowing the spring to oscillate indefinitely. The damping coefficient for all DNA models is assumed to be critically damped otherwise.

For the simulation setting command flags, the "-c" flag simulates the model at a constant temperature. The "-r" flag is followed by an integer that sets the number of times to run the simulation before writing the final data to the output files. If the number is greater than one, each data point is averaged over the time step or the temperature step depending on the presence of the constant temperature flag. The "-b" flag is used to apply Brownian force to the model during simulation.

In the data collection/data output flags, the "-a" flag is needed to sum the appropriate data (e.g., absorbance, length) to treat the model as a single unit when there are multiple segments in the model, such as for the case of DNA nanostructures.

Without this flag present for models containing multiple segments, the simulator records data separately for every segment in the model to the output. The data in this case is useful for individually studying the behavior of specific segments in the model. The "-v" and "-n" command flags generate a separate output file for recording mass coordinate data for a constant temperature mode or melting mode, respectively. The mass coordinate data file is required for replaying the simulation. The "-f" flag is used to specify the source file and five (data output file, binary file/text file for constant temperature simulation, binary file/test file for melting simulation) output files. The "-x" flag indicates the simulator to generate binary files for the mass coordinate files. This option is selected for storage economy because binary data consumes less memory.

To run the oscillation test, the oscillation, constant temperature and number-of-run command flags are entered on the command prompt. The user may also define the initial simulation settings in the "U" identifier in the input file. To run the diffusivity test, the Brownian force application, constant temperature and number-of-run command flags are entered on the command prompt are used for this test. To run a melting simulation, the range is defined in the input file with the line beginning with the "U" identifier. The user omits the command flag for running the simulation at constant temperature. The default setting uses the temperatures defined in the input file when the constant temperature flag is not set on the command prompt.

3. Source Code for the DNA-STRAIN Simulator

Main source code for the simulator.

```
//--------------------------------------//
//      DNA-STRAIN Simulator                        //
//      author: Vincent Mao                                 //
//      written: 9.11.09                                    //
//  last edited: 1.25.10                            //
//--------------------------------------//

/*--------------------------------------------------------------------
        This program reads a meshfile that defines springs, masses,
        and their properties and increases temperature over a defined
        range.  An absorbance value is calculated based on the change in
        length of the spring.  All data is recorded in an array.
*/

#define _CRT_SECURE_NO_WARNINGS

// PPD's
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "Init_Structs.h"
#include "Globals.h"

int loadMeshFile(void);
int initSprings(void);
int runSim(void);
int collectData(SPRING_node *sh, MASS_node *mh);
int writeStructData(SPRING_node *sh, MASS_node *mh);
int resetSim(void);
int processData(void);
int avgData(void);
int writeData(void);
int exportData(void);
int readBinData(void);
int releaseMem(void);

int main(int argc, char *argv[])
{
        int i = 0;
        int argPos = 0;
        int comPos1 = 0;
        int comPos2 = 0;
        int comPos3 = 0;

        char *msg = NULL;
        char fchr;

        // Check for command line parameters
        if(argc == 1){
                printf("ERROR: No command parameters found.  Type \"-h\" for a list of
commands...\n");
                return(-1);
        }
        else{
                // Identify command line parameters
```

```
		for(i; i < argc; i++)
		{
			if(argv[i][0] != '-'){
				continue;
			}

			fchr = argv[i][1];

			switch(fchr)
			{
				case 't':
					printf("Testing mode...outputting flag
detectors\n");
					testMode = 1;
					break;

				case 'h':
					printf("List of possible command flags: \n \
-t: List all active flags and generate outputs in a text file\n \
-m: Output mesh file data\n \
-b: Brownian force applied to simulation\n \
-s: Simple mass/spring model\n \
-l: WLC model\n \
-a: Calculate properties for entire model\n \
-v: Export snapshot mass vectors to a file for constant temperature\n \
-n: Export snapshot mass vectors to a file for melting simulation\n \
-o: Basic spring behavior test\n \
-p: Apply a constant force laterally to the model\n \
-f: Specify temp and output file names <src> <out> <bin> <bin> <out> <struct>\n \
-r: Specify number of simulation runs <runs>\n \
-c: Specify constant temperature simulation <temperature>\n \
-x: Toggle writing binary files as well as text files\n");
					return(-1);
					break;

				case 'm':
					msg = "Displaying mesh file data...\n";
					meshOutput = 1;
					break;

				case 'b':
					msg = "Brownian force enabled...\n";
					brownianOn = 1;
					break;

				case 's':
					msg = "Simple segment model...\n";
					simpleFlag = 1;
					break;

				case 'l':
					msg = "Worm-like chain model...\n";
					wlcModel = 1;
					break;

				case 'a':
					msg = "Total model calculations...\n";
					totalModel = 1;
					break;

				case 'v':
					msg = "Recording snapshot information for mass and
springs for constant temperature...\n";
```

```
                                        vecExport = 1;
                                        break;

                                case 'n':
                                        msg = "Recording snapshot information for mass and
springs for melting simulation...\n";
                                        snapShot = 1;
                                        break;

                                case 'o':
                                        msg = "Oscillation test on the spring...\n";
                                        oscTest = 1;
                                        dataRes = 1;
                                        break;

                                case 'p':
                                        msg = "Constant force applied laterally...\n";
                                        constForce = 1;
                                        break;

                                case 'f':
                                        msg = "Searching for file names...\n";
                                        nameFiles = 1;
                                        comPos1 = i;
                                        break;

                                case 'r':
                                        numRuns = 1;
                                        comPos2 = i;
                                        break;

                                case 'c':
                                        constTemp = 1;
                                        comPos3 = i;
                                        break;

                                case 'x':
                                        binary = 1;
                                        break;
                        }
                }
                if(testMode){
                        if(msg){
                                printf(msg);
                        }
                }
        }

        // Check for errors (does not detect incorrect input types)
        if(nameFiles){
                if(binary){
                        if( (argc - (comPos1 + 1)) < 6){
                                printf("ERROR: Missing file input parameters...\n");
                                return(-1);
                        }
                        else{
                                strcpy(srcFile, argv[comPos1 + 1]);
                                strcpy(outputFile, argv[comPos1 + 2]);
                                strcpy(rawFile, argv[comPos1 + 3]);
                                strcpy(rawMassFile, argv[comPos1 + 4]);
                                strcpy(massFile, argv[comPos1 + 5]);
                                strcpy(snapshotFile, argv[comPos1 + 6]);
```

```
                                // The output file is incremented in the Python script on
the platypus server
                                while(argPos != strlen(argv[comPos1 + 2])){
                                        factor += (int)(argv[comPos1 + 2][argPos]);
                                        argPos++;
                                }

                                if(testMode){
                                        printf("Source data file is \"%s\".\n", srcFile);
                                        printf("Output data file is \"%s\".\n", outputFile);
                                        printf("Binary data file is \"%s\".\n", rawFile);
                                        printf("Binary mass vector file is \"%s\".\n",
rawMassFile);
                                        printf("Output mass vector file is \"%s\".\n",
massFile);
                                        printf("Melt mass vector file is \"%s\".\n",
snapshotFile);
                                }
                        }
                }
                else{
                        if( (argc - (comPos1 + 1)) < 4){
                                printf("ERROR: Missing file input parameters...\n");
                                return(-1);
                        }
                        else{
                                strcpy(srcFile, argv[comPos1 + 1]);
                                strcpy(outputFile, argv[comPos1 + 2]);
                                strcpy(massFile, argv[comPos1 + 3]);
                                strcpy(snapshotFile, argv[comPos1 + 4]);

                                // The output file is incremented in the Python script on
the platypus server
                                while(argPos != strlen(argv[comPos1 + 2])){
                                        factor += (int)(argv[comPos1 + 2][argPos]);
                                        argPos++;
                                }

                                if(testMode){
                                        printf("Source data file is \"%s\".\n", srcFile);
                                        printf("Output data file is \"%s\".\n", outputFile);
                                        printf("Output mass vector file is \"%s\".\n",
massFile);
                                        printf("Melt mass vector file is \"%s\".\n",
snapshotFile);
                                }
                        }
                }
        }
        else{
                strcpy(srcFile, "mesh.txt");
                strcpy(outputFile, "rundata.txt");
                strcpy(rawFile, "rundata.bin");
                strcpy(rawMassFile, "massdata.bin");
                strcpy(massFile, "massdata.txt");
                strcpy(snapshotFile, "structdata.txt");

                if(testMode){
                        printf("Using default filenames:\n");
                        printf("\tSource file is \"%s\".\n", srcFile);
                        printf("\tOutput data file is \"%s\".\n", outputFile);
                        printf("\tMelt mass vector file is \"%s\".\n", snapshotFile);
                }
```

```
	}

	if(numRuns){
		if( (argc - comPos2) < 1){
			printf("ERROR: Missing number of runs...\n");
			return(-1);
		}
		else{
			simRuns = atoi(argv[comPos2 + 1]);
			if(testMode){
				printf("Running simulation %.d times...\n", simRuns);
			}
		}
	}
	else{
		if(testMode){
			printf("Running simulation %.d times (DEFAULT SETTING)...\n",
simRuns);
		}
	}
	if(constTemp){
		if( (argc - comPos3) <= 1){
			printf("ERROR: Missing temperature parameter...\n");
			return(-1);
		}
		else{
			tempStart = atof(argv[comPos3 + 1]);
			tempStop = tempStart;
			if(tempStart < 253 || tempStart > 370){
				printf("ERROR: Unreasonable temperature...exiting.\n");
				return(-1);
			}
			if(testMode){
				printf("Running simulation at %.2f K...\n", tempStart);
				printf("NOTE: Defining the temperature in the mesh file
overwrites inputs here!!\n");
			}
		}
	}
	else{
		if(testMode){
			printf("Running simulation starting at %.2f K (DEFAULT
SETTING)...\n", tempStart);
		}
	}

	// Load the meshfile
	loadMeshFile();

	// Initialize simulation run parameters
	currentTemp = tempStart;
	if(constTemp){
		tempStop = tempStart;
	}
	tempRange = tempStop - tempStart;

	if(!constTemp){
		if(tempRange < 0){
			printf("ERROR: The temperature ramp cannot be
negative...exiting.\n");
			return(-1);
		}
		if(dataRes == 0 || targetRes == 0){
```

```
                        printf("ERROR: Bad data recording parameters in user command
input...exiting.\n");
                        return(-1);
                }
                tempInc = tempRange / (rampRate * tempRange);
                targetTemp1 = tempStart;
                targetTemp2 = targetTemp1 + tempInc;
                targetInc = tempRange / (targetRes * tempRange);

                if(testMode){
                        printf("Temperature incrementing in steps of %.2f K...\n",
tempInc);
                }
        }

        initSprings();

        // Run the simulation simRuns times
        while(currentSimNum < simRuns)
        {
                resetSim();

                if(!constTemp){
                        currentTemp = tempStart;
                        stepsPerMelt = 0;
                }

                runSim();

                if(constTemp){
                        dataPts = arrayPos;
                }
                else{
                        dataPts = stepsPerMelt;
                }

                processData();

                currentSimNum++;
                printf("Sim is %.2f%% complete...\n", (float)currentSimNum /
(float)simRuns * 100);
        }
        avgData();

        if(!binary){
                writeData();
        }
        else{
                exportData();
                readBinData();
        }
        releaseMem();

        return(0);
}

int loadMeshFile()
{
        meshFile = fopen(srcFile, "rt");

        if(meshFile != NULL)
        {
                while(!feof(meshFile))
```

```
		{
			// Node identification variables
			char type;
			int nodeDef = -1;

			// Temperature variables and concentration values
			double userTempStart, userTempStop, userConc, userSimTime;
			int userDataRes, userSnapRes;

			// Face deflection calculation variables
			int segRef1, faceRef1, segRef2, faceRef2;

			// Mass node variables
			int brownianFlag;
			double mass, x, y, z;

			// Spring node variables
			int nM1, nM2, nS1, nS2, nS3;
			double restLength, meltTemp, basePairs, extCoeff, extCoeff2,
scaleFactor;

			// Scan the mesh file and assign values to structs
			fscanf(meshFile, "%c ", &type);
			switch(type)
			{
				case 'D':
				case 'd':
					nodeDef = 0;
					fscanf(meshFile, "%d %d %d %d", &segRef1, &faceRef1,
&segRef2, &faceRef2);
					break;

				case 'M':
				case 'm':
					nodeDef = 1;
					fscanf(meshFile, "%lf %lf %lf %lf %d", &mass, &x,
&y, &z, &brownianFlag);
					break;

				case 'S':
				case 's':
					nodeDef = 2;
					fscanf(meshFile, "%d %d %d %d %d %lf %lf %lf %lf %lf
%lf", \
						&nS1, &nS2, &nS3, &nM1, &nM2, &restLength,
&meltTemp, &basePairs, &extCoeff, &extCoeff2, &scaleFactor);
					break;

				case 'B':
				case 'b':
					nodeDef = 3;
					fscanf(meshFile, "%d %d %d %d %d %lf %lf", \
						&nS1, &nS2, &nS3, &nM1, &nM2, &restLength,
&scaleFactor);
					break;

				case 'T':
				case 't':
					nodeDef = 4;
					fscanf(meshFile, "%d %d %d %d %d %lf %lf", \
						&nS1, &nS2, &nS3, &nM1, &nM2, &restLength,
&scaleFactor);
					break;
```

```
                                case 'C':
                                case 'c':
                                        nodeDef = 5;
                                        fscanf(meshFile, "%d %d %d %d %d %lf %lf", \
                                                &nS1, &nS2, &nS3, &nM1, &nM2, &restLength,
&scaleFactor);
                                        break;

                                case 'U':
                                case 'u':
                                        nodeDef = 6;
                                        fscanf(meshFile, "%lf %lf %lf %lf %d %d", \
                                                &userTempStart, &userTempStop, &userConc,
&userSimTime, &userDataRes, &userSnapRes);
                                        break;

                                case 'R':
                                case 'r':
                                        nodeDef = 7;
                                        fscanf(meshFile, "%d %d %d %d %d %lf %lf", \
                                                &nS1, &nS2, &nS3, &nM1, &nM2, &restLength,
&scaleFactor);
                                        break;

                                case 'X':
                                case 'x':
                                        nodeDef = 8;
                                        fscanf(meshFile, "%d %d %d %d %d %lf %lf %lf %lf %lf
%lf", \
                                                &nS1, &nS2, &nS3, &nM1, &nM2, &restLength,
&meltTemp, &basePairs, &extCoeff, &extCoeff2, &scaleFactor);
                                        break;

        //                      // Ignore new line
                                case 10:
                                        break;

        //                      // Ignore space at end of line
                                case 32:
                                        break;

                                default:
                                        printf("Unrecognized line starting with '%c'\n",
type);
                                        break;
                        }

                        if(nodeDef == 0){
                                if(meshOutput == 1){
                                        printf("\nDeflection check: (Segment, Face 1 = L, 2
= R) \n\tFace 1: (%d, %d)\n\tFace 2: (%d, %d)\n\n", \
                                                segRef1, faceRef1, segRef2, faceRef2);
                                }
                        }
                        if(nodeDef == 1){
                                // Mass definition...
                                addMass(&mh, mass, x, y, z, brownianFlag);
                                if(meshOutput){
                                        printf("MASS NUM: %d\t MASS: %.2e kg\t MASS COORDS:
(%.1lf nm, %.1lf nm, %.1lf nm)\n", \
                                                 mh->massNum, mh->mass, mh->x * 1e9, mh->y *
1e9, mh->z * 1e9);
```

```
                        }
                }
                if(nodeDef == 6){
                        tempStart               = userTempStart;
                        tempStop                = userTempStop;
                        concentration   = userConc;
                        simTime                 = userSimTime;
                        dataRes                 = userDataRes;
                        targetRes               = userSnapRes;
                        if(meshOutput){
                                printf("\nTEMP RANGE: %.2fK - %.2fK every %es for
%eM sample\n", tempStart, tempStop, simTime, concentration);
                        }
                }

                if(nodeDef > 1 && nodeDef < 6 || nodeDef >= 7) {
                        // Spring definition...
                        MASS_node* pM1 = NULL, *pM2 = NULL;

                        if(nM1 < massCount){
                                pM1 = findmass(&mh, nM1);
                        }
                        else {
                                printf("Mass %d beyond limit.\n", nM1);
                        }
                        if(nM2 < massCount){
                                pM2 = findmass(&mh, nM2);
                        }
                        else{
                                printf("Mass %d beyond limit.\n", nM2);
                        }

                        // Add springs
                        if(nodeDef == 2 || nodeDef == 8){
                                addSpring(&sh, nS1, nS2, 0, \
                                        restLength, meltTemp, basePairs, extCoeff,
extCoeff2, \
                                        pM1, pM2, \
                                        segRef1, faceRef1, segRef2, faceRef2,
scaleFactor, nodeDef);
                        }
                        if(nodeDef == 3){
                                addSpring(&sh, nS1, nS2, nS3,
                                        restLength, 0, -1, 0, 0, \
                                        pM1, pM2, \
                                        segRef1, faceRef1, segRef2, faceRef2,
scaleFactor, nodeDef);
                        }
                        if(nodeDef == 4){
                                addSpring(&sh, nS1, nS2, nS3,
                                        restLength, 0, -2, 0, 0, \
                                        pM1, pM2, \
                                        segRef1, faceRef1, segRef2, faceRef2,
scaleFactor, nodeDef);
                        }
                        if(nodeDef == 5 || nodeDef == 7){
                                addSpring(&sh, nS1, nS2, nS3,
                                        restLength, 0, -3, 0, 0, \
                                        pM1, pM2, \
                                        segRef1, faceRef1, segRef2, faceRef2,
scaleFactor, nodeDef);
                        }
                        if(meshOutput){
```

```
                                        printf("\nSPRING DEF %d: Segment/Spring/Section:
(%d, %d, %d)\n", nodeDef, nS1, nS2, nS3);
                                        printf("\tMass %d -> Mass %d\t", nM1, nM2);
                                        printf("Rest Length : %.1lf nm\t Tm : %.1lf K\t
\n\tBase Pairs : %.0lf\t Extinction Coefficientx : %.2lf\t %.2f\n", \
                                                sh->restLength * 1e9, sh->meltTemp, sh-
>basePairs, sh->extCoeff, sh->extCoeff2);
                                        printf("\tFace: %d\t Markers: %d %d \t Spring
constant: %e N/m\n", \
                                                sh->face, sh->marker1, sh->marker2, sh->k);
                                }
                        }
                }
                fclose(meshFile);
        }
        else {
                perror("Source file not found in the current directory...exiting\n");
                return(-1);
        }

        return(0);
}

int initSprings()
{
        // Structural springs assigned to DNA springs
        refSpringSearch(sh);
        // Initialize spring lengths, angles, spring constants
        initSpringLength(sh);
        initBraceAngles(sh);
        initSprConsts(sh);
        if(!simpleFlag){
                // Initialize deflection angles of the structure faces
                initDefAngles(sh);
        }
        // Initialize position of mass for diffusion recording and time step
        initCenterPos(mh);
        detTimestep(sh);
        // Seed random number generator
        srand(time(NULL)+factor);
        return(0);
}

int runSim()
{
        while(currentTemp <= tempStop){
                // For melting mode, reset the time and counter, write snapshot data if
it's the last temperature and update spring costants
                if(!constTemp){
                        currentTime = 0;
                        stepsPerRun = 0;
                        resetClengthRMS(sh);
                        if(snapShot && currentSimNum == (simRuns - 1) ){
                                writeStructData(sh, mh);
                        }

                        updateDNASprConsts(sh);
                        if(!simpleFlag){
                                updateBraceSprConsts(sh);
                        }
                }
                // The following loop runs over the time period defined
                while(currentTime < simTime)
```

```
		{
			if(brownianOn){
				if(!constForce){
					applyBrownian(sh);
				}
				else{
					applyConstForce(sh);
				}
			}

			if(!simpleFlag){
				updateDefAngles(sh);
			}
			// Mark the mass positions, determine all forces on masses and move
them
			updateDiffusionPos(mh);
			processSprings(sh);
			moveMasses(mh);

			// Write data to arrays, resolution depends on sim mode (const or
melt)
			if(constTemp){
				if(stepsPerRun % dataRes == 0){ // write every dataRes
steps (1 is every step)
					collectData(sh, mh);
				}
			}
			else{
				collectData(sh, mh);
			}

			currentTime += timeStep;
			stepsPerRun++;
		}

		if(!constTemp){
			dataArray[arrayPos][0] = currentTemp;
			dataArray[arrayPos][1] /= stepsPerRun;
			dataArray[arrayPos][2] /= stepsPerRun;
			dataArray[arrayPos][3] /= stepsPerRun;
			dataArray[arrayPos][4] /= stepsPerRun;
			dataArray[arrayPos][5] /= stepsPerRun;
			dataArray[arrayPos][6] /= stepsPerRun;
			arrayPos++;

			if(vecExport){
				massPosArray[massArrayPos][0] = currentTemp;
				massPosArray[massArrayPos][1] = currentTime;
				massPosArray[massArrayPos][2] = mh->x;
				massPosArray[massArrayPos][3] = mh->y;
				massPosArray[massArrayPos][4] = mh->z;
				massArrayPos++;
			}
			currentTemp += tempInc;
			printf("Currently at %.2f K\r", currentTemp);
			stepsPerMelt++;
		}
		else{
			break;
		}
	}
	return(0);
}
```

```
int collectData(SPRING_node *sh, MASS_node *mh)
{
	// Collect spring data
	while(sh != NULL){
		if(sh->nodeDef == 2 && sh->springRef == 1){
			if(constTemp){
				dataArray[arrayPos][0] = currentTime;
				dataArray[arrayPos][1] = sh->absChange;
				dataArray[arrayPos][2] = sqrt(sh->lCsqr/(stepsPerRun + 1));
				dataArray[arrayPos][3] = sh->strandAbs;
				dataArray[arrayPos][4] = sh->currentLength;
				dataArray[arrayPos][5] = dotProdRatio;
				dataArray[arrayPos][6] = sqrt(sqDiffPos);
				dataArray[arrayPos][7] = sh->energy;

				totalModelLength += sh->currentLength;
				totalModelAbsorbance += sh->strandAbs;
				totalDiffusion += sqrt(sqDiffPos / segCount) ;
				totalEnergy += sh->energy;

				if(!totalModel){
					arrayPos++;
					if(arrayPos > DATALIMIT){
						printf("Data limit exceeded...exiting.\n");
						return(-1);
					}
				}
			}
			else{
				dataArray[arrayPos][0] += currentTime;
				dataArray[arrayPos][1] += sh->absChange;
				dataArray[arrayPos][2] += sqrt(sh->lCsqr/(stepsPerRun +
1));
				dataArray[arrayPos][3] += sh->strandAbs;
				dataArray[arrayPos][4] += sh->currentLength;
				dataArray[arrayPos][5] += dotProdRatio / segCount;
				dataArray[arrayPos][6] += sqrt(sqDiffPos);
				dataArray[arrayPos][7] = sh->energy;
			}
		}
		sh = sh->next;
	}

	if(constTemp && totalModel){
		dataArray[arrayPos][3] = totalModelAbsorbance;
		dataArray[arrayPos][4] = totalModelLength;
		dataArray[arrayPos][6] = totalDiffusion;
		dataArray[arrayPos][7] = totalEnergy;
		arrayPos++;
		if(arrayPos > DATALIMIT){
			printf("Data limit exceeded...exiting.\n");
			return(-1);
		}
		totalModelLength = 0;
		totalDiffusion = 0;
		totalModelAbsorbance = 0;
		totalEnergy = 0;
	}

	// Collect mass coord data
	if(vecExport && currentSimNum == (simRuns - 1) ){
		// This code is used for scattering analysis
```

```
		/*if(constTemp){
			while(mh != NULL){
				massPosArray[massArrayPos][0] = currentTemp;
				massPosArray[massArrayPos][1] = currentTime;
				massPosArray[massArrayPos][2] = mh->x;
				massPosArray[massArrayPos][3] = mh->y;
				massPosArray[massArrayPos][4] = mh->z;

				massArrayPos++;
				mh = mh->next;
			}
		}
		else{
			while(mh != NULL){
				massPosArray[massArrayPos][0] = currentTemp;
				massPosArray[massArrayPos][1] = currentTime;
				massPosArray[massArrayPos][2] = mh->x;
				massPosArray[massArrayPos][3] = mh->y;
				massPosArray[massArrayPos][4] = mh->z;

				mh = mh->next;
			}
		}*/
		snapFile = fopen(snapshotFile, "a+");

		fprintf(snapFile, "T %e\n", currentTime);

		while(mh != NULL){
			fprintf(snapFile, "M %e %e %e %e %d\n", mh->mass, mh->x, mh->y, mh-
>z, mh->brownianFlag);
			mh = mh->next;
		}

		fclose(snapFile);
	}
	return(0);
}
int writeStructData(SPRING_node *sh, MASS_node *mh)
{
	snapFile = fopen(snapshotFile, "a+");

	if(currentTemp >= targetTemp1 && currentTemp < targetTemp2){
		fprintf(snapFile, "T %.2f\n", currentTemp);

		while(mh != NULL){
			fprintf(snapFile, "M %e %e %e %e %d\n", mh->mass, mh->x, mh->y, mh-
>z, mh->brownianFlag);
			mh = mh->next;
		}
		/*while(sh != NULL){
			if(sh->nodeDef == 2){
				printf("S %d %d %d %d %.14e %.0f %.0f %.0f %.0f\n", \
					sh->segNum, sh->springRef, sh->springSec, sh->m1-
>massNum, sh->m2->massNum, \
					sh->restLength, sh->meltTemp - 273, sh->basePairs,
sh->extCoeff, sh->scaleFactor);
			}
			else if(sh->nodeDef > 2 && sh->nodeDef < 6){
				printf("S %d %d %d %d %.14e %.0f\n", \
					sh->segNum, sh->springRef, sh->springSec, sh->m1-
>massNum, sh->m2->massNum, \
					sh->restLength, sh->scaleFactor);
			}
```

```
                        sh = sh->next;
                }*/
                targetTemp1 += targetInc;
                if(targetTemp1 >= tempStop){
                        targetTemp1 = tempStop - tempInc;
                }
                targetTemp2 = targetTemp1 + tempInc;
        }
        fclose(snapFile);

        return(0);
}
int processData()
{
        int i = 0;

        while(i < dataPts){
                avgArray[i][0] += dataArray[i][0];
                avgArray[i][1] += dataArray[i][1];
                avgArray[i][2] += dataArray[i][2];
                avgArray[i][3] += dataArray[i][3];
                avgArray[i][4] += dataArray[i][4];
                avgArray[i][5] += dataArray[i][5];
                avgArray[i][6] += dataArray[i][6];
                avgArray[i][7] += dataArray[i][7];

                i++;
        }
        return(0);
}
int resetSim()
{
        arrayPos = 0;
        massArrayPos = 0;
        resetMassPos(mh);
        resetClengthRMS(sh);
        initCenterPos(mh);
        currentTime = 0;
        stepsPerRun = 0;

        return(0);
}
int resetMassPos(MASS_node *mh)
{
        while(mh != NULL){
                mh->x = mh->initx;
                mh->y = mh->inity;
                mh->z = mh->initz;

                mh = mh->next;
        }
        return(0);
}
int resetClengthRMS(SPRING_node *sh)
{
        while(sh != NULL){
                sh->lCsqr = 0;

                sh = sh->next;
        }
        return(0);
}
```

```
int avgData()
{
	int i = 0;

	while (i < dataPts){
		avgArray[i][0] /= simRuns;
		avgArray[i][1] /= simRuns;
		avgArray[i][2] /= simRuns;
		avgArray[i][3] /= simRuns;
		avgArray[i][4] /= simRuns;
		avgArray[i][5] /= simRuns;
		avgArray[i][6] /= simRuns;
		avgArray[i][7] /= simRuns;

		i++;
	}
	return(0);
}
int writeData()
{
	int i = 0;
	int j = 0;
	int k = 0;
	int l = 0;

	dataFile = fopen(outputFile, "wt");

	if(vecExport){
		massVecFile = fopen(massFile, "wt");
	}

	while(i < dataPts){
		j = 0;
		while(j < 8){
			fprintf(dataFile, "%e\t", avgArray[i][j]);
			j++;
		}
		fprintf(dataFile, "\n");
		i++;
	}

	if(vecExport){
		while(k < massArrayPos){
			l = 0;
			while(l < 5){
				fprintf(massVecFile, "%e\t", massPosArray[k][l]);
				l++;
			}
			fprintf(massVecFile, "\n");
			k++;
		}
	}

	fclose(dataFile);
	if(vecExport){
		fclose(massVecFile);
	}

	return 0;
}
int exportData()
{
	binFile = fopen(rawFile, "wb");
```

```
        fwrite(avgArray, sizeof(double), dataPts * 8, binFile);
        fclose(binFile);

        if(vecExport){
                vecFile = fopen(rawMassFile, "wb");
                fwrite(massPosArray, sizeof(double), massArrayPos * 5, vecFile);
                fclose(vecFile);
        }

        return(0);
}

int readBinData()
{
        int i = 0;
        int j = 0;
        int k = 0;
        int l = 0;

        dataFile = fopen(outputFile, "wt");
        binFile = fopen(rawFile, "rb");

        if(vecExport){
                massVecFile = fopen(massFile, "wt");
                vecFile = fopen(rawMassFile, "rb");
        }

        if(fread(outputArray, sizeof(double), dataPts * 8, binFile) != dataPts * 8){
                if(feof(binFile)){
                        printf("Premature end of file.\n");
                        return(-1);
                }
                else{
                        printf("File read error.\n");
                        return(-1);
                }
        }

        if(vecExport){
                if(fread(massPosArray, sizeof(double), massArrayPos * 5, vecFile) !=
massArrayPos * 5){
                        if(feof(vecFile)){
                                printf("Premature end of file.\n");
                                return(-1);
                        }
                        else{
                                printf("File read error.\n");
                                return(-1);
                        }
                }
        }
        while(i < dataPts){
                j = 0;
                while(j < 8){
                        fprintf(dataFile, "%e\t", outputArray[i][j]);
                        j++;
                }
                fprintf(dataFile, "\n");
                i++;
        }

        if(vecExport){
                while(k < massArrayPos){
```

```
                        l = 0;
                        while(l < 5){
                                fprintf(massVecFile, "%e\t", massPosArray[k][l]);
                                l++;
                        }
                        fprintf(massVecFile, "\n");
                        k++;
                }
        }

        fclose(dataFile);
        fclose(binFile);
        if(vecExport){
                fclose(massVecFile);
                fclose(vecFile);
        }
        return(0);
}
int releaseMem()
{
        // Clear the memory occupied by springs and masses
        while(mh != NULL)
        {
                mtemp = mh->next;
                free(mh);
                mh = mtemp;
        }
        while(sh != NULL)
        {
                stemp = sh->next;
                free(sh);
                sh = stemp;
        }
        return(0);
}
```

Computational functions for the DNA-STRAIN simulator

```
//-----------------------------------------//
//      Simulation Functions of DNA-STRAIN Sim       //
//      author: Vincent Mao                                          //
//      written: 9.11.09                                             //
//  last edited: 3.1.10                                              //
//-----------------------------------------//

/*-------------------------------------------------------------------
        This file contains all the functions that define springs, masses,
        and their properties.  All data is recorded in structs.  This file
        also contains the simulation functions and the data collection
        functions.
*/

#define _CRT_SECURE_NO_WARNINGS

// PPD's
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "Init_Structs.h"
```

```
#include "Comp_Vars.h"

/*-------------------------------------------------------------------------
        Initialization functions
        All struct values are assigned a value based on the information
        contained in the mesh file before running the simulation.
*/

int addMass(MASS_node **headptr, double mass, double x, double y, double z, int b)
{
        MASS_node *newm;       /* Pointer to the new mass */

        if ( (newm = (MASS_node *)malloc(sizeof(MASS_node))) == NULL )
        {
                perror("add_mass (cannot allocate more memory)");
                return(-1);    /* FAIL */
        }

        newm->next = *headptr; /* Insert before head of the list */
        *headptr   = newm;         /* Point the head to this entry */

        newm->mass                    = mass;
        newm->x                       = x;         // Units are in METERS
        newm->y                       = y;
        newm->z                       = z;
        newm->initx                   = x;
        newm->inity                   = y;
        newm->initz                   = z;
        newm->brownianFlag     = b;

        // Initialize all values not provided by the meshfile
        newm->damping          = 0;//3.05e-15; // 2 * sqrt(spring constant * mass)
        /*if(oscTest){
                newm->damping = 0;
        }*/
        newm->vx = newm->vy = newm->vz                        = 0.0;
        newm->fx = newm->fy = newm->fz                        = 0.0;

        /* Assign the mass to a mass node number that
           matches the one in the meshfile
        */
        newm->massNum  = massCount;
        massCount++;

    return(0);         /* SUCCESS */
}
/* end of add_mass */
/* end of overall mass routine */

/* This routine will add a spring to the head of a list of springs.
   It gets a pointer to the pointer to the head of the list.  It will put
   the new spring at the head of the list.  It works properly if the head
   of the list is NULL to begin with.  m1 and m2 should point to the masses
   at either end of the spring.
   The function returns -1 on failure and 0 on success.
*/
int     addSpring(SPRING_node **headptr,
                  int segNum, int springRef, int springSec,
                  double restLength, double meltTemp, double basePairs, double extCoeff,
double extCoeff2,
                      MASS_node *m1, MASS_node *m2,
                      int segRef1, int faceRef1, int segRef2, int faceRef2,
                      double scaleFactor, int nodeDef)
```

```
{
        SPRING_node    *news;  /* The new spring node location pointer */

        if ( (m1 == NULL) || (m2 == NULL) )
        {
                perror("addSpring (bad mass)");
                return(-1);     /* BAD MASS */
        }

        if ( (news = (SPRING_node *)malloc(sizeof(SPRING_node))) == NULL)
        {
                perror("addSpring (cannot allocate more memory)");
                return(-1);     /* FAIL */
        }

        news->next                    = *headptr; /* Place new entry at head of list */
        *headptr                      = news;    /* Make the head point to the new one */

    // Assign parameter values provided by the meshfile */
        news->segNum           = segNum;                      // Segment number
        news->springRef               = springRef;           // Spring number
        news->springSec               = springSec;           // Section number
        news->m1                      = m1;                // First mass node identifier
        news->m2                      = m2;                           // Second mass node
identifier
        news->restLength     = restLength;            // Rest length of the spring
        news->meltTemp       = meltTemp + 273;   // Melting temperature in K (C in
meshfile)
        news->basePairs               = basePairs;           // Number of bases in a
segment
        news->extCoeff       = extCoeff;          // Extinction coefficient for ssDNA
        news->extCoeff2               = extCoeff2;           // Extinction coefficient for
dsDNA
        news->scaleFactor      = scaleFactor;         // Spring constant scaling factor
        news->nodeDef          = nodeDef;                     // Type of spring

        // Initialize values
        news->marker1    = 0;
        news->marker2    = 0;
        news->face             = 0;
        news->k          = 0.0;
        news->sh_ref    = NULL;
        news->sh_ref2   = NULL;
        news->ref_k            = 0.0;
        news->K                = 0.0;
        news->f                = 0.0;
        news->H                = 0.0;
        news->angle            = 0.0;
        news->harmFreq  = 0.0;
        news->dx               = news->dy     = news->dz     = 0.0;
        news->dfx              = news->dfy = news->dfz = 0.0;
        news->crossx   = news->crossy = news->crossz = 0.0;
        news->dfx1             = news->dfy1 = news->dfz1 = 0.0;
        news->dfx2             = news->dfy2 = news->dfz2 = 0.0;
        news->currentLength           = restLength;
        news->lengthChange            = 0.0;
        news->lCsqr            = 0.0;
        news->energy    = 0.0;
        news->initAbs   = 0.0;
        news->meltAbs   = 0.0;
        news->absRange = 0.0;
        news->strandAbs = 0.0;
        news->brownianForce = 0.0;
```

```
	news->restoreForce = 0.0;

	if(news->nodeDef == 2){
		springCount++;
		initSpringConstant(news);
		initAbsorbance(news);
	}
	if(news->nodeDef == 8){
		initAbsorbance(news);
	}

	// Identify the mass nodes as being on one face or the other (0-3, 4-7)
	// This is done for every face in the normal model
	// nodeDef 4 is for the stiff spring, nodeDef 5 is for the connecting spring
	if(news->nodeDef == 4 || news->nodeDef == 5){
		if(news->m1->massNum % 8 < 4 && news->m2->massNum % 8 < 4){
			news->face = 1;
		}
		if(news->m1->massNum % 8 > 3 && news->m2->massNum % 8 > 3){
			news->face = 2;
		}

		// Identify which face is being monitored for deflection test
		// When complete, two faces should be marked
		if(news->segNum == segRef1 && news->face == faceRef1){
			news->marker1  = 1;
		}
		if(news->segNum == segRef2 && news->face == faceRef2){
			news->marker2  = 1;
		}
	}

	segCount       = 1;
	if(wlcModel == 1){
		segCount = springCount / 4;
	}
	totalSpringCount++;

	return(0);     /* SUCCESS */
}
/* end of addSpring */

int initSpringConstant(SPRING_node *sh)
{
	if(sh->basePairs > 1){
		// Avg NN bp enthalpy value is -8.7 kcal/mol (SantaLucia Jr. et al.) for
dsDNA
		sh->H = -8.7e3 * (sh->basePairs - 1);
	}
	else{
		sh->H = -8.7e3;
	}

	// Calculate fraction of ssDNA (D) (John et al.)
	sh->K = exp( -((sh->H) / R) * ( (1 / (sh->meltTemp)) - (1 / currentTemp)) ); //
negative sign used to calculate ssDNA fraction
	sh->f = (sh->K) / (sh->K + 1); // 0 to 1 as temperature increases

	// For the DNA spring, multiply by 1-f to characterize the degradation of the k
constant
	if(simpleFlag == 1){
		sh->k =  ( (1.5 * BOLTZMANN * tempStart) / (PLENGTH * (sh->restLength) ) )
* ((1 - sh->f)) * sh->scaleFactor;
```

```
                sh->ref_k = 0;
        }
        else{
                sh->k =  ( (1.5 * BOLTZMANN * tempStart) / (PLENGTH * (sh->restLength) ) )
* ((1 - sh->f)) * sh->scaleFactor / sfactor;// divide by number of brace springs in the
model;
                sh->ref_k = sh->k;
        }

        if(!oscTest){
                sh->m1->damping = 2 * sqrt(sh->k * sh->m1->mass);
                sh->m2->damping = 2 * sqrt(sh->k * sh->m1->mass);
        }
        else{
                sh->m1->damping = 0;
                sh->m2->damping = 0;
        }

        if(sh->k < 0){
                printf("ERROR: Bad spring constant...exiting\n");
                return(-1);
        }
        return(0);
}

int initAbsorbance(SPRING_node *sh)
{
        // Calculate the absorbance for a fully denatured structure per molecule
        sh->initAbs    = sh->extCoeff2 * concentration;// / AVOGADRO;              // A_f
        sh->meltAbs = sh->extCoeff * concentration;//  / AVOGADRO;          // A_0
        sh->absRange = (sh->initAbs - sh->meltAbs);  // alpha_0 = A_0 - A_f
        totalBasePairs += sh->basePairs * 0.25;
        totalModelAbsRange += sh->absRange * 0.25;
        totalModelAbsInit += sh->initAbs * 0.25;
        totalModelAbsFinal += sh->meltAbs * 0.25;

        return(0);
}

/* This function adds a mass to the head of a list of masses.  It gets
   a pointer to the pointer to the head of the list.  If the head of the
   list is null, then it is initialized.  In all cases, the new mass
   becomes the head of the mass list.
   The function returns -1 on failure, 0 on success.
*/
MASS_node* findmass(MASS_node **headptr, int MassIndex)
{
        MASS_node* pNode = *headptr;
        int nMassIndex = (massCount - 1) - MassIndex;
        int i;

        for(i = 0; i < nMassIndex; i++){
                pNode = pNode->next;
        }

        return pNode;
}

int refSpringSearch(SPRING_node *sh)
{
        SPRING_node *search = sh;
        SPRING_node *search2 = sh;
```

```
	int currentNum = 0;
	int currentRef = 0;
	int springType = -1;

	while(sh != NULL)
	{
		currentNum = sh->segNum;
		currentRef = sh->springRef;

		springType = sh->nodeDef;

		switch(springType)
		{
			// Assign 2-4 DNA springs to reference 1 for Brownian force
			case 2:
				if(sh->sh_ref == NULL && sh->springRef != 1){		// DNA
springs 2-4 - reference1 to 1
					sh->sh_ref = DNASpringSearch(search, currentNum);
				}
				if(sh->sh_ref2 == NULL && sh->springRef != 2){ // DNA
springs 1, 3 and 4 - reference2 to 2
					sh->sh_ref2 = DNASpringSearch2(search2, currentNum);
				}
				break;
			// Assign brace springs to their DNA springs
			case 3:
				if(sh->sh_ref == NULL){
					sh->sh_ref = braceSpringSearch(search, currentNum,
currentRef);
				}
				break;

			// Assign R, T and C springs to their DNA springs
			default:
				if(sh->sh_ref == NULL){
					sh->sh_ref = stiffSpringSearch(search, currentNum);
				}
				break;
		}
		sh = sh->next;
	}
	return(0);
}
SPRING_node *DNASpringSearch(SPRING_node *refsh, int currentSeg)
{
	while(refsh != NULL)
	{
		if(refsh->nodeDef == 2 && refsh->segNum == currentSeg && refsh->springRef
== 1){
			break;
		}
		refsh = refsh->next;
	}
	return(refsh);
}
SPRING_node *DNASpringSearch2(SPRING_node *refsh, int currentSeg)
{
	while(refsh != NULL)
	{
		if(refsh->nodeDef == 2 && refsh->segNum == currentSeg && refsh->springRef
== 2){
			break;
		}
```

```
                refsh = refsh->next;
        }
        return(refsh);
}
SPRING_node *braceSpringSearch(SPRING_node *refsh, int currentSeg, int currentRef)
{
        while(refsh != NULL)
        {
                if(refsh->nodeDef == 2 && refsh->segNum == currentSeg && refsh->springRef
== currentRef){
                        break;
                }
                refsh = refsh->next;
        }
        return(refsh);
}
SPRING_node *stiffSpringSearch(SPRING_node *refsh, int currentSeg)
{
        while(refsh != NULL)
        {
                if(refsh->nodeDef == 2 && refsh->segNum == currentSeg){
                        break;
                }
                refsh = refsh->next;
        }
        return(refsh);
}
int initSpringLength(SPRING_node *sh)
{
        while(sh != NULL){
                sh->dx = (sh->m2->x) - (sh->m1->x);
                sh->dy = (sh->m2->y) - (sh->m1->y);
                sh->dz = (sh->m2->z) - (sh->m1->z);

                sh->currentLength = sqrt((sh->dx * sh->dx) + (sh->dy * sh->dy) + (sh->dz *
sh->dz));
                sh->lCsqr = 0.0;
                sh = sh->next;
        }

        return(0);
}
int initSprConsts(SPRING_node *sh)
{
        int type = 0;

        while(sh != NULL){
                type = sh->nodeDef;

                switch(type)
                {
                        case 3: // Brace springs melt with their DNA springs
                                //sh->k = sh->sh_ref->ref_k * sh->scaleFactor / sh->angle;
                                sh->k = ( (1.5 * BOLTZMANN * currentTemp) / (PLENGTH * (sh-
>restLength) ) ) * ((1 - sh->sh_ref->f)) * sh->scaleFactor / sh->angle / sfactor2;
                                break;
                        case 4: // Stiff and Cross linked springs don't melt
                        case 8:
                                sh->k = ( (1.5 * BOLTZMANN * currentTemp) / (PLENGTH * (sh-
>restLength) ) ) * sh->scaleFactor;
                                break;
                        case 5: // Connecting and Crossover, Referring springs melt with
their assigned DNA springs but have their own k constants
```

```
                        case 7:
                                sh->k = ( (1.5 * BOLTZMANN * currentTemp) / (PLENGTH * (sh-
>restLength) ) ) * ((1 - sh->sh_ref->f)) * sh->scaleFactor;
                                break;
                }

                sh = sh->next;
        }
        return(0);
}
int initBraceAngles(SPRING_node *sh)
{
        while(sh != NULL)
        {
                if(sh->nodeDef == 3){
                        sh->angle = sh->sh_ref->currentLength / sh->currentLength;
                }
                else{
                        sh->angle = 0;
                }
                sh = sh->next;
        }
        return(0);
}

int initDefAngles(SPRING_node *sh)
{
        detCrossProds(sh);
        detDotProds(sh);
        return(0);
}
int detCrossProds(SPRING_node *sh)
{
        double x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4, z4;
        int check1, check2, check3, check4;
        check1 = check2 = check3 = check4 = 0;

        while(sh != NULL)
        {
                if(sh->marker1 == 1){
                        //printf("%d %d\n", sh->m1->massNum, sh->m2->massNum);
                        if(sh->m2->massNum % 4 == 1){
                                x1 = sh->m2->x - sh->m1->x;
                                y1 = sh->m2->y - sh->m1->y;
                                z1 = sh->m2->z - sh->m1->z;
                                check1 = 1;
                                /*printf("MARKER 1 FOUND\n");
                                printf("%d %d %e %e %e\n\n", sh->springRef, sh->springSec,
x1,y1, z1);*/
                        }
                        if(sh->m2->massNum % 4 == 2){
                                x2 = sh->m2->x - sh->m1->x;
                                y2 = sh->m2->y - sh->m1->y;
                                z2 = sh->m2->z - sh->m1->z;
                                check2 = 1;
                                /*printf("MARKER 1 FOUND\n");
                                printf("%d %d %e %e %e\n\n", sh->springRef, sh->springSec,
x2,y2, z2);*/
                        }
                }
                if(sh->marker2 == 1){
                        if(sh->m2->massNum % 4 == 1){
                                x3 = sh->m2->x - sh->m1->x;
```

```
                                y3 = sh->m2->y - sh->m1->y;
                                z3 = sh->m2->z - sh->m1->z;
                                check3 = 1;
                                /*printf("MARKER 2 FOUND\n");
                                printf("%d %d %e %e %e\n\n", sh->springRef, sh->springSec,
x3,y3, z3);*/
                        }
                        if(sh->m2->massNum % 4 == 2){
                                x4 = sh->m2->x - sh->m1->x;
                                y4 = sh->m2->y - sh->m1->y;
                                z4 = sh->m2->z - sh->m1->z;
                                check4 = 1;
                                /*printf("MARKER 2 FOUND\n");
                                printf("%d %d %e %e %e\n\n", sh->springRef, sh->springSec,
x4,y4, z4);*/
                        }
                }
                if(check1 == 1 && check2 == 1){
                        sh->crossx = (y1*z2 - z1*y2);
                        sh->crossy = (z1*x2 - x1*z2);
                        sh->crossz = (x1*y2 - y1*x2);
                        //printf("%e %e %e\n", sh->crossx, sh->crossy, sh->crossz);
                        check1 = 0;
                        check2 = 0;
                }
                if(check3 == 1 && check4 == 1){
                        sh->crossx = (y3*z4 - z3*y4);
                        sh->crossy = (z3*x4 - x3*z4);
                        sh->crossz = (x3*y4 - y3*x4);
                        //printf("%e %e %e\n", sh->crossx, sh->crossy, sh->crossz);
                        check3 = 0;
                        check4 = 0;
                }
                sh = sh->next;
        }
        return(0);
}

int detDotProds(SPRING_node *sh)
{
        double dotProd = 0;

        double angle = 0;
        double length1 = 0;
        double length2 = 0;

        double  x1, y1, z1, x2, y2, z2;

        SPRING_node *refsh = sh;

        while(refsh != NULL)
        {
                if(refsh->marker1 == 1){
                        x1 = refsh->crossx;
                        y1 = refsh->crossy;
                        z1 = refsh->crossz;

                        length1 = sqrt(x1*x1 + y1*y1 + z1*z1);
                }
                if(refsh->marker2 == 1){
                        x2 = refsh->crossx;
                        y2 = refsh->crossy;
                        z2 = refsh->crossz;
```

```
                        length2 = sqrt(x2*x2 + y2*y2 + z2*z2);
                }
                refsh = refsh->next;
        }
        angle = length1 / length2;

        dotProd = x1*x2 + y1*y2 + z1*z2;
        dotProdRatio = dotProd / (length1 * length2);

        return(0);
}
int initCenterPos(MASS_node *mh)
{
        cx = cy = cz = 0.0;

        while (mh != NULL)
    {
                cx += mh->x / massCount;
                cy += mh->y / massCount;
                cz += mh->z / massCount;

                mh = mh->next;
        }
        return(0);
}
int detTimestep(SPRING_node *sh)
{
        double minTimeStep = 0;
        double tempSprConst = 0;
        double maxSprConst = 0;
        double testTemp, rateConst, f;

        // Find the maximum spring constant value for each segment
        while(sh != NULL)
        {
                if(sh->nodeDef == 2)
                {
                        testTemp = 253;
                        while(testTemp <= sh->meltTemp)
                        {
                                rateConst = exp( -(sh->H / R) * (1 / (sh->meltTemp) - 1 /
testTemp) );
                                f = rateConst / (rateConst + 1);

                                tempSprConst = ( (1.5 * BOLTZMANN * testTemp) / (PLENGTH *
sh->restLength) ) * (1-f);

                                // Find the max k over the temp ramp for graphics
                                if(tempSprConst > maxSprConst){
                                        maxSprConst = tempSprConst;
                                }
                                testTemp += tempInc;
                        }
                        // The natural oscillation frequency is calculated based on the
mass at the end of the spring
                        // Treat this calculation based on the simple 2 mass/1 spring model
                        // The time step is reduced by the TIMESTEPDIV value
                        if(simpleFlag){
                                sh->harmFreq = sqrt((maxSprConst * sh->scaleFactor) / (sh-
>m1->mass)) / (2.0 * M_PI);
                        }
                        else{
```

```
                                sh->harmFreq = sqrt((maxSprConst * sh->scaleFactor) / (4 *
sh->m1->mass)) / (2.0 * M_PI);
                        }

                        minTimeStep = (1.0 / sh->harmFreq) / TIMESTEPDIV;
                        if(maxSprConst == 0){
                                printf("Time_step(): spring constant = 0...");
                                return(-1);     /* FAIL */
                        }
                        // Compare the max k for all DNA segments
                        if(minTimeStep < timeStep){
                                timeStep = minTimeStep;
                        }
                }
                sh = sh->next;
    }
        return(0);
}
/* end of initialization functions */

/*-------------------------------------------------------------------------
        Simulation functions
*/
int updateDNASprConsts(SPRING_node *sh)
{
        while(sh != NULL)
        {
                if(sh->nodeDef == 2){
                        sh->K = exp((-(sh->H) / R) * ((1 / (sh->meltTemp)) - (1 /
currentTemp)));
                        sh->f = (sh->K) / (sh->K + 1);
                        // For the DNA spring, multiply by 1-f to account for ssDNA
                        if(simpleFlag){
                                sh->k =  ( (1.5 * BOLTZMANN * currentTemp) / (PLENGTH *
(sh->restLength) ) ) * ((1- sh->f)) * sh->scaleFactor;
                                sh->ref_k = 0;
                        }
                        else{
                                sh->k =  ( (1.5 * BOLTZMANN * currentTemp) / (PLENGTH *
(sh->restLength) ) ) * ((1 - sh->f)) * sh->scaleFactor / sfactor;
                                sh->ref_k = sh->k;
                        }
                }
                sh = sh->next;
        }
        return(0);
}
int updateBraceSprConsts(SPRING_node *sh)
{
        while(sh != NULL)
        {
                if(sh->nodeDef == 3){ // Brace springs
                        sh->k = ( (1.5 * BOLTZMANN * currentTemp) / (PLENGTH * (sh-
>restLength) ) ) * ((1 - sh->sh_ref->f)) * sh->scaleFactor / sfactor2 / sh->angle;
                }
                // added for ds -> ss model
                if(sh->nodeDef == 5 || sh->nodeDef == 7){ // Crossover/Connecting and
Referring springs
                        sh->k = ( (1.5 * BOLTZMANN * currentTemp) / (PLENGTH * (sh-
>restLength) ) ) * ((1 - sh->sh_ref->f)) * sh->scaleFactor;
                }
                if(sh->nodeDef == 4 || sh->nodeDef == 8){ // Stiff and Cross linked
springs
```

```
                        sh->k = ( (1.5 * BOLTZMANN * currentTemp) / (PLENGTH * (sh-
>restLength) ) ) * sh->scaleFactor;
                }

                sh = sh->next;
        }
        return(0);
}
int generateRN(SPRING_node *sh)
{
        double alpha1, beta1, gamma1, alpha2, beta2, gamma2, normFactor1, normFactor2;
        double RN1, RN2, RN3, RN4, RN5, RN6;
        double scale, scaleMass, segment, random;
        double brownianForce;

        /* Random number generator
                rm = 100;
                n = 0,99;
                n'= n-50; --> 99-50=49; 0-50=-50;
                n'= n/50; --> ~1... -1
        */
        random = 0.5 * RAND_MAX;

        // Brownian force equation: [(3/2)*((k_B*T)/P)] * mass-dependent scaling factor
expression
        scaleMass = sh->m1->mass * 1e25 * massCount; /* Calculate the total mass of the
DNA */

    scale = 0.0962 * pow(scaleMass, 0.841);

        /*if(simpleFlag){
                scale = 0.0421 * pow(scaleMass, 0.8132);
        }*/

        /* Scale Brownian force by the segments in WLC and the viscosity changes in the
medium */
        segment = 1;
        if(segCount > 1 && wlcModel){
                segment = -0.232 * log( (double)segCount) + 1.056;
        }

        brownianForce = 1.5 * ((BOLTZMANN * currentTemp) / PLENGTH) / (scale * segment);
        sh->brownianForce = brownianForce;

        RN1 = ((double)rand() - random) / random;
        RN2 = ((double)rand() - random) / random;
        RN3 = ((double)rand() - random) / random;
        RN4 = ((double)rand() - random) / random;
        RN5 = ((double)rand() - random) / random;
        RN6 = ((double)rand() - random) / random;

        normFactor1 = sqrt(RN1*RN1 + RN2*RN2 + RN3*RN3);
        normFactor2 = sqrt(RN4*RN4 + RN5*RN5 + RN6*RN6);

        // Generating random unit vector coefficients for the mass
        alpha1 = RN1 / normFactor1;
        beta1  = RN2 / normFactor1;
        gamma1 = RN3 / normFactor1;

        alpha2 = RN4 / normFactor2;
        beta2  = RN5 / normFactor2;
        gamma2 = RN6 / normFactor2;
```

```
        sh->dfx1 = brownianForce * alpha1;
        sh->dfy1 = brownianForce * beta1;
        sh->dfz1 = brownianForce * gamma1;

        sh->dfx2 = brownianForce * alpha2;
        sh->dfy2 = brownianForce * beta2;
        sh->dfz2 = brownianForce * gamma2;

        return(0);
}
int applyBrownian(SPRING_node *sh)
{
        SPRING_node *reset = sh;

        while(sh != NULL)
        {
                if(currentTemp <= sh->meltTemp){
                        if(sh->nodeDef == 2 && sh->springRef == 1){
                                generateRN(sh);
                                // Initialize the coordinate forces for DNA Spring 1 in
each segment while the structure is dsDNA
                                if(sh->m1->brownianFlag == 0){
                                        sh->m1->fx += sh->dfx1;
                                        sh->m1->fy += sh->dfy1;
                                        sh->m1->fz += sh->dfz1;
                                }
                                if(sh->m2->brownianFlag == 0){
                                        sh->m2->fx += sh->dfx2;
                                        sh->m2->fy += sh->dfy2;
                                        sh->m2->fz += sh->dfz2;
                                }
                        }
                }
                else{
                        if(sh->nodeDef == 2 && (sh->springRef == 1 || sh->springRef == 2)){
                                generateRN(sh);
                                // Initialize the coordinate forces for DNA Spring 1 and 2
in each segment, the second set of forces for ds->ss model
                                if(sh->m1->brownianFlag == 0){
                                        sh->m1->fx += sh->dfx1;
                                        sh->m1->fy += sh->dfy1;
                                        sh->m1->fz += sh->dfz1;
                                }
                                if(sh->m2->brownianFlag == 0){
                                        sh->m2->fx += sh->dfx2;
                                        sh->m2->fy += sh->dfy2;
                                        sh->m2->fz += sh->dfz2;
                                }
                        }
                }
                sh = sh->next;
        }

        sh = reset;

        while(sh != NULL)
        {
                if(currentTemp <= sh->meltTemp){
                        if(sh->nodeDef == 2 && sh->springRef != 1){
                                // Assign the coordinate forces to be the same as DNA
Spring 1 in each segment while structure is dsDNA
                                if(sh->m1->brownianFlag == 0){
                                        sh->m1->fx = sh->sh_ref->m1->fx;
```

```
                                        sh->m1->fy = sh->sh_ref->m1->fy;
                                        sh->m1->fz = sh->sh_ref->m1->fz;
                                }
                                if(sh->m2->brownianFlag == 0){
                                        sh->m2->fx = sh->sh_ref->m2->fx;
                                        sh->m2->fy = sh->sh_ref->m2->fy;
                                        sh->m2->fz = sh->sh_ref->m2->fz;
                                }
                        }
                }
                else{
                        if(sh->nodeDef == 2 && (sh->springRef != 1 && sh->springRef != 2)){
                                // Assign the coordinate forces to be the same as DNA
Spring 1 or 2 in each segment for ds->ss model
                                if(sh->springRef == 3){       // Assign forces for DNA
spring 3 to be the same as 1
                                        if(sh->m1->brownianFlag == 0){
                                                sh->m1->fx = sh->sh_ref->m1->fx;
                                                sh->m1->fy = sh->sh_ref->m1->fy;
                                                sh->m1->fz = sh->sh_ref->m1->fz;
                                        }
                                        if(sh->m2->brownianFlag == 0){
                                                sh->m2->fx = sh->sh_ref->m2->fx;
                                                sh->m2->fy = sh->sh_ref->m2->fy;
                                                sh->m2->fz = sh->sh_ref->m2->fz;
                                        }
                                }
                                if(sh->springRef == 4){       // Assign forces for DNA
spring 4 to be the same as 2
                                        if(sh->m1->brownianFlag == 0){
                                                sh->m1->fx = sh->sh_ref2->m1->fx;
                                                sh->m1->fy = sh->sh_ref2->m1->fy;
                                                sh->m1->fz = sh->sh_ref2->m1->fz;
                                        }
                                        if(sh->m2->brownianFlag == 0){
                                                sh->m2->fx = sh->sh_ref2->m2->fx;
                                                sh->m2->fy = sh->sh_ref2->m2->fy;
                                                sh->m2->fz = sh->sh_ref2->m2->fz;
                                        }
                                }
                        }
                }
                sh = sh->next;
        }

        return(0);
}
int applyConstForce(SPRING_node *sh)
{
        double alpha, beta, gamma, brownianForce;
        double scale, scaleMass, segment;
        double dfx, dfy, dfz;

        while(sh != NULL)
        {
                // Apply force to all coordinates for each mass
                if(sh->nodeDef == 2)
                {
                        sh->dx = (sh->m2->x) - (sh->m1->x);
                        sh->dy = (sh->m2->y) - (sh->m1->y);
                        sh->dz = (sh->m2->z) - (sh->m1->z);
                        sh->currentLength = sqrt( sh->dx*sh->dx + sh->dy*sh->dy + sh-
>dz*sh->dz );
```

```
                        // Calculate the unit vector components
                        alpha = sh->dx / sh->currentLength;
                        beta  = sh->dy / sh->currentLength;
                        gamma = sh->dz / sh->currentLength;

                        // Brownian force equation: [(3/2)*((k_B*T)/P)] * base pair-
dependent scaling factor expression
                        scaleMass = sh->m1->mass * 1e25;
                        scale = 0.761 * pow(scaleMass, 0.8456);
                        segment = 1;
                        if(segCount > 1){
                                segment = 1.0344 * pow(segCount, 0.4427);
                        }

                        // Brownian force equation: [(3/2)*((k_B*T)/P)] * (1/10)
                        brownianForce = 0.15 * ((BOLTZMANN * currentTemp) / PLENGTH) /
(scale * segment);

                        dfx = brownianForce * alpha;
                        dfy = brownianForce * beta;
                        dfz = brownianForce * gamma;

                //      if(sh->avgcount < 50){ // Activate for periodic application of
force
                                sh->m1->fx -= dfx;
                                sh->m2->fx += dfx;

                                sh->m1->fy -= dfy;
                                sh->m2->fy += dfy;

                                sh->m1->fz -= dfz;
                                sh->m2->fz += dfz;
                //      }
                        }
                        sh = sh->next;
        }
        return(0);
}
int updateDefAngles(SPRING_node *sh)
{
        detCrossProds(sh);
        detDotProds(sh);
        return(0);
}
int updateCenterPos(MASS_node *mh)
{
        px = py = pz = 0.0;

        while (mh != NULL)
    {
                px += mh->x / massCount;
                py += mh->y / massCount;
                pz += mh->z / massCount;

                mh = mh->next;
        }
        return(0);
}
int updateDiffusionPos(MASS_node *mh)
{
        double diffPos = 0;
```

```
        updateCenterPos(mh);

        // Take the vector of the current position from the original position
        diffx = px - cx;
        diffy = py - cy;
        diffz = pz - cz;
        diffPos = sqrt((diffx * diffx) + (diffy * diffy) + (diffz * diffz));
        sqDiffPos = diffPos * diffPos;

        return(0);
}
int processSprings(SPRING_node *sh)
{
        while(sh != NULL){
                updateSpringLength(sh);
                initBraceAngles(sh);
                updateRestoringForce(sh);
                if(sh->nodeDef == 2 || sh->nodeDef == 8){
                        updateAbsorbance(sh);
                }
                sh = sh->next;
        }
        return(0);
}
int updateSpringLength(SPRING_node *sh)
{

        sh->dx = (sh->m2->x) - (sh->m1->x);
        sh->dy = (sh->m2->y) - (sh->m1->y);
        sh->dz = (sh->m2->z) - (sh->m1->z);

        sh->currentLength = sqrt((sh->dx * sh->dx) + (sh->dy * sh->dy) + (sh->dz * sh-
>dz));

        // This is for limiting the dsDNA springs from going any further than their max
length after melting (ds->ss model)
        /*if(sh->currentLength > (2.5 * sh->restLength) && currentTemp > sh->meltTemp){
                sh->currentLength = 2.5 * sh->restLength;
        }*/

        sh->lengthChange = sh->currentLength - sh->restLength;
        sh->lCsqr += sh->lengthChange * sh->lengthChange;

        return(0);
}

int updateRestoringForce(SPRING_node *sh)
{
        double restoreForce    = 0.0;
        double scaledResForce = 0.0;
        double tempx, tempy, tempz;

        restoreForce = -sh->k * sh->lengthChange;
        sh->restoreForce = restoreForce;
        // Check if it's a static single strand.
        if((sh->lengthChange / sh->restLength < 1e-9) && (sh->lengthChange / sh-
>restLength > -1e-9)){
                scaledResForce = 0;
        }
        else{
                // Divide the force between 2 masses
                scaledResForce = restoreForce * 0.5;
                sh->energy = 0.5 * sh->k * sh->lengthChange * sh->lengthChange;
```

```
        }

        // Calculate the unit vector components of the spring
        tempx = (sh->dx / sh->currentLength);
        tempy = (sh->dy / sh->currentLength);
        tempz = (sh->dz / sh->currentLength);

        // This calculates the unit vector scaled force
        sh->dfx = tempx * scaledResForce;
        sh->dfy = tempy * scaledResForce;
        sh->dfz = tempz * scaledResForce;

        sh->m1->fx -= sh->dfx;
        sh->m1->fy -= sh->dfy;
        sh->m1->fz -= sh->dfz;
        sh->m2->fx += sh->dfx;
        sh->m2->fy += sh->dfy;
        sh->m2->fz += sh->dfz;

        sh->dfx = sh->dfy = sh->dfz = 0.0;

        return(0);
}

int moveMasses(MASS_node *mh)
{
        /* Acceleration in x,y, and z coords */
        double ax, ay, az;

        while (mh != NULL)
    {
                // Forces acting on the object = Force there - damping * velocity
                mh->fx -= mh->damping * mh->vx;
                mh->fy -= mh->damping * mh->vy;
                mh->fz -= mh->damping * mh->vz;

         // Calculate the acceleration based on mh->fd from the Calc_res_forces()
function.
                // No gravity effects on the mass...otherwise add 9.8 to ay.
                // accel = force / mass
                ax = mh->fx / mh->mass;
                ay = mh->fy / mh->mass;
                az = mh->fz / mh->mass;

                // Current velocity = accel * timestep
                mh->vx += ax * timeStep; // (=) to remove the momentum, (+=) with critical
damping
                mh->vy += ay * timeStep;
                mh->vz += az * timeStep;

                // Apply the displacement on the coordinates based on the timestep
                // coord = coord + current velocity * time step
                mh->x += mh->vx * timeStep;
                mh->y += mh->vy * timeStep;
                mh->z += mh->vz * timeStep;

                // Clear values for the next round
                mh->fx = mh->fy = mh->fz = 0.0;

                mh = mh->next;
        }
        return(0);
}
```

```
int updateAbsorbance(SPRING_node *sh)
{
        // Calculate absorbance based on the derivation
        // The average difference between dsDNA and ssDNA is based on analysis from
Tataurov et al. '08
        alpha0 = sh->absRange;
        alpha1 = alpha0 / BPLENGTH;
        sh->absChange = alpha1 * sqrt(sh->lCsqr/(stepsPerRun + 1)) * (1 - sh->f) * (1 -
sh->f);
        sh->strandAbs = sh->meltAbs + sh->absChange;
        return(0);
}
/* end of simulation functions */
```

Global variable declaration file for the DNA-STRAIN simulator

```
//--------------------------------------//
//      Global variables for Main.cpp         //
//      author: Vincent Mao                                 //
//      written: 9.11.09                                    //
//  last edited: 10.12.09                           //
//--------------------------------------//

#ifndef _GLOBALS_H_
#define _GLOBALS_H_

#define _CRT_SECURE_NO_WARNINGS

#ifndef NULL
#define NULL    0
#endif

#ifndef DATALIMIT
#define DATALIMIT      65536
#endif

// Default values if not specified by the meshfile
double tempStart       = 253;
double tempStop                = 303;
double targetTemp1     = 0;
double targetTemp2     = 0;
double tempRange       = 0;
double currentTemp     = 0;
double tempInc         = 10;
double targetInc       = 0;
double rampRate                = 10;
double timeStep                = 10;          // Time step initialized

// Initialize command flags
int testMode           = 0;
int meshOutput         = 0;
int brownianOn         = 0;
int simpleFlag         = 0;
int wlcModel           = 0;
int nameFiles          = 0;
int numRuns                    = 0;
int constTemp          = 0;
int vecExport          = 0;
int totalModel         = 0;
int snapShot           = 0;
```

```
int oscTest                    = 0;
int constForce         = 0;
int factor                     = 0;
int binary                     = 0;

// Default variable values
int simRuns                    = 1;
int currentSimNum      = 0;

char srcFile[50];
char outputFile[50];
char rawFile[50];
char massFile[50];
char rawMassFile[50];
char snapshotFile[50];

FILE *meshFile         = NULL;
FILE *dataFile                 = NULL;
FILE *binFile          = NULL;
FILE *vecFile          = NULL;
FILE *massVecFile      = NULL;
FILE *snapFile         = NULL;

MASS_node *mh          = NULL;         /* Point to first node in mass list */
SPRING_node    *sh             = NULL;         /* Points to first node in spring list */

// Variables from other functions
extern int massCount;
extern int totalSpringCount;
extern int segCount;

// Simulation variables
double currentTime     = 0;            // The current simulation time point
double simTime         = 5e-8;         // Simulation time in seconds
double concentration= 625E-8; // Sample concentration
int dataRes                    = 100;          // Data resolution; record every nth point
int targetRes          = 2;            // Snapshot resolution; record every nth temperature

extern double totalBasePairs;
extern double dotProdRatio;

double dataArray[DATALIMIT][8]                 = [118];
double avgArray[DATALIMIT][8]          = [118];
double outputArray[DATALIMIT][8]       = [118];
double massPosArray[DATALIMIT][5]      = [118];

int arrayPos           = 0;
int massArrayPos       = 0;
int stepsPerRun                = 0;
int stepsPerMelt       = 0;
int dataPts                    = 0;
double totalModelLength                = 0;
double totalDiffusion          = 0;
double totalModelAbsorbance = 0;
double totalEnergy                     = 0;

extern double cx, cy ,cz;
extern double px, py ,pz;
extern double sqDiffPos;
extern double totalModelAbsRange;
extern double totalModelAbsInit;
extern double totalModelAbsFinal;
```

```
/* For releasing the memory after a run */
MASS_node      *mtemp  = NULL;
SPRING_node    *stemp  = NULL;

#endif
```

Initialization of functions for the DNA-STRAIN simulator

```
//-----------------------------------------------//
//      Global variables for Computation.cpp         //
//      author: Vincent Mao                                                    //
//      written: 9.11.09                                                       //
//  last edited: 1.20.10                                              //
//-----------------------------------------------//

#ifndef _INIT_STRUCTS_H_
#define _INIT_STRUCTS_H_

#define _CRT_SECURE_NO_WARNINGS

#ifndef NULL
#define NULL    0
#endif

typedef struct MASS /* Used to allow a pointer to this type in struct */
{
        // Defined by the mesh file
        double  mass;                   /* The mass of this point */
        double  x, y, z;                /* Mass coordinates */
        int             massNum;                /* Mass number assigned as identifier */
        int             brownianFlag;   /* Flag for brownian force */

        double  initx, inity, initz;    /* Initial positions of each mass */
        double  damping;                                /* Mass damping coefficient */
        double   vx,vy,vz;                              /* The velocity of this point */
        double  fx,fy,fz;                               /* The forces acting on this point */
        struct  MASS *next;                             /* The next mass in the list */
}
MASS_node;

typedef struct SPRING /* Used to allow a pointer to this type in struct */
{
        // Defined by the mesh file
        int             segNum;                 // Segment number identifier
        int             springRef;              // Spring reference number for the B spring
to find its S spring
        int             springSec;              // Assign the T spring the B spring is
associated with
        double  restLength;             // The rest length of the spring
        double   meltTemp;       // Melting temperature of the spring at which k = 0.5*k
        double   basePairs;             // The number of base pairs in the segment
        double   extCoeff;              // ssDNA extinction coefficient
        double  extCoeff2;              // dsDNA extinction coefficient
        double  scaleFactor;    // Scaling factor for the spring
        int             marker1, marker2;
        int             face;

        //Calculated after reading values from the mesh file
        int             nodeDef;                // Spring type identifier
        double  H;                              // Enthalpy of dsDNA
```

```
        double  K;                              // Dissociation rate constant
        double  f;                              // Fraction of ssDNA
        double  k;                              // Spring constant
        double  ref_k;                 // Reference spring constant from the DNA spring for
structural springs
        double  angle;                 // Angle formed between DNA and brace springs
        double   harmFreq;       // Harmonic frequency of oscillation
        double  lengthChange;  // The difference in length from rest
        double  lCsqr;                 // The RMS of change in length
        double  currentLength; // The current length of the spring
        double  dx, dy, dz;            // Change in coords for spring length
        double  energy;                // Kinetic energy of the spring
        double  dfx, dfy, dfz; // Change in force in springs
        double  initAbs;               // Initialized absorbance for strands
        double  meltAbs;               // Absorbance for fuller denatured strands
        double  absRange;              // Absorbance range from native to denatured strands
        double   strandAbs;      // Absorbance the strand contributes
        double  absChange;             // Change in absorbance based on change in strand
length
        double  crossx, crossy, crossz;        // Cross product coords
        double  dfx1, dfy1, dfz1;              // Change in Brownian force for masses
        double  dfx2, dfy2, dfz2;
        double   brownianForce;
        double  restoreForce;

        MASS_node *m1;                 // Pointer to first mass attached to the spring
        MASS_node *m2;                 // Pointer to second mass attatched to the spring

        struct SPRING *sh_ref; // The reference spring for structural springs
        struct SPRING *sh_ref2;        // A second reference spring for when dsDNA becomes
ssDNA
        struct SPRING  *next;  // Points to the next spring in the list
}
SPRING_node;

extern MASS_node* findmass(MASS_node **headptr, int massIndex);

extern  int     addMass(MASS_node **headptr,
                        double mass, double x, double y, double z, int b);

extern  int     addSpring(SPRING_node **headptr,
                    int segNum, int springRef, int springSec,
                    double restLength, double meltTemp, double bases, double extCoeff,
double extCoeff2,
                        MASS_node *m1, MASS_node *m2,
                        int segRef1, int faceRef1, int segRef2, int faceRef2,
                        double scaleFactor, int nodeDef);

int initSpringConstant(SPRING_node *sh);

int initAbsorbance(SPRING_node *sh);

extern int refSpringSearch(SPRING_node *sh);

extern SPRING_node *DNASpringSearch(SPRING_node *refsh, int currentSeg);

extern SPRING_node *DNASpringSearch2(SPRING_node *refsh, int currentSeg);

extern SPRING_node *braceSpringSearch(SPRING_node *refsh, int currentSeg, int
currentRef);

extern SPRING_node *stiffSpringSearch(SPRING_node *refsh, int currentSeg);
```

```
extern int initSpringLength(SPRING_node *sh);

extern int initBraceAngles(SPRING_node *sh);

extern int initSprConsts(SPRING_node *sh);

extern int initDefAngles(SPRING_node *sh);

int detCrossProds(SPRING_node *sh);

int detDotProds(SPRING_node *sh);

extern int initCenterPos(MASS_node *mh);

extern int detTimestep(SPRING_node *sh);

int updateDNASprConsts(SPRING_node *sh);

int updateBraceSprConsts(SPRING_node *sh);

int generateRN(SPRING_node *sh);

extern int applyBrownian(SPRING_node *sh);

extern int applyConstForce(SPRING_node *sh);

extern int updateDefAngles(SPRING_node *sh);

extern int updateDiffusionPos(MASS_node *mh);

extern int processSprings(SPRING_node *sh);

extern int updateSpringLength(SPRING_node *sh);

extern int updateRestoringForce(SPRING_node *sh);

extern int moveMasses(MASS_node *mh);

extern int updateAbsorbance(SPRING_node *sh);

extern int resetMassPos(MASS_node *mh);

extern int resetClengthRMS(SPRING_node *sh);

#endif
```

Variable declarations for the DNA-STRAIN simulator

```
//-----------------------------------------------//
//      Computation variables for Computation.cpp    //
//      author: Vincent Mao                                                    //
//      written: 9.11.09                                                       //
//  last edited: 3.1.10                                                        //
//-----------------------------------------------//

#ifndef _COMP_VARS_H_
#define _COMP_VARS_H_

#define _CRT_SECURE_NO_WARNINGS

#ifndef NULL
```

```
#define NULL    0
#endif

#ifndef M_PI
#define M_PI 3.1415926               // PI
#endif

#define BPMASS         0.660         // Mass of a base pair in kDaltons (g/L)-->kg/mol
#define AVOGADRO       6.022E23      // Avogadro's number
#define BPLENGTH       3.4E-10     // Contour length of a single dsDNA base pair is 0.34
angstroms
#define PLENGTH               53E-9       // Persistence length for dsDNA is 53 nm
#define BOLTZMANN      1.38E-23    // Boltzmann constant in Nm/K
#define DNA_DIAM       2e-9          // Diameter of DNA is 2 nm
#define R                     1.98719        // Gas constant in cal/mol
#define TIMESTEPDIV    51                    // Resolution of the time step

int massCount          = 0;          // Total mass nodes in the structure
int springCount               = 0;          // Total DNA springs in the structure
int segCount           = 0;          // Total number of segments in the structure for WLC
model
int totalSpringCount= 0;             // Total springs in the structure

int sfactor                   = 5;
int sfactor2           = 20;

extern int stepsPerRun;
extern int stepsPerMelt;

double totalBasePairs         = 0.0;
double totalAbsorbance        = 0.0;
double alpha0                         = 0.0;
double alpha1                         = 0.0;
double dotProdRatio                   = 0.0;
double totalModelAbsRange     = 0.0;
double totalModelAbsInit      = 0.0;
double totalModelAbsFinal     = 0.0;

extern double tempStart;
extern double currentTemp;
extern double tempInc;
extern int simpleFlag;
extern int wlcModel;
extern int oscTest;
extern int constForce;
extern double timeStep;
extern double concentration;

double cx, cy, cz;
double px, py, pz;
double diffx, diffy, diffz;
double sqDiffPos;

#endif
```

4. The Input Mesh File Generator

The input mesh file specifies the initial values for the mass nodes and the springs in the model. There are four types of identifiers in the input file: 1) the user-defined settings, 2) the deflection identifier, 3) the mass identifier and 4) the spring identifier. The following paragraphs describe each in detail.

The "U" defines the user identifier in which the starting and ending temperatures are specified in Kelvin, followed by the molar concentration, the simulation time in seconds, the time step resolution for recording data and the resolution for recording mass coordinate points during simulation. An example "U" line is presented in Figure 66.

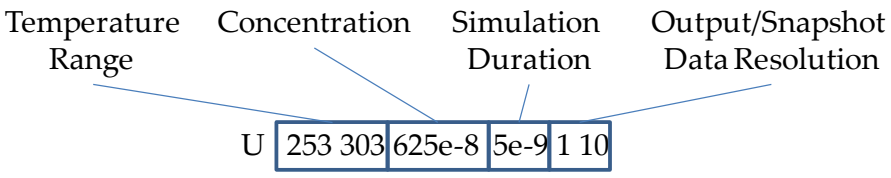


Figure 66: User defined simulation settings in the input file.

In the example above, the user identifier line sets the temperature range to be over a 50 K range from 253 K to 303 K. The input temperatures are in Kelvin because the calculations in the simulator are in these units. The concentration of the DNA sample is 62.5 μM, followed by a 5 ns simulation time period per temperature increment. The output data resolution is set at one, indicating that each time step is recorded. A number

higher than one will cause a decrease in data resolution. The final number is the snapshot data file resolution, indicating that the mass node positions should be recorded every tenth of a degree in resolution. A higher number will result in an increase in the snapshot resolution.

The "D" identifier defines the rectangular faces that are identified for calculation of the extent of deflection for a multi-segment 3D spring model, referred to as a deflection ratio. The four identifier numbers following the identifier indicate the first segment, the first face, the second segment and the second face needed to calculate the deflection. The range of segment numbers depends on the number of segments in the model. The face number is either "1", being the left face of the rectangular segment, or "2", being the right face of the segment. An example is illustrated in the following diagram:

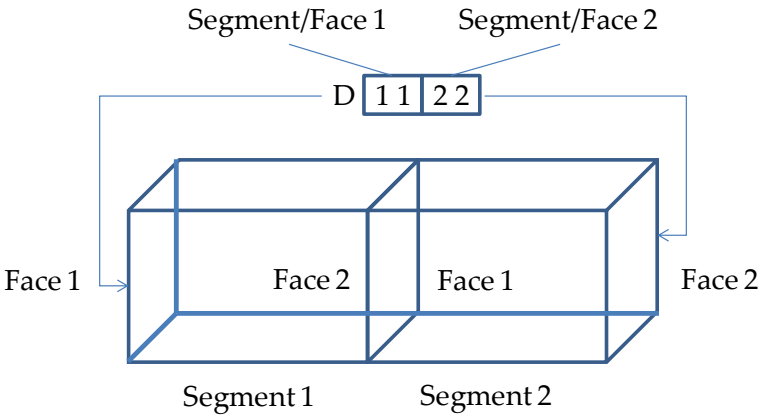


Figure 67: The "D" input file identifier.

In the example above, a two-segment 3D model is shown with the identifier referring to the faces at either end of the model for calculating the deflection ratio. The calculation of the deflection ratio is the dot product of the cross products of the two faces as shown in Figure 68.

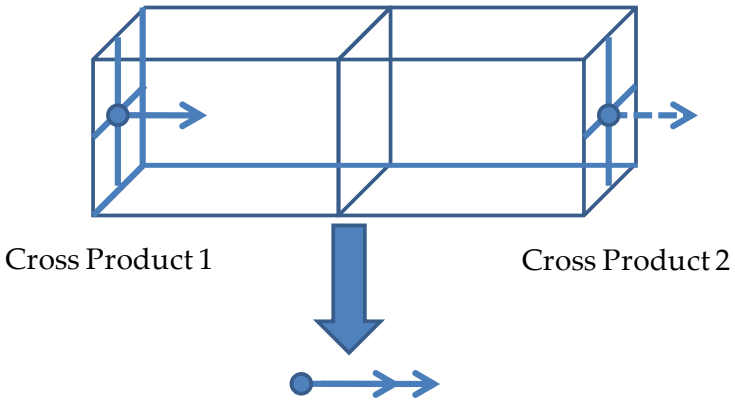


Figure 68: The deflection ratio calculation.

The dot product value ranges from "1", indicating the faces are parallel, to "-1" meaning the faces are inverted. However, this value is impossible for a dsDNA model to achieve at temperatures below the melting point and does not occur unless the model indicates a denatured structure. At temperatures below the melting point, the ratio should not significantly drop for multi-segment 3D models with total contour lengths below the persistence length.

"M" defines the mass identifier in which the mass value, in kilograms, the coordinates, in meters, and the flag for "pinning" the mass node in place are initialized. An example is given in Figure 69.

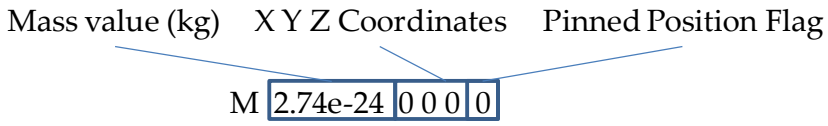


Figure 69: The "M" identifier in the input file.

The mass node in the example has a mass value of 2.74e-24 kg, is located at the (0, 0, 0) coordinate meters and is flagged as unpinned. This output format is the same for the snapshot output file required for graphical output of the simulation.

The spring identifier can be any of the spring types described in Appendix B. The DNA spring identifiers for the 3D spring model include the spring identifier numbers, mass node numbers, the rest length, the melting temperature, the number of base pairs, the extinction coefficients (ssDNA and dsDNA) and the spring constant scaling factor for initialization. The support springs do not have melting temperatures, base pairs, or extinction coefficients. Their identifier numbers correspond to the DNA springs they follow during simulation. The "S" spring identifier is used as an example in Figure 70 because it includes all possible inputs.

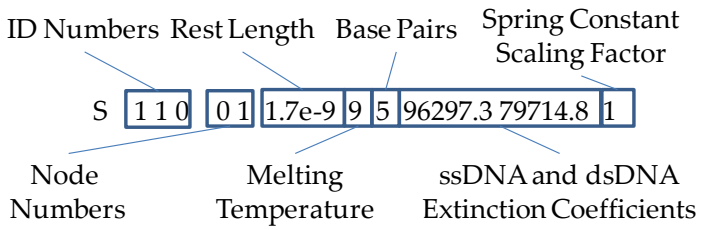


Figure 70: The spring identifier in the input file.

The example "S" spring identifier above indicates that it is a DNA spring. The number identifiers (1 1 0) locate it in the first segment, identifying it as the first of four "S" springs in that segment. The mass nodes that the spring connects are mass nodes 0 and 1. Its initial length is set at 1.7 nm with a melting temperature of 9°C. The double stranded and single stranded extinction coefficients are 79714.8 $M^{-1}cm^{-1}$ and 96297.3 $M^{-1}cm^{-1}$, respectively, and its scale factor for the spring constant is 1.

Once the simulator has initialized all values from the input file and the command prompt, it sums the forces acting on each mass node in the model, generated from the Brownian force and the spring restoring forces, and updates the mass nodes to a new position at the end of each time step. When the total time has reached the maximum limit, the temperature is incremented and the process repeats until the maximum temperature is reached. Data is recorded according to the parameters set by the user. The standard data output includes the time step or the temperature, the total absorbance of the model, the spring length, the deflection ratio of the faces of the model, and the spring constant. The process for generating trajectories is repeated until either the total

time has elapsed for a constant temperature run or the final temperature range has been reached.

5. Input Mesh File Generator Source Code

The main MATLAB program for generating the 3D DNA-spring models

```
% --------------------------------------
% 3D dsDNA Segment Mass Generator v 4.0
% Vincent Mao
% written: Mar 21, 2009
% last modified: Nov 23, 2009
% --------------------------------------
% This program generates the mass node file for the spring similator.  The
% generator was originally a template for the tile generation.  It is now
% a generator for new structures being tested.
% This is a 3D model representing the double helix DNA structure.
% Modifications to the code allow for models to represent melting
% transition and cross-linking modifications.

clear;
clc;
close all;

%-----------------------------------------------------------------
% Change these parameters
%-----------------------------------------------------------------
num_seg = 1;        % Segments to generate in the WLC
bp_per_block = 15;   % Number of bp represented by each block

% Scaling factors
DNA_scale = 1;
Stiff_scale = 1;
Brace_scale = 1;
Cross_scale = 1;

% Modification in the model
simple_gen = 0;     % Simple model  (2 masses, 1 spring)
ds_to_ss = 1;       % Melting model
xlink = 0;          % Cross link modification
%-----------------------------------------------------------------

if(simple_gen == 0)
    Mass_gen
    if(ds_to_ss == 0)
        Seg_gen
    else
        Seg_gen_melt
    end
else
    Mass_gen_simple
    Seg_gen_simple
end

mass_coords
segments
```

The mass node identifier generator source code for simple and 3D models:

```
% This file is called by the Block_gen.m program
% Simple
%-----------------------------------------------
% Number of bases in segment sequence
%-----------------------------------------------
bp = 0.34e-9;  % nm/bp
avg_mass_per_base = 0.66; % kg
Avogadro = 6.022e23;
total_bp = num_seg * bp_per_block;
total_mass = (total_bp * avg_mass_per_base) / Avogadro;
total_masses = num_seg + 1;
mass_value_per_node = total_mass / total_masses; % kg

% Counters
mass_counter    = 1;
base_position   = 0;

% Generate the array for the x coords
for i = 1:(num_seg + 1)
    bases_across(i) = base_position;
    base_position = base_position + bp_per_block;
end

% Scale the coords to the DNA lengths
coord_x =   bp * bases_across;
coord_y =   [0    2e-9    0    2e-9];
coord_z =   [0   0   -2e-9    -2e-9];

% Generate the mass coords
for i = 1:length(coord_x)
    m = ['M ' num2str(mass_value_per_node) ' ' num2str(coord_x(i)) ' '
num2str(coord_y(i)) ' ' num2str(coord_z(i)) ' ' num2str(0) ];
    mass_coords{mass_counter,1} = m;
    mass_counter = mass_counter + 1;
end
```

```
% This file is called by the Block_gen.m program
% 3D
%-----------------------------------------------
% Number of bases in segment sequence
%-----------------------------------------------
bp = 0.34e-9;  % nm/bp
avg_mass_per_base = 0.66; % kg
Avogadro = 6.022e23;
total_bp = num_seg * bp_per_block;
total_mass = (total_bp * avg_mass_per_base) / Avogadro;
total_mass_nodes = (num_seg + 1) * 4;
mass_value_per_node = total_mass / total_mass_nodes; % kg

% Counters
mass_counter    = 1;
base_position   = 0;

% Generate the array for the x coords
```

```
for i = 1:(num_seg + 1)
    bases_across(i) = base_position;
    base_position = base_position + bp_per_block;
end

% Scale the coords to the DNA lengths
coord_x =   bp * bases_across;
coord_y =   [0    2e-9    0    2e-9];
coord_z =   [0   0   -2e-9    -2e-9];

% Generate the mass coords
for i = 1:length(coord_x)
    for j = 1:4
        m = ['M ' num2str(mass_value_per_node, 15) ' ' num2str(coord_x(i)) ' '
num2str(coord_y(j)) ' ' num2str(coord_z(j)) ' ' num2str(0) ];
        mass_coords{mass_counter,1} = m;
        mass_counter = mass_counter + 1;
    end
end
```

The spring node identifier generator for simple and 3D models

```
% Spring segment numbering per block - simple
spring_seg = [1 2 3 4];

DNA_mass_node_1 = [0 1 2 3];
DNA_mass_node_2 = DNA_mass_node_1 + 1;

Stiff_spring_ref = [2 1 2 1 3 4 4 3];
Stiff_spring_sec = [1 2 3 4 5 6 7 8];
Stiff_mass_node_1 = [0 4 1 0 2 6 5 4];
Stiff_mass_node_2 = [1 5 3 2 3 7 7 6];

Brace_spring_ref = [2 1 2 4 3 4 1 3];
Brace_spring_sec = [2 1 7 3 6 5 8 4];
Brace_mass_node_1 = [0 1 1 3 2 3 0 2];
Brace_mass_node_2 = [5 4 7 5 7 6 6 4];

Cross_mass_node_1 = [0 1 4 5];
Cross_mass_node_2 = [3 2 7 6];

%------------------------------
% Melting Temps
%------------------------------
melt_across = [9];

%------------------------------
% Counters and offset values
%------------------------------
mass_counter     = 1;
spring_counter   = 1;

block_inc = 1;

bases = bases_across;
ext_coeff = bases * 9584 / 4;
ext_coeff2 = bases * 7972 / 4;
melt_temp = melt_across;
```

```
%------------------------------
% Spring Segment Generator
%------------------------------
for(i = 1:num_seg)
    % DNA springs
    s = ['S ' num2str(i) ' ' num2str(spring_seg(i)) ' ' num2str(0) ' '
num2str(DNA_mass_node_1(j) + block_inc * (i-1)) ' '  num2str(DNA_mass_node_2(j) +
block_inc * (i-1)) ' ' num2str(bp_per_block * bp) ' ' num2str(melt_temp) ' '
num2str(bases(2)) ' ' num2str(ext_coeff(2)) ' ' num2str(ext_coeff2(2)) ' '
num2str(DNA_scale)];
    segments{spring_counter,1} = s;
    spring_counter = spring_counter + 1;
end;



% Spring segment numbering per block - simple
spring_seg = [1 2 3 4];

DNA_mass_node_1 = [0 1 2 3];
DNA_mass_node_2 = DNA_mass_node_1 + 1;

Stiff_spring_ref = [2 1 2 1 3 4 4 3];
Stiff_spring_sec = [1 2 3 4 5 6 7 8];
Stiff_mass_node_1 = [0 4 1 0 2 6 5 4];
Stiff_mass_node_2 = [1 5 3 2 3 7 7 6];

Brace_spring_ref = [2 1 2 4 3 4 1 3];
Brace_spring_sec = [2 1 7 3 6 5 8 4];
Brace_mass_node_1 = [0 1 1 3 2 3 0 2];
Brace_mass_node_2 = [5 4 7 5 7 6 6 4];

Cross_mass_node_1 = [0 1 4 5];
Cross_mass_node_2 = [3 2 7 6];

%------------------------------
% Melting Temps
%------------------------------
melt_across = [9];

%------------------------------
% Counters and offset values
%------------------------------
mass_counter     = 1;
spring_counter   = 1;

block_inc = 1;

bases = bases_across;
ext_coeff = bases * 9584 / 4;
ext_coeff2 = bases * 7972 / 4;
melt_temp = melt_across;

%------------------------------
% Spring Segment Generator
%------------------------------
for(i = 1:num_seg)
    % DNA springs
    s = ['S ' num2str(i) ' ' num2str(spring_seg(i)) ' ' num2str(0) ' '
num2str(DNA_mass_node_1(j) + block_inc * (i-1)) ' '  num2str(DNA_mass_node_2(j) +
block_inc * (i-1)) ' ' num2str(bp_per_block * bp) ' ' num2str(melt_temp) ' '
```

```
num2str(bases(2)) ' ' num2str(ext_coeff(2)) ' ' num2str(ext_coeff2(2)) ' '
num2str(DNA_scale)];
    segments{spring_counter,1} = s;
    spring_counter = spring_counter + 1;
end;



% Spring segment numbering per block - 3D
spring_seg = [1 2 3 4];

DNA_mass_node_1 = [0 1 2 3];
DNA_mass_node_2 = [4 5 6 7];

Stiff_spring_ref = [2 1 2 1 3 4 4 3];
Stiff_spring_sec = [1 2 3 4 5 6 7 8];
Stiff_mass_node_1 = [0 4 1 0 2 6 5 4];
Stiff_mass_node_2 = [1 5 3 2 3 7 7 6];

Brace_spring_ref = [2 1 2 4 3 4 1 3];
Brace_spring_sec = [2 1 7 3 6 5 8 4];
Brace_mass_node_1 = [0 1 1 3 2 3 0 2];
Brace_mass_node_2 = [5 4 7 5 7 6 6 4];

Cross_mass_node_1 = [0 1 4 5];
Cross_mass_node_2 = [3 2 7 6];

%------------------------------
% Melting Temps
%------------------------------
melt_across = [70];

%------------------------------
% Counters and offset values
%------------------------------
mass_counter     = 1;
spring_counter   = 1;

block_inc = 4;

avg_mass_per_bp = 0.66;     % kg/mole
avg_mass_per_base = 0.33;   % kg/mole
dsDNA_EC = 24156;
ssDNA_EC = 29181;
bases = bases_across;
ext_coeff = avg_mass_per_bp * ssDNA_EC * bases;%bases * 9777; % kg * 1/(M * cm), assume 1
cm
ext_coeff2 = avg_mass_per_bp * dsDNA_EC * bases; %bases * 7984;
melt_temp = melt_across;

hypotenuse = sqrt((bp_per_block * bp) ^ 2 + 2e-9 ^ 2);

%------------------------------
% Spring Segment Generator
%------------------------------
for(i = 1:num_seg)
    % DNA springs
    for j = 1:4
        s = ['S ' num2str(i) ' ' num2str(spring_seg(j)) ' ' num2str(0) ' '
num2str(DNA_mass_node_1(j) + block_inc * (i-1)) ' '  num2str(DNA_mass_node_2(j) +
block_inc * (i-1)) ' ' num2str(bp_per_block * bp) ' ' num2str(melt_temp) ' '
```

```
num2str(bases(2)) ' ' num2str(ext_coeff(2)) ' ' num2str(ext_coeff2(2)) ' '
num2str(DNA_scale)];
        segments{spring_counter,1} = s;
        spring_counter = spring_counter + 1;
    end;

    % Stiff Springs
    for j = 1:8
        s = ['T ' num2str(i) ' ' num2str(Stiff_spring_ref(j)) ' '
num2str(Stiff_spring_sec(j)) ' ' num2str(Stiff_mass_node_1(j) + block_inc * (i-1)) ' '
num2str(Stiff_mass_node_2(j) + block_inc * (i-1)) ' ' num2str(2e-9) ' '
num2str(Stiff_scale)];
        segments{spring_counter,1} = s;
        spring_counter = spring_counter + 1;
    end;

    % Brace Springs
    for j = 1:8
        s = ['B ' num2str(i) ' ' num2str(Brace_spring_ref(j)) ' '
num2str(Brace_spring_sec(j)) ' ' num2str(Brace_mass_node_1(j) + block_inc * (i-1)) ' '
num2str(Brace_mass_node_2(j) + block_inc * (i-1)) ' ' num2str(hypotenuse, 15) ' '
num2str(Brace_scale)];
        segments{spring_counter,1} = s;
        spring_counter = spring_counter + 1;
    end;

    % Cross Springs
    for j = 1:4
        s = ['C ' num2str(i) ' ' num2str(i) ' ' num2str(0) ' '
num2str(Cross_mass_node_1(j) + block_inc * (i-1)) ' '  num2str(Cross_mass_node_2(j) +
block_inc * (i-1)) ' ' num2str(2e-9*sqrt(2), 15) ' ' num2str(Cross_scale)];
        segments{spring_counter,1} = s;
        spring_counter = spring_counter + 1;
    end;
end;
```

The following is the MATLAB source code for generating input files for DNA grid nanostructures. The source code for the main is provided, followed by functions that are called by the main file.

```
% Written by: Vincent Mao
% last modified: 2.14.09
% This program generates a meshfile that contains the positions of all the
% dsDNA segments in an A tile.  The lengths of the segments are dependent
% on the position of the tile in the 8 tile structure that can be
% generated.

% Tiles can only range from 1-8 to generate an 8T structure.
% Tile generation:
%   1 2
%   3 4
%   5 6
%   7 8
%--------------------------------------------------------------------------
```

```
% Clear all data before generation
clear;
clc;
close all;

%--------------------------------------------------------------------------
% Change these parameters
% These parameters should be the same as in Sticky_End_Generator.m
tiles = 8;
DNA_scale = [1 1 1 1 1 1 1 1];
Stiff_scale = 1;
theory = 4;
ds_to_ss = 1;   % new model
x_link = 1;     % cross-linked model
%--------------------------------------------------------------------------


%--------------------------------------------------------------------------
% Set global constants
bp              = 0.34e-9;  % nm/bp
avg_mass_per_bp = 0.66;     % kg/mole
avg_mass_per_base = 0.33;   % kg/mole
bp_per_tile     = 164;      % (23 + 18) * 2 arms * 2 sides (dsDNA only)
Avogadro        = 6.022e23;
ext_coeff_scaling = 1 / (6.022e26); % extinction coefficient per molecule
dsDNA_EC = 24156;
ssDNA_EC = 29181;
%--------------------------------------------------------------------------

%--------------------------------------------------------------------------
% Correct number of base pairs for sticky ends joining for multiple tiles
if(tiles <= 3)
    bp_per_tile = 164 + (5 * (tiles - 1));
end
if(tiles == 4 || tiles == 5)
    bp_per_tile = 164 + (5 * tiles);
end
if(tiles > 5 && tiles < 8)
    bp_per_tile = 164 + (5 * (tiles + 1));
end
if(tiles == 8)
    bp_per_tile = 164 + (5 * (tiles + 2));
end
%--------------------------------------------------------------------------

%--------------------------------------------------------------------------
% Calculate the mass for each mass node assuming even distribution of mass
total_mass = (bp_per_tile * avg_mass_per_bp) / Avogadro;

% For the normal model:
% Total Mass nodes :    16 segments * 8 mass nodes per segment
%
% For cross linked model:
% Total Mass nodes :    8 segments * 8 mass nodes per segment +
%                       1 segment * 12 mass nodes per segment +
%                       4 segments * 16 mass nodes per segment +
%                       3 segments * 24 mass nodes per segment +
%                       7 segments * 8 mass nodes per sticky +
%                       1 segment * 4 mass nodes per sticky
if(x_link == 1)
    total_mass_nodes = 212;
else
    total_mass_nodes = 128;
```

```
end
mass_value_per_node = total_mass / total_mass_nodes;
%--------------------------------------------------------------------------

% Position 1 starts at A+ (no vertical or horizontal flipping).
% vertical flipping flag = 0
% horizontal flipping flag = 0

position = 1;
strain = 2.11243934824e-009;
sStrain = 2e-9;
while(position <= tiles)
    if(mod(position, 4) == 1)
        vert_flip = 0;
        hori_flip = 0;
    end

    if(mod(position, 4) == 2)
        vert_flip = 1;
        hori_flip = 0;
    end

    if(mod(position, 4) == 3)
        vert_flip = 0;
        hori_flip = 1;
    end

    if(mod(position, 4) == 0)
        vert_flip = 1;
        hori_flip = 1;
    end

    if(x_link == 0)
        % Horizontal Arm Generation
        Horizontal_Mass_Gen
        Horizontal_Seg_Gen

        % Vertical Arm Generation
        Vertical_Mass_Gen
        Vertical_Seg_Gen

        if(ds_to_ss == 0)
            % Linking neighboring segments
            Horizontal_Thermo_Sep_mod
            Vertical_Thermo_Sep_mod
            % Crossover generators between the arms of the tile
            Xover_Gen
            % Core connection generator (assuming the core is a perfect square)
            Core_Gen
        else
            Xover_Gen_mod
            Core_Gen_mod
        end
    else
        % Cross link model generator calls here
        % Horizontal Arm Generation
        Horizontal_Mass_Gen_X
        if(mod(position, 4) == 1)
            Horizontal_Seg_Gen_X
        else
            Horizontal_Seg_Gen_X_V
        end
```

```
        % Vertical Arm Generation
        Vertical_Mass_Gen_X
        if(mod(position, 4) == 1)
            Vertical_Seg_Gen_X
        else
            Vertical_Seg_Gen_X_V
        end

        Xover_Gen_mod_X
        Core_Gen_mod_X
    end
    % Go to the next position to generate the next tile
    position = position+1;
end
```

Horizontal arm mass node generator source code

```
% Written by: Vincent Mao
% last modified: 8.18.09
%
% This program generates the horizontal mass nodes for each tile arm.  The
% sequences are generated from left to right according to the A Tile
% layout assuming a square core with the same number of base pairs for
% both the upper and lower dsDNA helices that compose each of the arms. The
% meshfile format is:
%       (M) <Node mass> <X coord> <Y coord> <Z coord> <Fixed mass flag>

% Patterns the base lengths for each segment in the arm
% Left Arm

if(hori_flip == 0)
    bases_across1 = [0 8 9 19];
    initial_y = 0;
end

if(hori_flip == 1)
    bases_across1 = [0 13 14 24];
    % initial_y = 1 bp + 24 bp + 5 bp + 19 bp + coregap - 1 bp
    initial_y = -2.134e-8;
end

initial_x = -bases_across1(4) * bp;

% Right Arm
if(hori_flip == 0)
    bases_across2 = [0 10 11 24] + bases_across1(4);
end

if(hori_flip == 1)
    bases_across2 = [0 10 11 19] + bases_across1(4);
end

% Initialize coordinates
coord_y =   [0    2e-9    0    2e-9];
coord_z =   [0   0   -2e-9    -2e-9];

% Position counters for the meshfile
masscount = 1;
```

```
coregap = 5.02e-9;

%------------------------------
% Mass Node Generator
%------------------------------
coord2 = coord_y;
coord3 = coord_z;

if(position > 4 && position <= 8)
    initial_y = initial_y + -2.134e-8 * 2;
end

% Generate the left and right arms      <======>      <======>
%                                       <======>      <======>
for(iter3 = 1:2)
    % Generate the whole arm on one side    <======>
    %                                       <======>
    for(iter2 = 1:2)
        if(iter2 == 1 && iter3 == 1)
            coord1 = bp*bases_across1;
        end
        if(iter2 == 2 && iter3 == 1)
            coord1 = bp*bases_across1;
        end
        if(iter2 == 1 && iter3 == 2)
            coord1 = bp*bases_across2;
        end
        if(iter2 == 2 && iter3 == 2)
            coord1 = bp*bases_across2;
        end

        if(mod(position, 2) == 0)
            coord1 = coord1 + bp * bases_across2(4) + 6.72e-9;
        end

        % Generate one pair of horizontal segments <======>
        for i = 1:4
            % Generate the face nodes on the segment < (2 X 2 nm)
            for j = 1:4
                m = ['M ' num2str(mass_value_per_node) ' ' num2str(initial_x + coord1(i)
+ coregap * (iter3-1)) ' ' num2str(initial_y + coord2(j)+ (2.34e-9 *(iter2-1))) ' '
num2str(coord3(j)) ' ' num2str(0) ];
                Hori_mass_n{masscount,1} = m;
                masscount = masscount + 1;
            end
        end
    end
end
Hori_mass_n
```

Horizontal mass node generator source code for cross-linked model:

```
% Written by: Vincent Mao
% last modified: 12.13.09
%
% This program generates the horizontal mass nodes for each tile arm based
% on the cross linked sites that are available on the A tile.  The
% sequences are generated from left to right according to the A Tile
% layout assuming a square core with the same number of base pairs for
```

```
% both the upper and lower dsDNA helices that compose each of the arms. The
% meshfile format is:
%       (M) <Node mass> <X coord> <Y coord> <Z coord> <Fixed mass flag>

% Patterns the base lengths for each segment in the arm
% The mass numbering system is in a counter clockwise system starting with
% the back face as shown in the following diagram:
%
%         1 o-----------o 5      9 o------------o 13
%     3 o-----------o 7 |   11 o-----------o 15 |
%       |   |       |   |      |   |       |    |
%       | 0 o-------|---o 4    | 8 o-------|----o 12
%     2 o-----------o 6     10 o-----------o 14
%       Thermo region 1       Thermo region 2
%              Lower left arm node map
%
% The thermo region map for the horizontal mass generation goes like this:
%
%   3   4  | Core |   7   8
%   1   2  | Core |   5   6
%--------------------------------------------------------------------------
% hori_flip flag: flip the horizontal arms
% 1x1 = lower left regions (1 and 2)
% 1x2 = upper left regions (3 and 4)
% 2x1 = lower right regions (5 and 6)
% 2x2 = upper right regions (7 and 8)
%--------------------------------------------------------------------------

coregap = 5.02e-9;
% Calculated y offset for multiple tile generation:
%  1 bp + 24 bp + 5 bp + 19 bp + coregap - 1 bp
y_tile_offset = bp + 24 * bp + 5 * bp + 19 * bp + coregap - bp;

%-----------
% Left Arm
%-----------
if(vert_flip == 0 && hori_flip == 0)
    bases_across1x1 = [0 8 9 19];           % [8    10]
    bases_across1x2 = [0 1 2 8 9 12 13 19]; % [1 1 6    3 1 6]
    initial_x = -bases_across1x1(4) * bp;
    initial_y = 0;
end

if(vert_flip == 1 && hori_flip == 0)
    bases_across1x1 = [0 1 2 8 9 12 13 19]; % [1 1 6    3 1 6]
    bases_across1x2 = [0 8 9 19];           % [8    10]
    initial_x = -bases_across1x1(8) * bp;
    initial_y = 0;
end

if(vert_flip == 0 && hori_flip == 1)
    bases_across1x1 = [0 13 14 18 19 22 23 24]; % [13  4 1 3 1 1]
    bases_across1x2 = [0 5 6 10 11 13 14 24];   % [5 1 4 1 2    10]
    initial_x = -bases_across1x1(8) * bp;
    initial_y = -y_tile_offset;
end

if(vert_flip == 1 && hori_flip == 1)
    bases_across1x1 = [0 5 6 10 11 13 14 24];   % [5 1 4 1 2    10]
    bases_across1x2 = [0 13 14 18 19 22 23 24]; % [13   4 1 3 1 1]
    initial_x = -bases_across1x1(8) * bp;
    initial_y = -y_tile_offset;
end
```

```
%-----------
% Right Arm
%-----------
if(vert_flip == 0 && hori_flip == 0)
    bases_across2x1 = [0 2 3 6 7 10 11 24] + bases_across1x1(4);    % [2 1 3 1 3   13]
    bases_across2x2 = [0 10 11 14 15 19 20 24] + bases_across1x1(4);% [10   3 1 4 1 4]
end

if(vert_flip == 1 && hori_flip == 0)
    bases_across2x1 = [0 10 11 14 15 19 20 24] + bases_across1x1(8);% [10   3 1 4 1 4]
    bases_across2x2 = [0 2 3 6 7 10 11 24] + bases_across1x1(8);    % [2 1 3 1 3    13]
end

if(vert_flip == 0 && hori_flip == 1)
    bases_across2x1 = [0 10 11 19] + bases_across1x1(8);            % [10   8]
    bases_across2x2 = [0 7 8 10 11 17 18 19] + bases_across1x1(8);  % [7 1 2    6 1 1]
end

if(vert_flip == 1 && hori_flip == 1)
    bases_across2x1 = [0 7 8 10 11 17 18 19] + bases_across1x1(8);  % [7 1 2    6 1 1]
    bases_across2x2 = [0 10 11 19] + bases_across1x1(8);            % [10   8]
end

% Initialize coordinates
coord2 =   [0    2e-9    0    2e-9];
coord3 =   [0   0   -2e-9    -2e-9];

% Position counter for the meshfile
masscount = 1;

%------------------------------
% Mass Node Generator
%------------------------------
if(position > 4 && position <= 8)
    initial_y = initial_y + -y_tile_offset * 2;
end

%-------------------------------------------------------------------------
% Generate the left and right arms      <===><===>      <===><===>
%                                       <===><===>      <===><===>
%                                          L               R
% arm_gen is the left/right arm identifier: 1 is left, 2 is right
%-------------------------------------------------------------------------
for(arm_gen = 1:2)
    %---------------------------------------------------------------------
    % Generate the whole arm on one side  2:  <===><===>
    %                                     1:  <===><===>
    % reg_gen is the vertical region identifier: 1 is bottom, 2 is top
    %---------------------------------------------------------------------
    for(reg_gen = 1:2)
        if(reg_gen == 1 && arm_gen == 1)
            coord1 = bp * bases_across1x1;
        end
        if(reg_gen == 2 && arm_gen == 1)
            coord1 = bp * bases_across1x2;
        end
        if(reg_gen == 1 && arm_gen == 2)
            coord1 = bp * bases_across2x1;
        end
        if(reg_gen == 2 && arm_gen == 2)
            coord1 = bp * bases_across2x2;
        end
```

```
        if(mod(position, 2) == 0)
            coord1 = coord1 + bp * bases_across2x1(8) + 6.72e-9;
        end

        % Generate one pair of horizontal segments <===><===>
        for i = 1:length(coord1)
            % Generate the face nodes on the segment < (2 X 2 nm)
            for j = 1:4
                % Calculated xyz coords
                x_coord = initial_x + coord1(i) + coregap * (arm_gen - 1);
                y_coord = initial_y + coord2(j) + (2.34e-9 * (reg_gen - 1));
                z_coord = coord3(j);

                % Generate the meshfile information
                m = ['M ' num2str(mass_value_per_node) ' ' num2str(x_coord) ' '
num2str(y_coord) ' ' num2str(z_coord) ' ' num2str(0) ];
                Hori_mass_n{masscount,1} = m;
                masscount = masscount + 1;
            end
        end
    end
end
Hori_mass_n
```

Horizontal spring segment generator source code

```
% Calculate the length of each segment
length1 = (bases_across1(2) - bases_across1(1)) * bp;
length2 = (bases_across1(4) - bases_across1(3)) * bp;
length3 = (bases_across2(2) - bases_across2(1)) * bp;
length4 = (bases_across2(4) - bases_across2(3)) * bp;

bases_across1 = bases_across1 + bases_across1(4);
bases_across2 = bases_across2 + bases_across2(4);

% Initialize the arrays
length_across = [length1 length2 length1 length2 length3 length4 length3 length4];
bases = length_across/bp;
ext = avg_mass_per_bp * ssDNA_EC * bases;% bases * 9777;
ext2 = avg_mass_per_bp * dsDNA_EC * bases; %bases * 7984;

% Reference numbers
springseg = [1 2 3 4];

rodL = [2 1 2 1];
rodR = [3 4 4 3];
stiffL = [1 2 3 4];
stiffR = [5 6 7 8];

braceL = [2 1 2 4];
braceR = [3 4 1 3];
braceLS = [2 1 7 3];
braceRS = [6 5 8 4];

% DNA Spring connections
armSM1 = [0 1 2 3];
armSM2 = [4 5 6 7];
```

```
% Stiff Spring connections
armLM1 = [0 4 1 0];
armLM2 = [1 5 3 2];
armRM1 = [2 6 5 4];
armRM2 = [3 7 7 6];

% Side cross braces
armCM1 = [0 1 4 5];
armCM2 = [3 2 7 6];
armCM3 = [0 1 1 3];
armCM4 = [5 4 7 5];
armCM5 = [2 3 0 2];
armCM6 = [7 6 6 4];

%------------------------------
% Melting Temps and Cross Links
%------------------------------
% Four sets of Tm Theories:
%   1. 16 Thermo regions
%   2. Crossover theory links thermo regions
%   3. Lowest melting temperature on the shell
%   4. Average core melting temperatures with outer thermo regions
if(mod(position, 4) == 1)
    if(theory == 1)
        melt = [30 48 35 51 49 65 45 55];
    end
    if(theory == 2)
        melt = [39 60.2 43 60.2 58 55 58 50];
    end
    if(theory == 3)
        melt = [30 30 35 35 42 42 31 31];
    end
    if(theory == 4)
        melt = [45.1 60.2 47.6 60.2 58 61.5 58 56.5];
    end

%     melt = [32.5 60.2 32.5 60.2 58 64.4 58 64.4];
%     melt = [52.2 60.2 52.2 60.2 58 64.4 58 64.4];
end

if(mod(position, 4) == 2)
    if(theory == 1)
        melt = [35 51 30 48 45 55 49 65];
    end
    if(theory == 2)
        melt = [43 60.2 39 60.2 58 50 58 55];
    end
    if(theory == 3)
        melt = [35 35 30 30 31 31 42 42];
    end
    if(theory == 4)
        melt = [47.6 60.2 45.1 60.2 58 56.5 58 61.5];
    end
%     melt = [52.2 60.2 52.2 60.2 58 60 58 60];
%     melt = [52.2 60.2 52.2 60.2 58 64.4 58 64.4];
end

if(mod(position, 4) == 3)
    if(theory == 1)
        melt = [65 49 55 45 48 30 51 35];
    end
    if(theory == 2)
        melt = [55 58 50 58 60.2 39 60.2 43];
```

```
    end
    if(theory == 3)
        melt = [31 31 42 42 35 35 30 30];
    end
    if(theory == 4)
        melt = [61.5 58 56.5 58 60.2 45.1 60.2 47.6];
    end
%     melt = [65 49 55 45 48 30 51 35];
%     melt = [60 58 60 58 60.2 52.2 60.2 52.2];
%     melt = [64.4 58 64.4 58 60.2 52.2 60.2 52.2];

end

if(mod(position, 4) == 0)
    if(theory == 1)
        melt = [55 45 65 49 51 35 48 30];
    end
    if(theory == 2)
        melt = [50 58 55 58 60.2 43 60.2 39];
    end
    if(theory == 3)
        melt = [42 42 31 31 30 30 35 35];
    end
    if(theory == 4)
        melt = [56.5 58 61.5 58 60.3 47.6 60.2 45.1];
    end
%     melt = [50 58 55 58 60.2 43 60.2 39];
%     melt = [55 45 65 49 51 35 48 30];
%     melt = [64.4 58 64.4 58 60.3 32.5 60.2 32.5];
%     melt = [64.4 58 64.4 58 60.3 52.2 60.2 52.2];
end

%------------------------------
% Counters and offset values
%------------------------------
springcount = 1;

sticky_inc = 4;
mass_seg_inc = 8;
arm_node_offset = 32;
arm_segment_offset = 16;
offset = 64;
tile_offset = 128;

%------------------------------
% Spring Segment Generator
%------------------------------
seg = 4;
count = 2;
counter = 8;

% Real spring links
for(iter2 = 1:count)
    for(iter = 1:seg)
        for g = 1:seg
            s = ['S ' num2str( ((position-1) * arm_segment_offset ) + iter + (seg*
(iter2-1))) ' ' num2str(springseg(g)) ' ' num2str(0) ' ' num2str(armSM1(g)+
mass_seg_inc*(iter-1) + arm_node_offset*(iter2-1) + ((position-1) * tile_offset )) ' '
num2str(armSM2(g)+ mass_seg_inc*(iter-1)+ arm_node_offset*(iter2-1) + ((position-1) *
tile_offset )) ' ' num2str(length_across(iter + (seg* (iter2-1)))) ' ' num2str(melt(iter
+ (seg* (iter2-1)))) ' ' num2str(bases(iter + (seg* (iter2-1)))) ' ' num2str(ext(iter +
(seg* (iter2-1)))) ' ' num2str(ext2(iter + (seg* (iter2-1)))) ' ' num2str(DNA_scale(iter
+ (seg* (iter2-1)))) ];
```

```
            Hori_seg{springcount,1} = s;
            springcount = springcount + 1;
        end
    end
end

for(iter = 1:counter)
    % Stiff springs
    for g = 1:length(armLM1)
        if(ds_to_ss == 0)
            s = ['T ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(rodL(g)) ' ' num2str(stiffL(g)) ' ' num2str(armLM1(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armLM2(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
        end
        if(ds_to_ss == 1)
            if(g > 2)
                s = ['T ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(rodL(g)) ' ' num2str(stiffL(g)) ' ' num2str(armLM1(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armLM2(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
            else
                s = ['C ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(rodL(g)) ' ' num2str(stiffL(g)) ' ' num2str(armLM1(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armLM2(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
            end
        end
        Hori_seg{springcount,1} = s;
        springcount = springcount + 1;
    end;
    for g = 1:length(armRM1)
        if(ds_to_ss == 0)
            s = ['T ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(rodR(g)) ' ' num2str(stiffR(g)) ' ' num2str(armRM1(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armRM2(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
        end
        if(ds_to_ss == 1)
            if(g > 2)
                s = ['T ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(rodR(g)) ' ' num2str(stiffR(g)) ' ' num2str(armRM1(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armRM2(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
            else
                s = ['C ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(rodR(g)) ' ' num2str(stiffR(g)) ' ' num2str(armRM1(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armRM2(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
            end
        end
        Hori_seg{springcount,1} = s;
        springcount = springcount + 1;
    end;
    % Stiff springs on the face of the cube
    for g = 1:length(armCM1)
        s = ['C ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(((position-1) * arm_segment_offset ) + iter) ' ' num2str(0) ' '
num2str(armCM1(g)+ mass_seg_inc*(iter-1) + ((position-1) * tile_offset )) ' '
num2str(armCM2(g)+ mass_seg_inc*(iter-1) + ((position-1) * tile_offset )) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
        Hori_seg{springcount,1} = s;
        springcount = springcount+1;
```

```
    end;
    % Braces
    for g = 1:length(armCM1)
        s = ['B ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(braceL(g)) ' ' num2str(braceLS(g)) ' ' num2str(armCM3(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armCM4(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(sqrt(2e-9^2 + length_across(iter)^2)) ' '
num2str(1)];
        Hori_seg{springcount,1} = s;
        springcount = springcount+1;
    end;
    for g = 1:length(armCM1)
        s = ['B ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(braceR(g)) ' ' num2str(braceRS(g)) ' ' num2str(armCM5(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armCM6(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(sqrt(2e-9^2 + length_across(iter)^2)) ' '
num2str(1)];
        Hori_seg{springcount,1} = s;
        springcount = springcount+1;
    end;
end
Hori_seg
```

Horizontal spring segment generator source code for cross-linked model

```
% Calculate the length of each segment
length1 = (bases_across1(2) - bases_across1(1)) * bp;
length2 = (bases_across1(4) - bases_across1(3)) * bp;
length3 = (bases_across2(2) - bases_across2(1)) * bp;
length4 = (bases_across2(4) - bases_across2(3)) * bp;

bases_across1 = bases_across1 + bases_across1(4);
bases_across2 = bases_across2 + bases_across2(4);

% Initialize the arrays
length_across = [length1 length2 length1 length2 length3 length4 length3 length4];
bases = length_across/bp;
ext = avg_mass_per_bp * ssDNA_EC * bases;% bases * 9777;
ext2 = avg_mass_per_bp * dsDNA_EC * bases; %bases * 7984;

% Reference numbers
springseg = [1 2 3 4];

rodL = [2 1 2 1];
rodR = [3 4 4 3];
stiffL = [1 2 3 4];
stiffR = [5 6 7 8];

braceL = [2 1 2 4];
braceR = [3 4 1 3];
braceLS = [2 1 7 3];
braceRS = [6 5 8 4];

% DNA Spring connections
armSM1 = [0 1 2 3];
armSM2 = [4 5 6 7];

% Stiff Spring connections
armLM1 = [0 4 1 0];
```

```
armLM2 = [1 5 3 2];
armRM1 = [2 6 5 4];
armRM2 = [3 7 7 6];

% Side cross braces
armCM1 = [0 1 4 5];
armCM2 = [3 2 7 6];
armCM3 = [0 1 1 3];
armCM4 = [5 4 7 5];
armCM5 = [2 3 0 2];
armCM6 = [7 6 6 4];

%------------------------------
% Melting Temps and Cross Links
%------------------------------
% Four sets of Tm Theories:
%   1. 16 Thermo regions
%   2. Crossover theory links thermo regions
%   3. Lowest melting temperature on the shell
%   4. Average core melting temperatures with outer thermo regions
if(mod(position, 4) == 1)
    if(theory == 1)
        melt = [30 48 35 51 49 65 45 55];
    end
    if(theory == 2)
        melt = [39 60.2 43 60.2 58 55 58 50];
    end
    if(theory == 3)
        melt = [30 30 35 35 42 42 31 31];
    end
    if(theory == 4)
        melt = [45.1 60.2 47.6 60.2 58 61.5 58 56.5];
    end

%     melt = [32.5 60.2 32.5 60.2 58 64.4 58 64.4];
%     melt = [52.2 60.2 52.2 60.2 58 64.4 58 64.4];
end

if(mod(position, 4) == 2)
    if(theory == 1)
        melt = [35 51 30 48 45 55 49 65];
    end
    if(theory == 2)
        melt = [43 60.2 39 60.2 58 50 58 55];
    end
    if(theory == 3)
        melt = [35 35 30 30 31 31 42 42];
    end
    if(theory == 4)
        melt = [47.6 60.2 45.1 60.2 58 56.5 58 61.5];
    end
%     melt = [52.2 60.2 52.2 60.2 58 60 58 60];
%     melt = [52.2 60.2 52.2 60.2 58 64.4 58 64.4];
end

if(mod(position, 4) == 3)
    if(theory == 1)
        melt = [65 49 55 45 48 30 51 35];
    end
    if(theory == 2)
        melt = [55 58 50 58 60.2 39 60.2 43];
    end
    if(theory == 3)
```

```
        melt = [31 31 42 42 35 35 30 30];
    end
    if(theory == 4)
        melt = [61.5 58 56.5 58 60.2 45.1 60.2 47.6];
    end
%     melt = [65 49 55 45 48 30 51 35];
%     melt = [60 58 60 58 60.2 52.2 60.2 52.2];
%     melt = [64.4 58 64.4 58 60.2 52.2 60.2 52.2];

end

if(mod(position, 4) == 0)
    if(theory == 1)
        melt = [55 45 65 49 51 35 48 30];
    end
    if(theory == 2)
        melt = [50 58 55 58 60.2 43 60.2 39];
    end
    if(theory == 3)
        melt = [42 42 31 31 30 30 35 35];
    end
    if(theory == 4)
        melt = [56.5 58 61.5 58 60.3 47.6 60.2 45.1];
    end
%     melt = [50 58 55 58 60.2 43 60.2 39];
%     melt = [55 45 65 49 51 35 48 30];
%     melt = [64.4 58 64.4 58 60.3 32.5 60.2 32.5];
%     melt = [64.4 58 64.4 58 60.3 52.2 60.2 52.2];
end

%------------------------------
% Counters and offset values
%------------------------------
springcount = 1;

sticky_inc = 4;
mass_seg_inc = 8;
arm_node_offset = 32;
arm_segment_offset = 16;
offset = 64;
tile_offset = 128;

%------------------------------
% Spring Segment Generator
%------------------------------
seg = 4;
count = 2;
counter = 8;

% Real spring links
for(iter2 = 1:count)
    for(iter = 1:seg)
        for g = 1:seg
            s = ['S ' num2str( ((position-1) * arm_segment_offset ) + iter + (seg*
(iter2-1))) ' ' num2str(springseg(g)) ' ' num2str(0) ' ' num2str(armSM1(g)+
mass_seg_inc*(iter-1) + arm_node_offset*(iter2-1) + ((position-1) * tile_offset )) ' '
num2str(armSM2(g)+ mass_seg_inc*(iter-1)+ arm_node_offset*(iter2-1) + ((position-1) *
tile_offset )) ' ' num2str(length_across(iter + (seg* (iter2-1)))) ' ' num2str(melt(iter
+ (seg* (iter2-1)))) ' ' num2str(bases(iter + (seg* (iter2-1)))) ' ' num2str(ext(iter +
(seg* (iter2-1)))) ' ' num2str(ext2(iter + (seg* (iter2-1)))) ' ' num2str(DNA_scale(iter
+ (seg* (iter2-1)))) ];
            Hori_seg{springcount,1} = s;
            springcount = springcount + 1;
```

```
        end
    end
end

for(iter = 1:counter)
    % Stiff springs
    for g = 1:length(armLM1)
        if(ds_to_ss == 0)
            s = ['T ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(rodL(g)) ' ' num2str(stiffL(g)) ' ' num2str(armLM1(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armLM2(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
        end
        if(ds_to_ss == 1)
            if(g > 2)
                s = ['T ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(rodL(g)) ' ' num2str(stiffL(g)) ' ' num2str(armLM1(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armLM2(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
            else
                s = ['C ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(rodL(g)) ' ' num2str(stiffL(g)) ' ' num2str(armLM1(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armLM2(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
            end
        end
        Hori_seg{springcount,1} = s;
        springcount = springcount + 1;
    end;
    for g = 1:length(armRM1)
        if(ds_to_ss == 0)
            s = ['T ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(rodR(g)) ' ' num2str(stiffR(g)) ' ' num2str(armRM1(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armRM2(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
        end
        if(ds_to_ss == 1)
            if(g > 2)
                s = ['T ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(rodR(g)) ' ' num2str(stiffR(g)) ' ' num2str(armRM1(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armRM2(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
            else
                s = ['C ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(rodR(g)) ' ' num2str(stiffR(g)) ' ' num2str(armRM1(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armRM2(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
            end
        end
        Hori_seg{springcount,1} = s;
        springcount = springcount + 1;
    end;
    % Stiff springs on the face of the cube
    for g = 1:length(armCM1)
        s = ['C ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(((position-1) * arm_segment_offset ) + iter) ' ' num2str(0) ' '
num2str(armCM1(g)+ mass_seg_inc*(iter-1) + ((position-1) * tile_offset )) ' '
num2str(armCM2(g)+ mass_seg_inc*(iter-1) + ((position-1) * tile_offset )) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
        Hori_seg{springcount,1} = s;
        springcount = springcount+1;
    end;
    % Braces
```

```
    for g = 1:length(armCM1)
        s = ['B ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(braceL(g)) ' ' num2str(braceLS(g)) ' ' num2str(armCM3(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armCM4(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(sqrt(2e-9^2 + length_across(iter)^2)) ' '
num2str(1)];
        Hori_seg{springcount,1} = s;
        springcount = springcount+1;
    end;
    for g = 1:length(armCM1)
        s = ['B ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(braceR(g)) ' ' num2str(braceRS(g)) ' ' num2str(armCM5(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armCM6(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(sqrt(2e-9^2 + length_across(iter)^2)) ' '
num2str(1)];
        Hori_seg{springcount,1} = s;
        springcount = springcount+1;
    end;
end
Hori_seg


% Calculate the length of each segment
length1 = (bases_across1(2) - bases_across1(1)) * bp;
length2 = (bases_across1(4) - bases_across1(3)) * bp;
length3 = (bases_across2(2) - bases_across2(1)) * bp;
length4 = (bases_across2(4) - bases_across2(3)) * bp;

bases_across1 = bases_across1 + bases_across1(4);
bases_across2 = bases_across2 + bases_across2(4);

% Initialize the arrays
length_across = [length1 length2 length1 length2 length3 length4 length3 length4];
bases = length_across/bp;
ext = avg_mass_per_bp * ssDNA_EC * bases;% bases * 9777;
ext2 = avg_mass_per_bp * dsDNA_EC * bases; %bases * 7984;

% Reference numbers
springseg = [1 2 3 4];

rodL = [2 1 2 1];
rodR = [3 4 4 3];
stiffL = [1 2 3 4];
stiffR = [5 6 7 8];

braceL = [2 1 2 4];
braceR = [3 4 1 3];
braceLS = [2 1 7 3];
braceRS = [6 5 8 4];

% DNA Spring connections
armSM1 = [0 1 2 3];
armSM2 = [4 5 6 7];

% Stiff Spring connections
armLM1 = [0 4 1 0];
armLM2 = [1 5 3 2];
armRM1 = [2 6 5 4];
armRM2 = [3 7 7 6];

% Side cross braces
armCM1 = [0 1 4 5];
```

```
armCM2 = [3 2 7 6];
armCM3 = [0 1 1 3];
armCM4 = [5 4 7 5];
armCM5 = [2 3 0 2];
armCM6 = [7 6 6 4];

%------------------------------
% Melting Temps and Cross Links
%------------------------------
% Four sets of Tm Theories:
%   1. 16 Thermo regions
%   2. Crossover theory links thermo regions
%   3. Lowest melting temperature on the shell
%   4. Average core melting temperatures with outer thermo regions
if(mod(position, 4) == 1)
    if(theory == 1)
        melt = [30 48 35 51 49 65 45 55];
    end
    if(theory == 2)
        melt = [39 60.2 43 60.2 58 55 58 50];
    end
    if(theory == 3)
        melt = [30 30 35 35 42 42 31 31];
    end
    if(theory == 4)
        melt = [45.1 60.2 47.6 60.2 58 61.5 58 56.5];
    end

%     melt = [32.5 60.2 32.5 60.2 58 64.4 58 64.4];
%     melt = [52.2 60.2 52.2 60.2 58 64.4 58 64.4];
end

if(mod(position, 4) == 2)
    if(theory == 1)
        melt = [35 51 30 48 45 55 49 65];
    end
    if(theory == 2)
        melt = [43 60.2 39 60.2 58 50 58 55];
    end
    if(theory == 3)
        melt = [35 35 30 30 31 31 42 42];
    end
    if(theory == 4)
        melt = [47.6 60.2 45.1 60.2 58 56.5 58 61.5];
    end
%     melt = [52.2 60.2 52.2 60.2 58 60 58 60];
%     melt = [52.2 60.2 52.2 60.2 58 64.4 58 64.4];
end

if(mod(position, 4) == 3)
    if(theory == 1)
        melt = [65 49 55 45 48 30 51 35];
    end
    if(theory == 2)
        melt = [55 58 50 58 60.2 39 60.2 43];
    end
    if(theory == 3)
        melt = [31 31 42 42 35 35 30 30];
    end
    if(theory == 4)
        melt = [61.5 58 56.5 58 60.2 45.1 60.2 47.6];
    end
%     melt = [65 49 55 45 48 30 51 35];
```

```
%     melt = [60 58 60 58 60.2 52.2 60.2 52.2];
%     melt = [64.4 58 64.4 58 60.2 52.2 60.2 52.2];

end

if(mod(position, 4) == 0)
    if(theory == 1)
        melt = [55 45 65 49 51 35 48 30];
    end
    if(theory == 2)
        melt = [50 58 55 58 60.2 43 60.2 39];
    end
    if(theory == 3)
        melt = [42 42 31 31 30 30 35 35];
    end
    if(theory == 4)
        melt = [56.5 58 61.5 58 60.3 47.6 60.2 45.1];
    end
%     melt = [50 58 55 58 60.2 43 60.2 39];
%     melt = [55 45 65 49 51 35 48 30];
%     melt = [64.4 58 64.4 58 60.3 32.5 60.2 32.5];
%     melt = [64.4 58 64.4 58 60.3 52.2 60.2 52.2];
end

%------------------------------
% Counters and offset values
%------------------------------
springcount = 1;

sticky_inc = 4;
mass_seg_inc = 8;
arm_node_offset = 32;
arm_segment_offset = 16;
offset = 64;
tile_offset = 128;

%------------------------------
% Spring Segment Generator
%------------------------------
seg = 4;
count = 2;
counter = 8;

% Real spring links
for(iter2 = 1:count)
    for(iter = 1:seg)
        for g = 1:seg
            s = ['S ' num2str( ((position-1) * arm_segment_offset ) + iter + (seg*
(iter2-1))) ' ' num2str(springseg(g)) ' ' num2str(0) ' ' num2str(armSM1(g)+
mass_seg_inc*(iter-1) + arm_node_offset*(iter2-1) + ((position-1) * tile_offset )) ' '
num2str(armSM2(g)+ mass_seg_inc*(iter-1)+ arm_node_offset*(iter2-1) + ((position-1) *
tile_offset )) ' ' num2str(length_across(iter + (seg* (iter2-1)))) ' ' num2str(melt(iter
+ (seg* (iter2-1)))) ' ' num2str(bases(iter + (seg* (iter2-1)))) ' ' num2str(ext(iter +
(seg* (iter2-1)))) ' ' num2str(ext2(iter + (seg* (iter2-1)))) ' ' num2str(DNA_scale(iter
+ (seg* (iter2-1)))) ];
            Hori_seg{springcount,1} = s;
            springcount = springcount + 1;
        end
    end
end

for(iter = 1:counter)
    % Stiff springs
```

```
    for g = 1:length(armLM1)
        if(ds_to_ss == 0)
            s = ['T ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(rodL(g)) ' ' num2str(stiffL(g)) ' ' num2str(armLM1(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armLM2(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
        end
        if(ds_to_ss == 1)
            if(g > 2)
                s = ['T ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(rodL(g)) ' ' num2str(stiffL(g)) ' ' num2str(armLM1(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armLM2(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
            else
                s = ['C ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(rodL(g)) ' ' num2str(stiffL(g)) ' ' num2str(armLM1(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armLM2(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
            end
        end
        Hori_seg{springcount,1} = s;
        springcount = springcount + 1;
    end;
    for g = 1:length(armRM1)
        if(ds_to_ss == 0)
            s = ['T ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(rodR(g)) ' ' num2str(stiffR(g)) ' ' num2str(armRM1(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armRM2(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
        end
        if(ds_to_ss == 1)
            if(g > 2)
                s = ['T ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(rodR(g)) ' ' num2str(stiffR(g)) ' ' num2str(armRM1(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armRM2(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
            else
                s = ['C ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(rodR(g)) ' ' num2str(stiffR(g)) ' ' num2str(armRM1(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armRM2(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
            end
        end
        Hori_seg{springcount,1} = s;
        springcount = springcount + 1;
    end;
    % Stiff springs on the face of the cube
    for g = 1:length(armCM1)
        s = ['C ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(((position-1) * arm_segment_offset ) + iter) ' ' num2str(0) ' '
num2str(armCM1(g)+ mass_seg_inc*(iter-1) + ((position-1) * tile_offset )) ' '
num2str(armCM2(g)+ mass_seg_inc*(iter-1) + ((position-1) * tile_offset )) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
        Hori_seg{springcount,1} = s;
        springcount = springcount+1;
    end;
    % Braces
    for g = 1:length(armCM1)
        s = ['B ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(braceL(g)) ' ' num2str(braceLS(g)) ' ' num2str(armCM3(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armCM4(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(sqrt(2e-9^2 + length_across(iter)^2)) ' '
num2str(1)];
```

```
        Hori_seg{springcount,1} = s;
        springcount = springcount+1;
    end;
    for g = 1:length(armCM1)
        s = ['B ' num2str(((position-1) * arm_segment_offset ) + iter) ' '
num2str(braceR(g)) ' ' num2str(braceRS(g)) ' ' num2str(armCM5(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' '  num2str(armCM6(g)+ mass_seg_inc*(iter-1) +
((position-1) * tile_offset )) ' ' num2str(sqrt(2e-9^2 + length_across(iter)^2)) ' '
num2str(1)];
        Hori_seg{springcount,1} = s;
        springcount = springcount+1;
    end;
end
Hori_seg
```

Vertical mass generator source code:

```
% Written by: Vincent Mao
% last modified: 8.18.09
%
% This program generates the vertical mass nodes for each tile arm.  The
% sequences are generated from bottom to top according to the A Tile
% layout assuming a square core with the same number of base pairs for
% both the left and right dsDNA helices that compose each of the arms. The
% meshfile format is:
%       (M) <Node mass> <X coord> <Y coord> <Z coord> <Fixed mass flag>

% Lower Arm
if(vert_flip == 0)
bases_vert1 = [0 13 14 24];
end

if(vert_flip == 1)
bases_vert1 = [0 8 9 19];
end

% Upper Arm
if(vert_flip == 0 && hori_flip == 0)
bases_vert2 = [0 10 11 19]+ bases_vert1(4);
initial_x = bp; % Shift over one bp to account for offset for core
initial_y = -8.5e-9; % 24 bp from lower arm + 1 bp for core
end

if(vert_flip == 0 && hori_flip == 1)
bases_vert2 = [0 10 11 19]+ bases_vert1(4); % [0 11 12 20]
initial_x = bp;
initial_y = -2.984e-8; % 24 bp + 1 bp + 21.34 nm
end

if(vert_flip == 1 && hori_flip == 0)
bases_vert2 = [0 10 11 24]+ bases_vert1(4);
initial_x = 2.168e-8; % Coregap + 24 bp + 5 bp + 19 bp + 1 bp for gap
initial_y = -6.8e-9; % 19 bp + 1 bp for gap
end

if(vert_flip == 1 && hori_flip == 1)
bases_vert2 = [0 10 11 24]+ bases_vert1(4);
initial_x = 2.168e-8; % Coregap + 24 bp + 5 bp + 19 bp + 1 bp for gap
initial_y = -2.814e-8; % 19 bp + 1 bp for gap + 21.34 nm
```

```
end

% Initialize coordinates
coord_x =   [0    2e-9    0    2e-9];
coord_y =   bp * bases_vert1;
coord_z =   [0   0   -2e-9    -2e-9];

% Position counter for the meshfile
masscount = 1;
count = 2;
coregap = 5.02e-9;

%------------------------------
% Mass Node Generator
%------------------------------
coord1 = coord_x;
coord3 = coord_z;

if(position > 4 && position <= 8)
    initial_y = initial_y + -2.134e-8 * 2;
end

for(iter3 = 1:count)
    for(iter2 = 1:count)
        if(iter2 == 1 && iter3 == 1)
            coord2 = bp * bases_vert1;
        end
        if(iter2 == 1 && iter3 == 2)
            coord2 = bp * bases_vert2;
        end

        for i = 1:4
            for j = 1:4
                m = ['M ' num2str(mass_value_per_node) ' ' num2str(initial_x + coord1(j)+
(2.34e-9 *(iter2-1))) ' ' num2str(initial_y + coord2(i) + coregap*(iter3-1)) ' '
num2str(coord3(j)) ' ' num2str(0) ];
                Vert_mass_n{masscount,1} = m;
                masscount = masscount + 1;
            end
        end
    end
end
Vert_mass_n
```

Vertical mass node generator source code for the cross-linked model:

```
% Written by: Vincent Mao
% last modified: 12.13.09
%
% This program generates the vertical mass nodes for each tile arm.  The
% sequences are generated from bottom to top according to the A Tile
% layout assuming a square core with the same number of base pairs for
% both the left and right dsDNA helices that compose each of the arms. The
% meshfile format is:
%       (M) <Node mass> <X coord> <Y coord> <Z coord> <Fixed mass flag>
% Patterns the base lengths for each segment in the arm
% The mass numbering system is in a counter clockwise system starting with
% the back face as shown in the following diagram:
%
```

```
%        12 o------------o 13
%    14 o------------o 15 |
%       |   |        |    |       Thermo region 10
%       | 8 o--------|----o 9
%    10 o------------o 11
%
%         4 o------------o 5
%     6 o------------o 7  |
%       |   |        |    |       Thermo region 9
%       | 0 o--------|----o 1
%     2 o------------o 3
%       Left arm node map
%
% The thermo region map for the horizontal mass generation goes like this:
%       14  16
%       13  15
%       ------
%        Core
%        Core
%       ------
%       10  12
%        9  11
%-------------------------------------------------------------------------
% vert_flip flag: flip the vertical arms
% 1x1 = lower right regions (11 and 15)
% 1x2 = lower left regions  (9 and 13)
% 2x1 = upper right regions (12 and 16)
% 2x2 = upper left regions  (10 and 14)
%-------------------------------------------------------------------------

coregap = 5.02e-9;
% Calculated x offset for multiple tile generation:
%   Coregap + 24 bp + 5 bp + 19 bp + 1 bp for gap
x_tile_offset = coregap + 24 * bp + 5 * bp + 19 * bp + bp;
% Calculated y offset for multiple tile generation:
%  1 bp + 24 bp + 5 bp + 19 bp + coregap - 1 bp
y_tile_offset = bp + 24 * bp + 5 * bp + 19 * bp + coregap - bp;

if(vert_flip == 0)
    bases_vert1 = [0 13 14 24];
    bases_vert2 = [0 10 11 19]          + bases_vert1(4);
    initial_x = bp; % Shift over one bp to account for offset for core
end

if(vert_flip == 1)
    bases_vert1 = [0 8 9 19];
    bases_vert2 = [0 10 11 24]          + bases_vert1(4);
    initial_x = x_tile_offset;
end

%------------
% Lower Arm
%------------
if(vert_flip == 0 && hori_flip == 0)
    bases_vert1x1 = [0 5 6 13 14 24];   % [5 1 7    10 ]
    bases_vert1x2 = [0 13 14 17 18 24]; % [13   3 1 6]
    initial_x = bp; % Shift over one bp to account for offset for core
end

if(vert_flip == 1 && hori_flip == 0)
    bases_vert1x1 = [0 1 8 9 19];           % [1 7  10]
    bases_vert1x2 = [0 8 9 12 13 16 17 19]; % [8    3 1 3 1 2]
    initial_x = x_tile_offset;
```

```
end

if(vert_flip == 0 && hori_flip == 1)
    bases_vert1x1 = [0 13 14 17 18 24]; % [13   3 1 6]
    bases_vert1x2 = [0 5 6 13 14 24];   % [5 1 7    10]
    initial_x = bp; % Shift over one bp to account for offset for core
end

if(vert_flip == 1 && hori_flip == 1)
    bases_vert1x1 = [0 8 9 12 13 16 17 19]; % [ 8   3 1 3 1 2]
    bases_vert1x2 = [0 1 8 9 19];           % [1 7  10]
    initial_x = x_tile_offset;
end

%------------
% Upper Arm
%------------
if(vert_flip == 0 && hori_flip == 0)
    bases_vert2x1 = [0 10 11 18 19]     + bases_vert1(4);   % [10   7 1]
    bases_vert2x2 = [0 3 4 7 8 10 11 19]+ bases_vert1(4);   % [3 1 3 1 2    8]
    initial_y = -8.5e-9;    % 24 bp from lower arm + 1 bp for core
end

if(vert_flip == 0 && hori_flip == 1)
    bases_vert2x1 = [0 3 4 7 8 10 11 19]+ bases_vert1(4);   % [3 1 3 1 2    8]
    bases_vert2x2 = [0 10 11 18 19]     + bases_vert1(4);   % [10   7 1]
    initial_y = -2.984e-8; % 24 bp + 1 bp + 21.34 nm
end

if(vert_flip == 1 && hori_flip == 0)
    bases_vert2x1 = [0 10 11 19 20 24]  + bases_vert1(4);   % [10   8 1 4]
    bases_vert2x2 = [0 7 8 10 11 24]    + bases_vert1(4);   % [7 1 2    13]
    initial_y = -6.8e-9;    % 19 bp + 1 bp for gap
end

if(vert_flip == 1 && hori_flip == 1)
    bases_vert2x1 = [0 7 8 10 11 24]    + bases_vert1(4);   % [7 1 2    13]
    bases_vert2x2 = [0 10 11 19 20 24]  + bases_vert1(4);   % [10   8 1 4]
    initial_y = -2.814e-8;  % 19 bp + 1 bp for gap + 21.34 nm
end

% Initialize coordinates
coord1 =   [0    2e-9    0    2e-9];
coord3 =   [0   0   -2e-9    -2e-9];

% Position counter for the meshfile
masscount = 1;
count = 2;

%------------------------------
% Mass Node Generator
%------------------------------
if(position > 4 && position <= 8)
    initial_y = initial_y + -y_tile_offset * 2;
end

for(arm_gen = 1:count)
    %-------------------------------------------------------------------------
    % Generate the whole arm                          _  _
    %                                                | || |
    %                                                | || |
    %                                                |_||_|
    %                                                | || |
```

```
    %                                       | || |
    %                                       |_||_|
    %                                        1  2
    % reg_gen - horizontal identifier: 1 is left side, 2 is right
    % arm_gen - vertical identifier: 1 is the arm below the core, 2 is above
    %-------------------------------------------------------------------------
    for(reg_gen = 1:count)
        if(reg_gen == 1 && arm_gen == 1)    % lower left
            coord2 = bp * bases_vert1x2;
        end
        if(reg_gen == 1 && arm_gen == 2)    % upper left
            coord2 = bp * bases_vert2x2;
        end
        if(reg_gen == 2 && arm_gen == 1)    % lower right
            coord2 = bp * bases_vert1x1;
        end
        if(reg_gen == 2 && arm_gen == 2)    % upper right
            coord2 = bp * bases_vert2x1;
        end

        for i = 1:length(coord2)
            for j = 1:4
                % Calculated xyz coords
                x_coord = initial_x + coord1(j) + (2.34e-9 * (reg_gen - 1));
                y_coord = initial_y + coord2(i) + coregap * (arm_gen - 1);
                z_coord = coord3(j);

                % Generate the meshfile information
                m = ['M ' num2str(mass_value_per_node) ' ' num2str(x_coord) ' '
num2str(y_coord) ' ' num2str(z_coord) ' ' num2str(0) ];
                Vert_mass_n{masscount,1} = m;
                masscount = masscount + 1;
            end
        end
    end
end
Vert_mass_n
```

Vertical spring segment generator source code

```
% Calculate the length of each segment
length1 = (bases_vert1(2) - bases_vert1(1)) * bp;
length2 = (bases_vert1(4) - bases_vert1(3)) * bp;
length3 = length1;
length4 = length2;
length5 = (bases_vert2(2) - bases_vert2(1)) * bp;
length6 = (bases_vert2(4) - bases_vert2(3)) * bp;
length7 = length5;
length8 = length6;

bases_vert1 = bases_vert1 + bases_vert1(4);
bases_vert2 = bases_vert2 + bases_vert2(4);

% Initialize the arrays
bases = [length1 length2 length3 length4 length5 length6 length7 length8] /bp;
length_vert = [length1 length2 length3 length4 length5 length6 length7 length8];
ext = avg_mass_per_bp * ssDNA_EC * bases;% bases * 9777;
ext2 = avg_mass_per_bp * dsDNA_EC * bases; %bases * 7984;
```

```
% Spring connections
springseg = [1 2 3 4];

rodL = [2 1 2 1];
rodR = [3 4 4 3];
stiffL = [1 2 3 4];
stiffR = [5 6 7 8];

braceL = [2 1 2 4];
braceR = [3 4 1 3];
braceLS = [2 1 7 3];
braceRS = [6 5 8 4];

armSM1 = [0 1 2 3];
armSM2 = [4 5 6 7];

% Face cross braces
armLM1 = [0 4 1 0];
armLM2 = [1 5 3 2];
armRM1 = [2 6 5 4];
armRM2 = [3 7 7 6];

% Side cross braces
armCM1 = [0 1 4 5];
armCM2 = [3 2 7 6];
armCM3 = [0 1 1 3];
armCM4 = [5 4 7 5];
armCM5 = [2 3 0 2];
armCM6 = [7 6 6 4];

%------------------------------
% Melting Temps and Cross Links
%------------------------------
% Four sets of Tm Theories:
%   1. 16 Thermo regions
%   2. Crossover theory links thermo regions
%   3. Lowest melting temperature on the shell
%   4. Average core melting temperatures with outer thermo regions
if(mod(position, 4) == 1)
    if(theory == 1)
        melt = [65 53 60 42 43 44 56 31];
    end
    if(theory == 2)
        melt = [59 58.3 51 58.3 62.3 43.5 62.3 43.5];
    end
    if(theory == 3)
        melt = [30 30 42 42 35 35 31 31];
    end
    if(theory == 4)
        melt = [61.7 58.3 59.2 58.3 62.3 53.2 62.3 46.7];
    end

%     if(mod(position, 8) == 1)
% %         melt = [66.7 58.3 66.7 58.3 62.3 37.5 62.3 37.5];
%         melt = [66.7 58.3 66.7 58.3 62.3 56.8 62.3 56.8];
%     end
%     if(mod(position, 8) == 5)
% %         melt = [66.7 58.3 66.7 58.3 62.3 56.8 62.3 56.8];
%         melt = [66.7 58.3 66.7 58.3 62.3 56.8 62.3 56.8];
%     end
end

if(mod(position, 4) == 2)
```

```
    if(theory == 1)
        melt = [44 43 31 56 53 65 42 60];
    end
    if(theory == 2)
        melt = [43.5 62.3 43.5 62.3 58.3 59 58.3 51];
    end
    if(theory == 3)
        melt = [31 31 35 35 42 42 30 30];
    end
    if(theory == 4)
        melt = [53.2 62.3 46.7 62.3 58.3 61.7 58.3 59.2];
    end
%     if(mod(position, 8) == 2)
% %         melt = [56.8 62.3 56.8 62.3 58.3 62.5 58.3 62.5];
%         melt = [56.8 62.3 56.8 62.3 58.3 66.7 58.3 66.7];
%     end
%     if(mod(position, 8) == 6)
% %         melt = [56.8 62.3 56.8 62.3 58.3 66.7 58.3 66.7];
%         melt = [56.8 62.3 56.8 62.3 58.3 66.7 58.3 66.7];
%     end
end

if(mod(position, 4) == 3)
    if(theory == 1)
        melt = [60 42 65 53 56 31 43 44];
    end
    if(theory == 2)
        melt = [51 58.3 59 58.3 62.3 43.5 62.3 43.5];
    end
    if(theory == 3)
        melt = [30 30 42 42 35 35 31 31];
    end
    if(theory == 4)
        melt = [59.2 58.3 61.7 58.3 62.3 46.7 62.3 53.2];
    end
%     if(mod(position, 8) == 3)
% %         melt = [66.7 58.3 66.7 58.3 62.3 56.8 62.3 56.8];
%         melt = [66.7 58.3 66.7 58.3 62.3 56.8 62.3 56.8];
%     end
%     if(mod(position, 8) == 7)
% %         melt = [62.5 58.3 62.5 58.3 62.3 56.8 62.3 56.8];
%         melt = [66.7 58.3 66.7 58.3 62.3 56.8 62.3 56.8];
%     end
end

if(mod(position, 4) == 0)
    if(theory == 1)
         melt = [31 56 44 43 42 60 53 65];
    end
    if(theory == 2)
        melt = [43.5 62.3 43.5 62.3 58.3 51 58.3 59];
    end
    if(theory == 3)
        melt = [35 35 31 31 30 30 42 42];
    end
    if(theory == 4)
        melt = [46.7 62.3 53.2 62.3 58.3 59.2 58.3 61.7];
    end

%     if(mod(position, 8) == 4)
% %         melt = [56.8 62.3 56.8 62.3 58.3 66.7 58.3 66.7];
%         melt = [56.8 62.3 56.8 62.3 58.3 66.7 58.3 66.7];
%     end
```

```
%     if(mod(position, 8) == 0)
% %         melt = [37.5 62.3 37.5 62.3 58.3 66.7 58.3 66.7];
%         melt = [56.8 62.3 56.8 62.3 58.3 66.7 58.3 66.7];
%     end
end

%------------------------------
% Counters and offset values
%------------------------------
springcount = 1;

sticky_inc = 4;
mass_seg_inc = 8;
arm_node_offset = 32;
arm_segment_offset = 16;
offset = 64;
tile_offset = 128;

%------------------------------
% Spring Segment Generator
%------------------------------
seg = 4;
count = 2;
counter = 8;

% Real spring links
for(iter2 = 1:count)
    for(iter = 1:seg)
        for g = 1:seg
            s = ['S ' num2str( ((position-1) * arm_segment_offset ) + mass_seg_inc + iter
+ (seg* (iter2-1))) ' ' num2str(springseg(g)) ' ' num2str(0) ' ' num2str(armSM1(g)+
mass_seg_inc*(iter-1) + arm_node_offset*(iter2-1) + offset + ((position-1) * tile_offset
)) ' '  num2str(armSM2(g)+ mass_seg_inc*(iter-1)+ arm_node_offset*(iter2-1) + offset +
((position-1) * tile_offset )) ' ' num2str(length_vert(iter + (seg* (iter2-1)))) ' '
num2str(melt(iter + (seg* (iter2-1)))) ' ' num2str(bases(iter + (seg* (iter2-1)))) ' '
num2str(ext(iter + (seg* (iter2-1)))) ' ' num2str(ext2(iter + (seg* (iter2-1)))) ' '
num2str(DNA_scale(iter + (seg* (iter2-1)))) ];
            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end
    end
end

for(iter = 1:counter)
    % Stiff springs
    for g = 1:length(armLM1)
        if(ds_to_ss == 0)
            s = ['T ' num2str(((position-1) * arm_segment_offset ) + mass_seg_inc + iter)
' ' num2str(rodL(g)) ' ' num2str(stiffL(g)) ' ' num2str(armLM1(g)+ mass_seg_inc*(iter-1)
+ offset + ((position-1) * tile_offset )) ' '  num2str(armLM2(g)+ mass_seg_inc*(iter-1) +
offset + ((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
        end
        if(ds_to_ss == 1)
            if(g > 2)
                s = ['T ' num2str(((position-1) * arm_segment_offset ) + mass_seg_inc +
iter) ' ' num2str(rodL(g)) ' ' num2str(stiffL(g)) ' ' num2str(armLM1(g)+
mass_seg_inc*(iter-1) + offset + ((position-1) * tile_offset )) ' '  num2str(armLM2(g)+
mass_seg_inc*(iter-1) + offset + ((position-1) * tile_offset )) ' ' num2str(2e-9) ' '
num2str(Stiff_scale)];
            else
                s = ['C ' num2str(((position-1) * arm_segment_offset ) + mass_seg_inc +
iter) ' ' num2str(rodL(g)) ' ' num2str(stiffL(g)) ' ' num2str(armLM1(g)+
mass_seg_inc*(iter-1) + offset + ((position-1) * tile_offset )) ' '  num2str(armLM2(g)+
```

```
mass_seg_inc*(iter-1) + offset + ((position-1) * tile_offset )) ' ' num2str(2e-9) ' '
num2str(Stiff_scale)];
            end
        end
        Vert_seg{springcount,1} = s;
        springcount = springcount + 1;
    end;
    for g = 1:length(armRM1)
        if(ds_to_ss == 0)
            s = ['T ' num2str(((position-1) * arm_segment_offset ) + mass_seg_inc + iter)
' ' num2str(rodR(g)) ' ' num2str(stiffR(g)) ' ' num2str(armRM1(g)+ mass_seg_inc*(iter-1)
+ offset + ((position-1) * tile_offset )) ' '  num2str(armRM2(g)+ mass_seg_inc*(iter-1) +
offset + ((position-1) * tile_offset )) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
        end
        if(ds_to_ss == 1)
            if(g > 2)
                s = ['T ' num2str(((position-1) * arm_segment_offset ) + mass_seg_inc +
iter) ' ' num2str(rodR(g)) ' ' num2str(stiffR(g)) ' ' num2str(armRM1(g)+
mass_seg_inc*(iter-1) + offset + ((position-1) * tile_offset )) ' '  num2str(armRM2(g)+
mass_seg_inc*(iter-1) + offset + ((position-1) * tile_offset )) ' ' num2str(2e-9) ' '
num2str(Stiff_scale)];
            else
                s = ['C ' num2str(((position-1) * arm_segment_offset ) + mass_seg_inc +
iter) ' ' num2str(rodR(g)) ' ' num2str(stiffR(g)) ' ' num2str(armRM1(g)+
mass_seg_inc*(iter-1) + offset + ((position-1) * tile_offset )) ' '  num2str(armRM2(g)+
mass_seg_inc*(iter-1) + offset + ((position-1) * tile_offset )) ' ' num2str(2e-9) ' '
num2str(Stiff_scale)];
            end
        end
        Vert_seg{springcount,1} = s;
        springcount = springcount + 1;
    end;
    % Stiff springs on the face of the cube
    for g = 1:length(armCM1)
        s = ['C ' num2str(((position-1) * arm_segment_offset ) + mass_seg_inc + iter) ' '
num2str(((position-1) * arm_segment_offset ) + iter) ' ' num2str(0) ' '
num2str(armCM1(g)+ mass_seg_inc*(iter-1) + offset + ((position-1) * tile_offset )) ' '
num2str(armCM2(g)+ mass_seg_inc*(iter-1) + offset + ((position-1) * tile_offset )) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
        Vert_seg{springcount,1} = s;
        springcount = springcount+1;
    end;
    % Braces
    for g = 1:length(armCM1)
        s = ['B ' num2str(((position-1) * arm_segment_offset ) + mass_seg_inc + iter) ' '
num2str(braceL(g)) ' ' num2str(braceLS(g)) ' ' num2str(armCM3(g)+ mass_seg_inc*(iter-1) +
offset + ((position-1) * tile_offset )) ' '  num2str(armCM4(g)+ mass_seg_inc*(iter-1) +
offset + ((position-1) * tile_offset )) ' ' num2str(sqrt(2e-9^2 + length_vert(iter)^2)) '
' num2str(1)];
        Vert_seg{springcount,1} = s;
        springcount = springcount+1;
    end;
    for g = 1:length(armCM1)
        s = ['B ' num2str(((position-1) * arm_segment_offset ) + mass_seg_inc + iter) ' '
num2str(braceR(g)) ' ' num2str(braceRS(g)) ' ' num2str(armCM5(g)+ mass_seg_inc*(iter-1) +
offset + ((position-1) * tile_offset )) ' '  num2str(armCM6(g)+ mass_seg_inc*(iter-1) +
offset + ((position-1) * tile_offset )) ' ' num2str(sqrt(2e-9^2 + length_vert(iter)^2)) '
' num2str(1)];
        Vert_seg{springcount,1} = s;
        springcount = springcount+1;
    end;
end
Vert_seg
```

Vertical spring segment generator code for cross-linked models:

```
% Calculate the length of each segment
length1 = (bases_vert1(2) - bases_vert1(1)) * bp;
length2 = (bases_vert1(4) - bases_vert1(3)) * bp;
length3 = length1;
length4 = length2;
length5 = (bases_vert2(2) - bases_vert2(1)) * bp;
length6 = (bases_vert2(4) - bases_vert2(3)) * bp;
length7 = length5;
length8 = length6;

bases_vert1 = bases_vert1 + bases_vert1(4);
bases_vert2 = bases_vert2 + bases_vert2(4);

% Initialize the arrays
bases = [length1 length2 length3 length4 length5 length6 length7 length8] /bp;
length_vert = [length1 length2 length3 length4 length5 length6 length7 length8];
ext = avg_mass_per_bp * ssDNA_EC * bases;% bases * 9777;
ext2 = avg_mass_per_bp * dsDNA_EC * bases; %bases * 7984;
ext_c1 = avg_mass_per_bp * ssDNA_EC;
ext_c2 = avg_mass_per_bp * dsDNA_EC;

% Spring connections
springseg = [1 2 3 4];

rodL = [2 1 2 1];
rodR = [3 4 4 3];
stiffL = [1 2 3 4];
stiffR = [5 6 7 8];

braceL = [2 1 2 4];
braceR = [3 4 1 3];
braceLS = [2 1 7 3];
braceRS = [6 5 8 4];

armSM1 = [0 1 2 3];
armSM2 = [4 5 6 7];

% Face cross braces
armLM1 = [0 4 1 0];
armLM2 = [1 5 3 2];
armRM1 = [2 6 5 4];
armRM2 = [3 7 7 6];

% Side cross braces
armCM1 = [0 1 4 5];
armCM2 = [3 2 7 6];
armCM3 = [0 1 1 3];
armCM4 = [5 4 7 5];
armCM5 = [2 3 0 2];
armCM6 = [7 6 6 4];

%------------------------------
% Melting Temps and Cross Links
%------------------------------
% Four sets of Tm Theories:
%   1. 16 Thermo regions
%   2. Crossover theory links thermo regions
%   3. Lowest melting temperature on the shell
```

```
%   4. Average core melting temperatures with outer thermo regions
if(mod(position, 4) == 1)
    if(theory == 1)
        melt = [65 53 60 42 43 44 56 31];
    end
    if(theory == 2)
        melt = [59 58.3 51 58.3 62.3 43.5 62.3 43.5];
    end
    if(theory == 3)
        melt = [30 30 42 42 35 35 31 31];
    end
    if(theory == 4)
        melt = [61.7 58.3 59.2 58.3 62.3 53.2 62.3 46.7];
    end

%     if(mod(position, 8) == 1)
% %         melt = [66.7 58.3 66.7 58.3 62.3 37.5 62.3 37.5];
%         melt = [66.7 58.3 66.7 58.3 62.3 56.8 62.3 56.8];
%     end
%     if(mod(position, 8) == 5)
% %         melt = [66.7 58.3 66.7 58.3 62.3 56.8 62.3 56.8];
%         melt = [66.7 58.3 66.7 58.3 62.3 56.8 62.3 56.8];
%     end
end

if(mod(position, 4) == 2)
    if(theory == 1)
        melt = [44 43 31 56 53 65 42 60];
    end
    if(theory == 2)
        melt = [43.5 62.3 43.5 62.3 58.3 59 58.3 51];
    end
    if(theory == 3)
        melt = [31 31 35 35 42 42 30 30];
    end
    if(theory == 4)
        melt = [53.2 62.3 46.7 62.3 58.3 61.7 58.3 59.2];
    end
%     if(mod(position, 8) == 2)
% %         melt = [56.8 62.3 56.8 62.3 58.3 62.5 58.3 62.5];
%         melt = [56.8 62.3 56.8 62.3 58.3 66.7 58.3 66.7];
%     end
%     if(mod(position, 8) == 6)
% %         melt = [56.8 62.3 56.8 62.3 58.3 66.7 58.3 66.7];
%         melt = [56.8 62.3 56.8 62.3 58.3 66.7 58.3 66.7];
%     end
end

if(mod(position, 4) == 3)
    if(theory == 1)
        melt = [60 42 65 53 56 31 43 44];
    end
    if(theory == 2)
        melt = [51 58.3 59 58.3 62.3 43.5 62.3 43.5];
    end
    if(theory == 3)
        melt = [30 30 42 42 35 35 31 31];
    end
    if(theory == 4)
        melt = [59.2 58.3 61.7 58.3 62.3 46.7 62.3 53.2];
    end
%     if(mod(position, 8) == 3)
% %         melt = [66.7 58.3 66.7 58.3 62.3 56.8 62.3 56.8];
```

```
%         melt = [66.7 58.3 66.7 58.3 62.3 56.8 62.3 56.8];
%     end
%     if(mod(position, 8) == 7)
% %         melt = [62.5 58.3 62.5 58.3 62.3 56.8 62.3 56.8];
%         melt = [66.7 58.3 66.7 58.3 62.3 56.8 62.3 56.8];
%     end
end

if(mod(position, 4) == 0)
    if(theory == 1)
         melt = [31 56 44 43 42 60 53 65];
    end
    if(theory == 2)
        melt = [43.5 62.3 43.5 62.3 58.3 51 58.3 59];
    end
    if(theory == 3)
        melt = [35 35 31 31 30 30 42 42];
    end
    if(theory == 4)
        melt = [46.7 62.3 53.2 62.3 58.3 59.2 58.3 61.7];
    end

%     if(mod(position, 8) == 4)
% %         melt = [56.8 62.3 56.8 62.3 58.3 66.7 58.3 66.7];
%         melt = [56.8 62.3 56.8 62.3 58.3 66.7 58.3 66.7];
%     end
%     if(mod(position, 8) == 0)
% %         melt = [37.5 62.3 37.5 62.3 58.3 66.7 58.3 66.7];
%         melt = [56.8 62.3 56.8 62.3 58.3 66.7 58.3 66.7];
%     end
end

%------------------------------
% Counters and offset values
%------------------------------
springcount = 1;

sticky_inc = 4;
mass_seg_inc = 8;
arm_node_offset = 32;
arm_segment_offset = 16;
offset = 64;
tile_offset = 212;

%------------------------------
% Spring Segment Generator
%------------------------------
seg = 4;
count = 2;
counter = 8;

%--------------------------------------------------------------------------
% DNA springs
for(iter2 = 1:count)
    for(iter = 1:seg)
        seg_num = ((position-1) * arm_segment_offset ) + mass_seg_inc + iter + (seg*
(iter2-1));
        offset2 = mass_seg_inc*(iter-1) + arm_node_offset*(iter2-1) + offset +
((position-1) * tile_offset ) + 48;
        element = iter + (seg* (iter2-1));
        if(mod(seg_num, arm_segment_offset) == 9 || mod(seg_num, arm_segment_offset) ==
12 || mod(seg_num, arm_segment_offset) == 14 || mod(seg_num, arm_segment_offset) == 15)
            for g = 1:seg
```

```
                s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' ' num2str(0) ' '
num2str(armSM1(g)+ offset2) ' '  num2str(armSM2(g)+ offset2) ' '
num2str(length_vert(element)) ' ' num2str(melt(element)) ' ' num2str(bases(element)) ' '
num2str(ext(element)) ' ' num2str(ext2(element)) ' ' num2str(1) ];
                Vert_seg{springcount,1} = s;
                springcount = springcount + 1;
            end
        else

        if(mod(seg_num, arm_segment_offset) == 10)
            base_length = [3 2 6];
            for h = 1:3
                x_length = base_length(h) * bp;
                ext_1 = base_length(h) * ext_c1;
                ext_2 = base_length(h) * ext_c2;
                if(h > 1)
                    armSM1 = armSM1 + 4;
                    armSM2 = armSM2 + 4;
                end
                if(h == 2)
                    for g = 1:seg
                        s = ['X ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(h) ' ' num2str(armSM1(g) + offset2) ' '  num2str(armSM2(g) + offset2) ' '
num2str(x_length) ' ' num2str(0) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' '
num2str(ext_2) ' ' num2str(1) ];
                        Vert_seg{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                else
                    if(h == 1 || h == 3)
                        for g = 1:seg
                            s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armSM1(g) + offset2) ' '  num2str(armSM2(g) + offset2) ' '
num2str(x_length) ' ' num2str(melt(element)) ' ' num2str(base_length(h)) ' '
num2str(ext_1) ' ' num2str(ext_2) ' ' num2str(1) ];
                            Vert_seg{springcount,1} = s;
                            springcount = springcount + 1;
                        end
                    end
                end
            end
        end

        if(mod(seg_num, arm_segment_offset) == 11)
            base_length = [5 2 7];
            for h = 1:3
                x_length = base_length(h) * bp;
                ext_1 = base_length(h) * ext_c1;
                ext_2 = base_length(h) * ext_c2;
                if(h > 1)
                    armSM1 = armSM1 + 4;
                    armSM2 = armSM2 + 4;
                end
                if(h == 2)
                    for g = 1:seg
                        s = ['X ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(h) ' ' num2str(armSM1(g) + offset2) ' '  num2str(armSM2(g) + offset2) ' '
num2str(x_length) ' ' num2str(0) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' '
num2str(ext_2) ' ' num2str(1) ];
                        Vert_seg{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                else
```

```
                    if(h == 1 || h == 3)
                        for g = 1:seg
                            s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armSM1(g) + offset2) ' '  num2str(armSM2(g) + offset2) ' '
num2str(x_length) ' ' num2str(melt(element)) ' ' num2str(base_length(h)) ' '
num2str(ext_1) ' ' num2str(ext_2) ' ' num2str(1) ];
                            Vert_seg{springcount,1} = s;
                            springcount = springcount + 1;
                        end
                    end
                end
            end
        end

        if(mod(seg_num, arm_segment_offset) == 13)
           base_length = [3 2 3 2 2];
           for h = 1:5
                x_length = base_length(h) * bp;
                ext_1 = base_length(h) * ext_c1;
                ext_2 = base_length(h) * ext_c2;
                if(h > 1)
                    armSM1 = armSM1 + 4;
                    armSM2 = armSM2 + 4;
                end
                if(h == 2 || h == 4)
                    for g = 1:seg
                        s = ['X ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(h) ' ' num2str(armSM1(g)+ offset2) ' '  num2str(armSM2(g)+ offset2) ' '
num2str(x_length) ' ' num2str(0) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' '
num2str(ext_2) ' ' num2str(1) ];
                        Vert_seg{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                else
                    for g = 1:seg
                        s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armSM1(g) + offset2) ' '  num2str(armSM2(g) + offset2) ' '
num2str(x_length) ' ' num2str(melt(element)) ' ' num2str(base_length(h)) ' '
num2str(ext_1) ' ' num2str(ext_2) ' ' num2str(1) ];
                        Vert_seg{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
            end
        end
             if(mod(seg_num, arm_segment_offset) == 0)
                base_length = [7 2];
                for h = 1:2
                    x_length = base_length(h) * bp;
                    ext_1 = base_length(h) * ext_c1;
                    ext_2 = base_length(h) * ext_c2;
                    if(h == 2)
                        armSM1 = armSM1 + 4;
                        armSM2 = armSM2 + 4;
                        for g = 1:seg
                            s = ['X ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(h) ' ' num2str(armSM1(g)+ offset2) ' '  num2str(armSM2(g)+ offset2) ' '
num2str(x_length) ' ' num2str(0) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' '
num2str(ext_2) ' ' num2str(1) ];
                            Vert_seg{springcount,1} = s;
                            springcount = springcount + 1;
                        end
                    else
```

```
                        for g = 1:seg
                            s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armSM1(g) + offset2) ' '  num2str(armSM2(g) + offset2) ' '
num2str(x_length) ' ' num2str(melt(element)) ' ' num2str(base_length(h)) ' '
num2str(ext_1) ' ' num2str(ext_2) ' ' num2str(1) ];
                            Vert_seg{springcount,1} = s;
                            springcount = springcount + 1;
                        end
                    end
                end
            end
        end
    end
end
%-------------------------------------------------------------------------

%-------------------------------------------------------------------------
% Structural springs
for(iter = 1:counter)
    % Stiff springs
    seg_num = ((position-1) * arm_segment_offset ) + mass_seg_inc + iter;
    offset2 = mass_seg_inc*(iter-1) + arm_node_offset*(iter2-1) + offset + ((position-1)
* tile_offset );
    if(mod(seg_num, arm_segment_offset) == 9)
        for g = 1:length(armLM1)
            if(ds_to_ss == 0)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL(g)) ' ' num2str(stiffL(g)) '
' num2str(armLM1(g) + offset2 + 16) ' '  num2str(armLM2(g) + offset2 + 16) ' '
num2str(2e-9) ' ' num2str(Stiff_scale)];
            end
            if(ds_to_ss == 1)
                if(g > 2)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL(g)) ' '
num2str(stiffL(g)) ' ' num2str(armLM1(g) + offset2 + 16) ' '  num2str(armLM2(g) + offset2
+ 16) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                else
                    s = ['C ' num2str(seg_num) ' ' num2str(rodL(g)) ' '
num2str(stiffL(g)) ' ' num2str(armLM1(g) + offset2 + 16) ' '  num2str(armLM2(g) + offset2
+ 16) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                end
            end
            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end
        for g = 1:length(armRM1)
            if(ds_to_ss == 0)
                s = ['T ' num2str(seg_num) ' ' num2str(rodR(g)) ' ' num2str(stiffR(g)) '
' num2str(armRM1(g)+ offset2 + 16) ' '  num2str(armRM2(g)+ offset2 + 16) ' ' num2str(2e-
9) ' ' num2str(Stiff_scale)];
            end
            if(ds_to_ss == 1)
                if(g > 2)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodR(g)) ' '
num2str(stiffR(g)) ' ' num2str(armRM1(g)+ offset2 + 16) ' '  num2str(armRM2(g)+ offset2 +
16) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                else
                    s = ['C ' num2str(seg_num) ' ' num2str(rodR(g)) ' '
num2str(stiffR(g)) ' ' num2str(armRM1(g)+ offset2 + 16) ' '  num2str(armRM2(g)+ offset2 +
16) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                end
            end
            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
```

```
        end;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1)
            s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1(g)+ offset2 + 16) ' '  num2str(armCM2(g)+ offset2 + 16) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        % Braces
        for g = 1:length(armCM1)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3(g)+ offset2 + 16) ' '  num2str(armCM4(g)+ offset2 + 16) ' '
num2str(sqrt(2e-9^2 + length_vert(iter)^2)) ' ' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5(g)+ offset2 + 16) ' '  num2str(armCM6(g)+ offset2 + 16) ' '
num2str(sqrt(2e-9^2 + length_vert(iter)^2)) ' ' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
    end

    if(mod(seg_num, arm_segment_offset) == 12)
        for g = 1:length(armLM1)
            if(ds_to_ss == 0)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL(g)) ' ' num2str(stiffL(g)) '
' num2str(armLM1(g) + offset2 + 32) ' '  num2str(armLM2(g) + offset2 + 32) ' '
num2str(2e-9) ' ' num2str(Stiff_scale)];
            end
            if(ds_to_ss == 1)
                if(g > 2)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL(g)) ' '
num2str(stiffL(g)) ' ' num2str(armLM1(g) + offset2 + 32) ' '  num2str(armLM2(g) + offset2
+ 32) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                else
                    s = ['C ' num2str(seg_num) ' ' num2str(rodL(g)) ' '
num2str(stiffL(g)) ' ' num2str(armLM1(g) + offset2 + 32) ' '  num2str(armLM2(g) + offset2
+ 32) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                end
            end
            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end
        for g = 1:length(armRM1)
            if(ds_to_ss == 0)
                s = ['T ' num2str(seg_num) ' ' num2str(rodR(g)) ' ' num2str(stiffR(g)) '
' num2str(armRM1(g)+ offset2 + 32) ' '  num2str(armRM2(g)+ offset2 + 32) ' ' num2str(2e-
9) ' ' num2str(Stiff_scale)];
            end
            if(ds_to_ss == 1)
                if(g > 2)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodR(g)) ' '
num2str(stiffR(g)) ' ' num2str(armRM1(g)+ offset2 + 32) ' '  num2str(armRM2(g)+ offset2 +
32) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                else
                    s = ['C ' num2str(seg_num) ' ' num2str(rodR(g)) ' '
num2str(stiffR(g)) ' ' num2str(armRM1(g)+ offset2 + 32) ' '  num2str(armRM2(g)+ offset2 +
32) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                end
```

```
            end
            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1)
            s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1(g)+ offset2 + 32) ' '  num2str(armCM2(g)+ offset2 + 32) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        % Braces
        for g = 1:length(armCM1)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3(g)+ offset2 + 32) ' '  num2str(armCM4(g)+ offset2 + 32) ' '
num2str(sqrt(2e-9^2 + length_vert(iter)^2)) ' ' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5(g)+ offset2 + 32) ' '  num2str(armCM6(g)+ offset2 + 32) ' '
num2str(sqrt(2e-9^2 + length_vert(iter)^2)) ' ' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
    end

    if(mod(seg_num, arm_segment_offset) == 14 || mod(seg_num, arm_segment_offset) == 15)
        for g = 1:length(armLM1)
            if(ds_to_ss == 0)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL(g)) ' ' num2str(stiffL(g)) '
' num2str(armLM1(g) + offset2 + 48) ' '  num2str(armLM2(g) + offset2 + 48) ' '
num2str(2e-9) ' ' num2str(Stiff_scale)];
            end
            if(ds_to_ss == 1)
                if(g > 2)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL(g)) ' '
num2str(stiffL(g)) ' ' num2str(armLM1(g) + offset2 + 48) ' '  num2str(armLM2(g) + offset2
+ 48) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                else
                    s = ['C ' num2str(seg_num) ' ' num2str(rodL(g)) ' '
num2str(stiffL(g)) ' ' num2str(armLM1(g) + offset2 + 48) ' '  num2str(armLM2(g) + offset2
+ 48) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                end
            end
            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end
        for g = 1:length(armRM1)
            if(ds_to_ss == 0)
                s = ['T ' num2str(seg_num) ' ' num2str(rodR(g)) ' ' num2str(stiffR(g)) '
' num2str(armRM1(g)+ offset2 + 48) ' '  num2str(armRM2(g)+ offset2 + 48) ' ' num2str(2e-
9) ' ' num2str(Stiff_scale)];
            end
            if(ds_to_ss == 1)
                if(g > 2)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodR(g)) ' '
num2str(stiffR(g)) ' ' num2str(armRM1(g)+ offset2 + 48) ' '  num2str(armRM2(g)+ offset2 +
48) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                else
```

```
                    s = ['C ' num2str(seg_num) ' ' num2str(rodR(g)) ' '
num2str(stiffR(g)) ' ' num2str(armRM1(g)+ offset2 + 48) ' '  num2str(armRM2(g)+ offset2 +
48) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                end
            end
            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1)
            s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1(g)+ offset2 + 48) ' '  num2str(armCM2(g)+ offset2 + 48) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        % Braces
        for g = 1:length(armCM1)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3(g)+ offset2 + 48) ' '  num2str(armCM4(g)+ offset2 + 48) ' '
num2str(sqrt(2e-9^2 + length_vert(iter)^2)) ' ' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5(g)+ offset2 + 48) ' '  num2str(armCM6(g)+ offset2 + 48) ' '
num2str(sqrt(2e-9^2 + length_vert(iter)^2)) ' ' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
    end
%-------------------------------------------------------------------------

%-------------------------------------------------------------------------
% Structural springs for cross linked regions
    adjust = ((position - 1) * tile_offset);
    if(mod(seg_num, arm_segment_offset) == 10)
    % new node connections for cross-linked springs
        armLM1_10 =  [36 38 40 42 32 33 36 37 40 41 44 45 36 37 38 39 36 38 37 39 36 37
40 41 32 34 44 46] + 88 + adjust;
        armLM2_10 =  [37 39 41 43 34 35 38 39 42 43 46 47 41 40 43 42 42 40 43 41 39 38
43 42 33 35 45 47] + 88 + adjust;
        rodL_10 =    [2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1];
        stiffL_10 =  [1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4];

        for g = 1:length(armLM1_10)
            if(g < 13)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_10(g)) ' '
num2str(stiffL_10(g)) ' ' num2str(armLM1_10(g)) ' '  num2str(armLM2_10(g)) ' '
num2str(sStrain) ' ' num2str(Stiff_scale)];
            end
            if(g > 12 && g < 21)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_10(g)) ' '
num2str(stiffL_10(g)) ' ' num2str(armLM1_10(g)) ' '  num2str(armLM2_10(g)) ' '
num2str(strain) ' ' num2str(Stiff_scale)];
            end
            % Stiff springs for cross braces on the cross-linked face
            if(g > 20 && g < 25)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_10(g)) ' '
num2str(stiffL_10(g)) ' ' num2str(armLM1_10(g)) ' '  num2str(armLM2_10(g)) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
            end
```

```
            % Melting ends of the region
            if(g > 24)
                s = ['C ' num2str(seg_num) ' ' num2str(rodL_10(g)) ' '
num2str(stiffL_10(g)) ' ' num2str(armLM1_10(g)) ' '  num2str(armLM2_10(g)) ' '
num2str(2e-9) ' ' num2str(Stiff_scale)];
            end
            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end

        armCM1_10 = armCM1 + 120 + adjust;
        armCM2_10 = armCM2 + 120 + adjust;
        armCM3_10 = armCM3 + 120 + adjust;
        armCM4_10 = armCM4 + 120 + adjust;
        armCM5_10 = armCM5 + 120 + adjust;
        armCM6_10 = armCM6 + 120 + adjust;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1_10)
            if(g < 3)
                s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1_10(g)) ' '  num2str(armCM2_10(g)) ' '  '2.82842712474e-009' ' '
num2str(Stiff_scale)];
                Vert_seg{springcount,1} = s;
                springcount = springcount+1;
            end
        end;
        % Braces
        for g = 1:length(armCM1_10)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3_10(g)) ' '  num2str(armCM4_10(g)) ' ' num2str(sqrt(2e-9^2 + (3 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1_10)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5_10(g)) ' '  num2str(armCM6_10(g)) ' ' num2str(sqrt(2e-9^2 + (3 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;

        armCM1_10 = armCM1 + 132 + adjust;
        armCM2_10 = armCM2 + 132 + adjust;
        armCM3_10 = armCM3 + 128 + adjust;
        armCM4_10 = armCM4 + 128 + adjust;
        armCM5_10 = armCM5 + 128 + adjust;
        armCM6_10 = armCM6 + 128 + adjust;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1_10)
            if(g < 3)
                s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1_10(g)) ' '  num2str(armCM2_10(g)) ' '  '2.82842712474e-009' ' '
num2str(Stiff_scale)];
                Vert_seg{springcount,1} = s;
                springcount = springcount+1;
            end
        end;
        % Braces
        for g = 1:length(armCM1_10)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3_10(g)) ' '  num2str(armCM4_10(g)) ' ' num2str(sqrt(2e-9^2 + (6 * bp)^2)) '
' num2str(1)];
```

```
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1_10)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5_10(g)) ' '  num2str(armCM6_10(g)) ' ' num2str(sqrt(2e-9^2 + (6 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
    end

    if(mod(seg_num, arm_segment_offset) == 11)
    % new node connections for cross-linked springs
        armLM1_11 =  [36 38 40 42 32 33 36 37 40 41 44 45 36 37 38 39 36 38 37 39 36 37
40 41 32 34 44 46] + 104 + adjust;
        armLM2_11 =  [37 39 41 43 34 35 38 39 42 43 46 47 41 40 43 42 42 40 43 41 39 38
43 42 33 35 45 47] + 104 + adjust;
        rodL_11 =    [2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1];
        stiffL_11 =  [1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4];

        for g = 1:length(armLM1_10)
            if(g < 13)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_11(g)) ' '
num2str(stiffL_11(g)) ' ' num2str(armLM1_11(g)) ' '  num2str(armLM2_11(g)) ' '
num2str(sStrain) ' ' num2str(Stiff_scale)];
            end
            if(g > 12 && g < 21)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_11(g)) ' '
num2str(stiffL_11(g)) ' ' num2str(armLM1_11(g)) ' '  num2str(armLM2_11(g)) ' '
num2str(strain) ' ' num2str(Stiff_scale)];
            end
            % Stiff springs for cross braces on the cross-linked face
            if(g > 20 && g < 25)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_11(g)) ' '
num2str(stiffL_11(g)) ' ' num2str(armLM1_11(g)) ' '  num2str(armLM2_11(g)) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
            end
            % Melting ends of the region
            if(g > 24)
                s = ['C ' num2str(seg_num) ' ' num2str(rodL_11(g)) ' '
num2str(stiffL_11(g)) ' ' num2str(armLM1_11(g)) ' '  num2str(armLM2_11(g)) ' '
num2str(2e-9) ' ' num2str(Stiff_scale)];
            end
            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end

        armCM1_11 = armCM1 + 136 + adjust;
        armCM2_11 = armCM2 + 136 + adjust;
        armCM3_11 = armCM3 + 136 + adjust;
        armCM4_11 = armCM4 + 136 + adjust;
        armCM5_11 = armCM5 + 136 + adjust;
        armCM6_11 = armCM6 + 136 + adjust;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1_10)
            if(g < 3)
                s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1_11(g)) ' '  num2str(armCM2_11(g)) ' '  '2.82842712474e-009' ' '
num2str(Stiff_scale)];
                Vert_seg{springcount,1} = s;
                springcount = springcount+1;
            end
```

```
        end;
        % Braces
        for g = 1:length(armCM1_10)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3_11(g)) ' '  num2str(armCM4_11(g)) ' ' num2str(sqrt(2e-9^2 + (5 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1_10)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5_11(g)) ' '  num2str(armCM6_11(g)) ' ' num2str(sqrt(2e-9^2 + (5 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;

        armCM1_11 = armCM1 + 148 + adjust;
        armCM2_11 = armCM2 + 148 + adjust;
        armCM3_11 = armCM3 + 144 + adjust;
        armCM4_11 = armCM4 + 144 + adjust;
        armCM5_11 = armCM5 + 144 + adjust;
        armCM6_11 = armCM6 + 144 + adjust;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1_10)
            if(g < 3)
                s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1_11(g)) ' '  num2str(armCM2_11(g)) ' '  '2.82842712474e-009' ' '
num2str(Stiff_scale)];
                Vert_seg{springcount,1} = s;
                springcount = springcount+1;
            end
        end;
        % Braces
        for g = 1:length(armCM1_10)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3_11(g)) ' '  num2str(armCM4_11(g)) ' ' num2str(sqrt(2e-9^2 + (7 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1_10)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5_11(g)) ' '  num2str(armCM6_11(g)) ' ' num2str(sqrt(2e-9^2 + (7 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
    end

    if(mod(seg_num, arm_segment_offset) == 13)
        % new node connections for cross-linked springs
        armLM1_13 =  [52 54 56 58 60 62 64 66 48 49 52 53 56 57 60 61 64 65 68 69 52 53
54 55 60 61 62 63 52 53 54 55 60 61 62 63 52 53 56 57 60 61 64 65 48 50 68 70] + 112 +
adjust;
        armLM2_13 =  [53 55 57 59 61 63 65 67 50 51 54 55 58 59 62 63 66 67 70 71 57 56
59 58 65 64 67 66 58 59 56 57 66 67 64 65 55 54 59 58 63 62 67 66 49 51 69 71] + 112 +
adjust;
        rodL_13 =    [2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
2 1 2 1 2 1 2 1 2 1 2 1 2 1];
        stiffL_13 =  [1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
3 4 1 2 3 4 1 2 3 4 1 2 3 4];
```

```
        for g = 1:length(armLM1_13)
            % Normal links
            if(g < 21)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_13(g)) ' '
num2str(stiffL_13(g)) ' ' num2str(armLM1_13(g)) ' '  num2str(armLM2_13(g)) ' '
num2str(sStrain) ' ' num2str(Stiff_scale)];
            end
            % Stiff springs for cross braces on the spring face
            if(g > 20 && g < 37)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_13(g)) ' '
num2str(stiffL_13(g)) ' ' num2str(armLM1_13(g)) ' '  num2str(armLM2_13(g)) ' '
num2str(strain) ' ' num2str(Stiff_scale)];
            end
            % Stiff springs for cross braces on the cross-linked face
            if(g > 36 && g < 45)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_13(g)) ' '
num2str(stiffL_13(g)) ' ' num2str(armLM1_13(g)) ' '  num2str(armLM2_13(g)) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
            end
            % Melting ends of the region
            if(g > 44)
                s = ['C ' num2str(seg_num) ' ' num2str(rodL_13(g)) ' '
num2str(stiffL_13(g)) ' ' num2str(armLM1_13(g)) ' '  num2str(armLM2_13(g)) ' '
num2str(2e-9) ' ' num2str(Stiff_scale)];
            end
            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end
        % left segment
        armCM1_13 = armCM1 + 160 + adjust;
        armCM2_13 = armCM2 + 160 + adjust;
        armCM3_13 = armCM3 + 160 + adjust;
        armCM4_13 = armCM4 + 160 + adjust;
        armCM5_13 = armCM5 + 160 + adjust;
        armCM6_13 = armCM6 + 160 + adjust;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1_13)
            if(g < 3)
                s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1_13(g)) ' '  num2str(armCM2_13(g)) ' '  '2.82842712474e-009' ' '
num2str(Stiff_scale)];
                Vert_seg{springcount,1} = s;
                springcount = springcount+1;
            end
        end;
        % Braces
        for g = 1:length(armCM1_13)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3_13(g)) ' '  num2str(armCM4_13(g)) ' ' num2str(sqrt(2e-9^2 + (3 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1_13)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5_13(g)) ' '  num2str(armCM6_13(g)) ' ' num2str(sqrt(2e-9^2 + (3 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        % middle segment
        armCM1_13 = armCM1 + 168 + adjust;
        armCM2_13 = armCM2 + 168 + adjust;
```

```
        armCM3_13 = armCM3 + 168 + adjust;
        armCM4_13 = armCM4 + 168 + adjust;
        armCM5_13 = armCM5 + 168 + adjust;
        armCM6_13 = armCM6 + 168 + adjust;

        % Braces
        for g = 1:length(armCM1_13)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3_13(g)) ' '  num2str(armCM4_13(g)) ' ' num2str(sqrt(2e-9^2 + (3 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1_13)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5_13(g)) ' '  num2str(armCM6_13(g)) ' ' num2str(sqrt(2e-9^2 + (3 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        % right segment
        armCM1_13 = armCM1 + 180 + adjust;
        armCM2_13 = armCM2 + 180 + adjust;
        armCM3_13 = armCM3 + 176 + adjust;
        armCM4_13 = armCM4 + 176 + adjust;
        armCM5_13 = armCM5 + 176 + adjust;
        armCM6_13 = armCM6 + 176 + adjust;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1_13)
            if(g < 3)
                s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1_13(g)) ' '  num2str(armCM2_13(g)) ' '  '2.82842712474e-009' ' '
num2str(Stiff_scale)];
                Vert_seg{springcount,1} = s;
                springcount = springcount+1;
            end
        end;
        % Braces
        for g = 1:length(armCM1_13)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3_13(g)) ' '  num2str(armCM4_13(g)) ' ' num2str(sqrt(2e-9^2 + (2 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1_13)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5_13(g)) ' '  num2str(armCM6_13(g)) ' ' num2str(sqrt(2e-9^2 + (2 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
    end

    if(mod(seg_num, arm_segment_offset) == 0)
        % new node connections for cross-linked springs
        armLM1_16 =  [20 22 24 26  16 17 20 21 24 25  20 21 22 23  20 22 21 23  20 21 24
25  16 18 ] + 184 + adjust;
        armLM2_16 =  [21 23 25 27  18 19 22 23 26 27  25 24 27 26  26 24 27 25  23 22 27
26  17 19 ] + 184 + adjust;
        rodL_16 =    [2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
2 1 2 1 2 1 2 1 2 2 1];
```

```
        stiffL_16 =  [1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2
3 4 1 2 3 4 1 2 3 3 4];

        for g = 1:length(armLM1_16)
            % Normal stiff springs
            if(g < 11)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_16(g)) ' '
num2str(stiffL_16(g)) ' ' num2str(armLM1_16(g)) ' '  num2str(armLM2_16(g)) ' '
num2str(sStrain) ' ' num2str(Stiff_scale)];
            end
            % Cross springs across the cross-links
            if(g > 10 && g < 19)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_16(g)) ' '
num2str(stiffL_16(g)) ' ' num2str(armLM1_16(g)) ' '  num2str(armLM2_16(g)) ' '
num2str(strain) ' ' num2str(Stiff_scale)];
            end
            % Cross springs across the face
            if(g > 18 && g < 24)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_16(g)) ' '
num2str(stiffL_16(g)) ' ' num2str(armLM1_16(g)) ' '  num2str(armLM2_16(g)) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
            end
            % Melting springs
            if(g > 22)
                s = ['C ' num2str(seg_num) ' ' num2str(rodL_16(g)) ' '
num2str(stiffL_16(g)) ' ' num2str(armLM1_16(g)) ' '  num2str(armLM2_16(g)) ' '
num2str(2e-9) ' ' num2str(Stiff_scale)];
            end

            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end
        armCM1_16 = armCM1 + 196 + adjust;
        armCM2_16 = armCM2 + 196 + adjust;
        armCM3_16 = armCM3 + 200 + adjust;
        armCM4_16 = armCM4 + 200 + adjust;
        armCM5_16 = armCM5 + 200 + adjust;
        armCM6_16 = armCM6 + 200 + adjust;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1_16)
            if(g > 2)
                s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1_16(g)) ' '  num2str(armCM2_16(g)) ' '  '2.82842712474e-009' ' '
num2str(Stiff_scale)];
                Vert_seg{springcount,1} = s;
                springcount = springcount+1;
            end
        end;
        % Braces
        for g = 1:length(armCM1_16)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3_16(g)) ' '  num2str(armCM4_16(g)) ' ' num2str(sqrt(2e-9^2 + (7 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1_16)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5_16(g)) ' '  num2str(armCM6_16(g)) ' ' num2str(sqrt(2e-9^2 + (7 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
```

```
    end

end
Vert_seg


% Calculate the length of each segment
length1 = (bases_vert1(2) - bases_vert1(1)) * bp;
length2 = (bases_vert1(4) - bases_vert1(3)) * bp;
length3 = length1;
length4 = length2;
length5 = (bases_vert2(2) - bases_vert2(1)) * bp;
length6 = (bases_vert2(4) - bases_vert2(3)) * bp;
length7 = length5;
length8 = length6;

bases_vert1 = bases_vert1 + bases_vert1(4);
bases_vert2 = bases_vert2 + bases_vert2(4);

% Initialize the arrays
bases = [length1 length2 length3 length4 length5 length6 length7 length8] /bp;
length_vert = [length1 length2 length3 length4 length5 length6 length7 length8];
ext = avg_mass_per_bp * ssDNA_EC * bases;% bases * 9777;
ext2 = avg_mass_per_bp * dsDNA_EC * bases; %bases * 7984;
ext_c1 = avg_mass_per_bp * ssDNA_EC;
ext_c2 = avg_mass_per_bp * dsDNA_EC;

% Spring connections
springseg = [1 2 3 4];

rodL = [2 1 2 1];
rodR = [3 4 4 3];
stiffL = [1 2 3 4];
stiffR = [5 6 7 8];

braceL = [2 1 2 4];
braceR = [3 4 1 3];
braceLS = [2 1 7 3];
braceRS = [6 5 8 4];

armSM1 = [0 1 2 3];
armSM2 = [4 5 6 7];

% Face cross braces
armLM1 = [0 4 1 0];
armLM2 = [1 5 3 2];
armRM1 = [2 6 5 4];
armRM2 = [3 7 7 6];

% Side cross braces
armCM1 = [0 1 4 5];
armCM2 = [3 2 7 6];
armCM3 = [0 1 1 3];
armCM4 = [5 4 7 5];
armCM5 = [2 3 0 2];
armCM6 = [7 6 6 4];

%------------------------------
% Melting Temps and Cross Links
%------------------------------
% Four sets of Tm Theories:
%   1. 16 Thermo regions
```

```
%   2. Crossover theory links thermo regions
%   3. Lowest melting temperature on the shell
%   4. Average core melting temperatures with outer thermo regions
if(mod(position, 4) == 1)
    if(theory == 1)
        melt = [65 53 60 42 43 44 56 31];
    end
    if(theory == 2)
        melt = [59 58.3 51 58.3 62.3 43.5 62.3 43.5];
    end
    if(theory == 3)
        melt = [30 30 42 42 35 35 31 31];
    end
    if(theory == 4)
        melt = [61.7 58.3 59.2 58.3 62.3 53.2 62.3 46.7];
    end

%     if(mod(position, 8) == 1)
% %         melt = [66.7 58.3 66.7 58.3 62.3 37.5 62.3 37.5];
%         melt = [66.7 58.3 66.7 58.3 62.3 56.8 62.3 56.8];
%     end
%     if(mod(position, 8) == 5)
% %         melt = [66.7 58.3 66.7 58.3 62.3 56.8 62.3 56.8];
%         melt = [66.7 58.3 66.7 58.3 62.3 56.8 62.3 56.8];
%     end
end

if(mod(position, 4) == 2)
    if(theory == 1)
        melt = [44 43 31 56 53 65 42 60];
    end
    if(theory == 2)
        melt = [43.5 62.3 43.5 62.3 58.3 59 58.3 51];
    end
    if(theory == 3)
        melt = [31 31 35 35 42 42 30 30];
    end
    if(theory == 4)
        melt = [53.2 62.3 46.7 62.3 58.3 61.7 58.3 59.2];
    end
%     if(mod(position, 8) == 2)
% %         melt = [56.8 62.3 56.8 62.3 58.3 62.5 58.3 62.5];
%         melt = [56.8 62.3 56.8 62.3 58.3 66.7 58.3 66.7];
%     end
%     if(mod(position, 8) == 6)
% %         melt = [56.8 62.3 56.8 62.3 58.3 66.7 58.3 66.7];
%         melt = [56.8 62.3 56.8 62.3 58.3 66.7 58.3 66.7];
%     end
end

if(mod(position, 4) == 3)
    if(theory == 1)
        melt = [60 42 65 53 56 31 43 44];
    end
    if(theory == 2)
        melt = [51 58.3 59 58.3 62.3 43.5 62.3 43.5];
    end
    if(theory == 3)
        melt = [30 30 42 42 35 35 31 31];
    end
    if(theory == 4)
        melt = [59.2 58.3 61.7 58.3 62.3 46.7 62.3 53.2];
    end
```

```
%     if(mod(position, 8) == 3)
% %         melt = [66.7 58.3 66.7 58.3 62.3 56.8 62.3 56.8];
%         melt = [66.7 58.3 66.7 58.3 62.3 56.8 62.3 56.8];
%     end
%     if(mod(position, 8) == 7)
% %         melt = [62.5 58.3 62.5 58.3 62.3 56.8 62.3 56.8];
%         melt = [66.7 58.3 66.7 58.3 62.3 56.8 62.3 56.8];
%     end
end

if(mod(position, 4) == 0)
    if(theory == 1)
         melt = [31 56 44 43 42 60 53 65];
    end
    if(theory == 2)
        melt = [43.5 62.3 43.5 62.3 58.3 51 58.3 59];
    end
    if(theory == 3)
        melt = [35 35 31 31 30 30 42 42];
    end
    if(theory == 4)
        melt = [46.7 62.3 53.2 62.3 58.3 59.2 58.3 61.7];
    end

%     if(mod(position, 8) == 4)
% %         melt = [56.8 62.3 56.8 62.3 58.3 66.7 58.3 66.7];
%         melt = [56.8 62.3 56.8 62.3 58.3 66.7 58.3 66.7];
%     end
%     if(mod(position, 8) == 0)
% %         melt = [37.5 62.3 37.5 62.3 58.3 66.7 58.3 66.7];
%         melt = [56.8 62.3 56.8 62.3 58.3 66.7 58.3 66.7];
%     end
end

%------------------------------
% Counters and offset values
%------------------------------
springcount = 1;

sticky_inc = 4;
mass_seg_inc = 8;
arm_node_offset = 32;
arm_segment_offset = 16;
offset = 64;
tile_offset = 212;

%------------------------------
% Spring Segment Generator
%------------------------------
seg = 4;
count = 2;
counter = 8;

%--------------------------------------------------------------------------
% DNA springs
for(iter2 = 1:count)
    for(iter = 1:seg)
        seg_num = ((position-1) * arm_segment_offset ) + mass_seg_inc + iter + (seg*
(iter2-1));
        offset2 = mass_seg_inc*(iter-1) + arm_node_offset*(iter2-1) + offset +
((position-1) * tile_offset ) + 48;
        element = iter + (seg* (iter2-1));
```

```
        if(mod(seg_num, arm_segment_offset) == 9 || mod(seg_num, arm_segment_offset) ==
12 || mod(seg_num, arm_segment_offset) == 14 || mod(seg_num, arm_segment_offset) == 15)
            for g = 1:seg
                s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' ' num2str(0) ' '
num2str(armSM1(g)+ offset2) ' '  num2str(armSM2(g)+ offset2) ' '
num2str(length_vert(element)) ' ' num2str(melt(element)) ' ' num2str(bases(element)) ' '
num2str(ext(element)) ' ' num2str(ext2(element)) ' ' num2str(1) ];
                Vert_seg{springcount,1} = s;
                springcount = springcount + 1;
            end
        else

        if(mod(seg_num, arm_segment_offset) == 10)
            base_length = [3 2 6];
            for h = 1:3
                x_length = base_length(h) * bp;
                ext_1 = base_length(h) * ext_c1;
                ext_2 = base_length(h) * ext_c2;
                if(h > 1)
                    armSM1 = armSM1 + 4;
                    armSM2 = armSM2 + 4;
                end
                if(h == 2)
                    for g = 1:seg
                        s = ['X ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(h) ' ' num2str(armSM1(g) + offset2) ' '  num2str(armSM2(g) + offset2) ' '
num2str(x_length) ' ' num2str(0) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' '
num2str(ext_2) ' ' num2str(1) ];
                        Vert_seg{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                else
                    if(h == 1 || h == 3)
                        for g = 1:seg
                            s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armSM1(g) + offset2) ' '  num2str(armSM2(g) + offset2) ' '
num2str(x_length) ' ' num2str(melt(element)) ' ' num2str(base_length(h)) ' '
num2str(ext_1) ' ' num2str(ext_2) ' ' num2str(1) ];
                            Vert_seg{springcount,1} = s;
                            springcount = springcount + 1;
                        end
                    end
                end
            end
        end

        if(mod(seg_num, arm_segment_offset) == 11)
            base_length = [5 2 7];
            for h = 1:3
                x_length = base_length(h) * bp;
                ext_1 = base_length(h) * ext_c1;
                ext_2 = base_length(h) * ext_c2;
                if(h > 1)
                    armSM1 = armSM1 + 4;
                    armSM2 = armSM2 + 4;
                end
                if(h == 2)
                    for g = 1:seg
                        s = ['X ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(h) ' ' num2str(armSM1(g) + offset2) ' '  num2str(armSM2(g) + offset2) ' '
num2str(x_length) ' ' num2str(0) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' '
num2str(ext_2) ' ' num2str(1) ];
                        Vert_seg{springcount,1} = s;
```

```
                        springcount = springcount + 1;
                    end
                else
                    if(h == 1 || h == 3)
                        for g = 1:seg
                            s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armSM1(g) + offset2) ' '  num2str(armSM2(g) + offset2) ' '
num2str(x_length) ' ' num2str(melt(element)) ' ' num2str(base_length(h)) ' '
num2str(ext_1) ' ' num2str(ext_2) ' ' num2str(1) ];
                            Vert_seg{springcount,1} = s;
                            springcount = springcount + 1;
                        end
                    end
                end
            end
        end

        if(mod(seg_num, arm_segment_offset) == 13)
           base_length = [3 2 3 2 2];
           for h = 1:5
                x_length = base_length(h) * bp;
                ext_1 = base_length(h) * ext_c1;
                ext_2 = base_length(h) * ext_c2;
                if(h > 1)
                    armSM1 = armSM1 + 4;
                    armSM2 = armSM2 + 4;
                end
                if(h == 2 || h == 4)
                    for g = 1:seg
                        s = ['X ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(h) ' ' num2str(armSM1(g)+ offset2) ' '  num2str(armSM2(g)+ offset2) ' '
num2str(x_length) ' ' num2str(0) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' '
num2str(ext_2) ' ' num2str(1) ];
                        Vert_seg{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                else
                    for g = 1:seg
                        s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armSM1(g) + offset2) ' '  num2str(armSM2(g) + offset2) ' '
num2str(x_length) ' ' num2str(melt(element)) ' ' num2str(base_length(h)) ' '
num2str(ext_1) ' ' num2str(ext_2) ' ' num2str(1) ];
                        Vert_seg{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
            end
        end
             if(mod(seg_num, arm_segment_offset) == 0)
                base_length = [7 2];
                for h = 1:2
                    x_length = base_length(h) * bp;
                    ext_1 = base_length(h) * ext_c1;
                    ext_2 = base_length(h) * ext_c2;
                    if(h == 2)
                        armSM1 = armSM1 + 4;
                        armSM2 = armSM2 + 4;
                        for g = 1:seg
                            s = ['X ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(h) ' ' num2str(armSM1(g)+ offset2) ' '  num2str(armSM2(g)+ offset2) ' '
num2str(x_length) ' ' num2str(0) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' '
num2str(ext_2) ' ' num2str(1) ];
                            Vert_seg{springcount,1} = s;
```

```
                        springcount = springcount + 1;
                    end
                else
                    for g = 1:seg
                        s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armSM1(g) + offset2) ' '  num2str(armSM2(g) + offset2) ' '
num2str(x_length) ' ' num2str(melt(element)) ' ' num2str(base_length(h)) ' '
num2str(ext_1) ' ' num2str(ext_2) ' ' num2str(1) ];
                        Vert_seg{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
            end
        end
    end
end
%-------------------------------------------------------------------------

%-------------------------------------------------------------------------
% Structural springs
for(iter = 1:counter)
    % Stiff springs
    seg_num = ((position-1) * arm_segment_offset ) + mass_seg_inc + iter;
    offset2 = mass_seg_inc*(iter-1) + arm_node_offset*(iter2-1) + offset + ((position-1)
* tile_offset );
    if(mod(seg_num, arm_segment_offset) == 9)
        for g = 1:length(armLM1)
            if(ds_to_ss == 0)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL(g)) ' ' num2str(stiffL(g)) '
' num2str(armLM1(g) + offset2 + 16) ' '  num2str(armLM2(g) + offset2 + 16) ' '
num2str(2e-9) ' ' num2str(Stiff_scale)];
            end
            if(ds_to_ss == 1)
                if(g > 2)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL(g)) ' '
num2str(stiffL(g)) ' ' num2str(armLM1(g) + offset2 + 16) ' '  num2str(armLM2(g) + offset2
+ 16) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                else
                    s = ['C ' num2str(seg_num) ' ' num2str(rodL(g)) ' '
num2str(stiffL(g)) ' ' num2str(armLM1(g) + offset2 + 16) ' '  num2str(armLM2(g) + offset2
+ 16) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                end
            end
            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end
        for g = 1:length(armRM1)
            if(ds_to_ss == 0)
                s = ['T ' num2str(seg_num) ' ' num2str(rodR(g)) ' ' num2str(stiffR(g)) '
' num2str(armRM1(g)+ offset2 + 16) ' '  num2str(armRM2(g)+ offset2 + 16) ' ' num2str(2e-
9) ' ' num2str(Stiff_scale)];
            end
            if(ds_to_ss == 1)
                if(g > 2)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodR(g)) ' '
num2str(stiffR(g)) ' ' num2str(armRM1(g)+ offset2 + 16) ' '  num2str(armRM2(g)+ offset2 +
16) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                else
                    s = ['C ' num2str(seg_num) ' ' num2str(rodR(g)) ' '
num2str(stiffR(g)) ' ' num2str(armRM1(g)+ offset2 + 16) ' '  num2str(armRM2(g)+ offset2 +
16) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                end
```

```
            end
            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1)
            s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1(g)+ offset2 + 16) ' '  num2str(armCM2(g)+ offset2 + 16) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        % Braces
        for g = 1:length(armCM1)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3(g)+ offset2 + 16) ' '  num2str(armCM4(g)+ offset2 + 16) ' '
num2str(sqrt(2e-9^2 + length_vert(iter)^2)) ' ' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5(g)+ offset2 + 16) ' '  num2str(armCM6(g)+ offset2 + 16) ' '
num2str(sqrt(2e-9^2 + length_vert(iter)^2)) ' ' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
    end

    if(mod(seg_num, arm_segment_offset) == 12)
        for g = 1:length(armLM1)
            if(ds_to_ss == 0)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL(g)) ' ' num2str(stiffL(g)) '
' num2str(armLM1(g) + offset2 + 32) ' '  num2str(armLM2(g) + offset2 + 32) ' '
num2str(2e-9) ' ' num2str(Stiff_scale)];
            end
            if(ds_to_ss == 1)
                if(g > 2)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL(g)) ' '
num2str(stiffL(g)) ' ' num2str(armLM1(g) + offset2 + 32) ' '  num2str(armLM2(g) + offset2
+ 32) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                else
                    s = ['C ' num2str(seg_num) ' ' num2str(rodL(g)) ' '
num2str(stiffL(g)) ' ' num2str(armLM1(g) + offset2 + 32) ' '  num2str(armLM2(g) + offset2
+ 32) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                end
            end
            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end
        for g = 1:length(armRM1)
            if(ds_to_ss == 0)
                s = ['T ' num2str(seg_num) ' ' num2str(rodR(g)) ' ' num2str(stiffR(g)) '
' num2str(armRM1(g)+ offset2 + 32) ' '  num2str(armRM2(g)+ offset2 + 32) ' ' num2str(2e-
9) ' ' num2str(Stiff_scale)];
            end
            if(ds_to_ss == 1)
                if(g > 2)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodR(g)) ' '
num2str(stiffR(g)) ' ' num2str(armRM1(g)+ offset2 + 32) ' '  num2str(armRM2(g)+ offset2 +
32) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                else
```

```
                    s = ['C ' num2str(seg_num) ' ' num2str(rodR(g)) ' '
num2str(stiffR(g)) ' ' num2str(armRM1(g)+ offset2 + 32) ' '  num2str(armRM2(g)+ offset2 +
32) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                end
            end
            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1)
            s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1(g)+ offset2 + 32) ' '  num2str(armCM2(g)+ offset2 + 32) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        % Braces
        for g = 1:length(armCM1)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3(g)+ offset2 + 32) ' '  num2str(armCM4(g)+ offset2 + 32) ' '
num2str(sqrt(2e-9^2 + length_vert(iter)^2)) ' ' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5(g)+ offset2 + 32) ' '  num2str(armCM6(g)+ offset2 + 32) ' '
num2str(sqrt(2e-9^2 + length_vert(iter)^2)) ' ' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
    end

    if(mod(seg_num, arm_segment_offset) == 14 || mod(seg_num, arm_segment_offset) == 15)
        for g = 1:length(armLM1)
            if(ds_to_ss == 0)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL(g)) ' ' num2str(stiffL(g)) '
' num2str(armLM1(g) + offset2 + 48) ' '  num2str(armLM2(g) + offset2 + 48) ' '
num2str(2e-9) ' ' num2str(Stiff_scale)];
            end
            if(ds_to_ss == 1)
                if(g > 2)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL(g)) ' '
num2str(stiffL(g)) ' ' num2str(armLM1(g) + offset2 + 48) ' '  num2str(armLM2(g) + offset2
+ 48) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                else
                    s = ['C ' num2str(seg_num) ' ' num2str(rodL(g)) ' '
num2str(stiffL(g)) ' ' num2str(armLM1(g) + offset2 + 48) ' '  num2str(armLM2(g) + offset2
+ 48) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                end
            end
            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end
        for g = 1:length(armRM1)
            if(ds_to_ss == 0)
                s = ['T ' num2str(seg_num) ' ' num2str(rodR(g)) ' ' num2str(stiffR(g)) '
' num2str(armRM1(g)+ offset2 + 48) ' '  num2str(armRM2(g)+ offset2 + 48) ' ' num2str(2e-
9) ' ' num2str(Stiff_scale)];
            end
            if(ds_to_ss == 1)
                if(g > 2)
```

```
                    s = ['T ' num2str(seg_num) ' ' num2str(rodR(g)) ' '
num2str(stiffR(g)) ' ' num2str(armRM1(g)+ offset2 + 48) ' '  num2str(armRM2(g)+ offset2 +
48) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                else
                    s = ['C ' num2str(seg_num) ' ' num2str(rodR(g)) ' '
num2str(stiffR(g)) ' ' num2str(armRM1(g)+ offset2 + 48) ' '  num2str(armRM2(g)+ offset2 +
48) ' ' num2str(2e-9) ' ' num2str(Stiff_scale)];
                end
            end
            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1)
            s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1(g)+ offset2 + 48) ' '  num2str(armCM2(g)+ offset2 + 48) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        % Braces
        for g = 1:length(armCM1)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3(g)+ offset2 + 48) ' '  num2str(armCM4(g)+ offset2 + 48) ' '
num2str(sqrt(2e-9^2 + length_vert(iter)^2)) ' ' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5(g)+ offset2 + 48) ' '  num2str(armCM6(g)+ offset2 + 48) ' '
num2str(sqrt(2e-9^2 + length_vert(iter)^2)) ' ' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
    end
%--------------------------------------------------------------------------

%--------------------------------------------------------------------------
% Structural springs for cross linked regions
    adjust = ((position - 1) * tile_offset);
    if(mod(seg_num, arm_segment_offset) == 10)
    % new node connections for cross-linked springs
        armLM1_10 =  [36 38 40 42 32 33 36 37 40 41 44 45 36 37 38 39 36 38 37 39 36 37
40 41 32 34 44 46] + 88 + adjust;
        armLM2_10 =  [37 39 41 43 34 35 38 39 42 43 46 47 41 40 43 42 42 40 43 41 39 38
43 42 33 35 45 47] + 88 + adjust;
        rodL_10 =    [2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1];
        stiffL_10 =  [1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4];

        for g = 1:length(armLM1_10)
            if(g < 13)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_10(g)) ' '
num2str(stiffL_10(g)) ' ' num2str(armLM1_10(g)) ' '  num2str(armLM2_10(g)) ' '
num2str(sStrain) ' ' num2str(Stiff_scale)];
            end
            if(g > 12 && g < 21)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_10(g)) ' '
num2str(stiffL_10(g)) ' ' num2str(armLM1_10(g)) ' '  num2str(armLM2_10(g)) ' '
num2str(strain) ' ' num2str(Stiff_scale)];
            end
            % Stiff springs for cross braces on the cross-linked face
            if(g > 20 && g < 25)
```

```
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_10(g)) ' '
num2str(stiffL_10(g)) ' ' num2str(armLM1_10(g)) ' '  num2str(armLM2_10(g)) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
            end
            % Melting ends of the region
            if(g > 24)
                s = ['C ' num2str(seg_num) ' ' num2str(rodL_10(g)) ' '
num2str(stiffL_10(g)) ' ' num2str(armLM1_10(g)) ' '  num2str(armLM2_10(g)) ' '
num2str(2e-9) ' ' num2str(Stiff_scale)];
            end
            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end

        armCM1_10 = armCM1 + 120 + adjust;
        armCM2_10 = armCM2 + 120 + adjust;
        armCM3_10 = armCM3 + 120 + adjust;
        armCM4_10 = armCM4 + 120 + adjust;
        armCM5_10 = armCM5 + 120 + adjust;
        armCM6_10 = armCM6 + 120 + adjust;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1_10)
            if(g < 3)
                s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1_10(g)) ' '  num2str(armCM2_10(g)) ' '  '2.82842712474e-009' ' '
num2str(Stiff_scale)];
                Vert_seg{springcount,1} = s;
                springcount = springcount+1;
            end
        end;
        % Braces
        for g = 1:length(armCM1_10)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3_10(g)) ' '  num2str(armCM4_10(g)) ' ' num2str(sqrt(2e-9^2 + (3 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1_10)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5_10(g)) ' '  num2str(armCM6_10(g)) ' ' num2str(sqrt(2e-9^2 + (3 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;

        armCM1_10 = armCM1 + 132 + adjust;
        armCM2_10 = armCM2 + 132 + adjust;
        armCM3_10 = armCM3 + 128 + adjust;
        armCM4_10 = armCM4 + 128 + adjust;
        armCM5_10 = armCM5 + 128 + adjust;
        armCM6_10 = armCM6 + 128 + adjust;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1_10)
            if(g < 3)
                s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1_10(g)) ' '  num2str(armCM2_10(g)) ' '  '2.82842712474e-009' ' '
num2str(Stiff_scale)];
                Vert_seg{springcount,1} = s;
                springcount = springcount+1;
            end
        end;
        % Braces
```

```
        for g = 1:length(armCM1_10)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3_10(g)) ' '  num2str(armCM4_10(g)) ' ' num2str(sqrt(2e-9^2 + (6 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1_10)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5_10(g)) ' '  num2str(armCM6_10(g)) ' ' num2str(sqrt(2e-9^2 + (6 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
    end

    if(mod(seg_num, arm_segment_offset) == 11)
    % new node connections for cross-linked springs
        armLM1_11 =  [36 38 40 42 32 33 36 37 40 41 44 45 36 37 38 39 36 38 37 39 36 37
40 41 32 34 44 46] + 104 + adjust;
        armLM2_11 =  [37 39 41 43 34 35 38 39 42 43 46 47 41 40 43 42 42 40 43 41 39 38
43 42 33 35 45 47] + 104 + adjust;
        rodL_11 =    [2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1];
        stiffL_11 =  [1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4];

        for g = 1:length(armLM1_10)
            if(g < 13)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_11(g)) ' '
num2str(stiffL_11(g)) ' ' num2str(armLM1_11(g)) ' '  num2str(armLM2_11(g)) ' '
num2str(sStrain) ' ' num2str(Stiff_scale)];
            end
            if(g > 12 && g < 21)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_11(g)) ' '
num2str(stiffL_11(g)) ' ' num2str(armLM1_11(g)) ' '  num2str(armLM2_11(g)) ' '
num2str(strain) ' ' num2str(Stiff_scale)];
            end
            % Stiff springs for cross braces on the cross-linked face
            if(g > 20 && g < 25)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_11(g)) ' '
num2str(stiffL_11(g)) ' ' num2str(armLM1_11(g)) ' '  num2str(armLM2_11(g)) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
            end
            % Melting ends of the region
            if(g > 24)
                s = ['C ' num2str(seg_num) ' ' num2str(rodL_11(g)) ' '
num2str(stiffL_11(g)) ' ' num2str(armLM1_11(g)) ' '  num2str(armLM2_11(g)) ' '
num2str(2e-9) ' ' num2str(Stiff_scale)];
            end
            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end

        armCM1_11 = armCM1 + 136 + adjust;
        armCM2_11 = armCM2 + 136 + adjust;
        armCM3_11 = armCM3 + 136 + adjust;
        armCM4_11 = armCM4 + 136 + adjust;
        armCM5_11 = armCM5 + 136 + adjust;
        armCM6_11 = armCM6 + 136 + adjust;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1_10)
            if(g < 3)
```

```
                s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1_11(g)) ' '  num2str(armCM2_11(g)) ' '  '2.82842712474e-009' ' '
num2str(Stiff_scale)];
                Vert_seg{springcount,1} = s;
                springcount = springcount+1;
            end
        end;
        % Braces
        for g = 1:length(armCM1_10)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3_11(g)) ' '  num2str(armCM4_11(g)) ' ' num2str(sqrt(2e-9^2 + (5 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1_10)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5_11(g)) ' '  num2str(armCM6_11(g)) ' ' num2str(sqrt(2e-9^2 + (5 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;

        armCM1_11 = armCM1 + 148 + adjust;
        armCM2_11 = armCM2 + 148 + adjust;
        armCM3_11 = armCM3 + 144 + adjust;
        armCM4_11 = armCM4 + 144 + adjust;
        armCM5_11 = armCM5 + 144 + adjust;
        armCM6_11 = armCM6 + 144 + adjust;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1_10)
            if(g < 3)
                s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1_11(g)) ' '  num2str(armCM2_11(g)) ' '  '2.82842712474e-009' ' '
num2str(Stiff_scale)];
                Vert_seg{springcount,1} = s;
                springcount = springcount+1;
            end
        end;
        % Braces
        for g = 1:length(armCM1_10)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3_11(g)) ' '  num2str(armCM4_11(g)) ' ' num2str(sqrt(2e-9^2 + (7 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1_10)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5_11(g)) ' '  num2str(armCM6_11(g)) ' ' num2str(sqrt(2e-9^2 + (7 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
    end

    if(mod(seg_num, arm_segment_offset) == 13)
        % new node connections for cross-linked springs
        armLM1_13 =  [52 54 56 58 60 62 64 66 48 49 52 53 56 57 60 61 64 65 68 69 52 53
54 55 60 61 62 63 52 53 54 55 60 61 62 63 52 53 56 57 60 61 64 65 48 50 68 70] + 112 +
adjust;
```

```
        armLM2_13 =  [53 55 57 59 61 63 65 67 50 51 54 55 58 59 62 63 66 67 70 71 57 56
59 58 65 64 67 66 58 59 56 57 66 67 64 65 55 54 59 58 63 62 67 66 49 51 69 71] + 112 +
adjust;
        rodL_13 =    [2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
2 1 2 1 2 1 2 1 2 1 2 1 2 1];
        stiffL_13 =  [1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2
3 4 1 2 3 4 1 2 3 4 1 2 3 4];

        for g = 1:length(armLM1_13)
            % Normal links
            if(g < 21)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_13(g)) ' '
num2str(stiffL_13(g)) ' ' num2str(armLM1_13(g)) ' '  num2str(armLM2_13(g)) ' '
num2str(sStrain) ' ' num2str(Stiff_scale)];
            end
            % Stiff springs for cross braces on the spring face
            if(g > 20 && g < 37)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_13(g)) ' '
num2str(stiffL_13(g)) ' ' num2str(armLM1_13(g)) ' '  num2str(armLM2_13(g)) ' '
num2str(strain) ' ' num2str(Stiff_scale)];
            end
            % Stiff springs for cross braces on the cross-linked face
            if(g > 36 && g < 45)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_13(g)) ' '
num2str(stiffL_13(g)) ' ' num2str(armLM1_13(g)) ' '  num2str(armLM2_13(g)) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
            end
            % Melting ends of the region
            if(g > 44)
                s = ['C ' num2str(seg_num) ' ' num2str(rodL_13(g)) ' '
num2str(stiffL_13(g)) ' ' num2str(armLM1_13(g)) ' '  num2str(armLM2_13(g)) ' '
num2str(2e-9) ' ' num2str(Stiff_scale)];
            end
            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end
        % left segment
        armCM1_13 = armCM1 + 160 + adjust;
        armCM2_13 = armCM2 + 160 + adjust;
        armCM3_13 = armCM3 + 160 + adjust;
        armCM4_13 = armCM4 + 160 + adjust;
        armCM5_13 = armCM5 + 160 + adjust;
        armCM6_13 = armCM6 + 160 + adjust;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1_13)
            if(g < 3)
                s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1_13(g)) ' '  num2str(armCM2_13(g)) ' '  '2.82842712474e-009' ' '
num2str(Stiff_scale)];
                Vert_seg{springcount,1} = s;
                springcount = springcount+1;
            end
        end;
        % Braces
        for g = 1:length(armCM1_13)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3_13(g)) ' '  num2str(armCM4_13(g)) ' ' num2str(sqrt(2e-9^2 + (3 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1_13)
```

```
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5_13(g)) ' '  num2str(armCM6_13(g)) ' ' num2str(sqrt(2e-9^2 + (3 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        % middle segment
        armCM1_13 = armCM1 + 168 + adjust;
        armCM2_13 = armCM2 + 168 + adjust;
        armCM3_13 = armCM3 + 168 + adjust;
        armCM4_13 = armCM4 + 168 + adjust;
        armCM5_13 = armCM5 + 168 + adjust;
        armCM6_13 = armCM6 + 168 + adjust;

        % Braces
        for g = 1:length(armCM1_13)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3_13(g)) ' '  num2str(armCM4_13(g)) ' ' num2str(sqrt(2e-9^2 + (3 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1_13)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5_13(g)) ' '  num2str(armCM6_13(g)) ' ' num2str(sqrt(2e-9^2 + (3 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        % right segment
        armCM1_13 = armCM1 + 180 + adjust;
        armCM2_13 = armCM2 + 180 + adjust;
        armCM3_13 = armCM3 + 176 + adjust;
        armCM4_13 = armCM4 + 176 + adjust;
        armCM5_13 = armCM5 + 176 + adjust;
        armCM6_13 = armCM6 + 176 + adjust;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1_13)
            if(g < 3)
                s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1_13(g)) ' '  num2str(armCM2_13(g)) ' '  '2.82842712474e-009' ' '
num2str(Stiff_scale)];
                Vert_seg{springcount,1} = s;
                springcount = springcount+1;
            end
        end;
        % Braces
        for g = 1:length(armCM1_13)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3_13(g)) ' '  num2str(armCM4_13(g)) ' ' num2str(sqrt(2e-9^2 + (2 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
        for g = 1:length(armCM1_13)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5_13(g)) ' '  num2str(armCM6_13(g)) ' ' num2str(sqrt(2e-9^2 + (2 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
    end
```

```
    if(mod(seg_num, arm_segment_offset) == 0)
        % new node connections for cross-linked springs
        armLM1_16 =  [20 22 24 26  16 17 20 21 24 25  20 21 22 23  20 22 21 23  20 21 24
25  16 18 ] + 184 + adjust;
        armLM2_16 =  [21 23 25 27  18 19 22 23 26 27  25 24 27 26  26 24 27 25  23 22 27
26  17 19 ] + 184 + adjust;
        rodL_16 =    [2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
2 1 2 1 2 1 2 1 2 2 1];
        stiffL_16 =  [1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2
3 4 1 2 3 4 1 2 3 3 4];

        for g = 1:length(armLM1_16)
            % Normal stiff springs
            if(g < 11)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_16(g)) ' '
num2str(stiffL_16(g)) ' ' num2str(armLM1_16(g)) ' '  num2str(armLM2_16(g)) ' '
num2str(sStrain) ' ' num2str(Stiff_scale)];
            end
            % Cross springs across the cross-links
            if(g > 10 && g < 19)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_16(g)) ' '
num2str(stiffL_16(g)) ' ' num2str(armLM1_16(g)) ' '  num2str(armLM2_16(g)) ' '
num2str(strain) ' ' num2str(Stiff_scale)];
            end
            % Cross springs across the face
            if(g > 18 && g < 24)
                s = ['T ' num2str(seg_num) ' ' num2str(rodL_16(g)) ' '
num2str(stiffL_16(g)) ' ' num2str(armLM1_16(g)) ' '  num2str(armLM2_16(g)) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
            end
            % Melting springs
            if(g > 22)
                s = ['C ' num2str(seg_num) ' ' num2str(rodL_16(g)) ' '
num2str(stiffL_16(g)) ' ' num2str(armLM1_16(g)) ' '  num2str(armLM2_16(g)) ' '
num2str(2e-9) ' ' num2str(Stiff_scale)];
            end

            Vert_seg{springcount,1} = s;
            springcount = springcount + 1;
        end
        armCM1_16 = armCM1 + 196 + adjust;
        armCM2_16 = armCM2 + 196 + adjust;
        armCM3_16 = armCM3 + 200 + adjust;
        armCM4_16 = armCM4 + 200 + adjust;
        armCM5_16 = armCM5 + 200 + adjust;
        armCM6_16 = armCM6 + 200 + adjust;
        % Stiff springs on the face of the cube
        for g = 1:length(armCM1_16)
            if(g > 2)
                s = ['C ' num2str(seg_num) ' ' num2str(seg_num) ' ' num2str(0) ' '
num2str(armCM1_16(g)) ' '  num2str(armCM2_16(g)) ' '  '2.82842712474e-009' ' '
num2str(Stiff_scale)];
                Vert_seg{springcount,1} = s;
                springcount = springcount+1;
            end
        end;
        % Braces
        for g = 1:length(armCM1_16)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g)) ' '
num2str(armCM3_16(g)) ' '  num2str(armCM4_16(g)) ' ' num2str(sqrt(2e-9^2 + (7 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
```

```
        end;
        for g = 1:length(armCM1_16)
            s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g)) ' '
num2str(armCM5_16(g)) ' '  num2str(armCM6_16(g)) ' ' num2str(sqrt(2e-9^2 + (7 * bp)^2)) '
' num2str(1)];
            Vert_seg{springcount,1} = s;
            springcount = springcount+1;
        end;
    end

end
Vert_seg
```

Core model generator source code

```
% Core rods
CoreM1 = [12 14 32 34];
CoreM2 = [76 78 93 95];
CoreM3 = [29 31 49 51];
CoreM4 = [96 98 113 115];

CoreM5 = [13 15 33 35];
CoreM6 = [77 79 92 94];
CoreM7 = [28 30 48 50];
CoreM8 = [97 99 112 114];

CoreM9 = [77 79 92 94];
CoreM10 = [97 99 112 114];
CoreM11 = [76 78 93 95];
CoreM12 = [96 98 113 115];

CoreRB1 = [78 76 95 93];
CoreRB2 = [98 96 115 113];
CoreRB3 = [79 77 94 92];
CoreRB4 = [99 97 114 112];

ref1 = [2 2 4 4];
ref2 = [5 5 7 7];

node_offset = 128;
springcount = 1;
arm_segment_offset = 16;
mass_seg_inc = 32;

iter = 1;

% Cross spring links
% No curvature code (includes the interior springs)
offset = ((position-1) * node_offset );
seg_num1 = ref1(g)+((position-1) * arm_segment_offset );
seg_num2 = ref2(g)+((position-1) * arm_segment_offset );
for g = 1:length(CoreM1)
    s = ['C ' num2str(seg_num1) ' ' '0 0' ' ' num2str(CoreM1(g)+offset) ' '
num2str(CoreM9(g)+offset) ' ' '2.36457184285e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;
for g = 1:length(CoreM1)
```

```
    s = ['C ' num2str(seg_num2) ' ' '0 0' ' ' num2str(CoreM3(g)+offset) ' '
num2str(CoreM10(g)+offset) ' ' '2.36457184285e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;
for g = 1:length(CoreM1)
    s = ['C ' num2str(seg_num1) ' ' '0 0' ' ' num2str(CoreM5(g)+offset) ' '
num2str(CoreM11(g)+offset) ' ' '2.36457184285e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;
for g = 1:length(CoreM1)
    s = ['C ' num2str(seg_num2) ' ' '0 0' ' ' num2str(CoreM7(g)+offset) ' '
num2str(CoreM12(g)+offset) ' ' '2.36457184285e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;

for g = 1:length(CoreM1)
    s = ['C ' num2str(seg_num1) ' ' '0 0' ' ' num2str(CoreM1(g)+offset) ' '
num2str(CoreRB1(g)+offset) ' ' '2.13962613557e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;
for g = 1:length(CoreM1)
    s = ['C ' num2str(seg_num2) ' ' '0 0' ' ' num2str(CoreM3(g)+offset) ' '
num2str(CoreRB2(g)+offset) ' ' '2.13962613557e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;
for g = 1:length(CoreM1)
    s = ['C ' num2str(seg_num1) ' ' '0 0' ' ' num2str(CoreM5(g)+offset) ' '
num2str(CoreRB3(g)+offset) ' ' '4.08142132106e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;
for g = 1:length(CoreM1)
    s = ['C ' num2str(seg_num2) ' ' '0 0' ' ' num2str(CoreM7(g)+offset) ' '
num2str(CoreRB4(g)+offset) ' ' '4.08142132106e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;

for g = 1:length(CoreM1)
    s = ['R ' num2str(seg_num1) ' ' '0 0' ' ' num2str(CoreM1(g)+offset) ' '
num2str(CoreM2(g)+offset) ' ' '4.80832611206e-10' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;
for g = 1:length(CoreM1)
    s = ['R ' num2str(seg_num2) ' ' '0 0' ' ' num2str(CoreM3(g)+offset) ' '
num2str(CoreM4(g)+offset) ' ' '4.80832611206e-10' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;
for g = 1:length(CoreM1)
    s = ['R ' num2str(seg_num1) ' ' '0 0' ' ' num2str(CoreM5(g)+offset) ' '
num2str(CoreM6(g)+offset) ' ' '3.30925973595e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;
for g = 1:length(CoreM1)
```

```
    s = ['R ' num2str(seg_num2) ' ' '0 0' ' ' num2str(CoreM7(g)+offset) ' '
num2str(CoreM8(g)+offset) ' ' '3.30925973595e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;
Core
```

Core model generator for cross-linked models:

```
if(mod(position, 4) == 1)
    % Core DNA Springs - Shell Connections
    CoreM1 = [12 14 45 47];
    CoreM2 = [132 134 160 162];
    CoreM3 = [48 50 81 83];
    CoreM4 = [157 159 193 195];
end


if(mod(position, 4) == 2)
    % Core DNA Springs - Shell Connections
    CoreM1 = [28 30 45 47];
    CoreM2 = [140 142 164 166];
    CoreM3 = [48 50 81 83];
    CoreM4 = [161 163 189 191];
end


if(mod(position, 4) == 3)
    % Core DNA Springs - Shell Connections
    CoreM1 = [28 30 61 63];
    CoreM2 = [132 134 160 162];
    CoreM3 = [64 66 81 83];
    CoreM4 = [157 159 181 183];
end

if(mod(position, 4) == 0)
    % Core DNA Springs - Shell Connections
    CoreM1 = [28 30 61 63];
    CoreM2 = [128 130 164 166];
    CoreM3 = [64 66 97 99];
    CoreM4 = [161 163 189 191];
end

% Core DNA Springs - Poly T segments
CoreM5 = CoreM1 + [1 1 -1 -1];
CoreM6 = CoreM2 + [ 1 1 1 1 ];
CoreM7 = CoreM3 + [1 1 -1 -1];
CoreM8 = CoreM4 + [-1 -1 -1 -1];

% Melting springs - On the faces
CoreM9 = CoreM6;
CoreM10 = CoreM8;
CoreM11 = CoreM2;
CoreM12 = CoreM4;

% Melting springs - On the faces
CoreRB1 = CoreM6 + [ 1 -3 1 -3 ];
CoreRB2 = CoreM8 + [ 3 -1 3 -1 ];
% Melting springs - On the faces
CoreRB3 = CoreRB1 + [1 1 1 1];
```

```
CoreRB4 = CoreRB2 + [-1 -1 -1 -1];

ref1 = [2 2 4 4];
ref2 = [5 5 7 7];

node_offset = 212;
springcount = 1;
arm_segment_offset = 16;
mass_seg_inc = 32;

iter = 1;

% Cross spring links
% No curvature code (includes the interior springs)
offset = ((position-1) * node_offset );
seg_num1 = ref1(g)+((position-1) * arm_segment_offset );
seg_num2 = ref2(g)+((position-1) * arm_segment_offset );
% DNA spring connections for the core - shell connections
for g = 1:length(CoreM1)
    s = ['R ' num2str(seg_num1) ' ' '0 0' ' ' num2str(CoreM1(g)+offset) ' '
num2str(CoreM2(g)+offset) ' ' '4.80832611206e-10' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;
for g = 1:length(CoreM1)
    s = ['R ' num2str(seg_num2) ' ' '0 0' ' ' num2str(CoreM3(g)+offset) ' '
num2str(CoreM4(g)+offset) ' ' '4.80832611206e-10' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;
% DNA spring connections for the core - poly T's
for g = 1:length(CoreM1)
    s = ['R ' num2str(seg_num1) ' ' '0 0' ' ' num2str(CoreM5(g)+offset) ' '
num2str(CoreM6(g)+offset) ' ' '3.30925973595e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;
for g = 1:length(CoreM1)
    s = ['R ' num2str(seg_num2) ' ' '0 0' ' ' num2str(CoreM7(g)+offset) ' '
num2str(CoreM8(g)+offset) ' ' '3.30925973595e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;

% Melting springs - On the faces
for g = 1:length(CoreM1)
    s = ['C ' num2str(seg_num1) ' ' '0 0' ' ' num2str(CoreM1(g)+offset) ' '
num2str(CoreM9(g)+offset) ' ' '2.36457184285e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;
for g = 1:length(CoreM1)
    s = ['C ' num2str(seg_num2) ' ' '0 0' ' ' num2str(CoreM3(g)+offset) ' '
num2str(CoreM10(g)+offset) ' ' '2.36457184285e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;
for g = 1:length(CoreM1)
    s = ['C ' num2str(seg_num1) ' ' '0 0' ' ' num2str(CoreM5(g)+offset) ' '
num2str(CoreM11(g)+offset) ' ' '2.36457184285e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;
```

```
for g = 1:length(CoreM1)
    s = ['C ' num2str(seg_num2) ' ' '0 0' ' ' num2str(CoreM7(g)+offset) ' '
num2str(CoreM12(g)+offset) ' ' '2.36457184285e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;

% Melting springs - On the shell faces
for g = 1:length(CoreM1)
    s = ['C ' num2str(seg_num1) ' ' '0 0' ' ' num2str(CoreM1(g)+offset) ' '
num2str(CoreRB1(g)+offset) ' ' '2.13962613557e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;
for g = 1:length(CoreM1)
    s = ['C ' num2str(seg_num2) ' ' '0 0' ' ' num2str(CoreM3(g)+offset) ' '
num2str(CoreRB2(g)+offset) ' ' '2.13962613557e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;
% Melting springs - On the poly T faces
for g = 1:length(CoreM1)
    s = ['C ' num2str(seg_num1) ' ' '0 0' ' ' num2str(CoreM5(g)+offset) ' '
num2str(CoreRB3(g)+offset) ' ' '4.08142132106e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;
for g = 1:length(CoreM1)
    s = ['C ' num2str(seg_num2) ' ' '0 0' ' ' num2str(CoreM7(g)+offset) ' '
num2str(CoreRB4(g)+offset) ' ' '4.08142132106e-9' ' ' num2str(Stiff_scale)];
    Core{springcount,1} = s;
    springcount = springcount+1;
end;

Core
```

Crossover model generator source code:

```
% DNA Springs
CrossSM1 = [4 6 21 23];
CrossSM2 = [8 10 25 27];
CrossSM3 = [5 7 9 11];
CrossSM4 = [20 22 24 26];

% Crossover braces
CrossBM1 = [10 8 27 25];
CrossBM2 = [22 20 26 24];
CrossBM3 = [24 26 20 22];

CrossBM4 = [5 7 20 22];
CrossBM5 = [9 11 24 26];

CrossBM6 = [11 9 26 24];

ref1 = [1 1 3 3 ];
ref2 = [1 1 2 2 ];
ref3 = ref1 + 4;
ref4 = ref2 + 4;

seg = 4;
```

```
count = 4;
counter = 8;

sticky_inc = 4;
mass_seg_inc = 32;
arm_segment_offset = 16;
offset = 64;

node_offset = 128;
springcount = 1;
melt = 200;
basenum = 0.01;

for(iter = 1:count)
    if(iter == 1 || iter == 3)
        ref_a = ref1;
        ref_b = ref2;
    end
    if(iter == 2 || iter == 4)
        ref_a = ref3;
        ref_b = ref4;
    end
    % DNA Springs
    offset = mass_seg_inc*(iter-1)+((position-1) * node_offset );
    for g = 1:length(CrossSM1)
        s = ['R ' num2str(ref_a(g) +((position-1) * counter )) ' ' '0 0' ' '
num2str(CrossSM1(g) + offset) ' '  num2str(CrossSM2(g)+ offset) ' ' num2str(3.4e-10) ' '
num2str(Stiff_scale)];
        Xover{springcount,1} = s;
        springcount = springcount+1;
    end;
    for g = 1:length(CrossSM3)
        s = ['R ' num2str(ref_b(g) +((position-1) * counter )) ' ' '0 0' ' '
num2str(CrossSM3(g)+ offset) ' '  num2str(CrossSM4(g)+ offset) ' ' num2str(3.4e-10) ' '
num2str(Stiff_scale)];
        Xover{springcount,1} = s;
        springcount = springcount+1;
    end;

    % Crossover Braces
    % Bottom/Top crossovers
    for g = 1:length(CrossBM1)
        s = ['C ' num2str(ref_a(g) +((position-1) * counter )) ' ' '0 0' ' '
num2str(CrossSM1(g)+ offset) ' '  num2str(CrossBM1(g)+ offset) ' ' '2.02869416128e-009' '
' num2str(Stiff_scale)];
        Xover{springcount,1} = s;
        springcount = springcount+1;
    end;
    % Middle crossover - left/right side
    for g = 1:length(CrossBM3)
        s = ['C ' num2str(ref_b(g) +((position-1) * counter )) ' ' '0 0' ' '
num2str(CrossSM3(g)+ offset) ' '  num2str(CrossBM2(g)+ offset) ' ' '2.02869416128e-009' '
' num2str(Stiff_scale)];
        Xover{springcount,1} = s;
        springcount = springcount+1;
    end;
    % Middle crossover - front/back
    for g = 1:length(CrossBM3)
        s = ['C ' num2str(ref_a(g) +((position-1) * counter )) ' ' '0 0' ' '
num2str(CrossSM3(g)+ offset) ' '  num2str(CrossBM3(g)+ offset) ' ' '4.80832611206e-10' '
' num2str(Stiff_scale)];
        Xover{springcount,1} = s;
        springcount = springcount+1;
```

```
    end;
    % Middle crossover - lower/upper
    for g = 1:length(CrossBM3)
        s = ['C ' num2str(ref_a(g) +((position-1) * counter )) ' ' '0 0' ' '
num2str(CrossBM4(g)+ offset) ' '  num2str(CrossBM6(g)+ offset) ' ' '2.02869416128e-009' '
' num2str(Stiff_scale)];
        Xover{springcount,1} = s;
        springcount = springcount+1;
    end;
    % Middle connector - lower/upper
    for g = 1:length(CrossBM3)
        s = ['C ' num2str(ref_a(g) +((position-1) * counter )) ' ' '0 0' ' '
num2str(CrossBM4(g)+ offset) ' '  num2str(CrossBM5(g)+ offset) ' ' num2str(3.4e-10) ' '
num2str(Stiff_scale)];
        Xover{springcount,1} = s;
        springcount = springcount+1;
    end;
end
Xover
```

Crossover model generator source code for cross-linked models:

```
% DNA Springs
if(mod(position, 4) == 1)
    CrossSM1_A1 = [4 6 29 31];
    CrossSM2_A1 = CrossSM1_A1 + 4;
    CrossSM3_A1 = CrossSM1_A1 + [1 1 -20 -20];
    CrossSM4_A1 = CrossSM1_A1 + [24 24 3 3];

    CrossSM1_A2 = [68 70 85 87] - 32;
    CrossSM2_A2 = CrossSM1_A2 + 4;
    CrossSM3_A2 = CrossSM1_A2 + [1 1 -12 -12];
    CrossSM4_A2 = CrossSM1_A2 + [16 16 3 3];

    CrossSM1_A3 = [116 118 149 151] - 64;
    CrossSM2_A3 = CrossSM1_A3 + 4;
    CrossSM3_A3 = CrossSM1_A3 + [1 1 -28 -28];
    CrossSM4_A3 = CrossSM1_A3 + [32 32 3 3];

    CrossSM1_A4 = [180 182 197 199] - 96;
    CrossSM2_A4 = CrossSM1_A4 + 4;
    CrossSM3_A4 = CrossSM1_A4 + [1 1 -12 -12];
    CrossSM4_A4 = CrossSM1_A4 + [16 16 3 3];

    % Crossover braces
    CrossBM1_A1 = CrossSM1_A1 + [ 6 2 6 2 ];
    CrossBM2_A1 = CrossSM1_A1 + [ 26 22 5 1 ];
    CrossBM3_A1 = CrossSM1_A1 + [ 28 28 -1 -1 ];
    CrossBM4_A1 = CrossSM1_A1 + [ 1 1 -1 -1 ];
    CrossBM5_A1 = CrossSM1_A1 + [ 5 5 3 3 ];
    CrossBM6_A1 = CrossSM1_A1 + [ 7 3 5 1 ];

    CrossBM1_A2 = [74 72 91 89] - 32;
    CrossBM2_A2 = [86 84 90 88] - 32;
    CrossBM3_A2 = [88 90 84 86] - 32;
    CrossBM4_A2 = [69 71 84 86] - 32;
    CrossBM5_A2 = [73 75 88 90] - 32;
    CrossBM6_A2 = [75 73 90 88] - 32;
```

```
    CrossBM1_A3 = [122 120 155 153] - 64;
    CrossBM2_A3 = [150 148 154 152] - 64;
    CrossBM3_A3 = [152 154 148 150] - 64;
    CrossBM4_A3 = [117 119 150 148] - 64;
    CrossBM5_A3 = [121 123 154 152] - 64;
    CrossBM6_A3 = [123 121 152 154] - 64;

    CrossBM1_A4 = [186 184 203 201] - 96;
    CrossBM2_A4 = [198 196 202 200] - 96;
    CrossBM3_A4 = [200 202 196 198] - 96;
    CrossBM4_A4 = [181 183 198 196] - 96;
    CrossBM5_A4 = [185 187 202 200] - 96;
    CrossBM6_A4 = [187 185 200 202] - 96;
end

if(mod(position, 4) == 2)
    CrossSM1_A1 = [12 14 37 39];
    CrossSM2_A1 = CrossSM1_A1 + 4;
    CrossSM3_A1 = CrossSM1_A1 + [1 1 -20 -20];
    CrossSM4_A1 = CrossSM1_A1 + [24 24 3 3];

    CrossSM1_A2 = [52 54 101 103] - 32;
    CrossSM2_A2 = CrossSM1_A2 + 4;
    CrossSM3_A2 = CrossSM1_A2 + [1 1 -44 -44];
    CrossSM4_A2 = CrossSM1_A2 + [48 48 3 3];

    CrossSM1_A3 = [116 118 153 155] - 64;
    CrossSM2_A3 = CrossSM1_A3 + 4;
    CrossSM3_A3 = CrossSM1_A3 + [1 1 -32 -32];
    CrossSM4_A3 = CrossSM1_A3 + [36 36 3 3];

    CrossSM1_A4 = [176 178 193 195] - 96;
    CrossSM2_A4 = CrossSM1_A4 + 4;
    CrossSM3_A4 = CrossSM1_A4 + [1 1 -12 -12];
    CrossSM4_A4 = CrossSM1_A4 + [16 16 3 3];

    % Crossover braces
    CrossBM1_A1 = CrossSM1_A1 + [ 6 2 6 2 ];
    CrossBM2_A1 = CrossSM1_A1 + [ 26 22 5 1 ];
    CrossBM3_A1 = CrossSM1_A1 + [ 28 28 -1 -1 ];
    CrossBM4_A1 = CrossSM1_A1 + [ 1 1 -1 -1 ];
    CrossBM5_A1 = CrossSM1_A1 + [ 5 5 3 3 ];
    CrossBM6_A1 = CrossSM1_A1 + [ 7 3 5 1 ];

    CrossBM1_A2 = CrossSM1_A2 + [ 6 2 6 2 ];
    CrossBM2_A2 = CrossSM1_A2 + [ 50 46 5 1 ];
    CrossBM3_A2 = CrossSM1_A2 + [ 52 52 -1 -1 ];
    CrossBM4_A2 = CrossSM1_A2 + [ 1 1 -1 -1 ];
    CrossBM5_A2 = CrossSM1_A2 + [ 5 5 3 3 ];
    CrossBM6_A2 = CrossSM1_A2 + [ 7 3 5 1 ];

    CrossBM1_A3 = CrossSM1_A3 + [ 6 2 6 2 ];
    CrossBM2_A3 = CrossSM1_A3 + [ 38 34 5 1 ];
    CrossBM3_A3 = CrossSM1_A3 + [ 40 40 -1 -1 ];
    CrossBM4_A3 = CrossSM1_A3 + [ 1 1 -1 -1 ];
    CrossBM5_A3 = CrossSM1_A3 + [ 5 5 3 3 ];
    CrossBM6_A3 = CrossSM1_A3 + [ 7 3 5 1 ];

    CrossBM1_A4 = CrossSM1_A4 + [ 6 2 6 2 ];
    CrossBM2_A4 = CrossSM1_A4 + [ 18 14 5 1 ];
    CrossBM3_A4 = CrossSM1_A4 + [ 20 20 -1 -1 ];
    CrossBM4_A4 = CrossSM1_A4 + [ 1 1 -1 -1 ];
    CrossBM5_A4 = CrossSM1_A4 + [ 5 5 3 3 ];
```

```
    CrossBM6_A4 = CrossSM1_A4 + [ 7 3 5 1 ];
end

if(mod(position, 4) == 3)
    CrossSM1_A1 = [4 6 53 55];
    CrossSM2_A1 = CrossSM1_A1 + 4;
    CrossSM3_A1 = CrossSM1_A1 + [1 1 -44 -44];
    CrossSM4_A1 = CrossSM1_A1 + [48 48 3 3];

    CrossSM1_A2 = [68 70 93 95] - 32;
    CrossSM2_A2 = CrossSM1_A2 + 4;
    CrossSM3_A2 = CrossSM1_A2 + [1 1 -20 -20];
    CrossSM4_A2 = CrossSM1_A2 + [24 24 3 3];

    CrossSM1_A3 = [124 126 141 143] - 64;
    CrossSM2_A3 = CrossSM1_A3 + 4;
    CrossSM3_A3 = CrossSM1_A3 + [1 1 -12 -12];
    CrossSM4_A3 = CrossSM1_A3 + [16 16 3 3];

    CrossSM1_A4 = [164 166 201 203] - 96;
    CrossSM2_A4 = CrossSM1_A4 + 4;
    CrossSM3_A4 = CrossSM1_A4 + [1 1 -32 -32];
    CrossSM4_A4 = CrossSM1_A4 + [36 36 3 3];

    % Crossover braces
    CrossBM1_A1 = CrossSM1_A1 + [ 6 2 6 2 ];
    CrossBM2_A1 = CrossSM1_A1 + [ 50 46 5 1 ];
    CrossBM3_A1 = CrossSM1_A1 + [ 52 52 -1 -1 ];
    CrossBM4_A1 = CrossSM1_A1 + [ 1 1 -1 -1 ];
    CrossBM5_A1 = CrossSM1_A1 + [ 5 5 3 3 ];
    CrossBM6_A1 = CrossSM1_A1 + [ 7 3 5 1 ];

    CrossBM1_A2 = CrossSM1_A2 + [ 6 2 6 2 ];
    CrossBM2_A2 = CrossSM1_A2 + [ 26 22 5 1 ];
    CrossBM3_A2 = CrossSM1_A2 + [ 28 28 -1 -1 ];
    CrossBM4_A2 = CrossSM1_A2 + [ 1 1 -1 -1 ];
    CrossBM5_A2 = CrossSM1_A2 + [ 5 5 3 3 ];
    CrossBM6_A2 = CrossSM1_A2 + [ 7 3 5 1 ];

    CrossBM1_A3 = CrossSM1_A3 + [ 6 2 6 2 ];
    CrossBM2_A3 = CrossSM1_A3 + [ 18 14 5 1 ];
    CrossBM3_A3 = CrossSM1_A3 + [ 20 20 -1 -1 ];
    CrossBM4_A3 = CrossSM1_A3 + [ 1 1 -1 -1 ];
    CrossBM5_A3 = CrossSM1_A3 + [ 5 5 3 3 ];
    CrossBM6_A3 = CrossSM1_A3 + [ 7 3 5 1 ];

    CrossBM1_A4 = CrossSM1_A4 + [ 6 2 6 2 ];
    CrossBM2_A4 = CrossSM1_A4 + [ 38 34 5 1 ];
    CrossBM3_A4 = CrossSM1_A4 + [ 40 40 -1 -1 ];
    CrossBM4_A4 = CrossSM1_A4 + [ 1 1 -1 -1 ];
    CrossBM5_A4 = CrossSM1_A4 + [ 5 5 3 3 ];
    CrossBM6_A4 = CrossSM1_A4 + [ 7 3 5 1 ];
end
if(mod(position, 4) == 0)
    CrossSM1_A1 = [20 22 37 39];
    CrossSM2_A1 = CrossSM1_A1 + 4;
    CrossSM3_A1 = CrossSM1_A1 + [1 1 -12 -12];
    CrossSM4_A1 = CrossSM1_A1 + [16 16 3 3];

    CrossSM1_A2 = [76 78 101 103] - 32;
    CrossSM2_A2 = CrossSM1_A2 + 4;
    CrossSM3_A2 = CrossSM1_A2 + [1 1 -20 -20];
    CrossSM4_A2 = CrossSM1_A2 + [24 24 3 3];
```

```
    CrossSM1_A3 = [120 122 137 139] - 64;
    CrossSM2_A3 = CrossSM1_A3 + 4;
    CrossSM3_A3 = CrossSM1_A3 + [1 1 -12 -12];
    CrossSM4_A3 = CrossSM1_A3 + [16 16 3 3];

    CrossSM1_A4 = [168 170 201 203] - 96;
    CrossSM2_A4 = CrossSM1_A4 + 4;
    CrossSM3_A4 = CrossSM1_A4 + [1 1 -28 -28];
    CrossSM4_A4 = CrossSM1_A4 + [32 32 3 3];

    % Crossover braces
    CrossBM1_A1 = CrossSM1_A1 + [ 6 2 6 2 ];
    CrossBM2_A1 = CrossSM1_A1 + [ 18 14 5 1 ];
    CrossBM3_A1 = CrossSM1_A1 + [ 20 20 -1 -1 ];
    CrossBM4_A1 = CrossSM1_A1 + [ 1 1 -1 -1 ];
    CrossBM5_A1 = CrossSM1_A1 + [ 5 5 3 3 ];
    CrossBM6_A1 = CrossSM1_A1 + [ 7 3 5 1 ];

    CrossBM1_A2 = CrossSM1_A2 + [ 6 2 6 2 ];
    CrossBM2_A2 = CrossSM1_A2 + [ 26 22 5 1 ];
    CrossBM3_A2 = CrossSM1_A2 + [ 28 28 -1 -1 ];
    CrossBM4_A2 = CrossSM1_A2 + [ 1 1 -1 -1 ];
    CrossBM5_A2 = CrossSM1_A2 + [ 5 5 3 3 ];
    CrossBM6_A2 = CrossSM1_A2 + [ 7 3 5 1 ];

    CrossBM1_A3 = CrossSM1_A3 + [ 6 2 6 2 ];
    CrossBM2_A3 = CrossSM1_A3 + [ 18 14 5 1 ];
    CrossBM3_A3 = CrossSM1_A3 + [ 20 20 -1 -1 ];
    CrossBM4_A3 = CrossSM1_A3 + [ 1 1 -1 -1 ];
    CrossBM5_A3 = CrossSM1_A3 + [ 5 5 3 3 ];
    CrossBM6_A3 = CrossSM1_A3 + [ 7 3 5 1 ];

    CrossBM1_A4 = CrossSM1_A4 + [ 6 2 6 2 ];
    CrossBM2_A4 = CrossSM1_A4 + [ 34 30 5 1 ];
    CrossBM3_A4 = CrossSM1_A4 + [ 36 36 -1 -1 ];
    CrossBM4_A4 = CrossSM1_A4 + [ 1 1 -1 -1 ];
    CrossBM5_A4 = CrossSM1_A4 + [ 5 5 3 3 ];
    CrossBM6_A4 = CrossSM1_A4 + [ 7 3 5 1 ];
end
% Ref numbers

ref1 = [1 1 3 3 ];
ref2 = [1 1 2 2 ];
ref3 = ref1 + 4;
ref4 = ref2 + 4;

seg = 4;
count = 4;
counter = 8;

sticky_inc = 4;
mass_seg_inc = 32;
arm_segment_offset = 16;
offset = 64;

node_offset = 212;
springcount = 1;
melt = 200;

for(iter = 1:count)
    if(iter == 1 || iter == 3)
        ref_a = ref1;
```

```
    ref_b = ref2;
end
if(iter == 2 || iter == 4)
    ref_a = ref3;
    ref_b = ref4;
end
if(iter == 1)
    CrossSM1 = CrossSM1_A1;
    CrossSM2 = CrossSM2_A1;
    CrossSM3 = CrossSM3_A1;
    CrossSM4 = CrossSM4_A1;

    % Crossover braces
    CrossBM1 = CrossBM1_A1;
    CrossBM2 = CrossBM2_A1;
    CrossBM3 = CrossBM3_A1;
    CrossBM4 = CrossBM4_A1;
    CrossBM5 = CrossBM5_A1;
    CrossBM6 = CrossBM6_A1;
end
if(iter == 2)
    CrossSM1 = CrossSM1_A2;
    CrossSM2 = CrossSM2_A2;
    CrossSM3 = CrossSM3_A2;
    CrossSM4 = CrossSM4_A2;

    % Crossover braces
    CrossBM1 = CrossBM1_A2;
    CrossBM2 = CrossBM2_A2;
    CrossBM3 = CrossBM3_A2;
    CrossBM4 = CrossBM4_A2;
    CrossBM5 = CrossBM5_A2;
    CrossBM6 = CrossBM6_A2;
end
if(iter == 3)
    CrossSM1 = CrossSM1_A3;
    CrossSM2 = CrossSM2_A3;
    CrossSM3 = CrossSM3_A3;
    CrossSM4 = CrossSM4_A3;

    % Crossover braces
    CrossBM1 = CrossBM1_A3;
    CrossBM2 = CrossBM2_A3;
    CrossBM3 = CrossBM3_A3;
    CrossBM4 = CrossBM4_A3;
    CrossBM5 = CrossBM5_A3;
    CrossBM6 = CrossBM6_A3;
end
if(iter == 4)
    CrossSM1 = CrossSM1_A4;
    CrossSM2 = CrossSM2_A4;
    CrossSM3 = CrossSM3_A4;
    CrossSM4 = CrossSM4_A4;

    % Crossover braces
    CrossBM1 = CrossBM1_A4;
    CrossBM2 = CrossBM2_A4;
    CrossBM3 = CrossBM3_A4;
    CrossBM4 = CrossBM4_A4;
    CrossBM5 = CrossBM5_A4;
    CrossBM6 = CrossBM6_A4;
end
```

```
    % DNA Springs
    offset = mass_seg_inc*(iter-1)+((position-1) * node_offset );
    % Outer (Top/Bottom or Left/Right) DNA springs
    for g = 1:length(CrossSM1)
        s = ['R ' num2str(ref_a(g) +((position-1) * counter )) ' ' '0 0' ' '
num2str(CrossSM1(g) + offset) ' '  num2str(CrossSM2(g)+ offset) ' ' num2str(3.4e-10) ' '
num2str(Stiff_scale)];
        Xover{springcount,1} = s;
        springcount = springcount+1;
    end;
    % Inner (Top/Bottom or Left/Right) DNA springs
    for g = 1:length(CrossSM3)
        s = ['R ' num2str(ref_b(g) +((position-1) * counter )) ' ' '0 0' ' '
num2str(CrossSM3(g)+ offset) ' '  num2str(CrossSM4(g)+ offset) ' ' num2str(3.4e-10) ' '
num2str(Stiff_scale)];
        Xover{springcount,1} = s;
        springcount = springcount+1;
    end;

    % Crossover Braces
    % Bottom/Top crossovers
    for g = 1:length(CrossBM1)
        s = ['C ' num2str(ref_a(g) +((position-1) * counter )) ' ' '0 0' ' '
num2str(CrossSM1(g)+ offset) ' '  num2str(CrossBM1(g)+ offset) ' ' '2.02869416128e-009' '
' num2str(Stiff_scale)];
        Xover{springcount,1} = s;
        springcount = springcount+1;
    end;
    % Middle crossover - left/right side
    for g = 1:length(CrossBM3)
        s = ['C ' num2str(ref_b(g) +((position-1) * counter )) ' ' '0 0' ' '
num2str(CrossSM3(g)+ offset) ' '  num2str(CrossBM2(g)+ offset) ' ' '2.02869416128e-009' '
' num2str(Stiff_scale)];
        Xover{springcount,1} = s;
        springcount = springcount+1;
    end;
    % Middle crossover - front/back
    for g = 1:length(CrossBM3)
        s = ['C ' num2str(ref_a(g) +((position-1) * counter )) ' ' '0 0' ' '
num2str(CrossSM3(g)+ offset) ' '  num2str(CrossBM3(g)+ offset) ' ' '4.80832611206e-10' '
' num2str(Stiff_scale)];
        Xover{springcount,1} = s;
        springcount = springcount+1;
    end;
    % Middle crossover - lower/upper
    for g = 1:length(CrossBM3)
        s = ['C ' num2str(ref_a(g) +((position-1) * counter )) ' ' '0 0' ' '
num2str(CrossBM4(g)+ offset) ' '  num2str(CrossBM6(g)+ offset) ' '  '2.02869416128e-009'
' ' num2str(Stiff_scale)];
        Xover{springcount,1} = s;
        springcount = springcount+1;
    end;
    % Middle connector - lower/upper
    for g = 1:length(CrossBM3)
        s = ['C ' num2str(ref_a(g) +((position-1) * counter )) ' ' '0 0' ' '
num2str(CrossBM4(g)+ offset) ' '  num2str(CrossBM5(g)+ offset) ' ' num2str(3.4e-10) ' '
num2str(Stiff_scale)];
        Xover{springcount,1} = s;
        springcount = springcount+1;
    end;
end
Xover
```

Source code for sticky end mass node generator for cross-linked models:

```
% Written by: Vincent Mao
% last modified: 12.16.09
%
% This program generates the mass nodes for the cross-linked sticky ends.
% The sequences are generated from left to right according to the A Tile
% layout assuming a square core with the same number of base pairs for
% both the upper and lower dsDNA helices that compose each of the arms. The
% meshfile format is:
%       (M) <Node mass> <X coord> <Y coord> <Z coord> <Fixed mass flag>

clear;
clc;
close all;

% Initialize coordinates
coord2 =   [0    2e-9    0    2e-9];
coord3 =   [0   0   -2e-9    -2e-9];

% Position counters for the meshfile
masscount = 1;

%------------------------------
% Mass Node Generator
%------------------------------

% initial_y = initial_y + -2.134e-8 * 2;
bp = 3.4e-10;
bp_per_tile = 164 * 8 + 50;
total_mass = (bp_per_tile * 0.66) / 6.022e23;
total_mass_nodes = 212 * 8 + 60;
mass_value_per_node = total_mass / total_mass_nodes;
regionMax = 2;

for(horiSticky = 1:4)
    % Horizontal Mass Generator
    initial_y = -21.34e-9 * (horiSticky - 1);
    if(horiSticky == 1 || horiSticky == 3)
        initial_x = 13.18e-9;
    end
    if(horiSticky == 2 || horiSticky == 4)
        initial_x = 11.14e-9 + bp;
    end
    if(horiSticky > 1)
        regionMax = 1;
    end
    for(region = 1:regionMax)
        if(horiSticky > 1)
            increment = 2.34e-9;
        else
            increment = 2.34e-9 * (region-1);
        end
        for(segment = 1:2)
            % Generate the face nodes on the segment
            %         1 o
            %     3 o   |
            %       |   |
```

```
            %       | 0 o
            %     2 o
            for (massPos = 1:4)
                coord_x = initial_x + bp * segment;
                coord_y = initial_y + coord2(massPos) + increment;
                coord_z = coord3(massPos);

                m = ['M ' num2str(mass_value_per_node) ' ' num2str(coord_x) ' '
num2str(coord_y) ' ' num2str(coord_z) ' ' num2str(0) ];
                Sticky_mass_n{masscount,1} = m;
                masscount = masscount + 1;
            end
        end
    end
end


for(region = 1:3)
    segmentMax = 2;
    if(region == 1)
        initial_y = -29.84e-9;
        initial_x = 2e-9 + 2 * bp;
        increment = (2.34e-9 *(region-1));
    end
    if(region == 2)
        segmentMax = 1;
        initial_y = -28.14e-9 - (3 * bp);
        initial_x = 21.68e-9 - 4.68e-9;
        increment = (2.34e-9 * region);
    end
    if(region == 3)
        initial_y = -28.14e-9;
        initial_x = 24.02e-9 - 4.68e-9;
        increment = (2.34e-9 *(region-1));
    end
    for(segment = 1:segmentMax)

        % Generate the face nodes on the segment
        for massPos = 1:4
            coord_x = initial_x + coord2(massPos)+ increment;
            coord_y = initial_y - bp * segment;
            coord_z = coord3(massPos);

            m = ['M ' num2str(mass_value_per_node) ' ' num2str(coord_x) ' '
num2str(coord_y) ' ' num2str(coord_z) ' ' num2str(0) ];
            Sticky_mass_n{masscount,1} = m;
            masscount = masscount + 1;
        end
    end
end

Sticky_mass_n
```

Source code for sticky end segment generator source code:

```
clear;
clc;
close all;
```

```
avg_mass_per_bp = 0.66;     % kg/mole
avg_mass_per_base = 0.33;   % kg/mole
dsDNA_EC = 24156;
ssDNA_EC = 29181;
ext = avg_mass_per_bp * ssDNA_EC;% 9777;
ext2 = avg_mass_per_bp * dsDNA_EC; % 7984;

seg = 4;
count = 2;
counter = 8;

sticky_inc = 4;
mass_seg_inc = 16;
arm_node_offset = 32;
sticky_offset = 256;

springcount = 1;

%------------------------
% Change these parameters
tiles = 8;
DNA_scale = [1 1];
Stiff_scale = 1;
bases = 5;
theory = 3;
% 1. As given in the map.
% 2. Summed.
% 3. Avg the neighboring thermo regions with sum.
% 4. Avg the neighboring thermo regions
x_link = 0;
%------------------------

if(tiles == 2)
    count1 = 1;
    count2 = 1;
end;

if(tiles == 4)
    count1 = 2;
    count2 = 2;
end;

if(tiles == 8)
    count1 = 2;
    count2 = 6;
end;

springnum = 800;
% Spring connections
springseg = [1 2 3 4];

rodL = [2 1 2 1];
rodR = [3 4 4 3];
stiffL = [1 2 3 4];
stiffR = [5 6 7 8];

braceL = [2 1 2 4];
braceR = [3 4 1 3];
braceLS = [2 1 7 3];
braceRS = [6 5 8 4];

for(iter2 = 1:count1)
```

```
if(tiles == 8 && iter2 == 1)
    count2 = 4;
end
if(tiles == 8 && iter2 == 2)
   count2 = 6;
end
for(iter = 1:count2)
    % Set nodes depending on tiles being connected:
    if(iter2 == 1) % Between horizontal tiles
        % Sticky end connections

        armStM1 = [44 45 46 47];
        armStM2 = [128 129 130 131];

        armStM3 = [60 61 62 63];
        armStM4 = [144 145 146 147];

        % Sticky end cross braces
        armStBM1 = [129 128 131 130];
        armStBM2 = [130 131 128 129];

        armStBM3 = [145 144 147 146];
        armStBM4 = [146 147 144 145];
        if(iter == 1)
            if(theory == 1)
                melt = [13 13];
            end
            if(theory == 2)
                melt = [26 26];
            end
            if(theory == 3)
                melt = [45 42.5];
            end
            if(theory == 4)
                melt = [54.6 50.8];
            end
        end;
        if(iter == 2)
            if(theory == 1)
                melt = [9 10];
            end
            if(theory == 2)
                melt = [19 19];
            end
            if(theory == 3)
                melt = [40.2 42.7];
            end
            if(theory == 4)
                melt = [50.8 54.6];
            end
        end;
        if(iter == 3)
            if(theory == 1)
                melt = [12 12];
            end
            if(theory == 2)
                melt = [24 24];
            end
            if(theory == 3)
                melt = [44.4 41.9];
            end
            if(theory == 4)
                melt = [54.6 50.8];
```

```
        end
    end;
    if(iter == 4)
        if(theory == 1)
            melt = [10 10];
        end
        if(theory == 2)
            melt = [20 20];
        end
        if(theory == 3)
            melt = [40.5 43];
        end
        if(theory == 4)
            melt = [50.8 54.6];
        end
    end;
end;

if(iter2 == 2) % Between vertical tiles

    % Sticky end connections
    armStM1 = [64 65 66 67];
    armStM2 = [364 365 366 367];

    armStM3 = [80 81 82 83];
    armStM4 = [380 381 382 383];

    % Sticky end cross braces
    armStBM1 = [365 364 367 366];
    armStBM2 = [366 367 364 365];

    armStBM3 = [381 380 383 382];
    armStBM4 = [382 383 380 381];

    if(iter == 1)
        if(theory == 1)
            melt = [18 23];
        end
        if(theory == 2)
            melt = [41 41];
        end
        if(theory == 3)
            melt = [45.9 49.8];
        end
        if(theory == 4)
            melt = [56.2 54.2];
        end
    end;
    if(iter == 2)
        if(theory == 1)
            melt = [16 18];
        end
        if(theory == 2)
            melt = [34 34];
        end
        if(theory == 3)
            melt = [47.5 48.8];
        end
        if(theory == 4)
            melt = [54.2 56.2];
        end
        sticky_offset = 128;
    end;
```

```
    if(iter == 3)
        if(theory == 1)
            melt = [11 11];
        end
        if(theory == 2)
            melt = [22 22];
        end
        if(theory == 3)
            melt = [43.5 44.8];
        end
        if(theory == 4)
            melt = [56.2 54.2];
        end
        sticky_offset = 128;
    end;
    if(iter == 4)
        if(theory == 1)
            melt = [9 9];
        end
        if(theory == 2)
            melt = [18 18];
        end
        if(theory == 3)
            melt = [43.5 42.1];
        end
        if(theory == 4)
            melt = [54.2 56.2];
        end
        sticky_offset = 128;
    end;
    if(iter == 5)
        if(theory == 1)
            melt = [16 16];
        end
        if(theory == 2)
            melt = [32 32];
        end
        if(theory == 3)
            melt = [48.8 46.8];
        end
        if(theory == 4)
            melt = [56.2 54.2];
        end
        sticky_offset = 128;
    end;
    if(iter == 6)
        if(theory == 1)
            melt = [15 15];
        end
        if(theory == 2)
            melt = [30 30];
        end
        if(theory == 3)
            melt = [46.1 47.5];
        end
        if(theory == 4)
            melt = [54.2 56.2];
        end
        sticky_offset = 128;
    end;
end;

% Sticky end links
```

```
        seg_num = springnum + iter + 4 * (iter2 - 1);
        for g = 1:length(armStM1)
            s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' 0 '
num2str(armStM1(g)+ sticky_offset*(iter-1)) ' '  num2str(armStM2(g)+ sticky_offset*(iter-
1)) ' ' num2str(1.7e-9) ' ' num2str(melt(1)) ' ' num2str(bases) ' ' num2str(ext * bases)
' ' num2str(ext2 * bases) ' ' num2str(DNA_scale(1))];
            l{springcount,1} = s;
            springcount = springcount+1;
        end;

        for g = 1:length(armStM1)
            s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' 0 '
num2str(armStM3(g)+ sticky_offset*(iter-1)) ' '  num2str(armStM4(g)+ sticky_offset*(iter-
1)) ' ' num2str(1.7e-9) ' ' num2str(melt(2)) ' ' num2str(bases) ' ' num2str(ext * bases)
' ' num2str(ext2 * bases) ' ' num2str(DNA_scale(2))];
            l{springcount,1} = s;
            springcount = springcount+1;
        end;

        % Cross brace links
        for g = 1:length(armStM1)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))  '
' num2str(armStM1(g)+ sticky_offset*(iter-1)) ' '  num2str(armStBM1(g)+
sticky_offset*(iter-1)) ' 2.62488094968e-009 ' num2str(Stiff_scale) ];
            l{springcount,1} = s;
            springcount = springcount+1;
        end;

        for g = 1:length(armStM1)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))  '
' num2str(armStM1(g)+ sticky_offset*(iter-1)) ' '  num2str(armStBM2(g)+
sticky_offset*(iter-1)) ' 2.62488094968e-009 ' num2str(Stiff_scale) ];
            l{springcount,1} = s;
            springcount = springcount+1;
        end;

        for g = 1:length(armStM1)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))  '
' num2str(armStM3(g)+ sticky_offset*(iter-1)) ' '  num2str(armStBM3(g)+
sticky_offset*(iter-1)) ' 2.62488094968e-009 ' num2str(Stiff_scale) ];
            l{springcount,1} = s;
            springcount = springcount+1;
        end;

        for g = 1:length(armStM1)
            s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))  '
' num2str(armStM3(g)+ sticky_offset*(iter-1)) ' '  num2str(armStBM4(g)+
sticky_offset*(iter-1)) ' 2.62488094968e-009 ' num2str(Stiff_scale) ];
            l{springcount,1} = s;
            springcount = springcount+1;
        end;
    end
end
l
```

Source code for sticky end segment model generator for cross-linked models:

```
clear;
clc;
```

```
close all;

avg_mass_per_bp = 0.66;     % kg/mole
avg_mass_per_base = 0.33;   % kg/mole
dsDNA_EC = 24156;
ssDNA_EC = 29181;
ext = avg_mass_per_bp * ssDNA_EC;% 9777;
ext2 = avg_mass_per_bp * dsDNA_EC; % 7984;

seg = 4;
count = 2;
counter = 8;

sticky_inc = 4;
mass_seg_inc = 16;
arm_node_offset = 32;
sticky_offset = 288;

bp = 3.4e-10;
springcount = 1;

%------------------------
% Change these parameters
tiles = 8;
DNA_scale = [1 1];
Stiff_scale = 1;
theory = 3;
x_link = 0;
%------------------------

if(tiles == 2)
    count1 = 1;
    count2 = 1;
end;

if(tiles == 4)
    count1 = 2;
    count2 = 2;
end;

if(tiles == 8)
    count1 = 2;
    count2 = 6;
end;

springnum = 800;
% Spring connections
springseg = [1 2 3 4];
strain = 2.11243934824e-009;
sStrain = 2e-9;

rodL = [2 1 2 1];
rodR = [3 4 4 3];
stiffL = [1 2 3 4];
stiffR = [5 6 7 8];

braceL = [2 1 2 4];
braceR = [3 4 1 3];
braceLS = [2 1 7 3];
braceRS = [6 5 8 4];

for(iter2 = 1:count1)
```

```
if(tiles == 8 && iter2 == 1)
    count2 = 4;
end
if(tiles == 8 && iter2 == 2)
   count2 = 6;
end
for(iter = 1:count2)
    % Set nodes depending on tiles being connected:
    if(iter2 == 1) % Between horizontal tiles
        if(iter == 1)
            adjust = [32 84 48 100 84 100];

            armStM5 = [0 1 2 3] + 1696;
            armStM6 = [4 5 6 7] + 1696;

            if(theory == 1)
                melt = [13 13];
            end
            if(theory == 2)
                melt = [26 26];
            end
            if(theory == 3)
                melt = [45 42.5];
            end
            if(theory == 4)
                melt = [54.6 50.8];
            end
        end;
        if(iter == 2)
            adjust = [32 84 48 100 84 100] + 136;

            armStM5 = [0 1 2 3] + 1712;
            armStM6 = [4 5 6 7] + 1712;

            if(theory == 1)
                melt = [9 10];
            end
            if(theory == 2)
                melt = [19 19];
            end
            if(theory == 3)
                melt = [40.2 42.7];
            end
            if(theory == 4)
                melt = [50.8 54.6];
            end
        end;
        if(iter == 3)
            adjust = [32 84 48 100 84 100] + 136 * 2;

            armStM5 = [0 1 2 3] + 1720;
            armStM6 = [4 5 6 7] + 1720;

            if(theory == 1)
                melt = [12 12];
            end
            if(theory == 2)
                melt = [24 24];
            end
            if(theory == 3)
                melt = [44.4 41.9];
            end
            if(theory == 4)
```

```
            melt = [54.6 50.8];
        end
    end;
    if(iter == 4)
        adjust = [32 84 48 100 84 100] + 136 * 3;

        armStM5 = [0 1 2 3] + 1728;
        armStM6 = [4 5 6 7] + 1728;

        if(theory == 1)
            melt = [10 10];
        end
        if(theory == 2)
            melt = [20 20];
        end
        if(theory == 3)
            melt = [40.5 43];
        end
        if(theory == 4)
            melt = [50.8 54.6];
        end
    end;
    % Sticky end connections
    armStM1 = [44 45 46 47] + adjust(1);
    armStM2 = [128 129 130 131] + adjust(2);

    armStM3 = [60 61 62 63] + adjust(3);
    armStM4 = [144 145 146 147] + adjust(4);

    % Sticky end cross braces
    armStBM1 = [129 128 131 130] + adjust(5);
    armStBM2 = [130 131 128 129] + adjust(5);

    armStBM3 = [145 144 147 146] + adjust(6);
    armStBM4 = [146 147 144 145] + adjust(6);
end;

if(iter2 == 2) % Between vertical tiles
    if(iter == 1)
        adjust = [0 236 0 252 236 252];

        if(theory == 1)
            melt = [18 23];
        end
        if(theory == 2)
            melt = [41 41];
        end
        if(theory == 3)
            melt = [45.9 49.8];
        end
        if(theory == 4)
            melt = [56.2 54.2];
        end
    end;
    if(iter == 2)
        adjust = [-4 240 4 248 240 248] + 88;

        if(theory == 1)
            melt = [16 18];
        end
        if(theory == 2)
            melt = [34 34];
        end
```

```
    if(theory == 3)
        melt = [47.5 48.8];
    end
    if(theory == 4)
        melt = [54.2 56.2];
    end
    sticky_offset = 128;
end;
if(iter == 3)
    adjust = [-8 240 -8 244 240 244] + 88 * 2;
    armStM5 = [0 1 2 3] + 1736;
    armStM6 = [4 5 6 7] + 1736;

    if(theory == 1)
        melt = [11 11];
    end
    if(theory == 2)
        melt = [22 22];
    end
    if(theory == 3)
        melt = [43.5 44.8];
    end
    if(theory == 4)
        melt = [56.2 54.2];
    end
    sticky_offset = 128;
end;
if(iter == 4)
    adjust = [-12 232 -16 240 232 240] + 88 * 3;
    armStM5 = [0 1 2 3] + 1744;
    armStM6 = [4 5 6 7] + 1744;
    if(theory == 1)
        melt = [9 9];
    end
    if(theory == 2)
        melt = [18 18];
    end
    if(theory == 3)
        melt = [43.5 42.1];
    end
    if(theory == 4)
        melt = [54.2 56.2];
    end
    sticky_offset = 128;
end;
if(iter == 5)
    adjust = [-16 220 -16 236 220 236] + 88 * 4;
    if(theory == 1)
        melt = [16 16];
    end
    if(theory == 2)
        melt = [32 32];
    end
    if(theory == 3)
        melt = [48.8 46.8];
    end
    if(theory == 4)
        melt = [56.2 54.2];
    end
    sticky_offset = 128;
end;
if(iter == 6)
    adjust = [-20 224 -12 232 224 232] + 88 * 5;
```

```
                if(theory == 1)
                    melt = [15 15];
                end
                if(theory == 2)
                    melt = [30 30];
                end
                if(theory == 3)
                    melt = [46.1 47.5];
                end
                if(theory == 4)
                    melt = [54.2 56.2];
                end
                sticky_offset = 128;
            end;

            % Sticky end connections
            armStM1 = [112 113 114 115] + adjust(1);
            armStM2 = [364 365 366 367] + adjust(2);

            armStM3 = [136 137 138 139] + adjust(3);
            armStM4 = [380 381 382 383] + adjust(4);

            % Sticky end cross braces
            armStBM1 = [365 364 367 366] + adjust(5);
            armStBM2 = [366 367 364 365] + adjust(5);

            armStBM3 = [381 380 383 382] + adjust(6);
            armStBM4 = [382 383 380 381] + adjust(6);
        end;

        % Sticky end links
        seg_num = springnum + iter + 4 * (iter2 - 1);

        if(seg_num == 805 || seg_num == 806 || seg_num == 809 || seg_num == 810)
            for g = 1:length(armStM1)
                s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' 0 '
num2str(armStM1(g)+ sticky_offset*(iter-1)) ' '  num2str(armStM2(g)+ sticky_offset*(iter-
1)) ' ' num2str(1.7e-9) ' ' num2str(melt(1)) ' ' num2str(5) ' ' num2str(5 * ext) ' '
num2str(5 * ext2) ' ' num2str(DNA_scale(1))];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;

            for g = 1:length(armStM1)
                s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' 0 '
num2str(armStM3(g)+ sticky_offset*(iter-1)) ' '  num2str(armStM4(g)+ sticky_offset*(iter-
1)) ' ' num2str(1.7e-9) ' ' num2str(melt(2)) ' ' num2str(5) ' ' num2str(5 * ext) ' '
num2str(5 * ext2) ' ' num2str(DNA_scale(2))];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;

            % Cross brace links
            for g = 1:length(armStM1)
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armStM1(g)+ sticky_offset*(iter-1)) ' '  num2str(armStBM1(g)+
sticky_offset*(iter-1)) ' 2.62488094968e-009 ' num2str(Stiff_scale) ];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;

            for g = 1:length(armStM1)
```

```
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armStM1(g)+ sticky_offset*(iter-1)) ' '  num2str(armStBM2(g)+
sticky_offset*(iter-1)) ' 2.62488094968e-009 ' num2str(Stiff_scale) ];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;

            for g = 1:length(armStM1)
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armStM3(g)+ sticky_offset*(iter-1)) ' '  num2str(armStBM3(g)+
sticky_offset*(iter-1)) ' 2.62488094968e-009 ' num2str(Stiff_scale) ];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;

            for g = 1:length(armStM1)
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armStM3(g)+ sticky_offset*(iter-1)) ' '  num2str(armStBM4(g)+
sticky_offset*(iter-1)) ' 2.62488094968e-009 ' num2str(Stiff_scale) ];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
        end

        if(seg_num >= 802 && seg_num <= 804 || seg_num == 807)
            for g = 1:length(armStM1)
                s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' 0 '
num2str(armStM1(g)+ sticky_offset*(iter-1)) ' '  num2str(armStM2(g)+ sticky_offset*(iter-
1)) ' ' num2str(1.7e-9) ' ' num2str(melt(1)) ' ' num2str(5) ' ' num2str(5 * ext) ' '
num2str(5 * ext2) ' ' num2str(DNA_scale(1))];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;


            % Cross brace links
            for g = 1:length(armStM1)
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armStM1(g)+ sticky_offset*(iter-1)) ' '  num2str(armStBM1(g)+
sticky_offset*(iter-1)) ' 2.62488094968e-009 ' num2str(Stiff_scale) ];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;

            for g = 1:length(armStM1)
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armStM1(g)+ sticky_offset*(iter-1)) ' '  num2str(armStBM2(g)+
sticky_offset*(iter-1)) ' 2.62488094968e-009 ' num2str(Stiff_scale) ];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
        end

        if(seg_num == 801)
            base_length = [1 2 3];
            for h = 1:3
                x_length = base_length(h) * 3.4e-10;
                offset = 1696;
                ext_1 = base_length(h) * ext;
                ext_2 = base_length(h) * ext2;

                if(h == 1)
                    for g = 1:seg
```

```
                        s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armStM1(g)) ' '  num2str(armStM5(g)) ' ' num2str(x_length) ' '
num2str(melt(1)) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' ' num2str(ext_2) ' '
num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                    for g = 1:seg
                        s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armStM3(g)) ' '  num2str(armStM5(g) + 8) ' ' num2str(x_length) ' '
num2str(melt(1)) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' ' num2str(ext_2) ' '
num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
                if(h == 2)
                    for g = 1:seg
                        s = ['X ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(h) ' ' num2str(armStM5(g)) ' '  num2str(armStM6(g)) ' ' num2str(x_length) ' '
num2str(0) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' ' num2str(ext_2) ' '
num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                    for g = 1:seg
                        s = ['X ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(h) ' ' num2str(armStM5(g) + 8) ' '  num2str(armStM6(g) + 8) ' ' num2str(x_length)
' ' num2str(0) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' ' num2str(ext_2) ' '
num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
                if(h == 3)
                    for g = 1:seg
                        s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armStM6(g)) ' '  num2str(armStM2(g)) ' ' num2str(x_length) ' '
num2str(melt(1)) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' ' num2str(ext_2) ' '
num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                    for g = 1:seg
                        s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armStM6(g) + 8) ' '  num2str(armStM4(g)) ' ' num2str(x_length) ' '
num2str(melt(1)) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' ' num2str(ext_2) ' '
num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
            end

            % Cross brace links
            % Bottom arm
            armLM1_3 =  [ 1696 1698 1700 1702 1696 1697 1700 1701  1696 1697 1698 1699
1696 1697 1698 1699  1696 1698 1700 1702 ];
            armLM2_3 =  [ 1697 1699 1701 1703 1698 1699 1702 1703  1701 1700 1703 1702
1702 1703 1700 1701  1699 1697 1703 1701 ];
            rodL_3 =    [2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 ];
            stiffL_3 =  [1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4];
```

```
            for g = 1:length(armLM1_3)
                if(g < 9)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
num2str(sStrain) ' ' num2str(Stiff_scale)];
                end
                if(g > 8 && g < 17)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
num2str(strain) ' ' num2str(Stiff_scale)];
                end
                if(g > 16)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
                end
                l{springcount,1} = s;
                springcount = springcount + 1;
            end

            armCM3_3 = [ 76   77   78   79   ];
            armCM3_4 = [ 1697 1696 1699 1698 ];
            armCM3_5 = [ 76   78   77   79];
            armCM3_6 = [ 1698 1696 1699 1697];

            % Braces
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armCM3_3(g)) ' '  num2str(armCM3_4(g)) ' ' num2str(sqrt(2e-9^2 + (1*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armCM3_5(g)) ' '  num2str(armCM3_6(g)) ' ' num2str(sqrt(2e-9^2 + (1*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;

            armCM3_3 = [ 1700 1701 1702 1703 ];
            armCM3_4 = [ 213  212  215  214 ];
            armCM3_5 = [ 1700 1702 1701 1703 ];
            armCM3_6 = [ 214  212  215  213 ];

            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g))
' ' num2str(armCM3_3(g)) ' '  num2str(armCM3_4(g)) ' ' num2str(sqrt(2e-9^2 + (3*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g))
' ' num2str(armCM3_5(g)) ' '  num2str(armCM3_6(g)) ' ' num2str(sqrt(2e-9^2 + (3*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;

            % Top arm
```

```
            armLM1_3 =  [ 1696 1698 1700 1702 1696 1697 1700 1701  1696 1697 1698 1699
1696 1697 1698 1699  1696 1698 1700 1702 ] + 8;
            armLM2_3 =  [ 1697 1699 1701 1703 1698 1699 1702 1703  1701 1700 1703 1702
1702 1703 1700 1701  1699 1697 1703 1701 ] + 8;
            rodL_3 =    [2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 ];
            stiffL_3 =  [1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4];

            for g = 1:length(armLM1_3)
                if(g < 9)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' ' num2str(2e-9)
' ' num2str(Stiff_scale)];
                end
                if(g > 8 && g < 17)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
num2str(strain) ' ' num2str(Stiff_scale)];
                end
                if(g > 16)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
                end
                l{springcount,1} = s;
                springcount = springcount + 1;
            end

            armCM3_3 = [ 76   77   78   79   ] + 32;
            armCM3_4 = [ 1697 1696 1699 1698 ] + 8;
            armCM3_5 = [ 76   78   77   79] + 32;
            armCM3_6 = [ 1698 1696 1699 1697] + 8;

            % Braces
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armCM3_3(g)) ' '  num2str(armCM3_4(g)) ' ' num2str(sqrt(2e-9^2 + (1*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armCM3_5(g)) ' '  num2str(armCM3_6(g)) ' ' num2str(sqrt(2e-9^2 + (1*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;

            armCM3_3 = [ 1700 1701 1702 1703 ] + 8;
            armCM3_4 = [ 213  212  215  214 ] + 32;
            armCM3_5 = [ 1700 1702 1701 1703 ] + 8;
            armCM3_6 = [ 214  212  215  213 ] + 32;

            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g))
' ' num2str(armCM3_3(g)) ' '  num2str(armCM3_4(g)) ' ' num2str(sqrt(2e-9^2 + (3*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
            for g = 1:length(armCM3_3)
```

```
                s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g))
' ' num2str(armCM3_5(g)) ' '  num2str(armCM3_6(g)) ' ' num2str(sqrt(2e-9^2 + (3*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
        end


        if(seg_num == 802)
            base_length = [1 2 3];
            for h = 1:3
                x_length = base_length(h) * 3.4e-10;
                offset = 136 * 2 + 12;
                ext_1 = base_length(h) * ext;
                ext_2 = base_length(h) * ext2;

                if(h == 1)
                    for g = 1:seg
                        s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armStM3(g) + sticky_offset*(iter-1)) ' '  num2str(armStM5(g)) ' '
num2str(x_length) ' ' num2str(melt(1)) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' '
num2str(ext_2) ' ' num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
                if(h == 2)
                    for g = 1:seg
                        s = ['X ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(h) ' ' num2str(armStM5(g)) ' '  num2str(armStM6(g)) ' ' num2str(x_length) ' '
num2str(0) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' ' num2str(ext_2) ' '
num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
                if(h == 3)
                    for g = 1:seg
                        s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armStM6(g)) ' '  num2str(armStM4(g) + sticky_offset*(iter-1)) ' '
num2str(x_length) ' ' num2str(melt(1)) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' '
num2str(ext_2) ' ' num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
            end

            % Cross brace links
            % Top arm
            adjust = 16;
            armLM1_3 =  [ 1696 1698 1700 1702 1696 1697 1700 1701  1696 1697 1698 1699
1696 1697 1698 1699  1696 1698 1700 1702 ] + adjust;
            armLM2_3 =  [ 1697 1699 1701 1703 1698 1699 1702 1703  1701 1700 1703 1702
1702 1703 1700 1701  1699 1697 1703 1701 ] + adjust;
            rodL_3 =    [2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 ];
            stiffL_3 =  [1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4];

            for g = 1:length(armLM1_3)
                if(g < 9)
```

```
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
num2str(sStrain) ' ' num2str(Stiff_scale)];
                end
                if(g > 8 && g < 17)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
num2str(strain) ' ' num2str(Stiff_scale)];
                end
                if(g > 16)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
                end
                l{springcount,1} = s;
                springcount = springcount + 1;
            end


            armCM3_3 = [ 528 529 530 531 ] + 4;
            armCM3_4 = [ 1697 1696 1699 1698 ] + adjust;
            armCM3_5 = [ 528 530 529 531] + 4;
            armCM3_6 = [ 1698 1696 1699 1697] + adjust;

            % Braces
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armCM3_3(g)) ' '  num2str(armCM3_4(g)) ' ' num2str(sqrt(2e-9^2 + (1*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armCM3_5(g)) ' '  num2str(armCM3_6(g)) ' ' num2str(sqrt(2e-9^2 + (1*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
            adjust = 16;
            armCM3_3 = [ 1700 1701 1702 1703 ] + adjust;
            armCM3_4 = [ 669 668 671 670 ];
            armCM3_5 = [ 1700 1702 1701 1703 ] + adjust;
            armCM3_6 = [ 670 668 671 669 ];

            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g))
' ' num2str(armCM3_3(g)) ' '  num2str(armCM3_4(g)) ' ' num2str(sqrt(2e-9^2 + (3*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g))
' ' num2str(armCM3_5(g)) ' '  num2str(armCM3_6(g)) ' ' num2str(sqrt(2e-9^2 + (3*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
        end

        if(seg_num == 803)
            base_length = [1 2 3];
```

```
            for h = 1:3
                x_length = base_length(h) * 3.4e-10;
                offset = 148 * 4 - 16 ;
                ext_1 = base_length(h) * ext;
                ext_2 = base_length(h) * ext2;

                if(h == 1)
                    for g = 1:seg
                        s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armStM3(g) + offset) ' '  num2str(armStM5(g)) ' '
num2str(x_length) ' ' num2str(melt(1)) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' '
num2str(ext_2) ' ' num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
                if(h == 2)
                    for g = 1:seg
                        s = ['X ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(h) ' ' num2str(armStM5(g)) ' '  num2str(armStM6(g)) ' ' num2str(x_length) ' '
num2str(0) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' ' num2str(ext_2) ' '
num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
                if(h == 3)
                    for g = 1:seg
                        s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armStM6(g)) ' '  num2str(armStM4(g) + offset) ' '
num2str(x_length) ' ' num2str(melt(1)) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' '
num2str(ext_2) ' ' num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
            end

            % Cross brace links
            % Top arm
            adjust = 24;
            armLM1_3 =  [ 1696 1698 1700 1702 1696 1697 1700 1701  1696 1697 1698 1699
1696 1697 1698 1699  1696 1698 1700 1702 ] + adjust;
            armLM2_3 =  [ 1697 1699 1701 1703 1698 1699 1702 1703  1701 1700 1703 1702
1702 1703 1700 1701  1699 1697 1703 1701 ] + adjust;
            rodL_3 =    [2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 ];
            stiffL_3 =  [1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4];

            for g = 1:length(armLM1_3)
                if(g < 9)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
num2str(sStrain) ' ' num2str(Stiff_scale)];
                end
                if(g > 8 && g < 17)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
num2str(strain) ' ' num2str(Stiff_scale)];
                end
                if(g > 16)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
```

```
                end
                l{springcount,1} = s;
                springcount = springcount + 1;
            end

            armCM3_3 = [ 528 529 530 531 ] + 428;
            armCM3_4 = [ 1697 1696 1699 1698 ] + adjust;
            armCM3_5 = [ 528 530 529 531] + 428;
            armCM3_6 = [ 1698 1696 1699 1697] + adjust;

            % Braces
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armCM3_3(g)) ' '  num2str(armCM3_4(g)) ' ' num2str(sqrt(2e-9^2 + (1*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armCM3_5(g)) ' '  num2str(armCM3_6(g)) ' ' num2str(sqrt(2e-9^2 + (1*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;

            armCM3_3 = [ 1700 1701 1702 1703 ] + adjust;
            armCM3_4 = [ 669 668 671 670 ] + 424;
            armCM3_5 = [ 1700 1702 1701 1703 ] + adjust;
            armCM3_6 = [ 670 668 671 669 ] + 424;

            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g))
' ' num2str(armCM3_3(g)) ' '  num2str(armCM3_4(g)) ' ' num2str(sqrt(2e-9^2 + (3*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g))
' ' num2str(armCM3_5(g)) ' '  num2str(armCM3_6(g)) ' ' num2str(sqrt(2e-9^2 + (3*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
        end

        if(seg_num == 804)
            base_length = [1 2 3];
            for h = 1:3
                x_length = base_length(h) * 3.4e-10;
                offset = 148 * 6 - 24 ;
                ext_1 = base_length(h) * ext;
                ext_2 = base_length(h) * ext2;

                if(h == 1)
                    for g = 1:seg
                        s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armStM3(g) + offset) ' '  num2str(armStM5(g)) ' '
num2str(x_length) ' ' num2str(melt(1)) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' '
num2str(ext_2) ' ' num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
```

```
                    end
                end
                if(h == 2)
                    for g = 1:seg
                        s = ['X ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(h) ' ' num2str(armStM5(g)) ' '  num2str(armStM6(g)) ' ' num2str(x_length) ' '
num2str(0) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' ' num2str(ext_2) ' '
num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
                if(h == 3)
                    for g = 1:seg
                        s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armStM6(g)) ' '  num2str(armStM4(g) + offset) ' '
num2str(x_length) ' ' num2str(melt(1)) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' '
num2str(ext_2) ' ' num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
            end

            % Cross brace links
            % Top arm
            adjust = 32;
           armLM1_3 =  [ 1696 1698 1700 1702 1696 1697 1700 1701  1696 1697 1698 1699
1696 1697 1698 1699  1696 1698 1700 1702 ] + adjust;
            armLM2_3 =  [ 1697 1699 1701 1703 1698 1699 1702 1703  1701 1700 1703 1702
1702 1703 1700 1701  1699 1697 1703 1701 ] + adjust;
            rodL_3 =    [2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 ];
            stiffL_3 =  [1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4];

            for g = 1:length(armLM1_3)
                if(g < 9)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
num2str(sStrain) ' ' num2str(Stiff_scale)];
                end
                if(g > 8 && g < 17)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
num2str(strain) ' ' num2str(Stiff_scale)];
                end
                if(g > 16)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
                end
                l{springcount,1} = s;
                springcount = springcount + 1;
            end

            armCM3_3 = [ 528 529 530 531 ] + 852;
            armCM3_4 = [ 1697 1696 1699 1698 ] + adjust;
            armCM3_5 = [ 528 530 529 531] + 852;
            armCM3_6 = [ 1698 1696 1699 1697] + adjust;

            % Braces
            for g = 1:length(armCM3_3)
```

```
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armCM3_3(g)) ' '  num2str(armCM3_4(g)) ' ' num2str(sqrt(2e-9^2 + (1*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armCM3_5(g)) ' '  num2str(armCM3_6(g)) ' ' num2str(sqrt(2e-9^2 + (1*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;

            armCM3_3 = [ 1700 1701 1702 1703 ] + adjust;
            armCM3_4 = [ 669 668 671 670 ] + 848;
            armCM3_5 = [ 1700 1702 1701 1703 ] + adjust;
            armCM3_6 = [ 670 668 671 669 ] + 848;

            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g))
' ' num2str(armCM3_3(g)) ' '  num2str(armCM3_4(g)) ' ' num2str(sqrt(2e-9^2 + (3*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g))
' ' num2str(armCM3_5(g)) ' '  num2str(armCM3_6(g)) ' ' num2str(sqrt(2e-9^2 + (3*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
        end

        if(seg_num == 807)
            base_length = [1 2 3];
            for h = 1:3
                x_length = base_length(h) * 3.4e-10;
                offset = 148 * 6 - 24 ;
                ext_1 = base_length(h) * ext;
                ext_2 = base_length(h) * ext2;

                if(h == 1)
                    for g = 1:seg
                        s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armStM3(g)+ sticky_offset*(iter-1)) ' '  num2str(armStM5(g)) ' '
num2str(x_length) ' ' num2str(melt(1)) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' '
num2str(ext_2) ' ' num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
                if(h == 2)
                    for g = 1:seg
                        s = ['X ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(h) ' ' num2str(armStM5(g)) ' '  num2str(armStM6(g)) ' ' num2str(x_length) ' '
num2str(0) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' ' num2str(ext_2) ' '
num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
```

```
                if(h == 3)
                    for g = 1:seg
                        s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armStM6(g)) ' '  num2str(armStM4(g)+ sticky_offset*(iter-1)) ' '
num2str(x_length) ' ' num2str(melt(1)) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' '
num2str(ext_2) ' ' num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
            end

            % Cross brace links
            % Top arm
            adjust = 40;
            armLM1_3 =  [ 1696 1698 1700 1702 1696 1697 1700 1701  1696 1697 1698 1699
1696 1697 1698 1699  1696 1698 1700 1702 ] + adjust;
            armLM2_3 =  [ 1697 1699 1701 1703 1698 1699 1702 1703  1701 1700 1703 1702
1702 1703 1700 1701  1699 1697 1703 1701 ] + adjust;
            rodL_3 =    [2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 ];
            stiffL_3 =  [1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4];

            for g = 1:length(armLM1_3)
                if(g < 9)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
num2str(sStrain) ' ' num2str(Stiff_scale)];
                end
                if(g > 8 && g < 17)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
num2str(strain) ' ' num2str(Stiff_scale)];
                end
                if(g > 16)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
                end
                l{springcount,1} = s;
                springcount = springcount + 1;
            end

            armCM3_3 = [ 528 529 530 531 ] + 32;
            armCM3_4 = [ 1697 1696 1699 1698 ] + adjust;
            armCM3_5 = [ 528 530 529 531] + 32;
            armCM3_6 = [ 1698 1696 1699 1697] + adjust;

            % Braces
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armCM3_3(g)) ' '  num2str(armCM3_4(g)) ' ' num2str(sqrt(2e-9^2 + (1*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armCM3_5(g)) ' '  num2str(armCM3_6(g)) ' ' num2str(sqrt(2e-9^2 + (1*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
```

```
            armCM3_3 = [ 1700 1701 1702 1703 ] + adjust;
            armCM3_4 = [ 669 668 671 670 ] + 388;
            armCM3_5 = [ 1700 1702 1701 1703 ] + adjust;
            armCM3_6 = [ 670 668 671 669 ] + 388;

            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g))
' ' num2str(armCM3_3(g)) ' '  num2str(armCM3_4(g)) ' ' num2str(sqrt(2e-9^2 + (3*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g))
' ' num2str(armCM3_5(g)) ' '  num2str(armCM3_6(g)) ' ' num2str(sqrt(2e-9^2 + (3*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
        end

        if(seg_num == 808)
            base_length = [4 2];
            for h = 1:2
                x_length = base_length(h) * 3.4e-10;
                ext_1 = base_length(h) * ext;
                ext_2 = base_length(h) * ext2;

                if(h == 1)
                    for g = 1:seg
                        s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armStM1(g)+ sticky_offset*(iter-1)) ' '  num2str(armStM5(g)) ' '
num2str(x_length) ' ' num2str(melt(1)) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' '
num2str(ext_2) ' ' num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
                if(h == 2)
                    for g = 1:seg
                        s = ['X ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(h) ' ' num2str(armStM5(g)) ' '  num2str(armStM2(g)+ sticky_offset*(iter-1)) ' '
num2str(x_length) ' ' num2str(0) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' '
num2str(ext_2) ' ' num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
            end

            % Cross brace links
            % Right arm

            armCM3_3 = [ 1744 1745 1746 1747 ];
            armCM3_4 = [ 749  748  751  750];
            armCM3_5 = [ 1744 1745 1746 1747 ];
            armCM3_6 = [ 750  751  748  749 ];

            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g))
' ' num2str(armCM3_3(g)) ' '  num2str(armCM3_4(g)) ' ' num2str(sqrt(2e-9^2 + (4*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
```

```
                springcount = springcount+1;
            end;
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g))
' ' num2str(armCM3_5(g)) ' '  num2str(armCM3_6(g)) ' ' num2str(sqrt(2e-9^2 + (4*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;

            armLM1_3 =  [ 1744 1745 1746 1744  1744 1746 1746 1747 1747 1745 1745 1744
1744 1745 ];
            armLM2_3 =  [ 1746 1747 1747 1745  1246 1244 1247 1246 1245 1247 1244 1245
1747 1746 ];
            rodL_3 =    [2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 ];
            stiffL_3 =  [1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 ];

            for g = 1:length(armLM1_3)
                if(g < 5)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
num2str(sStrain) ' ' num2str(Stiff_scale)];
                end
                if(g > 4 && g < 13)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
num2str(strain) ' ' num2str(Stiff_scale)];
                end
                if(g > 12)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
                end
                l{springcount,1} = s;
                springcount = springcount + 1;
            end

            base_length = [1 2 3];
            for h = 1:3
                x_length = base_length(h) * 3.4e-10;
                ext_1 = base_length(h) * ext;
                ext_2 = base_length(h) * ext2;

                if(h == 1)
                    for g = 1:seg
                        s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armStM3(g)+ sticky_offset*(iter-1)) ' '  num2str(armStM5(g) + 4) '
' num2str(x_length) ' ' num2str(melt(1)) ' ' num2str(base_length(h)) ' ' num2str(ext_1) '
' num2str(ext_2) ' ' num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
                if(h == 2)
                    for g = 1:seg
                        s = ['X ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(h) ' ' num2str(armStM5(g) + 4) ' '  num2str(armStM6(g) + 4) ' ' num2str(x_length)
' ' num2str(0) ' ' num2str(base_length(h)) ' ' num2str(ext_1) ' ' num2str(ext_2) ' '
num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
```

```
                if(h == 3)
                    for g = 1:seg
                        s = ['S ' num2str(seg_num) ' ' num2str(springseg(g)) ' '
num2str(0) ' ' num2str(armStM6(g) + 4) ' '  num2str(armStM4(g)+ sticky_offset*(iter-1)) '
' num2str(x_length) ' ' num2str(melt(1)) ' ' num2str(base_length(h)) ' ' num2str(ext_1) '
' num2str(ext_2) ' ' num2str(1) ];
                        l{springcount,1} = s;
                        springcount = springcount + 1;
                    end
                end
            end

            % Cross brace links
            adjust = 52;
            armLM1_3 =  [ 1696 1698 1700 1702 1696 1697 1700 1701  1696 1697 1698 1699
1696 1697 1698 1699  1696 1698 1700 1702 ] + adjust;
            armLM2_3 =  [ 1697 1699 1701 1703 1698 1699 1702 1703  1701 1700 1703 1702
1702 1703 1700 1701  1699 1697 1703 1701 ] + adjust;
            rodL_3 =    [2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 ];
            stiffL_3 =  [1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4];

            for g = 1:length(armLM1_3)
                if(g < 9)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
num2str(sStrain) ' ' num2str(Stiff_scale)];
                end
                if(g > 8 && g < 17)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
num2str(strain) ' ' num2str(Stiff_scale)];
                end
                if(g > 16)
                    s = ['T ' num2str(seg_num) ' ' num2str(rodL_3(g)) ' '
num2str(stiffL_3(g)) ' ' num2str(armLM1_3(g)) ' '  num2str(armLM2_3(g)) ' '
'2.82842712474e-009' ' ' num2str(Stiff_scale)];
                end
                l{springcount,1} = s;
                springcount = springcount + 1;
            end

            armCM3_3 = [ 76   77   78   79   ] + 692;
            armCM3_4 = [ 1697 1696 1699 1698 ] + adjust;
            armCM3_5 = [ 76   78   77   79] + 692;
            armCM3_6 = [ 1698 1696 1699 1697] + adjust;

            % Braces
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armCM3_3(g)) ' '  num2str(armCM3_4(g)) ' ' num2str(sqrt(2e-9^2 + (1*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceL(g)) ' ' num2str(braceLS(g))
' ' num2str(armCM3_5(g)) ' '  num2str(armCM3_6(g)) ' ' num2str(sqrt(2e-9^2 + (1*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;

            armCM3_3 = [ 1700 1701 1702 1703 ] + adjust;
```

```
            armCM3_4 = [ 213  212  215  214 ] + 1056;
            armCM3_5 = [ 1700 1702 1701 1703 ] + adjust;
            armCM3_6 = [ 214  212  215  213 ] + 1056;

            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g))
' ' num2str(armCM3_3(g)) ' '  num2str(armCM3_4(g)) ' ' num2str(sqrt(2e-9^2 + (3*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;
            for g = 1:length(armCM3_3)
                s = ['B ' num2str(seg_num) ' ' num2str(braceR(g)) ' ' num2str(braceRS(g))
' ' num2str(armCM3_5(g)) ' '  num2str(armCM3_6(g)) ' ' num2str(sqrt(2e-9^2 + (3*bp)^2)) '
' num2str(1)];
                l{springcount,1} = s;
                springcount = springcount+1;
            end;


        end
    end
end
1
```

6. The Thermodynamic Parameter Extraction Program Source Code

The thermodynamic parameters for the van't Hoff plots generated from the DNA-STRAIN simulator are determined using Equation 10 as a best fit curve to the plot by varying the parameters for the magnitude of the peak, the van't Hoff enthalpy, and the closest melting temperature.

```
% Pielak paper method processing
% Vincent Mao
% written: December 5, 2007
% last modified: October 22, 2008
% --------------------------------
% This program is used to optimize 3 adjustable parameters for
% a nonlinear curve fitting of data for absorbance data of DNA
% as it is denatured and reannealed over a temperature ramp.
% The method uses the van't Hoff analysis as described in the paper
% from John and Weeks '00 "van't Hoff enthalpies without baselines".  The
% correction to their typo in terms of the temperature term is included.

%==================================
% Equations
%==================================

% rate constant :K = exp((Hvh/R*(1/Tm - 1./T)));
% fraction of denatured DNA: f = K ./(K+1);

% change of absorbance over temperature: dAbsdT = (A .* f .* (1-f))./T.^2;

% full expression by substituting all known equations into dAbsdT
```

```
% v(1) = A = scaling factor, v(2) = Hvh = van't Hoff enthalpy, v(3) = Tm =
% melting temperature

close all;
clear;
clc;

format short g;
fsum = 0;
R = 8.314;
HV = 33000*4.184;
Tm = 273 + 56;
x = 125; %25
A= (x*HV)/(Tm*Tm*R);
v0 = [ A HV Tm ];
final = 1;

% Load data

% xdata = xlsread('1TTests.xls', 5, 'A100:A120') +  273;
% ydata = xlsread('1TTests.xls', 5, 'E100:E120');

% xdata = xlsread('1TTestsOpt.xls', 1, 'A140:A220') +  273;
% ydata = xlsread('1TTestsOpt.xls', 1, 'C140:C220');

% xdata = xlsread('NSTests.xls', 14, 'A540:A550');
% ydata = xlsread('NSTests.xls', 14, 'J540:J550');

xdata = xlsread('NSTests2.xls', 4, 'A700:A890');
ydata = xlsread('NSTests2.xls', 4, 'K700:K890');

% xdata = xlsread('HystMelts.xls', 2, 'AM80:AM98')+273;
% ydata = xlsread('HystMelts.xls', 2, 'AV80:AV98');

% xdata = xlsread('SimpleMeltOutput1k.xls', 14, 'K3:K1390');
% ydata = xlsread('SimpleMeltOutput1k.xls', 14, 'O3:O1390');%* 1e19;

% xdata = xlsread('SimpleMeltOutput1k.xls', 15, 'A3:A440');
% ydata = xlsread('SimpleMeltOutput1k.xls', 15, 'J3:J440');

% xdata = xlsread('ComplexMeltOutput1k.xls', 13, 'A800:A890');
% ydata = xlsread('ComplexMeltOutput1k.xls', 13, 'J800:J890'); % * 1e19;

% xdata = xlsread('8TTests.xls', 9, 'A540:A660');
% ydata = xlsread('8TTests.xls', 9, 'J540:J660');% * 1e19;

% xdata = xlsread('CtlMelts.xls', 5, 'A155:A195') + 273;
% ydata = xlsread('CtlMelts.xls', 5, 'H155:H195');

% xdata = xlsread('XLMelts.xls', 5, 'D160:D200')+273;
% ydata = xlsread('XLMelts.xls', 5, 'H160:H200');

% xdata = xlsread('CtlMelts.xls', 10, 'A202:A237')+273;
% ydata = xlsread('CtlMelts.xls', 10, 'D202:D237');

% xdata = xlsread('XLMelts.xls', 9, 'A190:A230')+273;
% ydata = xlsread('XLMelts.xls', 9, 'F190:F230');

% xdata = xlsread('XLModelMelts.xls', 3, 'A580:A880');
% ydata = xlsread('XLModelMelts.xls', 3, 'J580:J880');

xdata = xdata ;
fitdata = zeros(length(xdata),1);
```

```
fitdata2 = zeros(length(xdata),1);

miny = min(ydata);
xlow = min(xdata);
xhigh = max(xdata);
inc = (xhigh-xlow)/(length(xdata)-1);
xfit = xlow:inc:xhigh;

% Generating the equation for fitting
for i = 1:length(xdata)
    s = ['(exp(  (v(2)/8.314) * (1/v(3)-1/' num2str(xdata(i)) ') ) )' ];
    strfrac = ['(' s '/(' s '+1))'];
    out = ['(v(1)*' strfrac '*(1-' strfrac ')/' num2str(xdata(i)) '^2)'];
    e = ['(' out '-' num2str(ydata(i)) ')^2'];
    if i == 1
        fsum = ['@(v)' e];
    else
        fsum = [fsum '+' e];
    end
end

f = str2num(fsum);
options = optimset('MaxFunEvals',100000,'MaxIter',100000);

% Calculating and writing parameters to array
[v, fval] = fminsearch(f, v0,options);
ot(1,:) = [v,fval];

% Output and analyze results
Hf = ot(final,2);
Tf = ot(final,3)-273;
Km = exp( Hf/R * (1/(Tf + 273) - 1/298) );
Gf = R * (Tf + 273) * log(Km);
Sf = (Gf - Hf) / (-(Tf + 273));
results = [ot(final,1) Tf Hf/4184 Gf/4184 Sf/4.184];
disp('      A              T(C)      H(KCal/mol)    G(KCal/mol)      S(cal/mol K)')
disp(results)

% Generating result of the equation from the fit
for i = 1:length(xfit)
    K = exp((ot(final,2)/R*(1/ot(final,3)-1/(xlow+(i-1)*.01))));
    frac = K/(K+1);
    fit = ot(final,1)*frac*(1-frac)/((xlow+(i-1)*.01)^2);
    fitdata(i) = fit;
end

for g = 1:length(ydata)
    K = exp((ot(final,2)/R*(1/ot(final,3)-1/(xdata(g)))));
    frac = K/(K+1);
    fit2 = ot(final,1)*frac*(1-frac)/((xdata(g))^2);
    fitdata2(g) = fit2;
end

% Calculating statistical data
[x, resnorm, residual] = lsqnonneg(ydata, fitdata2);
lsqnonneg(ydata, fitdata2);
residualssum = sum(residual);
Pearson = corr2(ydata, fitdata2);
chisq = resnorm;

stats = [residualssum Pearson chisq];
disp('   Residual Sum       R        Chi^2')
disp(stats)
```

```
% Plot data

xdata = xdata - 273;
xfit = xfit - 273;

plot(xdata,ydata,'k')
hold on
plot(xfit, fitdata2, '--k')

xlabel('Temp(\circC)')
ylabel('dA/dT')
title(['Fitting to function dA/dT = A*f*(1-f)/T^2']);
title(['van''t Hoff Plot']);
legend('data', ['van''t Hoff fit'])
hold off
```

7. Savitsky-Golay Filter Source Code

```
% Savitsky-Golay Filter for DNA Melt Data
% Vincent Mao
% written: March 4, 2008
% last modified: Feb 12, 2010
% --------------------------------
% This program smooths absorbance data using the Savitsky-Golay FIR
% smoothing filter.  This filter fits a curve using the least squares
% method and preserves the features of the data.

clear;
clc;
close all;

% Load data

% xdata = xlsread('1TTests.xls', 5, 'A4:A240') +  273;
% ydata = xlsread('1TTests.xls', 5, 'B4:B240');

% xdata = xlsread('1TTestsOpt.xls', 2, 'A2:A191') + 273;
% ydata = xlsread('1TTestsOpt.xls', 2, 'B2:B191');

% xdata = xlsread('SimpleMeltOutput1k.xls', 15, 'A1:A450');
% ydata = xlsread('SimpleMeltOutput1k.xls', 15, 'D1:D450');

% xdata = xlsread('ComplexMeltOutput1k.xls', 13, 'A1:A1000');
% ydata = xlsread('ComplexMeltOutput1k.xls', 13, 'D1:D1000');

% xdata = xlsread('8TTests.xls', 9, 'A1:A1000');
% ydata = xlsread('8TTests.xls', 9, 'D1:D1000');

% xdata = xlsread('NSTests.xls', 14, 'A100:A1000');
% ydata = xlsread('NSTests.xls', 14, 'D100:D1000');

xdata = xlsread('NSTests2.xls', 3, 'A1:A1000');
ydata = xlsread('NSTests2.xls', 3, 'D1:D1000');

% xdata = xlsread('HystMelts.xls', 2, 'AM11:AM247')+273;
% ydata = xlsread('HystMelts.xls', 2, 'AU11:AU247');

% xdata = xlsread('CtlMelts.xls', 5, 'A4:A243') +  273;
% ydata = xlsread('CtlMelts.xls', 5, 'E4:E243');

% xdata = xlsread('CtlMelts.xls', 4, 'P8:P247') +  273;
```

```
% ydata = xlsread('CtlMelts.xls', 4, 'U8:U247');

% xdata = xlsread('CtlMelts.xls', 10, 'A4:A360') +  273;
% ydata = xlsread('CtlMelts.xls', 10, 'B4:B360');

% xdata = xlsread('XLMelts.xls', 5, 'D9:D253') +  273;
% ydata = xlsread('XLMelts.xls', 5, 'E9:E253');

% xdata = xlsread('XLMelts.xls', 4, 'P9:P247') +  273;
% ydata = xlsread('XLMelts.xls', 4, 'U9:U247');

% xdata = xlsread('XLMelts.xls', 9, 'A4:A360') +  273;
% ydata = xlsread('XLMelts.xls', 9, 'B4:B360');

% xdata = xlsread('XLModelMelts.xls', 10, 'A50:A1000');
% ydata = xlsread('XLModelMelts.xls', 10, 'D50:D1000')*1e16;

% Define filter order and size of data frame
order = 3;
data =  101;
data2 = 101;

xmin = min(xdata);
xmax = max(xdata);

filtered_data = sgolayfilt(ydata, order, data);
dydtdata = diff(filtered_data);
filtered_dydt = sgolayfilt(dydtdata, order, data2);

[filter, coeff] = sgolay(order,data);

% Calculating statistical data compared against original data after
% filtering and after filtering dydt data.
[x, resnorm, residual] = lsqnonneg(ydata, filtered_data);
lsqnonneg(ydata, filtered_data);
residualssum = sum(residual);
Pearson = corr2(ydata, filtered_data);
chisq = resnorm;

[x2, resnorm2, residual2] = lsqnonneg(dydtdata, filtered_dydt);
lsqnonneg(dydtdata, filtered_dydt);
residualssum2 = sum(residual2);
Pearson2 = corr2(dydtdata, filtered_dydt);
chisq2 = resnorm2;

% This part directly applies SG coefficients using the SG polynomial
% equations
ydata = ydata';
xdata = xdata';

F = data;
g = coeff;

for t = (F+1)/2:length(xdata)-(F+1)/2,
% 0th order derivative
    w0(t)=g(:,1)'*ydata(t - (F+1)/2 + 1: t + (F+1)/2 - 1)';
% 1st order derivative
% Use absolute value for identifying any transitions below zero
%     w1(t)=abs(g(:,2)'*ydata(t - (F+1)/2 + 1: t + (F+1)/2 - 1)');
    w1(t)=(g(:,2)'*ydata(t - (F+1)/2 + 1: t + (F+1)/2 - 1)');
end

ydata = ydata';
```

```
xdata = xdata';

% Statistical analysis outputs from earlier
stats1 = [residualssum Pearson chisq];
stats2 = [residualssum2 Pearson2 chisq2];
disp('     Residual Sum       R        Chi^2')
disp(' Raw data: ');
disp(stats1);
disp(' dy/dt:    ');
disp(stats2);

ymin = min(ydata);
ymax = max(ydata);
ymin2 = min(dydtdata);
ymax2 = max(dydtdata);

figure
subplot(2,1,1)
plot(xdata - 273,ydata);
% plot(ydata);
title('Experimental Data'); grid;
axis([xmin - 273, xmax - 273, ymin, ymax]);
subplot(2,1,2)
plot(xdata - 273, filtered_data);
% plot(filtered_data);
title('SG Filtered Data'); grid;
axis([xmin - 273, xmax - 273, ymin, ymax]);

figure
subplot(2,1,1)
plot(xdata(1:length(dydtdata)) - 273, dydtdata);
% plot( dydtdata,'k');
title('dA/dT Using Filtered Data'); grid;
axis([xmin - 273, xmax - 273, ymin2, ymax2]);
subplot(2,1,2)
plot(xdata(1:length(filtered_dydt)) - 273, filtered_dydt)
% plot(filtered_dydt,'k')
title('Filtered dA/dT Data'); grid;
axis([xmin - 273, xmax - 273, ymin2, ymax2]);
```

8. Van't Hoff Sum Source Code

The following source code is used to generate a series of van't Hoff plots, the values of which are summed to generate a complete melting profile based on the number of transitions provided.

```
% Pielak Paper van't Hoff Plot Generator
% Vincent Mao
% written: October 30, 2009
% last modified: January 18, 2010
% -------------------------------
% This program plots the van't Hoff plots for a series of separate
% thermodynamic regions based on melting temperatures and van't Hoff
% enthalpies defined by the user.
% The method uses the van't Hoff analysis as described in the paper
```

```
% from John and Weeks '00 "van't Hoff enthalpies without baselines".  The
% correction to their typo in terms of the temperature term is included.

%==================================
% Equations
%==================================

% rate constant :K = exp((Hvh/R*(1/Tm - 1./T)));
% fraction of denatured DNA: f = K ./(K+1);

% change of absorbance over temperature: dAbsdT = (A .* f .* (1-f))./T.^2;

% full expression by substituting all known equations into dAbs/dT
% v(1) = A = scaling factor, v(2) = Hvh = van't Hoff enthalpy, v(3) = Tm =
% melting temperature

close all;
clear;
clc;

% Constants for the van't Hoff enthalpy curves
format short g;
R   = 8.314;

% The empirical data fits these parameters.  This is based on a hypothesis
% for general cooperation in the interior arms of the 8 tile model during
% melting.  The rule is that free ends of arms have their Tm's averaged
% from the separate strands due to the crossover connection.  For interior
% core and arm strands, the Tm is that of the dsDNA strand.

% 4 Tile thermo region base pair sequence left to right, bottom to top.
% HvH = [ 8 8 11 11 10 10 13 13    13 13 10 10 11 11 8 8 ...
%         8 8 11 11 10 10 13 13    8 8 11 11 10 10 13 13 ...
%         13 13 10 10 11 11 8 8    13 13 10 10 11 11 8 8 ...
%         13 13 10 10 11 11 8 8    8 8 11 11 10 10 13 13 ...
%         5 5 5 5 5 5 5 5
%       ] * 8700 * 4.184;

% HvHa = [ 13 13 18 18 13 13 18 18 ] * 8700 * 4.184;
% HvHa = [ 10 10 10 10 10 10 10 10 ] * 8700 * 4.184;
% HvHa = [ 5 5 5 5 5 5 5 5 ] * 8700 * 4.184;

% 8 Tile thermo region base pair sequence left to right, bottom to top.
HvH = [ 8 8 11 11 10 10 13 13    13 13 10 10 11 11 8 8 ...
        8 8 11 11 10 10 13 13    8 8 11 11 10 10 13 13 ...
        13 13 10 10 11 11 8 8    13 13 10 10 11 11 8 8 ...
        13 13 10 10 11 11 8 8    8 8 11 11 10 10 13 13 ...
        8 8 11 11 10 10 13 13    13 13 10 10 11 11 8 8 ...
        8 8 11 11 10 10 13 13    8 8 11 11 10 10 13 13 ...
        13 13 10 10 11 11 8 8    13 13 10 10 11 11 8 8 ...
        13 13 10 10 11 11 8 8    8 8 11 11 10 10 13 13 ...
        5 5 5 5 5 5 5 5 5 5 ...
        5 5 5 5 5 5 5 5 5 5
      ] * 8700 * 4.184;
%  10 10 10 10 10 10 10 10 10 10
% HvH = [ 8 8 11 11 10 10 13 13    13 13 10 10 11 11 8 8 ...
%         8 8 11 11 10 10 13 13    8 8 11 11 10 10 13 13 ...
%         13 13 10 10 11 11 8 8    13 13 10 10 11 11 8 8 ...
%         13 13 10 10 11 11 8 8    8 8 11 11 10 10 13 13 ...
%         8 8 11 11 10 10 13 13    13 13 10 10 11 11 8 8 ...
%         8 8 11 11 10 10 13 13    8 8 11 11 10 10 13 13 ...
%         13 13 10 10 11 11 8 8    13 13 10 10 11 11 8 8 ...
%         13 13 10 10 11 11 8 8    8 8 11 11 10 10 13 13 ...
```

```
%         10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
%       ] * 8700 * 4.184;

% HvH = [ 8 8 11 11 10 10 18 18    18 18 10 10 11 11 8 8 ...
%          13 13 11 11 10 10 13 13    13 13 11 11 10 10 18 18 ...
%          13 13 10 10 11 11 13 13    18 18 10 10 11 11 13 13 ...
%          18 18 10 10 11 11 8 8    13 13 11 11 10 10 18 18 ...
%          8 8 11 11 10 10 18 18    18 18 10 10 11 11 13 13 ...
%          13 13 11 11 10 10 13 13    13 13 11 11 10 10 18 18 ...
%          13 13 10 10 11 11 13 13    13 13 10 10 11 11 13 13 ...
%          18 18 10 10 11 11 8 8    8 8 11 11 10 10 18 18 ...
%          5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
%       ] * 8700 * 4.184;

% CONTROL: Default Tm for each thermo region in an 8 Tile structure
% assuming independent thermodynamic regions
% Tm  = [ 30 35 48 51 45 49 55 65 ...
%          65 50 53 42 56 43 44 31 ...
%          30 35 48 51 45 49 55 65 ...
%          65 55 49 45 51 48 35 30 ...
%          65 55 49 45 51 48 35 30 ...
%          65 50 53 42 56 43 44 31 ...
%          65 55 49 45 51 48 35 30 ...
%          65 50 53 42 56 43 44 31 ...
%          30 35 48 51 45 49 55 65 ...
%          65 50 53 42 56 43 44 31 ...
%          30 35 48 51 45 49 55 65 ...
%          65 55 49 45 51 48 35 30 ...
%          65 55 49 45 51 48 35 30 ...
%          65 50 53 42 56 43 44 31 ...
%          65 55 49 45 51 48 35 30 ...
%          65 50 53 42 56 43 44 31 ...
%          13 13 23 18 18 16 10 9 11 11 ...
%          9 9 12 12 16 16 15 15 10 10
%        ] + 273;

% Tm  = [ 30 35 48 51 45 49 55 65 ...
%          65 50 53 42 56 43 44 31 ...
%          30 35 48 51 45 49 55 65 ...
%          65 55 49 45 51 48 35 30 ...
%          65 55 49 45 51 48 35 30 ...
%          65 50 53 42 56 43 44 31 ...
%          65 55 49 45 51 48 35 30 ...
%          65 50 53 42 56 43 44 31 ...
%          13 13 23 18 18 16 10 9
%        ] + 273;

% Core-Shell Interaction modification - Tm is based on sequence between
% poly-T segments of the core
% Tm  = [ 30 35 60.2 60.2 58 58 55 65 ...
%          65 50 58.3 58.3 62.3 62.3 44 31 ...
%          30 35 60.2 60.2 58 58 55 65 ...
%          44 31 62.3 62.3 58.3 58.3 65 50 ...
%          55 65 58 58 60.2 60.2 30 35 ...
%          65 50 58.3 58.3 62.3 62.3 44 31 ...
%          55 65 58 58 60.2 60.2 30 35 ...
%          44 31 62.3 62.3 58.3 58.3 65 50 ...
%          30 35 60.2 60.2 58 58 55 65 ...
%          65 50 58.3 58.3 62.3 62.3 44 31 ...
%          30 35 60.2 60.2 58 58 55 65 ...
%          44 31 62.3 62.3 58.3 58.3 65 50 ...
%          55 65 58 58 60.2 60.2 30 35 ...
%          65 50 58.3 58.3 62.3 62.3 44 31 ...
```

```
%         55 65 58 58 60.2 60.2 30 35 ...
%         44 31 62.3 62.3 58.3 58.3 65 50 ...
%         13 13 23 18 18 16 10 9 11 11 ...
%         9 9 12 12 16 16 15 15 10 10
%        ] + 273;

% Tm  = [ 30 35 60.2 60.2 58 58 55 65 ...
%         65 50 58.3 58.3 62.3 62.3 44 31 ...
%         30 35 60.2 60.2 58 58 55 65 ...
%         44 31 62.3 62.3 58.3 58.3 65 50 ...
%         55 65 58 58 60.2 60.2 30 35 ...
%         65 50 58.3 58.3 62.3 62.3 44 31 ...
%         55 65 58 58 60.2 60.2 30 35 ...
%         44 31 62.3 62.3 58.3 58.3 65 50 ...
%         26 41 34 19  ...
%         26 41 34 19
%        ] + 273;

% Core-Shell/Shell-Arm Interaction 1 - Tm for outer regions based on avg of
% two neighboring thermodynamic regions
% Tm  = [ 43 39 60.2 60.2 58 58 50 55 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         43 39 60.2 60.2 58 58 50 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         43 39 60.2 60.2 58 58 50 55 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         43 39 60.2 60.2 58 58 50 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         13 13 23 18 18 16 10 9 11 11 ...
%         9 9 12 12 16 16 15 15 10 10
%        ] + 273;
%
% Tm  = [ 43 39 60.2 60.2 58 58 50 55 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         43 39 60.2 60.2 58 58 50 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         13 13 23 18 18 16 10 9 ...
%        ] + 273;

% Core-Shell/Shell-Arm Interaction 1/Sticky End Sum - Tm for for sticky
% ends are also summed
% Tm  = [ 43 39 60.2 60.2 58 58 50 55 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         43 39 60.2 60.2 58 58 50 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         43 39 60.2 60.2 58 58 50 55 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
```

```
%         43 39 60.2 60.2 58 58 50 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         26 41 34 19 22 18 24 32 30 20 ...
%         26 41 34 19 22 18 24 32 30 20
%        ] + 273;

% Tm  = [ 43 39 60.2 60.2 58 58 50 55 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         43 39 60.2 60.2 58 58 50 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         26 41 34 19 26 41 34 19
%        ] + 273;

% Core-Shell/Shell-Arm Interaction 2 - Tm for outer regions based on avg of
% the core Tm and neighboring thermodynamic regions
Tm  = [ 47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
        61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
        47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
        53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
        56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
        61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
        56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
        53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
        47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
        61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
        47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
        53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
        56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
        61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
        56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
        53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
        49.8 45.9 42.5 45 48.8 47.5 42.7 40.2 ...
        44.8 43.5 42.1 43.5 41.9 44.4 46.8 48.8 ...
        47.5 46.1 40.5 43
       ] + 273;
%    26 41 34 19 22 18 24 32 30 20 ...
% 13 13 23 18 18 16 10 9 11 11 ...
%          9 9 12 12 16 16 15 15 10 10
% Tm  = [ 47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 47.6 ...
%         47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 47.6 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         49.8 45.9 42.5 45 48.8 47.5 42.7 40.2
%        ] + 273;
%    43.8 43.5 38.2 40.7 43.5 41.5 39.7 36.9
%         26 26 41 41 34 34 19 19
% Core-Shell/Shell-Arm Interaction 2/Arm-Arm Interaction 1 - Tm for outer
% regions based on avg of the core Tm and neighboring thermodynamic regions
% Tm  = [ 47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
```

```
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         42.5 50 48 52 52 48 50 42.5 52 48 ...
%         48 52 48 52 52 48 50 42.5 50 42.5
%        ] + 273;

% Core/Shell/Shell-Arm/Arm-Arm Interaction 1 - Tm for arm regions between
% tiles are averaged
% Tm  = [ 43 39 60.2 60.2 58 58 50 55 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         43 39 60.2 60.2 58 58 50 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         43 39 60.2 60.2 58 58 50 55 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         43 39 60.2 60.2 58 58 50 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         42.5 50 48 52 52 48 50 42.5 52 48 ...
%         48 52 48 52 52 48 50 42.5 50 42.5
%        ] + 273;

% Core/Shell/Shell-Arm/Arm-Arm Interaction 2 - Tm for arm regions between
% tiles are averaged
% Tm  = [ 47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         42.5 50 48 52 52 48 50 42.5 52 48 ...
%         48 52 48 52 52 48 50 42.5 50 42.5
%        ] + 273;
   %-------------------------------------------------------------------
% Avg Tm for free ends and sticky ends
% Tm  = [ 32.5 32.5 60.2 60.2 58 58 64.4 64.4 ...
```

```
%         66.7 66.7 58.3 58.3 62.3 62.3 37.5 37.5 ...
%         52.2 52.2 60.2 60.2 58 58 60 60 ...
%         56.8 56.8 62.3 62.3 58.3 58.3 62.5 62.5 ...
%         60 60 58 58 60.2 60.2 52.2 52.2 ...
%         66.7 66.7 58.3 58.3 62.3 62.3 56.8 56.8 ...
%         64.4 64.4 58 58 60.2 60.2 32.5 32.5 ...
%         56.8 56.8 62.3 62.3 58.3 58.3 66.7 66.7 ...
%         32.5 32.5 60.2 60.2 58 58 64.4 64.4 ...
%         66.7 66.7 58.3 58.3 62.3 62.3 56.8 56.8 ...
%         52.2 52.2 60.2 60.2 58 58 60 60 ...
%         56.8 56.8 62.3 62.3 58.3 58.3 66.7 66.7 ...
%         60 60 58 58 60.2 60.2 52.2 52.2 ...
%         62.5 62.5 58.3 58.3 62.3 62.3 56.8 56.8 ...
%         64.4 64.4 58 58 60.2 60.2 32.5 32.5 ...
%         37.5 37.5 62.3 62.3 58.3 58.3 66.7 66.7 ...
%         20.5 13 9.5 17 11 9 12 16 15 10 ...
%         20.5 13 9.5 17 11 9 12 16 15 10
%        ] + 273;

% Treat the free ends as one arm sequence (for XL model)
% Tm  = [ 52.2 52.2 60.2 60.2 58 58 64.4 64.4 ...
%         66.7 66.7 58.3 58.3 62.3 62.3 56.8 56.8 ...
%         52.2 52.2 60.2 60.2 58 58 64.4 64.4 ...
%         56.8 56.8 62.3 62.3 58.3 58.3 66.7 66.7 ...
%         64.4 64.4 58 58 60.2 60.2 52.2 52.2 ...
%         66.7 66.7 58.3 58.3 62.3 62.3 56.8 56.8 ...
%         64.4 64.4 58 58 60.2 60.2 52.2 52.2 ...
%         56.8 56.8 62.3 62.3 58.3 58.3 66.7 66.7 ...
%         52.2 52.2 60.2 60.2 58 58 64.4 64.4 ...
%         66.7 66.7 58.3 58.3 62.3 62.3 56.8 56.8 ...
%         52.2 52.2 60.2 60.2 58 58 64.4 64.4 ...
%         56.8 56.8 62.3 62.3 58.3 58.3 66.7 66.7 ...
%         64.4 64.4 58 58 60.2 60.2 52.2 52.2 ...
%         66.7 66.7 58.3 58.3 62.3 62.3 56.8 56.8 ...
%         64.4 64.4 58 58 60.2 60.2 52.2 52.2 ...
%         56.8 56.8 62.3 62.3 58.3 58.3 66.7 66.7 ...
%         20.5 13 9.5 17 11 9 12 16 15 10 ...
%         20.5 13 9.5 17 11 9 12 16 15 10
%        ] + 273;

% Tm for free ends are separate
% Tm  = [ 35 30 60.2 60.2 58 58 55 65 ...
%         65 60 58.3 58.3 62.3 62.3 44 31 ...
%         30 35 60.2 60.2 58 58 65 55 ...
%         44 31 62.3 62.3 58.3 58.3 65 60 ...
%         55 65 58 58 60.2 60.2 35 30 ...
%         65 60 58.3 58.3 62.3 62.3 31 44 ...
%         65 55 58 58 60.2 60.2 30 35 ...
%         44 31 62.3 62.3 58.3 58.3 60 65 ...
%         35 30 60.2 60.2 58 58 55 65 ...
%         65 60 58.3 58.3 62.3 62.3 44 31 ...
%         30 35 60.2 60.2 58 58 65 55 ...
%         44 31 62.3 62.3 58.3 58.3 65 60 ...
%         55 65 58 58 60.2 60.2 35 30 ...
%         65 60 58.3 58.3 62.3 62.3 31 44 ...
%         65 55 58 58 60.2 60.2 30 35 ...
%         44 31 62.3 62.3 58.3 58.3 60 65 ...
%         20.5 13 9.5 17 11 9 12 16 15 10 ...
%         20.5 13 9.5 17 11 9 12 16 15 10
%        ] + 273;

% Avg across neighboring thermo regions
% Tm  = [ 43 39 60.2 60.2 58 58 50 55 ...
```

```
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         43 39 60.2 60.2 58 58 50 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         43 39 60.2 60.2 58 58 50 55 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         43 39 60.2 60.2 58 58 50 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         20.5 13 9.5 17 11 9 12 16 15 10 ...
%         20.5 13 9.5 17 11 9 12 16 15 10
%        ] + 273;

% Avg across neighboring thermo regions with core averaged
% Tm  = [ 43 39 59.7 59.7 59.7 59.7 50 55 ...
%         59 51 59.7 59.7 59.7 59.7 43.5 43.5 ...
%         43 39 59.7 59.7 59.7 59.7 50 55 ...
%         43.5 43.5 59.7 59.7 59.7 59.7 59 51 ...
%         50 55 59.7 59.7 59.7 59.7 43 39 ...
%         59 51 59.7 59.7 59.7 59.7 43.5 43.5 ...
%         50 55 59.7 59.7 59.7 59.7 43 39 ...
%         43.5 43.5 59.7 59.7 59.7 59.73 59 51 ...
%         43 39 59.7 59.7 59.7 59.7 50 55 ...
%         59 51 59.7 59.7 59.7 59.7 43.5 43.5 ...
%         43 39 59.7 59.7 59.7 59.7 50 55 ...
%         43.5 43.5 59.7 59.7 59.7 59.7 59 51 ...
%         50 55 59.7 59.7 59.7 59.7 43 39 ...
%         59 51 59.7 59.7 59.7 59.7 43.5 43.5 ...
%         50 55 59.7 59.7 59.7 59.7 43 39 ...
%         43.5 43.5 59.7 59.7 59.7 59.73 59 51 ...
%         20.5 13 9.5 17 11 9 12 16 15 10 ...
%         20.5 13 9.5 17 11 9 12 16 15 10
%        ] + 273;

% Avg across neighboring thermo regions for interior arms only
% Tm  = [ 35 30 60.2 60.2 58 58 50 55 ...
%         59 51 58.3 58.3 62.3 62.3 44 31 ...
%         43 39 60.2 60.2 58 58 65 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 65 60 ...
%         55 65 58 58 60.2 60.2 43 39 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 60.2 60.2 35 30 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         35 30 60.2 60.2 58 58 50 55 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         43 39 60.2 60.2 58 58 65 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         55 65 58 58 60.2 60.2 43 39 ...
%         65 60 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 60.2 60.2 35 30 ...
%         44 31 62.3 62.3 58.3 58.3 59 51 ...
%         20.5 13 9.5 17 11 9 12 16 15 10 ...
%         20.5 13 9.5 17 11 9 12 16 15 10
%        ] + 273;

% Avg across neighboring thermo regions for interior core and arms
% Tm  = [ 35 30 51 48 58 58 50 55 ...
```

```
%         59 51 58.3 58.3 43 56 44 31 ...
%         43 39 60.2 60.2 45 49 65 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 65 60 ...
%         55 65 45 49 60.2 60.2 43 39 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 51 48 35 30 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         35 30 51 48 58 58 50 55 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         43 39 60.2 60.2 45 49 65 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         55 65 45 49 60.2 60.2 43 39 ...
%         65 60 53 42 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 51 48 35 30 ...
%         44 31 43 56 58.3 58.3 59 51 ...
%         20.5 13 9.5 17 11 9 12 16 15 10 ...
%         20.5 13 9.5 17 11 9 12 16 15 10
%        ] + 273;

% Avg across crossover regions
% Tm  = [ 41.5 40.5 60.2 60.2 58 58 52 55 ...
%         54.5 56.5 58.3 58.3 62.3 62.3 55 37 ...
%         41.5 40.5 60.2 60.2 58 58 52 55 ...
%         55 37 62.3 62.3 58.3 58.3 54.5 56.5 ...
%         52 55 58 58 60.2 60.2 41.5 40.5 ...
%         54.5 56.5 58.3 58.3 62.3 62.3 55 37 ...
%         52 55 58 58 60.2 60.2 41.5 40.5 ...
%         55 37 62.3 62.3 58.3 58.3 54.5 56.5 ...
%         41.5 40.5 60.2 60.2 58 58 52 55 ...
%         54.5 56.5 58.3 58.3 62.3 62.3 55 37 ...
%         41.5 40.5 60.2 60.2 58 58 52 55 ...
%         55 37 62.3 62.3 58.3 58.3 54.5 56.5 ...
%         52 55 58 58 60.2 60.2 41.5 40.5 ...
%         54.5 56.5 58.3 58.3 62.3 62.3 55 37 ...
%         52 55 58 58 60.2 60.2 41.5 40.5 ...
%         55 37 62.3 62.3 58.3 58.3 54.5 56.5 ...
%         26 41 34 19 22 18 24 32 30 20 ...
%         26 41 34 19 22 18 24 32 30 20
%        ] + 273;

% Avg the core Tms with the arms
% Tm  = [ 47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%        ] + 273;

% Avg across crossover regions for interior arms only
% Tm  = [ 35 30 60.2 60.2 58 58 52 55 ...
%         54.5 56.5 58.3 58.3 62.3 62.3 44 31 ...
%         41.5 40.5 60.2 60.2 58 58 55 65 ...
```

```
%         55 37 62.3 62.3 58.3 58.3 65 60 ...
%         55 65 58 58 60.2 60.2 41.5 40.5 ...
%         54.5 56.5 58.3 58.3 62.3 62.3 55 37 ...
%         52 55 58 58 60.2 60.2 35 30 ...
%         55 37 62.3 62.3 58.3 58.3 54.5 56.5 ...
%         35 30 60.2 60.2 58 58 52 55 ...
%         54.5 56.5 58.3 58.3 62.3 62.3 55 37 ...
%         41.5 40.5 60.2 60.2 58 58 35 30 ...
%         55 37 62.3 62.3 58.3 58.3 54.5 56.5 ...
%         52 55 58 58 60.2 60.2 35 30 ...
%         65 60 58.3 58.3 62.3 62.3 55 37 ...
%         52 55 58 58 60.2 60.2 35 30 ...
%         44 31 62.3 62.3 58.3 58.3 54.5 56.5 ...
%         26 41 34 19 22 18 24 32 30 20 ...
%         26 41 34 19 22 18 24 32 30 20
%        ] + 273;

% Avg the core Tms with the interior arms only
% Tm  = [ 35 30 60.2 60.2 58 58 56.5 61.5 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 44 31 ...
%         47.6 45.1 60.2 60.2 58 58 55 65 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 65 60 ...
%         65 55 58 58 60.2 60.2 47.6 45.1 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         56.5 61.5 58 58 60.2 60.2 35 30 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         35 30 60.2 60.2 58 58 56.5 61.5 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         47.6 45.1 60.2 60.2 58 58 55 65 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         65 55 58 58 60.2 60.2 47.6 45.1 ...
%         65 60 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         56.5 61.5 58 58 60.2 60.2 35 30 ...
%         44 31 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         20.5 13 9.5 17 11 9 12 16 15 10 ...
%         20.5 13 9.5 17 11 9 12 16 15 10
%        ] + 273;

% Avg neighboring regions and avg the sticky ends with arms
% Tm  = [ 43 39 60.2 60.2 58 58 50 55 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         43 39 60.2 60.2 58 58 50 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         43 39 60.2 60.2 58 58 50 55 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         43 39 60.2 60.2 58 58 50 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         32.7 37.7 39 40.7 40.7 37.3 36.7 31.3 38.4 35 ...
%         35 37.7 32.3 37.3 37.3 40 39.7 37 36.7 31.7
%        ] + 273;

% Avg the core Tms with the arms with avg sticky ends with arms
% Tm  = [ 47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
```

```
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         32.7 37.7 39 40.7 40.7 37.3 36.7 31.3 38.4 35 ...
%         35 37.7 32.3 37.3 37.3 40 39.7 37 36.7 31.7
%        ] + 273;

% Avg the core Tms with the arms with avg summed sticky ends with arms
% Tm  = [ 47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         56.5 61.5 58 58 60.2 60.2 47.6 45.1 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         49.8 45.9 42.5 45 48.8 47.5 42.7 40.2 ...
%         44.8 43.5 42.1 43.5 41.9 44.4 46.8 48.8 ...
%         47.5 46.1 43 40.5
%        ] + 273;
% 37 42 45.7 48.3 46 43.3 39.7 34.7 42 39.3 ...
%         38 40.7 36.3 41.3 42.7 45.3 39.7 42 40 35
% Avg the core Tms with the interior arms only with avg summed sticky ends
% with the arms
% Tm  = [ 35 30 60.2 60.2 58 58 56.5 61.5 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 44 31 ...
%         47.6 45.1 60.2 60.2 58 58 55 65 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 65 60 ...
%         65 55 58 58 60.2 60.2 47.6 45.1 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         56.5 61.5 58 58 60.2 60.2 35 30 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         35 30 60.2 60.2 58 58 56.5 61.5 ...
%         61.7 59.2 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         47.6 45.1 60.2 60.2 58 58 55 65 ...
%         53.2 42.7 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         65 55 58 58 60.2 60.2 47.6 45.1 ...
%         65 60 58.3 58.3 62.3 62.3 53.2 42.7 ...
%         56.5 61.5 58 58 60.2 60.2 35 30 ...
%         44 31 62.3 62.3 58.3 58.3 61.7 59.2 ...
%         37 42 45.7 48.3 46 43.3 39.7 34.7 42 39.3 ...
%         38 40.7 36.3 41.3 42.7 45.3 39.7 42 40 35
%        ] + 273;

% Avg across neighboring thermo regions for interior arms only
```

```
% Tm  = [ 35 30 60.2 60.2 58 58 50 55 ...
%         59 51 58.3 58.3 62.3 62.3 44 31 ...
%         43 39 60.2 60.2 58 58 65 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 65 60 ...
%         55 65 58 58 60.2 60.2 43 39 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 60.2 60.2 35 30 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         35 30 60.2 60.2 58 58 50 55 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         43 39 60.2 60.2 58 58 65 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         55 65 58 58 60.2 60.2 43 39 ...
%         65 60 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 60.2 60.2 35 30 ...
%         44 31 62.3 62.3 58.3 58.3 59 51 ...
%         37 42 45.7 48.3 46 43.3 39.7 34.7 42 39.3 ...
%         38 40.7 36.3 41.3 42.7 45.3 39.7 42 40 35
%        ] + 273;

% Avg the arm Tm's for the sticky ends
% Avg neighboring regions and avg the sticky ends with arms
% Tm  = [ 43 39 60.2 60.2 58 58 50 55 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         43 39 60.2 60.2 58 58 50 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         43 39 60.2 60.2 58 58 50 55 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         43 39 60.2 60.2 58 58 50 55 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ...
%         50 55 58 58 60.2 60.2 43 39 ...
%         43.5 43.5 62.3 62.3 58.3 58.3 59 51 ...
%         42.5 50 48 52 52 48 50 42.5 52 48 ...
%         48 52 48 52 52 48 50 42.5 50 42.5
%        ] + 273;


% % Ideal case for a tile (matches the experimental data)
% HvH = [ 13 13 13 13 13 13 13 13 ...
%     13 13 13 13 13 13 13 13 19 ] * 8700 * 4.184;
% Tm  = [36 39 42 46 48 50 52 58 ...
%     58 58 58 57 57 57 57 57 69 ] + 273;
%
% % Using the current bp's in the thermo regions with the Tm's
% HvH = [ 8 8 11 11 10 10 13 13 ...
%         13 13 10 10 11 11 8 8 ] * 8700 * 4.184;
% Tm  = [ 36 39 58 58 48 50 58 57  ...
%         57 57 52 58 57 57 42 46] + 273;
% Tm  = [ 30 35 48 51 49 45 65 55 ...
%         65 60 53 42 43 56 44 31 ] + 273;
% Shell dominant Tm theory for the 1 Tile melt
% Tm  = [ 30 35 30 35 42 31 42 31 ...
%         30 42 30 42 35 31 35 31 ] + 273;

% Avg neighboring regions
% HvH = [ 8 8 11 11 10 10 13 13 ...
%         13 13 10 10 11 11 8 8 ] * 8700 * 4.184;
```

```
% Tm  = [ 43 39 60.2 60.2 58 58 50 55 ...
%         59 51 58.3 58.3 62.3 62.3 43.5 43.5 ] + 273;

% Avg the core but leave the outer arms Tm
% Tm  = [ 35 30 60.2 60.2 58 58 55 65 ...
%         65 60 58.3 58.3 62.3 62.3 44 31 ] + 273;

% Avg all neighboring regions
% HvH = [ 8 8 11 11 10 10 13 13 ...
%         13 13 10 10 11 11 8 8 ] * 8700 * 4.184;
% Tm  = [ 46.9 44.4 55.6 54.1 51.5 53.5 60.4 65.4 ...
%         68.4 65.9 55.7 50.2 52.7 59.2 51.1 44.6 ] + 273;

% Arm regions treated as one sequence
% Tm  = [52.2 52.2 60.2 60.2 58 58 64.4 64.4 ...
%     66.7 66.7 58.3 58.3 62.3 62.3 56.8 56.8  ] + 273;

% Avg across crossover regions
% Tm  = [ 41.5 40.5 60.2 60.2 58 58 52 55 ...
%         54.5 56.5 58.3 58.3 62.3 62.3 55 37 ] + 273;

% Avg core with neighbor region tile (BEST THEORY)
% Tm = [47.6 45.1 60.2 60.2 58 58 56.5 61.5 ...
%     61.7 59.2 58.3 58.3 62.3 62.3 53.2 46.7 ] + 273;

% Ta = [31 28.3 47 40 31 31 37.7 42.3] + 273;
% Ta = [26 26 41 41 34 34 19 19] + 273;
% Ta = [43.8 43.5 38.2 40.7 43.5 41.5 39.7 36.9] + 273;
% Ta = [49.8 45.9 42.5 45 48.8 47.5 42.7 40.2] + 273;
% Ta = [54.2 56.2 50.8 54.6 56.2 54.2 54.6 50.8] + 273;
A = 1;

% Load simulation data and empircal data

% xdata = xlsread('1TTests.xls', 5, 'A4:A240') +  273;
% ydata = xlsread('1TTests.xls', 5, 'D4:D240');

% xdata = xlsread('1TTestsOpt.xls', 2, 'A39:A190') +  273;
% ydata = xlsread('1TTestsOpt.xls', 2, 'C39:C190');

% xdata = xlsread('NSTests.xls', 14, 'A100:A940');
% ydata = xlsread('NSTests.xls', 14, 'J100:J940');

xdata2 = xlsread('NSTests2.xls', 4, 'A400:A999');
ydata2 = xlsread('NSTests2.xls', 4, 'K400:K999');

% xdata = xlsread('SimpleMeltOutput1k.xls', 16, 'A1:A940');
% ydata = xlsread('SimpleMeltOutput1k.xls', 16, 'J1:J940');

% xdata = xlsread('ComplexMeltOutput1k.xls', 13, 'A1:A870');
% ydata = xlsread('ComplexMeltOutput1k.xls', 13, 'J1:J870');

% xdata = xlsread('8TTests.xls', 9, 'A60:A870');
% ydata = xlsread('8TTests.xls', 9, 'J60:J870');

% Avg Control Melt Data
xdata = xlsread('CtlMelts.xls', 5, 'A5:A242')+273;
ydata = xlsread('CtlMelts.xls', 5, 'H5:H242');

% xdata2 = xlsread('CtlMelts.xls', 10, 'A29:A252')+273;
% ydata2 = xlsread('CtlMelts.xls', 10, 'C29:C252');
```

```
% Avg XL Melt Data
% xdata2 = xlsread('XLMelts.xls', 5, 'D9:D217');
% ydata2 = xlsread('XLMelts.xls', 5, 'G9:G217');

% xdata2 = xlsread('XLMelts.xls', 9, 'A29:A238');
% ydata2 = xlsread('XLMelts.xls', 9, 'F29:F238');

% Determine temperature range
lowTemp     =   253;
highTemp    =   353;
tempRes     =   10;
tempRange = highTemp - lowTemp;
tempInc = tempRange / (tempRange * tempRes);
currentTemp = lowTemp:tempInc:highTemp;

% Keep a running sum of all the van't Hoff enthalpy curves
vantHoffTotal = zeros(1,length(currentTemp));
for g = 1:length(Tm)
    for i = 1:length(currentTemp)
        Km = exp( HvH(g) / R * (1/Tm(g) - 1./currentTemp) );
        DNAfrac = Km ./ (Km + 1);
        vantHoff = ( A .* DNAfrac .* (1 - DNAfrac) ) ./ (currentTemp .^ 2);
        vantHoffTotal = vantHoffTotal + vantHoff;
    end
end

% for g = 1:length(Ta)
%     for i = 1:length(currentTemp)
%         Km = exp( HvHa(g) / R * (1/Ta(g) - 1./currentTemp) );
%         DNAfrac = Km ./ (Km + 1);
%         vantHoff = -( A .* DNAfrac .* (1 - DNAfrac) ) ./ (currentTemp .^ 2);
%         vantHoffTotal = vantHoffTotal + vantHoff;
%     end
% end

% Normalize the data
maxy            = max(ydata);
normFactory     = 1 / maxy;
maxData         = max(vantHoffTotal);
normFactorData  = 1 / maxData;
maxy2           = max(ydata2);
normFactory2    = 1 / maxy2;


% Convolve the data from the experimental and the simulation
% correl_data = conv(vantHoffTotal * normFactorData, ydata2 * normFactory2);
% plot(correl_data,'b')
% ylabel('Convolution of Normalized dAbs/dT');
% title(['Convolution of van''t Hoff Plots of Experimental and Simulated Data']);
% legend('Convolved Data');

% Plot the data
figure;
hold on
plot(xdata - 273, ydata * normFactory,'--k')
plot(xdata2 - 273, ydata2 * normFactory2,'-.k')
plot(currentTemp - 273, vantHoffTotal * normFactorData, 'k')

xlabel('Temp(\circC)');
ylabel('Normalized dA/dT');
title(['van''t Hoff Plot']);
legend('Experimental Melt', ['Simulation'], ['vant Hoff sum']);
% legend('Simulation', ['Experiment'], ['vant Hoff sum']);
```

```
% legend( ['Simulation'],['vant Hoff sum']);
% legend( ['Experiment'],['Simulation']);
```

9. The Graphical Output Diagnostic Program

As the complexity of the model increases, it becomes increasingly time-consuming to run the thousands of trajectories needed to generate meaningful data for analysis. A visualization and coordinate recording function for the simulator is a diagnostic tool that can be used to study the physical features of the models over the course of the simulation after the simulation is completed. The program requires the input mesh file to initialize the mass node and spring properties along with the snapshot data file. During simulation, the mass coordinates for each mass node in the model can be recorded and stored in a file for a separate C program to replay for additional graphical output. The appropriate flags for recording the mass node coordinates must be entered on the command prompt for constant temperature mode or melting mode. The output file contains the temperature at which each "snapshot" of the mass node positions are recorded, followed by the coordinates. The graphics program, implemented in OpenGL, displays the positions for each snapshot. The absorbance of the model is also calculated. The "U" identifier line that defines the resolution of the temperature increment to record the data also defines the temperature range over which to replay the simulation. Table 17 presents the command flags used in this program.

Table 17: Command flags for the graphical program.

Flag	Description
h	Show all possible command flags
m	Display mesh file data
r	Read the mesh file for first snapshot
p	Play the entire video
d	Replay the simulation over a specific temperature range
c	Constant temperature simulation mode
e	Export absorbance data to file
f	Specify input and output files

In Table 17, the first three flags (h, m, r) are for diagnostic purposes, the next three flags (p, d, c) determine simulation settings and the last two flags (e, f) determine data collection and data output. These flags, like in the DNA-STRAIN simulator, must be set in order for the program to run properly. The "-h" flag and "-m" flags serve the same function as previously described. The "-r" flag indicates that the program will scan through the mesh file and output the first snapshot. This flag is for checking that the input mesh file and the snapshot file match for the replay. For the simulation setting flags, the "-p" flag is used to play the entire trajectory frame by frame in one second intervals. This flag can be used with the "-d" flag to replay a video over a specific temperature range. The "-c" flag is used to indicate that the mass trajectory file is for constant temperature simulation. In the data input and output flags, the "-e" flag is used to run the entire trajectory while calculating the absorbance for each temperature or time increment, exporting the values to a data file. The "-f" flag allows the user to

specify both the mesh file and the mass trajectory file names from which the graphical program generates the video.

The analytical and diagnostic tools described in this section aid in validating the DNA-STRAIN simulator. The input file generator is used to create the input files for the 3D DNA models. The post processing tools are used to filter the data and extract the thermodynamic parameters needed for analysis and comparison to experimental data. The graphical output program aids in observing the physical behavior of the DNA nanostructure and is useful for modifying the 3D spring network models because it aids in visual verification of the proper connections during the construction process, serving as a means for validating the input file generator code.

10. Source Code for the Graphical Output Program

Main source code:

```
//--------------------------//
//      Spring Model Component  //
//      written: 11.2.09              //
//      last edited: 12.29.09  //
//  author: Vincent Mao               //
//--------------------------//

/*      This program reads a meshfile that defines springs and masses,
        over a defined temperature range.  The data is used to output
        a graphical image of the entire 3D spring structure.
        The program also has to read the mass coordinates that are used
        to compile the graphical cartoon and assigns each line of
        coordinates to the appropriate mass position for updating until
        it reaches the end of the file.  It also determines the absorbance
        for each point and can store it in a file based on trajectories.
*/

#define _CRT_SECURE_NO_DEPRECATE      // Turn off deprecation warnings in VS05
(NO_WARNINGS VS08)

// PPD's
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <math.h>
#include <time.h>
#include "meshcomp.h"
#include "graphics.h"
#include "glut.h"

#define BPMASS         0.660          // Mass of a base pair in Daltons (g/L)-->kg/mol

typedef unsigned char BYTE;
BYTE *g_pFrameData     = NULL;

double tempStart       =       0;
double tempStop                =       0;
double tempStartCom =  0;
double tempStopCom  =  0;
double timeStart       =       0;
double tempRange;
double timeRange;
double targetTemp      =       0;
double targetTemp1     =       0;
double currentTemp;
double currentTime;
double tempInc;
double timeInc;
double concentration;
double temp                    =       0;
double timePos         =       0;

// Variables from other functions
extern int massCount;
extern int totalSpringCount;
extern int springCount;
extern int segCount;

// Simulation variables
double currentTime     = 0;           // The current simulation time point
double simTime         = 5e-8;        // Simulation time in seconds
double concentration= 625E-8; // Sample concentration
int dataRes                    = 100;         // Data resolution; record every nth point
int targetRes          = 2;           // Snapshot resolution; record every nth temperature

double angleFreeX      = 0;
double angleFreeY      = 0;
double angleX          = 0;
double angleY          = 0;
double zoomInc         = 0.01;
double screenSize      = 0.025;

double posX                    = 0;
double posY                    = 0;

// This initializes the dimensions of the array from the source file
double massArray[2000000][6]  = [118];
double watch           = 0;
double minAngleInc     = 0;
double angleInc                = 0;
double angleIncY       = 0;
double maxAngleInc     = 1;

// Array counters
int masses             = 0;
int pos                = 0;
```

```
int maxpos             = 0;
int tempPoints = 0;
int timePoints = 0;
int frames             = 0;

// Initialize command flags
int nameFiles          = 0;
int duration           = 0;
int meshOutput         = 0;
int playBack           = 0;
int exportData         = 0;
int readMesh           = 0;
int constTemp          = 0;

// File names
char srcFile[50];
char snapFile[50];

MASS_node      *mh = NULL;                  /* Points to first node in mass list */
SPRING_node    *sh = NULL;                  /* Points to first node in spring list */
MASS_node *headpointer = NULL; /* Points to reference node in the mass list */

FILE *meshFile = NULL;
FILE *massFile = NULL;
FILE *dataFile = NULL;

// Function Prototypes
int loadMeshFile(void);
int loadSnapFile(void);
int initSettings(void);
int assignArrayCoords(int pos, double x, double y, double z);
void display_func(void);
void processNormalKeys(unsigned char key, int x, int y);
void processSpecialKeys(int key, int x, int y);
void processMouse(int button, int state, int x, int y);
void processMouseEntry(int state);
void processMousePassiveMotion(int x, int y);
void processMouseActiveMotion(int x, int y);

int initAbsorbance(SPRING_node *sh);
int initPos();
int updateFrac(SPRING_node *sh);
int updateSpringLength(SPRING_node *sh);
int updateAbsorbance(SPRING_node *sh);
int refSpringSearch(SPRING_node *sh);
SPRING_node *DNASpringSearch(SPRING_node *refsh, int currentSeg);
SPRING_node *braceSpringSearch(SPRING_node *refsh, int currentSeg, int currentRef);
SPRING_node *stiffSpringSearch(SPRING_node *refsh, int currentSeg);


/* The main function is where the temperature/time is first set
   and the temperature/time increment is calculated.  This
   is the only point where the temperature/time should be
   modified.  The value is then passed to all other functions
   for calculation purposes.
*/
int main(int argc, char *argv[])
{
        // Search for command line paramters
        int i = 0;
        int comPos1 = 0;
        int comPos2 = 0;
```

```
	char *msg = NULL;
	char fchr;

	// Check for command line parameters
	if(argc == 1){
		printf("ERROR: No command parameters found.  Type \"-h\" for a list of
commands...\n");
		return(-1);
	}
	else{
		// Identify command line parameters
		for(i; i < argc; i++)
		{
			if(argv[i][0] != '-'){
				continue;
			}

			fchr = argv[i][1];

			switch(fchr)
			{
				case 'h':
					printf("List of possible command flags: \n \
-f: Specify source file name <src>\n \
-d: Specify which part of the sim to replay <start temp> <stop temp>\n \
-m: Output mesh file data\n \
-p: Enable playback of the video frame by frame \n \
-e: Export absorbance data to file\n \
-r: Read the mesh file only\n \
-c: Constant temperature\n");
					return(-1);
					break;

				case 'f':
					msg = "Source files specified...\n";
					nameFiles = 1;
					comPos1 = i;
					break;

				case 'd':
					msg = "Replaying part of the sim...\n";
					duration = 1;
					comPos2 = i;
					break;

				case 'm':
					msg = "Displaying mesh file data...\n";
					meshOutput = 1;
					break;

				case 'p':
					msg = "Playing video...\n";
					playBack = 1;
					break;

				case 'e':
					msg = "Exporting data to ""abs.txt""...\n";
					exportData = 1;
					break;

				case 'r':
					msg = "Reading only the initial settings for model
checking...\n";
```

```
                                readMesh = 1;
                                break;

                        case 'c':
                                msg = "Constant temperature simulation replay...\n";
                                constTemp = 1;
                                break;
                }
        }
        if(msg){
                printf(msg);
        }
}

// Check for errors (does not detect incorrect input types)
if(nameFiles){
        if( (argc - (comPos1 + 2)) < 1){
                printf("ERROR: Missing file input parameters...\n");
                return(-1);
        }
        else{
                strcpy(srcFile, argv[comPos1 + 1]);
                strcpy(snapFile, argv[comPos1 + 2]);
        }
}
else{
        // Default file names
        strcpy(srcFile, "Mesh.txt");
        strcpy(snapFile, "snap.txt");
}

if(duration){
        if( (argc - (comPos2 + 2)) < 1){
                printf("ERROR: Missing temperature input parameters...\n");
                return(-1);
        }
        else{
                tempStartCom    =       atof(argv[comPos2 + 1]);
                tempStopCom             =       atof(argv[comPos2 + 2]);
        }
}

if(exportData){
        dataFile = fopen("abs.txt", "w");
}

// Load the mesh file

loadMeshFile();

maxpos = massCount;
if(!readMesh){
        // Read the snapshot file and initialize pointer position
        loadSnapFile();
        mh = headpointer;

        // Initialize values and springs for the replay
        initSettings();
        refSpringSearch(sh);
        initAbsorbance(sh);
}

//init_graphics("Mass-Spring DNA Nanostructure Test", display_func);
```

```
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
        glutInitWindowPosition(100,100);
        glutInitWindowSize(320,320);
        glutCreateWindow("Mass-Spring DNA Nanostructure");

        glutDisplayFunc(display_func);
        glutIdleFunc(display_func);

        glutKeyboardFunc(processNormalKeys);
        glutSpecialFunc(processSpecialKeys);

        glutMouseFunc(processMouse);

        glEnable(GL_DEPTH_TEST);
        glutMainLoop();

        return(0);
}

void display_func()
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glScaled(screenSize, screenSize, screenSize);

        glPushMatrix();
        glRotatef(angleFreeX,0.0,1.0,0.0); // (change view angle, x, y, z)
        glRotatef(angleFreeY, 1.0,0.0,0.0);
        glRotatef(angleX, 1.0,0.0,0.0);
        glRotatef(angleY, 0.0,1.0,0.0);
        glTranslatef(posX, posY, 0.0f);

        if (draw_springs(sh, watch)){
                exit(-1);
        }
        if(!readMesh){
                updateFrac(sh);
                updateSpringLength(sh);
                updateAbsorbance(sh);
        }

        if(exportData){
                fprintf(dataFile, "%e\t %e\t %e\n", massArray[pos][0], massArray[pos][5],
massArray[pos][6]);
        }

        glutPostRedisplay();
        glPopMatrix();
        glutSwapBuffers();

        if(!readMesh){
                // Go through each mass node in the list and update coordinates
                for(pos; pos < maxpos; pos++)
                {
                        if(!constTemp){
                                printf("Current Temp: %.2f\r", watch);
                        }
                        else{
                                printf("Current Time: %.2f\r", watch * 1e9);
                        }
```

```
                        assignArrayCoords(pos,massArray[pos][1], massArray[pos][2],
massArray[pos][3]);

                        if(watch < massArray[pos+1][0]){
                                watch = massArray[pos+1][0];
                                if(pos > 0){    // corrects bug in the output of the model
for constant temperature
                                        mh = headpointer;
                                }
                        }
                }
        }
        angleFreeX += angleInc;
        angleFreeY += angleIncY;

        if(playBack == 1)
        {
                time_t start;
                time_t current;

                if(duration){
                        if(massArray[pos][0] > tempStopCom){
                                exit(-1);
                        }
                }

                if(frames < tempPoints-1){
                        maxpos += massCount;
                        pos = maxpos - massCount;
                        frames++;
                        watch = massArray[pos][0];
                }
                else if(frames == tempPoints-1){
                        maxpos = massCount;
                        pos = maxpos - massCount;
                        frames = 0;
                        watch = massArray[pos][0];
                        mh = headpointer;
                        if(exportData){
                                fclose(dataFile);
                                exit(-1);
                        }
                }
                if(!exportData){
                        time(&start);
                        do{
                                time(&current);
                        }
                        while(difftime(current,start) < 1.0);
                }
        }
}

int loadMeshFile()
{
        meshFile = fopen(srcFile, "rt");

        if(meshFile != NULL)
        {
                while(!feof(meshFile))
                {
                        // Node identification variables
                        char type;
```

```
                        int nodeDef = -1;

                        // Temperature variables and concentration values
                        double userTempStart, userTempStop, userConc, userSimTime;
                        int userDataRes, userSnapRes;

                        // Face deflection calculation variables
                        int segRef1, faceRef1, segRef2, faceRef2;

                        // Mass node variables
                        int brownianFlag;
                        double mass, x, y, z;

                        // Spring node variables
                        int nM1, nM2, nS1, nS2, nS3;
                        double restLength, meltTemp, basePairs, extCoeff, extCoeff2,
scaleFactor;

                        // Scan the mesh file and assign values to structs
                        fscanf(meshFile, "%c ", &type);
                        switch(type)
                        {
                                case 'D':
                                case 'd':
                                        nodeDef = 0;
                                        fscanf(meshFile, "%d %d %d %d", &segRef1, &faceRef1,
&segRef2, &faceRef2);
                                        break;

                                case 'M':
                                case 'm':
                                        nodeDef = 1;
                                        fscanf(meshFile, "%lf %lf %lf %lf %d", &mass, &x,
&y, &z, &brownianFlag);
                                        break;

                                case 'S':
                                case 's':
                                        nodeDef = 2;
                                        fscanf(meshFile, "%d %d %d %d %d %lf %lf %lf %lf %lf
%lf", \
                                                &nS1, &nS2, &nS3, &nM1, &nM2, &restLength,
&meltTemp, &basePairs, &extCoeff, &extCoeff2, &scaleFactor);
                                        break;

                                case 'B':
                                case 'b':
                                        nodeDef = 3;
                                        fscanf(meshFile, "%d %d %d %d %d %lf %lf", \
                                                &nS1, &nS2, &nS3, &nM1, &nM2, &restLength,
&scaleFactor);
                                        break;

                                case 'T':
                                case 't':
                                        nodeDef = 4;
                                        fscanf(meshFile, "%d %d %d %d %d %lf %lf", \
                                                &nS1, &nS2, &nS3, &nM1, &nM2, &restLength,
&scaleFactor);
                                        break;

                                case 'C':
                                case 'c':
```

```
                                        nodeDef = 5;
                                        fscanf(meshFile, "%d %d %d %d %d %lf %lf", \
                                                &nS1, &nS2, &nS3, &nM1, &nM2, &restLength,
&scaleFactor);
                                        break;

                                case 'U':
                                case 'u':
                                        nodeDef = 6;
                                        fscanf(meshFile, "%lf %lf %lf %lf %d %d", \
                                                &userTempStart, &userTempStop, &userConc,
&userSimTime, &userDataRes, &userSnapRes);
                                        break;

                                case 'R':
                                case 'r':
                                        nodeDef = 7;
                                        fscanf(meshFile, "%d %d %d %d %d %lf %lf", \
                                                &nS1, &nS2, &nS3, &nM1, &nM2, &restLength,
&scaleFactor);
                                        break;

                                case 'X':
                                case 'x':
                                        nodeDef = 8;
                                        fscanf(meshFile, "%d %d %d %d %d %lf %lf %lf %lf %lf
%lf", \
                                                &nS1, &nS2, &nS3, &nM1, &nM2, &restLength,
&meltTemp, &basePairs, &extCoeff, &extCoeff2, &scaleFactor);
                                        break;

        //                      // Ignore new line
                                case 10:
                                        break;

        //                      // Ignore space at end of line
                                case 32:
                                        break;

                                default:
                                        printf("Unrecognized line starting with '%c'\n",
type);
                                        break;
                        }

                        if(nodeDef == 0){
                                if(meshOutput == 1){
                                        printf("\nDeflection check: (Segment, Face 1 = L, 2
= R) \n\tFace 1: (%d, %d)\n\tFace 2: (%d, %d)\n\n", \
                                                segRef1, faceRef1, segRef2, faceRef2);
                                }
                        }
                        if(nodeDef == 1){
                                // Mass definition...
                                addMass(&mh, mass, x, y, z, brownianFlag);
                                if(meshOutput){
                                        printf("MASS NUM: %d\t MASS: %.2e kg\t MASS COORDS:
(%.1lf nm, %.1lf nm, %.1lf nm)\n", \
                                                 mh->massNum, mh->mass, mh->x * 1e9, mh->y *
1e9, mh->z * 1e9);
                                }
                        }
                        if(nodeDef == 6){
```

```
                                tempStart               = userTempStart;
                                tempStop                = userTempStop;
                                concentration   = userConc;
                                simTime                 = userSimTime;
                                dataRes                 = userDataRes;
                                targetRes               = userSnapRes;
                                if(meshOutput){
                                        printf("\nTEMP RANGE: %.2fK - %.2fK every %es for
%eM sample\n", tempStart, tempStop, simTime, concentration);
                                }
                        }

                        if(nodeDef > 1 && nodeDef < 6 || nodeDef >= 7) {
                                // Spring definition...
                                MASS_node* pM1 = NULL, *pM2 = NULL;

                                if(nM1 < massCount){
                                        pM1 = findmass(&mh, nM1);
                                }
                                else {
                                        printf("Mass %d beyond limit.\n", nM1);
                                }
                                if(nM2 < massCount){
                                        pM2 = findmass(&mh, nM2);
                                }
                                else{
                                        printf("Mass %d beyond limit.\n", nM2);
                                }

                                // Add springs
                                if(nodeDef == 2 || nodeDef == 8){
                                        addSpring(&sh, nS1, nS2, 0, \
                                                restLength, meltTemp, basePairs, extCoeff,
extCoeff2, \
                                                pM1, pM2, \
                                                segRef1, faceRef1, segRef2, faceRef2,
scaleFactor, nodeDef);
                                }
                                if(nodeDef == 3){
                                        addSpring(&sh, nS1, nS2, nS3,
                                                restLength, 0, -1, 0, 0, \
                                                pM1, pM2, \
                                                segRef1, faceRef1, segRef2, faceRef2,
scaleFactor, nodeDef);
                                }
                                if(nodeDef == 4){
                                        addSpring(&sh, nS1, nS2, nS3,
                                                restLength, 0, -2, 0, 0, \
                                                pM1, pM2, \
                                                segRef1, faceRef1, segRef2, faceRef2,
scaleFactor, nodeDef);
                                }
                                if(nodeDef == 5 || nodeDef == 7){
                                        addSpring(&sh, nS1, nS2, nS3,
                                                restLength, 0, -3, 0, 0, \
                                                pM1, pM2, \
                                                segRef1, faceRef1, segRef2, faceRef2,
scaleFactor, nodeDef);
                                }
                                if(meshOutput){
                                        printf("\nSPRING DEF %d: Segment/Spring/Section:
(%d, %d, %d)\n", nodeDef, nS1, nS2, nS3);
                                        printf("\tMass %d -> Mass %d\t", nM1, nM2);
```

```
                                        printf("Rest Length : %.1lf nm\t Tm : %.1lf K\t
\n\tBase Pairs : %.0lf\t Extinction Coefficients : %.2lf \t %.2lf\n", \
                                                sh->restLength * 1e9, sh->meltTemp, sh-
>basePairs, sh->extCoeff, sh->extCoeff2);
                                        printf("\tFace: %d\t Markers: %d %d\n", \
                                                sh->face, sh->marker1, sh->marker2);
                                }
                        }
                }
                fclose(meshFile);
        }
        else {
                perror("Source file not found in the current directory...exiting\n");
                return(-1);
        }

        return(0);
}

int initSettings(void)
{
        // Calc the tempInc, temp range and check if inputs are correct
        if(!constTemp){
                if(!duration){
                        tempRange = tempStop - tempStart;
                        targetTemp1 = tempStart;
                        watch = tempStart;
                }
                else{
                        tempRange = tempStopCom - tempStartCom;
                        targetTemp1 = tempStartCom;
                        watch = tempStartCom;
                }
                tempInc = tempRange / targetRes;
        }

        // Calc the timeInc, time range and check if inputs are correct
        if(constTemp){
                tempRange = simTime;
                targetTemp1 = timeStart;
                watch = timeStart;
                tempInc = tempRange / dataRes;
        }

        while(massArray[pos][0] < targetTemp1){
                pos++;
                watch = massArray[pos][0];
                maxpos = massCount + pos;
        }
        return 0;
}


int loadSnapFile(void)
{
        massFile = fopen(snapFile, "rt");
        headpointer = mh;

        if(massFile != NULL)
        {
                while(!feof(massFile))
                {
                        // Node identification variables
```

```
                        char type;
                        int nodeDef = -1;

                        // Mass node variables
                        int brownianFlag;
                        double mass, x, y, z;

                        // Scan the mesh file and assign values to structs
                        fscanf(meshFile, "%c ", &type);
                        switch(type)
                        {
                                case 'M':
                                case 'm':
                                        nodeDef = 1;

                                        fscanf(massFile, "%lf %lf %lf %lf %d", &mass, &x,
&y, &z, &brownianFlag);
                                        if(!constTemp){
                                                massArray[masses][0] = currentTemp;
                                        }
                                        else{
                                                massArray[masses][0] = currentTime;
                                        }
                                        massArray[masses][1] = x;
                                        massArray[masses][2] = y;
                                        massArray[masses][3] = z;
                                        break;

                                case 'T':
                                case 't':
                                        nodeDef = 2;
                                        if(!constTemp){
                                                fscanf(massFile, "%lf", &currentTemp);
                                                if(temp < currentTemp){
                                                        mh = headpointer;
                                                        temp = currentTemp;
                                                }
                                        }
                                        else{
                                                fscanf(massFile, "%lf", &currentTime);
                                                if(temp < currentTime){
                                                        mh = headpointer;
                                                        temp = currentTime;
                                                }
                                        }
                                        break;

        //                      // Ignore new line
                                case 10:
                                        break;

        //                      // Ignore space at end of line
                                case 32:
                                        break;

                                default:
                                        printf("Unrecognized line starting with '%c'\n",
type);
                                        break;
                        }

                        if(nodeDef == 1){
                                masses++;
```

```
                        }
                        if(nodeDef == 2){
                                tempPoints++;
                        }
                }
                fclose(massFile);
        }
        else {
                perror("Mass file not found in the current directory...exiting\n");
                return(-1);
        }
        watch = massArray[0][0];
        return(0);
}

int assignArrayCoords(int pos, double x, double y, double z)
{
        mh->x = x;
        mh->y = y;
        mh->z = z;

        mh = mh->next;

        return(0);
}

void processNormalKeys(unsigned char key, int x, int y)
{
        switch(key)
        {
                case 27: // ESC
                        exit(-1);
                break;
                // A S D W Rotating camera controls
                case 97:
                        angleY += 1;
                break;
                case 100:
                        angleY -= 1;
                break;
                case 119:
                        angleX += 1;
                break;
                case 115:
                        angleX -= 1;
                break;
                // J K L I Position camera controls
                case 105:
                        posY -= 1;
                break;
                case 107:
                        posY += 1;
                break;
                case 106:
                        posX += 1;
                break;
                case 108:
                        posX -=1;
                break;
        }
}

void processSpecialKeys(int key, int x, int y)
```

```
{
	int mod;
	switch(key)
	{
		// LR for frame controls
		// Add SHIFT for free rotation on the X axis
		// UD for free rotation on the Y axis
		case GLUT_KEY_LEFT :
			mod = glutGetModifiers();
			if (mod == GLUT_ACTIVE_SHIFT){
				angleIncY -= 1e-2;
			}
			else{
				if(frames > 0){
					maxpos -= massCount;
					pos = maxpos - massCount;
					frames--;
					watch = massArray[pos][0];
				}
			}

			break;
		case GLUT_KEY_RIGHT :
			mod = glutGetModifiers();
			if (mod == GLUT_ACTIVE_SHIFT){
				angleIncY += 1e-2;
			}
			else{
				if(frames < tempPoints-1){
					maxpos += massCount;
					pos = maxpos - massCount;
					frames++;
					watch = massArray[pos][0];
				}
				else if(frames == tempPoints-1){
					mh = headpointer;
				}
			}
			break;
		case GLUT_KEY_UP :
			mod = glutGetModifiers();
			if (mod == GLUT_ACTIVE_CTRL){
				screenSize += zoomInc;
			}
			else{
				angleInc += 1e-2;
				if(angleInc > maxAngleInc){
					angleInc = maxAngleInc;
				}
			}
			break;
		case GLUT_KEY_DOWN :
			mod = glutGetModifiers();
			if (mod == GLUT_ACTIVE_CTRL){
				screenSize -= zoomInc;
			}
			else{
				angleInc -= 1e-2;
				if(angleInc < minAngleInc){
					angleInc = minAngleInc;
				}
			}
			break;
```

```
        }
}

int initAbsorbance(SPRING_node *sh)
{
        while(sh != NULL){
                if(sh->nodeDef == 2){
                        // Calculate the absorbance for a fully denatured structure

                        sh->initAbs    = sh->extCoeff2 * concentration; // A_f per molecule
                        sh->meltAbs = sh->extCoeff * concentration;  // A_0
                        sh->absRange = sh->initAbs - sh->meltAbs;
                }

                sh = sh->next;
        }

        return(0);
}
int updateSpringLength(SPRING_node *sh)
{
        while(sh != NULL){
                sh->dx = (sh->m2->x) - (sh->m1->x);
                sh->dy = (sh->m2->y) - (sh->m1->y);
                sh->dz = (sh->m2->z) - (sh->m1->z);

                sh->currentLength = sqrt((sh->dx * sh->dx) + (sh->dy * sh->dy) + (sh->dz *
sh->dz));

                sh->lengthChange = sh->currentLength - sh->restLength;
                /*if(sh->lengthChange < 0){
                        sh->lengthChange = -1 * sh->lengthChange;
                }*/

                sh= sh->next;
        }

        return(0);
}
int updateAbsorbance(SPRING_node *sh)
{
        double totalAbs = 0;
        double lengthChange = 0;
        double alpha1 = 0;

        while(sh != NULL){
                if(sh->nodeDef == 2){
                        // Calculate absorbance based on the derivation
                        // The average difference between dsDNA and ssDNA is based on
analysis from Tataurov et al. '08
                        alpha1 = (sh->absRange * 10) / sh->restLength;
                        sh->absChange = (alpha1 * sh->lengthChange) * (1 - sh->f) * (1 -
sh->f);
                        sh->strandAbs = sh->meltAbs + sh->absChange;
                        lengthChange = sh->lengthChange / springCount;

                        totalAbs += sh->strandAbs / springCount;
                }
                sh = sh->next;
        }
        printf("\t\t\tAbs: %.3lf OD\r", totalAbs * concentration * 6.022e23);
        massArray[pos][5] = totalAbs * concentration * 6.022e23;
        massArray[pos][6] = lengthChange;
```

```
	return(0);
}


int updateFrac(SPRING_node *sh)
{
	double R = 1.98719;
	while(sh != NULL){
		if(sh->basePairs > 1){
			// Avg NN bp enthalpy value is -8.7 kcal/mol (SantaLucia Jr. et
al.)
			sh->H = -8.7e3 * (sh->basePairs - 1);
		}
		else{
			sh->H = -8.7e3;
		}
		// Calculate fraction of ssDNA (D) (John et al.)
		sh->K = exp((-(sh->H) / R) * ((1 / (sh->meltTemp)) - (1 / watch)));
		sh->f = (sh->K) / (sh->K + 1);
		sh = sh->next;
	}
	return(0);
}

void processMouse(int button, int state, int x, int y)
{
	switch(button)
	{
		case GLUT_LEFT_BUTTON:
			screenSize += zoomInc;
		break;
		case GLUT_RIGHT_BUTTON:
			screenSize -= zoomInc;
		break;
	}
}

int refSpringSearch(SPRING_node *sh)
{
	SPRING_node *search = sh;

	int currentNum = 0;
	int currentRef = 0;
	int springType = -1;

	while(sh != NULL)
	{
		currentNum = sh->segNum;
		currentRef = sh->springRef;

		springType = sh->nodeDef;

		switch(springType)
		{
			// Assign 2-4 DNA springs to reference 1 for Brownian force
			case 2:
				if(sh->sh_ref == NULL && sh->springRef != 1){	// DNA
springs 2-4 - reference1 to 1
					sh->sh_ref = DNASpringSearch(search, currentNum);
				}
				break;
			// Assign brace springs to their DNA springs
			case 3:
```

```
                                if(sh->sh_ref == NULL){
                                        sh->sh_ref = braceSpringSearch(search, currentNum,
currentRef);
                                }
                                break;

                        // Assign T and C springs to their DNA springs
                        default:
                                if(sh->sh_ref == NULL){
                                        sh->sh_ref = stiffSpringSearch(search, currentNum);
                                }
                                break;
                }
                sh = sh->next;
        }
        return(0);
}
SPRING_node *DNASpringSearch(SPRING_node *refsh, int currentSeg)
{
        while(refsh != NULL)
        {
                if(refsh->nodeDef == 2 && refsh->segNum == currentSeg && refsh->springRef
== 1){
                        break;
                }
                refsh = refsh->next;
        }
        return(refsh);
}

SPRING_node *braceSpringSearch(SPRING_node *refsh, int currentSeg, int currentRef)
{
        while(refsh != NULL)
        {
                if(refsh->nodeDef == 2 && refsh->segNum == currentSeg && refsh->springRef
== currentRef){
                        break;
                }
                refsh = refsh->next;
        }
        return(refsh);
}
SPRING_node *stiffSpringSearch(SPRING_node *refsh, int currentSeg)
{
        while(refsh != NULL)
        {
                if(refsh->nodeDef == 2 && refsh->segNum == currentSeg){
                        break;
                }
                refsh = refsh->next;
        }
        return(refsh);
}
```

Computation code for the graphical output simulator:

```
//--------------------------//
//      Meshfile Computations   //
//      written: 8.20.08              //
//      last edited: 12.1.09   //
//  author: Vincent Mao               //
```

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

/*  This program takes the values read from the meshfile
        and calculates the forces, absorbance readings, and
        positions of each mass and spring over each iteration
        of temperature.
*/

// Turn off deprecation warnings
#define _CRT_SECURE_NO_DEPRECATE

// PPD's
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "constants.h"
#include "meshcomp.h"
#include "graphics.h"


/* This routine will add a mass to the head of a list of masses.  It gets
   a pointer to the pointer to the head of the list.  If the head of the
   list is null, then it is initialized.  In all cases, the new mass
   becomes the head of the mass list. The initial force and velocity on the
   new mass are set to zero.
   The function returns -1 on failure, 0 on success.
*/

MASS_node* findmass(MASS_node **headptr, int MassIndex)
{
        MASS_node* pNode = *headptr;

        int nMassIndex = (massCount - 1) - MassIndex;
        int i;

        for(i = 0; i < nMassIndex; i++){
                pNode = pNode->next;
        }

        return pNode;
}
/* end of findmass() */

int addMass(MASS_node **headptr, double mass, double x, double y, double z, int b)
{
        MASS_node *newm;       /* Pointer to the new mass */

        if ( (newm = (MASS_node *)malloc(sizeof(MASS_node))) == NULL )
        {
                perror("add_mass (cannot allocate more memory)");
                return(-1);    /* FAIL */
        }

        newm->next = *headptr; /* Insert before head of the list */
        *headptr   = newm;         /* Point the head to this entry */

        newm->mass                    = mass;
        newm->x                       = x;         // Units are in METERS
        newm->y                       = y;
        newm->z                       = z;
        newm->initx                   = x;
        newm->inity                   = y;
        newm->initz                   = z;
```

```
        newm->brownianFlag     = b;

        // Initialize all values not provided by the meshfile
        newm->damping          = 0;
        newm->vx = newm->vy = newm->vz                      = 0.0;
        newm->fx = newm->fy = newm->fz                      = 0.0;

        /* Assign the mass to a mass node number that
           matches the one in the meshfile
        */
        newm->massNum  = massCount;
        massCount++;

    return(0);         /* SUCCESS */
}
/* end of add_mass */
/* end of overall mass routine */

/* This routine will add a spring to the head of a list of springs.
   It gets a pointer to the pointer to the head of the list.  It will put
   the new spring at the head of the list.  It works properly if the head
   of the list is NULL to begin with.  m1 and m2 should point to the masses
   at either end of the spring.
   The function returns -1 on failure and 0 on success.
*/
int     addSpring(SPRING_node **headptr,
                  int segNum, int springRef, int springSec,
                  double restLength, double meltTemp, double basePairs, double extCoeff,
double extCoeff2,
                      MASS_node *m1, MASS_node *m2,
                      int segRef1, int faceRef1, int segRef2, int faceRef2,
                      double scaleFactor, int nodeDef)
{
        SPRING_node    *news;  /* The new spring node location pointer */

        if ( (m1 == NULL) || (m2 == NULL) )
        {
               perror("addSpring (bad mass)");
               return(-1);    /* BAD MASS */
        }

        if ( (news = (SPRING_node *)malloc(sizeof(SPRING_node))) == NULL)
        {
               perror("addSpring (cannot allocate more memory)");
               return(-1);    /* FAIL */
        }

        news->next                    = *headptr; /* Place new entry at head of list */
        *headptr                      = news;    /* Make the head point to the new one */

    // Assign parameter values provided by the meshfile */
        news->segNum          = segNum;                     // Segment number
        news->springRef               = springRef;          // Spring number
        news->springSec               = springSec;          // Section number
        news->m1                      = m1;               // First mass node identifier
        news->m2                      = m2;                         // Second mass node
identifier
        news->restLength    = restLength;            // Rest length of the spring
        news->meltTemp      = meltTemp + 273;   // Melting temperature in K (C in
meshfile)
        news->basePairs               = basePairs;          // Number of bases in a
segment
        news->extCoeff      = extCoeff;         // Extinction coefficient for ssDNA
```

```
        news->extCoeff2              = extCoeff2;          // Extinction coefficient for
dsDNA
        news->scaleFactor     = scaleFactor;        // Spring constant scaling factor
        news->nodeDef         = nodeDef;                  // Type of spring

        // Initialize values
        news->marker1   = 0;
        news->marker2   = 0;
        news->face            = 0;
        news->k         = 0.0;
        news->sh_ref    = NULL;
        news->ref_k           = 0.0;
        news->K               = 0.0;
        news->f               = 0.0;
        news->H               = 0.0;
        news->angle           = 0.0;
        news->harmFreq  = 0.0;
        news->dx              = news->dy     = news->dz     = 0.0;
        news->dfx             = news->dfy = news->dfz = 0.0;
        news->crossx   = news->crossy = news->crossz = 0.0;
        news->dfx1            = news->dfy1 = news->dfz1 = 0.0;
        news->dfx2            = news->dfy2 = news->dfz2 = 0.0;
        news->currentLength          = restLength;
        news->lengthChange           = 0.0;
        news->energy   = 0.0;
        news->initAbs  = 0.0;
        news->meltAbs  = 0.0;
        news->absRange = 0.0;
        news->strandAbs = 0.0;

        if(news->nodeDef == 2){
                springCount++;
        }

        // Identify the mass nodes as being on one face or the other (0-3, 4-7)
        // This is done for every face in the model
        if(news->nodeDef == 4){
                if(news->m1->massNum % 8 < 4 && news->m2->massNum % 8 < 4){
                        news->face = 1;
                }
                if(news->m1->massNum % 8 > 3 && news->m2->massNum % 8 > 3){
                        news->face = 2;
                }

                // Identify which face is being monitored for deflection test
                // When complete, two faces should be marked
                if(news->segNum == segRef1 && news->face == faceRef1){
                        news->marker1  = 1;
                }
                if(news->segNum == segRef2 && news->face == faceRef2){
                        news->marker2  = 1;
                }
        }

        totalSpringCount++;

        return(0);      /* SUCCESS */
}
/* end of addSpring */
```

OpenGL source code for the graphical output simulator:

```
//--------------------------//
//      Graphics Code                //
//      last edited: 12.1.09   //
//  editor: Vincent Mao              //
//--------------------------//

#ifdef _WIN32
#include <windows.h>
#include <float.h>
#define finite(x) _finite(x)
#endif

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <GL/gl.h>
#include <GL/glu.h>

#ifdef _WIN32
#include "glut.h"
#endif

#ifndef _WIN32
#include <GL/glut.h>
#endif

#include "meshcomp.h"
#include "graphics.h"

const int SCREEN_X=1024;
const int SCREEN_Y=1024;
double width, height;

int getmaxx(void) { return SCREEN_X; }
int getmaxy(void) { return SCREEN_Y; }

double  SCREEN_SCALE = 0.025;



static int vertex(MASS_node *mn)
{
        // Returns error if the points are not finite.
        if ( !finite(mn->x * 1e9)  || !finite(mn->y * 1e9) || !finite(mn->z * 1e9) ) {
                fprintf(stderr,"Infinite points can't be drawn (%lf, %lf, %lf)...\n", mn-
>x, mn->y, mn->z);
                return -1;
        }
        glVertex3f(mn->x * 1e9, mn->y * 1e9, mn->z * 1e9);

        return 0;
}

int     draw_masses(MASS_node *mn)
{
    if (mn != NULL) {
                glPushMatrix();
                glBegin(GL_LINES);
                while (mn != NULL) {
                        glColor3f(1,0.75,0);
                        vertex(mn);
                        mn = mn->next;
                }
```

```
                glEnd();
                glPopMatrix();
        }
        return 0;
}

int     draw_springs(SPRING_node *sn, double temp)
{
        if (sn != NULL)
                {
                glBegin(GL_LINES);
                // Select type of lines to draw based on node definition:(2 = DNA 'S', 3 =
Stiff 'T', 4 = Brace 'B', 5 = Cross 'C', 7 = Restore, 8 = Cross-link)
                while (sn != NULL)
                {
                        if(sn->nodeDef == 2 || sn->nodeDef == 7){
                                glColor3f(0,1,0);
                                vertex(sn->m1);
                                vertex(sn->m2);
                        }

                        if(sn->nodeDef == 3){
                                if(temp < sn->sh_ref->meltTemp){
                                        glColor3f(0,0,1);
                                        vertex(sn->m1);
                                        vertex(sn->m2);
                                }
                        }
                        if(sn->nodeDef == 4 || sn->nodeDef == 8){
                                glColor3f(1,1,0);
                                vertex(sn->m1);
                                vertex(sn->m2);
                        }
                        if(sn->nodeDef == 5){
                                if(temp < sn->sh_ref->meltTemp){
                                        glColor3f(1,0,0);
                                        vertex(sn->m1);
                                        vertex(sn->m2);
                                }
                        }
                        sn = sn->next;
                }
                glEnd();
        }
        return 0;
}

int     setup_graphics_frame(void)
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glScaled(SCREEN_SCALE, SCREEN_SCALE, SCREEN_SCALE);

        glRotatef(45,1.0,1.0,0.0); // (change view angle, x, y, z)

        return 0;
}

int     init_graphics(char *name, Display_Func df)
{
        static  int     no_args = 1;
        static  char    *fargv[1] = {"Glut_program"};
```

```
        int     glut_window;

        static int glutAttributeList = GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH;
        glutInit(&no_args, fargv);
        glutInitDisplayMode(glutAttributeList);
        glutInitWindowPosition(0,0);
        glutInitWindowSize(SCREEN_X, SCREEN_Y);
        glut_window = glutCreateWindow(name);
        glutSetWindow(glut_window);
        glutDisplayFunc(df);
        glutIdleFunc(df);
        //glutReshapeFunc(changeSize);
        glEnable(GL_DEPTH_TEST);
        glClearColor(0.0,0.0,0.0,0.0);
        glClearDepth(1.0f);

        return 0;
}
```

Header file source code for the graphical output program:

```
//--------------------------//
//      Meshfile Header                     //
//      written: 8.20.08              //
//      last edited: 11.9.09   //
//  author: Vincent Mao               //
//--------------------------//

#ifndef MESHCOMP_H
#define MESHCOMP_H

#ifndef NULL
#define NULL    0
#endif

typedef struct MASS /* Used to allow a pointer to this type in struct */
{
        // Defined by the mesh file
        double  mass;                   /* The mass of this point */
        double  x, y, z;                /* Mass coordinates */
        int             massNum;                /* Mass number assigned as identifier */
        int             brownianFlag;   /* Flag for brownian force */

        double  initx, inity, initz;    /* Initial positions of each mass */
        double  damping;                                /* Mass damping coefficient */
        double   vx,vy,vz;                              /* The velocity of this point */
        double  fx,fy,fz;                               /* The forces acting on this point */
        struct  MASS *next;                             /* The next mass in the list */
}
MASS_node;

typedef struct SPRING /* Used to allow a pointer to this type in struct */
{
        // Defined by the mesh file
        int             segNum;                 // Segment number identifier
        int             springRef;              // Spring reference number for the B spring
to find its S spring
        int             springSec;              // Assign the T spring the B spring is
associated with
        double  restLength;             // The rest length of the spring
```

```
        double  meltTemp;       // Melting temperature of the spring at which k = 0.5*k
        double  basePairs;            // The number of base pairs in the segment
        double  extCoeff;             // ssDNA extinction coefficient
        double  extCoeff2;            // dsDNA extinction coefficient
        double  scaleFactor;   // Scaling factor for the spring
        int             marker1, marker2;
        int             face;

        //Calculated after reading values from the mesh file
        int             nodeDef;              // Spring type identifier
        double  H;                            // Enthalpy of dsDNA
        double  K;                            // Dissociation rate constant
        double  f;                            // Fraction of ssDNA
        double  k;                            // Spring constant
        double  ref_k;                // Reference spring constant from the DNA spring for
structural springs
        double  angle;                // Angle formed between DNA and brace springs
        double  harmFreq;       // Harmonic frequency of oscillation
        double  lengthChange;  // The difference in length from rest
        double  currentLength; // The current length of the spring
        double  dx, dy, dz;           // Change in coords for spring length
        double  energy;               // Kinetic energy of the spring
        double  dfx, dfy, dfz; // Change in force in springs
        double  initAbs;              // Initialized absorbance for strands
        double  meltAbs;              // Absorbance for fuller denatured strands
        double  absRange;             // Absorbance range from native to denatured strands
        double  strandAbs;      // Absorbance the strand contributes
        double  absChange;            // Change in absorbance based on change in strand
length
        double  crossx, crossy, crossz;       // Cross product coords
        double  dfx1, dfy1, dfz1;             // Change in Brownian force for masses
        double  dfx2, dfy2, dfz2;

        MASS_node *m1;                // Pointer to first mass attached to the spring
        MASS_node *m2;                // Pointer to second mass attatched to the spring

        struct SPRING *sh_ref; // The reference spring for structural springs
        struct SPRING  *next;  // Points to the next spring in the list
}
SPRING_node;

extern MASS_node* findmass(MASS_node **headptr, int a_nMassIndex);

extern  int     addMass(MASS_node **headptr,
                        double mass, double x, double y, double z, int b);

extern  int     addSpring(SPRING_node **headptr,
                    int segNum, int springRef, int springSec,
                    double restLength, double meltTemp, double bases, double extCoeff,
double extCoeff2,
                        MASS_node *m1, MASS_node *m2,
                        int segRef1, int faceRef1, int segRef2, int faceRef2,
                        double scaleFactor, int nodeDef);

#endif
```

Source code for the graphics header for OpenGL in the graphical output program:

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

```
//      Graphics Header                        //
//      last edited: 11.2.09   //
//  editor: Vincent Mao               //
//---------------------------//

#ifndef GRAPHICS_H
#define GRAPHICS_H

#ifdef _WIN32
#include <windows.h>
#endif

#include "meshcomp.h"

extern int getmaxx(void);
extern int getmaxy(void);

extern int draw_masses(MASS_node *mn);
extern int draw_springs(SPRING_node *sn, double temp);

typedef void (*Display_Func)(void);

extern int init_graphics(char *name, Display_Func df);
extern int setup_graphics_frame(void);

#endif
```

Source code for variables in the graphical output program:

```
//---------------------------//
//      Constants and Variables //
//      written: 10.16.08             //
//      last edited: 11.2.09   //
//  author: Vincent Mao               //
//---------------------------//

#ifndef _CONSTANTS_H_
#define _CONSTANTS_H_

// Turn off deprecation warnings
#define _CRT_SECURE_NO_WARNINGS

#ifndef NULL
#define NULL    0
#endif

// Globals
#ifndef M_PI
#define M_PI 3.1415926                // PI
#endif

#define BMASS          0.660          // Mass of a base pair in Daltons (g/L)-->kg/mol
#define AVOGADRO       6.022E23       // Avogadro's number
#define BLENGTH                3.4E-10     // Contour length of a single dsDNA base pair
is 0.34 angstroms
#define PLENGTH                53E-9       // Persistence length for dsDNA is 53 nm
#define BOLTZMANN      1.38E-23    // Boltzmann constant in Nm/K
#define DNA_DIAM       2e-9           // Diameter of DNA is 2 nm
#define R                      1.98719        // Gas constant in cal/mol
```

```
int massCount          = 0;           // Total mass nodes in the structure
int springCount                = 0;           // Total DNA springs in the structure
int segCount           = 0;           // Total number of segments in the structure for WLC
model
int totalSpringCount= 0;              // Total springs in the structure

double totalBasePairs          = 0.0;
double totalAbsorbance         = 0.0;
double alpha0                          = 0.0;
double alpha1                          = 0.0;
double dotProdRatio                    = 0.0;

#endif
```

11. The Crossover and Core Region Models

The crossover region model is composed of four 3D rectangular models connected together using four "R" springs and twenty four "C" springs. These springs follow the melting behavior of their neighboring DNA springs. Two "R" springs connect the DNA springs together to represent the single shell strand in the DNA tile motif. The ten "C" springs that connect the two structures are used as supports to prevent them from buckling into each other. The lower part of the crossover region is connected in the same way, except that the "R" springs are used to connect the bottom two DNA springs to represent the single shell strand along the bottom. Figure 71 shows a screen capture of the crossover region of the model.

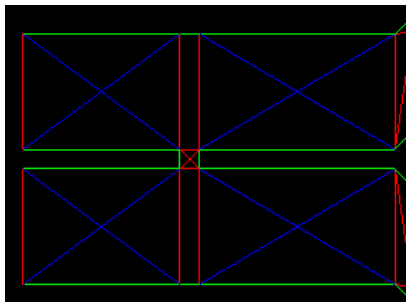


Figure 71: The Crossover spring model.

The two halves of the crossover region are connected and supported by four "R" springs that represent the single core strands and the single arms strands on the inner and outer part of the tile model, respectively. The eight "C" springs are used to prevent the two halves of the crossover region from collapsing together. Each of the crossover regions melts with their respective neighboring segments.

For the core region, sixteen "R" springs are used to represent poly-T segments in the core region model that allow bending of the core strand into the tile motif shape. These springs are supported by thirty two "C" springs, eight for each turn. Figure 72 illustrates the core model of the tile motif:

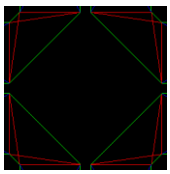


Figure 72: The tile motif core spring model.

The melting temperature of each of these springs is determined by their neighboring DNA springs. These regions represent ssDNA poly-T segments and do not have any melting temperature of their own as they are part of the core strand.

12. The Cross-linking Model

The "X" cross-linking springs have no melting temperature but do contribute to the absorbance because the model includes the base pairs they cross-link. The spring constant of these springs is determined using Equation 7, making these springs much

stiffer than the rest of the 3D spring model. The rest length of the spring is equivalent of two bases (i.e., 6.8 Å) because intercalation of the TMP molecule is the equivalent of adding an extra base pair to the structure. In addition to changing the rest length and the type of spring at these locations in the tile motif model, the supporting springs are also changed to "X" springs. The following diagram illustrates the difference between a normal 5 bp spring model and one that has a cross-link between the second and third bases.

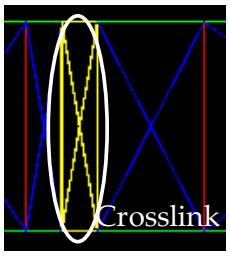


Figure 73: The cross-link model in a sticky end.

The cross-link spring has the same input format in the input file as the DNA spring as described in Section 4 of this appendix, except that the identifier is "X" instead of "S".

C. The Calculation of ssDNA and dsDNA Extinction Coefficients

The calculation of the ssDNA, complementary ssDNA, and dsDNA extinction coefficients were determined using the nearest-neighbor values and the expression for determining extinction coefficients for specific sequences in [23]. The ssDNA sequences for which the extinction coefficients were determined were the core, shell, and arm

strands for the 4 x 4 DNA grid nanostructure, for a total fifty-eight strands ranging from 26 to 100 bases in length. The extinction coefficients for each strand were then determined per base pair and averaged to determine the final value. These coefficients were also determined per kg to calculate the extinction coefficients in terms of their mass. Table 18, reproduced from [82], presents the specific nearest neighbor sequences with their extinction coefficient values. Table 19 provides the resulting calculations for each strand based on these values.

Table 18: Extinction coefficient values for nearest neighbor pairs.

ij (5′-3′)	ε_{ij}^{nn} *(260 nm) (10^3 M^{-1} cm^{-1})*
EA, AE	7.70
EC, CE	3.70
EG, GE	5.75
ET, TE	4.35
AA	12.00
AC	9.80
AG	11.55
AT	10.75
CA	9.80
CC	7.20
CG	8.55
CT	7.15
GA	11.75
GC	8.15
GG	10.10
GT	9.90
TA	11.35
TC	8.15
TG	8.90
TT	8.10

Table 19: Data Table for Extinction Coefficient Calculation

Extinction Coefficient Analysis															
Strand	Bases	Conc (uM)	Coeff (ssDNA)	Comp Coeff (ssDNA)	Coeff (dsDNA)	Coeff/base			Ratio	dsDNA Mass (kg)	ssDNA Mass (kg)	Coeff/kg (dsDNA)	Coeff/kg (ssDNA)	Coeff/kg (CssDNA)	Tm (°C)
Core A	100	0.3700	922000	991800	1573986	9220	9918	15740	1.707143	66	33	23848	27939	30055	72.3
Core B	100	0.7650	910600	1007600	1581978	9106	10076	15820	1.737292	66	33	23969	27594	30533	72
S-A1	42	0.5103	396000	402200	668778	9429	9576	15923	1.688833	27.72	13.86	24126	28571	29019	70.3
S-A2	47	0.5598	432900	462000	755277	9211	9830	16070	1.744692	31.02	15.51	24348	27911	29787	71.4
S-A3	42	0.4902	403100	387900	671333	9598	9236	15984	1.665425	27.72	13.86	24218	29084	27987	71.4
S-A4	37	0.4196	353900	342400	755277	9565	9254	20413	2.134154	24.42	12.21	30929	28984	28043	71.3
S-B1	42	0.5314	403100	398100	671291	9598	9479	15983	1.665321	27.72	13.86	24217	29084	28723	68.5
S-B2	47	0.6769	464600	443300	753035	9885	9432	16022	1.620824	31.02	15.51	24276	29955	28582	67.9
S-B3	42	0.5018	394100	408900	677158	9383	9736	16123	1.718239	27.72	13.86	24428	28434	29502	69.5
S-B4	37	0.4700	347800	353100	594761	9400	9543	16075	1.710066	24.42	12.21	24355	28485	28919	68.6
Arm 1.1	26	0.1437	251600	254800	414352	9677	9800	15937	1.646868	17.16	8.58	24146	29324	29697	58.2
Arm 1.2	36	0.1700	352700	338100	575667	9797	9392	15991	1.632172	23.76	11.88	24228	29689	28460	65.7
Arm 1.3	36	0.1140	334400	336400	580252	9289	9344	16118	1.735203	23.76	11.88	24421	28148	28316	71.7
Arm 1.4	26	0.1371	239300	268500	411045	9204	10327	15809	1.717697	17.16	8.58	23954	27890	31294	58.8
Arm 2.1	26	0.1455	246000	240000	414708	9462	9231	15950	1.685805	17.16	8.58	24167	28671	27972	65.9
Arm 2.2	36	0.1500	363700	329300	577500	10103	9147	16042	1.587847	23.76	11.88	24306	30614	27719	66.1
Arm 2.3	36	0.1620	349200	344000	573276	9700	9556	15924	1.641684	23.76	11.88	24128	29394	28956	66.6
Arm 2.4	26	0.1454	245200	251600	410854	9431	9677	15802	1.675587	17.16	8.58	23943	28578	29324	60.4
Arm 3.1	26	0.1600	262500	251500	416063	10096	9673	16002	1.585002	17.16	8.58	24246	30594	29312	57.3
Arm 3.2	36	0.1994	337800	346200	567322	9383	9617	15759	1.679461	23.76	11.88	23877	28434	29141	67.1
Arm 3.3	36	0.2433	346100	344900	580210	9614	9581	16117	1.676423	23.76	11.88	24420	29133	29032	66.8
Arm 3.4	26	0.1615	262300	251500	415901	10088	9673	15996	1.585593	17.16	8.58	24237	30571	29312	55.5
Arm 4.1	26	0.1669	253800	239000	416189	9762	9192	16007	1.639831	17.16	8.58	24253	29580	27855	64.4
Arm 4.2	36	0.2152	349400	348400	568242	9706	9678	15785	1.626337	23.76	11.88	23916	29411	29327	62.7
Arm 4.3	36	0.1999	338000	366200	573454	9389	10172	15929	1.696609	23.76	11.88	24135	28451	30825	64
Arm 4.4	26	0.1727	262400	247400	412663	10092	9515	15872	1.572649	17.16	8.58	24048	30583	28834	57.2
Arm 5.1	26	0.1266	243400	232200	414174	9362	8931	15930	1.701619	17.16	8.58	24136	28368	27063	69.8
Arm 5.2	36	0.1500	364000	337400	575616	10111	9372	15989	1.581363	23.76	11.88	24226	30640	28401	64.5
Arm 5.3	36	0.1872	339000	345000	574332	9417	9583	15954	1.694195	23.76	11.88	24172	28535	29040	69.2
Arm 5.4	26	0.1298	240000	262000	415154	9231	10077	15967	1.729808	17.16	8.58	24193	27972	30536	61.9
Arm 6.1	26	0.1470	240300	253500	412703	9242	9750	15873	1.717449	17.16	8.58	24050	28007	29545	63.5
Arm 6.2	36	0.2145	361300	332500	578167	10036	9236	16060	1.600241	23.76	11.88	24334	30412	27988	66
Arm 6.3	36	0.2201	340100	337500	577541	9447	9375	16043	1.698151	23.76	11.88	24307	28628	28409	70.5
Arm 6.4	26	0.1476	239400	259400	412508	9208	9977	15866	1.723091	17.16	8.58	24039	27902	30233	61.3
Arm 7.1	26	0.1450	253600	244400	420580	9754	9400	16176	1.658438	17.16	8.58	24509	29557	28485	63.6
Arm 7.2	36	0.2298	341900	347100	569803	9497	9642	15828	1.666578	23.76	11.88	23982	28779	29217	65.4
Arm 7.3	36	0.2010	351000	349200	574631	9750	9700	15962	1.637125	23.76	11.88	24185	29545	29394	64.5
Arm 7.4	26	0.1528	266000	243400	412340	10231	9362	15859	1.55015	17.16	8.58	24029	31002	28368	55.9
Arm 8.1	26	0.1485	260700	249500	412987	10027	9596	15884	1.584147	17.16	8.58	24067	30385	29079	58.6
Arm 8.2	36	0.1575	337900	348900	567984	9386	9692	15777	1.680923	23.76	11.88	23905	28443	29369	67.3
Arm 8.3	36	0.2441	340900	343700	574836	9469	9547	15968	1.686231	23.76	11.88	24193	28695	28931	68.6
Arm 8.4	26	0.1611	259600	261600	412750	9985	10062	15875	1.589946	17.16	8.58	24053	30256	30490	53.5
Arm 9.1	26	0.1445	253900	239500	412369	9765	9212	15860	1.624139	17.16	8.58	24031	29592	27914	63.6
Arm 9.2	36	0.1940	361300	333500	579000	10036	9264	16083	1.602546	23.76	11.88	24369	30412	28072	66
Arm 9.3	36	0.2224	342000	331600	574132	9500	9211	15948	1.678749	23.76	11.88	24164	28788	27912	70.8
Arm 9.4	26	0.1512	236200	270800	410397	9085	10415	15785	1.737498	17.16	8.58	23916	27529	31562	58.9
Arm 10.1	26	0.1646	233800	247600	410782	8992	9523	15799	1.75698	17.16	8.58	23938	27249	28858	66.8
Arm 10.2	36	0.2247	359100	333300	577000	9975	9258	16028	1.606795	23.76	11.88	24285	30227	28056	66.4
Arm 10.3	36	0.2145	354000	333400	577187	9833	9261	16033	1.630472	23.76	11.88	24292	29798	28064	68.7
Arm 10.4	26	0.1400	241500	258900	413831	9288	9958	15917	1.713586	17.16	8.58	24116	28147	30175	60.8
Arm 11.1	26	0.1529	260800	247800	416152	10031	9531	16006	1.595675	17.16	8.58	24251	30396	28881	58.2
Arm 11.2	36	0.2156	333700	357300	571457	9269	9925	15874	1.712487	23.76	11.88	24051	28089	30076	64.9
Arm 11.3	36	0.2118	348800	338600	577187	9689	9406	16033	1.654779	23.76	11.88	24292	29360	28502	67.7
Arm 11.4	26	0.1541	265700	248500	416225	10219	9558	16009	1.566522	17.16	8.58	24256	30967	28963	57.4
Arm 12.1	26	0.1383	251300	241900	416526	9665	9304	16020	1.657485	17.16	8.58	24273	29289	28193	62.9
Arm 12.2	36	0.2050	342400	359800	571825	9511	9994	15884	1.67005	23.76	11.88	24067	28822	30286	64.2
Arm 12.3	36	0.2034	352600	351800	573616	9794	9772	15934	1.626818	23.76	11.88	24142	29680	29613	64.5
Arm 12.4	26	0.1512	274700	241500	417844	10565	9288	16071	1.521092	17.16	8.58	24350	32016	28147	55.3
Arm 13.1	26	0.1434	241300	241500	411977	9281	9288	15845	1.707323	17.16	8.58	24008	28124	28147	67.5
Arm 13.2	36	0.2171	363300	338100	575616	10092	9392	15989	1.58441	23.76	11.88	24226	30581	28460	64.4
Arm 13.3	36	0.1360	363500	333700	576584	10097	9269	16016	1.586201	23.76	11.88	24267	30598	28089	65.9
Arm 13.4	26	0.1474	244600	257200	414989	9408	9892	15961	1.696603	17.16	8.58	24184	28508	29977	61.8
Arm 14.1	26	0.1452	244800	260000	413043	9415	10000	15886	1.687267	17.16	8.58	24070	28531	30303	60.7
Arm 14.2	36	0.2273	361400	335200	580500	10039	9311	16125	1.606253	23.76	11.88	24432	30421	28215	65.7
Arm 14.3	36	0.2295	350300	326300	576689	9731	9064	16019	1.646272	23.76	11.88	24271	29487	27466	70.6
Arm 14.4	26	0.1603	235100	264300	413004	9042	10165	15885	1.756716	17.16	8.58	24068	27401	30804	61.5
Arm 15.1	26	0.1460	247500	245700	416526	9519	9450	16020	1.682933	17.16	8.58	24273	28846	28636	63.7
Arm 15.2	36	0.1964	347100	343700	571292	9642	9547	15869	1.6459	23.76	11.88	24044	29217	28931	65.4
Arm 15.3	36	0.1949	342200	360000	571825	9506	10000	15884	1.671026	23.76	11.88	24067	28805	30303	64.5
Arm 15.4	26	0.1629	253500	257900	413959	9750	9919	15922	1.632974	17.16	8.58	24123	29545	30058	57.3
Arm 16.1	26	0.1590	259600	250000	412502	9985	9615	15865	1.588991	17.16	8.58	24039	30256	29138	57.1
Arm 16.2	36	0.2123	351000	336600	568645	9750	9350	15796	1.620071	23.76	11.88	23933	29545	28333	67.5
Arm 16.3	36	0.2081	344400	341600	576011	9567	9489	16000	1.672506	23.76	11.88	24243	28990	28754	67.4
Arm 16.4	26	0.1840	276200	247200	414493	10623	9508	15942	1.500699	17.16	8.58	24155	32191	28811	52.2
Average	34	0.2230	327557	326719	545133	9681	9568	15943	1.65	20	10	24156	29338	29025	
											Per Mass EC	24156		29181	

D. Construction and Alignment of the DBS

The DBS system was designed from first principles based in optical physics as well as fluorescence spectroscopy [91] to determine the appropriate light source, lenses, and data acquisition instruments. All materials were purchased from Thorlabs, Edmund Optics, Vincent Associates, Hamamatsu, and Ocean Optics. Alignment of the instrument was done using a pair of adjustable irises, adjustable beam slits, and an Ocean Optics S2000 spectrometer with data acquisition using SpectraSuite software from Ocean Optics.

The dual beam monochromator design uses a 500W Hg/Xe L8288 Hamamatsu light source with a spectral range from 240 nm to 850 nm. Only the initial collimation stage at the light source uses a 75 mm focal length lens. The remaining lenses have a 100 mm focal length. All lenses were made with UV-fused silica plano-convex lenses from Thorlabs allowing 170-2000 nm wavelengths. Collimation lens pairs and manually adjustable entrance iris and exit slits are used to control the power throughput to the sample and reference chambers of the instrument. Filter wheels and/or by 1200 groove/mm diffraction gratings on motorized rotation mounts controlled by the appropriate software select the specific wavelength. Ocean Optics beam couplers, sample mounts, optical fibers, and software for the spectrometer collect the data from

the sample. Filter wheels, irises, and electronically-controlled shutters control sample exposure.

The DBS was concurrently aligned and constructed by placing the proper lenses on a Thorlabs optical table that had a 25 mm pitch between holes. The alignment of each stage of the DBS was verified before proceeding. The primary collimation leg was constructed and aligned using a 150 μm pinhole and an adjustable iris to optimize beam centering. Power was maximized at each stage using a series 818 low power silicon photodetector from Newport to measure power throughput. The initial collimation leg only required a 75 mm focal length lens and an Hg/Xe light source. Thorlabs travel translational stages were used for manual positioning of the beam until it was properly distanced for approximate collimation on all degrees of freedom. The 100 mm focal length lenses were then adjusted for proper collimation. The holographic grating was aligned to reflect the beam back through the system to verify the alignment. Figure 74 illustrates the initial alignment schematic of the primary collimation leg.

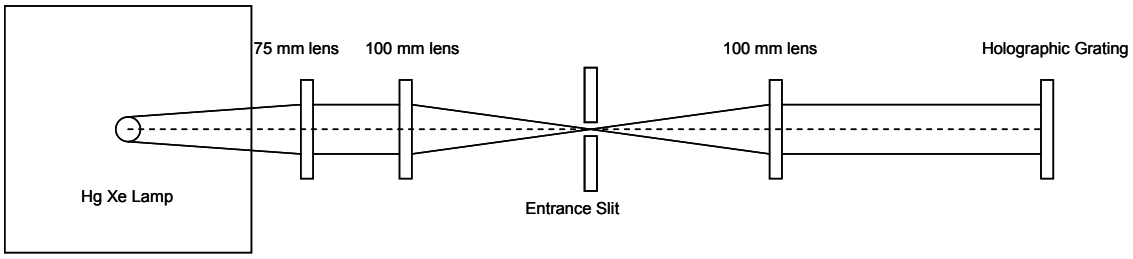


Figure 74: Primary collimation leg alignment of the DBS.

A 50/50 beam splitter was placed 50 mm after the pinhole for the next stage of construction. A 100 mm collimation lens was placed 50 mm after the beam splitter, setting the path length to be equivalent for both collimation lenses from the beam splitter. The second holographic grating was then positioned for maximum beam coverage. The beam splitter was then adjusted to a 90° reflectance angle using the lens and the grating as reference. Verification of this section of the collimation was done in the same fashion as described for the primary collimation leg. Figure 75 presents the resulting alignment.

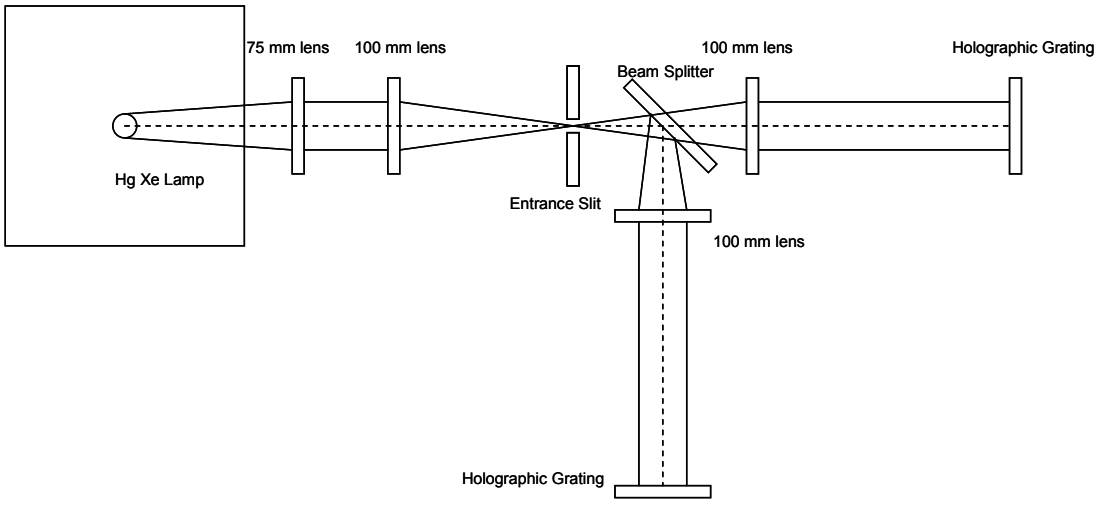


Figure 75: Alignment of the second leg.

The same alignment procedures were repeated for the third stage of construction, treating each grating as a reference light source. The exit slits, and subsequently the beam couplers, were used in place of the grating as the alignment target. The beam combiner was then positioned for maximum overlap of the two beams and minute

adjustments were made. With the exception of the beam combiner and the exit slits due to ghosting effects from the beam splitter, all lens post holders were mounted directly on the optical table.

The DBS monochromator holographic gratings, filter wheels, and the spectrometer are controlled by the computer using Thorlabs and Ocean Optics software. The monochromators are capable of sweeping over the full range the light source provides from ~240 nm to ~840 nm with a 10 nm FWHM. Maximum power of approximately 90 μW can be obtained at the 0^{th} order (full spectrum) position.

Temperature control to both cuvette sample chambers is facilitated using two temperature-controlled water baths from PolyScience. The design allows for control over the water bath source to feature rapid temperature swings. The plumbing system consists of a series of plastic tubes connected to brass valves, all purchased from McMaster Carr, in the appropriate configuration for switching between water baths. The baths are capable of temperatures ranging from 4°C to 100°C as well as adjustable pump rates up to 12L/min. The schematic for the plumbing system including the temperature baths is illustrated in Figure 76.

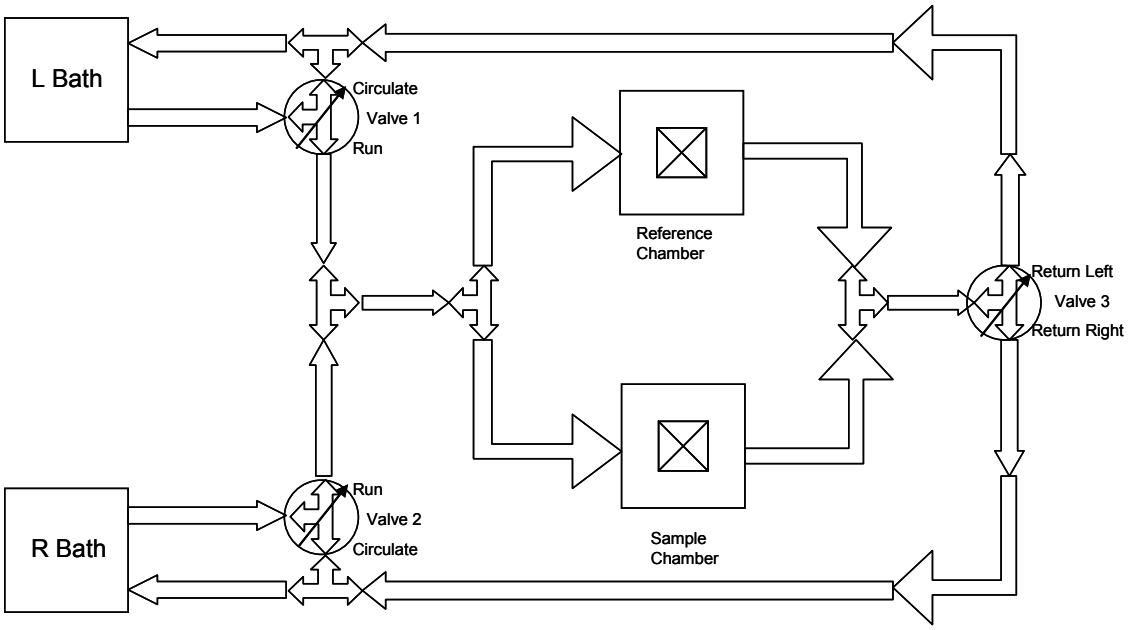


Figure 76: Plumbing system for temperature control to the sample chambers.

There are three valves which control the water flow from baths. Valves 1 and 2 on the far left of the figure control whether the water returns to the bath or flows through the system. Valve 3 on the far right selects which bath the water goes to after going through the system.

E. The TMP Cross-linking Procedure into 8-Tile DNA Nanostructures

For self-assembly of the tile motifs, oligonucleotide sequences from IDTDNA were annealed in Eppendorf tubes over a 90-minute period of cooling from 95°C to 4°C in a PolyScience water bath. Equal volumes of the individual tiles were then combined in an Eppendorf tube and incubated for 4 hours in a 23°C temperature water bath. The resulting 125 nM 8-tile grid was diluted to 40 nM by adding 1x TAE Mg^{2+} buffer to the

solution. The sample was prepared by adding 15 μL of 100 μM TMP dissolved in pure ethanol to 135 μL of 8-tile DNA grid. The monoadduct formed over a 60-minute incubation period at 10°C in complete darkness in the DBS to prevent heating of the DNA. The fluorescence of the sample was monitored using a 330 nm excitation wavelength before and after the incubation period using the SpectraSuite spectral data acquisition software from Ocean Optics. At the end of the incubation period, another 330 nm wavelength spectra was taken before the 365 nm excitation wavelength exposure of the sample to complete the cross-linking process. The sample was illuminated at this wavelength for 45 minutes and 0.66 $\mu W/cm^2$ at 10°C. Upon completion of the cross-linking step, fluorescence spectra were taken at both 365 nm and 330 nm wavelengths.

F. Procedures for the Denaturation and Reannealing of Cross-linked DNA Nanostructures

A 5 nM sample is prepared by adding 630 μL of 1X TAE Mg^{2+} buffer with 70 μL ethanol to 100 μL of the 40 nM cross-linked sample in a 2 mL volume cuvette. A mass of 250 mg paraffin chips are added to the solution to act as a vapor barrier at temperatures above 55°C. Six inches of Teflon tape are used to seal the cuvette for further prevention of vapor leakage. The control and the cross-linked sample are placed in the sample chambers of the DBS at a 23°C initial temperature. The DBS is configured for absorbance readings at 260 nm with irises adjusted at each leg to attenuate the intensity. Both the exit slits and irises for each monochromator leg of the DBS are adjusted for

~3500 intensity counts in the SpectraSuite software with a 1-second integration time at 10 readings per average, and a 50 data point boxcar smoothing to minimize the signal-to-noise ratio at the wavelength. Psoralen cross-links can be broken if exposed to UV wavelengths below 300 nm at an excitation intensity above 1 W/cm^2 [111]. However, the DBS intensity at this wavelength is many orders of magnitude below this threshold, and it is assumed that this dosage is not achieved over the course of the four hour experiment.

The intensity data is numerically analyzed to calculate the absorbance data. A 2^{nd} order polynomial expression is fitted to each run with the coefficients averaged for each term. The first derivative of each averaged term is subtracted from the first derivative average of the fitted curves. The resulting derivative equation is reintegrated, and the values calculated using the initial intensity value acts as the constant offset. The absorbance values are calculated from the values determined from a previous control run using 1x TAE Mg^{2+} buffer with 10% volume ethanol. Another control is run using a single strand DNA oligonucleotide sequence to verify the proper operation of the DBS over the temperature range for the experiment.

The temperature water bath is programmed to ramp over a temperature range using the following steps: 1) ramp up from 23°C to 90°C over a 2 hour period, 2) maintain 90°C for 5 min, and 3) ramp down to 23°C over a 2 hour interval. A

thermocouple attached to the reference chamber monitors the temperature of the sample at 5 minute intervals.

G. Correction of Absorbance Data using a Polynomial Fit

The numerical analysis applies a 2nd order polynomial expression to correct the absorbance data assuming stable values over the duration of the temperature ramp. The following figures present data from a control buffer temperature ramp and a ramp using an ssDNA strand as the sample.

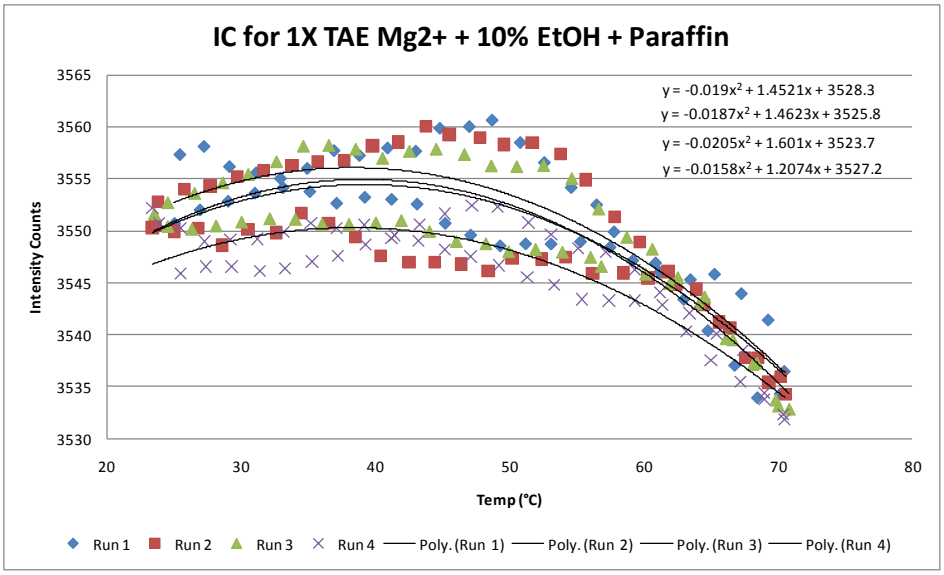


Figure 77: Smoothed IC data (3 point average) with 2nd order polynomial line fits for four successive temperature ramps.

In Figure 77, the general line fits have similar curves, as indicated in the four polynomial equations next to the figure. These expressions are processed to determine

the corrective terms by averaging the coefficients of these terms and taking their derivative to obtain the following expression:

$$y'_{avg} = -0.0368x + 1.42205$$

Equation 39

Taking the difference of the derivative of the line fit equations with Equation 39 gives the following correction equations:

$$y'_1 = -0.002x + 0.07165$$

$$y'_2 = 0.00965$$

$$y'_3 = -0.0032x + 0.13335$$

$$y'_4 = 0.0052x - 0.21465$$

Equation 40

Integrating the four equations above and reapplying the constants from the original equations, respectively, the plots are corrected for thermal expansion of the cuvette chamber over the course of the run. Figure 78 shows the corrected plots.

Validation of the numerical method is done by applying it to a similar run with an ssDNA oligonucleotide sequence. Figure B.3 shows the absorbance of the corrected runs for the single strand control using the solvent values shown in Figure 78 as the baseline.

The corrected baseline intensity counts can be used to reliably determine the absorbance values for the samples.

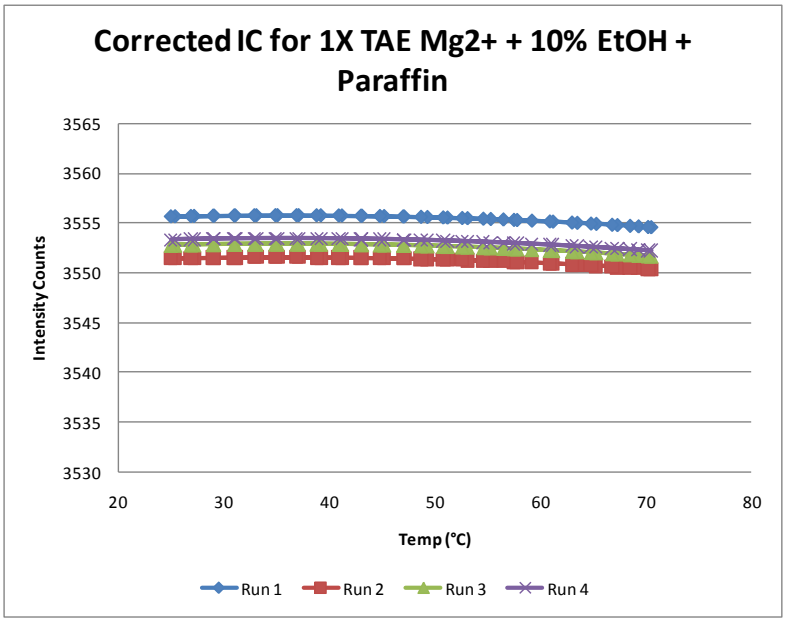


Figure 78: Corrected solvent plots after applying numerical analysis.

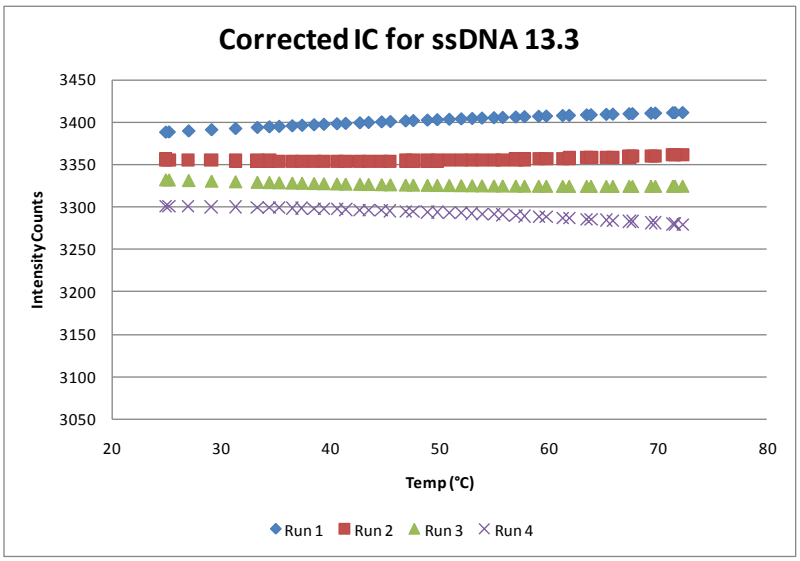


Figure 79: Corrected intensity count plots for the 13.3 ssDNA oligonucleotide using the solvent values as the baseline.

H. Activity Factor of Psoralen

The concentration of the TMP used during the intercalation process is 100 μM, a factor of 250 times greater than the nanostructure concentration (i.e., 40 nM). The required concentration of TMP for cross-linking of all sites can be determined by calculating the ratio of total cross-link sites available to the total number of base pairs in the nanostructure. There are 1412 base pairs in the 2 × 4 grid nanostructure, with 104 base pairs available for cross-linking, giving a ratio of 0.0737. The concentration of TMP required for a 40 nM sample is 2.9 nM to form one cross-link per nanostructure. The concentration is multiplied by the total number of cross-links to result in the required concentration for full cross-linking to be 306 nM. This value compared to the 10 μM concentration of TMP for the sample in the procedure is 32 times larger than the required concentration. Thus, a 100% activity factor for the TMP sample is assumed.

Bibliography

1. Ke, Y.G., et al., *Self-assembled water-soluble nucleic acid probe tiles for label-free RNA hybridization assays.* Science, 2008. **319**(5860): p. 180-183.

2. Watson, J.D. and F.H.C. Crick, *The Structure of DNA.* Cold Spring Harbor Symposia on Quantitative Biology, 1953. **18**: p. 123-131.

3. *Genomics Home Reference: The Structure of DNA*. 2007 Oct 12 2007 Oct 18 2007]; Available from: http://ghr.nlm.nih.gov/handbook/illustrations/dnastructure.

4. Bustamante, C., et al., *Entropic Elasticity of Lambda-Phage DNA.* Science, 1994. **265**(5178): p. 1599-1600.

5. Yan, H., et al., *Self-assembled DNA structures for nanoconstruction.* DNA-Based Molecular Electronics, 2004. **725**: p. 43-52, 116.

6. Gore, J., et al., *DNA overwinds when stretched.* Nature, 2006. **442**(7104): p. 836-839.

7. Bustamante, C., Z. Bryant, and S.B. Smith, *Ten years of tension: single-molecule DNA mechanics.* Nature, 2003. **421**(6921): p. 423-427.

8. Meiners, J.C. and S.R. Quake, *Femtonewton force spectroscopy of single extended DNA molecules.* Physical Review Letters, 2000. **84**(21): p. 5014-5017.

9. Bustamante, C., et al., *Single-molecule studies of DNA mechanics.* Current Opinion in Structural Biology, 2000. **10**(3): p. 279-285.

10. Quake, S.R., H. Babcock, and S. Chu, *The dynamics of partially extended single molecules of DNA.* Nature, 1997. **388**(6638): p. 151-154.

11. Marko, J.F. and E.D. Siggia, *Stretching DNA.* Macromolecules, 1995. **28**(26): p. 8759-8770.

12. Zhou, X.F., et al., *Direct measurement of compression spring constant of single DNA molecule with AFM.* Chinese Science Bulletin, 2005. **50**(10): p. 954-957.

13. Hall, A.R., et al., *Electrophoretic Force on a Protein-Coated DNA Molecule in a Solid-State Nanopore.* Nano Letters, 2009. **9**(12): p. 4441-4445.

14. Peyrard, M., *Nonlinear dynamics and statistical physics of DNA.* Nonlinearity, 2004. **17**(2): p. R1-R40.

15. Mathew-Fenn, R.S., R. Das, and P.A.B. Harbury, *Remeasuring the double helix.* Science, 2008. **322**(5900): p. 446-449.

16. Strick, T.R., et al., *Physical approaches to the study of DNA.* Journal of Statistical Physics, 1998. **93**(3-4): p. 647-672.

17. Ogden, R.W., G. Saccomandi, and I. Sgura, *Computational aspects of Worm-Like-Chain interpolation formulas.* Computers & Mathematics with Applications, 2007. **53**(2): p. 276-286.

18. Vesnaver, G. and K.J. Breslauer, *The Contribution of DNA Single-Stranded Order to the Thermodynamics of Duplex Formation.* Proceedings of the National Academy of Sciences of the United States of America, 1991. **88**(9): p. 3569-3573.

19. SantaLucia, J. and D. Hicks, *The thermodynamics of DNA structural motifs.* Annual Review of Biophysics and Biomolecular Structure, 2004. **33**: p. 415-440.

20. VorontsovVelyaminov, P.N., et al., *Free energy calculations by expanded ensemble method for lattice and continuous polymers.* Journal of Physical Chemistry, 1996. **100**(4): p. 1153-1158.

21. John, D.M. and K.M. Weeks, *van't Hoff enthalpies without baselines.* Protein Science, 2000. **9**(7): p. 1416-1419.

22. Zuker, M., *Mfold web server for nucleic acid folding and hybridization prediction.* Nucleic Acids Research, 2003. **31**(13): p. 3406-3415.

23. Kibbe, W.A., *OligoCalc: an online oligonucleotide properties calculator.* Nucleic Acids Research, 2007. **35**: p. W43-W46.

24. Markham, N.R. and M. Zuker, *DINAMelt web server for nucleic acid melting prediction.* Nucleic Acids Research, 2005. **33**: p. W577-W581.

25. Chaires, J.B. and J.M. Sturtevant, *Thermodynamics of the B to Z Transition in Poly(M5dg-Dc).* Proceedings of the National Academy of Sciences of the United States of America, 1986. **83**(15): p. 5479-5483.

26. Peck, L.J. and J.C. Wang, *Energetics of B-to-Z Transition in DNA.* Proceedings of the National Academy of Sciences of the United States of America-Biological Sciences, 1983. **80**(20): p. 6206-6210.

27. Chaires, J.B., *Inhibition of the Thermally Driven B to Z Transition by Intercalating Drugs.* Biochemistry, 1986. **25**(26): p. 8436-8439.

28. Gonzalez, O. and J. Li, *Modeling the sequence-dependent diffusion coefficients of short DNA molecules.* Journal of Chemical Physics, 2008. **129**(16): p. -.

29. Knotts, T.A., et al., *A coarse grain model for DNA.* Journal of Chemical Physics, 2007. **126**(8): p. -.

30. Sales-Pardo, M., et al., *Mesoscopic modeling for nucleic acid chain dynamics.* Physical Review E, 2005. **71**(5): p. -.

31. Resendis-Antonio, O., L.S. Garcia-Colin, and H. Larralde, *A statistical model of DNA denaturation.* Physica a-Statistical Mechanics and Its Applications, 2003. **318**(3-4): p. 435-446.

32. Drukker, K., G.S. Wu, and G.C. Schatz, *Model simulations of DNA denaturation dynamics.* Journal of Chemical Physics, 2001. **114**(1): p. 579-590.

33. Drukker, K. and G.C. Schatz, *A model for simulating dynamics of DNA denaturation.* Journal of Physical Chemistry B, 2000. **104**(26): p. 6108-6111.

34. Lavery, R. and A. Lebrun, *Modelling DNA stretching for physics and biology.* Genetica, 1999. **106**(1-2): p. 75-84.

35. Delatorre, J.G., J.J. Freire, and A. Horta, *Bead and Spring Model for Stiffness of DNA.* Biopolymers, 1975. **14**(7): p. 1327-1335.

36. Zhu, J.H., et al., *UNIQUIMER 3D, a software system for structural DNA nanotechnology design, analysis and evaluation.* Nucleic Acids Research, 2009. **37**(7): p. 2164-2175.

37. Liu, Z.R. and H.S. Chan, *Efficient chain moves for Monte Carlo simulations of a wormlike DNA model: Excluded volume, supercoils, site juxtapositions, knots, and comparisons with random-flight and lattice models.* Journal of Chemical Physics, 2008. **128**(14): p. -.

38. Bombelli, F.B., et al., *DNA Closed Nanostructures: A Structural and Monte Carlo Simulation Study.* Journal of Physical Chemistry B, 2008. **112**(48): p. 15283-15294.

39. Blake, R.D., et al., *Statistical mechanical simulation of polymeric DNA melting with MELTSIM.* Bioinformatics, 1999. **15**(5): p. 370-375.

40. Braun, E., et al., *DNA-templated assembly and electrode attachment of a conducting silver wire.* Nature, 1998. **391**(6669): p. 775-778.

41. Stoltenberg, R.M. and A.T. Woolley, *DNA-templated nanowire fabrication.* Biomedical Microdevices, 2004. **6**(2): p. 105-111.

42. Kudo, H. and M. Fujihira, *DNA-Templated copper nanowire fabrication by a two-step process involving electroless metallization.* Ieee Transactions on Nanotechnology, 2006. **5**(2): p. 90-92.

43. Richter, J., et al., *Construction of highly conductive nanowires on a DNA template.* Applied Physics Letters, 2001. **78**(4): p. 536-538.

44. Mertig, M., et al., *DNA as a selective metallization template.* Nano Letters, 2002. **2**(8): p. 841-844.

45. Gu, Q., et al., *DNA nanowire fabrication.* Nanotechnology, 2006. **17**(1): p. R14-R25.

46. Gu, Q. and D.T. Haynie, *Palladium nanoparticle-controlled growth of magnetic cobalt nanowires on DNA templates.* Materials Letters, 2008. **62**(17-18): p. 3047-3050.

47. Burley, G.A., et al., *Directed DNA metallization.* Journal of the American Chemical Society, 2006. **128**(5): p. 1398-1399.

48. Gu, Q., C.D. Cheng, and D.T. Haynie, *Cobalt metallization of DNA: toward magnetic nanowires.* Nanotechnology, 2005. **16**(8): p. 1358-1363.

49. He, Y., et al., *DNA-based nanofabrications.* Microscopy Research and Technique, 2007. **70**(6): p. 522-529.

50. Yang, T., et al., *Fabrication of linear aniline-DNA complex nanowires and DNA-templated polyaniline nanowires.* Chemical Journal of Chinese Universities-Chinese, 2006. **27**(6): p. 1126-1130.

51. Xiao, S.J., et al., *Selfassembly of metallic nanoparticle arrays by DNA scaffolding.* Journal of Nanoparticle Research, 2002. **4**(4): p. 313-317.

52. Li, H.Y., et al., *DNA-templated self-assembly of protein and nanoparticle linear arrays.* Journal of the American Chemical Society, 2004. **126**(2): p. 418-419.

53. Zheng, J.W., et al., *Two-dimensional nanoparticle arrays show the organizational power of robust DNA motifs.* Nano Letters, 2006. **6**(7): p. 1502-1504.

54. Park, S.H., et al., *Optimized fabrication and electrical analysis of silver nanowires templated on DNA molecules.* Applied Physics Letters, 2006. **89**(3): p. -.

55. Sebba, D.S., T.H. LaBean, and A.A. Lazarides, *Plasmon coupling in binary metal core-satellite assemblies.* Applied Physics B-Lasers and Optics, 2008. **93**(1): p. 69-78.

56. Park, S.H., et al., *Programmable DNA self-assemblies for nanoscale organization of ligands and proteins.* Nano Letters, 2005. **5**(4): p. 729-733.

57. Sanchez-Mosteiro, G., et al., *DNA-based molecular wires: Multiple emission pathways of individual constructs.* Journal of Physical Chemistry B, 2006. **110**(51): p. 26349-26353.

58. Tinnefeld, P., M. Heilemann, and M. Sauer, *Design of molecular photonic wires based on multistep electronic excitation transfer.* Chemphyschem, 2005. **6**(2): p. 217-222.

59. Frezza, B.M., S.L. Cockroft, and M.R. Ghadiri, *Modular multi-level circuits from immobilized DNA-Based logic gates.* Journal of the American Chemical Society, 2007. **129**(48): p. 14875-14879.

60. Tang, Y.L., et al., *Multiply configurable optical-logic systems based on cationic conjugated polymer/DNA assemblies.* Advanced Materials, 2006. **18**(16): p. 2105-+.

61. Dwyer, C., A.R. Lebeck, and C. Pistol. *Energy Transfer Logic on DNA Nanostructures: Enabling Molecular-Scale Amorphous Computing*. in *4th Workshop on Non-Silicon Computing (NSC4)*. 2007.

62. Manning, B., et al., *DNA-templated assembly of nanoscale wires and switches.* Opto-Ireland 2005: Nanotechnology and Nanophotonics, 2005. **5824**: p. 93-101, 316.

63. Pistol, C., A.R. Lebeck, and C. Dwyer. *Design Automation for DNA Self-Assembled Nanostructures*. in *43rd Design Automation Conference (DAC)*. 2006.

64. Park, S.H., et al., *Finite-size, fully addressable DNA tile lattices formed by hierarchical assembly procedures.* Angewandte Chemie-International Edition, 2006. **45**(5): p. 735-739.

65. Liu, W.Y., et al., *PX DNA triangle oligomerized using a novel three-domain motif.* Nano Letters, 2008. **8**(1): p. 317-322.

66. He, Y., et al., *Highly connected two-dimensional crystals of DNA six-point-stars.* Journal of the American Chemical Society, 2006. **128**(50): p. 15978-15979.

67. Rothemund, P.W.K., *Folding DNA to create nanoscale shapes and patterns.* Nature, 2006. **440**(7082): p. 297-302.

68. Aldaye, F.A. and H.F. Sleiman, *Sequential self-assembly of a DNA hexagon as a template for the organization of gold nanoparticles.* Angewandte Chemie-International Edition, 2006. **45**(14): p. 2204-2209.

69. Pistol, C. and C. Dwyer, *Scalable, low-cost, hierarchical assembly of programmable DNA nanostructures.* Nanotechnology, 2007. **18**(12): p. -.

70. He, P.G., S.N. Li, and L.M. Dai, *DNA-modified carbon nanotubes for self-assembling and biosensing applications.* Synthetic Metals, 2005. **154**(1-3): p. 17-20.

71. Venkataraman, S., et al., *Automated image analysis of atomic force microscopy images of rotavirus particles.* Ultramicroscopy, 2006. **106**(8-9): p. 829-837.

72. Ficarra, E., et al., *Automated DNA fragments recognition and sizing through AFM image processing.* Ieee Transactions on Information Technology in Biomedicine, 2005. **9**(4): p. 508-517.

73. Wilson, D.L., et al., *Morphological Restoration of Atomic-Force Microscopy Images.* Langmuir, 1995. **11**(1): p. 265-272.

74. Marek, J., et al., *Interactive measurement and characterization of DNA molecules by analysis of AFM images.* Cytometry Part A, 2005. **63A**(2): p. 87-93.

75. Spisz, T.S., et al., *Automated sizing of DNA fragments in atomic force microscope images.* Medical & Biological Engineering & Computing, 1998. **36**(6): p. 667-672.

76. NIH, U.S. *NIH Image*. Available from: http://rsb.info.nih.gov/nih-image/.

77. Piskur, J. and A. Rupprecht, *Aggregated DNA in Ethanol Solution.* Febs Letters, 1995. **375**(3): p. 174-178.

78. Su, T.X. and P.K. Purohit, *Thermomechanics of a heterogeneous fluctuating chain.* Journal of the Mechanics and Physics of Solids, 2010. **58**(2): p. 164-186.

79. Nkodo, A.E., et al., *Diffusion coefficient of DNA molecules during free solution electrophoresis.* Electrophoresis, 2001. **22**(12): p. 2424-2432.

80. Nkodo, A.E. and B. Tinland, *Simultaneous measurements of the electrophoretic mobility, diffusion coefficient and orientation of dsDNA during electrophoresis in polymer solutions.* Electrophoresis, 2002. **23**(16): p. 2755-2765.

81. Mercier, J.F. and G.W. Slater, *Universal interpolating function for the dispersion coefficient of DNA fragments in sieving matrices.* Electrophoresis, 2006. **27**(8): p. 1453-1461.

82. Tataurov, A.V., Y. You, and R. Owczarzy, *Predicting ultraviolet spectrum of single stranded and double stranded deoxyribonucleic acids.* Biophysical Chemistry, 2008. **133**(1-3): p. 66-70.

83. Vanzandt, L.L., *Damping of DNA Vibration Modes by Viscous Solvents.* International Journal of Quantum Chemistry, 1981: p. 271-276.

84. Delcourt, S.G. and R.D. Blake, *Stacking Energies in DNA.* Journal of Biological Chemistry, 1991. **266**(23): p. 15160-15169.

85. Pistol, C., C. Dwyer, and A.R. Lebeck, *Nanoscale Optical Computing Using Resonance Energy Transfer Logic.* Ieee Micro, 2008. **28**(6): p. 7-19.

86. Pistol, C., et al., *Architectural Implications of Nanoscale Integrated Sensing and Computing.* Acm Sigplan Notices, 2009. **44**(3): p. 13-24.

87. Freedman, K.O., et al., *Diffusion of single star-branched dendrimer-like DNA.* Journal of Physical Chemistry B, 2005. **109**(19): p. 9839-9842.

88. McGown, E., M. Su, and R. Dennis, *Measurement of molecular beacons in the SRECTRAmax (R) GEMINI spectrofluorometer.* Journal of Chemical Technology and Biotechnology, 2000. **75**(10): p. 942-944.

89. Gornushkin, I.B., et al., *High-resolution two-grating spectrometer for dual wavelength spectral imaging.* Applied Spectroscopy, 2004. **58**(11): p. 1341-1346.

90. Franko, M. and C.D. Tran, *Development of a Double-Beam, Dual-Wavelength Thermal-Lens Spectrometer for Simultaneous Measurement of Absorption at 2 Different Wavelengths.* Analytical Chemistry, 1988. **60**(18): p. 1925-1928.

91. Lakowicz, J.R., *Principles of Fluorescence Spectroscopy*. 2nd ed. 1999, New York: Plenum Publishing.

92. Kim, K., K. Matsuura, and N. Kimizuka, *Binding of lectins to DNA micro-assemblies: Modification of nucleo-cages with lactose-conjugated psoralen.* Bioorganic & Medicinal Chemistry, 2007. **15**(12): p. 4311-4317.

93. Cohen, L.F., et al., *Interstrand DNA Crosslinking by 4,5',8-Trimethylpsoralen Plus Monochromatic Ultraviolet-Light - Studies by Alkaline Elution in Mouse L1210 Leukemia-Cells.* Biochimica Et Biophysica Acta, 1980. **610**(1): p. 56-63.

94. Arabzadeh, A., et al., *Studies on mechanism of 8-methoxypsoralen - DNA interaction in the dark.* International Journal of Pharmaceutics, 2002. **237**(1-2): p. 47-55.

95. Brownfield, J. and S. Collins, *The luminescence spectra of the 8-methoxypsoralen excited-state complexes and photochemical product in argon, methanol/argon, and water/argon matrices at 10 K.* Journal of Physical Chemistry A, 2000. **104**(16): p. 3759-3763.

96. Ramaswamy, M. and A.T. Yeung, *The Reactivity of 4,5',8-Trimethylpsoralen with Oligonucleotides Containing at Sites.* Biochemistry, 1994. **33**(18): p. 5411-5413.

97. Takasugi, M., et al., *Sequence-Specific Photoinduced Cross-Linking of the 2 Strands of Double-Helical DNA by a Psoralen Covalently Linked to a Triple Helix-Forming Oligonucleotide.* Proceedings of the National Academy of Sciences of the United States of America, 1991. **88**(13): p. 5602-5606.

98. Hearst, J.E., *Psoralen Photochemistry.* Annual Review of Biophysics and Bioengineering, 1981. **10**: p. 69-86.

99. Cecchini, S., et al., *Interstrand Cross-Link Induction by UV Radiation in Bromodeoxyuridine-Substituted DNA: Dependence on DNA Conformation.* Biochemistry, 2005. **44**(51): p. 16957-16966.

100. Wojcik, A., et al., *DNA interstrand crosslinks are induced in cells prelabelled with 5-bromo-2 '-deoxyuridine and exposed to UVC radiation.* Journal of Photochemistry and Photobiology B-Biology, 2006. **84**(1): p. 15-20.

101. Smith, S.B., Y.J. Cui, and C. Bustamante, *Overstretching B-DNA: The elastic response of individual double-stranded and single-stranded DNA molecules.* Science, 1996. **271**(5250): p. 795-799.

102. Yeung, A.T., B.K. Jones, and C.T. Chu, *Photoreactivities and Thermal-Properties of Psoralen Cross-Links.* Biochemistry, 1988. **27**(9): p. 3204-3210.

103. Tagawa, M., et al., *Heat-resistant DNA tile arrays constructed by template-directed photoligation through 5-carboxyvinyl-2 '-deoxyuridine.* Nucleic Acids Research, 2007. **35**(21): p. -.

104. Marzano, C., E. Severin, and F. Bordin, *Can a mixed damage interfere with DNA-protein cross-links repair?* Journal of Cellular and Molecular Medicine, 2001. **5**(2): p. 171-177.

105. *RSCB Protein Databank*. 2007.

106. Wiesehahn, G. and J.E. Hearst, *DNA Unwinding Induced by Photoaddition of Psoralen Derivatives and Determination of Dark-Binding Equilibrium Constants by Gel Electrophoresis.* PNAS, 1978. **75**(6): p. 2703-2707.

107. Shi, Y.B. and J.E. Hearst, *Thermostability of Double-Stranded Deoxyribonucleic Acids - Effects of Covalent Additions of a Psoralen.* Biochemistry, 1986. **25**(20): p. 5895-5902.

108. Shi, Y.B. and J.E. Hearst, *Wavelength Dependence for the Photoreactions of DNA-Psoralen Monoadducts .2. Photo-Cross-Linking of Monoadducts.* Biochemistry, 1987. **26**(13): p. 3792-3798.

109. Sogo, J.M. and F. Thoma, *Electron Microscopy of Chromatin.* Methods in Enzymology, 1989. **170**: p. 142-165.

110. Kumaresan, K.R., M. Ramaswamy, and A.T. Yeung, *Structure of the DNA Interstrand Cross-Link of 4,5',8-Trimethylpsoralen.* Biochemistry, 1992. **31**(29): p. 6774-6783.

111. Cimino, G.D., Y.B. Shi, and J.E. Hearst, *Wavelength Dependence for the Photoreversal of a Psoralen-DNA Cross-Link.* Biochemistry, 1986. **25**(10): p. 3013-3020.

112. Xiong, A.S., et al., *PCR-based accurate synthesis of long DNA sequences.* Nature Protocols, 2006. **1**(2): p. 791-797.

113. Cui, Y., et al., *Nanowire nanosensors for highly sensitive and selective detection of biological and chemical species.* Science, 2001. **293**(5533): p. 1289-1292.

114. Hernandez, R.M., et al., *Template fabrication of protein-functionalized gold-polypyrrole-gold segmented nanowires.* Chemistry of Materials, 2004. **16**(18): p. 3431-3438.

115. Dwyer, C., et al., *Design Tools for a DNA-guided Self-assembling Carbon Nanotube Technology.* Nanotechnology, 2004. **15**: p. 1240-1245.

116. Patwardhan, J.P., et al., *NANA: a Nano-scale Active Network Architecture.* J. Emerg. Technol. Comput. Syst., 2006. **2**(1): p. 1-30.

117. Patwardhan, J.P., et al. *A Defect Tolerant Self-organizing Nanoscale SIMD Architecture*. in *12th International Conference on Architectural Support for Programming Languages and Operating Systems*. 2006.

118. Dooley, K.A., et al., *montalcino, A zebrafish model for variegate porphyria.* Experimental Hematology, 2008. **36**(9): p. 1132-1142.

Biography

Vincent Mao was born on November 25, 1982 in Columbia, South Carolina. He received his Bachelor's of Science and Engineering in Electrical Engineering and Biomedical Engineering from Duke University in 2005. Upon graduation, he began his graduate research under the direction of Dr. Christopher Dwyer in the area of computer engineering, with a focus on design and fabrication of DNA self-assembled nanostructures. In 2008, he received a Master of Science degree, also from Duke University, presenting his work on connecting defects in DNA self-assembled nanostructures to corresponding fault models at the International Test Conference (ITC) at Santa Clara, California that year. The work was listed as one of the top ten papers in the ITC conference proceedings of 2009.

Selected Publications:

Encoded Multichromophore Response for Simultaneous Label-Free Detection, Pistol, C., V. Mao, et al. (2010). Small **6**(7): 843-850.

Connecting Fabrication Defects to Fault Models and SPICE Simulations for DNA Self-Assembled Nanoelectronics, V. Mao, V. Thusu, C. Dwyer, K. Chakrabarty. (2009) IET Computers & Digital Techniques **3**(6): 553-569.

Fabrication Defects and Fault Models for DNA Self-Assembled Nanoelectronics, V. Mao, C. Dwyer, K. Chakrabarty. (2008) IEEE International Test Conference Proceedings