A Nonlinear Parallel Model for Reversible Polymer Solutions in Steady and Oscillating Shear Flow

Erik Tracey Palmer

A Nonlinear Parallel Model for Reversible Polymer Solutions in Steady and Oscillating Shear Flow

by

Erik Tracey Palmer

Bachelor of Arts
University of California, Davis 2007

Master of Science
California State University, East Bay 2013

Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy in

Mathematics

College of Arts and Sciences

University of South Carolina

2019

Accepted by:

Acknowledgments

I would first like to thank my wife, Yulian Wu. Without her this would have never happened. To my children, thank you for sacrificing our time together. I am also thankful for the contributions from other family members. My mother and father unwaveringly supported my many educational pursuits over the years. My parents-in-law spared no effort in helping to get this project finished – particularly my mother-in-law. Her presence at crucial times allowed me space and energy to focus. My uncle, Owen Palmer, stepped up at several critical moments to offer timely support. I would like to thank him for responding with excessive patience and the willingness to invest time in my mathematical dilemmas.

I also want to thank a wonderful network of friends whose support was absolutely necessary. In particular, my friend Russell Mills Campisi, called me daily to keep me motivated and focused on my goals. My wife's colleagues shared vast amounts of insider information and graduate school expertise as well as good food and drinks.

Because creeks flow into streams and streams into rivers, I would like to thank Moon Duchin. Her encouragement in a History of Mathematics class launched this journey. My experiences at Cal State East Bay were also crucial to the success of this project. Therefore, I would like to thank Stuart Smith who forged the mindset I needed to be successful, Julia Olkin for her support and Shirley Yap who told me 'As long as you want to do math you will be able to'. My brief time as a summer intern at the Lawrence Berkeley Lab provided a lift that carried me through to the end of this project. For this reason, I want to thank Ann Almgren, Andy Nonaka and everyone else at the Center for Computational Sciences and Engineering.

I am also thankful for the many friendships I developed with my fellow graduate students. Classmates Xueping Zhao and Sameed Ahmed were supportive in every way and remain so to this day. With other classmates, I was inspired by their joy and motivated by their passion. My thanks to those who worked with me and listened to my ideas. These opportunities gave purpose to my studies. In particular, I want to express special thanks to the fellow students who stayed up late the night before my defense to give feedback and then woke up the next morning to show support by listening again. In all, our graduate student cohort looked out for one another and fostered an atmosphere I was happy to be a part of.

Finally, I would like to give thanks to my dissertation committee and the rest of the mathematics department for their support during this endeavor.

Erik Palmer

June 28, 2019

Abstract

A mathematical model for reversible polymers in steady and oscillating shear flows is presented. Using a mean-field approach, the behavior of the polymer network is characterized by a finitely extensible nonlinear elastic bead-spring model that stochastically transitions between dumbbell states to represent attachments, detachments and loops. An efficient parallel scheme for computation on GPUs utilizes populations of over a million dumbbells to characterize steady, large and small amplitude oscillatory shear (SAOS) flows in Brownian dynamics simulations. In steady-shear a novel attachment species transition function enables shear thickening and shear thinning by the adjustment of either attachment or detachment parameters. Three species simulations show the inclusion of loops modifies the strength of these nonlinear flow responses. In SAOS simulations, three species simulations show an increase in dynamic moduli at higher frequencies not present in two species models. Two approaches for a looped segment transitioning to dangling are explored, and the choice found to have substantial impact on the effect of adding a third species. Pipkin diagrams are also generated using large amplitude oscillatory flows.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Associative polymer models for viscoelastic fluids have held the attention of the scientific community for over 50 years. Interest remains high due to the numerous applications of transient polymers in industrial applications where control over their rheological behavior is crucial for achieving ideal performance. Transient polymers also serve important roles in the biomedical community as the material for soft sensors, drug carriers and in tissue engineering, just to name a few [2, 54, 24]. They are categorized as associative polymers due to the reversible cross-links of their molecular network that give rise to unique characteristics such as shear-thinning and shear-thickening.

The work of this dissertation focuses on the rheological behavior of tri-block polymers above the micelle concentration [42, 45, 34] . Tri-block polymers are defined as molecular chains that consist of a B-A-B configuration where the B-Blocks are hydrophobic and the A-blocks are hydrophilic. Above a certain concentration, interactions with the solvent cause the polymer chains to form micelles. At higher concentrations, thermodynamic vibration and flow forces then lead to bridging between micelles as the hydrophobic ends break out of their micelle cores and form bridges by embedding their hydrophobic ends in other micelle cores. Conversely, the same forces cause the destruction of bridges as polymer ends disassociate from either core. These mechanical behaviors are responsible for the dynamic characteristics of the polymer and is what leads scientists to classify them as telechelic or reversible associative polymers.

Since Green and Tobolsky [19] numerous models have been presented in an attempt to capture the characteristic behaviors of telechelic polymer networks [66, 65, 55, 62, 63, 64, 48, 49, 50] . An early history of constitutive models is well covered in the introduction of Tripathi et al. [51] and a thorough discussion in Wang and Larson [56] provides a recent update. Of present interest is the work done by Vaccaro and Marrucci in [53]. Based on theory and simulation results from van den Brule and Hoogerbrugge [8] , they model this system as finitely extensible dumbbells separated into two groups, active and dangling. Each segment has its own characteristics and thus effects the polymer network structure differently. In simulation, dumbbells stochastically switch between states. Representing each state as a single Fokker-Planck equation creates a system of equations which can be solved using closure approximations. A systematic evaluation of this approach is undertaken by [38]. On the other hand, Hernández Cifre [22] takes a Brownian Dynamics approach. Instead of Fokker-Plank equations, a Langevin equation describes the micro-scale dynamics of each dumbbell type. Macro-scale terms, such as the fluid stress, are then determined by averaging over many realizations -typically ensembles of 5000 dumbbells. In this way, Hernández Cifre shows the viscosity profile of associative polymers in simple shear flow can be captured without the closure approximations needed in other approaches.

In a more recent evolution of Vaccaro and Marrucci's approach, Sing et al [44]. adds a third looped dumbbell species. Through analysis of their three species Fokker -Plank system, they conclude that the inclusion of loops causes non-monotonic shear thickening and shear thinning at lower stresses. However, this type of simulation maybe more difficult to adapt to complex scenarios [32]. In this light, a Brownian Dynamics micro-macro scale approach provides a useful alternative.

A molecular dynamics model by Baljon et al. simulates telechelic polymers in small amplitude oscillatory shear flow and produces nonlinear stress responses [3, 60]. Their model is an extension of earlier work by Kremer and Grest [20, 29] where dynamics are driven by temperature and inter-chain interactions via energy potentials. The model by Baljon et al. adds the ability for endgroups on each chain to associate and detach from each other via Monte Carlo Dynamics. They simulate systems of 100 polymer chains with eight beads each. However, as they consider pairwise interactions between chains in the system, they are limited in the number of chains and beads they can model efficiently. To generate rheological data from SAOS flow, they attach polymer chains to an oscillating boundary and rely on strain rate-frequency superposition [61] to reconstruct a complete flow curve. While the complexities of this model make this admirable work, they also underscore the need for an efficient and straight forward method.

Although interest in elastic dumbbell models is high, the application of Brownian Dynamics models to associating polymers has not kept pace with advances in experimental techniques used to measure their rheology. In recent decades, small amplitude oscillatory shear (SAOS) has become the canonical method for rheological measurements [25]. Moreover, the developing field of non-linear rheology also relies on large amplitude oscillatory shear (LAOS) fluid flows that are more complex than steady shear and require additional sensitivity. However, few Brownian Dynamics models have yet to demonstrate capability in capturing rheological characteristics in this type of flow. The goal of this work is to present a micro-macro scale simulation capable of modelling associative polymers in steady, and small to large oscillatory shear flows. To this aim we present an extension of the mean-field method set forth in Hernández Cifre et al. and apply it to the more dynamic shear flows used in modern rheological measures. The parallel computation scheme allows for an efficient simulation of over a million dumbbells in the system and produces full flow curves with little stochastic noise or the need for additional frequency extension techniques. The inclusion of a third looping dumbbell species also creates a more dynamic non-linear fluid response and results in a wider range of dynamic moduli. Finally, the additional fidelity makes it possible to simulate large amplitude oscillatory shear measurements for a wide range of frequencies and flow rates.

Chapter 2

Model Description

Our model builds off the approach taken by Hernández Cifre in [22] by adding a third species of dumbbells and adopting modifications for efficient parallel computation. The key differences are the use of nonlinear elastic finitely extendable (FENE) dumbbells, reworked species transition probability functions, including an adaptation of Sing's method for incorporating looped dumbbells, and the use of a constant instantaneous lifetime for dumbbell chains in the network. The most important feature of the model is the Mean-Field approach. In this approach, explicit positions for dumbbells and their connections are not tracked; forgoing direct tracking of the network topology. Instead, each dumbbell follows a stochastic differential equation according to its species type that also varies stochastically according to prescribed rules based on the length. These characteristics allow parallel computation of each dumbbell configuration and thus result in efficient modelling of large populations.

2.1 Block Copolymers

Our modelling approach is based on the structure of BAB block copolymers in solvent. BAB block copolymers are made of long molecule chains with a backbone of two segments, B and A, that show differing preference for the solvent.

For example, for a BAB copolymer in water, the A block is hydrophobic and the B block is hydrophilic.

At concentrations above the critical micelle concentration (CMC), the varying preference for water causes the formation of a core of A-type blocks as they attempt to

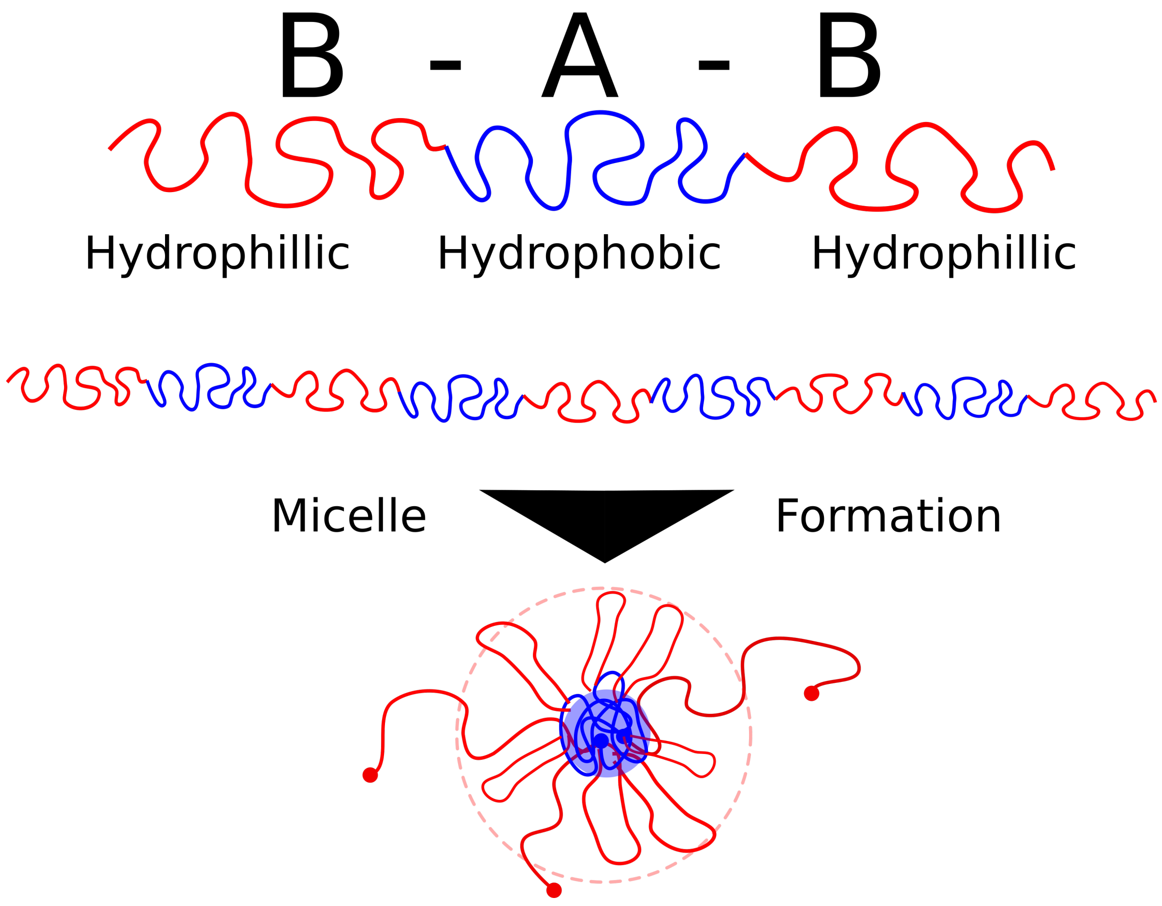

Figure 2.1: Cartoon of BAB-block copolymer. Hydrophobic and hydrophilic blocks in the polymer chain respond to the solvent by forming micelles.

avoid contact with the surrounding fluid. Meanwhile, because the B-type blocks prefer water, they are pushed outward forming a corona around the core. The resulting structure resembles a flower in two dimensions and is called a micelle. Figure 2.1 shows a cartoon representation of this description. Micelle formation is well studied for a variety of polymers [42, 45, 34, 4].

Precise measures at the molecular level for block copolymers can be challenging to obtain. In the case of the telechelic polymer, hydrophobically modified ethoxylated urethane (HEUR) with a backbone of polyethylene oxide (PEO), there are several factors which affect aggregate size, such as temperature, concentration and molecular weight. In one study by Kadam et al. using polyethylene oxide as the center of a triblock micelle forming polymer [26], aggregate sizes had a radius of gyration of that

Table 2.1: Experimental measurements for triblock polymers containing PEO. All measurements where taken at 20°C. [a]Data from Kadam et al. [26]. [b]Data from Zhao et al. [67] with reported relative error: M_w, ±5%; R_h, ±2%; R_g, ±8%.

Copolymer	Concentration	$M_w\,(g/mol)$	$R_h\,(nm)$	$R_g\,(nm)$
PMEA-PEO-PMEA$_a$	$1\,g/L$	3.60×10^5	13	-
PMEA-PEO-PMEA$_a$	$2\,g/L$	3.80×10^5	14	-
PMEA-PEO-PMEA$_a$	$4\,g/L$	5.90×10^5	17	-
PMEA-PEO-PMEA$_a$	$8\,g/L$	4.61×10^6	55	65
PMEA-PEO-PMEA$_a$	$10\,g/L$	1.48×10^7	74	103
PMEA-PEO-PMEA$_a$	$12\,g/L$	2.85×10^7	95	144
PCL-PEO-PCL$_b$	$2.74 \times 10^{-6}\,g/mL$	4.51×10^6	52	39
PCL-PEO-PCL$_b$	$2.74 \times 10^{-6}\,g/mL$	1.04×10^7	53	49
PCL-PEO-PCL$_b$	$2.74 \times 10^{-6}\,g/mL$	3.03×10^7	56	54

ranged from 65 nm to 144 nm while the hydrodynamic radius ranged from 13 nm to 95 nm. In another study by Zhao et al. they find radius of the formed micelles shrink as the temperature increases [67]. Moreover, they measure a radius of gyration from 54 nm to 39 nm and a hydrodynamic radius from 56 nm to 52 nm with errors of ±5%, ±8% and ±2% respectively. The table 2.1 contains complete measurements from these works.

In concentrations above the CMC, networks form as micelles come into increasing contact with one another and entangle. The focus of our model is at these concentrations, where micelle network attachment and detachment play a role in the mechanic response of the fluid. The progression of our model representation is illustrated in figure 2.2. On the far left is an example polymer network from Nykänen et al. [37]. The center illustration shows a cartoon version, with micelles displaying three features: 1) entanglements with other micelles; 2) dangling polymer chains, where one end is embedded in the micelle core and the other end explores the surrounding fluid; and 3) looping chains where both ends embed in the same micelle core. In our model, these three types of segments are classified as active (bridges), dangling and looped species types. On the right is a network diagram of the cartoon. In this diagram

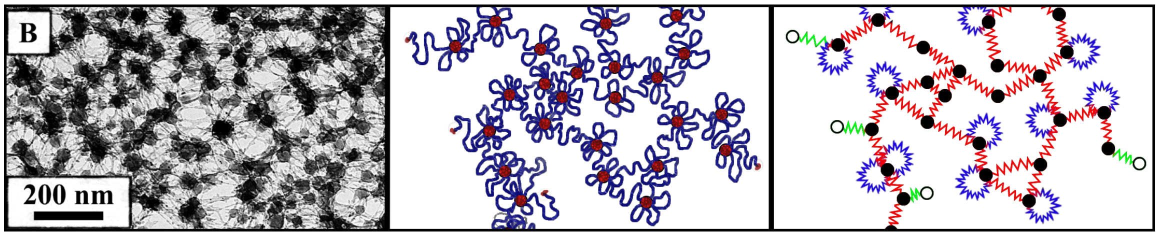

Figure 2.2: Visual representation of the model conceptualization process; from experimental form (*left*), cartoon representation (*center*), to network diagram (*right*). *Left.* Image of a block copolymer hydrogel sample from Nykänen et al. [37]. *Center.* Cartoon representation of a polymer network showing fully formed micelles connected by polymer chains, as well as dangling and looped segments. *Right.* Network diagram containing the three species types used in the model: active (red), dangling (green), and looped (blue).

each segment is clearly represented and color coded by species. Each segment is represented as a dumbbell with two endpoints connected by a spring. The endpoints are considering sticking points, where attaching and detaching to other nodes is possible. Endpoints can be micelle cores or the end of a free dangling polymer chain.

Our model will focus on tracking segment configuration dynamics and species type. Because fluid stress is a force per unit of area, and we consider all the parts of our network to be within sufficient proximity to one another to avoid unique considerations, we make the modelling assumption that is it not necessary to track the position of each segment in the network in order to resolve overall fluid stress in the cell. Using this, we approximate network attachments, detachments and looping with stochastic functions of the dumbbell length. Therefore the main mathematical concerns of our approach are an equation to track segment configuration (length and orientation but not position) and species transition probability functions. This focus allows for each segment to be evolved over time independently of the behavior of other dumbbells and results in gains in computational efficiency through parallel calculation. However, it also presents a challenging conceptual issue —a network model with no position.

There are several other modelling assumptions worth mentioning. First, hydrody-

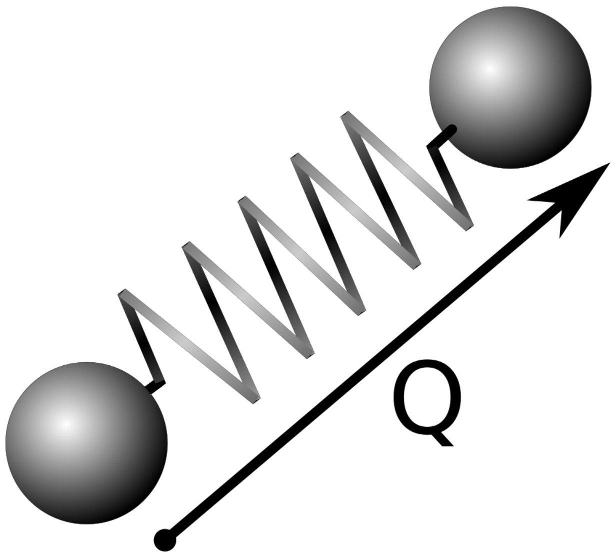

Figure 2.3: Elastic dumbbell. Dumbbells represent each, endpoint-edge-endpoint, segment of the polymer network. Notice that $\boldsymbol{Q}$ tracks only the configuration (length and orientation) and not the location of the segment.

namic interactions with the solvent are neglected. Second, we assume a free draining polymer network. That is, the dumbbells themselves do not impede the movement of the Brownian particles. Third, there are no boundaries. Therefore the model assumes the fluid cell being simulated is a sufficient distance from any wall, so that special boundary conditions are a non-factor.

2.2 Dumbbell Evolution Equation

The stochastic differential equation (SDE) describing the evolution of each dumbbell is,

$$\boldsymbol{Q}(t+\Delta t) = \boldsymbol{Q}(t)+\boldsymbol{\kappa}\cdot\boldsymbol{Q}(t)\Delta t-\left(\frac{\zeta_i+\zeta_j}{\zeta_i\zeta_j}\right)\boldsymbol{F}(\boldsymbol{Q})\Delta t+\sqrt{2k_BT\left(\frac{\zeta_i+\zeta_j}{\zeta_i\zeta_j}\right)}\Delta\boldsymbol{W}(t). \quad (2.1)$$

Here Q is a vector the represents the end-to-end length and angle (or configuration) of a single dumbbell, $\boldsymbol{\kappa}$ is the fluid velocity tensor, $\boldsymbol{F}(\boldsymbol{Q})$ is the FENE spring force, k_B is the Boltzmann constant and T is the temperature. The term $\boldsymbol{W}(t)$ is a Weiner process representing Brownian motion as a result of particle-solvent interactions at the ends of the dumbbell. Equations for active and dangling species type differ by the assigned drag terms. For active dumbbells, $\zeta_i = \zeta_j = \zeta_{node}$ and for dangling dumbbells, $\zeta_i = \zeta_{node}$ and $\zeta_j = \zeta_{free}$. Dumbbells in the looped state are considered to have negligible interactions with the fluid, and thus their configuration does not change until they return to a dangling state.

2.3 Attached, Dangling and Looped Transition Probability Functions

Three species types represent different states of a polymer chain under consideration. Attached dumbbells represent a BAB polymer chain where each sticky end (A-block) is attached to separate micelle cores. The dangling type describes a polymer chain, where a single sticky end is embedded in a micelle core, and rest of the polymer chain dangles freely in the solution. A looped type describes the state where the polymer chain has both ends embedded in its own micelle core; looping back upon itself. Each dumbbell evaluates its species type once per simulated time step, and changes according to the map,

$$\text{Active} \leftrightarrow \text{Dangling} \leftrightarrow \text{Looped}.$$

The active to dangling species transition models the situation where a polymer chain bridging two micelle cores separates from one core. A dangling to active transition represents the free end of a dangling chain embedding into the micelle core of another (unspecified) micelle core. The dangling to looped transition occurs when the free end of a polymer chain loops back upon itself. The looped to dangling transition indicates a single end of a looped chain breaking out of the micelle core to dangle freely in the solvent.

In simulation, each dumbbell species type follows a prescribed evolution equation according to the characteristics of its current state. After evolving orientation and length forward in time, transitions between species states occur according to dynamics unique to each type and are determined via stochastic comparison. First, a transition probability function that may depend on the length of the dumbbell, determines the likelihood of a transition occurring. Then random variables are generated and compared to the values from the transition functions. If the transition probability function value is higher than the generated uniform random variable, a species transition occurs. The following subsections describe the transition probability functions in greater detail.

2.3.1 Active Dumbbells

The active-to-dangling ($A \to D$) state transition function models the process of two micelles separating through the detachment of a molecular chain. The approach taken here arose out of a combined analysis of the methods used in Hernández Cifre et al. [22] and Sing et al. [44, 43] In Hernández Cifre et al., they model the attractive force between the chemical bonds as a potential well with the shape of a parabola. By balancing the energy to escape the well with the FENE force of the spring, they derive the expression given here in nondimensional form,

$$P_{A \to D} = 1 - \exp\left[-\frac{2\Delta t}{\tau_{fund}\exp(U_0)\exp\left(-\frac{d^2}{U_0}\frac{Q^2}{(1-Q^2/Q_{max}^2)^2}\right)}\right]. \tag{2.2}$$

On the other hand, Sing et al, cites Bell's law [5] for the rate of dissociation. This approach is derived from reaction rates and the lifetime of a bond from the kinetic theory of solids [68]. Together Bell's approach relates the rate of bond breakage to the strength of the force between potential bonding sites. In Sing et al, the rate of dissociation is presented as,

$$R_{\text{bridge dissociation}} = k_d \exp\left(B\left|\frac{Q}{1 - Q^2/Q_{max}^2}\right|\right) \tag{2.3}$$

By converting Eq. 2.3 to a probability and rearranging Eq. 2.2 we can compare the two probability transition functions as,

$$1 - \exp\left[-k_d \exp\left(B\left|\frac{Q}{1 - Q^2/Q_{max}^2}\right|\right)\Delta t\right] \tag{2.4}$$

and,

$$= 1 - \exp\left[-\frac{2}{\tau_{fund}\exp(U_0)}\exp\left(\frac{d^2}{U_0}\frac{Q^2}{(1 - Q^2/Q_{max}^2)^2}\right)\Delta t\right] \tag{2.5}$$

for the Sing et al. and Hernández Cifre et al. approaches, respectively. In this form we can see that these two approaches are similar but irreconcilable due to the squared FENE force dependence in Eq. 2.5 versus a non-squared dependence in Eq. 2.4.

For our model we chose to go with the expression from Sing et al. The dimensionless characteristic bond length, B, was set to 0.0325 to create transition probability curves similar to those using the approach and default parameters in Hernández Cifre et al. The parameter β is used to adjust overall rates of dissociation. Thus the probability transition function used in our simulations is,

$$P_{A\to D} = 1 - \exp\left[-\beta \exp\left(B\left|\frac{Q}{1 - Q^2/Q_{max}^2}\right|\right)\Delta t\right]. \tag{2.6}$$

2.3.2 Dangling Dumbbells

Dangling dumbbells can transition to the active (A) or (L) looped types. Due to this added complexity, a multistep process was used to determine the proper form of each transition probability function. First, transition probability functions for the two transitions, dangling-to-active and dangling-to-looped were considered independently. An asterisk, *, is used to denote these functions. Then these probabilities were combined using a two random variable scheme.

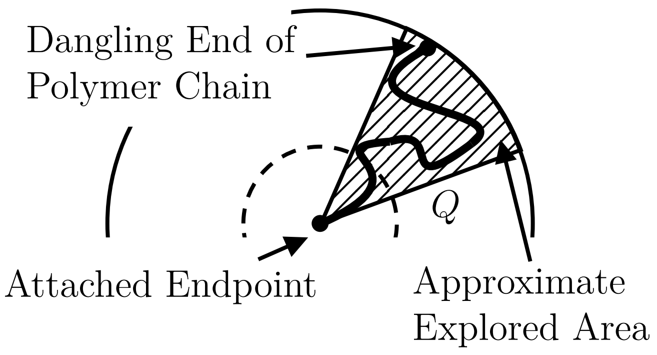

Figure 2.4: Diagram of a polymer chain with one attached end, and one free or dangling end. We model the area explored by the dangling end as the volume of a cone, thus leading to a αQ^2 dependence on the length in the probability of attachment.

Considering only the behavior of a dangling dumbbell transitioning to an active dumbbell we write,

$$P^*_{D\to A} = 1 - \exp\left[-\frac{\alpha Q^2}{1 - Q^2/Q^2_{max}}\Delta t\right]. \tag{2.7}$$

With this construction the probability of association increases with the length of the dumbbell. In addition, it follows the argument in Hernández Cifre et al. that states this probability should relate to the space explored by a sticky end as it retracts through the solvent. However, because the volume explored by the retracting dangling chain should grow with the same proportionality of the volume of a cone with height the length of the chain, αQ^2 is used for the numerator. Indeed, figure 2.4 illustrates this point.

To simulate the species transition from dangling to looped, a novel probability transition function is proposed. Considering the dynamics of micelle network formation, it is reasonable to assume dumbbells whose lengths tend toward zero have a higher likelihood of becoming loops, while longer dumbbells should have a very low probability of looping. This is because chains dangling from a micelle core should be more likely to self-embed forming loops if they are close to their core. Moreover, these characteristics align with features of the dangling-to-looped approach used in Sing et al. The probability transition function for this is,

$$P^*_{D\to L} = 1 - \exp\left[-\frac{\chi Q (Q_{max} - Q)^2}{1 - (Q_{max} - Q)^2 / Q^2_{max}} \Delta t \right]. \tag{2.8}$$

This formulation for the transition from a dangling to looped species closely mirrors the dangling-to-active transition making shorter dumbbells more likely to form loops and stretched dumbbells very unlikely to loop. Exactly mirroring the dangling-to-active transition calls for asymptotic growth as a dumbbell tends towards zero length. This was found to create too many loops in simulation and limited the dynamic behavior of the modelled polymer. To address this, the additional Q term was added. This term smooths the growth of the probability transition function as it tends towards zero and allows the maximum value to be adjusted through the parameter χ. We found that this construction allows for a wide range in the persistent number of loops present in simulations.

To accommodate all the possible outcomes for a dangling dumbbell we combine these two probability functions in the following scheme. Two random variables, X_1 and X_2 are drawn at each time step. Then they are compared to the computed probability values, $P^*_{D\to A}$ and $P^*_{D\to L}$ in the following way:

If $P^*_{D\to A} > X_1 \wedge P^*_{D\to L} < X_2$, then dangling species becomes active,

If $P^*_{D\to A} < X_1 \wedge P^*_{D\to L} > X_2$, then dangling species becomes looped,

else, remain dangling.

Formulating the approach in this way has the benefit of maintaining the underlying physical equations driving the creation of loops and active dumbbells segments.

2.3.3 Looped Dumbbells

Dumbbells in the looped state are given special consideration. In the looped state, the dumbbell does not change length or orientation as interactions with the fluid are considered negligibly small. Therefore, the only dynamic considered is the thermodynamically driven bond breakage that leads to a change in state to a dangling dumbbell ($L \rightarrow D$). Considering this, the transition probability function Eq. 2.6 describing bond detachment, simplifies to the expression,

$$P_{L \rightarrow D} = 1 - \exp\left[-\beta \Delta t\right]. \tag{2.9}$$

The parameter β governs the rate of dissociation of a polymer chain from a micelle core and is the same parameter used in the probability transition function for the transition from active to dangling. This construction matches the approach taken in Sing et al.

2.3.4 Additional Species Transitions Representations

A visual representation is helpful for understanding how the transition probability functions affect each dumbbell. Figure 2.5 provides an illustration of the probability functions for $\alpha = 1.7$, $\beta = 8.7$, $\chi = 1.0$, $Q_{\max} = 33.3334$ and $\Delta t = 5e-3$. In each plot, the x-axis is the spring length and the y-axis represents the probability of the shaded transition occurring. For example, looking at the center plot we see a dangling dumbbell with length 5, has roughly a 20% chance off becoming looped, a 70% chance of staying dangling, and a 10% chance of attaching to become active.

A single left stochastic matrix [18] can be used to represent all the species changes. First consider the following simplified representation of the above transition probability functions. Notice, that every term but the third, depends on the length of the dumbbell Q.

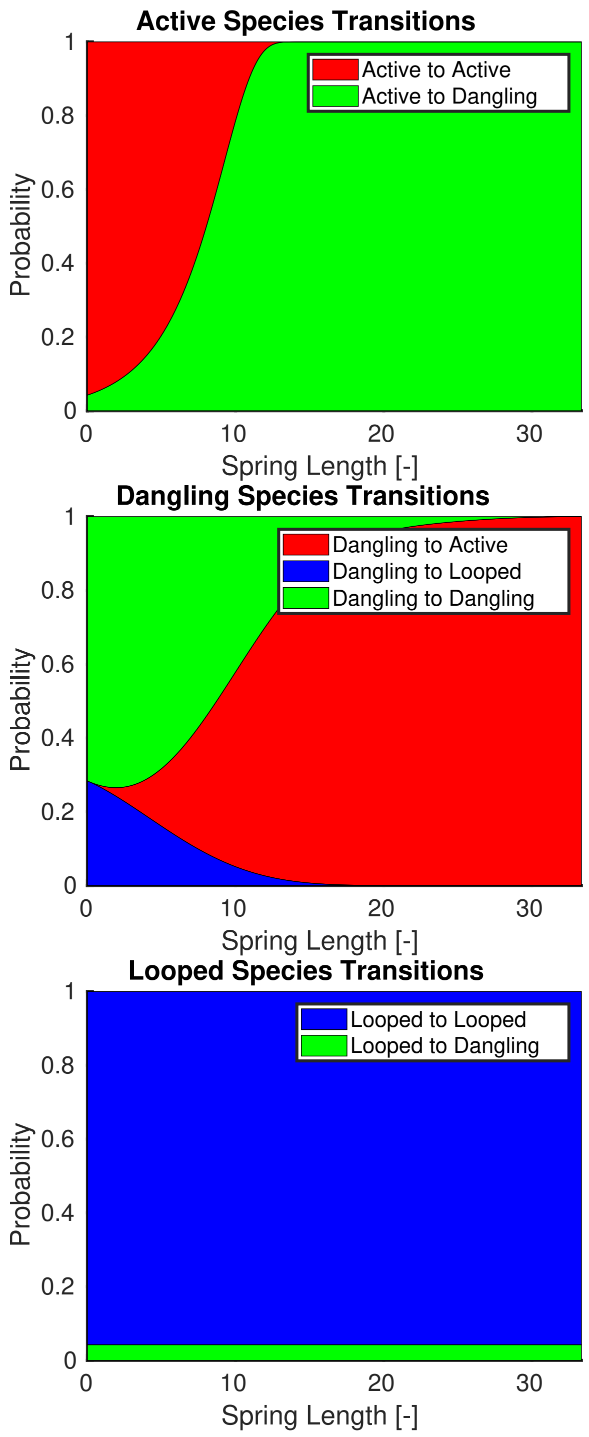

Figure 2.5: Transition probability functions. *Top.* Probability of transitioning from an active to dangling species type. *Center.* Probability of transitioning from a dangling species to either an active or looped type. *Bottom.* Probability of transitioning from a looped to dangling species type. *All.* Parameters used for the plots are $\alpha = 1.7$, $\beta = 8.7$, $\chi = 1.0$, $Q_{\max} = 33.3334$ and $\Delta t = 5e - 3$.

$$\alpha(Q) = \frac{\alpha Q^2}{1 - Q^2/Q_{max}^2} \tag{2.10}$$

$$\beta_A(Q) = \beta \exp\left(0.0325 \left|\frac{Q}{1 - Q^2/Q_{max}^2}\right|\right) \tag{2.11}$$

$$\beta_L(Q) = \beta \tag{2.12}$$

$$\chi(Q) = \frac{\chi Q (Q_{max} - Q)^2}{1 - (Q_{max} - Q)^2/Q_{max}^2} \tag{2.13}$$

Putting these expressions into a 3×3 matrix, we have

	Active	Dangling	Looped
Active	$e^{-\beta_A \Delta t}$	$\left(1 - e^{-\alpha\Delta t}\right) e^{-\chi\Delta t}$	0
Dangling	$1 - e^{-\beta_A \Delta t}$	$\left(1 - e^{-\alpha\Delta t}\right)\left(1 - e^{-\chi\Delta t}\right) + e^{-\chi\Delta t} e^{-\alpha\Delta t}$	$1 - e^{-\beta_L \Delta t}$
Looped	0	$e^{-\alpha\Delta t}\left(1 - e^{-\chi\Delta t}\right)$	$e^{-\beta_L \Delta t}$

The species along the top row indicate the current state, the column down the left side represents a transition to the named species. By the laws of probability, each column should sum to 1. For active and looped dumbbells, this is straight forward. For dangling dumbbells,

$$\begin{aligned}
&\left(1 - e^{-\alpha\Delta t}\right) e^{-\chi\Delta t} + \left(1 - e^{-\alpha\Delta t}\right)\left(1 - e^{-\chi\Delta t}\right) + e^{-\chi\Delta t} e^{-\alpha\Delta t} + e^{-\alpha\Delta t}\left(1 - e^{-\chi\Delta t}\right) \\
&= e^{-\chi\Delta t} - e^{-(\alpha+\chi)\Delta t} + 1 - e^{-\alpha\Delta t} - e^{-\chi\Delta t} + 2e^{-(\alpha+\chi)\Delta t} + e^{-\alpha\Delta t} - e^{-(\alpha+\chi)\Delta t} \\
&= 1.
\end{aligned}$$

2.4 Additional Equations

The drag on an attached segment should differ from a segment with one end dangling freely. The proportionality constant, Z, serves this purpose. In contrast with

Hernández Cifre, we make no connection between state of the overall dumbbell population and this term as they found it had little influence on the viscosity curve. The approach we present here simplifies computational complexity and maximizes the potential for parallel computation. The result is,

$$\zeta_{node} = Z\zeta_{free}. \tag{2.14}$$

The total stress from the polymer chain network on the solvent is determined by Kramer's type expression [6]. Looped dumbbells are not considered to contribute to the fluid stress because they do not carry tension in the looped the state [28]. However, their presence effects the number density of the chains in the solution, and therefore they are included in the count for the total number of dumbbells, N, in the simulation. This is in line with the approach taken by Sing et al. in [44]. The stress contribution σ is nondimensionalized by k_BTn, where n is the number density of polymer chains [15]. In non-dimensional form it is given by,

$$\sigma_{ij} = -\frac{2}{N}\left(\sum_{active} F(Q_i)Q_j + \sum_{dangling} F(Q_i)Q_j\right). \tag{2.15}$$

We follow viscosity, η and the first normal stress coefficient, Ψ_1, as defined in [35]:

$$\eta = \frac{\sigma_{xy}}{\dot{\gamma}} \qquad\qquad \Psi_1 = \frac{\sigma_{xx} - \sigma_{yy}}{\dot{\gamma}^2}.$$

2.4.1 Nondimensionalization

Equations are made nondimensional by the variables:

$$\tilde{t} = t\frac{\zeta_{free}}{4H} \qquad \tilde{Q} = Q\sqrt{\frac{2k_BT}{H}} \qquad \tilde{\sigma} = nk_BT\sigma.$$

Table 2.2: Model Parameters: Description and Values.

Parameter	Description	Simulated Values
α	Alters probability of attachment in the dangling to active species transition.	0.1-1000
β	Alters probability of detachment, from active to dangling, and looped to dangling species.	0.1-100
χ	Alters probability of looping attachment in dangling to looped species transition.	0.01-0.00015
B	Dimensionless characteristic bond length.	0.0325
Z	Balances the difference in drag between active and dangling dumbbells.	30
Q_{max}	Maximum dumbbell length.	33.3334
ζ_{free}	Drag of a free dumbbell.	12
H	Spring Constant.	3

2.5 Model Parameters

Although our model interpretation is straight forward it contains many parameters. Many of these parameters are representative of physical quantities and their values can thus be guided by measurement [23]. In this work, we focus on the default values used in Hernández Cifre [22] for Z, Q_{max}, ζ_{free} and H. The choice of the value of B is discussed above in section 2.3.1. Some parameters, such as those modifying the attachment detachment and looping probabilities, are more abstract in nature and are explored in our results. Descriptions and values for each parameter are listed in table 2.2.

Chapter 3

Simulation Method

3.1 Brief Overview

Each dumbbell's configuration and state are computed in parallel on graphical processing units (GPUs). A semi-implicit first order method evolves the stochastic differential equation 2.1 ensuring that dumbbells do not exceed their maximum length [40]. At each time step, dumbbell configurations evolve according the SDE and species type, probability transition values are computed and species types are altered appropriately. At regular intervals the configuration and type are used to calculate the fluid stress response. All simulations presented in this work contain 1024000 dumbbells. Simulations are run until a steady state is achieved which was determined by a combination of visual and analytical inspection. For simple shear flow, the final value is a mean over the steady state time period. For oscillatory flows, viscous and elastic coefficients are fitted to the steady state period using MATLAB's fit functions. For large amplitude oscillatory flow, the software MITLaos [17] is used to determine the dynamic moduli and higher harmonics present in nonlinear rheological measurements. Based on the Fourier transform spectrum, a technique described in [58, 59, 27], valid harmonics are identified from stochastic noise, and higher harmonics are filtered to improve clarity.

3.2 Coding for the High Performance Computing Environment

The programming scope of this project and fits squarely into the realm of Big Data. Big Data is defined as, “Information assets characterized by such a high volume, velocity and variety to require specific technology and analytical methods for its transformation into value”[13]. In this project, over one million dumbbells are simulated to provide clear results from the stochastic differential equations describing them; fulfilling the volume requirement. Second, in order to do this efficiently, parallel computation is written in CUDA C. This involves developing an understanding of the memory and executable hardware architecture [11]. Moreover, the scale necessitates moving to a high performance computing (HPC) cluster where GPU accelerators and file storage systems can process and store the large amounts of data produced. These are the specific technological requirements. Finally, a second set of MATLAB codes is used to analyze the data and produce useful information. Together these tools form the complex workflow necessary for Big Data. Indeed, the results achieved in this project are not currently possible with routine coding methods and computation platforms.

The simulation code consists of about 3400 lines of CUDA C. The cuRAND library is used to generate random numbers on the GPU [36]. Beyond this however, no special packages or libraries are employed. The main execution code follows a macro-micro loop design, macro for the CPU and micro for the GPU (See Figure 3.1). The main execution loop is as follows: Code on the CPU sends a full set of the dumbbell data to the GPU to be evolved for a set number of time steps; The GPU evolves each dumbbell the set amount of times in parallel and then returns the data back to the CPU; The new configuration is recorded by the CPU and the old is updated; The loop repeats until the desired time is reached. At the end of the simulation only the configurations recorded on the CPU are written to the csv file for output. The upshot of this arrangement is that it avoids sending large amounts of data between the CPU and GPU allowing for efficient computation. However, the downside is that configuration changes on the GPU are not recorded.

In certain configurations the simulation code can quickly produce large amount of data. At 20 bytes of data per dumbbell per time step its easy to see how 102400 dumbbells quickly add-up over the course of simulations which have an average of 10^8 time steps (That's 2048T of data from a single run). The macro-micro loop itself helps to limit the data size, however, it alone is not enough. In order to manage and derive information from the output much of the analysis revolves around metadata, such as the calculated stress, average lengths, average angle variance, etc. These results are stored in the output csv file. When the configuration of every dumbbell is needed, this is done at specific intervals and over time periods of interest, such as the steady state. This type of data was used to create figures such as the dumbbell configuration histograms in figure 4.15 for example. It is also possible to track every change of a single dumbbell as is displayed in figure 4.16. Enabling these features results in a bin file which is produced concurrently with the csv output.

Simulations where run in batches on multi-code nodes with Nvidia Tesla M2090, K80 and P100 GPU Accelerators. Since the memory footprint on the GPU is small, and the CPU core is usually fully utilized multiple simulations were run simultaneously depending on the environment. The table 3.1 lists the CPU-GPU combinations used in this work. Simulations were found to run significantly faster when writing to local storage. For certain file systems, not writing to local storage was enough to slow down a large cluster, and therefore care should be exercised if this is the case. Computation time ranges from mins to days depending on simulation parameters, with low flow rate steady shear flow and high frequency SAOS requiring the most time. Improving code performance beyond usable runtimes was not the primary concern of this work, and thus there are many areas for improvement.

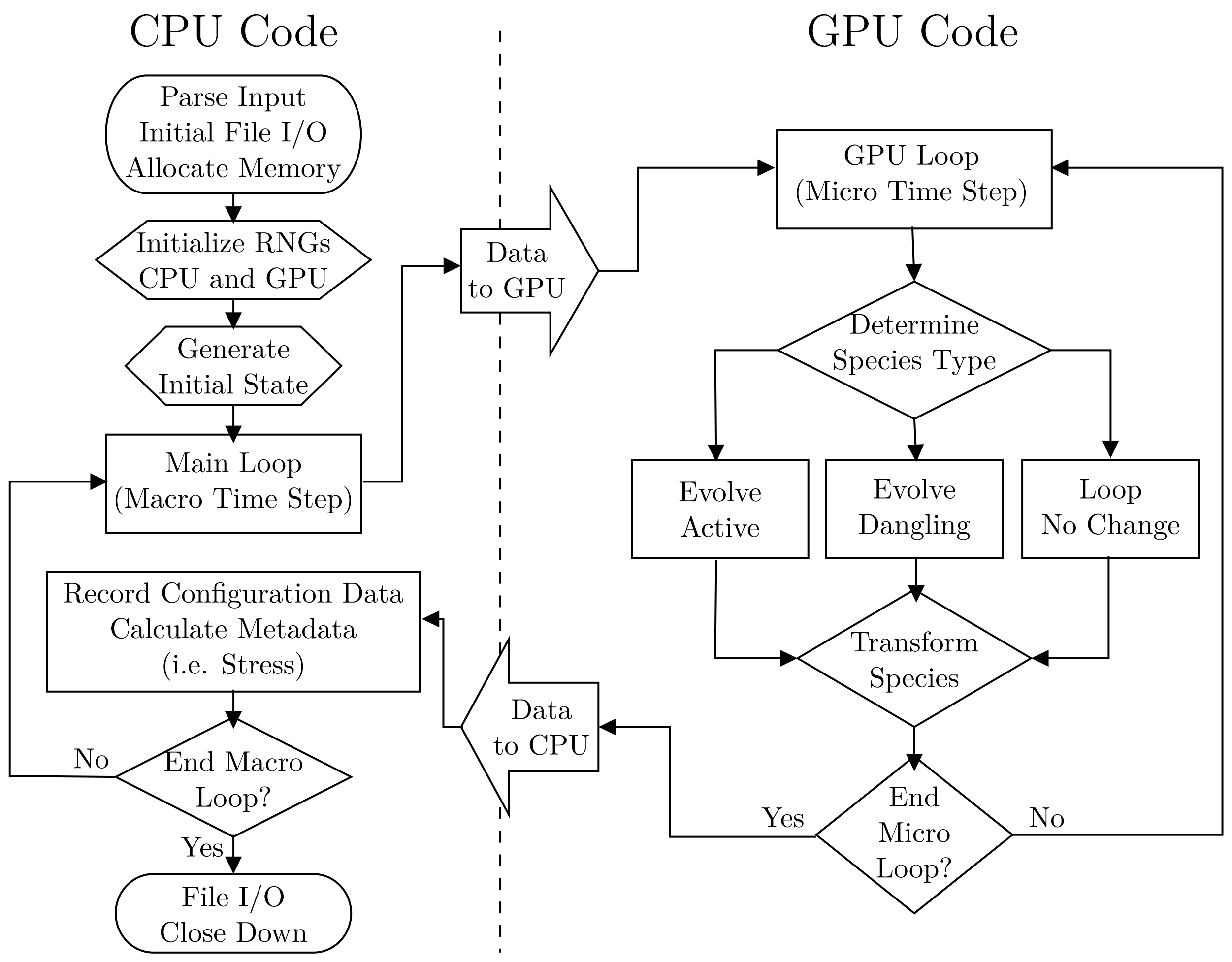

Figure 3.1: Code flow chart. The design of the code can be divided into CPU (left) and GPU (right) parts. Sending data between the CPU and GPU is a time intensive operation. The loop on the GPU represents a single thread and is run independently for each of the 1024000 dumbbells in the simulation. The term 'RNG' refers to random number generator.

Table 3.1: Table of computation environments and run combinations. The code has a small memory footprint on the GPU but fully utilizes the CPU core. Therefore, to saturate the hardware, multiple runs were executed simultaneously.

CPU	GPU	Number of Simultaneous Simulations
Intel Xeon X5660 @2.8GHz	Nvidia Tesla M2090	1
Intel Xeon E5-2620 v3 @2.4GHz	Nvidia Tesla K80	4 (2 per GPU core)
Intel Xeon E5-2680 v4 @2.4Ghz	Nvidia Tesla P100	3

3.3 Strain Simulations

The main body of this work is concerned with modelling two small strain experiments used in rheology; both of which are well established [31, 6, 35]. The first is steady shear flow. In steady shear flow simulations, a shear rate proportional to the vertical displacement imposes a force on each dumbbell. This simulates strain imposed via drag on the polymer network by the fluid flow that results from a sliding top plate moving in a single direction. For each steady shear flow simulation, the system is allowed to equilibrate at zero shear rate flow for 100s of non-dimensional time. Then the shear rate is imposed at the prescribed rate. This protocol was followed for each run although in practice it was not shown to affect the steady state stress response. The measured stress values used in the simulations are the result of an average taken over the final period of the steady state that was verified visually; typically the last 10% of simulated flow time. Figure 3.2 shows the output from a steady shear simulation.

The second simulated type of flow is small amplitude oscillatory shear (SAOS).

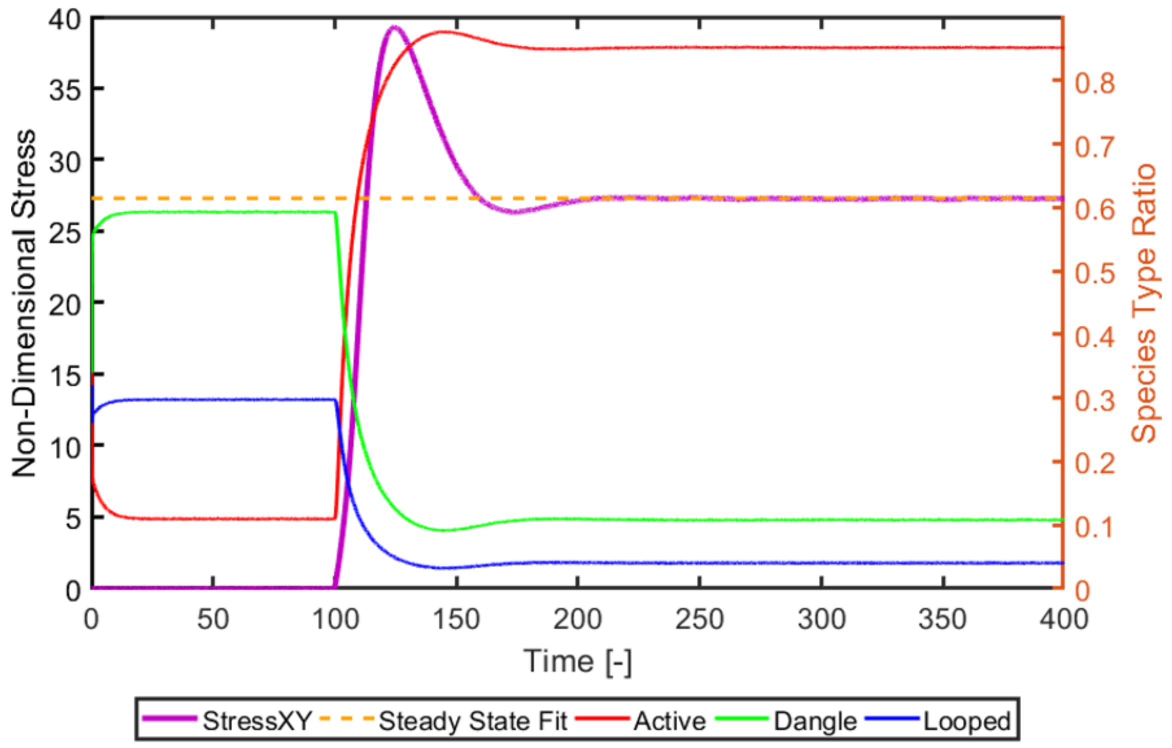

Figure 3.2: Plot of a typical steady shear flow simulation. The left axis shows the stress response, σ_{xy}, versus time. The right axis indicates the fraction of each species type. The system is allowed to equilibrate with zero flow until $t = 100$ then the flow is imposed. The dashed line indicates the steady state stress value measured as an average over the last 10% of simulated flow time.

Small amplitude oscillatory shear differs from steady shear in that imposed strain varies sinusoidally in time. This imposed strain is written as $\gamma = \gamma_0 \sin(\omega t)$, where ω is in radians per second. In these simulations, the viscoelastic moduli are calculated from the stress over a range of oscillation frequencies ω for a fixed strain amplitude, γ_0. The stress response is decomposed into in-phase and out-of-phase components, $G'(\omega)$ and $G''(\omega)$, the storage or elastic modulus and the loss or viscous modulus, respectively. In our simulations, these coefficients are determined by fitting the steady state period (typically the last 25%) to the expression, $G'\gamma_0 \sin(\omega t) + G''\gamma_0 \cos(\omega t)$, using the default fit routine in MATLAB. Figure 3.3 illustrates the performance of the curve fitting. The value of γ_0 is often described vaguely as "small enough" or $\gamma_0 \ll 1$ [31]. Based on Ewoldt [16], we use $\gamma_0 = 0.5$, and verify visually that the ratio

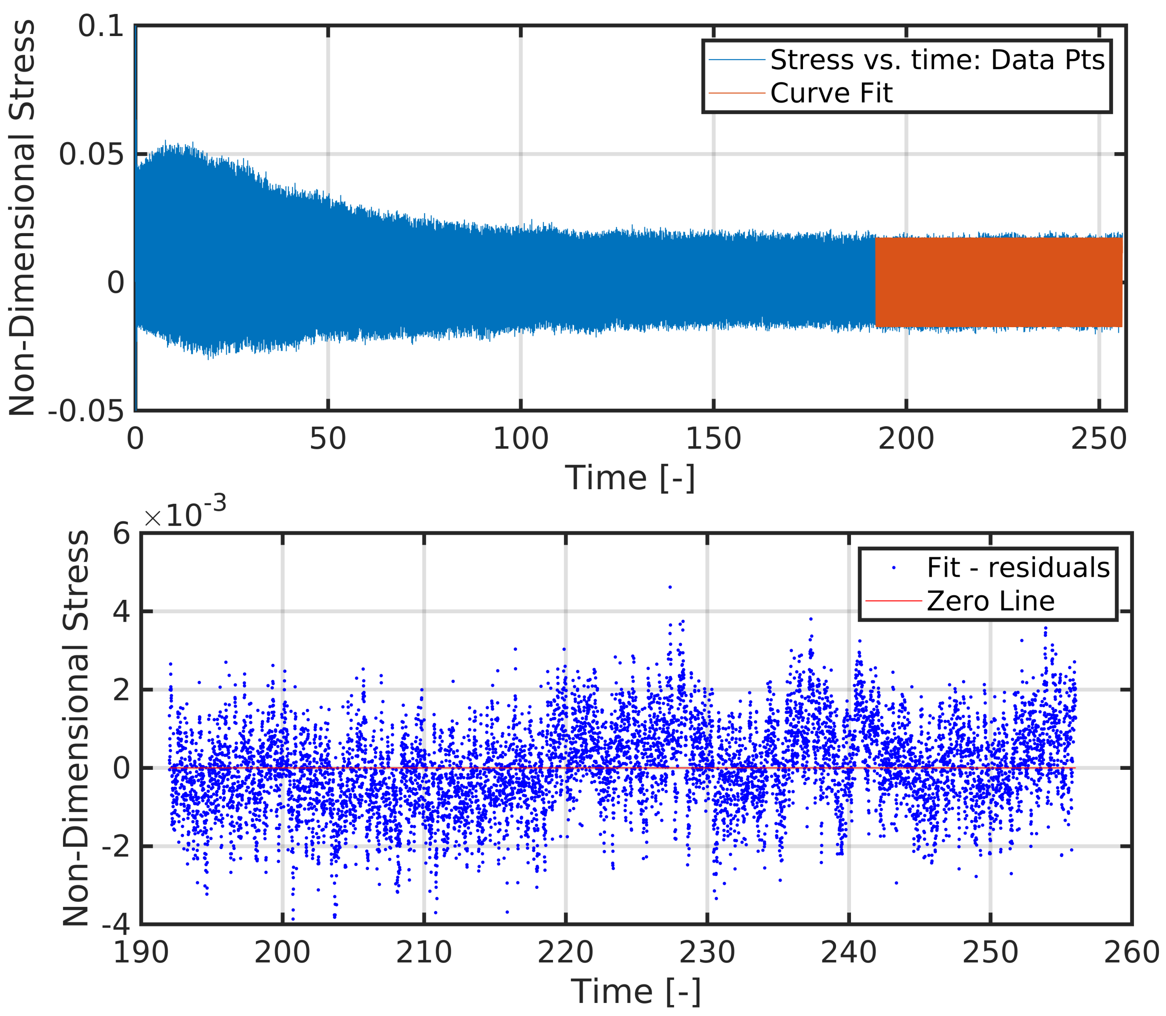

Figure 3.3: Figure illustrating the fitting of the coefficients of $G'\gamma_0 \sin(\omega t) + G''\gamma_0 \cos(\omega t)$ to the stress response of a SAOS simulation. The top plot shows the simulated stress response. The red curve is the fitted expression over the steady state period. The lower plot indicates the error between each data point and the computed curve.

of the third harmonic to the first is much less than one ($e_3/e_1 \ll 1$, in their notation) to ensure simulations are within the linear regime and thus qualify as small strain. This was visually verified for each SAOS simulation output using the plot seen in figure 3.4 that shows the Fourier transform of the stress from the steady state of a SAOS simulation.

Large amplitude oscillatory shear (LAOS) simulations differ from SAOS in only that the strain amplitude γ_0 is increased. For large enough γ_0 the stress signature

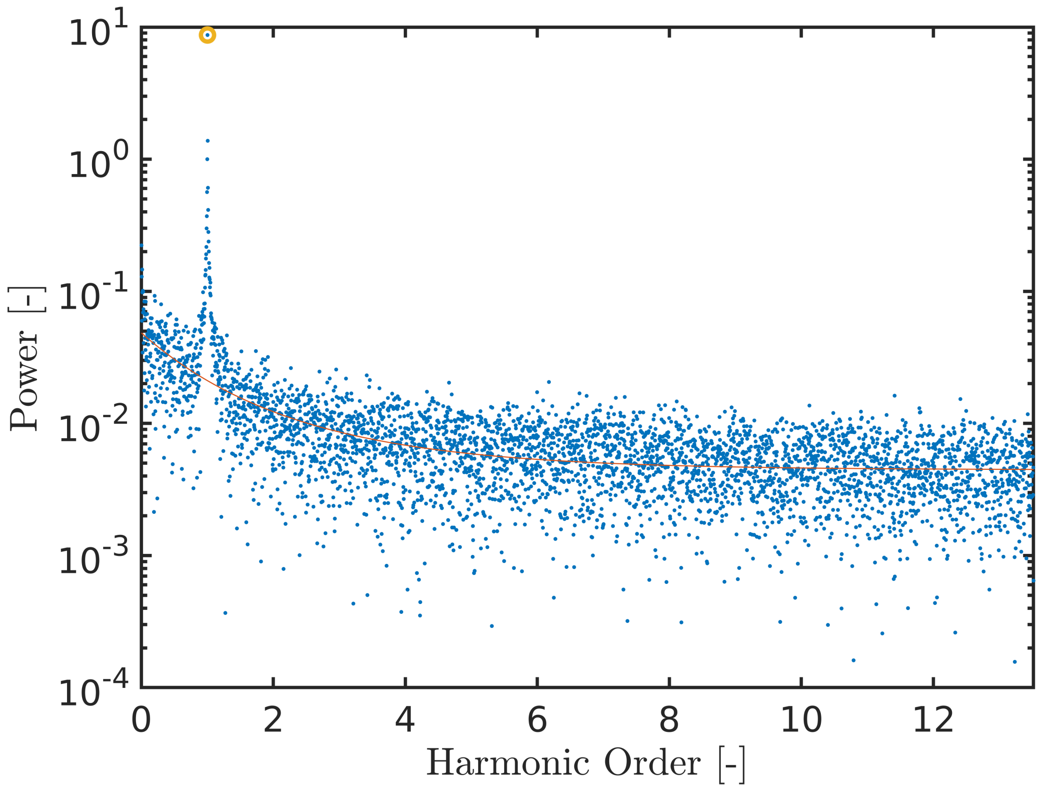

Figure 3.4: Plot showing the Fourier transform of the stress response from a SAOS simulation. The single peak at 1 indicates there is only a clear first harmonic present and thus this response should be considered within the linear regime.

contains multiple harmonics indicating entrance into the nonlinear viscoelastic regime [16]. A plot showing the multiple harmonics can be see in figure 3.5. A short MATLAB routine automatically identifies the largest harmonic and feeds the information to the MITLaos software where higher harmonics are filtered to smooth the output. The yellow circles indicate harmonics identified by the routine. Output from the steady state of LAOS simulations is analyzed via the MITLaos software [17]. MITLaos was used to construct the viscous and elastic Lissajous-Bowditch curves used in the Pipkin diagrams in figures 4.19 and 4.20 in section 4.4.

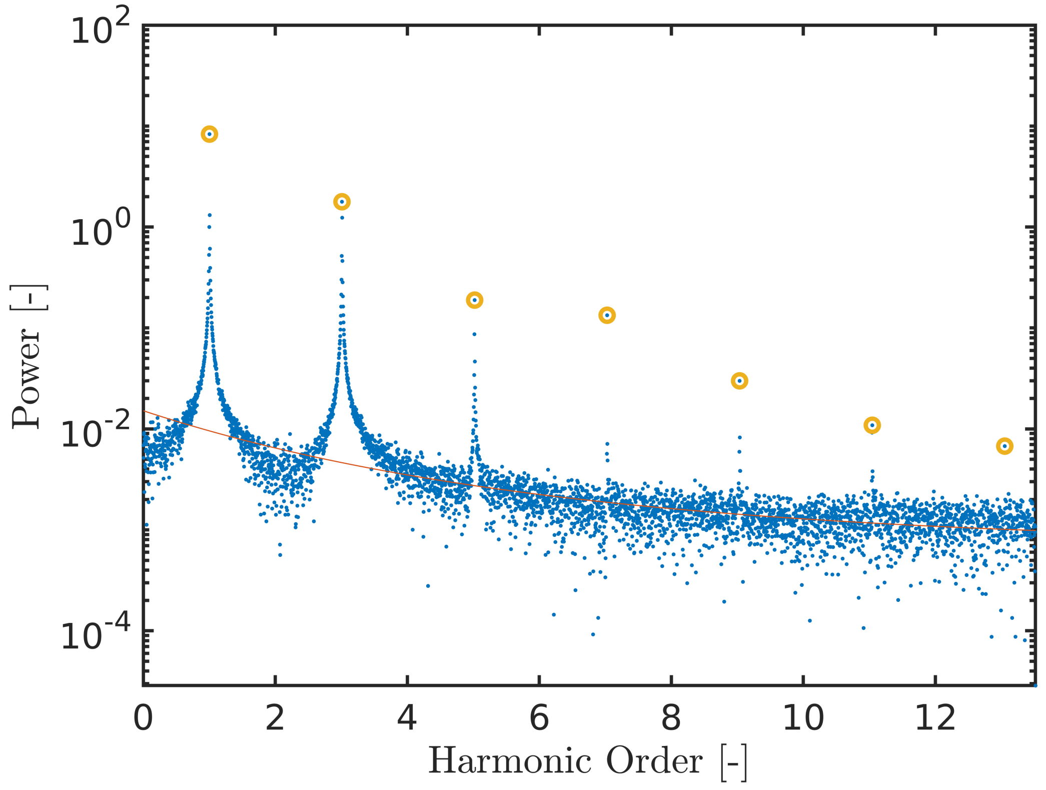

Figure 3.5: Plot showing the Fourier transform of the simulated stress response from a LAOS plot. The multiple peaks indicate additional harmonics present in the stress response. Harmonics automatically identified by the MATLAB routine are identified with a yellow circle.

3.4 Uncertainty Quantification

In stochastic simulation, it is best practice to indicate uncertainty in simulated values and provide an explanation of the "error bar" representation employed [21, 7]. However, the simulated values in the results section of this work do not contain error bars. The reason is that in our statistic simulations, the variation between runs was small enough that including representations of the uncertainty did not add to understanding in the presentation. The purpose of this section is therefore to accurately represent the uncertainty in our simulated values with analysis and figures purposed for the task.

Two example analyses of uncertainty in simulated values are included here; one

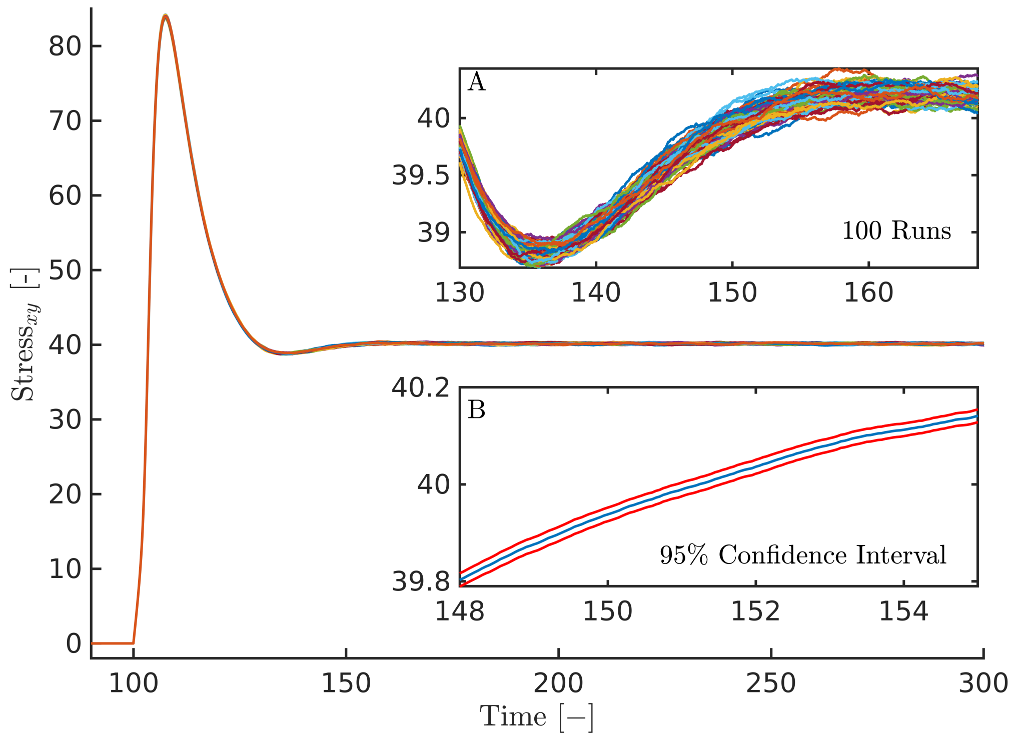

Figure 3.6: Figure showing the stress response from 100 steady shear simulations. The largest plot shows the full runtime of a collection of 100 simulations. Insert A is an enlarged view of the transient period where variations are largest. Insert B is a plot of the 95% confidence interval. These areas where enlarged to highlight the variation between simulations.

for steady shear and one for SAOS flow. Each example includes 100 runs with randomly generated initial states. The initial state consists of x and y lengths of each dumbbell randomly chosen from a normal distribution. Dumbbells assigned lengths longer than the maximum were reinitialized. Time was used to seed random number generation and efforts were made to avoid running multiple simulations at the same time. Confidence intervals (95%) were computed via established methods [21] and are illustrated in figures 3.6 and 3.7. It is important to note that while transient behavior is examined in these figures, only steady state measures were used elsewhere in this work. The uncertainty of these measures as it pertains to these two examples is described in table 3.2.

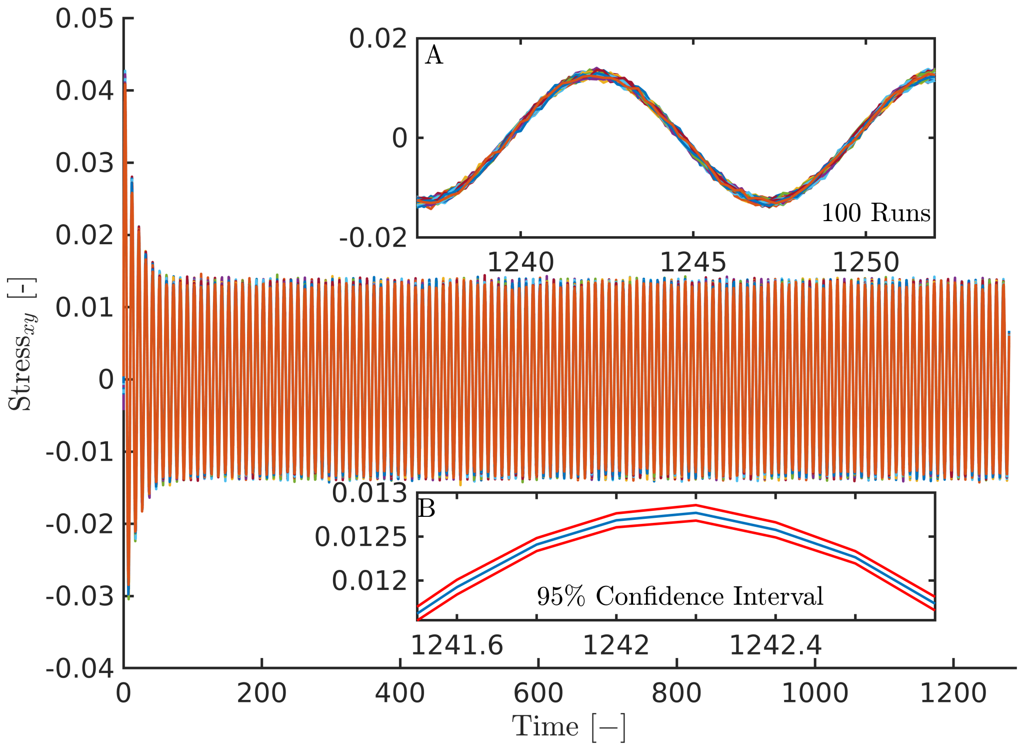

Figure 3.7: Figure showing the stress response from 100 SAOS simulations. The largest plot shows the full runtime of a collection of 100 simulations. Insert A is an enlarged view of a selected time period. Insert B is a plot of the 95% confidence interval. These areas where enlarged to highlight the variation between simulations.

By examining the figures and data above we can draw a few conclusions about the significance of simulated values from the model. Transient behavior was included in these examples because it is expected to show larger variations since the modelled system is out of equilibrium. However, both figures show that every simulation captured important features such as ringing in steady shear, decreasing stress amplitude in SAOS, and similar start times to the steady state period. Moreover, the quantitative difference in these features was small, indicating that there is very good agreement between multiple runs of the same simulation parameters. For the more robust steady state measures used in the results section, such as stress in the xy direction (σ_{xy}) and G' and G'', variation between runs was even more diminished. The small size of the confidence intervals indicates that computed values are close to

Table 3.2: Uncertainty in the steady state values from examples seen in figures 3.6 and 3.7.

Simulation	Computed Quantity	Mean	95% Confidence Interval	Standard Deviation	Figure
Steady Shear	σ_{xy}	40.1427	± 0.0062	0.0313	3.6
SAOS	G'	0.0250	$\pm 8.5167 \times 10^{-}6$	$4.2922 \times 10^{-}5$	3.7
SAOS	G''	0.0053	$\pm 7.7415 \times 10^{-}6$	$3.9015 \times 10^{-}5$	3.7

what would be expected in large systems of polymer chains. The standard deviation gives an estimate of how close we can expect the computed value from a single run to be to the mean of multiple runs. Indeed, the standard deviations for these values are small enough that we can be confident in the qualitative conclusions drawn from single runs in this work.

Chapter 4

Results

This chapters presents results obtained using the model. This includes two and three species simulations for steady shear flows. A comparison of SAOS output with two and three species. In addition, two methods for determining the configuration of a dangling dumbbell after being looped are compared. Finally, results from LAOS simulations are showcased.

4.1 Simple Shear

Simple shear experiments introduce a deformation flow parallel to the bottom of the fluid cell at a steady rate. This type of flow is also referred to as steady shear, sliding plate or Couette shearing flow. The velocity gradient is prescribed as,

$$\nabla v = \begin{bmatrix} 0 & 0 \\ \dot{\gamma}_0 & 0 \end{bmatrix} \tag{4.1}$$

In the presentation herein, we set $\kappa = (\nabla v)^T$. For each experiment the system is given time to equilibrate before flow starts. Then viscosity and the first normal stress coefficient are measured once a steady state is achieved.

Complex fluids can be categorized by their fluid response. In pure viscous fluids, the stress response decreases with increasing flow rate. This phenomena is called shear thinning. In an elastic material, the stress response increases with increasing shear deformation. Fluids that increases their stress response with increasing shear rate are called shear-thickening. The focus of this work is simulating the fluid response of a viscoelastic fluid, where a combination of shear thinning and shear thickening can be present.

Another method of categorizing complex fluids is to separate Newtonian and non-Newtonian. Two important characteristics of the latter fluid are; the fluid's viscosity does no increase proportionally with the rate of shear strain, and a positive normal stress coefficient. The non-Newtonian category incorporates viscoelastic fluids that exhibit shear thinning or shear thickening, and thus could be used to describe the scope of fluids simulated by this model as well.

In simple shear experiments, parameters describing individual dumbbell properties adopt the default values listed in table 2.2. Parameters the alter the species transition behavior (α, β, χ) were varied, and viscosity was examined.

4.1.1 Two Species Simple Shear

In a two species simulation, containing only active and dangling dumbbells, we find that shear thickening or shear thinning behavior can be altered by modifying either or both attachment or detachment mechanics. Consider the plot for $\alpha = 10$ and $\beta = 10$ in the center of Figure 4.1. By increasing the probability of attachment so that $\alpha = 100$ and $\beta = 10$, we can see from the run in the left plot, that the amount of shear thickening decreases and the amount of shear thinning increases. Conversely, if we increase the detachment parameter so that $\alpha = 10$ and $\beta = 100$, we find in the plot on the right that the amount of shear thickening increases. These two parameters can be combined into a ratio which determines the fluid response as show in 4.1.

Our simulations indicate that shear thickening and shear thinning behavior is caused by the interplay of attached and dangling network segments. These are shown in the species fraction plots in 4.2. At low flow rates when number attachments in the network is high relative to the number of dangling, less shear thickening occurs as the flow rate increases. This is because the length of active dumbbells increases

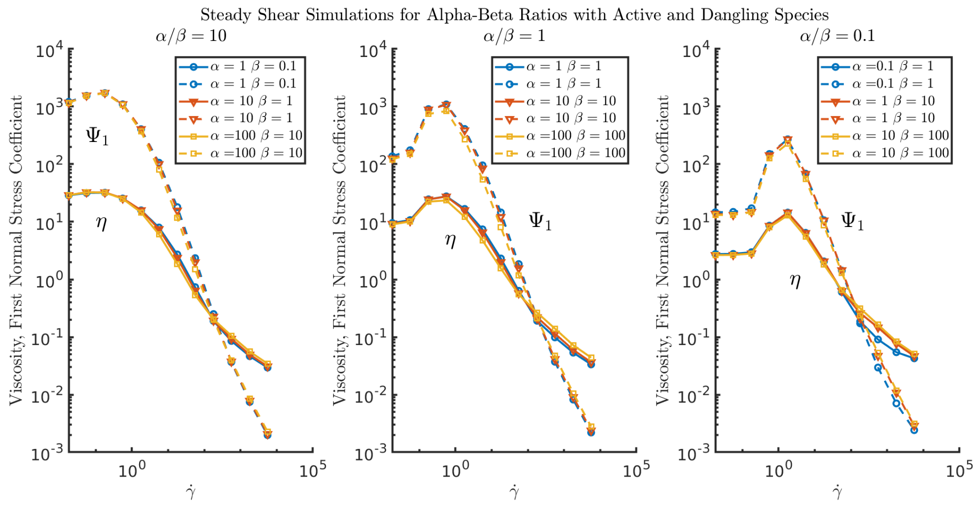

Figure 4.1: Viscosity η and First Normal Stress Coefficient Ψ_1 for steady shear flow simulations of a two species model. Each plot contains a single α/β ratio.

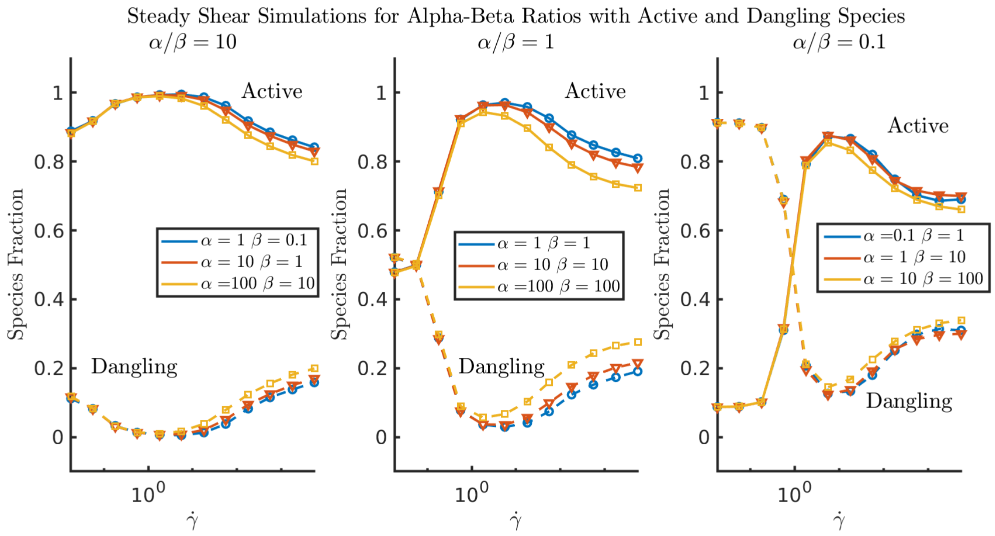

Figure 4.2: Species fractions in shear flow. Plots show the steady state species fraction of each type, active and dangling, for the simple shear simulations in 4.1.

proportionately with the flow rate. Therefore, when a large number of segments are already attached, their length grows and few species changes occur. The period of shear thinning that follows, is the result of length growing and dumbbells detaching to a dangling state where they are more likely to reduce length. This behavior is the result of having a detachment probability that depends on the length of the dumbbell.

Shear thickening is achieved in the model by our choice of attachment probability.

For simulations where the ratio of alpha to beta is near 1, there are a number of dangling segments available to attach at low flow rates. As flow rates increase, they lengthen and as a result of the form of the dangling-to-attached probability function, increasing numbers of dangling dumbbells change species to active and remain in an extended state. When the flow rate increases to the point where active dumbbells extend and break, transitioning back to dangling dumbbells, shear thinning appears.

A short mathematical analysis provides some insight into the α/β ratio control over shear thickening and shear thinning. To more easily make sense of the complex dynamics, let $F(Q)$ be an arbitrary function of the spring length and consider the simplified representation of the transition probability functions:

$$P_{Dangling \to Active} = 1 - \exp\left(-\alpha F\left(Q\right) \Delta t\right) \tag{4.2}$$

$$P_{Active \to Dangling} = 1 - \exp\left(-\beta F\left(Q\right) \Delta t\right) \tag{4.3}$$

For each length of spring Q the function of $F\left(Q\right)$ has a fixed value. After choosing values for α, β and Δt the value of the transition probability functions are fixed as well. In this way, the probability of being in one state or the other is set. Now, because the dumbbells transition from active to dangling and dangling to active, altering either α or β, shifts this preference. Therefore, the ratio $\frac{\alpha}{\beta}$ describes the relative difference in preference at any dumbbell length. Considering all the dumbbells in the simulation we see the ratios influences the species fractions of active and dangling species which thus influences the amount of shear thinning and shear thickening seen in the simulations.

Traditional network theory has placed a lot of focus on the attachment and detachment probabilities [51]. Many approaches are based on the Leonard-Jones electric potential in bond forming and breaking. Indeed, in Hernández Cifre, the association energy affects both the attachment and detachment of dumbbells. In our approach, we posit that the probability of dumbbell attachment has more to do with the physical space explored by an unattached end, then the electro-chemical potential of the bond to be formed. For detachment, we reason that overcoming the energy barrier to break the bond is a greater factor. In combination, we show that these two mechanics are each individually able to influence shear-thinning and shear-thickening behavior in steady shear flow. By delinking these processes, we give our model the ability to better simulate tunable polymer experiments that modify bond attachment and detachment independently, such as in several recent works [26, 10].

4.1.2 Three Species Simple Shear

In three species simple shear simulations we incorporate a third dumbbell species —loops. This looped species is assumed to have negligible interaction with the fluid flow and does not contribute to the stress. Including the third species, however, modifies the amount of shear thickening and shear thinning and thus breaks the α/β ratio symmetry seen in the two species simulations.

In three species simulations we find the behavior of the viscosity and the first normal stress coefficient is controlled by the two ratios, α/β and χ/β. In figure 4.3, these correspond with changes in the vertical direction and changes in the horizontal direction respectively. We see that increasing the looping ratio, χ/β, leads to increased shear thickening when there are a moderate number of active dumbbells. The number of active dumbbells in the simulation increases with a larger attachment ratio, α/β, and leads to less shear thickening and more shear thinning. At low flow rates, a higher fraction of active dumbbells increases the fluid stress response. As flow rates increase, the amount of shear thickening depends on how many loops are in the system. When flow rates increase over a rate of $\dot{\gamma} > 1.8$ all simulations show shear thinning.

By examining figure 4.3 we can identify several trends in the model. Starting

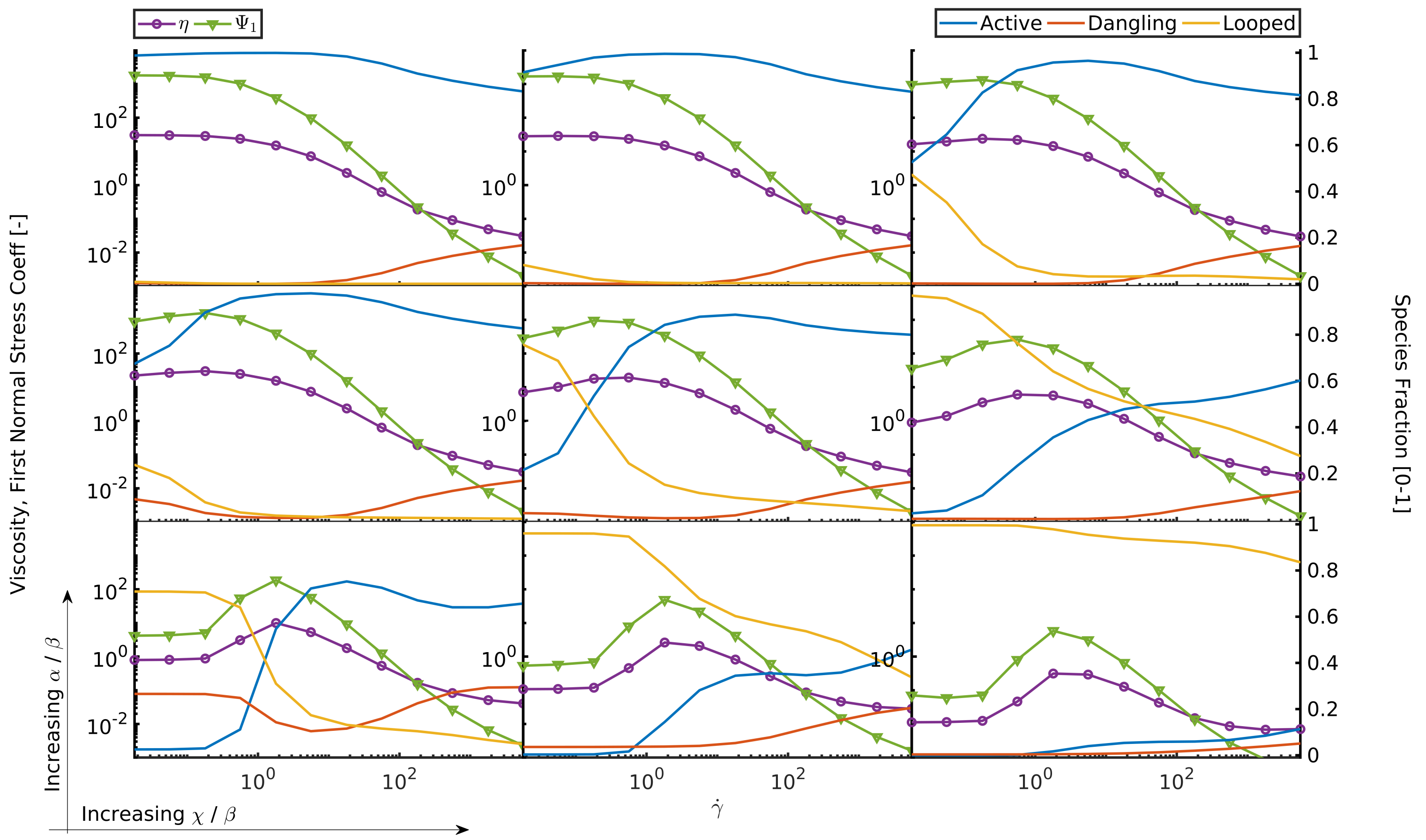

Figure 4.3: Three species steady shear flow simulations. Plot shows steady state viscosity η and first normal coefficient Ψ_1, organized by attachment, α/β, and looping, β/χ, ratios. The α/β ratio increases in the upward direction and the β/χ ratio increases left to right.

with the plot in the top left, there are a large amount of bridged segments and fewer dangling and looped. We could say that this is a polymer network with many interconnections and few loops. When flow rates increase the model shows shear-thinning with flow increase. For the plot in lower right, we have a large number of loops occurring in the network. The model shows a much reduced stress response, but still has shear thickening followed by shear thinning. Indeed, we see that as the flow rate increases, there are more dangling and active chains. This would correspond to the loops breaking out of the micelle formation and dangling or forming connections. Then because the rate of attachment is low, the amount of stress from dangling and active dumbbells extending with the fluid does not increase linearly.

On the other two extremes, we have the example in the top right. This shows a large number of attachments and loops in the network, but few dangling dumbbells that would form connections. In this case, any loops are quickly transitioning to active dumbbells as the flow increases. The result is a shear-thinning response. In the lower left, we see that the looped dumbbells enhanced the shear-thickening response. Without loops, the increase in flow rate and shift between active and dangling dumbbells is enough to generate shear thickening. When loops are added, they lower the fluid stress at low flow rates even further. Then like dangling dumbbells, the number of loops drops due to fluid extension. In contrast to dangling dumbbells, loops however do not again increase as chains extend and break in high rate flow.

Three Species Dissymmetry

In the two species model presented in 4.1.1, the fluid response was controlled by the ratio α/β. This is also true for χ/β when only looping and dangling species are present, as is shown in figure 4.4. However, when these two are combined into one three species model the symmetries are broken. The diagram in figure 4.5 provides a visual guide to the dynamics that lead to this phenomenon. Each circle in the diagram

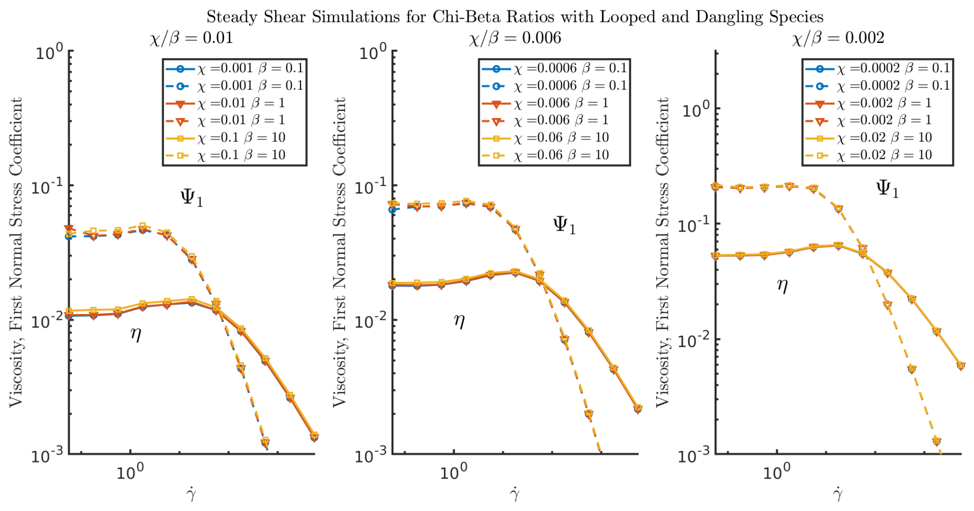

Figure 4.4: Two species, dangling and looped, steady shear flow simulations. Plots show the viscosity η and first normal stress coefficient Ψ_1. Each plot shows the result of three data sets with the same χ/β ratio.

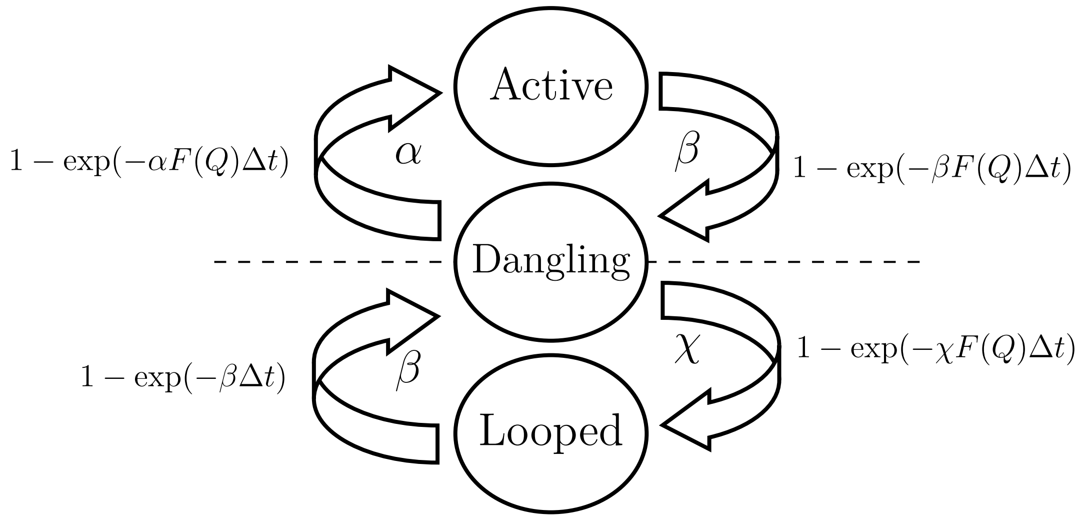
Active
Dangling
Looped
α
β
β
χ
$1 - \exp(-\alpha F(Q)\Delta t)$
$1 - \exp(-\beta F(Q)\Delta t)$
$1 - \exp(-\beta \Delta t)$
$1 - \exp(-\chi F(Q)\Delta t)$

Figure 4.5: Dumbbell species transition diagram. Equations are simplified in that $F(Q)$ represents different functions of the length of a dumbbell, Q. Each ellipse represents a species type. Each arrow represents a species transition and is labeled with the parameter that affects the transition probability. In the two species model, only the area above the dashed line is considered. In the three-species model, the entire diagram is considered.

represents a species type, dumbbell transitions are indicated by arrows between them, the equations on the side are simplified representations of the transition probabilities functions and the parameter is the one associated with each transition.

In steady state, we found species fractions remain constant. Therefore, there are a balanced number of dumbbells entering and exiting each species type. In the two species model, this balance exists between the active and dangling dumbbells and thus between the parameters α and β. This is also true for χ and β. When the two two species models are combined into a three model, because species fractions are constant in the steady state, there must be a balance between all three types. Since the dumbbells are now distributed among three types, and dangling dumbbells transition to active or loops, the original balances are upset.

These effects can be clearly seen by comparing the two species model to the behavior of the active and dangling dumbbells in the three species model as is done in figure refDissymmetry. Each plot in the figure has the same alpha-beta ratio. The top row does not contain loops, and therefore the flow curves show very similar behavior. Introducing loops with $\chi = 0.001$ immediately breaks the symmetry across the second row. This is because the beta-chi ratios are different.

By construction, only dangling dumbbells form loops in the simulation. This behavior means that the inclusion of loops has different effects on the fluid response depending on the fraction of dangling dumbbells in the simulation. Therefore, when flow rates are low dumbbells are less extended and we find an abundance of dangling dumbbells. Because there are more dangling dumbbells at shorter lengths, there are more dumbbells that will become looped. The presence of a higher fraction of loops in the simulation leads to lower overall fluid stresses. As the flow rate increases, dumbbells extend and fewer dangling dumbbells become looped. The overall effect is that shear thickening is more pronounced when loops are included.

In general, we find that by adjusting the likelihood of bridging in relation to the likelihood of disassociating via the α/β ratio, it is possible to control the fluid response. In addition, the ratio between the likelihood of looping and breaking out of the looped state enhances the nonlinear response. In this sense, in a polymer where loops are largely present, and endgroups are reluctant to disassociate we should see

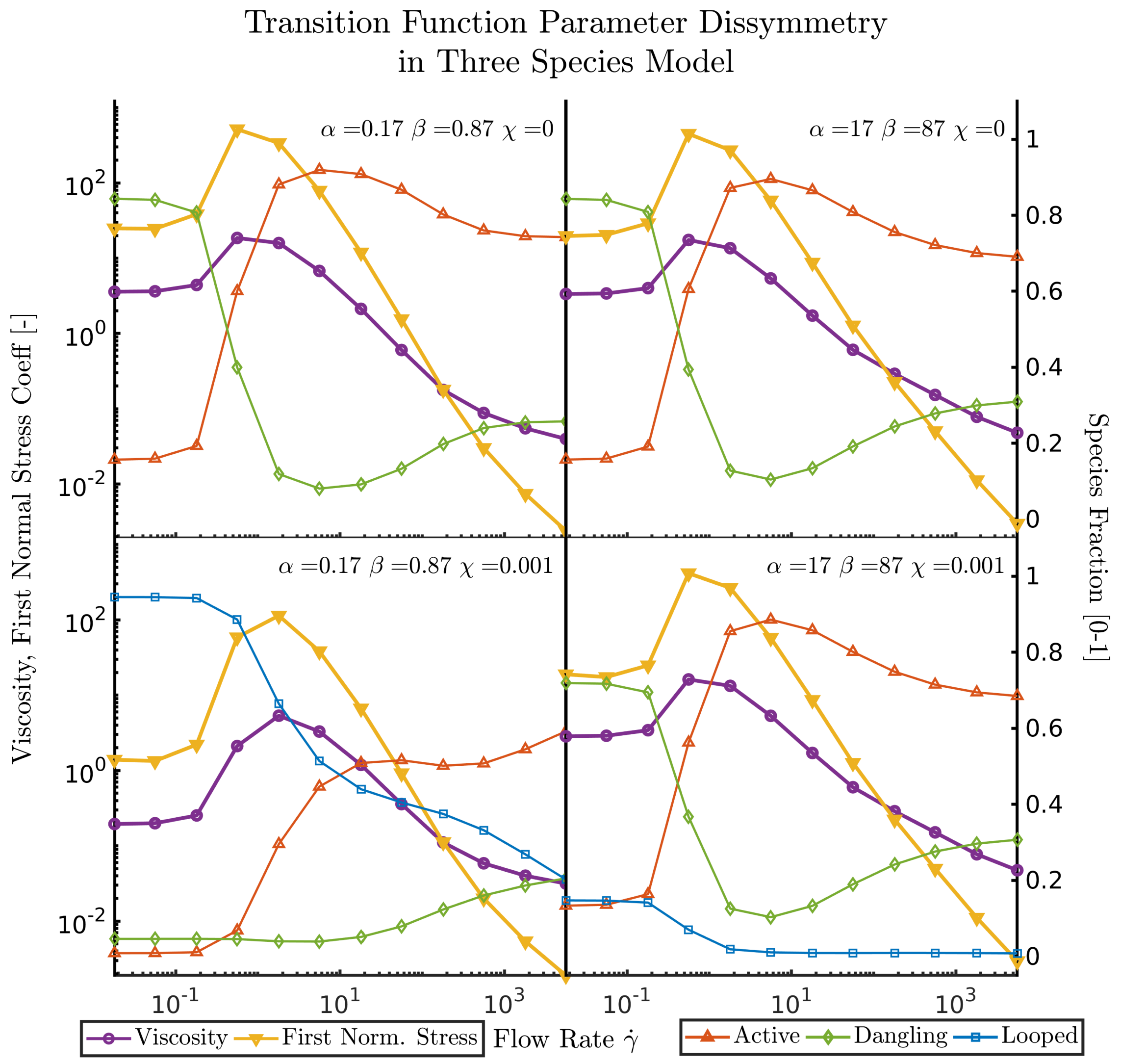

Figure 4.6: Transition function parameter dissymmetry in the three species model. Steady shear plots of viscous and first normal stress coefficient on the left axis. On the right axis, are species fractions. Plots on the first row include only active and dangling species. Plots on the second row also include loops.

greater shear thickening and shear thinning. On the other hand, in a polymer network consisting of mostly bridged networks and less loops, we expect to see higher stresses at low flow rates, followed by less shear thickening before giving way to shear thinning.

Our results compliment the conclusions Sing et al. [44] achieved with their reaction-diffusion Smoluchowski approach. In their work they conclude that the inclusion of loops enhances non-monotonic fluid responses. In examining results across our two and three species models, we too see that it is possible to generate non-monotonic behavior with only two species. However, adding loops enhances the non-monotonicities while lowering overall stress. These effects are most visible when the probability of attachment is not high.

4.2 Small Amplitude Oscillatory Shear

Small amplitude oscillatory shear (SAOS) measures the stress response to an oscillating shear flow to separate out-of-phase viscous and in-phase elastic forces. The shear flow is kept small in order to measure the properties the material without large disruptions to the structure [31]. In this type of simulation, the direction of shear flow is constantly changing at increasingly rapid rates, therefore the stress response has a larger dependence on the dumbbell orientation. In this section we compare SAOS simulations for two and three species models. Figure 4.7 illustrates the characteristics of each simulation across several metrics. In addition, figure 4.8 represents species fractions in terms of only the species that contribute to the fluid response, thereby providing insight into the effects that including the third looping species has on the behavior of the other two.

At low frequency oscillations, dumbbell extension in active and dangling types is similar among the two simulations implying that loops do not on average affect dumbbell length at low frequencies. Instead, the looping dynamic prevents a portion of the dumbbells from contributing to the overall stress calculation. This results in

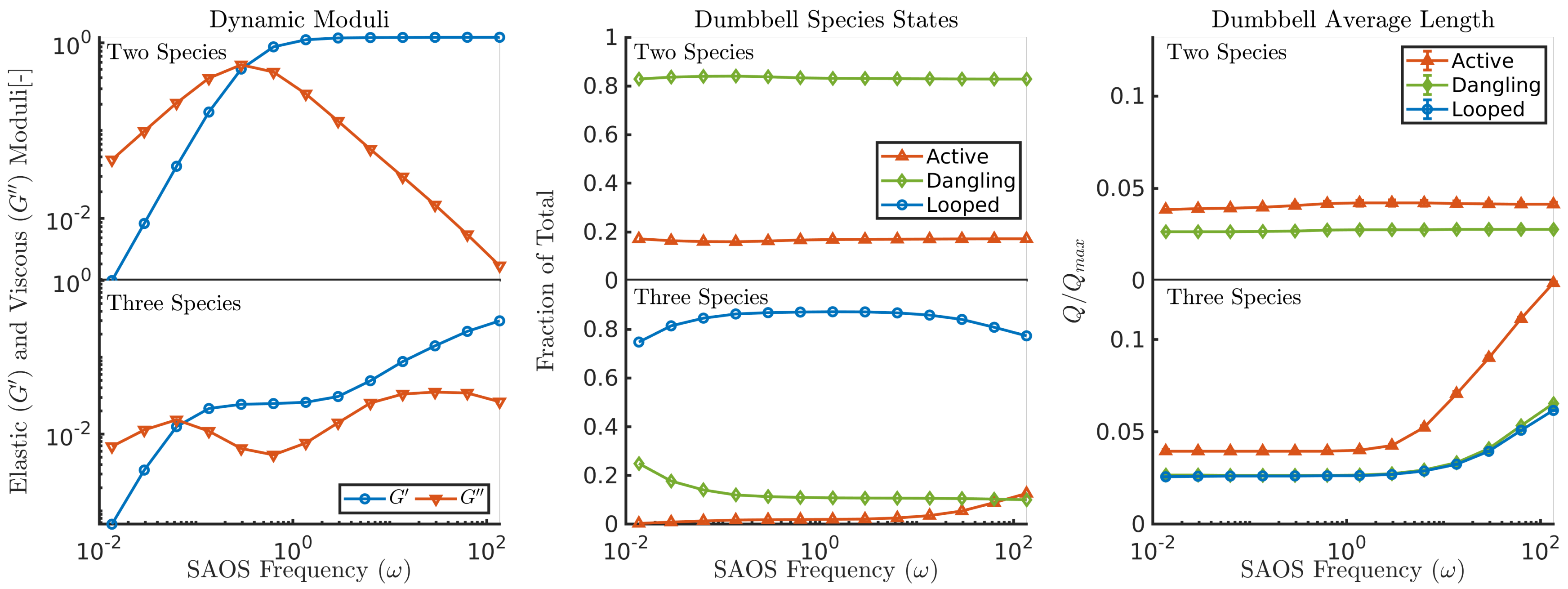

Figure 4.7: Three measurements from a two species and three species SAOS simulation. (Left) Dynamic moduli plots indicate the strength of the network response, and describe elastic-like and viscous-like behavior of the fluid. (Center) Species fraction separated by state. (Right) Average normed length of dumbbell segments.

smaller values for both dynamic moduli in the three-species model. The effect is more dramatic at lower frequencies because the dangling-to-looped transition probability is significantly higher than the dangling-to-active transition probability at shorter lengths. Thus, the correspondence with a drop in the fraction of active dumbbells among species contributing to the stress. In terms of physical behavior this implies that micelle looping stores the potential for greater increases in stress contributions, both viscous and elastic, for when the fluid is under a higher strain rate.

At middle frequencies, we begin to see the effect of incorporating loops on the shape of the dynamic moduli diminish. The reason is that as dumbbells increase in length the difference in transition probability between dangling-to-active and dangling-to-looping shrinks. By inspecting the ratios of species contributing to stress we see that it closely follows that of the two species simulation. Therefore, we can conclude that dumbbells going into the looping state are not demonstrating a length preference beyond what is seen in the two-species case. If they were, their inclusion would disproportionately affect the ratios of one species more than another. Instead, their inclusion effects the other species evenly. This is what leads to the similarities seen in the dynamic moduli characteristics –flat elastic curve and declining viscosity– at lower overall stress levels. In physical terms, this stage of the simulation represents a range of strain rates where loops are not playing a large role in the fluid response. Instead, it is the attachments and detachments of dangling chains in the network that are driving the fluid response.

At higher frequencies, dumbbell extension and orientation become the major factors in stress generation. In the three species model, active dumbbells are captured in an extended and aligned configuration greatly increasing the amount of stress they generate. In addition, the larger flow gradient and subsequent length cause the dangling dumbbells to transition to active more quickly. Meanwhile, dumbbells out of alignment with the fluid flow have shorter length and therefore follow similar loop-

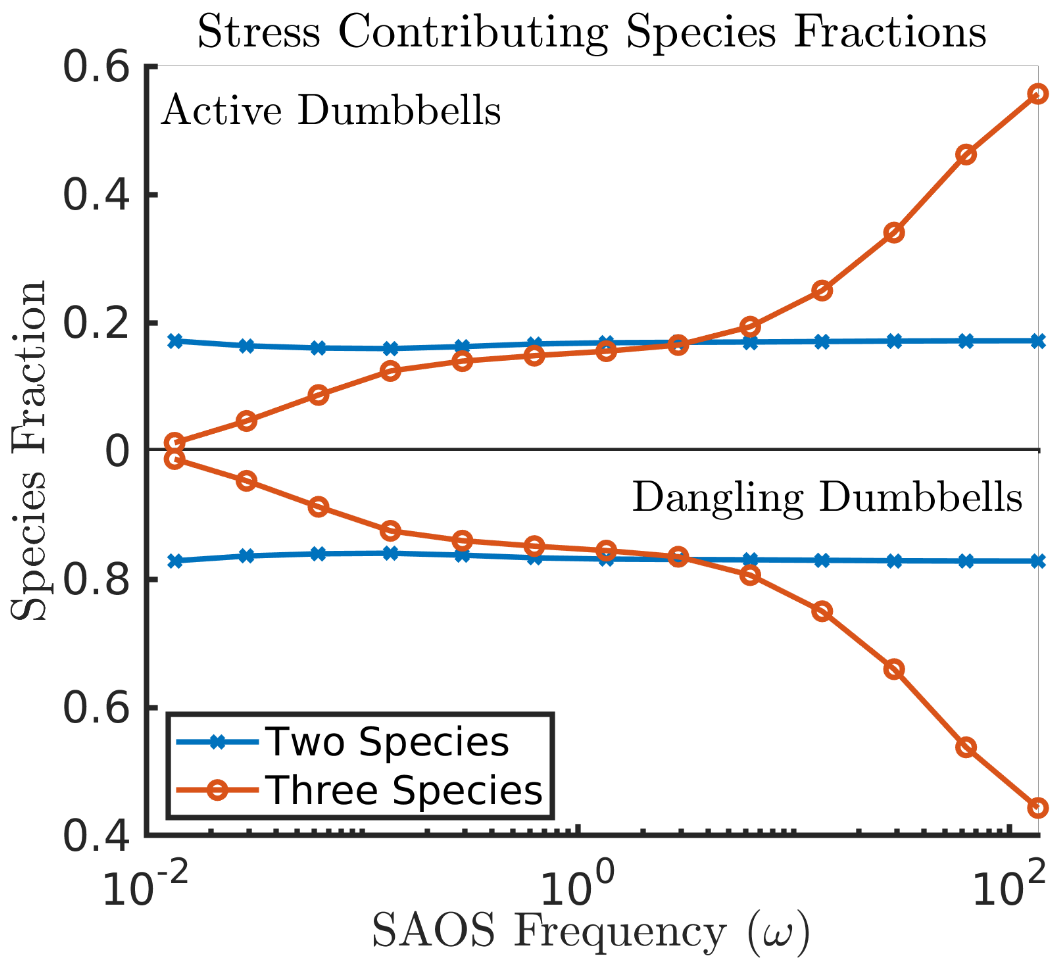

Figure 4.8: Stress contributing species fractions. Plots show the ratio of active (top) and dangling (bottom) dumbbells to active and dangling combined. Three distinct regions in the three species model are apparent; A low frequency region where including loops decreases the ratio of active dumbbells and increase the ratio of dangling. A mid-frequency region, where both models show similar ratios. A high frequency region, where the three species models shows increasing numbers of active dumbbells and decreasing numbers of dangling dumbbells.

dangling transition dynamics. These three factors combine to create a stronger fluid response that causes both moduli to turn upwards, a quality that compares favorably with experimental data the telechelic associative polymer, hydrophobically modified ethoxylated urethane (HEUR) [46, 52, 47].

Figure 4.9, compares model output to experimental data for HEUR measured by Suzuki et al. [46]. The experimental data in the figure is the result of multiple measurements superimposed to create a single master curve using time-temperature superposition [14]. The curves for each temperature show a plateau in the elastic modulus (G') after the relaxation time –where the curves for the moduli cross. In addition, after the relaxation point the loss modulus (G'') decreases before curling up. At this point no more data is provided. The model output matches both these features well with a plateau after the relaxation time in the elastic modulus and a similar drop and curl in the loss modulus. The filled symbols in the figure indicate this area of similarity. The unfilled symbols are included to show the behavior of the model at higher frequencies. The beginning of the unfilled section contains an upward curve at the end of the G' plateau that matches previously reported data for HEUR from Uneyama et al. [52].

Figures 4.10 through 4.15 show the intracycle behavior of the two and three species models. Each plot shows normalized intracycle stress and strain and species ratios on the top left. Numbers on the curve indicate the correspondence with the dumbbell configuration histograms on the right. The SAOS plot bottom left indicates the frequency from which the data is taken. On the right, dumbbell configuration histograms represent the amount of dumbbells at a specific length and orientation if one end is fixed at the origin. Dumbbell positions are not tracked so dumbbells oriented in the quadrants I and III, have the same contribution to the stress as dumbbells oriented in quadrants II and IV. Therefore, the configuration histograms are represented as a half circle.

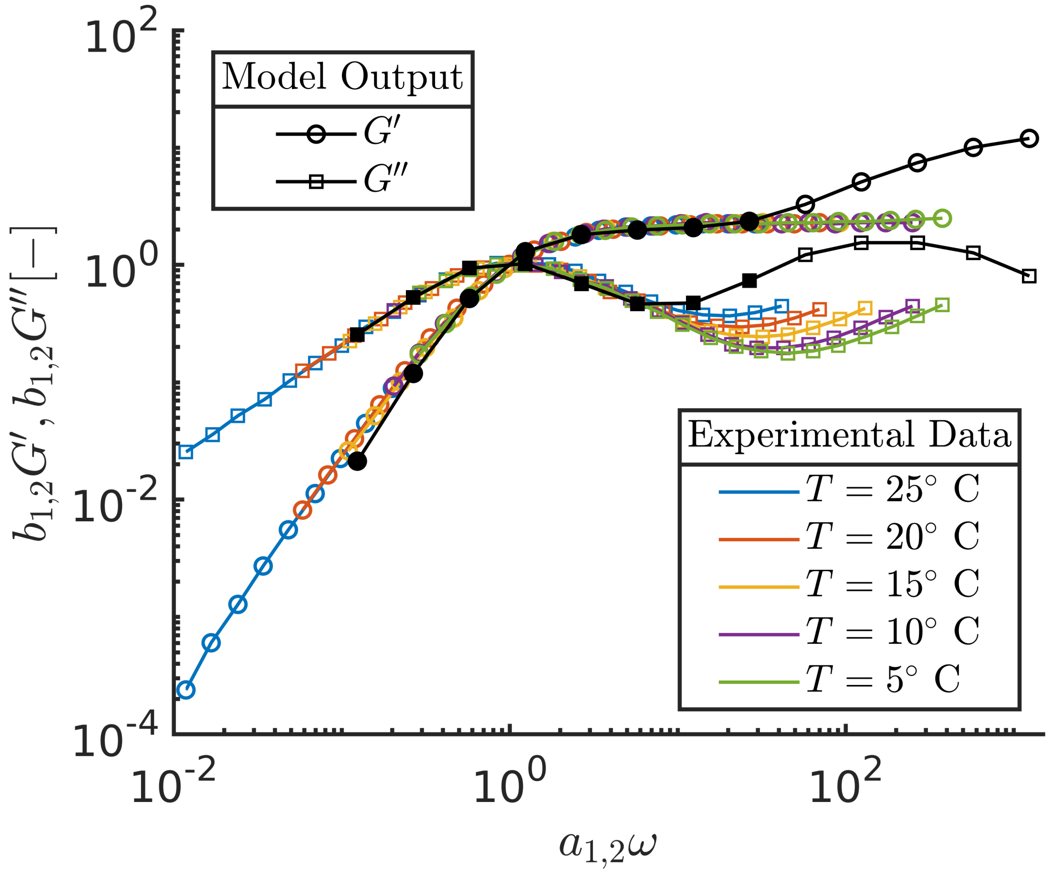

Figure 4.9: Figure showing the dynamic moduli from the model and experimental data for hydrophobically modified ethoxylated urethane. Experimental data and model values have been shifted by constant multiples to align relaxation points. Model parameters were set to $\alpha = 1.7$ $\beta = 8.7$ $\chi = 0.002$ for this simulation. Filled symbols indicate the frequency range with the best match to the data. Unfilled symbols indicate values without experimental data for qualitative comparison.

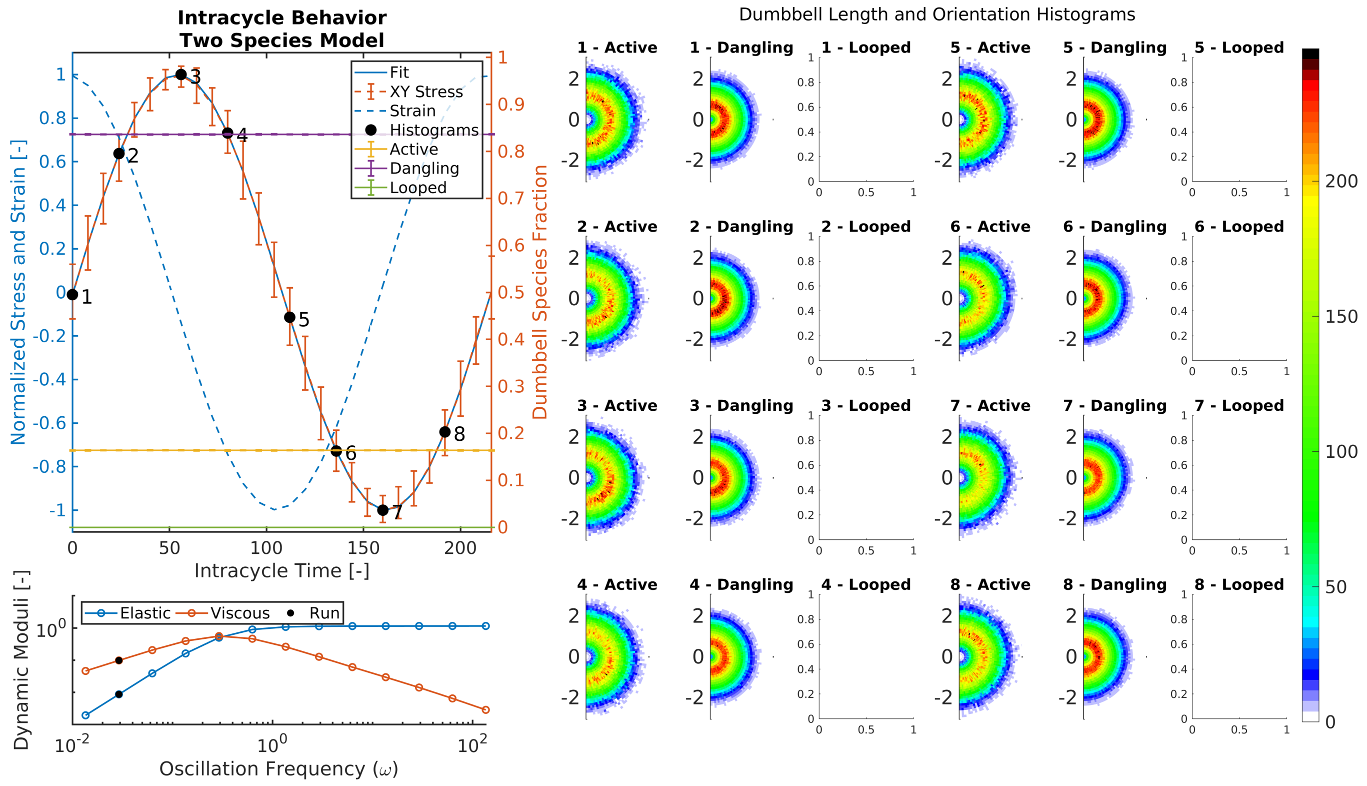

Figure 4.10: Intracycle analysis of a low frequency two species SAOS simulation. *Upper Left* Normalized σ_{xy} stress, strain, curve fit and species fractions over a single cycle. Error bars indicate the range of data over the steady state. Fit refers to the curve generated by fitting σ_{xy} to the function $A\cos\omega t + B\sin\omega t$. The right axis shows the species fractions over the cycle. Error bars indicate variation over the steady state. The numbered dots indicate the corresponding time in the cycle where the corresponding dumbbell configuration histogram is taken. *Right* Dumbbell configuration histograms. To generate the histograms, one end of the dumbbell is fixed at the origin. The placement of the other end is used to for the histogram. Colors represent the number of dumbbell ends in the bin. *Bottom Left* Dynamic moduli for the batch of runs from which the current data is taken. Dots indicate the frequency currently under examination.

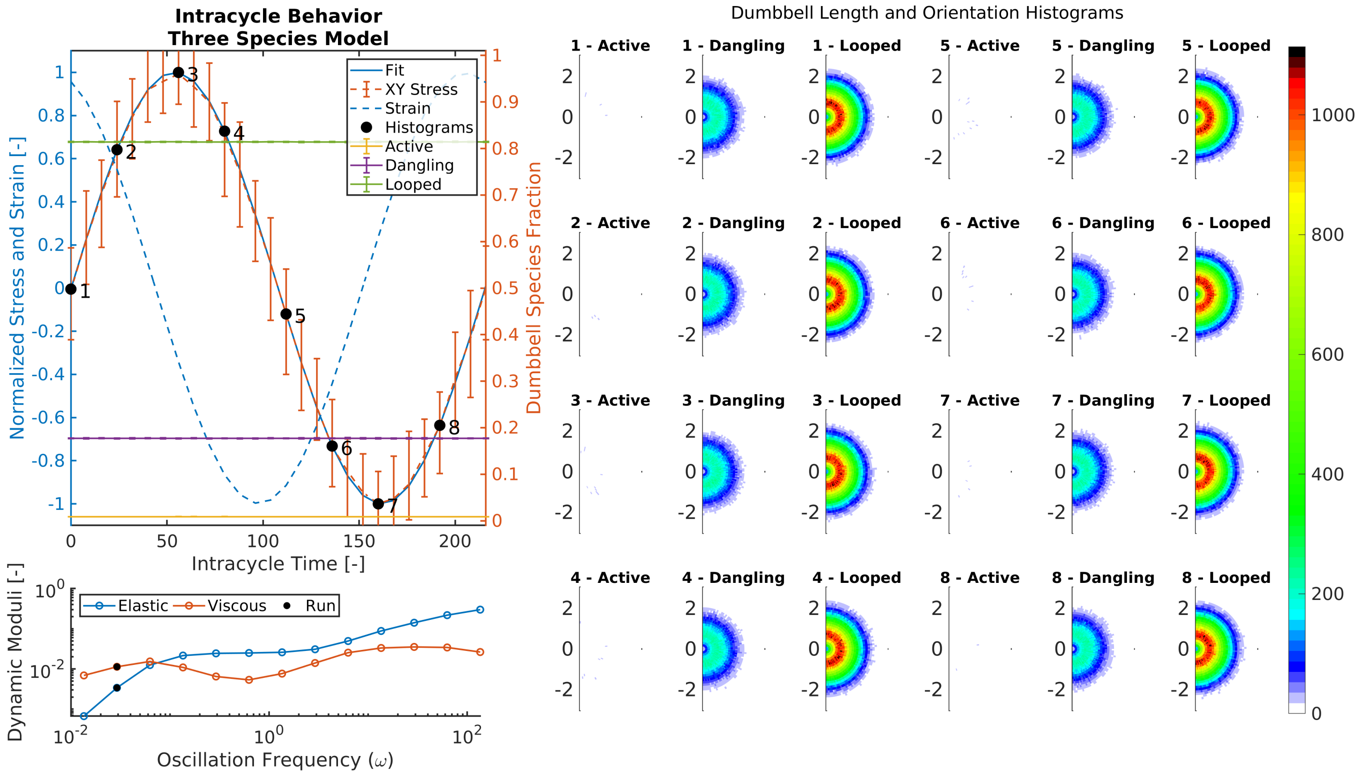

Figure 4.11: Intracycle analysis of a low frequency three species SAOS simulation. *Upper Left* Normalized σ_{xy} stress, strain, curve fit and species fractions over a single cycle. Error bars indicate the range of data over the steady state. Fit refers to the curve generated by fitting σ_{xy} to the function $A\cos\omega t + B\sin\omega t$. The right axis shows the species fractions over the cycle. Error bars indicate variation over the steady state. The numbered dots indicate the corresponding time in the cycle where the corresponding dumbbell configuration histogram is taken. *Right* Dumbbell configuration histograms. To generate the histograms, one end of the dumbbell is fixed at the origin. The placement of the other end is used to for the histogram. Colors represent the number of dumbbell ends in the bin. *Bottom Left* Dynamic moduli for the batch of runs from which the current data is taken. Dots indicate the frequency currently under examination.

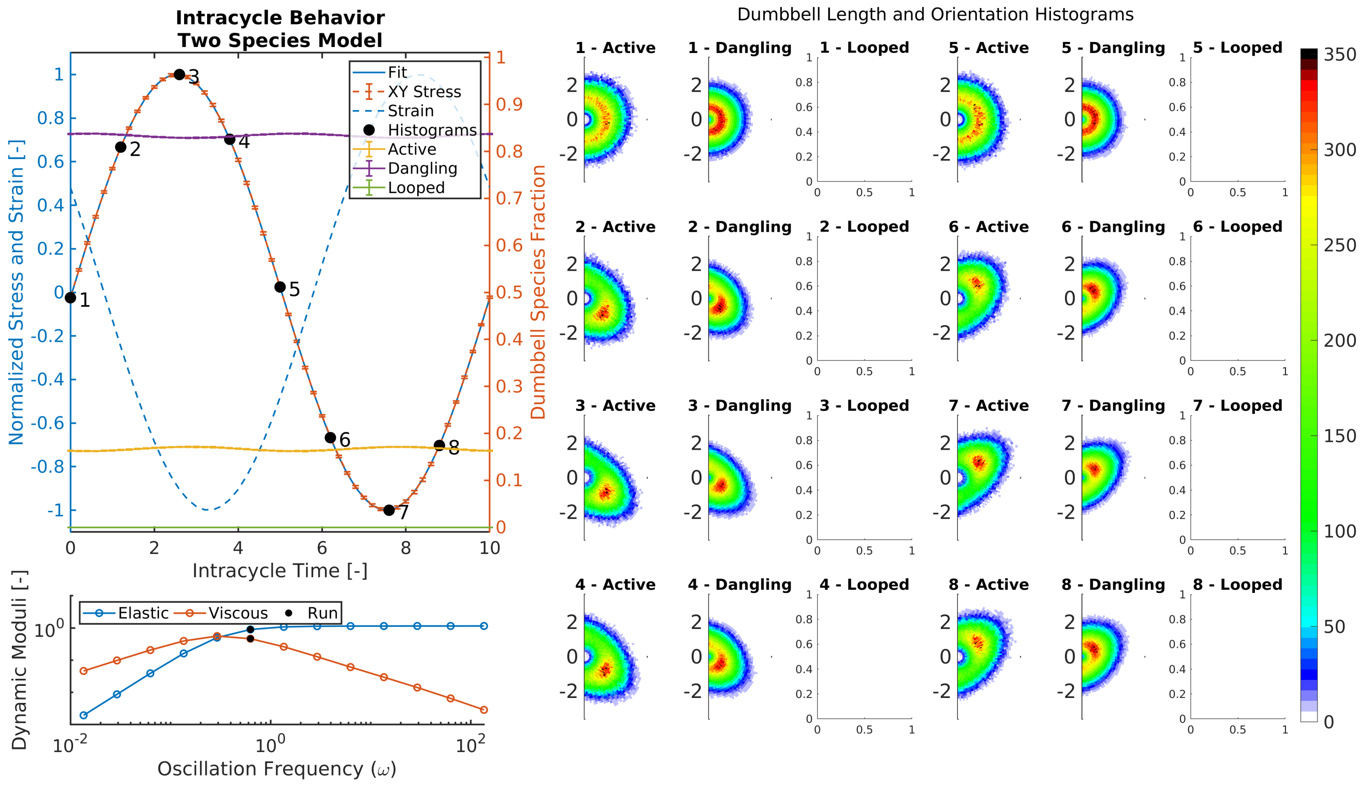

Figure 4.12: Intracycle analysis of a mid-frequency two species SAOS simulation. *Upper Left* Normalized σ_{xy} stress, strain, curve fit and species fractions over a single cycle. Error bars indicate the range of data over the steady state. Fit refers to the curve generated by fitting σ_{xy} to the function $A\cos\omega t + B\sin\omega t$. The right axis shows the species fractions over the cycle. Error bars indicate variation over the steady state. The numbered dots indicate the corresponding time in the cycle where the corresponding dumbbell configuration histogram is taken. *Right* Dumbbell configuration histograms. To generate the histograms, one end of the dumbbell is fixed at the origin. The placement of the other end is used to for the histogram. Colors represent the number of dumbbell ends in the bin. *Bottom Left* Dynamic moduli for the batch of runs from which the current data is taken. Dots indicate the frequency currently under examination.

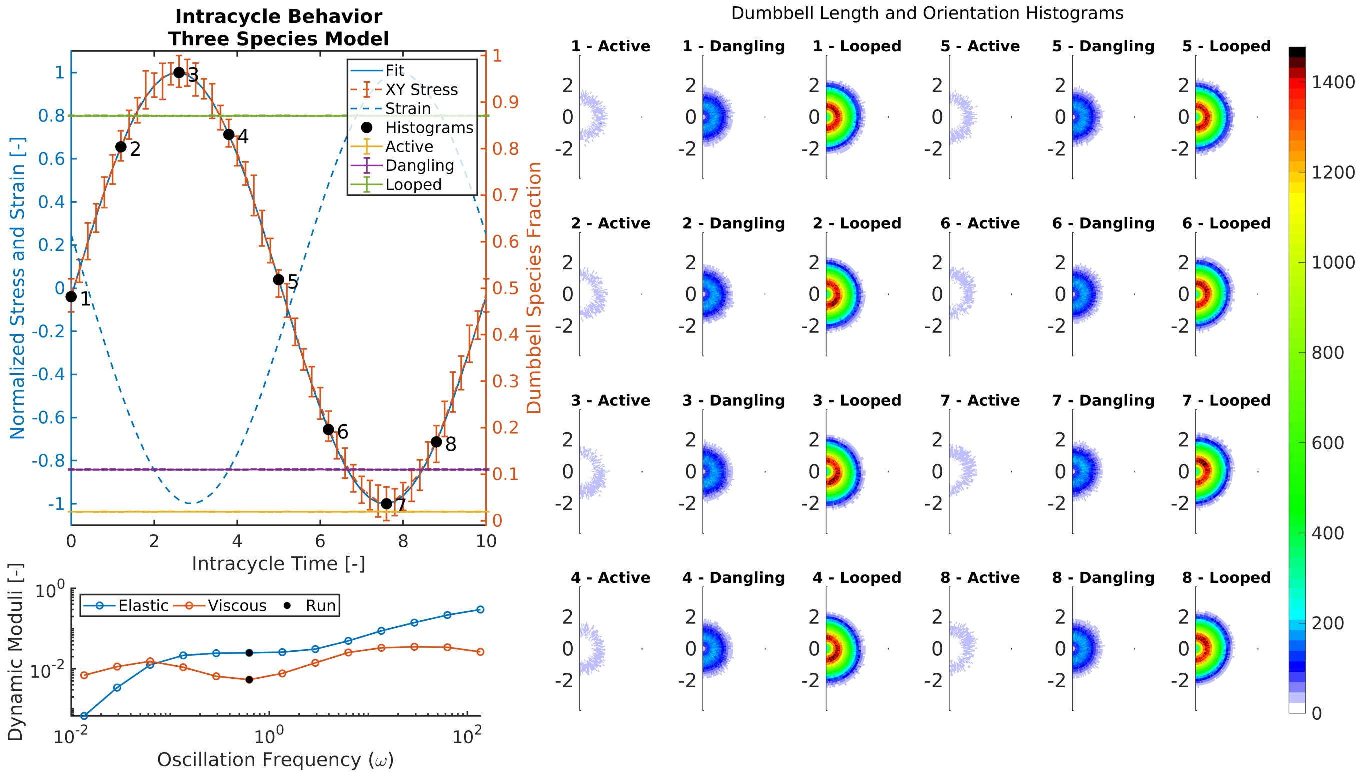

Figure 4.13: Intracycle analysis of a mid-frequency three species SAOS simulation. *Upper Left* Normalized σ_{xy} stress, strain, curve fit and species fractions over a single cycle. Error bars indicate the range of data over the steady state. Fit refers to the curve generated by fitting σ_{xy} to the function $A\cos\omega t + B\sin\omega t$. The right axis shows the species fractions over the cycle. Error bars indicate variation over the steady state. The numbered dots indicate the corresponding time in the cycle where the corresponding dumbbell configuration histogram is taken. *Right* Dumbbell configuration histograms. To generate the histograms, one end of the dumbbell is fixed at the origin. The placement of the other end is used to for the histogram. Colors represent the number of dumbbell ends in the bin. *Bottom Left* Dynamic moduli for the batch of runs from which the current data is taken. Dots indicate the frequency currently under examination.

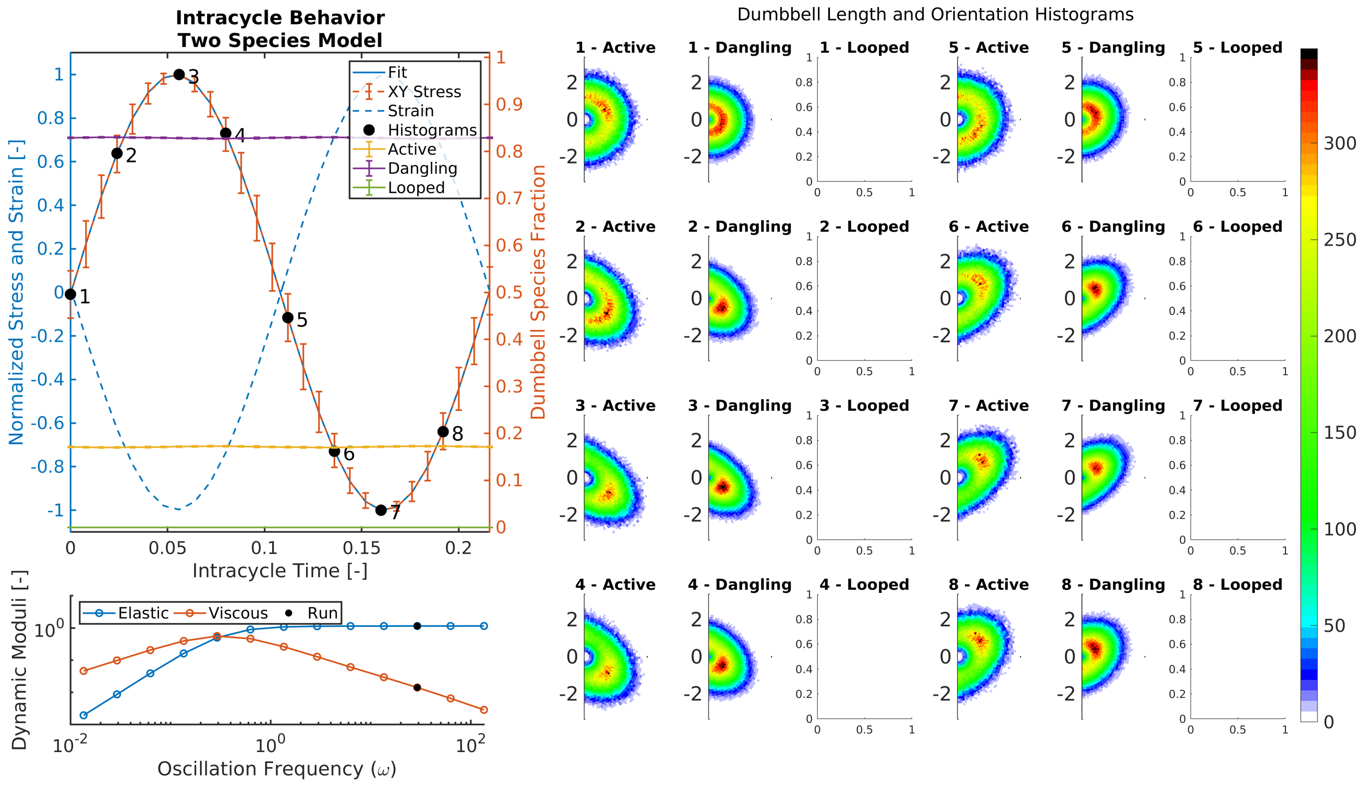

Figure 4.14: Intracycle analysis of a high frequency two species SAOS simulation. *Upper Left* Normalized σ_{xy} stress, strain, curve fit and species fractions over a single cycle. Error bars indicate the range of data over the steady state. Fit refers to the curve generated by fitting σ_{xy} to the function $A\cos\omega t + B\sin\omega t$. The right axis shows the species fractions over the cycle. Error bars indicate variation over the steady state. The numbered dots indicate the corresponding time in the cycle where the corresponding dumbbell configuration histogram is taken. *Right* Dumbbell configuration histograms. To generate the histograms, one end of the dumbbell is fixed at the origin. The placement of the other end is used to for the histogram. Colors represent the number of dumbbell ends in the bin. *Bottom Left* Dynamic moduli for the batch of runs from which the current data is taken. Dots indicate the frequency currently under examination.

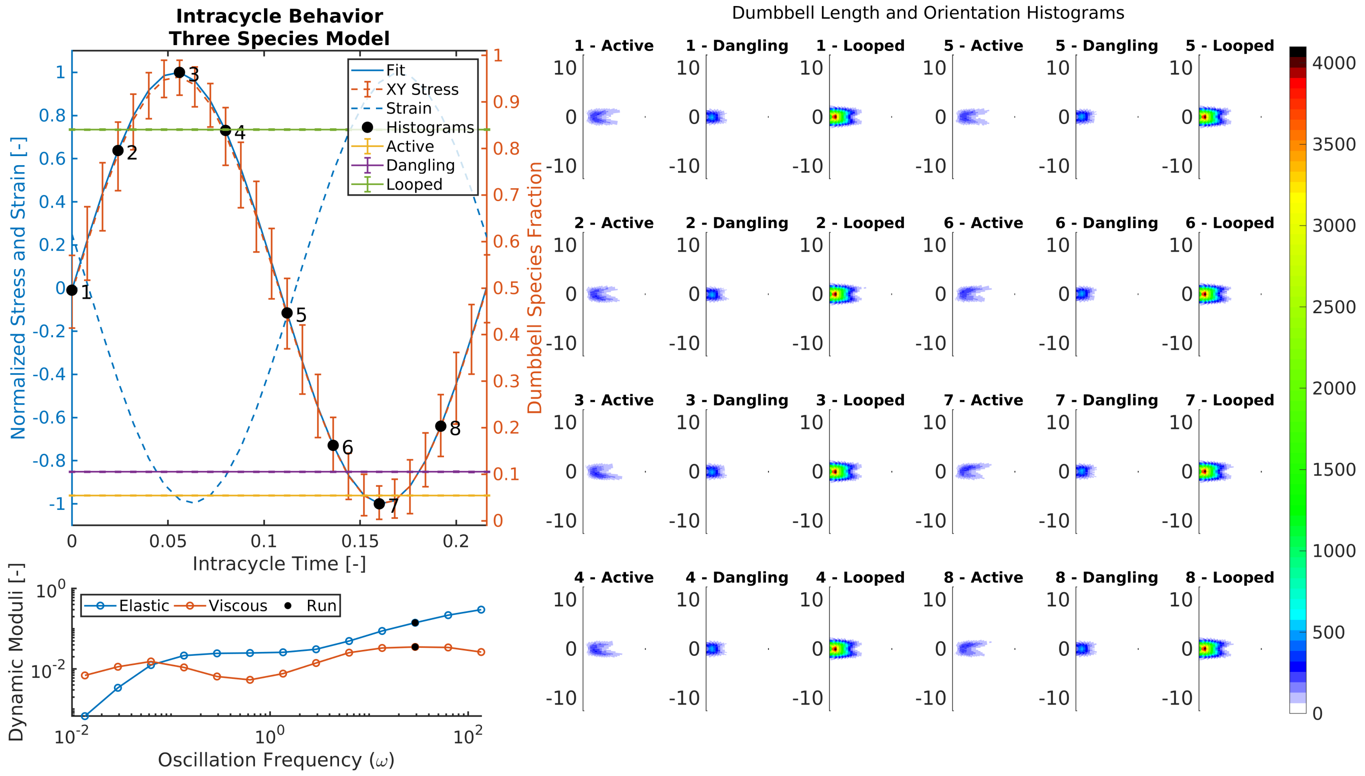

Figure 4.15: Intracycle analysis of a high frequency three species SAOS simulation. *Upper Left* Normalized σ_{xy} stress, strain, curve fit and species fractions over a single cycle. Error bars indicate the range of data over the steady state. Fit refers to the curve generated by fitting σ_{xy} to the function $A\cos\omega t + B\sin\omega t$. The right axis shows the species fractions over the cycle. Error bars indicate variation over the steady state. The numbered dots indicate the corresponding time in the cycle where the corresponding dumbbell configuration histogram is taken. *Right* Dumbbell configuration histograms. To generate the histograms, one end of the dumbbell is fixed at the origin. The placement of the other end is used to for the histogram. Colors represent the number of dumbbell ends in the bin. *Bottom Left* Dynamic moduli for the batch of runs from which the current data is taken. Dots indicate the frequency currently under examination.

Dumbbell configuration plots give insight in the effect of flow on dumbbell orientation. For example, we see in figures 4.10 and 4.11, that dumbbells show little orientation at low frequencies. Mid-range frequency results are shown in figures 4.12 and 4.13. In these plots we began to see the beginning of dumbbells orienting with the flow. At higher frequencies the three species simulation shown in figure 4.15 and the two species in figure 4.14 show different behavior. The two species dumbbell configurations show dumbbells moving with oscillations in the fluid flow. The three species dumbbell configuration shows active dumbbells captured an extended V-shape.

There are two factors behind the V-shape formation found in the configuration histograms for the three species model. One is the competing effects of the spring force and imposed solvent flow, which is also present in the two species model, and results in a preferred orientation angle for most stretched dumbbells. The balance between these forces is further explored mathematically in the following section. The second factor is the length and orientation assigned to a dumbbell after it transitions from the looped to dangling state. For example, when a dangling dumbbell extends in the direction of the flow and becomes looped, it retains that configuration when it transitions back to a dangling dumbbell later. If these dynamics coincide with the oscillations of the fluid flow, a dumbbell becomes looped for the time period where the fluid flow would cause it to retract had it stayed in the dangling state. Moreover, if it reenters as a dangling dumbbell when the flow is moving in the same direction, it will extend further than it would have otherwise. Figure 4.16 shows these dynamics occur in our simulations.

The topological interpretation of the inclusion of this looping behavior depends on the oscillation frequency. At lower frequencies, shorter dumbbells are forming loops correspondent with the lower force of the fluid flow that should accompany less micelle attachments and dangling chains. At mid-level flow force, the effect of loops decreases as dumbbells extend further in the dangling state and make more network

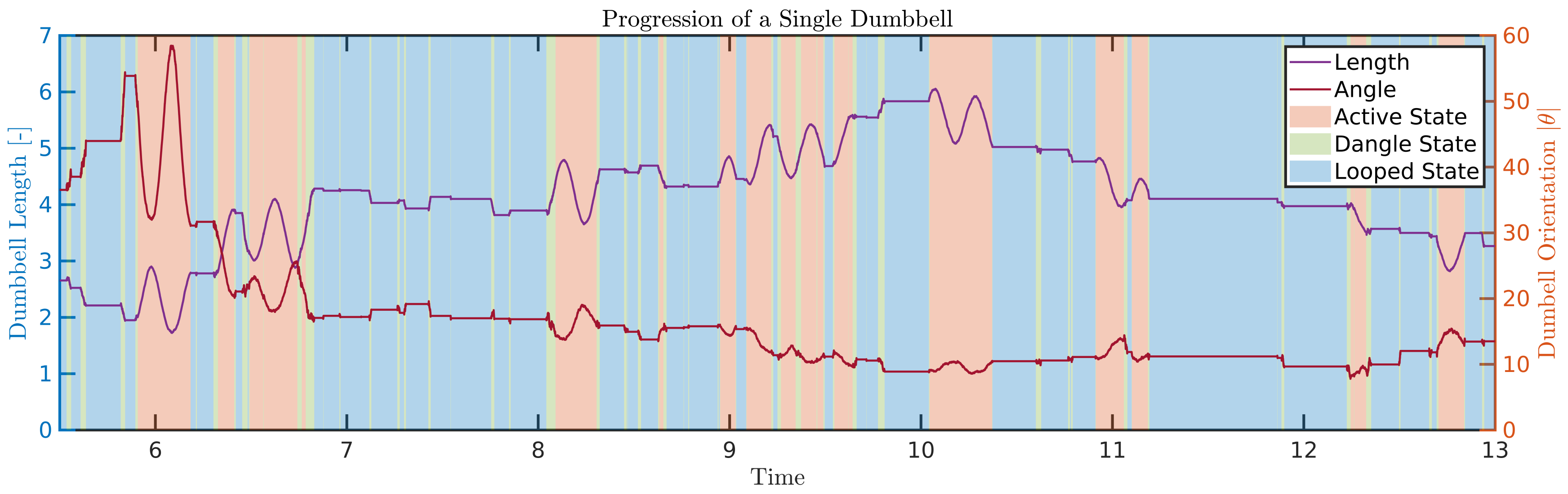

Figure 4.16: Progression of a single dumbbell. Plot shows the length and angle of a single dumbbell in the three species SAOS simulation. The background colors indicate the species state of the dumbbell. Oscillations occur in both directions when the dumbbell is not in the looped state. However, entering and exiting the looped state at opportune times allows the dumbbell to progressively extend in length.

attachments. At higher frequencies the role of loops is to impede movement with the flow in a one direction and increase the potential for extension in length when the flow reverses. The physical scenario that justifies this behavior is the case when a dangling dumbbell which is pushed back towards its micelle core by the oscillating flow causing it to fold back on itself and loop. In this situation, the dumbbell is unable to follow the fluid flow. However, when the flow reverses and the dumbbell breaks out of its looped state it forms a longer dangling dumbbell that extends further with the fluid flow, thus enhancing its effect on the stress response. This line of reasoning follows suggestions that loops or micelles could hinder the relaxation of chains made in previous work [38]. Together we see that a consistent physical interpretation of including a looping species, as we have, depends on a consideration for the dynamics imposed by the movement of the fluid. The merits of this approach are discussed further in the Loop Reentry Methods section.

4.2.1 Mathematical Analysis for Oscillatory Shear Flow

A mathematical analysis of the dumbbell evolution equation provides insight into these dynamics. The non-dimensional form of an explicit computation time step is given by,

$$Q(t+\Delta t) = Q(t) + \kappa \cdot Q(t)\Delta t - \zeta F(Q)\Delta t + \sqrt{\zeta \Delta t} dW \tag{4.4}$$

The constant ζ differs for active and dangling dumbbells and varies with the value of Z in the case of active dumbbells. The function $F(Q)$ is the FENE spring force function. The main terms that affect dumbbell length at the flow rate term, and the spring force term. The Brownian motion term plays a role, but its influence on average is small. The flow rate term for SAOS flow is know precisely. Out of the three terms, the FENE force is the only asymptotic term and therefore has the potential for the greatest impact. However, in our SAOS simulations most springs spend little

Table 4.1: Dumbbell configuration term approximations.

Purpose	Flow Rate	FENE Spring
Term	$\kappa \cdot Q(t)\Delta t$	$\zeta F(Q)\Delta t$
Approximation	$\begin{bmatrix} \gamma_0\omega \left\|Q_y(t)\right\| \Delta t \\ 0 \end{bmatrix}$	$-1.5\zeta \left\|Q(t)\right\| \Delta t$
Description	Accounts for the effects of the oscillating fluid flow. Notice, that the shearing motion only changes a dumbbell's x length based on the y length.	Approximates the spring force for dumbbells of moderate length ($Q < 10$). As dumbbells grow in length, the FENE spring force grows asymptotically and would therefore dominate over any other acting forces.

time in the asymptotic growth region. The nonlinear force in the FENE term, can thus be estimated as $F(Q) \approx 1.5\,|Q|$ for values of Q from 0 to 20. These terms are summarized in table 4.1.

By comparing the values for different orientation angles and flows, we can determine whether the flow force or the FENE spring force plays a more dominant role. This is illustrated by the figure 4.17. The plot contains three sections. In one section the orientation is such that for smaller oscillation rates, the spring force is more responsible for changing the dumbbell configuration. In the other extreme, the flow rate dominates the behavior of the spring. In the middle, it depends on the species of the dumbbell. The lines are estimated by the reasoning above and are not hard boundaries, but instead indicate a balance of forces.

By examining whether the spring force or fluid flow behavior dominates the change in spring configuration, we can clearly distinguish between the phenomena witness in the simulation data. For example, in SAOS simulations at low frequencies, $\omega\gamma_0$

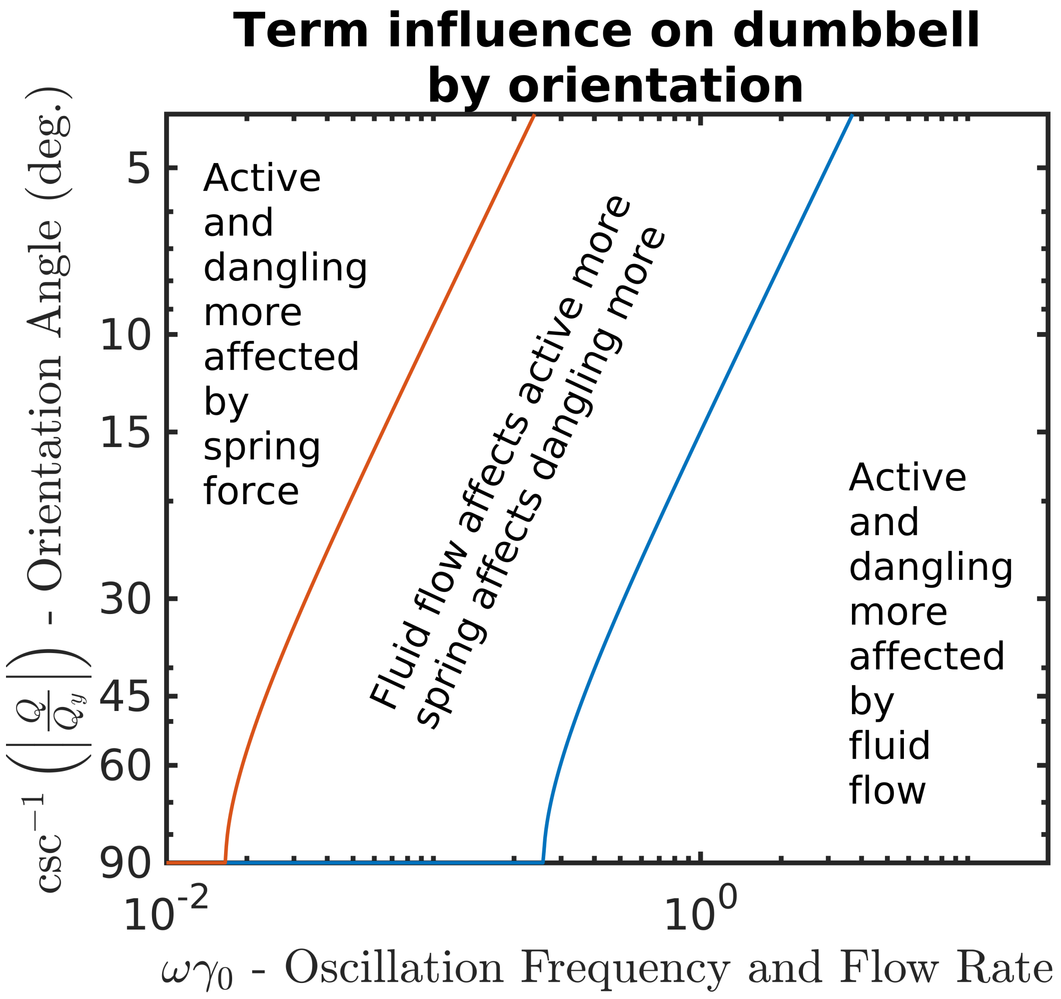

Figure 4.17: Orientation angle versus oscillation frequency and flow rate. There are three regions; a left region where the spring force is the largest factor influencing change in dumbbell configuration, a right region where frequency and flow rate are the larger factor, and a middle region where it depends on the dumbbell species type. Slope and position of dividing lines depend on the value of Z.

is small, and therefore spring behavior is dominated by the FENE force for all orientations. Because the FENE force is isotropic, springs show little alignment in a specific direction. In our simulations, all runs with $\omega \leq 10$ exhibit little alignment. At middle frequencies, springs show more alignment with the flow, however, there is still a large amount of springs randomly aligned. At high frequencies, the flow orients and stretches active springs. As active dumbbells break and become dangling, the spring force dominates and causes the dumbbell to retract.

In the three species model its common for active dumbbells to reach a length and angle where the movement of the fluid flow is not enough to move the dumbbell out of the region where flow is more influential nor is the flow strong enough to cause a breaking transition to the dangling species as they stretch. Instead, the extended dumbbells persist in the angle of their orientation and simply extend and retract with the change in shear direction. The result of this behavior can be seen in the V-shape that appears in the configuration histograms in figure 4.15.

4.3 Loop-to-Dangling Transition Methods

In the SAOS simulations we see that the method of reincorporating loops plays a significant role in the characteristics of the dynamic moduli. In this section, we highlight this difference by comparing two loop re-entry methods. This first method is the one presented in the previous section. In this approach when a dangling dumbbell transitions to the looped species, its length and orientation remain unchanged. When the loops transition back to dangling dumbbells and reenter the stress calculations, they regain their previous configuration. In this way, longer loops form longer dangling dumbbells and shorter loops form shorter dangling dumbbells when they transition species type. Moreover, this type of jumping from coiled to partial stretched state has been seen in other work examining polymer behavior [1]. On the other hand, the concern of this method is that it allows a dumbbell in the looped state to transition to the dangling state with a length and orientation out of phase with the fluid flow. However, the impact of this behavior is overwhelmed by the behavior of the flow itself. This is why we find dumbbells well aligned in the horizontal direction in steady shear flow simulations. Moreover, because the re-entry length and orientation are chosen from a position that resulted from the same type of flow, this method can only be said to be enhancing characteristics already present in the dumbbell population as a result of oscillating shear flow.

In this second method, loops are assigned random lengths chosen from a normal distribution when they transition back to dangling dumbbells. The results of this simulation are show in figure 4.18. The normal distribution is truncated to ensure dumbbells do not exceed the maximum length prescribed by the FENE condition. This approach can be said to replicate the equilibrium behavior of the dumbbells under no flow conditions. The main difference in this approach is that the transition from looped to dangling states causes the resulting dangling dumbbell to be shorter. Choosing from a normal distribution diminishes the impact of the shorter length and allows for differences arising from Brownian fluctuations in the fluid. The drawback of this approach is that the configuration the loops take on when they transition from loop to dangling is disconnected from the fluid flow. In this way, the resulting after-transition configuration is the same even for drastically different fluid flows, such as steady shear and oscillating shear.

In a comparison of simulations using the two methods we see that they lead to significantly different behavior. The looping behavior in the first simulation enhances the effect of oscillating shear flow on the dumbbell configuration. This results in longer dangling and active dumbbells that adapt a V-shape in the configuration histograms. The V-shape also corresponds with increases in both the dynamic moduli. In the second simulation, the looping reentry choice diminishes the effects of the imposed fluid flow because reentry configurations are the same at zero and non-zero flow. The

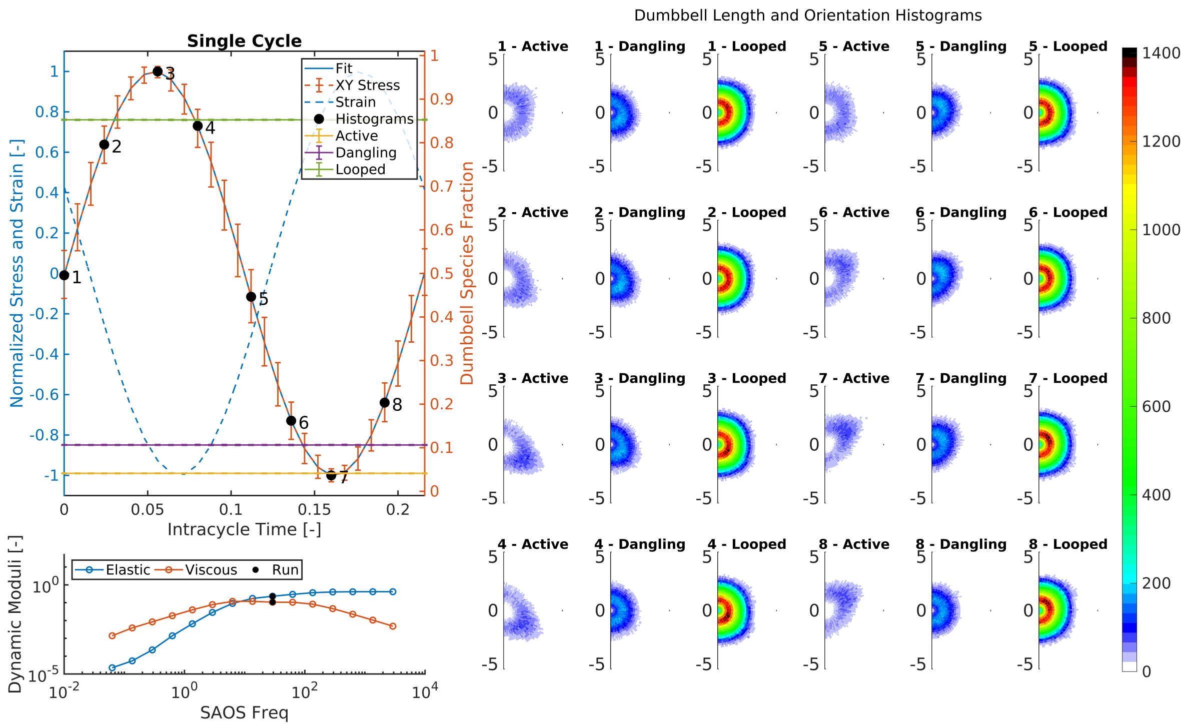

Figure 4.18: Intracycle behavior from a three species SAOS simulation where loops transition to dangling with length draw from a normal distribution. In these simulations, the looped-to-dangling dumbbells take configurations based on a truncated normal distribution. (Upper left) Plot showing intracycle stress, strain and species fractions. (Lower left) Dynamic moduli across a range of frequencies. The intra-cycle is taken from the run indicated by the black dot. (Right) Dumbbell configuration histograms separated by type and intracycle time.

result is that all three species types show more rounded distributions that have less alignment in any specific directions. In addition, the relaxation point shows a marked shift to higher frequencies, indicating the diminished influence of the oscillating shear flow.

On the whole, our simulations show that the method used to transition loops to dangling dumbbells has a significant effect on the dynamic moduli measured in SAOS flows. For both methods, there exists reasonable physical arguments for and against their implementation. However, we chose to focus our efforts on the first method due to the unique behaviors it exhibits and its similarities to experiment. Although comprehensive examination of loop reentry dynamics is beyond the scope of the current work, the results present herein show that it is an area worthy of further study.

4.4 Large Amplitude Oscillatory Shear

In comparison to SAOS, large amplitude oscillatory shear (LAOS) studies have become common place only recently with the advent of more sensitive transducers in commercially available rheometers [41]. LAOS studies go beyond SAOS, using larger deformations to probe nonlinear rheology, whereas the linear viscoelastic theory behind SAOS is only valid for small deformations. In most processing operations polymer deformation is both rapid and large and thus LAOS simulations are necessary for a complete understanding [25]. In light of the significance of this emerging filed, we simulated LAOS deformations and measured the stress response using our three species model with the same attachment detachment and looping parameter values.

A pipkin diagram provides a rheological fingerprint"that illustrates the nonlinear viscoelastic properties of the material response [16]. As with SAOS simulations, the resulting stress from LAOS can be decomposed into the elastic and viscous contributions [12]. For each of these, a pipkin diagram is provided in figure 4.19 for the

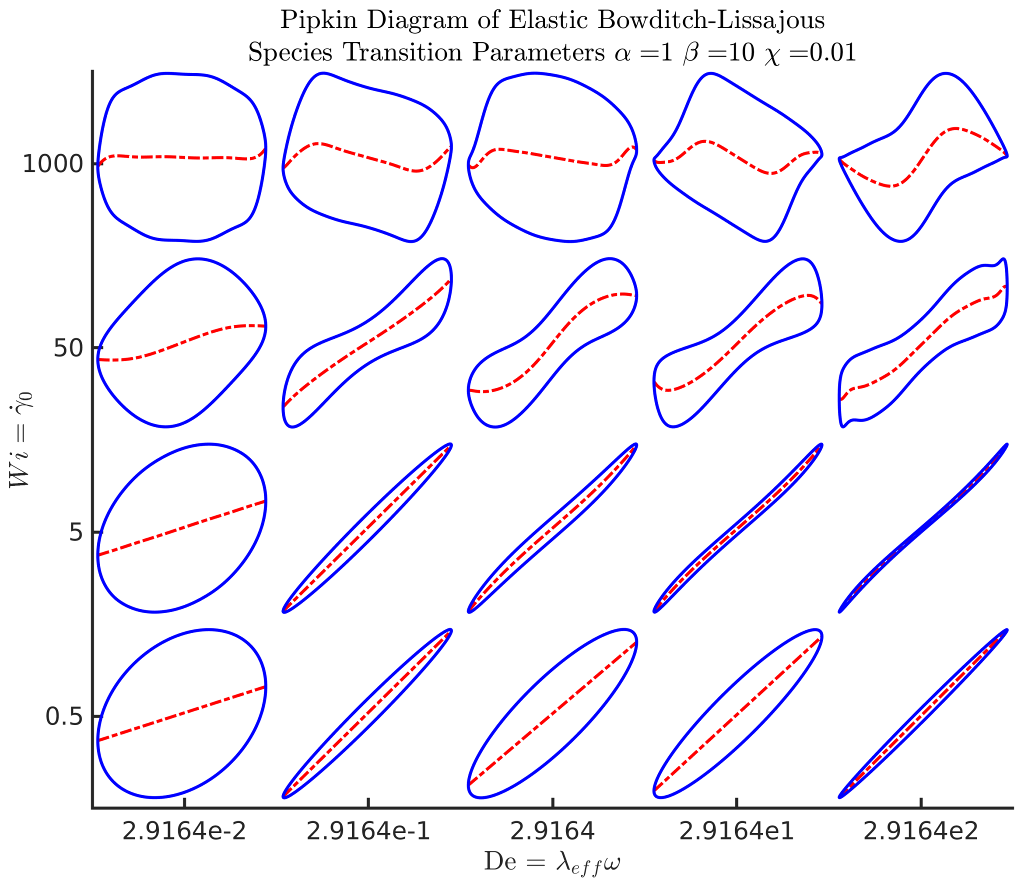

Figure 4.19: Pipkin diagrams showing elastic Bowditch-Lissajou curves form a three species large amplitude oscillatory shear flow simulation. Each simulation was done with parameters $\alpha = 1$, $\beta = 10$ and $\chi = 0.01$. ω is the oscillation frequency, $\lambda_{eff} = 1/\tau = 14.1343$.

elastic component and figure 4.20 for the viscous component. Within each diagram, a grid of Lissajou-Bowditch curves expresses the nature of the nonlinear response at intervals of frequency, ω, and strain amplitude, γ_0. The dashed line represents the elastic stress contribution for the elastic diagram, and the viscous stress in the viscous diagram. The total stress, including both elastic and viscous, is represented by the solid line [9].

In both figures, the bottom row of curves represents results within the linear regime. In this row, the undistorted ellipse indicates a linear material response (this was also confirmed via Fourier Transform techniques, see section 3.3). In figure 4.19,

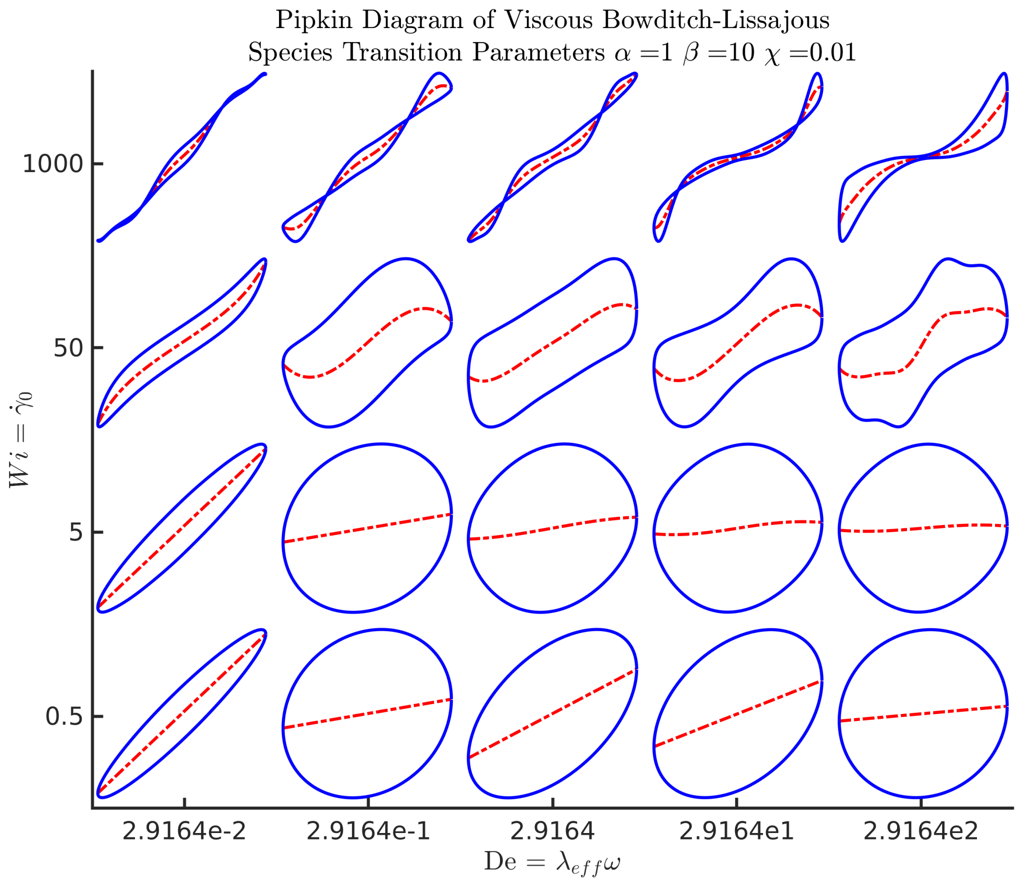

Figure 4.20: Pipkin diagrams showing viscous Bowditch-Lissajou curves form a three species large amplitude oscillatory shear flow simulation. Each simulation was done with parameters $\alpha = 1$, $\beta = 10$ and $\chi = 0.01$. ω is the oscillation frequency, $\lambda_{eff} = 1/\tau = 14.1343$.

a more circular curve indicates a more viscous response. Whereas, a single straight line represents a purely elastic response [16]. Applying this understanding, we see that the Lissajou-Bowditch curves at this strain-amplitude indicate a progression from a viscous response to something more elastic, followed by several more viscous responses, before again shifting back to a more elastic response. This is consistent with results seen in the dynamic moduli plots in figure 4.7 from section 4.2.

As the strain-amplitude increases both diagrams indicate increasing nonlinear responses. By examining the figures column-wise, we see a shift from smooth ellipses to increasingly distorted shapes. This indicates the presence of higher harmonics in the material response that can be used to provide additional detail about the material response. For example, the strength and sign of the third harmonic can be used to indicate strain stiffening or strain softening in the elastic diagram, or shear thickening or thinning in the viscous diagram [16]. In the first column of figure 4.19, the distortion from a circular ellipse to a rounded rectangle between the second and third rows is largely the result of contributions from the third harmonic and indicates intracycle softening. The additional distortions to the shape on the fourth row are an example of a material response with large contributions from additional harmonics beyond the third.

The purpose of the brief analysis above is not to detail all the intricacies of the non-linear response but to instead demonstrate the potential application to LAOS simulations due to the fidelity of the model. Indeed, generating the figures presented here required no modification to the model or oscillatory shear simulation code. Instead, they represent the result of increasing the strain-amplitude parameter and analyzing the results via the MITLaos software [17]. Through this natural extension our model shows the ability to simulate the additional harmonics of the material response found in LAOS simulations. By demonstrating this, we hope to lay the groundwork for a more detailed analysis to appear in future work.

Chapter 5

Conclusion

In this article we present an extension of the Brownian Dynamics approach inspired by the work of Hernández Cifre et al. We incorporate recent developments in the probability of network bridge destruction and offer a new form for the probability of bridge creation. We also include a third species type, looped dumbbells, and offer novel methods of incorporating them into the Brownian dynamics structure previously established. Furthermore, our work employs an efficient parallel computational scheme using GPUs that allows for accurate Brownian dynamics simulations under complex conditions.

Steady shear flow simulations show the ability of our scheme to generate shear-thickening and shear-thinning behavior by independently adjusting bridging and detaching parameters. This distinction demonstrates the added flexibility in the basis of our model which will allow it to be more readily adapted to additional complexities in the future. In three species simulations, we find that including looping dumbbells lowered overall stress and strengthened the nonlinear response. This was largely the result of the looping species regulating the number of dangling and active dumbbells contributing to the stress; a function, that decreased with increasing flow rate.

This work also presents the results of small amplitude oscillatory shear flow with ensembles of 1024000 dumbbells spanning a wide frequency spectrum. In these simulations, including a third looping species makes a clear impact across the frequency spectrum. The most notable portion being an upward turn in both dynamic moduli at medium-to-high frequency oscillations that is not found in two species simulations.

Moreover, we plot configuration histograms which show the existence of a V-shape distribution in the steady state that results from including looping dynamics. This V-shape corresponds to the unique upward turn in the dynamic moduli.

We further examine two separate approaches to the dynamics of the looped-to-dangling species transition. In the first, we allow dumbbells to take on their previous configuration when transitioning from looped to dangling. In the second, the dumbbells are assigned lengths according to a Gaussian distribution truncated to fit within the maximum dumbbell length. A comparison of these two approaches makes it clear that these dynamics play a significant role in the fluid response. Therefore, we identify this aspect of the model as an area important to future research.

To demonstrate the range of fluid flow simulations that our code platform is capable of modeling, we include elastic and viscous Bowditch-Lissajous plots from large amplitude oscillatory shear simulations. Together this work makes a strong case for revisiting the Brownian dynamics approach to simulating complex rheology modelled with FENE dumbbells. Three reasons are the straight forward equations used to describe the molecular motion, the wide range of flow types that can be simulated, and the ease with which multiple species dynamics can be incorporated. Each of these advantages are the result of the mean-field approach which allows the independent simulation of dumbbells to approximate overall network influence on the fluid. Moreover, because our approach can be updated easily, as scientific understanding of micro-rheology in telechelic polymers advances, the capability of this code platform will increase with it.

Bibliography

[1] US Agarwal, Rohit Bhargava, and RA Mashelkar. “Brownian dynamics simulation of a polymer molecule in solution under elongational flow”. In: *The Journal of chemical physics* 108.4 (1998), pp. 1610–1617.

[2] Carolina de las Heras Alarcón, Sivanand Pennadam, and Cameron Alexander. “Stimuli responsive polymers for biomedical applications”. In: *Chem. Soc. Rev.* 34 (3 2005), pp. 276–285. DOI: 10.1039/B406727D. URL: http://dx.doi.org/10.1039/B406727D.

[3] Arlette RC Baljon, Danny Flynn, and David Krawzsenek. “Numerical study of the gel transition in reversible associating polymers”. In: *The Journal of chemical physics* 126.4 (2007), p. 044907.

[4] NP Balsara, M Tirrell, and TP Lodge. “Micelle formation of BAB triblock copolymers in solvents that preferentially dissolve the A block”. In: *Macromolecules* 24.8 (1991), pp. 1975–1986.

[5] George I. Bell. “Models for the Specific Adhesion of Cells to Cells”. In: *Science* 200.4342 (1978), pp. 618–627. ISSN: 00368075, 10959203. URL: http://www.jstor.org/stable/1746930.

[6] R. B. Bird et al. *Dynamics of polymeric liquids.* John Wiley & Sons, 1987.

[7] Andrew H Briggs et al. “Model parameter estimation and uncertainty: a report of the ISPOR-SMDM Modeling Good Research Practices Task Force-6”. In: *Value in Health* 15.6 (2012), pp. 835–842.

[8] B.H.A.A van den Brule and P.J Hoogerbrugge. “Brownian Dynamics simulation of reversible polymeric networks”. In: *Journal of Non-Newtonian Fluid Mechanics* 60.2 (1995), pp. 303–334. ISSN: 0377-0257. DOI: https://doi.org/10.1016/0377-0257(95)01378-4. URL: http://www.sciencedirect.com/science/article/pii/0377025795013784.

[9] Roger W Chan. “Nonlinear viscoelastic characterization of human vocal fold tissues under large-amplitude oscillatory shear (LAOS)”. In: *Journal of rheology* 62.3 (2018), pp. 695–712.

[10] Céline Charbonneau et al. “Controlling the dynamics of self-assembled triblock copolymer networks via the pH”. In: *Macromolecules* 44.11 (2011), pp. 4487–4495.

[11] John Cheng, Max Grossman, and Ty McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014.

[12] Kwang Soo Cho et al. “A geometrical interpretation of large amplitude oscillatory shear response”. In: *Journal of rheology* 49.3 (2005), pp. 747–758.

[13] Andrea De Mauro, Marco Greco, and Michele Grimaldi. “What is big data? A consensual definition and a review of key research topics”. In: *AIP conference proceedings*. Vol. 1644. 1. AIP. 2015, pp. 97–104.

[14] John Dealy and Don Plazek. “Time-temperature superposition—a users guide”. In: *Rheology Bulletin* 78.2 (2009).

[15] Patrick S Doyle, Eric SG Shaqfeh, and Alice P Gast. “Dynamic simulation of freely draining flexible polymers in steady linear flows”. In: *Journal of Fluid Mechanics* 334 (1997), pp. 251–291.

[16] Randy H Ewoldt, AE Hosoi, and Gareth H McKinley. “New measures for characterizing nonlinear viscoelasticity in large amplitude oscillatory shear”. In: *Journal of Rheology* 52.6 (2008), pp. 1427–1458.

[17] RH Ewoldt, P Winter, and GH McKinley. “MITlaos version 2.1 Beta for MATLAB”. In: *Cambridge, MA, self-published: MATLAB-based data analysis software for characterizing nonlinear viscoelastic responses to oscillatory shear strain* (2007).

[18] Paul A Gagniuc. *Markov Chains: From Theory to Implementation and Experimentation*. John Wiley & Sons, 2017.

[19] M. S. Green and A. V. Tobolsky. “A New Approach to the Theory of Relaxing Polymeric Media”. In: *The Journal of Chemical Physics* 14.2 (1946), pp. 80–92. DOI: 10.1063/1.1724109. eprint: https://doi.org/10.1063/1.1724109. URL: https://doi.org/10.1063/1.1724109.

[20] Gary S Grest and Kurt Kremer. “Molecular dynamics simulation for polymers in the presence of a heat bath”. In: *Physical Review A* 33.5 (1986), p. 3628.

[21] Alan Grossfield et al. “Best Practices for Quantification of Uncertainty and Sampling Quality in Molecular Simulations [Article v1. 0]”. In: *Living journal of computational molecular science* 1.1 (2018).

[22] J G Hernández-Cifre et al. "Brownian dynamics simulation of reversible polymer networks under shear using a non-interacting dumbbell model". In: *Journal of Non-Newtonian Fluid Mechanics* 113.2 (2003), pp. 73–96. ISSN: 0377-0257. DOI: https://doi.org/10.1016/S0377-0257(03)00063-6. URL: http://www.sciencedirect.com/science/article/pii/S0377025703000636.

[23] J G Hernández-Cifre et al. "Brownian dynamics simulation of reversible polymer networks using a non-interacting bead-and-spring chain model". In: *Journal of Non-Newtonian Fluid Mechanics* 146.1 (2007). 3rd Annual European Rheology Conference, pp. 3–10. ISSN: 0377-0257. DOI: https://doi.org/10.1016/j.jnnfm.2006.08.010. URL: http://www.sciencedirect.com/science/article/pii/S0377025706002102.

[24] Allan S Hoffman. "Hydrogels for biomedical applications". In: *Advanced drug delivery reviews* 64 (2012), pp. 18–23.

[25] Kyu Hyun et al. "Progress in Polymer Science A review of nonlinear oscillatory shear tests : Analysis and application of large amplitude oscillatory shear (LAOS)". In: *Progress in Polymer Science* 36.12 (2011), pp. 1697–1753. ISSN: 0079-6700. DOI: 10.1016/j.progpolymsci.2011.02.002. URL: http://dx.doi.org/10.1016/j.progpolymsci.2011.02.002.

[26] Vijay Kadam et al. "Structure and Rheology of Self-Assembled Telechelic Associative Polymers in Aqueous Solution before and after Photo-Cross-Linking". In: *Macromolecules* 44.20 (2011), pp. 8225–8232.

[27] Stefan Kallus et al. "Characterization of polymer dispersions by Fourier transform rheology". In: *Rheologica acta* 40.6 (2001), pp. 552–559.

[28] H A Kramers. "The Behavior of Macromolecules in Inhomogeneous Flow". In: 415 (1946). DOI: 10.1063/1.1724163.

[29] Kurt Kremer and Gary S Grest. "Dynamics of entangled linear polymer melts: A molecular-dynamics simulation". In: *The Journal of Chemical Physics* 92.8 (1990), pp. 5057–5086.

[30] Werner Kuhn. "Über die gestalt fadenförmiger moleküle in lösungen". In: *Kolloid-Zeitschrift* 68.1 (1934), pp. 2–15.

[31] Ronald G Larson. *The structure and rheology of complex fluids*. Oxford University Press, 1999.

[32] Claude Le Bris and Tony Lelievre. "Multiscale modelling of complex fluids: a mathematical initiation". In: *Multiscale modeling and simulation in science*. Springer, 2009, pp. 49–137.

[33] Don S Lemons and Paul Langevin. *An introduction to stochastic processes in physics*. JHU Press, 2002.

[34] Kevin Letchford and Helen Burt. “A review of the formation and classification of amphiphilic block copolymer nanoparticulate structures: micelles, nanospheres, nanocapsules and polymersomes”. In: *European Journal of Pharmaceutics and Biopharmaceutics* 65.3 (2007). Drug delivery: a Canadian perspective, pp. 259–269. ISSN: 0939-6411. DOI: https://doi.org/10.1016/j.ejpb.2006.11.009. URL: http://www.sciencedirect.com/science/article/pii/S0939641106003316.

[35] Christopher W Macosko and Ronald G Larson. “Rheology: principles, measurements, and applications”. In: (1994).

[36] Markus Manssen, Martin Weigel, and Alexander K Hartmann. “Random number generators for massively parallel simulations on GPU”. In: *The European Physical Journal Special Topics* 210.1 (2012), pp. 53–71.

[37] Antti Nykänen et al. “Direct Imaging of Nanoscopic Plastic Deformation below Bulk T g and Chain Stretching in Temperature-Responsive Block Copolymer Hydrogels by Cryo-TEM”. In: (2008), pp. 3243–3249.

[38] Linda Pellens et al. “Evaluation of a transient network model for telechelic associative polymers”. In: 121 (2004), pp. 87–100. DOI: 10.1016/j.jnnfm.2004.05.002.

[39] E.A.J.F. Peters. “Polymers in flow — modelling and simulation”. PhD thesis. Sept. 2000. ISBN: 9037001831.

[40] E.A.J.F. Peters and Th.M.A.O.M. Barenbrug. “Efficient Brownian dynamics simulation of particles near walls. I. Reflecting and absorbing walls”. In: *Phys. Rev. E* 66 (5 Nov. 2002), p. 056701. DOI: 10.1103/PhysRevE.66.056701. URL: https://link.aps.org/doi/10.1103/PhysRevE.66.056701.

[41] Simon Rogers. “Large amplitude oscillatory shear: simple to describe, hard to interpret”. In: *Physics Today* 71.7 (2018), pp. 34–40.

[42] Y Séréro et al. “Associating polymers: from “flowers” to transient networks”. In: *Physical Review Letters* 81.25 (1998), p. 5584.

[43] Michelle K Sing, Jorge Ramírez, and Bradley D Olsen. “Mechanical response of transient telechelic networks with many-part stickers”. In: *The Journal of chemical physics* 147.19 (2017), p. 194902.

[44] Michelle K Sing et al. “Celebrating Soft Matter’s 10th Anniversary : Chain configuration and rate-dependent mechanical properties in transient networks”. In: *Soft Matter* 11 (2015), pp. 2085–2096. ISSN: 1744-683X. DOI: 10.1039/C4SM02181A. URL: http://dx.doi.org/10.1039/C4SM02181A.

[45] Ecole Nationale Supe. “Micellization of block copolymers”. In: 28 (2003), pp. 1107–1170. DOI: 10.1016/S0079-6700(03)00015-7.

[46] Shinya Suzuki et al. “Nonlinear rheology of telechelic associative polymer networks: Shear thickening and thinning behavior of hydrophobically modified ethoxylated urethane (HEUR) in aqueous solution”. In: *Macromolecules* 45.2 (2012), pp. 888–898.

[47] KC Tam et al. “A structural model of hydrophobically modified urethane-ethoxylate (HEUR) associative polymers in shear flows”. In: *Macromolecules* 31.13 (1998), pp. 4149–4159.

[48] F Tanaka and SF Edwards. “Viscoelastic properties of physically crosslinked networks: Part 1. Non-linear stationary viscoelasticity”. In: *Journal of Non-Newtonian Fluid Mechanics* 43.2-3 (1992), pp. 247–271.

[49] F Tanaka and SF Edwards. “Viscoelastic properties of physically crosslinked networks: Part 2. Dynamic mechanical moduli”. In: *Journal of non-newtonian fluid mechanics* 43.2-3 (1992), pp. 273–288.

[50] F Tanaka and SF Edwards. “Viscoelastic properties of physically crosslinked networks: Part 3. Time-dependent phenomena”. In: *Journal of non-newtonian fluid mechanics* 43.2-3 (1992), pp. 289–309.

[51] Anubhav Tripathi, Kam C Tam, and Gareth H Mckinley. “Rheology and Dynamics of Associative Polymers in Shear and Extension : Theory and Experiment”. In: 05 (2006), pp. 1981–1999. DOI: 10.1021/ma051614x.

[52] Takashi Uneyama, Shinya Suzuki, and Hiroshi Watanabe. “Concentration dependence of rheological properties of telechelic associative polymer solutions”. In: *Physical Review E* 86.3 (2012), p. 031802.

[53] A Vaccaro and G Marrucci. “A model for the nonlinear rheology of associating polymers”. In: 92.December 1999 (2000), pp. 261–273.

[54] Rui Wang et al. “Classical Challenges in the Physical Chemistry of Polymer Networks and the Design of New Materials”. In: *Accounts of Chemical Research* 49.12 (2016), pp. 2786–2795. ISSN: 15204898. DOI: 10.1021/acs.accounts.6b00454.

[55] Shi Qing Wang. “Transient network theory for shear-thickening fluids and physically crosslinked networks”. In: *Macromolecules* 25.25 (1992), pp. 7003–7010.

[56] Shihu Wang and Ronald G Larson. “Multiple relaxation modes in suspensions of colloidal particles bridged by telechelic polymers”. In: *Journal of Rheology* 62.2 (2018), pp. 477–490.

[57] Harold R Warner Jr. “Kinetic theory and rheology of dilute suspensions of finitely extendible dumbbells”. In: *Industrial & Engineering Chemistry Fundamentals* 11.3 (1972), pp. 379–387.

[58] Manfred Wilhelm, Pierre Reinheimer, and Martin Ortseifer. “High sensitivity Fourier-transform rheology”. In: *Rheologica Acta* 38.4 (Oct. 1999), pp. 349–356. ISSN: 1435-1528. DOI: 10.1007/s003970050185. URL: https://doi.org/10.1007/s003970050185.

[59] Manfred Wilhelm et al. “The crossover between linear and non-linear mechanical behaviour in polymer solutions as detected by Fourier-transform rheology”. In: *Rheologica Acta* 39.3 (2000), pp. 241–246.

[60] Mark Wilson, Avinoam Rabinovitch, and Arlette RC Baljon. “Computational study of the structure and rheological properties of self-associating polymer networks”. In: *Macromolecules* 48.17 (2015), pp. 6313–6320.

[61] Hans M Wyss et al. “Strain-rate frequency superposition: A rheological probe of structural relaxation in soft materials”. In: *Physical review letters* 98.23 (2007), p. 238303.

[62] Misazo Yamamoto. “The visco-elastic properties of network structure I. General formalism”. In: *Journal of the physical society of Japan* 11.4 (1956), pp. 413–421.

[63] Misazo Yamamoto. “The visco-elastic properties of network structure II. Structural viscosity”. In: *Journal of the physical society of Japan* 12.10 (1957), pp. 1148–1158.

[64] Misazo Yamamoto. “The visco-elastic properties of network structure III. Normal stress effect (Weissenberg effect)”. In: *Journal of the physical society of Japan* 13.10 (1958), pp. 1200–1211.

[65] Takehiro Yamamoto and Nobuhiro Kanda. “Brownian Dynamics Simulation for Shear Flow of Entangled Polymer Systems Using a Reversible Network Model”. In: *Nihonkikaigakkai ronbun-shū B-hen* 79.802 (2013), pp. 1072–1080. DOI: 10.1299/kikaib.79.1072.

[66] Takehiro Yamamoto and Nobuhiro Kanda. “Computational model for Brownian dynamics simulation of polymer/clay nanocomposites under flow”. In: *Journal of Non-Newtonian Fluid Mechanics* 181-182 (2012), pp. 1–10. ISSN: 0377-0257. DOI: `https://doi.org/10.1016/j.jnnfm.2012.06.005`. URL: `http://www.sciencedirect.com/science/article/pii/S0377025712001139`.

[67] Yue Zhao et al. “Self-assembly of poly (caprolactone-b-ethylene oxide-b-caprolactone) via a microphase inversion in water”. In: *The Journal of Physical Chemistry B* 105.4 (2001), pp. 848–851.

[68] SN Zhurkov. “Kinetic concept of the strength of solids”. In: *International Journal of Fracture* 26.4 (1984), pp. 295–307.

Appendix A

Derivation of the Dumbbell Equation

In the elastic bead-spring or dumbbell model [30] a macromolecule is idealized as an "elastic dumbbell". Each endpoint of the dumbbell undergoes drag from two sources. The first is due to movement of the endpoint itself. The second is due to the flow of fluid around it. Endpoint collisions with molecules in the solvent lead to Brownian motion. The edge between each endpoint represents a molecular chain which has a maximum length and a resistance to stretching due to a preferred configuration in the solution. Therefore, a FENE spring force [57] is employed. In the model interpretation we assume the inertial forces to be considered negligible in comparison to other forces in the model, as is convention [6]. Collecting these factors in a force balance equation yields a mathematical expression for the behavior of a single endpoint.

$$\sum F_i = \text{Movement Drag} + \text{Fluid Flow Drag} + \text{Spring Force} + \text{Brownian Motion} \tag{A.1}$$

Using Newton's second law $\sum F = ma$, expressions for the forces, and assuming negligible inertia $ma = 0$, we then get,

$$0 = -\frac{1}{\zeta} d\boldsymbol{x}_i + \boldsymbol{F}_{Spring}(\boldsymbol{x}_i)dt + \frac{1}{\zeta}\boldsymbol{v}_i dt + \sqrt{\frac{k_B T}{\zeta}dt}d\boldsymbol{W} \tag{A.2}$$

$$d\boldsymbol{x}_i = \zeta \boldsymbol{F}_{Spring}(\boldsymbol{x}_i)dt + \boldsymbol{v}dt + \sqrt{k_B T \zeta dt}d\boldsymbol{W}. \tag{A.3}$$

Now let $\boldsymbol{Q}$ be the edge connecting the two end points so that $\boldsymbol{Q} = \boldsymbol{x}_2 - \boldsymbol{x}_1$. The conceptualization of $\boldsymbol{Q}$ shown in figure 2.3 and is the source of the term 'Elastic Dumbbell'. Then,

$$d\left(\boldsymbol{x}_2-\boldsymbol{x}_1\right)=-\zeta \boldsymbol{F}_{Spring}(\boldsymbol{x}_2-\boldsymbol{x}_1)dt+\left(\boldsymbol{v}_2-\boldsymbol{v}_1\right)dt+\sqrt{k_BT\zeta dt}\left(d\boldsymbol{W}_2-d\boldsymbol{W}_1\right) \tag{A.4}$$

$$d\left(\boldsymbol{Q}\right)=-\zeta \boldsymbol{F}_{Spring}(\boldsymbol{Q})dt+\left(\nabla \boldsymbol{v}\right)dt+\sqrt{k_BT\zeta dt}\left(d\boldsymbol{W}_2-d\boldsymbol{W}_1\right). \tag{A.5}$$

By the Normal Sum Theorem [33], $d\boldsymbol{W}_2 \pm d\boldsymbol{W}_1 = \sqrt{2}d\boldsymbol{W}$. In addition, its common to let $\boldsymbol{\kappa} = (\nabla \boldsymbol{v})^T$. Incorporating these two adjustments leads to,

$$d\boldsymbol{Q}=-\zeta \boldsymbol{F}_{Spring}(\boldsymbol{Q})dt+\kappa \cdot \boldsymbol{Q}dt+\sqrt{2k_BT\zeta dt}d\boldsymbol{W}. \tag{A.6}$$

This equation is what is used to simulate the polymer segment behavior.

Appendix B

Numerical Scheme

A semi-implicit absolutely stable numerical scheme evolves the dumbbells over time. Put forth by E.A.L.F, Peters in [39], the scheme is only first order accurate. However, the value of the approach comes from the relative computational cost of each step. Due to the stochastic nature of the simulation, a large number of realizations must be computed to lower noise in the model. Under this consideration, the ability to calculate many individual dumbbells quickly is balanced by the need for accuracy in each. More accurate schemes exist, however employing and developing them was not a focus of the current work. The scheme as it applies to this application is derived in full below.

The following equation describes the change in configuration of a single dumbbell,

$$d\mathbf{Q} = \kappa \cdot \mathbf{Q}dt - H\zeta \frac{\mathbf{Q}}{1 - Q^2/Q_{max}^2} dt + \sqrt{2k_B T\zeta}d\mathbf{W}. \tag{B.1}$$

Using the formula,

$$\frac{d}{dt}\left|\mathbf{Q}\right|^2 = 2\left(\frac{d}{dt}\mathbf{Q}\right) \cdot \mathbf{Q}, \tag{B.2}$$

the change in dumbbell length can be found as,

$$d\left|Q\right|^2 = 2\mathbf{Q} \cdot \kappa \cdot \mathbf{Q}dt - 2H\zeta \frac{Q^2}{1 - Q^2/Q_{max}^2} dt + 2\mathbf{Q} \cdot \sqrt{2k_B T\zeta}d\mathbf{W}. \tag{B.3}$$

The numerical scheme first calculates changes in the orientation and length due to the flow and Brownian motion in an explicit step. Then the change in length due to the FENE spring force is added in a second implicit step.

$$\mathbf{Q_1} = \mathbf{Q} + \kappa \cdot \mathbf{Q}dt + \sqrt{2k_BT\zeta}d\mathbf{W} \tag{B.4}$$

$$Q^2 = |\mathbf{Q_1}|^2 - 2H\zeta\frac{Q^2}{1 - Q^2/Q_{max}^2} \tag{B.5}$$

$$\mathbf{Q} = \sqrt{\frac{Q^2}{Q_1^2}}\mathbf{Q_1} \tag{B.6}$$

Equation B.5 is quadratic in Q^2.

$$Q^2 = |\mathbf{Q_1}|^2 - 2H\zeta\frac{Q_{max}^2 Q^2}{Q_{max}^2 - Q^2}dt \tag{B.7}$$

$$\left(Q_{max}^2 - Q^2\right) Q^2 = |\mathbf{Q_1}|^2 \left(Q_{max}^2 - Q^2\right) - 2H\zeta Q_{max}^2 Q^2 dt \tag{B.8}$$

$$Q^4 + -Q^2 \left[1 + 2H\zeta dt + |\mathbf{Q_1}|^2 /Q_{max}^2\right] |\mathbf{Q_1}|^2 = 0 \tag{B.9}$$

This results in two solutions for Q^2. The solution with $|\mathbf{Q}| < Q_{max}$ is given by the solution:

$$\frac{2Q^2}{Q_{max}^2} = \left[1 + 2H\zeta dt + |\mathbf{Q_1}|^2 /Q_{max}^2\right] + \sqrt{\left[1 + 2H\zeta dt + |\mathbf{Q_1}|^2 /Q_{max}^2\right]^2 - 4\frac{|\mathbf{Q_1}|^2}{Q_{max}^2}} \tag{B.10}$$

Multiplying by the conjugate gives,

$$\frac{2Q^2}{Q_{max}^2} = \tag{B.11}$$

$$\frac{\left[1 + 2H\zeta dt + |\mathbf{Q_1}|^2/Q_{max}^2\right]^2 - \left[1 + 2H\zeta dt + |\mathbf{Q_1}|^2/Q_{max}^2\right]^2 + 4|\mathbf{Q_1}|^2/Q_{max}^2}{\left[1 + 2H\zeta dt + |\mathbf{Q_1}|^2/Q_{max}^2\right] + \sqrt{\left[1 + 2H\zeta dt + |\mathbf{Q_1}|^2/Q_{max}^2\right]^2 - 4|\mathbf{Q_1}|^2/Q_{max}^2}} \tag{B.12}$$

$$\frac{Q^2}{|\mathbf{Q_1}|^2} = \tag{B.13}$$

$$\frac{2}{\left[1 + 2H\zeta dt + |\mathbf{Q_1}|^2/Q_{max}^2\right] + \sqrt{\left[1 + 2H\zeta dt + |\mathbf{Q_1}|^2/Q_{max}^2\right]^2 - 4|\mathbf{Q_1}|^2/Q_{max}^2}}. \tag{B.14}$$

Appendix C

Cuda C Code

The CUDA C simulation code is presented below.

```
/*
 * Erik Palmer
 * 10-22-2015
 *
 * Three species dumbbell simulation
 *
 * Evolves population of dumbbells over time according to
 * flow characterists and species switching probabilities.
 * Produces a measure of the stresses on the fluid.
 *
 * To Compile:
 * nvcc <filename.cu> -lcurand -o <output file>
 *
 * GelModel:
 * Use transistion probabilities from the physical arguments.
 * Add ifdefs to control SAOS, and other aspects of the model
 *
 *
 * SPECIES GUIDE:
 * Int | Type
 * 0 | Polymer One — Active
 * 1 | Polymer One — Dangling
 * 2 | Polymer Two — Active
 * 3 | Polymer Two — Dangling
 *
 *
 */


#include <stdio.h>
//required to compile on windows, must be before math.h
#define _USE_MATH_DEFINES
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <errno.h>
#include <ctype.h>
#include <stdint.h> //added to use unsigned int32

#include <cuda.h>
#include <curand.h>
#include <curand_kernel.h>
//#include <math_functions.h>
#include <unistd.h> //added to check for file existence



//Define Macros for Error handling
```

```

#define CUDA_CALL(x) do { if((x)!=cudaSuccess) { \
          printf("Error at %s:%d\n", __FILE__,__LINE__); \
          return EXIT_FAILURE; }} while(0)
#define CURAND_CALL(x) do { if((x)!= CURAND_STATUS_SUCCESS) { \
          printf("Error at %s:%d\n", __FILE__, __LINE__); \
          return EXIT_FAILURE; }} while(0)
//This one is better because it also outputs the error message
#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__);}
inline void gpuAssert(cudaError_t code, const char *file, int line,
                    bool abort=true)
{
          if (code != cudaSuccess)
          {
          fprintf(stderr, "GPUassert: %s %s %d\n", cudaGetErrorString(code), file,
                      line);
          if (abort) exit(code);
          }
}

//define maximum filesize for raw data file 5e10 bytes = 50GB
#define RAWDATA_MAX_FILESIZE 1e10

#define MICRODATA_MAX_FILESIZE 2e11


//Define Macro for Histogram debugging
#define PRINT_VAR(x) printf("" #x "\n ")

```

```
//Debugging Macros
#define PRINT_VAR_FLOAT_VALUE(x) printf("" #x "=%f\n", x)
#define PRINT_VAR_INT_VALUE(x) printf("" #x "=%d\n", x)
//* Also useful: printf("DEBUG LINE %d\n", __LINE__);


//____velocity field on-off matrix _____
// note that this matrix is multiplied by the inputed flowrate value
#define U11 0.0
#define U12 0.0
#define U21 1.0
#define U22 0.0
//''''''''''''''''''''''''''''''''

//___ Name for .csv file ____
#define OUTPUT_FILENAME "THREESPECIES"

/* Name for Raw Output File */
#define RAWDATA_FILENAME "RAWDATA"



#define INIT_ACT_TO_DNG_RATIO 0.5
#define TAO_FUND 5e-6   //Default 5e-6
#define ZEE 10.0      //Default 10.0
#define CHI 0.03      //Default 0.83
#define ALPHA 0.1     //Default 0.17
#define BETA 0.1      //Default 0.17
#define D_FREE 12.0     //Default 12.0
```

```

#define A_COEFF 1.2026e6
#define B_COEFF 6.4286e-5

//''''''''''''''''''''''''''''''''''



//______Define Global Variables___________
//For GPU
__device__ double devStepSizeMicro;
__device__ double devFlowRate;
__device__ double devMaxSpringLength;
__device__ double devFreq;


//For CPU
static unsigned int hostNumberOfParticles = 0;
static double hostStepSizeMicroFirst = 0;
static double hostStepSizeMicroSecon = 0;
static unsigned int hostTimeStepsMicro = 0;
static unsigned int hostTimeStepsMacro = 0;
static double hostFlowRate;
static double hostMaxSpringLength;
static double hostFreq;
static unsigned int hostMacroStepSizeSplitPt = 0;

//Additional Commandline Arguments
//GPU
```

```
__device__ double devD_free;
__device__ double devZee;
__device__ double devChi;
__device__ double devAlpha;
__device__ double devBeta;

//CPU
static double hostD_free = D_FREE;
static double hostZee = ZEE;
static double hostChi = CHI;
static double hostAlpha = ALPHA;
static double hostBeta = BETA;

static double Init_Active_Ratio;
static double Init_Dangle_Ratio;

#ifdef NO_REPORT
  /*
   * This variable wasn't doing anything useful so I hijacked it to
   * create a period of the simulations where the output is not sent to the
   * cpu to report it. Instead it stays on the GPU. Seems to speed things
   * up quite a bit so far.
   */
  static unsigned long long hostA_coeff = A_COEFF;
#else
  static double hostA_coeff = A_COEFF;
#endif

static double hostB_coeff = B_COEFF;

static int GPU_select;
static char RawData_select[256];
static char DataFileName[256];
//''''''''''''''''''''''''''''''''''''


//____ Struct defintions ____
typedef struct SpeciesValue {
      double ActiveLen;
      double DangleLen;
      double LoopedLen;
      double ActiveAng;
      double DangleAng;
      double LoopedAng;
      double ActiveX;
      double DangleX;
      double LoopedX;
      double ActiveY;
      double DangleY;
      double LoopedY;
} SpeciesValue;

typedef struct SpeciesCount {
      int Active;
      int Dangle;
      int Looped;
} SpeciesCount;

```

```
typedef struct TwoDimSpring {
      double x;
      double y;
} TwoDimSpring;

typedef struct Stress {
      double XX;
      double XY;
      double YY;
} Stress;

typedef struct Dumbbell {
      int type;
      double x;
      double y;
} Dumbbell;

#ifdef SINGLE_MICRO
typedef struct DBSpecChng {
   int type;
   double time;
   double x;
   double y;
} DBSpecChng;
#endif
#ifdef MICRO_RAW
typedef struct DBSpecChng {
   int type;
   double length;
```

```
} DBSpecChng;
#endif

//''''''''''''

//Function: ParseInput
//Sorts and examines command line input for inappropriate data
int ParseInput(int argc, char *argv[]){



        if ( argc > 1 && argc != 22 ){
                printf("ERROR: Incorrect number of input arguments. 20 required.\n");
                printf("Format: %s \n [number of dumbbells]\n", argv[0]);
                printf(" [micro step size stage 1]\n [micro step size stage 2]\n ");
                printf(" [micro time steps per macro step]\n ");
                printf(" [total steps macro]\n ");
                printf(" [number of macro steps with micro step size stage 1]\n ");
                printf(" [flow rate]\n [Maximum Spring Length]\n");
                printf(" [SAOS frequency]\n [Drag Coefficient]\n [Z]\n"
                    "[Alpha0]\n [Alpha1]\n");
    printf(" [Beta]\n");
    printf(" [Initial Percentage of Active Dumbbells]\n");
    printf(" [Initial Percentage of Dangling Dumbbells]\n");
                printf(" [A Coefficient]\n [B Coefficient]\n");
                printf(" [GPU Device (0 or 1)]\n [Write Raw Data (Y or N)]\n");
                printf(" [Output Filename]\n");
                return EXIT_FAILURE;
        } else if (argc ==1){

            //Use default values
        printf("Using default values.\n");

        hostNumberOfParticles = 1048576;
                hostStepSizeMicroFirst = 0.1;
                hostStepSizeMicroSecon = 0.001;
                hostTimeStepsMicro = 100;
                hostTimeStepsMacro = 100;
                hostMacroStepSizeSplitPt = 50;
                hostFlowRate = 1.0;
                hostMaxSpringLength = 5.0;
                hostFreq = 1.0;
                hostD_free = D_FREE;
                hostZee = ZEE;
                hostChi = CHI;
                hostAlpha = ALPHA;
                hostA_coeff = A_COEFF;
                hostB_coeff = B_COEFF;
        Init_Active_Ratio = 0.5;
        Init_Dangle_Ratio = 0.5;

        GPU_select = 0;

        strcpy(RawData_select, "N");
        strcpy(DataFileName, "DEFAULT");

        return(0);
        }


            errno = 0;

      //Num of dumbbells
            hostNumberOfParticles = strtoul(argv[1], NULL, 10);
      //micro step size stage 1
            hostStepSizeMicroFirst = strtod(argv[2], NULL);
      //micro step size stage 2
            hostStepSizeMicroSecon = strtod(argv[3], NULL);
      //number of micro time steps looped on GPU per single CPU macro step
            hostTimeStepsMicro = strtoul(argv[4], NULL, 10);
      //total number of macro time steps for the simultation
            hostTimeStepsMacro = strtol(argv[5], NULL, 10);
      //number of macro steps using micro steps of size stage 1
            hostMacroStepSizeSplitPt = strtol(argv[6], NULL, 10);
            hostFlowRate = strtod(argv[7], NULL);
            hostMaxSpringLength = strtod(argv[8], NULL);
            hostFreq = strtod(argv[9], NULL);

            //additional command line arguments
            hostD_free = strtod(argv[10], NULL);
            hostZee = strtod(argv[11], NULL);
            hostChi = strtod(argv[12], NULL);
            hostAlpha = strtod(argv[13], NULL);
            hostBeta = strtod(argv[14], NULL);
            Init_Active_Ratio = strtod(argv[15], NULL);
            Init_Dangle_Ratio = strtod(argv[16], NULL);
    #ifdef NO_REPORT
        hostA_coeff = strtoull(argv[17], NULL, 10);
```

```
#else
        hostA_coeff = strtod(argv[17], NULL);
    //New use: Number of macro steps to skip before recording data.
   //(Speed sim runtime)
#endif
        hostB_coeff = strtod(argv[18], NULL);

    GPU_select = (int) strtol(argv[19], NULL,10);
    strcpy(RawData_select, argv[20]);
    strcpy(DataFileName, argv[21]);


        if (hostNumberOfParticles==0){
                printf("Unable to convert %s to positive integer\n", argv[1]);
                return EXIT_FAILURE;
        }

        if (hostStepSizeMicroFirst==0){
                printf("Unable to convert %s to double\n", argv[2]);
                return EXIT_FAILURE;
        }

        if (hostStepSizeMicroSecon==0){
                printf("Unable to convert %s to double\n", argv[3]);
                return EXIT_FAILURE;
        }

        if (hostTimeStepsMicro==0){
                printf("Unable to convert %s to positive integer\n", argv[4]);
```

```
            return EXIT_FAILURE;
        }

        if (hostTimeStepsMacro==0){
            printf("Unable to convert %s to positive integer\n", argv[5]);
            return EXIT_FAILURE;
        }

        if (hostMacroStepSizeSplitPt==0){
            printf("Unable to convert %s to positive integer\n", argv[6]);
            return EXIT_FAILURE;
        }

        if (hostMaxSpringLength == 0){
            printf("Unable to convert %s to positive double\n", argv[8]);
            return EXIT_FAILURE;
        }

        if (hostFreq == 0){
            printf("Unable to convert %s to positive double\n", argv[9]);
            return EXIT_FAILURE;
        }

        //___ additional command line arguments ____
        if (hostD_free == 0){
            printf("Unable to convert %s to positive double\n", argv[10]);
            return EXIT_FAILURE;
        }
        if (hostZee == 0){
                printf("Unable to convert %s to positive double\n", argv[11]);
                return EXIT_FAILURE;
        }

        if (hostA_coeff == 0){
                printf("hostA_coeff input error:"
                    " Unable to convert %s to positive double\n", argv[17]);
                return EXIT_FAILURE;
        }
        if (hostB_coeff == 0){
                printf("Unable to convert %s to positive double\n", argv[18]);
                return EXIT_FAILURE;
        }

    switch(RawData_select[0]){
        case 'N':
        case 'n':
            strcpy(RawData_select,"No");
            break;
        case 'Y':
        case 'y':
            strcpy(RawData_select,"Yes");
            break;
        default:
            printf("The only valid choices to write raw data file are:"
                           " Y,y,N,n\n");
            return EXIT_FAILURE;
    }

```

```
if ((RawData_select[0]!='Y')&&(RawData_select[0]!='N')
                    &&(RawData_select[0]!='y')&&(RawData_select[0]!='n'))
{
    printf("The only valid choices to write raw data file are: Y,y,N,n\n");
            return EXIT_FAILURE;
}
    //''''''''''''''''''''''''''''''''''''''''

//Check to see if same filename for output exists.
// If the file exists, exit the program.
// This was done to fix the restarting issue

char CheckFilename[264];

sprintf(CheckFilename, "%s.csv", DataFileName);

if ( access ( CheckFilename, F_OK) != - 1 ){

    printf("File: %s exists, exiting program.\n", CheckFilename);
    return EXIT_FAILURE;
}

else {
    //create empty file to hold the sapce.

    FILE *OutputFile = NULL;
        OutputFile = fopen(CheckFilename, "w");

        if (OutputFile == NULL){
                fprintf(stderr, "Couldn't open output file: %s!\n", CheckFilename);
                exit(1);
            }

        fclose(OutputFile);
    }

    //Also check for bin file
    sprintf(CheckFilename, "%s.bin", DataFileName);

    if ( access ( CheckFilename, F_OK) != -1 ){

        printf("File: %s exists, exiting program.\n", CheckFilename);
        return EXIT_FAILURE;
    }



        if (errno == ERANGE){
            printf("%s\n", strerror(errno));
            return EXIT_FAILURE;
        }



        return 0;
}
```

```



//Function PrinSimInfo
//Prints to terminal information about the current simulation
void PrintSimInfo(){

        // ____ Calculate and output program parameters _______
        printf("______________ Running Simulation ______________________\n");
#ifdef SIMPLE_SHEAR
    printf("|| Simple Shear Flow \n");
#else
    printf("|| Small Oscillatory Shear Flow\n");
#endif
#ifdef LOOPED_DUMBBELLS
    printf("|| Dumbbell Types: Active, Dangling and Looped\n");
#else
    printf("|| Dumbbell Types: Active, and Dangling\n");
#endif



        printf("|| Total Time: %g \n",
          (hostStepSizeMicroFirst * hostMacroStepSizeSplitPt
      + hostStepSizeMicroSecon
      * (hostTimeStepsMacro - hostMacroStepSizeSplitPt))
      * hostTimeStepsMicro );
        printf("|| ---- Time Step Parameters ---- \n" );
        printf("|| Total Number of Macro Steps: %u\n", hostTimeStepsMacro);
        printf("|| Micro Steps Per Macro Iteration: %u\n", hostTimeStepsMicro);
        printf("|| Macro Step Size Split Point: %u\n", hostMacroStepSizeSplitPt);
    printf("||\n");
        printf("|| --- Stage One --- \n");
        printf("|| Micro Step Size: %1.12g\n", hostStepSizeMicroFirst);
        printf("|| Macro Step Size: %1.12g\n", hostStepSizeMicroFirst
                                            * hostTimeStepsMicro);
        printf("|| Number of Macro Steps: %u\n", hostMacroStepSizeSplitPt);
        printf("|| Stage One Total Time: %1.12g\n", hostStepSizeMicroFirst
                               * hostTimeStepsMicro * hostMacroStepSizeSplitPt);
        printf("|| Flow Rate: 0 \n");
  printf("||\n");
        printf("|| --- Stage Two --- \n");
        printf("|| Micro Step Size: %1.12g\n", hostStepSizeMicroSecon);
        printf("|| Macro Step Size: %1.12g\n", hostStepSizeMicroSecon
                                              * hostTimeStepsMicro);
        printf("|| Number of Macro Steps: %u\n", hostTimeStepsMacro
                                                    - hostMacroStepSizeSplitPt);
        printf("|| Stage Two Total Time: %1.12g\n", hostStepSizeMicroSecon
      * hostTimeStepsMicro * (hostTimeStepsMacro - hostMacroStepSizeSplitPt));
        printf("|| Flow Rate: %g \n", hostFlowRate);
  printf("||\n");
        printf("|| ----- Simulation Parameters ---- \n");
        printf("|| Number of Particles: %u\n", hostNumberOfParticles);
        printf("|| Maximum Spring Length: %g\n", hostMaxSpringLength );
        printf("|| SAOS Frequency: %g\n", hostFreq );
        printf("|| d: %g\n", hostD_free );
        printf("|| Z: %g\n", hostZee );
        printf("|| Chi: %g\n", hostChi);
```

```
            printf("|| Alpha: %g\n", hostAlpha);
            printf("|| Beta: %g\n", hostBeta);
            printf("|| Initial Active Ratio: %g\n", Init_Active_Ratio);
            printf("|| Initial Dangling Ratio: %g\n", Init_Dangle_Ratio);
            printf("|| Initial Looped Ratio: %g\n",
                        1-Init_Dangle_Ratio-Init_Active_Ratio);
#ifdef NO_REPORT
            printf("|| Macro Step Jump: %llu\n", hostA_coeff);
#else
            printf("|| A Coefficient: %g\n", hostA_coeff);
#endif
            printf("|| B Coefficient: %g\n", hostB_coeff);
   printf("||\n");
   printf("|| ----- Program Options ----- \n");
   printf("|| Running on GPU device: %d\n", GPU_select);
   printf("|| Write Raw Data: %s\n", RawData_select);
   printf("|| Output Filename: %s\n", DataFileName);
            printf(" - - - - - - - - - - - - - - - - - - - - - - - - - \n");

            //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
}

//Function OutputToFile
//Writes header containing information about the similuation
//and contents of three vectors to file

/*
 * Write data to a .csv file
 *
 * Writes detailed parameter information and meta data to a csv
 * whose file name is specified by OUTPUT_FILENAME
 *
 *
 */
#ifdef SPEC_CHNG

void OutputToFile ( double XX[], double XY[], double YY[],
                    double TimeTrack[], double time_spent, int count,
                    char ProgName[],
                    double ActiveRatio[], double DangleRatio[],
                    double LoopedRatio[],
                    SpeciesValue AvgLen[], SpeciesValue Variance[],
                    int NumOfBins, SpeciesCount **Hist,
                    Stress Time_k_Stress[], Stress Active_Stress[],
                    Stress Dangle_Stress[],
                    double AvgSpringLife[],
                    unsigned int Dng2Act[], unsigned int Dng2Lpd[],
                    unsigned int Act2Dng[], unsigned int Lpd2Dng[],
                    char OutputFileName[]){

#else

        // Function Description: output results to .CSV file
void OutputToFile ( double XX[], double XY[], double YY[],
                    double TimeTrack[], double time_spent, int count,
                    char ProgName[],
                    double ActiveRatio[], double DangleRatio[],
                    double LoopedRatio[],
                    SpeciesValue AvgLen[], SpeciesValue Variance[],
                    int NumOfBins, SpeciesCount **Hist,
                    Stress Time_k_Stress[], Stress Active_Stress[],
                    Stress Dangle_Stress[],
                    double AvgSpringLife[],
                    char OutputFileName[]){

#endif


      FILE *OutputFile = NULL;

      sprintf(OutputFileName, "%s.csv", OutputFileName); //<---Filename

      OutputFile = fopen(OutputFileName, "w+"); //w+ to overwrite file

      if (OutputFile == NULL){
            fprintf(stderr, "Couldn't open output file: %s!\n", OutputFileName);
            exit(1);
      }

      // _____ Header for textfile ______________________________
      //Descrption

      fprintf(OutputFile,
    "********************************************************",
    "**************\n");
      fprintf(OutputFile,"* %60s *\n",ProgName);
      fprintf(OutputFile,"* Header - 8 lines, 1 thru 8,",
                                    "Parameters+2 − 25 lines, 9 thru 33 *\n");
            fprintf(OutputFile,"* Data header − 3 lines, 34 thru 36,",
                              " Stress Data − lines, 37+ *\n");

    /*
     * List preprocessor options so that it is clear in output.
     */

    fprintf(OutputFile, " Preprocessor Options: ");
#ifdef SIMPLE_SHEAR
    fprintf(OutputFile, "SIMPLE_SHEAR, ");
#else
    fprintf(OutputFile, "OSCILLATORY_SHEAR, ");
#endif

#ifdef LOOPED_DUMBBELLS
    fprintf(OutputFile, "LOOPED_DUMBBELLS, ");
#else
    fprintf(OutputFile, "ACTIVE_AND_DANGLING_ONLY, ");
#endif

#ifdef NEW_TAU
    fprintf(OutputFile, "NEW_TAU, ");
#endif

#ifdef RAW_OUT
    fprintf(OutputFile, "RAW_OUT, ");
#endif

```

```
#ifdef DEBUG
    fprintf(OutputFile, "DEBUG, ");
#endif

#ifdef SPEC_CHNG
    fprintf(OutputFile, "SPEC_CHNG, ");
#endif

#ifdef SINGLE_MICRO
    fprintf(OutputFile, "SINGLE_MICRO: ID(unit):%d Type(int):%lu",
                    " Length(double):%lu ",
                    SINGLE_MICRO, sizeof(int), sizeof(double));
#endif
#ifdef MICRO_RAW
    fprintf(OutputFile, "MICRO_RAW: ID(unit):%lu Type(int):%lu",
                    " Length(double):%lu ",
                    sizeof(unsigned int), sizeof(int), sizeof(double));
#endif


#ifdef FIXED_SEED
    fprintf(OutputFile, "FIXED_SEED, ");
#endif

#ifdef SINGLE_MICRO
    fprintf(OutputFile, "SINGLE_MICRO, ");
#endif

#ifdef MICRO_RAW
```

```
    fprintf(OutputFile, "MICRO_RAW, ");
#endif

#ifdef COND_PROB_METHOD
    fprintf(OutputFile, "COND_PROB_METHOD, ");
#endif

#ifdef CHK_DNG
    fprintf(OutputFile, "CHK_DNG, ");
#endif

#ifdef LEN_CHG
    fprintf(OutputFile, "LEN_CHG, ");
#endif

#ifdef PROB_TEST
    fprintf(OutputFile, "PROB_TEST, ");
#endif

#ifdef SINGLE_TRACK
    fprintf(OutputFile, "SINGLE_TRACK, ");
#endif

#ifdef LOOP_PROB_TWO
    fprintf(OutputFile, "LOOP_PROB_TWO, ");
#endif

#ifdef LOOP_PROB_THREE
    fprintf(OutputFile, "LOOP_PROB_THREE, ");
```

```
#endif

#ifdef LOOP_PROB_FOUR
    fprintf(OutputFile, "LOOP_PROB_FOUR, ");
#endif

#ifdef NO_REPORT
    fprintf(OutputFile, "NO_REPORT: %llu,", hostA_coeff);
#endif

#ifdef SKEW_START
    fprintf(OutputFile, "SKEW_START, ");
#endif

#ifdef LEN_CHN_NORM
    fprintf(OutputFile, "LEN_CHN_NORM, ");
#endif

#ifdef LOOP_FREEZE
    fprintf(OutputFile, "LOOP_FREEZE, ");
#endif

#ifdef FULL_DATA
    fprintf(OutputFile, "FULL_DATA");
#endif

    fprintf(OutputFile, "\n");


```

```
#ifdef SIMPLE_SHEAR
        fprintf(OutputFile,"* Simple Shear Flow ",
                  " *\n");
#else
        fprintf(OutputFile,"* Small Oscillatory Shear Flow ",
                  " *\n");
#endif
#ifdef LOOPED_DUMBBELLS
        fprintf(OutputFile,"* Dumbbell Types: Active, Dangling and Looped ",
                  " *\n");
#else
        fprintf(OutputFile,"* Dumbbell Types: Active, and Dangling ",
    " *\n");
#endif

#ifdef RAW_OUT
        fprintf(OutputFile,"*************** Has Raw Output Bin File",
                  " **********************************\n");
#else
        fprintf(OutputFile,"******************************************",
                  "**********************************\n");
#endif


#ifdef NO_REPORT
        fprintf(OutputFile,"Total_Time: %g \n",
          hostStepSizeMicroFirst * (hostMacroStepSizeSplitPt − 1) *
          hostTimeStepsMicro + //Stage 1
          hostStepSizeMicroSecon * hostA_coeff + //Macro Jump
          hostStepSizeMicroSecon *
          (hostTimeStepsMacro − hostMacroStepSizeSplitPt )
          * hostTimeStepsMicro ); //Stage 2
#else
        fprintf(OutputFile,"Total_Time: %g \n", (hostStepSizeMicroFirst *
          hostMacroStepSizeSplitPt + hostStepSizeMicroSecon *
          (hostTimeStepsMacro − hostMacroStepSizeSplitPt)) *
                  hostTimeStepsMicro );
#endif
        fprintf(OutputFile,"Total_Number_of_Macro_Steps: %u\n",
                  hostTimeStepsMacro);
        fprintf(OutputFile,"Micro_Steps_Per_Macro_Iteration: %u\n",
                  hostTimeStepsMicro);
        fprintf(OutputFile,"Macro_Step_Size_Split_Point: %u\n",
                  hostMacroStepSizeSplitPt);
        fprintf(OutputFile,"Micro_Step_Size_One: %1.12g\n",
                  hostStepSizeMicroFirst);
        fprintf(OutputFile,"Macro_Step_Size_One: %1.12g\n",
                  hostStepSizeMicroFirst * hostTimeStepsMicro);
        fprintf(OutputFile,"Number_of_Macro_Steps_Stage_One: %u\n",
                  hostMacroStepSizeSplitPt);
        fprintf(OutputFile,"Stage_One_Total_Time: %1.12g\n",
          hostStepSizeMicroFirst * hostTimeStepsMicro *
          hostMacroStepSizeSplitPt);
        fprintf(OutputFile,"Micro_Step_Size_Two: %1.12g\n",
                  hostStepSizeMicroSecon);
        fprintf(OutputFile,"Macro_Step_Size_Two: %1.12g\n",
                  hostStepSizeMicroSecon * hostTimeStepsMicro);
        fprintf(OutputFile,"Number_of_Macro_Steps_Stage_Two: %u\n",
                  hostTimeStepsMacro − hostMacroStepSizeSplitPt);
        fprintf(OutputFile,"Stage_Two_Total_Time: %1.12g\n",
                  hostStepSizeMicroSecon ∗ hostTimeStepsMicro ∗
                  (hostTimeStepsMacro − hostMacroStepSizeSplitPt));
        fprintf(OutputFile,"Number_of_Particles: %u\n", hostNumberOfParticles);
        fprintf(OutputFile,"Flow_Rate: %g \n", hostFlowRate);
        fprintf(OutputFile,"Maximum_Spring_Length: %g\n", hostMaxSpringLength );
        fprintf(OutputFile,"SAOS_Frequency: %g\n", hostFreq );
        fprintf(OutputFile,"d: %g\n", hostD_free );
        fprintf(OutputFile,"Z: %g\n", hostZee );
        fprintf(OutputFile,"Chi: %g\n", hostChi);
        fprintf(OutputFile,"Alpha: %g\n", hostAlpha);
        fprintf(OutputFile,"Beta: %g\n", hostBeta);
        fprintf(OutputFile,"Initial_Active_Ratio: %g\n", Init_Active_Ratio);
        fprintf(OutputFile,"Initial_Dangling_Ratio: %g\n", Init_Dangle_Ratio);
        fprintf(OutputFile,"Initial_Looped_Ratio: %g\n",
                  1−Init_Dangle_Ratio−Init_Active_Ratio);
#ifdef NO_REPORT
        fprintf(OutputFile,"Macro_Step_Jump: %llu\n", hostA_coeff);
#else
        fprintf(OutputFile,"A_Coefficient: %g\n", hostA_coeff);
#endif
        fprintf(OutputFile,"B_Coefficient: %g\n", hostB_coeff);
        fprintf(OutputFile,"Runtime: %g\n", time_spent);


        //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
```

```
    /*Nicer table output */

        fprintf(OutputFile," %20s," , "Time");
        fprintf(OutputFile," %20s," , "StressXX");
        fprintf(OutputFile," %20s," , "StressXY");
        fprintf(OutputFile," %20s," , "StressYY");
        fprintf(OutputFile," %20s," , "Active_Stress_XX");
        fprintf(OutputFile," %20s," , "Active_Stress_XY");
        fprintf(OutputFile," %20s," , "Active_Stress_YY");
        fprintf(OutputFile," %20s," , "Dangle_Stress_XX");
        fprintf(OutputFile," %20s," , "Dangle_Stress_XY");
        fprintf(OutputFile," %20s," , "Dangle_Stress_YY");
        fprintf(OutputFile," %20s," , "ActiveRatio");
        fprintf(OutputFile," %20s," , "DangleRatio");
        fprintf(OutputFile," %20s," , "LoopedRatio");
        fprintf(OutputFile," %20s," , "AvgLen.ActiveLen");
        fprintf(OutputFile," %20s," , "AvgLen.ActiveAng");
        fprintf(OutputFile," %20s," , "AvgLen.ActiveX");
        fprintf(OutputFile," %20s," , "AvgLen.ActiveY");
        fprintf(OutputFile," %20s," , "AvgLen.DangleLen");
        fprintf(OutputFile," %20s," , "AvgLen.DangleAng");
        fprintf(OutputFile," %20s," , "AvgLen.DangleX");
        fprintf(OutputFile," %20s," , "AvgLen.DangleY");
        fprintf(OutputFile," %20s," , "AvgLen.LoopedLen");
        fprintf(OutputFile," %20s," , "AvgLen.LoopedAng");
        fprintf(OutputFile," %20s," , "AvgLen.LoopedX");
        fprintf(OutputFile," %20s," , "AvgLen.LoopedY");
        fprintf(OutputFile," %20s," , "Variance.ActiveLen");
        fprintf(OutputFile," %20s," , "Variance.ActiveAng");
        fprintf(OutputFile," %20s," , "Variance.ActiveX");
        fprintf(OutputFile," %20s," , "Variance.ActiveY");
        fprintf(OutputFile," %20s," , "Variance.DangleLen");
        fprintf(OutputFile," %20s," , "Variance.DangleAng");
        fprintf(OutputFile," %20s," , "Variance.DangleX");
        fprintf(OutputFile," %20s," , "Variance.DangleY");
        fprintf(OutputFile," %20s," , "Variance.LoopedLen");
        fprintf(OutputFile," %20s," , "Variance.LoopedAng");
        fprintf(OutputFile," %20s," , "Variance.LoopedX");
        fprintf(OutputFile," %20s," , "Variance.LoopedY");
        fprintf(OutputFile," %20s," , "AvgSpringLife");

#ifdef SPEC_CHNG
        fprintf(OutputFile," %20s," , "Dng2Act");
        fprintf(OutputFile," %20s," , "Dng2Lpd");
        fprintf(OutputFile," %20s," , "Act2Dng");
        fprintf(OutputFile," %20s," , "Lpd2Dng");
#endif

        fprintf(OutputFile," %50s" , "Histogram Bins: Active, Dangle, Looped - ");
        fprintf(OutputFile,"%d each\n", NumOfBins);




    /* write data to file */

        unsigned int k;
        for (k=0; k<count; k++){
```

```
fprintf(OutputFile," % 20.6f," , TimeTrack[k]);
fprintf(OutputFile," % 20.10f," , XX[k]);
fprintf(OutputFile," % 20.10f," , XY[k]);
fprintf(OutputFile," % 20.10f," , YY[k]);
fprintf(OutputFile," % 20.10f," , Active_Stress[k].XX);
fprintf(OutputFile," % 20.10f," , Active_Stress[k].XY);
fprintf(OutputFile," % 20.10f," , Active_Stress[k].YY);
fprintf(OutputFile," % 20.10f," , Dangle_Stress[k].XX);
fprintf(OutputFile," % 20.10f," , Dangle_Stress[k].XY);
fprintf(OutputFile," % 20.10f," , Dangle_Stress[k].YY);
fprintf(OutputFile," % 20.6f," , ActiveRatio[k]);
fprintf(OutputFile," % 20.6f," , DangleRatio[k]);
fprintf(OutputFile," % 20.6f," , LoopedRatio[k]);
fprintf(OutputFile," % 20.6f," , AvgLen[k].ActiveLen);
fprintf(OutputFile," % 20.6f," , AvgLen[k].ActiveAng);
fprintf(OutputFile," % 20.6f," , AvgLen[k].ActiveX);
fprintf(OutputFile," % 20.6f," , AvgLen[k].ActiveY);
fprintf(OutputFile," % 20.6f," , AvgLen[k].DangleLen);
fprintf(OutputFile," % 20.6f," , AvgLen[k].DangleAng);
fprintf(OutputFile," % 20.6f," , AvgLen[k].DangleX);
fprintf(OutputFile," % 20.6f," , AvgLen[k].DangleY);
fprintf(OutputFile," % 20.6f," , AvgLen[k].LoopedLen);
fprintf(OutputFile," % 20.6f," , AvgLen[k].LoopedAng);
fprintf(OutputFile," % 20.6f," , AvgLen[k].LoopedX);
fprintf(OutputFile," % 20.6f," , AvgLen[k].LoopedY);
fprintf(OutputFile," % 20.6f," , Variance[k].ActiveLen);
fprintf(OutputFile," % 20.6f," , Variance[k].ActiveAng);
fprintf(OutputFile," % 20.6f," , Variance[k].ActiveX);
fprintf(OutputFile," % 20.6f," , Variance[k].ActiveY);
```

```
                    fprintf(OutputFile," % 20.6f," , Variance[k].DangleLen);
                    fprintf(OutputFile," % 20.6f," , Variance[k].DangleAng);
                    fprintf(OutputFile," % 20.6f," , Variance[k].DangleX);
                    fprintf(OutputFile," % 20.6f," , Variance[k].DangleY);
                    fprintf(OutputFile," % 20.6f," , Variance[k].LoopedLen);
                    fprintf(OutputFile," % 20.6f," , Variance[k].LoopedAng);
                    fprintf(OutputFile," % 20.6f," , Variance[k].LoopedX);
                    fprintf(OutputFile," % 20.6f," , Variance[k].LoopedY);
     fprintf(OutputFile," % 20.6f," , AvgSpringLife[k]);

#ifdef SPEC_CHNG
                    fprintf(OutputFile," %20u," , Dng2Act[k]);
                    fprintf(OutputFile," %20u," , Dng2Lpd[k]);
                    fprintf(OutputFile," %20u," , Act2Dng[k]);
                    fprintf(OutputFile," %20u," , Lpd2Dng[k]);
#endif


          for(unsigned int n=0; n<NumOfBins; n++){
               fprintf(OutputFile," % 12d,", Hist[n][k].Active);
          }
          for(unsigned int n=0; n<NumOfBins; n++){
               fprintf(OutputFile," % 12d,", Hist[n][k].Dangle);
          }
          for(unsigned int n=0; n<NumOfBins-1; n++){
               fprintf(OutputFile," % 12d,", Hist[n][k].Looped);
          }

          fprintf(OutputFile, " % 12d\n", Hist[NumOfBins-1][k].Looped);
```

```



            }
            //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''

            fclose(OutputFile);

}

/*
 * No longer being used
 * CPU Function: Raw Data
 *
 * Output all x, y and species data for each dumbbell and write it to
 * a second raw data csv file.
 *
 *
 */

void WriteRawDataFile(double TimeTrack[], Dumbbell *RawDBellData[])
{
      FILE *RawOutput = NULL;
      char RawDataFilename[] = RAWDATA_FILENAME;


      RawOutput = fopen(RawDataFilename, "w");

      if (RawOutput == NULL)
      {
        fprintf(stderr, "Could not open output file: %s!\n", RawDataFilename);
        exit(1);
    }

    for(unsigned int k=0; k<hostTimeStepsMacro+1; k++)
    {
        fprintf(RawOutput, "%9.6f, ", TimeTrack[k]);
        for(unsigned int j=0; j<hostNumberOfParticles; j++)
            fprintf(RawOutput, "% .2d, % 20.14f, % 20.14f, ",
                            RawDBellData[k][j].type,
                            RawDBellData[k][j].x,
                            RawDBellData[k][j].y);
        fprintf(RawOutput, "\n");
    }



    fclose(RawOutput);
}




//Function:
//GPU Function
//Caclulates the change of state probability of an active dumbbell
//given the spring length
//Tao must be commputed each time: See paper, use equations 10 AND 11.
__device__ double ActiveToDanglingProb (double SpringLen, double MicroStepSize){


    /*
     * These probabilities are based on the reasoning in notebook:
     * Transition Probability Physical Argugments
     *
     */
#ifdef PROB_TEST
    //transition probabilities held constant
    return devBeta;

#else
    //Normal transition probabilities


     double F_fene = SpringLen /
       ( 1 - SpringLen * SpringLen / devMaxSpringLength / devMaxSpringLength );


     return 1 - exp( - devBeta * exp( 0.0325 * abs( F_fene )) * MicroStepSize);

#endif
}

//Function:
//GPU Function
//Cacluates the change of state probability for a dangling dumbbell.
__device__ double DanglingToActiveProb (double SpringLen, double MicroStepSize){

```

```

#ifdef PROB_TEST
      //transition probabilities held constant
      return devAlpha;

#else
      //Normal transition probabilities

      double F_fene = SpringLen /
         ( 1 - SpringLen * SpringLen / devMaxSpringLength / devMaxSpringLength );

            return 1.0 - exp( - devAlpha * SpringLen * F_fene * MicroStepSize );
#endif
}

//Function:
//GPU Function
//Calculates the probability of an chain going active becoming
// a loop instead of a bridge
__device__ double DanglingToLoopedProb (double SpringLen,
                                                            double MicroStepSize){

#ifdef LOOP_PROB_TWO
      double F_fene_two = SpringLen /
         ( 1 - (devMaxSpringLength - SpringLen) *
            (devMaxSpringLength - SpringLen) /
            devMaxSpringLength / devMaxSpringLength );

      return 1 - exp( - devChi * (devMaxSpringLength - SpringLen)*
      (devMaxSpringLength - SpringLen) / F_fene_two * MicroStepSize );
#else
#ifdef LOOP_PROB_THREE
    double F_fene_two = 1 / ( 1 - (devMaxSpringLength - SpringLen) *
      (devMaxSpringLength - SpringLen) /
      devMaxSpringLength / devMaxSpringLength );

    return 1 - exp( - devChi * (devMaxSpringLength - SpringLen) *
      (devMaxSpringLength - SpringLen) * F_fene_two * MicroStepSize );
#else
#ifdef LOOP_PROB_FOUR
    double F_fene_two = 1 / ( 1 - (devMaxSpringLength - SpringLen) *
      (devMaxSpringLength - SpringLen) /
      devMaxSpringLength / devMaxSpringLength );

    return 1 - exp( - devChi * SpringLen *
             (devMaxSpringLength - SpringLen) *
             (devMaxSpringLength - SpringLen) * F_fene_two * MicroStepSize );
#else
    return exp( - SpringLen * SpringLen / devChi * MicroStepSize );
#endif
#endif
#endif



}

//Function:
```

```
//GPU Function
//Calculates the probability looped chain becoming a dangling chain
__device__ double LoopedToDanglingProb (double SpringLen,
                                        double MicroStepSize){

    /*
     * These probabilities are based on the reasoning in notebook:
     * Proposed Transition Probabilities One
     *
     * This probability doesn't depend on the length of the dumbbell
     * because the behavior of looped dumbbells are not modeled
     * [Although there are still some numbers in the simulation].
     *
     */

    return 1.0 - exp( - devBeta * MicroStepSize );
}

//Function:
//GPU Function
//Calculate the conditional probability of becoming a loop given
//that a dangling dumbbell does not become active.
__device__ double DangleNotActToLoop (double SpringLen,
                double MicroStepSize, int dbell, double time){

    double F_fene = SpringLen / ( 1 - SpringLen * SpringLen /
                    devMaxSpringLength / devMaxSpringLength );


    double cond_prob = exp( - SpringLen * SpringLen *
                    ( 1 / devChi - devAlpha / F_fene ) * MicroStepSize );


    if (cond_prob <= 1.0){
        return cond_prob;
    } else {
        return 1.0;
    }


}


__device__ double DanglingToAttachProb(double SpringLen, double MicroStepSize){

      double AttachProb;
      AttachProb = DanglingToActiveProb(SpringLen, MicroStepSize) +
                    DanglingToLoopedProb(SpringLen,MicroStepSize);

      if (AttachProb > 1.0) {
            return 1.0;
      } else {
            return AttachProb;
      }

}


```

```

//Function: AttachedLoop
__device__ double AttachedToLoopedProb(double SpringLen, double MicroStepSize){

    return DanglingToLoopedProb( SpringLen, MicroStepSize) /
         ( DanglingToLoopedProb( SpringLen, MicroStepSize) +
           DanglingToActiveProb( SpringLen, MicroStepSize) );

}


//Function: AttachedLoop
__device__ double AttachedToActiveProb(double SpringLen,
                                       double MicroStepSize){

    return DanglingToActiveProb( SpringLen, MicroStepSize) /
         ( DanglingToLoopedProb( SpringLen, MicroStepSize) +
           DanglingToActiveProb( SpringLen, MicroStepSize) );

}


//Function: Evolve
//GPU Function
// Describes how length evolves over the specified time step size
__device__ void Evolve(double *SpringLenX, double *SpringLenY, double randx,
                double randy, double drag_coeff, double *SimTime,
                                       double MicroStepSize, double FlowRate){

```

```
        double SpringLenXStep, SpringLenYStep;

        //___ Intermediate step variables
        double SpringLenXOne, SpringLenYOne;
        double ItermValueOne, ItermValueTwo;
        double LengthLimitingFactor;
        //''''''''''''''''''''''''''''''''''''''''''


//_____ Non−Dim Evolution Equations for dangling FENE dumbbells _____________



#ifdef SIMPLE_SHEAR //Uses a compiler flag to switch to simple_shear

    /* Simple Shear with Variable Flow Rate */

        SpringLenXOne = *SpringLenX
      + (U11 * *SpringLenX + U21 * *SpringLenY) * FlowRate * MicroStepSize
      + sqrt( drag_coeff * MicroStepSize) * randx;
#else

    /* Small Amplitude Oscillatory Shear Flow
     * with Variable Flow Rate (Allows for equilibrium period)
     */

        SpringLenXOne = *SpringLenX
                + (U11 * *SpringLenX + U21 * devFreq * cos(devFreq * *SimTime) *
                    *SpringLenY) * FlowRate * MicroStepSize
```

```
                        + sqrt( drag_coeff * MicroStepSize) * randx;

#endif

    /* note: flow in the y-direction is uneffected */

    SpringLenYOne = *SpringLenY
                        + (U12 * *SpringLenX + U22 * *SpringLenY)
        * FlowRate * MicroStepSize
                        + sqrt( drag_coeff * MicroStepSize) * randy;


        LengthLimitingFactor = ( SpringLenXOne * SpringLenXOne +
                                    SpringLenYOne * SpringLenYOne )
                                     / devMaxSpringLength / devMaxSpringLength;

        ItermValueOne = 1.0 + 2.0 * drag_coeff * MicroStepSize +
            LengthLimitingFactor;

        ItermValueTwo = 2 / ( ItermValueOne +
    sqrt( ItermValueOne * ItermValueOne - 4 * LengthLimitingFactor) );

        SpringLenXStep = sqrt ( ItermValueTwo ) * SpringLenXOne;

        SpringLenYStep = sqrt ( ItermValueTwo ) * SpringLenYOne;

        //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
```

```
            *SpringLenX = SpringLenXStep;
            *SpringLenY = SpringLenYStep;

}

//Function: Micro_Steps
//Loops through the Micro loop of the SDE
#ifdef SPEC_CHNG

__global__ void Micro_Steps( double *SpringLenX, double *SpringLenY,
            int *SpeciesType,
                  curandState *states, curandState *ProbStates,
                  double AvgSpringLifes, double *SimTime, double MicroStepSize,
            unsigned int TimeStepsMicro,
            double DangleAvgLen, double FlowRate,
            unsigned int *Dng2Act, unsigned int *Dng2Lpd,
            unsigned int *Act2Dng, unsigned int *Lpd2Dng){

#else
#ifdef MICRO_RAW

__global__ void Micro_Steps( double *SpringLenX, double *SpringLenY,
            int *SpeciesType,
            curandState *states, curandState *ProbStates,
            double AvgSpringLifes, double *SimTime, double MicroStepSize,
            unsigned int TimeStepsMicro,
            double DangleAvgLen, double FlowRate,
            DBSpecChng *SCArr, unsigned int NumberOfParticles){
#else
```

```
#ifdef NO_REPORT
__global__ void Micro_Steps( double *SpringLenX, double *SpringLenY,
            int *SpeciesType,
            curandState *states, curandState *ProbStates,
            double AvgSpringLifes, double *SimTime, double MicroStepSize,
            unsigned long long TimeStepsMicro,
            double DangleAvgLen, double FlowRate){

#else
#ifdef SINGLE_MICRO

__global__ void Micro_Steps( double *SpringLenX, double *SpringLenY,
            int *SpeciesType,
            curandState *states, curandState *ProbStates,
                  double AvgSpringLifes, double *SimTime, double MicroStepSize,
            unsigned int TimeStepsMicro,
            double DangleAvgLen, double FlowRate,
            DBSpecChng *SCArr, unsigned int NumberOfParticles){
#else
__global__ void Micro_Steps( double *SpringLenX, double *SpringLenY,
            int *SpeciesType,
            curandState *states, curandState *ProbStates,
            double AvgSpringLifes, double *SimTime, double MicroStepSize,
            unsigned int TimeStepsMicro,
            double DangleAvgLen, double FlowRate){
#endif //SINGLE_MICRO
#endif //NO_REPORT
#endif //MICRO_RAW
#endif //SPEC_CHNG
```

```
        int i = threadIdx.x + blockIdx.x * blockDim.x;


        //____Device API for Random Number Generation_____
        //copy state to local state for efficiency
        curandState localState = states[i];
        curandState localProbState = ProbStates[i];


    //____ Calculation values and constants



#ifdef SPEC_CHNG
  // set counters to 0
  Dng2Act[i]=0;
  Dng2Lpd[i]=0;
  Act2Dng[i]=0;
  Lpd2Dng[i]=0;
#endif


#ifdef MICRO_RAW
```

```
    /* printf("Thread:%d Precur:%d Time:%f Length:%f \n", i, */
    /* SCArr[i*TimeStepsMicro].type, */
    /* SCArr[i*TimeStepsMicro].time, */
    /* SCArr[i*TimeStepsMicro].length); */
#endif

        //____ Node drag value calculations ____

        double D_node = 0.5 * devZee * 6.0 * devD_free;
    //Equation (25) - non-dimensional

        double drag_coeff_active = devD_free / (2 * D_node);
    //Nondimensionalized

        double drag_coeff_dangle = (D_node + devD_free) / 4.0 / D_node;


        //''''''''''''''''''''''''''''''''''''''''''''''''''''''''



    //''''''''''''''''''''''''''''''''''''''''''''''''''''''''


        double2 RandNorm;
        double RandUniform;

#ifdef LOOPED_DUMBBELLS

```

```
            double RandUniform2;


#endif

#ifdef DEBUG
    //printf("TimeStepsMicro = %d\n", TimeStepsMicro);
#endif

            double SpringLen;
            for(unsigned long long j=0; j < TimeStepsMicro; j++){
    //changed to unsigned long long for large loop cnts





#ifdef SINGLE_TRACK
    if (i==0){
        printf("SingleTrack:%f,%d:%f:%f\n",SimTime[i],SpeciesType[i],
                        SpringLenX[i],SpringLenY[i]);
    }
#endif



                //generate new random number each time
                RandNorm = curand_normal2_double(&localState);
                RandUniform = curand_uniform_double(&localProbState);
```

```

#ifdef LOOPED_DUMBBELLS
    RandUniform2 = curand_uniform_double(&localProbState);


#endif

#ifdef LEN_CHG
  /*
    * Added to check if looped re-entry is responsible for phenomena
    * Reinserts loops as randomly angled short dumbbells.
    */
    double RandUniform3;
    double RandUniform4;

    RandUniform3 = curand_uniform_double(&localProbState);
    RandUniform4 = curand_uniform_double(&localProbState);
#endif


#ifdef LEN_CHN_NORM
  /*
    * Reinserts loops with length and orientation from a Gaussian dist.
    * Careful using this a short maximum dumbbell length.
    */
  double2 RandNorm2;
  RandNorm2 = curand_normal2_double(&localState);
#endif

```

```



                        //Calculate Spring Length
                        SpringLen = sqrt(SpringLenX[i] * SpringLenX[i] +
                             SpringLenY[i] * SpringLenY[i]);

#ifdef SINGLE_MICRO
        //record type, time, and length of each step
        if (i==SINGLE_MICRO){
          SCArr[j].type = SpeciesType[i];
          SCArr[j].time = SimTime[i];
          SCArr[j].x = SpringLenX[i];
          SCArr[j].y = SpringLenY[i];
        }
#endif

#ifdef MICRO_RAW
        //record type, time, and length of each step
        SCArr[i*TimeStepsMicro+j].type = SpeciesType[i];
        SCArr[i*TimeStepsMicro+j].length = SpringLen;
#endif



#ifdef LOOPED_DUMBBELLS
             switch(SpeciesType[i]) {

                case 0: //Active Type
```

```
                    Evolve(&SpringLenX[i], &SpringLenY[i],
               RandNorm.x, RandNorm.y,
               drag_coeff_active, &SimTime[i],
               MicroStepSize, FlowRate);

        if (ActiveToDanglingProb(SpringLen, MicroStepSize) >
                              RandUniform){
        // if threshold prob is higher than uniform rand number then..
          SpeciesType[i] = 1; //Change dumbbell to Dangling species
#ifdef SPEC_CHNG
          Act2Dng[i]++;
#endif
        }

      break;

      case 1: //Dangling Type

                    Evolve(&SpringLenX[i], &SpringLenY[i],
               RandNorm.x, RandNorm.y,
               drag_coeff_dangle, &SimTime[i],
               MicroStepSize, FlowRate);


#ifdef COND_PROB_METHOD


        if (DanglingToAttachProb(SpringLen, MicroStepSize) >
                          RandUniform){
```

```
                        //Dumbbell will become active or looped
                if (AttachedToActiveProb(SpringLen, MicroStepSize) >
                                RandUniform2){
                        SpeciesType[i] = 0; //Change to Active Dumbbell
                } else {
                        SpeciesType[i] = 2; //Change to Looped Dumbbell
                }
            }


#else
            if ((DanglingToLoopedProb(SpringLen, MicroStepSize) >
                        RandUniform)
            && (DanglingToActiveProb(SpringLen, MicroStepSize) <
                        RandUniform2)){

              SpeciesType[i] = 2; //Change to looped type

            } else if ((DanglingToLoopedProb(SpringLen, MicroStepSize) <
               RandUniform) && (DanglingToActiveProb(SpringLen, MicroStepSize)
                       > RandUniform2)){

              SpeciesType[i] = 0; //Change to Active species

            } //In all other cases dumbbells remain dangling
#endif



```

```
                    break;

#ifdef LOOP_FREEZE
#else //LOOP_FREEZE
                case 2: //Loooped Type

                    if ( LoopedToDanglingProb(SpringLen, MicroStepSize) >
                                RandUniform){





#ifdef LEN_CHG
                        SpringLenX[i] = (RandUniform3 - 0.5) * 2;
                        SpringLenY[i] = (RandUniform4 - 0.5) * 2;
#endif //LEN_CHG

#ifdef LEN_CHN_NORM

   /*
   * Note: There is no failsafe for this loop and it could continue infinitely.
   */

                        do {
                            SpringLenX[i] = RandNorm2.x;
                            SpringLenY[i] = RandNorm2.y;
                        } while ( SpringLenX[i] * SpringLenX[i] +
                                    SpringLenY[i] * SpringLenY[i] >=
                                devMaxSpringLength * devMaxSpringLength );

#endif //LEN_CHN_NORM

#ifdef CHK_DNG
                    printf("New DangleDumbbell[%d] X-len: %f",
                      " Y-Len: %f Time: %f\n", i, SpringLenX[i], SpringLenY[i],
                      SimTime[i]);
#endif //CHK_DNG
                    SpeciesType[i] = 1; //Change to dangling type
#ifdef SPEC_CHNG
                    Lpd2Dng[i]++;
#endif //SPEC_CHNG
                }
                break;
#endif //LOOP_FREEZE
        }
#else

        //Default to two dumbbell types

        switch(SpeciesType[i]) {

          case 0: //Active Type
            Evolve(&SpringLenX[i], &SpringLenY[i], RandNorm.x, RandNorm.y,
            drag_coeff_active, &SimTime[i], MicroStepSize, FlowRate);

            if (ActiveToDanglingProb(SpringLen, MicroStepSize)
                                > RandUniform){
```

```
                // if threshold prob is higher than uniform rand number then
                SpeciesType[i] = 1; //Change dumbbell to Dangling species
#ifdef SPEC_CHNG
                Act2Dng[i]++;
#endif
              }

              break;

              case 1: //Dangling Type
                            Evolve(&SpringLenX[i], &SpringLenY[i],
                         RandNorm.x, RandNorm.y,
                         drag_coeff_dangle, &SimTime[i],
                         MicroStepSize, FlowRate);

              if (DanglingToActiveProb(SpringLen, MicroStepSize) > RandUniform){
                // if dangling prob is higher than uniform rand number then
                SpeciesType[i] = 0; //Change dumbbell to Active species
#ifdef SPEC_CHNG
                Dng2Act[i]++;
#endif
              }

              break;
            }

#endif

                    //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''

                    SimTime[i] += MicroStepSize;



                    //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''


        }

        //copy random number generator state back
        states[i] = localState;
        ProbStates[i] = localProbState;
}

//Function: RandomGenInit
//Initialize the random number generator on each of the threads
//Gives each thread a different seed form *SeedList vector
__global__ void RandomGenInit(unsigned long long *SeedList,
                                        curandState *states){

        int tid = blockIdx.x * blockDim.x + threadIdx.x;

        //curand_init(SeedList[tid], tid, 0, &states[tid]);
  //previous method seems to be quite slow.

        curand_init((unsigned long long)SeedList[tid], 0, 0, &states[tid]);

}
```

```
//Function: RndNorm
//CPU Function to transform uniform random variable [0,1] to normal random
//variable with meand 0 and Variance defined in the function
double RndNorm (void){
        double Variance = 1;

        static int HasSpareRandomNum = 0;
        static double SpareRandomNum;

        if(HasSpareRandomNum == 1){
                HasSpareRandomNum = 0;
                return Variance * SpareRandomNum;
        }

        HasSpareRandomNum = 1;

        static double u,v,s;

        do{
                u = ( rand() / ((double) RAND_MAX)) * 2 - 1;
                v = ( rand() / ((double) RAND_MAX)) * 2 - 1;
                s = u * u + v * v;
        } while (s >= 1 || s == 0);

        s = sqrt (-2.0 * log(s) / s);

    SpareRandomNum = v * s; //Save spare random number for next function call
```

```

        return Variance * u * s;
}



double AvgSpringLife ( double *SpringLenX, double *SpringLenY,
                       int *SpeciesType){

//TODO: decide on method and clean up debug output
#ifdef NEW_TAU
    return 6.0;
#else

#ifdef DEBUG
        double Total1 = 0.0;
#endif
    double Total2 = 0.0;

        double SpringLen;
        int ActiveCount = 0;

        for (unsigned int j=0; j<hostNumberOfParticles; j++){

                if (SpeciesType[j] == 0){ //If active type

                ActiveCount++;
                SpringLen = sqrt( SpringLenX[j] * SpringLenX[j] +
                      SpringLenY[j] * SpringLenY[j]);
```

```

                        //___Hookean Springs___
                        //Total += Tao_zero * exp (- LITTLE_D * LITTLE_D *
        // SpringLen * SpringLen / U_ZERO );
                        //''''''''''''''''''

                        //___FENE Springs_____


#ifdef DEBUG

                /*
                 * If in debug mode, compute both and compare.
                 */

        Total1 += 2 / hostBeta * exp ( - 0.0325 * abs( SpringLen /
          ( 1 - SpringLen * SpringLen /
            hostMaxSpringLength / hostMaxSpringLength ) ));

                        Total2 += TAO_FUND * hostA_coeff *
          exp ( - ( hostB_coeff * SpringLen * SpringLen ) /
                                (( 1 - ( SpringLen / hostMaxSpringLength) *
                ( SpringLen / hostMaxSpringLength) ) *
                                 ( 1 - ( SpringLen / hostMaxSpringLength) *
                  ( SpringLen / hostMaxSpringLength) ) )
                            ); //Eqn (12)
#else

              //Hernandez-Cifre Tau Calculation
```

```
                    Total2 += TAO_FUND * hostA_coeff *
                            exp ( − ( hostB_coeff * SpringLen * SpringLen ) /
                                    (( 1 − ( SpringLen / hostMaxSpringLength) *
              ( SpringLen / hostMaxSpringLength) ) *
                                     ( 1 − ( SpringLen / hostMaxSpringLength) *
                   ( SpringLen / hostMaxSpringLength) ) )
                                ); //Eqn (12)


#endif //DEBUG


                    }

          }

     /* Fixed bug were −nan values returned */
     /*
      * A second possible solution would be to return
      * a value in [5.999 − 6.012] range. As this parameter
      * does not seem to vary much. In fact, for speed it
      * could be beneficial to simply set this parameter.
      *
      */

     if (ActiveCount == 0)
     {
          /*
           * Instead of exiting out here, return 0 value.
```

```
            * Then use zero value to exit main loop.
            *
            * This will hopefully preserve the data up
            * until the 0 value is reached.
            */

           return 0;
      }

#ifdef DEBUG
  printf("AvgSpringLife...Hernandez-Cife: %f Sing-McKinley: %f \n",
                  Total2/ActiveCount, Total1/ActiveCount);
#endif //DEBUG



  return Total2 / (double) ActiveCount;

#endif //NEW_TAU

}


void SpeciesRatioCount( int SpeciesType[], double *ActiveRatio,
                        double *DangleRatio, double *LoopedRatio){
    unsigned int j;
    int NumOfActive=0;
    int NumOfDangling=0;
    int NumOfLooped=0;
    for (j=0; j<hostNumberOfParticles; j++){

        switch (SpeciesType[j]){
            case 0:
               NumOfActive++;
            break;

            case 1:
                NumOfDangling++;
            break;

            case 2:
                NumOfLooped++;
            break;

            default:
                printf("Error: Undetermined Species Type!\n");
            break;

        }
    }

        *ActiveRatio = (double)NumOfActive / hostNumberOfParticles;
        *DangleRatio = (double)NumOfDangling / hostNumberOfParticles;
        *LoopedRatio = (double)NumOfLooped / hostNumberOfParticles;

}

```

```

//Function
//CPU
//Caluculate Ratios, Return Average Lengths and Variation, Histogram per type.
void Detailed_Info (int SpeciesType[], double SpringLenX[], double SpringLenY[],
                    SpeciesValue *AvgLen, SpeciesValue *Variance,
#ifdef NEW_DNG_LN
                    TwoDimSpring *AvgDng,
#endif
                    int NumOfBins, SpeciesCount **Hist, int t_step){

    //NOTES: *AvgLen.Active = AvgLen->Active

    //histogram
    int BinNumber = 0;

    //Initialize Bins to 0
    for(unsigned int i=0; i<NumOfBins; i++){
        Hist[i][t_step].Active = 0;
        Hist[i][t_step].Dangle = 0;
        Hist[i][t_step].Looped = 0;
    }



    double max = sqrt( 2 * hostMaxSpringLength * hostMaxSpringLength );
    double min = 0;


    double SpringLen = 0.0;
    double SpringAng = 0.0;



    SpeciesCount NumOf = {0};

    /*
    * Need to initialize average lengths to 0.
    */
    AvgLen->ActiveLen = 0;
    AvgLen->DangleLen = 0;
    AvgLen->LoopedLen = 0;

    AvgLen->ActiveAng = 0;
    AvgLen->DangleAng = 0;
    AvgLen->LoopedAng = 0;

    AvgLen->ActiveX = 0;
    AvgLen->DangleX = 0;
    AvgLen->LoopedX = 0;
    AvgLen->ActiveY = 0;
    AvgLen->DangleY = 0;
    AvgLen->LoopedY = 0;

    Variance->ActiveLen = 0;
    Variance->DangleLen = 0;
    Variance->LoopedLen = 0;

    Variance->ActiveAng = 0;
    Variance->DangleAng = 0;
    Variance->LoopedAng = 0;

    Variance->ActiveX = 0;
    Variance->DangleX = 0;
    Variance->LoopedX = 0;
    Variance->ActiveY = 0;
    Variance->DangleY = 0;
    Variance->LoopedY = 0;

    NumOf.Active = 0;
    NumOf.Dangle = 0;
    NumOf.Looped = 0;


#ifdef NEW_DNG_LN
    AvgDng->x = 0;
    AvgDng->y = 0
#endif


    for (unsigned int j=0; j<hostNumberOfParticles; j++){

        SpringLen = sqrt( SpringLenX[j] * SpringLenX[j] +
                          SpringLenY[j] * SpringLenY[j] );


        BinNumber = (int)(( SpringLen - min) / ((max-min)/ NumOfBins));


        switch (SpeciesType[j]){
            case 0:
                //Average Length
                NumOf.Active++;
                AvgLen->ActiveLen += SpringLen;
                AvgLen->ActiveAng += atan(SpringLenY[j] / SpringLenX[j]);
                AvgLen->ActiveX += SpringLenX[j];
                AvgLen->ActiveY += SpringLenY[j];
                Hist[BinNumber][t_step].Active++;

            break;

            case 1:
                //Average Length
                NumOf.Dangle++;
                AvgLen->DangleLen += SpringLen;
                AvgLen->DangleAng += atan(SpringLenY[j] / SpringLenX[j]);
                AvgLen->DangleX += SpringLenX[j];
                AvgLen->DangleY += SpringLenY[j];
                Hist[BinNumber][t_step].Dangle++;
#ifdef NEW_DNG_LN
                AvgDng->x += SpringLenX[j];
                AvgDng->y += SpringLenY[j];
#endif
            break;

```

```
                case 2:
                    //Average Length
                    NumOf.Looped++;
                    AvgLen->LoopedLen += SpringLen;
                    AvgLen->LoopedAng += atan(SpringLenY[j] / SpringLenX[j]);
                    AvgLen->LoopedX += SpringLenX[j];
                    AvgLen->LoopedY += SpringLenY[j];
                    Hist[BinNumber][t_step].Looped++;

                break;

                default:
                     printf("Error: Undetermined Species Type!\n");
                break;

            }
        }


#ifdef NEW_DNG_LN
        AvgDng->x = AvgDng->x / NumOf.Dangle;
        AvgDng->y = AvgDng->y / NumOf.Dangle;
#endif


        if ((AvgLen->ActiveLen/NumOf.Active > hostMaxSpringLength) ||
                        (AvgLen->DangleLen/NumOf.Dangle> hostMaxSpringLength) ||
                        (AvgLen->LoopedLen/NumOf.Looped > hostMaxSpringLength)){
            printf("Average Length - Active[%f] Dangling[%f] Looped[%f]\n",
                    AvgLen->ActiveLen, AvgLen->DangleLen, AvgLen->LoopedLen);
            printf("NumofActive: %d NumofDangle: %d NumofLooped: %d \n",
                            NumOf.Active, NumOf.Dangle, NumOf.Looped);
        }



        if (NumOf.Active == 0){
            AvgLen->ActiveLen = 0;
            AvgLen->ActiveAng = 0;
            AvgLen->ActiveX = 0;
            AvgLen->ActiveY = 0;
        } else {
            AvgLen->ActiveLen = (double) AvgLen->ActiveLen / NumOf.Active;
            AvgLen->ActiveAng = (double) AvgLen->ActiveAng / NumOf.Active;
            AvgLen->ActiveX = (double) AvgLen->ActiveX / NumOf.Active;
            AvgLen->ActiveY = (double) AvgLen->ActiveY / NumOf.Active;
        }

        if (NumOf.Dangle == 0){
            AvgLen->DangleLen = 0;
            AvgLen->DangleAng = 0;
            AvgLen->DangleX = 0;
            AvgLen->DangleY = 0;
        } else {
            AvgLen->DangleLen = (double) AvgLen->DangleLen / NumOf.Dangle;
            AvgLen->DangleAng = (double) AvgLen->DangleAng / NumOf.Dangle;
            AvgLen->DangleX = (double) AvgLen->DangleX / NumOf.Dangle;
            AvgLen->DangleY = (double) AvgLen->DangleY / NumOf.Dangle;
        }

        if (NumOf.Looped == 0){
            AvgLen->LoopedLen = 0;
            AvgLen->LoopedAng = 0;
            AvgLen->LoopedX = 0;
            AvgLen->LoopedY = 0;
        } else {
            AvgLen->LoopedLen = (double) AvgLen->LoopedLen / NumOf.Looped;
            AvgLen->LoopedAng = (double) AvgLen->LoopedAng / NumOf.Looped;
            AvgLen->LoopedX = (double) AvgLen->LoopedX / NumOf.Looped;
            AvgLen->LoopedY = (double) AvgLen->LoopedY / NumOf.Looped;
        }


       /*
        * Calculate Variance
        *
        */

        for (unsigned int j=0; j<hostNumberOfParticles; j++){

            SpringLen = sqrt( SpringLenX[j] * SpringLenX[j] +
                                      SpringLenY[j] * SpringLenY[j] );
            SpringAng = atan( SpringLenY[j] / SpringLenX[j]);

            switch (SpeciesType[j]){
                case 0:
                    Variance->ActiveLen += (SpringLen - AvgLen->ActiveLen) *
```

```
                    (SpringLen − AvgLen−>ActiveLen);
                Variance−>ActiveAng += (SpringAng − AvgLen−>ActiveAng) ∗
                    (SpringAng − AvgLen−>ActiveAng);
                Variance−>ActiveX += (SpringLenX[j] − AvgLen−>ActiveX) ∗
                    (SpringLenX[j] − AvgLen−>ActiveX);
                Variance−>ActiveY += (SpringLenY[j] − AvgLen−>ActiveY) ∗
                    (SpringLenY[j] − AvgLen−>ActiveY);
            break;

            case 1:
                Variance−>DangleLen += (SpringLen − AvgLen−>DangleLen) ∗
                    (SpringLen − AvgLen−>DangleLen);
                Variance−>DangleAng += (SpringAng − AvgLen−>DangleAng) ∗
                    (SpringAng − AvgLen−>DangleAng);
                Variance−>DangleX += (SpringLenX[j] − AvgLen−>DangleX) ∗
                    (SpringLenX[j] − AvgLen−>DangleX);
                Variance−>DangleY += (SpringLenY[j] − AvgLen−>DangleY) ∗
                    (SpringLenY[j] − AvgLen−>DangleY);
            break;

            case 2:
                Variance−>LoopedLen += (SpringLen − AvgLen−>LoopedLen) ∗
                    (SpringLen − AvgLen−>LoopedLen);
                Variance−>LoopedAng += (SpringAng − AvgLen−>LoopedAng) ∗
                    (SpringAng − AvgLen−>LoopedAng);
                Variance−>LoopedX += (SpringLenX[j] − AvgLen−>LoopedX) ∗
                    (SpringLenX[j] − AvgLen−>LoopedX);
                Variance−>LoopedY += (SpringLenY[j] − AvgLen−>LoopedY) ∗
                    (SpringLenY[j] − AvgLen−>LoopedY);
                break;

                default:
                    printf("Error: Undetermined Species Type!\n");
                break;

            }
        }


        if (NumOf.Active == 0){
            Variance->ActiveLen = 0.0;
            Variance->ActiveAng = 0.0;
            Variance->ActiveX = 0.0;
            Variance->ActiveY = 0.0;
        } else {
            Variance->ActiveLen = (double) Variance->ActiveLen / NumOf.Active;
            Variance->ActiveAng = (double) Variance->ActiveAng / NumOf.Active;
            Variance->ActiveX = (double) Variance->ActiveX / NumOf.Active;
            Variance->ActiveY = (double) Variance->ActiveY / NumOf.Active;
        }

        if (NumOf.Dangle == 0){
            Variance->DangleLen = 0.0;
            Variance->DangleAng = 0.0;
            Variance->DangleX = 0.0;
            Variance->DangleY = 0.0;
        } else {
            Variance->DangleLen = (double) Variance->DangleLen / NumOf.Dangle;
            Variance->DangleAng = (double) Variance->DangleAng / NumOf.Dangle;
            Variance->DangleX = (double) Variance->DangleX / NumOf.Dangle;
            Variance->DangleY = (double) Variance->DangleY / NumOf.Dangle;
        }

        if (NumOf.Looped == 0){
            Variance->LoopedLen = 0.0;
            Variance->LoopedAng = 0.0;
            Variance->LoopedX = 0.0;
            Variance->LoopedY = 0.0;
        } else {
            Variance->LoopedLen = (double) Variance->LoopedLen / NumOf.Looped;
            Variance->LoopedAng = (double) Variance->LoopedAng / NumOf.Looped;
            Variance->LoopedX = (double) Variance->LoopedX / NumOf.Looped;
            Variance->LoopedY = (double) Variance->LoopedY / NumOf.Looped;
        }



    }






    // Function:
    // CPU Function
    // Given Types, X and Y lengths, calculate ensemble
    // average stress XX, XY, and YY.
    void EnsembleAverage (int SpeciesType[], double SpringLenX[],
                          double SpringLenY[],
                          struct Stress *TotalStress, struct Stress *Active,
                          struct Stress *Dangling, int k){

        /*
        * Notes on Stress Calculation
        *
        * Weighted average calculated based on real-time numbers of active and
        * dangling dumbbells. Simplifaction causes division in the mean
        * calculation by the number of each type to cancel with the weighting
        * average factor. Leaving only division by the total number
        * of dumbbells.
        *
        * see the calculations in StressCalculation.pdf
        *
        */

        Active[k].XX = 0;
        Active[k].XY = 0;
        Active[k].YY = 0;

        Dangling[k].XX = 0;
        Dangling[k].XY = 0;
        Dangling[k].YY = 0;


        TotalStress[k].XX = 0;
        TotalStress[k].XY = 0;
        TotalStress[k].YY = 0;

        double LengthLimiter = 1.0;
        int NumberOfActive = 0;
        int NumberOfDangle = 0;

        /* DEBUG */
        /*************************/
        int ErrorFlag_ZeroSpringLength = 0;
        int ZeroSpringCount = 0;
        /*************************/

       /*
        * DETAILED METHOD
        *
        * It is also possible to simplify this calculation further if necessary.
        * However, this method is tested and provides additional information
        * To be exact, this methods shows stress contribution from each type
        * of dumbbell, and the total stress.
        */


            for (unsigned int j=0; j<hostNumberOfParticles; j++){

            /* DEBUG */
            /*************************/
            ZeroSpringCount = 0;
            /*************************/
            switch (SpeciesType[j]){

            case 0:
                  NumberOfActive++;

              //___FENE Springs_______

              LengthLimiter =
                (1.0 − (SpringLenX[j] ∗ SpringLenX[j]
                  + SpringLenY[j] ∗ SpringLenY[j])
                  / (hostMaxSpringLength∗hostMaxSpringLength));

              Active[k].XX += SpringLenX[j] ∗ SpringLenX[j] / LengthLimiter;
              Active[k].XY += SpringLenX[j] ∗ SpringLenY[j] / LengthLimiter;
              Active[k].YY += SpringLenY[j] ∗ SpringLenY[j] / LengthLimiter;

              //''''''''''''''''''''''''''''''''''
              /∗ DEBUG ∗/
              /*************************/
              if ((SpringLenX[j] == 0 ) && (SpringLenY[j] == 0))
              {
                ErrorFlag_ZeroSpringLength = 1;
                ZeroSpringCount++;
              }
                /*************************/
            break;
            case 1:
              NumberOfDangle++;

              //___FENE Springs______

              LengthLimiter =
                (1.0 - (SpringLenX[j] * SpringLenX[j]
                      + SpringLenY[j] * SpringLenY[j])
                      / (hostMaxSpringLength*hostMaxSpringLength));

              Dangling[k].XX += SpringLenX[j] * SpringLenX[j] / LengthLimiter;
              Dangling[k].XY += SpringLenX[j] * SpringLenY[j] / LengthLimiter;
              Dangling[k].YY += SpringLenY[j] * SpringLenY[j] / LengthLimiter;

              //''''''''''''''''''''''''''''''''
              /* DEBUG */
              /*************************/
              if ((SpringLenX[j] == 0 ) && (SpringLenY[j] == 0))
              {
                ErrorFlag_ZeroSpringLength = 1;
                ZeroSpringCount++;
              }
              /*************************/

            break;

            case 2:
            break;

            default:
                  printf("Error: Unable to Classify Species Type[%d] Of Dumbbell[%d]\n",
                     SpeciesType[j], j);
              exit(4);
            }
            }


        if (NumberOfActive == 0){
            TotalStress[k].XX = − 2.0 / (double) hostNumberOfParticles ∗
                    Dangling[k].XX;
            TotalStress[k].XY = − 2.0 / (double) hostNumberOfParticles ∗
                    Dangling[k].XY;
            TotalStress[k].YY = − 2.0 / (double) hostNumberOfParticles ∗
                    Dangling[k].YY;

            Active[k].XX = 0;
            Active[k].XY = 0;
            Active[k].YY = 0;

            Dangling[k].XX = − 2.0 / (double) hostNumberOfParticles ∗
                    Dangling[k].XX;
            Dangling[k].XY = − 2.0 / (double) hostNumberOfParticles ∗
                    Dangling[k].XY;
            Dangling[k].YY = − 2.0 / (double) hostNumberOfParticles ∗
                    Dangling[k].YY;


        } else {
            if (NumberOfDangle == 0){
                TotalStress[k].XX = − 2.0 / (double) hostNumberOfParticles ∗
                        ( Active[k].XX );
                TotalStress[k].XY = − 2.0 / (double) hostNumberOfParticles ∗
                        ( Active[k].XY );
                TotalStress[k].YY = − 2.0 / (double) hostNumberOfParticles ∗
                        ( Active[k].YY );

                Active[k].XX = − 2.0 / (double) hostNumberOfParticles ∗
                        Active[k].XX;
                Active[k].XY = − 2.0 / (double) hostNumberOfParticles ∗
                        Active[k].XY;
                Active[k].YY = − 2.0 / (double) hostNumberOfParticles ∗
                        Active[k].YY;

                Dangling[k].XX = 0;
                Dangling[k].XY = 0;
                Dangling[k].YY = 0;


            } else {
                TotalStress[k].XX = − 2.0 / (double) hostNumberOfParticles ∗
                        ( Active[k].XX + Dangling[k].XX );
                TotalStress[k].XY = − 2.0 / (double) hostNumberOfParticles ∗
                        ( Active[k].XY + Dangling[k].XY );
                TotalStress[k].YY = − 2.0 / (double) hostNumberOfParticles ∗
                        ( Active[k].YY + Dangling[k].YY );

                Active[k].XX = − 2.0 / (double) hostNumberOfParticles ∗
```

```
                    Active[k].XX;
                Active[k].XY = − 2.0 / (double) hostNumberOfParticles ∗
                    Active[k].XY;
                Active[k].YY = − 2.0 / (double) hostNumberOfParticles ∗
                    Active[k].YY;

                Dangling[k].XX = − 2.0 / (double) hostNumberOfParticles ∗
                    Dangling[k].XX;
                Dangling[k].XY = − 2.0 / (double) hostNumberOfParticles ∗
                    Dangling[k].XY;
                Dangling[k].YY = − 2.0 / (double) hostNumberOfParticles ∗
                    Dangling[k].YY;

        }
    }


    /∗
    ∗ END DETAILED METHOD
    ∗/


    /∗ DEBUG ∗/
    /∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗/
    if (ErrorFlag_ZeroSpringLength)
        printf("%d Active or Dangling springs had zero length at",
                        " time step %d\n", ZeroSpringCount, k);
    /∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗/

```

```
}


int RawOut_OSFlow(int loop_step){

    /*
     * For Oscillatory Shear Flow
     * Sample the last <Cycle_Num> cycles of the simutlation.
     * Records dumbbell positions and types for 100 snap shots during simulation.
     *
     */


    int Cycle_Num = 2; //take snap shots over final two cycles

    double Time = Cycle_Num * 2 * M_PI / hostFreq; //second for Cycle_Num cycles

    double MacroSizeStg2 = hostStepSizeMicroSecon * hostTimeStepsMicro;
    //seconds per Macro step in stage 2

    int MacroStepsXCycle = (int) floor(Time / MacroSizeStg2) + 1;

    int StartPt = 0;// = hostTimeStepsMacro - MacroStepsXCycle;

    int SampRate = 0; //(int) floor(MacroStepsXCycle / 100);

    int NumOfSamples = 200; //desired number of sample snapshots to take.

int Remdr = 0;


if ((NumOfSamples > MacroStepsXCycle) ||
                (NumOfSamples > hostTimeStepsMacro)){
    //then sample last NumOfSamples macro steps
    StartPt = hostTimeStepsMacro − NumOfSamples;


    if (loop_step > StartPt){
        return 1; //take raw data snap shot
    } else {
        return 0;
    }

} else {
    //evenly space NumOfSamples amoung the final macro steps.
    SampRate = (int) floor(MacroStepsXCycle / 200);

    StartPt = hostTimeStepsMacro − MacroStepsXCycle;

    Remdr = (loop_step − StartPt) % SampRate;


    if ((loop_step > StartPt) && (Remdr == 0)){
        //printf("[%d]: Take Snapshot!\n",loop_step);
        return 1;
    } else {
        return 0;
```

```
            }

    }
}


/*
 * For Steady Shear Flow
 * Records 100 snapshots of dumbbell positions and types
 * at even spaced intervals.
 *
 */

int RawOut_SSFlow(int loop_step){

    //For splitting up only the second stage


#ifdef FULL_DATA
    int SampleNum = (int) hostTimeStepsMacro / 800;
#else
    int SampleNum = (int) hostTimeStepsMacro / 100;
#endif

    if (loop_step % SampleNum == 0){
            return 1;
    } else {
            return 0;
    }
```

```

}


#ifdef SPEC_CHNG
//sums the entries of a vector
unsigned int VectorSum( unsigned int *Vector, unsigned int length){

    unsigned int sum = 0;

    for(unsigned int n=0; n<length; n++){
            sum += Vector[n];
    }

    return sum;
}
#endif


int main(int argc, char *argv[]){

#ifdef DEBUG
    printf("START DEBUG MODE\n");
    printf("DEBUG: Start Sim\n");
#endif

        //______Record Program Run Time
        clock_t begin, end, end2;
        begin = clock();
        double time_spent, time_spent2;
        //''''''''''''''''''''''''''''''''''''''''''''''''''''''



        // _____ Read Command Line Arguments ______

        if (ParseInput(argc, argv)==EXIT_FAILURE){
                //return EXIT_FAILURE;
                exit(2);
        }
        //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''

    /*
     * Select GPU Device
     */
    cudaSetDevice(GPU_select);

        PrintSimInfo(); //Output Simulation Variables to Terminal

        //--- Set Global Variable Values -------
        cudaMemcpyToSymbol(devFlowRate, &hostFlowRate, sizeof(double));
        cudaMemcpyToSymbol(devMaxSpringLength, &hostMaxSpringLength,
                        sizeof(double));
        cudaMemcpyToSymbol(devFreq, &hostFreq, sizeof(double));

        // additional command line arguments
        cudaMemcpyToSymbol(devD_free, &hostD_free, sizeof(double));
        cudaMemcpyToSymbol(devZee, &hostZee, sizeof(double));
        cudaMemcpyToSymbol(devChi, &hostChi, sizeof(double));
```

```
        cudaMemcpyToSymbol(devAlpha, &hostAlpha, sizeof(double));
        cudaMemcpyToSymbol(devBeta, &hostBeta, sizeof(double));

        //''''''''''''''''''''''''''''''''''''''''''''''''''


        //----define block and thread structure -----
        dim3 block;

        if (hostNumberOfParticles < 512){
                block.x = hostNumberOfParticles;
                block.y = 1;
        }
        else {
                block.x=512;
                block.y=1;
        }

        dim3 grid ((hostNumberOfParticles + block.x -1) / block.x,1);
        //''''''''''''''''''''''''''''''''''''''''''''''''


        //___Variables for random number generation on GPU kernels
        curandState *states = NULL;
        curandState *ProbStates = NULL;
        //''''''''''''''''''''''''''''''''''''''''''''

        //______allocate memory on GPU for random number generator states_________
        CUDA_CALL(cudaMalloc((void **)&states, sizeof(curandState) *
                           hostNumberOfParticles ));
        CUDA_CALL(cudaMalloc((void **)&ProbStates, sizeof(curandState) *
                           hostNumberOfParticles ));
        //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''

        //___create vectors of seeds_______
        unsigned long long *hostSeeds = NULL;
    unsigned long long *devSeeds = NULL;

        unsigned long long *hostProbSeeds = NULL;
    unsigned long long *devProbSeeds = NULL;


        hostSeeds = (unsigned long long *)malloc(hostNumberOfParticles *
                  sizeof(unsigned long long));
        hostProbSeeds = (unsigned long long *)malloc(hostNumberOfParticles *
                  sizeof(unsigned long long));

   /*
    * Verify memory allocated successfully.
    */
    if (hostSeeds == NULL)
        printf("hostSeeds memory error.\n");
    if (hostProbSeeds == NULL)
        printf("hostProbSeeds memory error.\n");


        CUDA_CALL(cudaMalloc((void **)&devSeeds, sizeof(unsigned long long) *
                           hostNumberOfParticles));
```

```
        CUDA_CALL(cudaMalloc((void **)&devProbSeeds,
                              sizeof(unsigned long long) *
                              hostNumberOfParticles));

#ifdef FIXED_SEED
    srand(1);
#else
    srand(time(NULL));
#endif
    //Start from one random number and count from there.

    hostSeeds[0] = abs(rand());
    hostProbSeeds[0] = abs(rand());

        for (unsigned int i=1; i<hostNumberOfParticles; i++){
                hostSeeds[i] = hostSeeds[i-1] + 1;
                hostProbSeeds[i] = hostProbSeeds[i-1] + 1;

        }
        //''''''''''''''''''''''''''''''''''''


        CUDA_CALL(cudaMemcpy(devSeeds, hostSeeds, sizeof(unsigned long long) *
                              hostNumberOfParticles, cudaMemcpyHostToDevice));
        CUDA_CALL(cudaMemcpy(devProbSeeds, hostProbSeeds,
                              sizeof(unsigned long long) *
                              hostNumberOfParticles, cudaMemcpyHostToDevice));


        //____initialze kernel random number generator on GPU threads_____
        RandomGenInit<<< grid, block >>>(devSeeds, states);
        gpuErrchk( cudaPeekAtLastError() ); //Error catching
        gpuErrchk( cudaDeviceSynchronize() );
//for catching errors. If removed, may give errors from other places
        RandomGenInit<<< grid, block >>>(devProbSeeds, ProbStates);
        gpuErrchk( cudaPeekAtLastError() );
        gpuErrchk( cudaDeviceSynchronize() );
        //''''''''''''''''''''''''''''''''''''''''

        //_____Spring Length variables_____
        double *devSpringLenX = NULL;
    double *devSpringLenY = NULL;
        double *hostSpringLenX = NULL;
    double *hostSpringLenY = NULL;
        //''''''''''''''''''''''''''''''''''''

        //____Dumbbell Species Type Variable____
        int *devSpeciesType = NULL;
        int *hostSpeciesType = NULL;
        //'''''''''''''''''''''''''''''''''''''

        //_________allocte memory on CPU
        hostSpringLenX = (double*)malloc(hostNumberOfParticles*sizeof(double));
        hostSpringLenY = (double*)malloc(hostNumberOfParticles*sizeof(double));
        hostSpeciesType = (int*)malloc(hostNumberOfParticles*sizeof(int));

    if (hostSpringLenX == NULL)
        printf("hostSpringLenX memory error.\n");
    if (hostSpringLenY == NULL)
        printf("hostSpringLenY memory error.\n");
    if (hostSpeciesType == NULL)
        printf("hostSpeciesType memory error.\n");
        //''''''''''''''''''''''''''''

        //_______allocate memory on GPU for spring length
        CUDA_CALL(cudaMalloc((double**)&devSpringLenX,
                                    hostNumberOfParticles*sizeof(double)));
        CUDA_CALL(cudaMalloc((double**)&devSpringLenY ,
                                    hostNumberOfParticles*sizeof(double)));
        CUDA_CALL(cudaMalloc((int**)&devSpeciesType,
                                    hostNumberOfParticles*sizeof(int)));
        //''''''''''''''''''''''''''''''''''''''''''


    //____ initialize memory _____
    for(unsigned int n=0; n < hostNumberOfParticles; n++)
    {
        hostSpringLenX[n] = 0.0;
        hostSpringLenY[n] = 0.0;
        hostSpeciesType[n] = 0;
    }



#ifdef SPEC_CHNG

    // Count species changes per macro time step

    unsigned int *hostDng2Act = NULL;
    unsigned int *hostDng2Lpd = NULL;
    unsigned int *hostAct2Dng = NULL;
    unsigned int *hostLpd2Dng = NULL;

    unsigned int *devDng2Act = NULL;
    unsigned int *devDng2Lpd = NULL;
    unsigned int *devAct2Dng = NULL;
    unsigned int *devLpd2Dng = NULL;

    hostDng2Act = (unsigned int*)malloc(hostNumberOfParticles *
                        sizeof(unsigned int));
    hostDng2Lpd = (unsigned int*)malloc(hostNumberOfParticles *
                        sizeof(unsigned int));
    hostAct2Dng = (unsigned int*)malloc(hostNumberOfParticles *
                        sizeof(unsigned int));
    hostLpd2Dng = (unsigned int*)malloc(hostNumberOfParticles *
                        sizeof(unsigned int));

    if (hostDng2Act == NULL) printf("hostDng2Act memory error.\n");
    if (hostDng2Lpd == NULL) printf("hostDng2Lpd memory error.\n");
    if (hostAct2Dng == NULL) printf("hostAct2Dng memory error.\n");
    if (hostLpd2Dng == NULL) printf("hostLpd2Dng memory error.\n");

    CUDA_CALL(cudaMalloc((unsigned int**)&devDng2Act, hostNumberOfParticles *
                            sizeof(unsigned int)));
    CUDA_CALL(cudaMalloc((unsigned int**)&devDng2Lpd, hostNumberOfParticles *
                            sizeof(unsigned int)));
```

```
CUDA_CALL(cudaMalloc((unsigned int**)&devAct2Dng, hostNumberOfParticles *
                        sizeof(unsigned int)));
CUDA_CALL(cudaMalloc((unsigned int**)&devLpd2Dng, hostNumberOfParticles *
                        sizeof(unsigned int)));

for(unsigned int n=0; n < hostNumberOfParticles; n++)
{
    hostDng2Act[n] = 0;
    hostDng2Lpd[n] = 0;
    hostAct2Dng[n] = 0;
    hostLpd2Dng[n] = 0;
}

// Save count for each loop
unsigned int *Dng2ActSum = NULL;
unsigned int *Dng2LpdSum = NULL;
unsigned int *Act2DngSum = NULL;
unsigned int *Lpd2DngSum = NULL;


Dng2ActSum = (unsigned int*)malloc((hostTimeStepsMacro+1) *
                sizeof(unsigned int));
Dng2LpdSum = (unsigned int*)malloc((hostTimeStepsMacro+1) *
                sizeof(unsigned int));
Act2DngSum = (unsigned int*)malloc((hostTimeStepsMacro+1) *
                sizeof(unsigned int));
Lpd2DngSum = (unsigned int*)malloc((hostTimeStepsMacro+1) *
                sizeof(unsigned int));

    if (Dng2ActSum == NULL) printf("Dng2ActSum memory error.\n");
    if (Dng2LpdSum == NULL) printf("Dng2LpdSum memory error.\n");
    if (Act2DngSum == NULL) printf("Act2DngSum memory error.\n");
    if (Lpd2DngSum == NULL) printf("Lpd2DngSum memory error.\n");

#endif

#ifdef SINGLE_MICRO
    // Tracks every species transition of every time step (micro)

    DBSpecChng *hostSCArr = NULL;

    DBSpecChng *devSCArr = NULL;

    hostSCArr = (DBSpecChng *) malloc ( hostTimeStepsMicro *
                    sizeof (DBSpecChng));
    CUDA_CALL(cudaMalloc((DBSpecChng **)&devSCArr, hostTimeStepsMicro *
                        sizeof(DBSpecChng)));



    // intialize array values to 0

      for(unsigned int m=0; m < hostTimeStepsMicro; m++){
        hostSCArr[m].type = 0;
        hostSCArr[m].time = 0.0;
        hostSCArr[m].x = 0.0;
        hostSCArr[m].y = 0.0;
      }


    CUDA_CALL(cudaMemcpy(devSCArr, hostSCArr,
                        hostTimeStepsMicro * sizeof(DBSpecChng),
                        cudaMemcpyHostToDevice));


    //file to write data to.

    char MicroDataFilename[256];

    sprintf(MicroDataFilename, "%s_single_micro.bin", DataFileName);

    FILE *MicroDataFilePtr = NULL;

    MicroDataFilePtr = fopen(MicroDataFilename, "wb");
    if (!MicroDataFilePtr) printf("Unable to open micro data file!\n");

    size_t MicroData_FileSize; //For checking file size.


#endif

#ifdef MICRO_RAW
    // Tracks every species transition of every time step (micro)

    DBSpecChng *hostSCArr = NULL;

    DBSpecChng *devSCArr = NULL;
```

```
hostSCArr = (DBSpecChng *) malloc ( hostNumberOfParticles *
                        hostTimeStepsMicro * sizeof (DBSpecChng));
CUDA_CALL(cudaMalloc((DBSpecChng **)&devSCArr, hostNumberOfParticles *
                                    hostTimeStepsMicro * sizeof(DBSpecChng)));



// intialize array values to 0

for(unsigned int n=0; n < hostNumberOfParticles; n++){
  for(unsigned int m=0; m < hostTimeStepsMicro; m++){
    hostSCArr[n*hostTimeStepsMicro+m].type = 0;
    hostSCArr[n*hostTimeStepsMicro+m].length = 0.0;
  }
}


CUDA_CALL(cudaMemcpy(devSCArr, hostSCArr,
                  hostNumberOfParticles * hostTimeStepsMicro * sizeof(DBSpecChng),
                  cudaMemcpyHostToDevice));


//file to write data to.

char MicroDataFilename[256];

sprintf(MicroDataFilename, "%s_micro.bin", DataFileName);

```

```
    FILE *MicroDataFilePtr = NULL;

    MicroDataFilePtr = fopen(MicroDataFilename, "wb");
    if (!MicroDataFilePtr) printf("Unable to open micro data file!\n");

    size_t MicroData_FileSize; //For checking file size.


#endif


        //____Simulation Time_____
        //Variables for tracking time t throughout simulation
        //Highest memory cost solution I can think of. There is probably a better way.

    /*
     *
     * First:
     * Create dynamically allocated array to store micro time step sizes.
     *
     * Second:
     * Transfer only the starting value to the GPU.
     * Return only the final value from the GPU.
     *
     *
     */

        double *devSimTime = NULL;
    double *hostSimTime = NULL;
```

```

        hostSimTime = (double *)malloc(hostNumberOfParticles*sizeof(double));

    if (hostSimTime == NULL)
        printf("hostSimTime memory error.\n");

        CUDA_CALL(cudaMalloc((double**)&devSimTime,
              hostNumberOfParticles*sizeof(double)));

    //____ initialize memory _____
    for(unsigned int n=0; n < hostNumberOfParticles; n++)
        hostSimTime[n] = 0.0;

        CUDA_CALL(cudaMemcpy(devSimTime, hostSimTime,
              hostNumberOfParticles*sizeof(double), cudaMemcpyHostToDevice));
        //''''''''''''''''''''''''''''''''''''''''''''''''''''''''



        //____ Set initial Spring Lengths to Normal Distributuion
        // "initially... equilibrium Gaussian distribution"


        double failsafe = 0.0;

        for (unsigned int i=0; i < hostNumberOfParticles; i++){



```

```
			if (hostMaxSpringLength < 1){
					hostSpringLenX[i] = RndNorm() * hostMaxSpringLength;
		//Shrink initial distribution to fit within maximum length
					hostSpringLenY[i] = RndNorm() * hostMaxSpringLength;
			} else {


#ifdef SKEW_START
	/*
	 * Start simulations from the V-shape position
	 * tests to see if this position is a potential well
	 */
					hostSpringLenX[i] = (double) ( RndNorm() + 10 );
					hostSpringLenY[i] = (double) ( RndNorm() + 2 );
#else
					//____ Set initial length randomly___
					hostSpringLenX[i] = RndNorm();
					hostSpringLenY[i] = RndNorm();
			//Starting from this appears to speed up
			// steady state for SAOS
					//''''''''''''''''''''''''''''''''''
#endif
			}



//___FENE SIM ____ Ensure initial spring lengths are within maximum allowed
// So far this construction seems effective in enforcing the
// initial condition.

        /*
         * This code inside the if can cause a sig fault if the first dumbbell
         * doesn't meet the condition.
         */


                if ( hostSpringLenX[i] * hostSpringLenX[i] +
                    hostSpringLenY[i] * hostSpringLenY[i] >
                    hostMaxSpringLength * hostMaxSpringLength){
      if ( i == 0 ) {
        printf("First dumbbell did not initialize under maximum length.\n");
        printf("Check parameters! Exiting to prevent seg fault.\n");
        exit(3);
      }
                        i--;
                        failsafe++;
                }
                if ( failsafe > 4 * hostNumberOfParticles ) {
                        printf("failed to initialze dumbbells\n");
                        exit(3);

                }
                //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''


                //____set initial species type___

        /*
         * Initial Species Assignment
         *
         * Active = 0
         * Dangling = 1
         * Looped = 2
         *
         */

    if ( i < hostNumberOfParticles * Init_Active_Ratio ){
      hostSpeciesType[i]=0;

    } else {

      if ( i < hostNumberOfParticles *
                                ( Init_Active_Ratio + Init_Dangle_Ratio ))
      {
        hostSpeciesType[i]=1;
      } else {
        hostSpeciesType[i]=2;
      }
    }



        //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''


                //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''

        }

    printf("Dumbbells Successfully Initialized.\n");

        //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''

        //______Copy to Gpu device
        CUDA_CALL(cudaMemcpy(devSpringLenX, hostSpringLenX,
                              hostNumberOfParticles*sizeof(double),
                              cudaMemcpyHostToDevice));
        CUDA_CALL(cudaMemcpy(devSpringLenY, hostSpringLenY,
                              hostNumberOfParticles*sizeof(double),
                              cudaMemcpyHostToDevice));
        CUDA_CALL(cudaMemcpy(devSpeciesType, hostSpeciesType,
                              hostNumberOfParticles*sizeof(int),
                              cudaMemcpyHostToDevice));
        //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''




        //____ initialize variables to calculate and store ensemble average
        double *Spring_AvgLen_XX = NULL;
        double *Spring_AvgLen_XY = NULL;
        double *Spring_AvgLen_YY = NULL;

        Spring_AvgLen_XX = (double*)malloc((hostTimeStepsMacro+1)
                          * sizeof(double));
        Spring_AvgLen_XY = (double*)malloc((hostTimeStepsMacro+1)
                        * sizeof(double));
        Spring_AvgLen_YY = (double*)malloc((hostTimeStepsMacro+1)
                        * sizeof(double));


    if (Spring_AvgLen_XX == NULL)
        printf("Spring_AvgLen_XX memory error.\n");
    if (Spring_AvgLen_XY == NULL)
        printf("Spring_AvgLen_XY memory error.\n");
    if (Spring_AvgLen_YY == NULL)
        printf("Spring_AvgLen_YY memory error.\n");


    //____ initialize memory _____
    for(unsigned int n=0; n < hostTimeStepsMacro+1; n++)
    {
        Spring_AvgLen_XX[n] = 0.0;
        Spring_AvgLen_XY[n] = 0.0;
        Spring_AvgLen_YY[n] = 0.0;
    }


        unsigned int k; //iterating variable used for main loop //Why here?

        //''''''''''''''''''''''''''''''''''''''''''''''''''''''''

        //____ Track Species Ratios ____
    double *ActiveRatio = NULL;
```

```
          double *DangleRatio = NULL;
          double *LoopedRatio = NULL;

          ActiveRatio = (double*)malloc((hostTimeStepsMacro+1)*sizeof(double));
          DangleRatio = (double*)malloc((hostTimeStepsMacro+1)*sizeof(double));
          LoopedRatio = (double*)malloc((hostTimeStepsMacro+1)*sizeof(double));

     if (ActiveRatio == NULL)
          printf("ActiveRatio memory error.\n");
     if (DangleRatio == NULL)
          printf("DangleRatio memory error.\n");
     if (LoopedRatio == NULL)
          printf("LoopedRatio memory error.\n");

     //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''


          //int NumberOfActive = 0;
          //int NumberOfDangling = 0;


     //___Calculate Ensemble Average at time = 0___
     struct Stress *Time_k_Stress = NULL;
     struct Stress *Active_Stress = NULL;
     struct Stress *Dangle_Stress = NULL;

     Time_k_Stress = (Stress*)malloc((hostTimeStepsMacro+1)*sizeof(Stress));
     Active_Stress = (Stress*)malloc((hostTimeStepsMacro+1)*sizeof(Stress));
     Dangle_Stress = (Stress*)malloc((hostTimeStepsMacro+1)*sizeof(Stress));
```

```

if (Time_k_Stress == NULL)
    printf("Time_k_Stress memory error.\n");
if (Active_Stress == NULL)
    printf("Active_Stress memory error.\n");
if (Dangle_Stress == NULL)
    printf("Dangle_Stress memory error.\n");


//____ initialize memory _____
for(unsigned int n=0; n < hostTimeStepsMacro+1; n++)
{
    Time_k_Stress[n].XX = 0.0;
    Time_k_Stress[n].XY = 0.0;
    Time_k_Stress[n].YY = 0.0;
    Active_Stress[n].XX = 0.0;
    Active_Stress[n].XY = 0.0;
    Active_Stress[n].YY = 0.0;
    Dangle_Stress[n].XX = 0.0;
    Dangle_Stress[n].XY = 0.0;
    Dangle_Stress[n].YY = 0.0;
}

//''''''''''


//___ Initial Species Count ____
SpeciesRatioCount(hostSpeciesType, &ActiveRatio[0], &DangleRatio[0],
                  &LoopedRatio[0]);
```

```

//____ NEW CODE ____
SpeciesValue *AvgLen;
AvgLen = (SpeciesValue*)malloc((hostTimeStepsMacro+1)*sizeof(SpeciesValue));

SpeciesValue *Variance;
Variance = (SpeciesValue*)malloc((hostTimeStepsMacro+1) *
                sizeof(SpeciesValue));


if (AvgLen == NULL)
    printf("AvgSpringLife_data memory error.\n");
if (Variance == NULL)
    printf("AvgSpringLife_data memory error.\n");

//____ initialize memory _____
for(unsigned int n=0; n < hostTimeStepsMacro+1; n++)
{
    AvgLen[n].ActiveLen = 0.0;
    AvgLen[n].ActiveX = 0.0;
    AvgLen[n].ActiveY = 0.0;
    AvgLen[n].DangleLen = 0.0;
    AvgLen[n].DangleX = 0.0;
    AvgLen[n].DangleY = 0.0;
    AvgLen[n].LoopedLen = 0.0;
    AvgLen[n].LoopedX = 0.0;
    AvgLen[n].LoopedY = 0.0;
    Variance[n].ActiveLen = 0.0;
    Variance[n].ActiveX = 0.0;
```

```
        Variance[n].ActiveY = 0.0;
        Variance[n].DangleLen = 0.0;
        Variance[n].DangleX = 0.0;
        Variance[n].DangleY = 0.0;
        Variance[n].LoopedLen = 0.0;
        Variance[n].LoopedX = 0.0;
        Variance[n].LoopedY = 0.0;
    }

    /*
     * Store Average Spring Life at each time step
     */
    double *AvgSpringLife_data = NULL;

        AvgSpringLife_data = (double*)malloc((hostTimeStepsMacro+1)
                                  * sizeof(double));

    if (AvgSpringLife_data == NULL)
        printf("AvgSpringLife_data memory error.\n");

    //____ initialize memory _____
    for(unsigned int n=0; n < hostTimeStepsMacro+1; n++)
        AvgSpringLife_data[n] = 0.0;

    /*
     * Histogram tracking:
     *
     * Dynamically allocate 2d struct array as points to pointers
     *
```

```
 * Notes: x-axis = bin number
 * y-axis = time
 *
 * Example Active bin2 and time step3: Hist[2][3].Active
 *
 */

int NumOfBins = 100;

SpeciesCount *Hist[100] = {NULL}; //Size should correspond to NumOfBins

for(int i=0; i < NumOfBins; i++){
    Hist[i]=(SpeciesCount *)malloc(sizeof(SpeciesCount) *
                                        (hostTimeStepsMacro+1));

    if (Hist[i] == NULL)
        printf("Hist[%d] memory error.\n",i);

    //____ initialize memory _____
    for(unsigned int n=0; n < hostTimeStepsMacro+1; n++)
    {
        Hist[i][n].Active = 0.0;
        Hist[i][n].Dangle = 0.0;
        Hist[i][n].Looped = 0.0;
    }
}


//''''''''''''''''''''''''
```

```
#ifdef RAW_OUT

    /* Write Dumbbell Raw Data
     *
     * Enables the option to write dumbbell positions to
     * a binary file at a set interval.
     * Filename is the same as the csv file except *bin
     * appended.
     *
     * Notes: Writes 4+8+8 = 20 bytes for each dumbbell.
     * Therefore can quickly result in large files.
     *
     */

    char RawDataFilename[256];

    sprintf(RawDataFilename, "%s.bin", DataFileName);

    FILE *RawDataFilePtr = NULL;





    long int RawData_FileSize; //for checking file size

    if (strcmp(RawData_select,"Yes")==0){

            RawDataFilePtr = fopen(RawDataFilename, "wb");
            if (!RawDataFilePtr){
                printf("Unable to open raw data file!\n");
            }
        }

    #ifdef DEBUG
            printf("DEBUG: RawData Filename = %s\n", RawDataFilename);
            printf("DEBUG: int = %zu double = %zu \n", sizeof(int), sizeof(double));
            printf("DEBUG: hostNumberOfParticles= %zu\n", hostNumberOfParticles);
    #endif
  #endif


            //______To Caclulate Average Length of all Active Dumbbells____

            double *hostAverageSpringLife = NULL;
        double *devAverageSpringLife = NULL;

            hostAverageSpringLife = (double *)malloc(sizeof(double));
            CUDA_CALL(cudaMalloc((double**)&devAverageSpringLife,sizeof(double)));

        if (hostAverageSpringLife == NULL)
            printf("hostAverageSpringLife memory error.\n");

        //____ initialize memory ______
        *hostAverageSpringLife = 0.0;
            //''''''''''''''''''''''''''''''''''''''''''''''''''''''
            //____ Array to record time steps ____
            double *TimeTrack = NULL;

            TimeTrack = (double*)malloc((hostTimeStepsMacro+1)*sizeof(double));

        if (TimeTrack == NULL)
            printf("TimeTrack memory error.\n");

            //_______ Macro Time step Loop _____
            // Main simulation loop

    double FlowRate = 0; /* FlowRate for each stage of simulation */
            double MicroStepSize = 0; /* Allocs two time step sizes */


        /*
         * Time step zero initializations
         */
        EnsembleAverage(hostSpeciesType, hostSpringLenX, hostSpringLenY,
                        Time_k_Stress, Active_Stress, Dangle_Stress, 0);



#ifdef NEW_DNG_LN
        TwoDimSpring *AvgDng;
#endif
```

```
    Spring_AvgLen_XX[0] = Time_k_Stress[0].XX;
    Spring_AvgLen_XY[0] = Time_k_Stress[0].XY;
    Spring_AvgLen_YY[0] = Time_k_Stress[0].YY;


    Detailed_Info(hostSpeciesType, hostSpringLenX, hostSpringLenY,
                    AvgLen, Variance,
#ifdef NEW_DNG_LN
                    AvgDng,
#endif
                    NumOfBins, Hist, 0);

#ifdef NEW_DNG_LN
    printf("The average dangling length is x: %f y: %f \n",
                      AvgDng->x, AvgDng->y);
#endif




        TimeTrack[0]=0.0;

    AvgSpringLife_data[0]=AvgSpringLife(hostSpringLenX, hostSpringLenY,
                            hostSpeciesType);



```

```

#ifdef SPEC_CHNG
    // At time step 0 there are no changes
    Dng2ActSum[0] = 0;
    Dng2LpdSum[0] = 0;
    Act2DngSum[0] = 0;
    Lpd2DngSum[0] = 0;
#endif


    /*
     * Begin main simulation loop
     */

        for (k=1; k<=hostTimeStepsMacro; k++){

#ifdef DEBUG
        printf("DEBUG: Main Loop [%u] ", k);
#endif
                //Calculate Average Length of all Active dumbbells
                AvgSpringLife_data[k] = AvgSpringLife(hostSpringLenX,
                                                      hostSpringLenY,
                                                      hostSpeciesType);


        if (AvgSpringLife_data[k]==0){
            *hostAverageSpringLife = AvgSpringLife_data[0];
            AvgSpringLife_data[k] = AvgSpringLife_data[0];
        } else {
```

```
                    *hostAverageSpringLife = AvgSpringLife_data[k];
            }



                    CUDA_CALL(cudaMemcpy(devAverageSpringLife,
                                        hostAverageSpringLife,
                                        sizeof(double),
                                        cudaMemcpyHostToDevice));

        // set micro time step size based on whether the loop is in stage 1 or
        // stage 2 of the simulation
        /*
         * First stage is designed as zero flow rate. Second stage
         * is the inputed flow rate.
         * Notes: This is a quick fix for implementing the zero flow rate
         * equalizing phase into the simulations.
         *
         */
                    if ( k < hostMacroStepSizeSplitPt){
                                MicroStepSize = hostStepSizeMicroFirst;
                FlowRate = 0;
#ifdef DEBUG
            printf("Stage 1\n");
#endif
                    } else {
                                MicroStepSize = hostStepSizeMicroSecon;
        FlowRate = hostFlowRate;
#ifdef DEBUG
```

```
            printf("Stage 2\n");
#endif
                    }



#ifdef NO_REPORT
    //Time is recorded in the next section when this option is selected
#else
                    //record time
                    TimeTrack[k] = TimeTrack[k−1] + MicroStepSize
                            ∗ hostTimeStepsMicro;
#endif

#ifdef SPEC_CHNG
                    //Call function to perform computations on GPU
                    Micro_Steps<<<grid,block>>>(devSpringLenX, devSpringLenY,
                                            devSpeciesType,
                                            states, ProbStates,
                                            AvgSpringLife_data[k],
                                            devSimTime, MicroStepSize,
                                            hostTimeStepsMicro,
                                            AvgLen[k−1].DangleLen, FlowRate,
                                            devDng2Act, devDng2Lpd,
                                            devAct2Dng, devLpd2Dng);
#else
#ifdef MICRO_RAW
                    //Call function to perform computations on GPU
                    Micro_Steps<<<grid,block>>>(devSpringLenX, devSpringLenY,
                                            devSpeciesType,
                                            states, ProbStates,
                                            AvgSpringLife_data[k],
                                            devSimTime, MicroStepSize,
                                            hostTimeStepsMicro,
                                            AvgLen[k-1].DangleLen, FlowRate,
                                            devSCArr, hostNumberOfParticles);
        //width in bytes -> hostTimeStepsMicro * sizeof(DBSpecChng)
        //height is hostNumberOfParticles

#else
#ifdef NO_REPORT
   /*
        * This option ups the number of Microsteps during a single macro loop.
        * This has the effect of reducing the amount of CPU-GPU communication
        * for the part of the simulation that is not usually used.
        */
            if (k == hostMacroStepSizeSplitPt){
            //record time
                    TimeTrack[k] = TimeTrack[k-1] + MicroStepSize * hostA_coeff;
                //Call function to perform computations on GPU
                Micro_Steps<<<grid,block>>>(devSpringLenX, devSpringLenY,
                                            devSpeciesType,
                                            states, ProbStates,
                                            AvgSpringLife_data[k],
                                            devSimTime, MicroStepSize,
                                            hostA_coeff,
                                            AvgLen[k-1].DangleLen, FlowRate);
            } else {
```

```
        //record time
                TimeTrack[k] = TimeTrack[k−1] + MicroStepSize
                        ∗ hostTimeStepsMicro;
                //Call function to perform computations on GPU
                Micro_Steps<<<grid,block>>>(devSpringLenX, devSpringLenY,
                                        devSpeciesType,
                                        states, ProbStates,
                                        AvgSpringLife_data[k],
                                        devSimTime, MicroStepSize,
                                        hostTimeStepsMicro,
                                        AvgLen[k−1].DangleLen, FlowRate);
        }
#else
#ifdef SINGLE_MICRO
                //Call function to perform computations on GPU
                Micro_Steps<<<grid,block>>>(devSpringLenX, devSpringLenY,
                                        devSpeciesType,
                                        states, ProbStates,
                                        AvgSpringLife_data[k],
                                        devSimTime, MicroStepSize,
                                        hostTimeStepsMicro,
                                        AvgLen[k−1].DangleLen, FlowRate,
                                        devSCArr, hostNumberOfParticles);
    //width in bytes −> hostTimeStepsMicro ∗ sizeof(DBSpecChng)
    //height is hostNumberOfParticles
#else
                //Call function to perform computations on GPU
                Micro_Steps<<<grid,block>>>(devSpringLenX, devSpringLenY,
                                        devSpeciesType,
                                        states, ProbStates,
                                        AvgSpringLife_data[k],
                                        devSimTime, MicroStepSize,
                                        hostTimeStepsMicro,
                                        AvgLen[k−1].DangleLen, FlowRate);
#endif //SINGLE_MICRO
#endif //NO_REPORT
#endif //MICRO_RAW
#endif //SPEC_CHNG




                  //read result from gpu(device) back to cpu(host)
                  CUDA_CALL(cudaMemcpy(hostSpringLenX, devSpringLenX,
                              hostNumberOfParticles*sizeof(double),
                              cudaMemcpyDeviceToHost));
                  CUDA_CALL(cudaMemcpy(hostSpringLenY, devSpringLenY,
                              hostNumberOfParticles*sizeof(double),
                              cudaMemcpyDeviceToHost));
                  CUDA_CALL(cudaMemcpy(hostSpeciesType, devSpeciesType,
                              hostNumberOfParticles*sizeof(int),
                              cudaMemcpyDeviceToHost));

                  //read sim time back from gpu(device) back to cpu(host)
                  CUDA_CALL(cudaMemcpy(hostSimTime, devSimTime,
                              hostNumberOfParticles*sizeof(double),
                              cudaMemcpyDeviceToHost));

```

```
#ifdef SPEC_CHNG
    //read species transitions back from gpu
                CUDA_CALL(cudaMemcpy(hostDng2Act, devDng2Act,
                            hostNumberOfParticles*sizeof(unsigned int),
                            cudaMemcpyDeviceToHost));
                CUDA_CALL(cudaMemcpy(hostDng2Lpd, devDng2Lpd,
                            hostNumberOfParticles*sizeof(unsigned int),
                            cudaMemcpyDeviceToHost));
                CUDA_CALL(cudaMemcpy(hostAct2Dng, devAct2Dng,
                            hostNumberOfParticles*sizeof(unsigned int),
                            cudaMemcpyDeviceToHost));
                CUDA_CALL(cudaMemcpy(hostLpd2Dng, devLpd2Dng,
                            hostNumberOfParticles*sizeof(unsigned int),
                            cudaMemcpyDeviceToHost));

    //call function that sums the values
    Dng2ActSum[k] = VectorSum(hostDng2Act,hostNumberOfParticles);
    Dng2LpdSum[k] = VectorSum(hostDng2Lpd,hostNumberOfParticles);
    Act2DngSum[k] = VectorSum(hostAct2Dng,hostNumberOfParticles);
    Lpd2DngSum[k] = VectorSum(hostLpd2Dng,hostNumberOfParticles);

#endif

#ifdef SINGLE_MICRO

    //transfer data back from GPU
    CUDA_CALL(cudaMemcpy(hostSCArr, devSCArr,
                        hostTimeStepsMicro * sizeof(DBSpecChng),
                        cudaMemcpyDeviceToHost));

    //check file size, error if too big
    MicroData_FileSize = ftell(MicroDataFilePtr);


      for(unsigned int m=0; m < hostTimeStepsMicro; m++){

        if (MicroData_FileSize < MICRODATA_MAX_FILESIZE){
          fwrite( &(hostSCArr[m].type),sizeof(int), 1, MicroDataFilePtr);
          fwrite( &(hostSCArr[m].time),sizeof(double),1, MicroDataFilePtr);
          fwrite( &(hostSCArr[m].x), sizeof(double),1, MicroDataFilePtr);
          fwrite( &(hostSCArr[m].y), sizeof(double),1, MicroDataFilePtr);
        } else {
          printf("WARNING: Micro data file size exceeded maximum. No longer ",
                                "writing to file. \n");
        }
      }


#endif

#ifdef MICRO_RAW

    //transfer data back from GPU
    CUDA_CALL(cudaMemcpy(hostSCArr, devSCArr,
                  hostNumberOfParticles * hostTimeStepsMicro * sizeof(DBSpecChng),
                  cudaMemcpyDeviceToHost));

    //check file size, error if too big
    MicroData_FileSize = ftell(MicroDataFilePtr);


    for(unsigned int n=0; n < hostNumberOfParticles; n++){
      for(unsigned int m=0; m < hostTimeStepsMicro; m++){

        if (MicroData_FileSize < MICRODATA_MAX_FILESIZE){
          fwrite( &(n),
                          sizeof(unsigned int), 1, MicroDataFilePtr);
          fwrite( &(hostSCArr[n*hostTimeStepsMicro+m].type),
                          sizeof(int), 1, MicroDataFilePtr);
          fwrite( &(hostSCArr[n*hostTimeStepsMicro+m].length),
                          sizeof(double), 1, MicroDataFilePtr);
        } else {
          printf("WARNING: Micro data file size exceeded maximum. No longer ",
                          "writing to file. \n");
        }
      }
    }


#endif


        EnsembleAverage(hostSpeciesType, hostSpringLenX, hostSpringLenY,
                    Time_k_Stress, Active_Stress, Dangle_Stress,k);

                Spring_AvgLen_XX[k] = Time_k_Stress[k].XX;
                Spring_AvgLen_XY[k] = Time_k_Stress[k].XY;
                Spring_AvgLen_YY[k] = Time_k_Stress[k].YY;
        //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''


        SpeciesRatioCount(hostSpeciesType, &ActiveRatio[k], &DangleRatio[k],
                          &LoopedRatio[k]);

        //____ NEW CODE ____
        Detailed_Info (hostSpeciesType, hostSpringLenX, hostSpringLenY,
                    &AvgLen[k], &Variance[k],
                    NumOfBins, Hist, k);


#ifdef RAW_OUT

        /* Write file output directly to file */

        #ifdef SIMPLE_SHEAR
        if ((strcmp(RawData_select,"Yes")==0) && RawOut_SSFlow(k))
        #else

         /*
          * FULL_DATA option to allow for 800 steps over entire
          * Oscillatory shear simulation
          */

        #ifdef FULL_DATA
        if ((strcmp(RawData_select,"Yes")==0) && RawOut_SSFlow(k))
        #else
        //default is oscillatory shear
        if ((strcmp(RawData_select,"Yes")==0) && RawOut_OSFlow(k))
        #endif
        #endif
        {

        //check file size, error if too big
        RawData_FileSize = ftell(RawDataFilePtr);



#ifdef DEBUG
            printf("DEBUG: Current file size: %ld\n", RawData_FileSize);
            printf("DEBUG: Writing to file: %s on step: %d at time: %f\n",
                            RawDataFilename, k, TimeTrack[k]);

#endif
          if (RawData_FileSize < RAWDATA_MAX_FILESIZE){


            fwrite(&(TimeTrack[k]),sizeof(double), 1,
                            RawDataFilePtr);
            fwrite(hostSpeciesType,sizeof(int) ,hostNumberOfParticles,
                            RawDataFilePtr);
            fwrite(hostSpringLenX ,sizeof(double),hostNumberOfParticles,
                            RawDataFilePtr);
            fwrite(hostSpringLenY ,sizeof(double),hostNumberOfParticles,
                            RawDataFilePtr);
          } else {
                printf("WARNING: Raw data file size exceeded %f bytes. No longer ",
                                "writing to file.\n", RAWDATA_MAX_FILESIZE);
              }
            }

#endif
            }
            //''''''''''''End Macro loop''''''''''''

#ifdef RAW_OUT

    if (strcmp(RawData_select,"Yes")==0){
      fclose(RawDataFilePtr);
    }

#endif


#ifdef MICRO_RAW
   fclose(MicroDataFilePtr);
#endif

#ifdef SINGLE_MICRO
   fclose(MicroDataFilePtr);
#endif

            // ___ stop computational clock _____
            end = clock();
            time_spent = double(end-begin)/ CLOCKS_PER_SEC;
```

```
          //''''''''''''''''''''''''''''''''''''


#ifdef SPEC_CHNG
          OutputToFile( Spring_AvgLen_XX, Spring_AvgLen_XY, Spring_AvgLen_YY,
                         TimeTrack, time_spent, k, argv[0],
                         ActiveRatio, DangleRatio, LoopedRatio,
                         AvgLen, Variance,
                         NumOfBins, Hist,
                         Time_k_Stress, Active_Stress, Dangle_Stress,
                         AvgSpringLife_data,
                         Dng2ActSum, Dng2LpdSum, Act2DngSum,
                         Lpd2DngSum,
                         DataFileName);

#else

          OutputToFile( Spring_AvgLen_XX, Spring_AvgLen_XY, Spring_AvgLen_YY,
                         TimeTrack, time_spent, k, argv[0],
                         ActiveRatio, DangleRatio, LoopedRatio,
                         AvgLen, Variance,
                         NumOfBins, Hist,
                         Time_k_Stress, Active_Stress, Dangle_Stress,
                         AvgSpringLife_data,
                         DataFileName);
          //''''''''''''''''''''''''''''''''
#endif


```

```
/*
 * Memory freed in the order it was initialized.
 *
 */


        CUDA_CALL(cudaFree(states));
        CUDA_CALL(cudaFree(ProbStates));


        free(hostSeeds);
        free(hostProbSeeds);


        CUDA_CALL(cudaFree(devSeeds));
        CUDA_CALL(cudaFree(devProbSeeds));

        free(hostSpringLenX);
        free(hostSpringLenY);
        free(hostSpeciesType);

        CUDA_CALL(cudaFree(devSpringLenX));
        CUDA_CALL(cudaFree(devSpringLenY));
        CUDA_CALL(cudaFree(devSpeciesType));


        free(hostSimTime);
        CUDA_CALL(cudaFree(devSimTime));

        free(Spring_AvgLen_XX);
        free(Spring_AvgLen_XY);
        free(Spring_AvgLen_YY);

        free(ActiveRatio);
        free(DangleRatio);
        free(LoopedRatio);

  free(Time_k_Stress);
  free(Active_Stress);
  free(Dangle_Stress);

  free(AvgLen);

  free(Variance);

  free(AvgSpringLife_data);

  for(int i=0; i<NumOfBins; i++)
      free(Hist[i]);

#ifdef SPEC_CHNG

    free(hostDng2Act);
    free(hostDng2Lpd);
    free(hostAct2Dng);
    free(hostLpd2Dng);

    CUDA_CALL(cudaFree(devDng2Act));
    CUDA_CALL(cudaFree(devDng2Lpd));
    CUDA_CALL(cudaFree(devAct2Dng));
    CUDA_CALL(cudaFree(devLpd2Dng));

    free(Dng2ActSum);
    free(Dng2LpdSum);
    free(Act2DngSum);
    free(Lpd2DngSum);

#endif

#ifdef SINGLE_MICRO

    free(hostSCArr);
    CUDA_CALL(cudaFree(devSCArr));

#endif
#ifdef MICRO_RAW

    free(hostSCArr);
    CUDA_CALL(cudaFree(devSCArr));

#endif



        free(hostAverageSpringLife);
        CUDA_CALL(cudaFree(devAverageSpringLife));

        free(TimeTrack);

        //''''''''''''''''''''''''''''''''''''''''''''''''


        cudaDeviceReset();


        //___ stop computational clock _____
        end2 = clock();
        time_spent2 = double(end2-begin)/ CLOCKS_PER_SEC;
        printf("Runtime: %f\n\n", time_spent2);
        //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''


        return EXIT_SUCCESS;

}
```