SIMULATIONS OF SUSPENSIONS OF BROWNIAN SPHEROCYLINDERS AND NON-BROWNIAN LINKED SPHEROCYLINDERS

by

JING-YAO CHEN

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy
(Chemical Engineering)

at the
UNIVERSITY OF WISCONSIN-MADISON
2020

Date of final oral examination: August 18, 2020

The dissertation is approved by the following members of the Final Oral Committee:

Daniel J. Klingenberg, Professor, Chemical and Biological Engineering
Michael D. Graham, Professor, Chemical and Biological Engineering
Thatcher W. Root, Professor, Chemical and Biological Engineering
Reid Van Lehn, Assistant Professor, Chemical and Biological Engineering
Izabela Szlufarska, Professor, Material Science Engineering

To my mom.

ACKNOWLEDGEMENTS

I would like to sincerely thank my adviser Prof. Daniel J. Klingenberg for providing insight into research, caring for my well-being, popping into the office to say hi, inviting us over for Thanksgiving, and offering to help me move. Without his guidance, I would not be able to accomplish this work and have an enjoyable time at Madison. I would additionally like to thank Prof. Michael D. Graham, Prof. Thatcher W. Root, Prof. Reid C. Van Lehn, and Prof. Izabela Szlufarska for serving on my defense committee.

I would like to thank my group members Jianeng Wang for lots of research discussions and letting me crash in the living room, Kevin Frankforter for fun conversations and insightful questions, Shalaka Burlawar for providing experimental backgrounds and feedback on my work, Joshua Duncan for inviting us over for games, and Ben Wilson for introducing me to CUDA and giving me career advice when I first joined the group. Additionally, I would like to thank my collaborator Zhuohan Li for asking insightful questions and offering a different perspective.

Through this journey, I was accompanied by my amazing batchmates, Saurabh, Ranjeet, Apoorva, Yifu, Ashwin, Huicheng, Jordan, and Coogan. Knowing that we all strive towards the same goal carried me through the burdens of assignments, projects, and research bottlenecks. I have learned so much about the world through my interactions with them. I would like to thank Anubhav, Sandy, Frank, Sarit, Alec, Kevin, Eric, and RJ for creating such a fun office environment. I will really miss the random nerve gun episodes and loud broadcasts along with racing whenever someone spots free food.

During my time here, I was fortunate to find a loving church family. I would like to thank Katie, Rochelle, Emily, Phoebe, Amy, Michelle, Charlene, Chelsea, Becky, Rachel,

Tia, Andrew, Sid, Preston, Tim, Phil, Derek, Jacob, Rob, and Aaron for walking with me through the highs and lows.

Lastly, I want to thank my mom and stepfather for unwavering belief in me and giving me an opportunity to pursue studies in the United States, even when I was just a hot-headed teenager who didn't really know what I had gotten myself into.

The research presented in this thesis was in part supported by a Vilas Life Cycle Professorship and the PPG Industries 2018 summer fellowship.

Jing-Yao Chen

Madison, WI

August 2020

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Fiber suspensions, both naturally occuring and synthetic, have many of applications, such as rheological modifiers and composite reinforcement. The rheological properties strongly depend on the microstructure, which varies with fiber properties, concentration, and processing methods. Rigid fiber suspensions form liquid crystalline phases. Flexible fiber suspensions form homogeneous networks and aggregates. In this thesis, fiber-level simulations were applied to systematically investigate the relationship between the microstructure and macroscopic properties, including the viscosity, normal stress differences, and diffusivities. Simulation programs were accelerated and parallelized for GPUs via CUDA.

Rigid fibers were modeled as spherocylinders that interacted only through a soft repulsive force. Brownian dynamics simulations were employed to obtain the translational and rotational diffusivities, matching reported values of hard spherocylinder suspensions. Liquid crystalline phases, including nematic, smectic, and solid phases, were obtained. For suspensions that were isotropic at rest, flow curves, which contained two shear thinning regions bracketing a viscosity plateau at intermediate Péclet numbers, qualitatively matched those for suspensions of cellulose nanocrystals. For suspensions that were nematic at rest, system-wide domains that aligned and kayaked about the vorticity direction, domains that rotated in the gradient direction, and layered domains were observed under shear. The transient rheological properties depended on the domain dynamics.

Flexible fibers were modeled as spherocylinders that were connected by joints with bending and twisting potentials. The fibers were non-Brownian, and interacted through

soft repulsion, short-ranged attraction, and friction. A drying and rehydration process was implemented. The volume fraction of the suspensions were raised 4 fold and subsequently lowered to the original value. The simulation box was shrunk and expanded with a constant number of fibers. During drying, the fibers were moved affinely after the equations of motion were integrated. When sufficient friction and attractive forces were applied, the viscosities for suspensions that were dried and rehydrated were lower than those for suspensions that were not dried and rehydrated. The reduction of viscosity was associated with the formation of dense and persistent flocs, which were observed experimentally for microfibrillated cellulose suspensions.

1

INTRODUCTION

1.1 Motivation and Overview

Fiber suspensions are prevalent in nature and often synthesized for a wide range of applications, such as composite reinforcement and rheological modifiers. The microstructure of fiber suspensions largely depends on the fiber properties; short and rigid fibers form liquid crystalline phases while long and flexible fibers form flocs. The goal of this study is to systematically investigate the relationship between fiber properties, the microstructure, and macroscopic properties for different fiber types. This is accomplished by fiber-level simulations of two models: Brownian rigid spherocylinders and non-Brownian linked-rigid spherocylinders.

This dissertation is organized as follows.

Chapter 1 – Introduction. The rest of the chapter provides background on the dynamics of isolated fibers and fibers in suspension. The rheology of rod-like fiber and flexible fiber suspensions are summarized with relevant experimental and simulations results. A literature survey of cellulose nanomaterials, which have garnered interest for applications

in renewable materials, is included; experimental results for these systems are compared with simulation results in later chapters.

Chapter 2 – Simulation methods. The two fiber models and relevant computational algorithms are presented.

Chapter 3 – Rheology and structure of suspensions of spherocylinders via Brownian dynamics simulations. Brownian dynamics simulations are employed to investigate the rheological properties and structure of suspensions of rigid spherocylinders, as a model for rod-like colloids. The spherocylinders interact only through a soft repulsive force that mimics a hard spherocylinder interaction. The translational and rotational diffusivities of hard spherocylinder suspensions are reproduced. Liquid crystalline phases, including isotropic, nematic, smectic, and solid phases, are identified using orientational and hexatic order parameters, and pair distribution functions. Typical flow curves observed experimentally for rod-like colloidal suspensions are reproduced in the simulations, with two shear thinning regions that bracket a viscosity plateau at intermediate Péclet numbers. The transient rheology and structure of suspensions that are nematic at rest exhibit a variety of behaviors that depend on the Péclet number and concentration. System-wide domains that align in and kayak about the vorticity direction, domains that rotate coherently locally, and layered domains are observed. Oscillations in the order parameter, viscosity, and the first and second normal stress differences correlate with the structure.

Chapter 4 – Effect of Drying and Rehydration on the Rheological Properties and Structure of Suspensions of Linked Spherocylinders. Fiber-level simulations are employed to investigate the effect of drying and rehydration on the rheological properties and structure of suspensions of flexible fibers. Each fiber consists of 5 rigid spherocylinders, connected by joints with bending and twisting potentials. Fibers interact through friction, and normal repulsive and attractive forces. The drying and rehydration process first raises the

volume fraction 4 fold from 0.003. During drying, the fibers are moved affinely with the simulation box after the equations of motion are integrated. Then, the suspensions are rehydrated by expanding the simulation box. When the friction and attractive forces are sufficiently large, suspensions that were dried and rehydrated have lower viscosity than those that were not dried and rehydrated. The reduction in viscosity is associated with the formation of dense and persistent aggregates, which have been observed eperimentally for microfibrillated cellulose suspensions.

Chapter 5 – Conclusions and future directions. A summary of the main results for both models is presented, along with recommendations for future research.

Appendices. Inertia is ignored in the equations of motion for the models used for results reported in this thesis. To validate the inertialess model, the translatonal and rotational diffusivities of suspensions of Brownian spherocylinders are compared for the inertial and inertialess models in Appendix A. Simulation programs are parallelized to run on the GPU using the CUDA language. An example procedure to optimize the performance of a CUDA program is presented in Appendix B. Codes used to simulate suspensions of Brownian spherocylinders and non-Brownian linked spherocylinders are included in Appendix C.

1.2 Single Fiber Dynamics

The motion of a fiber in suspension is strongly impacted by the suspending fluid. The velocity of the suspending fluid, v, is governed by the continuity equation and equation

of motion, which is called the Cauchy momentum equation (Graham, 2018),

$$\frac{\partial \rho}{\partial t} + \boldsymbol{\nabla} \cdot (\rho \boldsymbol{v}) = 0 \tag{1.1}$$

$$\rho \left(\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \boldsymbol{\nabla} \boldsymbol{v} \right) = \boldsymbol{\nabla} \cdot \boldsymbol{\sigma} + \boldsymbol{f} \tag{1.2}$$

where ρ is the density, $\boldsymbol{\sigma}$ is the stress, and $\boldsymbol{f}$ represents external body force densities, such as gravity. For incompressible Newtonian fluids,

$$\boldsymbol{\nabla} \cdot \boldsymbol{v} = 0 \tag{1.3}$$

With constant density, $\boldsymbol{\sigma}$ is

$$\boldsymbol{\sigma} = -p\boldsymbol{\delta} + \eta_0 \left[\boldsymbol{\nabla} \boldsymbol{v} + (\boldsymbol{\nabla} \boldsymbol{v})^T \right] \tag{1.4}$$

where p is the isotropic pressure and η_0 is the viscosity of the suspending fluid. Equation 1.2 can be non-dimensionalized for an incompressible fluid with constant ρ and η_0

$$\mathrm{Re} \left(\mathrm{Sr}^{-1} \frac{\partial \boldsymbol{v}^*}{\partial t^*} + \boldsymbol{v}^* \cdot \boldsymbol{\nabla} \boldsymbol{v}^* \right) = -\nabla p^* + \boldsymbol{\nabla}^2 \boldsymbol{v}^* + \boldsymbol{f}^*. \tag{1.5}$$

where the asterisk indicates nondimensionalized variables. The Reynolds (Re) and Strouhal (Sr) numbers are dimensionless groups defined

$$\mathrm{Re} = \frac{\rho U L}{\eta_0} \tag{1.6}$$

$$\mathrm{Sr} = \frac{TU}{L} \tag{1.7}$$

where T, L, and U are the time, length, and velocity scales characteristic to the fiber. If there is no imposed time scale, $T = L/U$ and $\mathrm{Sr} = 1$. For sufficiently small particles, Re

$\ll 1$ and $\mathrm{ReSr}^{-1} \ll 1$. With inertia neglected, Eq. 1.1 reduces to

$$-\nabla p + \eta_0 \boldsymbol{\nabla}^2 \boldsymbol{v} + \boldsymbol{f} = 0 \tag{1.8}$$

for an incompressible Newtonain fluid with constant ρ and η_0. Equation 1.8 is referred to as the Stokes equation, which is linear and satisfies reversibility. The properties of the Stokes equation allow the hydrodynamic force, $\boldsymbol{F}$, torque, $\boldsymbol{T}$, and stresslet, $\boldsymbol{S}$, on the fiber to depend on the fiber velocity, $\dot{\boldsymbol{r}}$, angular velocity, $\boldsymbol{w}$, and rate of strain, $\boldsymbol{E}^\infty$, according to

$$\begin{pmatrix} \boldsymbol{F} \\ \boldsymbol{T} \\ \boldsymbol{S} \end{pmatrix} = \begin{bmatrix} \boldsymbol{A} & \tilde{\boldsymbol{B}} & \tilde{\boldsymbol{G}} \\ \boldsymbol{B} & \boldsymbol{C} & \tilde{\boldsymbol{H}} \\ \boldsymbol{G} & \boldsymbol{H} & \boldsymbol{M} \end{bmatrix} \begin{pmatrix} \boldsymbol{U}^\infty - \dot{\boldsymbol{r}} \\ \boldsymbol{\Omega}^\infty - \boldsymbol{w} \\ \boldsymbol{E}^\infty \end{pmatrix} \tag{1.9}$$

where $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ are second-rank resistance tensors, $\boldsymbol{G}$ and $\boldsymbol{H}$ are third-rank resistance tensors, and $\boldsymbol{M}$ is a fourth-rank resistance tensor (Kim and Karilla, 1991). The resistance tensors depend only on the particle geometry. The block matrix is the grand resistance tensor; its inverse is the grand mobility tensor. The tilde notation indicates symmetry in the tensors, where $B_{ij} = \tilde{B}_{ji}$, $G_{ijk} = \tilde{G}_{kij}$, and $H_{ijk} = \tilde{H}_{kij}$. The ambient angular velocity and rate of strain tensor are related to the ambient velocity, $\boldsymbol{U}^\infty$, by

$$\boldsymbol{\Omega}^\infty = \frac{1}{2} \nabla \times \boldsymbol{U}^\infty \tag{1.10}$$

$$\boldsymbol{E}^\infty = \frac{1}{2} \left[\nabla \boldsymbol{U}^\infty + (\nabla \boldsymbol{U}^\infty)^T \right] \tag{1.11}$$

1.2.1 *Jeffery Orbits*

Rigid ellipsoids under creeping shear flow rotate and spin, and the major axis traces out a family of closed orbits. In polar coordinates (see Fig. 1.1), the azimuthal angle, $\phi \in [0, 2\pi)$, and polar angle, $\theta \in [0, \pi]$, define the orientation of the major axis of the ellipsoid, $\boldsymbol{p} = (\sin\theta \sin\phi, \sin\theta \cos\phi, \cos\theta)$. Jeffery (1922) derived expressions for the

periodic motion,

$$\tan\theta = \frac{Cr_e}{\sqrt{r_e^2\cos^2\phi + \sin^2\phi}} \tag{1.12}$$

$$\tan\phi = r_e \tan\left(2\pi\frac{t}{T}\right) \tag{1.13}$$

where r_e is ratio of the major and minor axis, C is the orbit constant, t is time, and T is the orbit period. The orbit constant is $C \in [0, \infty]$. At $C = 0$, the ellipsoid is oriented along the vorticity direction, and undergoes a log-rolling motion. As $C \to \infty$, the ellipsoid tumbles in the plane of shear. The orbit period is

$$T = \frac{2\pi}{\dot{\gamma}}\left(r_e + \frac{1}{r_e}\right) \tag{1.14}$$

where $\dot{\gamma}$ is the shear rate.

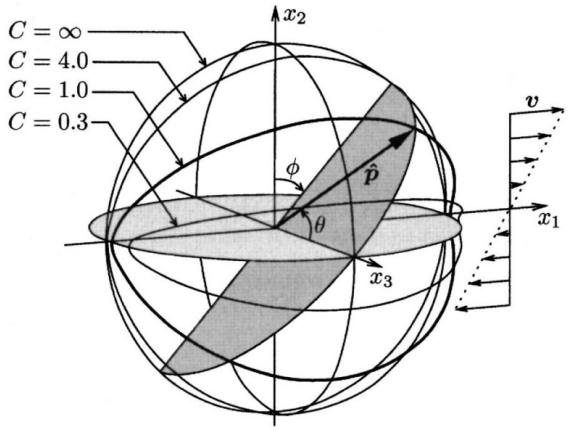


Figure 1.1: Schematic diagram of Jeffery orbits (Lindström and Uesaka, 2008a).

Bretherton (1962) found that the rotational motion of any body of revolution with fore-aft symmetry is periodic in shear flow. The orbit period is obtained by replacing r_e in Eq. 1.14 by an equivalent aspect ratio that depends on the shape of the body. Goldsmith and Mason (1962) verified experimentally that the orbit period of spheres, rods, and discs agree with Jeffery orbits under shear flow when equivalent aspect ratios are used. Cox

(1971) provided an expression for the equivalent aspect ratio, r_e, for cylinders with the aspect ratio defined as $r_p = L/D$, where L is the length, and D is the diameter,

$$r_e = 1.24 \frac{r_p}{\sqrt{\ln r_p}} \tag{1.15}$$

Trevelyan and Mason (1951) experimentally determined that $0.52 < r_e/r_p < 0.72$ for cylinders with $17.8 \leq r_p \leq 132$.

Unlike rigid fibers, which maintain the same shape and strictly follow the Jeffery orbits, flexible fibers tend to deform and their rotational motions deviate from Jeffery orbits. Under shear flow, the elongation and compressive forces on a fiber are maximum at $\pm 45^\circ$ from the direction of flow in the plane of shear. If the compressive force exceeds the critical stress, the straight fiber buckles. The critical shear stress for buckling was estimated as (Forgacs and Mason, 1959; Switzer, 2002).

$$(\dot{\gamma}\eta_0)_{\text{crit}} \simeq \frac{E_b \left[\ln(2r_p) - 1.75\right]}{2r_p^4} \tag{1.16}$$

for isolated fibers, where $E_b \approx 2E_Y$ is the bending modulus, and E_Y is the Young's modulus. Long and flexible fibers have a smaller critical stress and tend to buckle more easily.

1.2.2 *Nonaxisymmetric Fibers*

Nonaxisymmetric rigid particles exhibit more complex rotational behavior than Jeffery orbits. Hinch and Leal (1979) showed that the rotational motion of an isolated, non-Brownian, and non-axisymmetric rigid ellipsoid in simple shear flow at low Reynolds number exhibits a doubly periodic structure. Using slender body theory to calculate the resistance tensors, Kim and Rae (1991) found that a helix drifts in the vorticity direction in shear flow at low Reynolds numbers. In combination with gravity, this drift can be used to separate dilute racemic solutions. Wang et al. (2012a) investigated the dynamics of a rigid curved fiber. In shear flow at low Reynolds number, the fibers can drift in the

gradient direction via a flipping, scooping, and spinning mechanism for some initial configurations. The drift contributes to the diffusivity in addition to shear-induced diffusion discussed later in Sec. 1.3.1.

Lundell (2011) and Yarin et al. (1997) investigated the effect of inertia on the motion of an inertial and triaxial ellipsoid in shear flow. The ellipsoid drifted by rotating around the shortest axis, which aligned in the vorticity direction. At intermediate Stokes numbers, which quantifies the inertia, the ellipsoid underwent chaotic motion.

1.2.3 *Brownian Motion*

Brownian motion is the random motion that a particle undergoes in fluid from collisions with solvent molecules. The particle also experiences hydrodynamic resistance to motion. While the random force and torque are normally distributed with zero means, the fluctuation-dissipation theorem expresses the force, $\boldsymbol{F}^{\mathrm{Br}}$, and torque, $\boldsymbol{T}^{\mathrm{Br}}$, autocorrelations as (Kubo, 1966).

$$\left\langle \boldsymbol{F}^{\mathrm{Br}}(t)\, \boldsymbol{F}^{\mathrm{Br}}(t+\tau) \right\rangle = 2\boldsymbol{A} k_{\mathrm{B}} T \delta(\tau) \tag{1.17}$$

$$\left\langle \boldsymbol{T}^{\mathrm{Br}}(t)\, \boldsymbol{T}^{\mathrm{Br}}(t+\tau) \right\rangle = 2\boldsymbol{C} k_{\mathrm{B}} T \delta(\tau) \tag{1.18}$$

where k_{B} is the Boltzmann constant, T is the temperature, and δ is the Dirac-Delta function. The resistance tensors, $\boldsymbol{A}$ and $\boldsymbol{C}$, are the those used in the grand resistance tensor in Eq. 1.9.

Molecular dynamics and Brownian dynamics simulations can be utilized to investigate Brownian motion (Chen and Kim, 2004). Molecular dynamics simulations include the Brownian particles and the surrounding fluid molecules. Using the perturbation theory, Kubo et al. (1957) showed that dynamic properties, such as the diffusivity, can be extracted from the time correlations of the particle velocities. The computational cost is reduced for Brownian dynamics simulations, which account for the motion of the fluid

molecules by random forces and torques statistically satisfying Eq. 1.17 and 1.18. For isolated and axisymmetric particles, the generalized Langevin equations for translational and rotational motion for low Reynolds number flow, where the drag and velocities are linearly correlated, are (Mori, 1965; Hubbard, 1972; Ford et al., 1977; Avalos, 1996; Zwanzig, 2001)

$$m\frac{d^2\boldsymbol{x}}{dt^2} = -\boldsymbol{A}\cdot\frac{d\boldsymbol{x}}{dt} + \boldsymbol{F}^{\mathrm{Br}} \tag{1.19}$$

$$\boldsymbol{I}\cdot\frac{d\boldsymbol{w}}{dt} + \boldsymbol{w}\times(\boldsymbol{I}\cdot\boldsymbol{w}) = -\boldsymbol{C}\cdot\boldsymbol{w} + \boldsymbol{T}^{\mathrm{Br}} \tag{1.20}$$

where m is the particle mass, $\boldsymbol{I}$ is the moment of inertia, $\boldsymbol{x}$ is the position, and $\boldsymbol{w}$ is the angular velocity.

The equations of motions are integrated to obtain the motion of the Brownian particles. The forces and torques are not continuous; the trajectory from one simulation is not reproduced by simulations using a smaller time step starting from the same position. The velocity autocorrelation follows from the equations of motion (Zwanzig, 2001)

$$\langle\dot{\boldsymbol{x}}(t)\,\dot{\boldsymbol{x}}(t+\tau)\rangle = \frac{k_{\mathrm{B}}T}{m}\exp\left(-\frac{\boldsymbol{A}}{m}\tau\right), \tag{1.21}$$

which satisfies equipartition at $\tau = 0$. The mean-square displacement is

$$\langle\boldsymbol{x}\boldsymbol{x}\rangle = 2k_{\mathrm{B}}T\boldsymbol{A}^{-1}t - 2mk_{\mathrm{B}}T\boldsymbol{A}^{-2}\left[\delta - \exp\left(-\frac{\boldsymbol{A}}{m}t\right)\right] \tag{1.22}$$

At short times, $\langle\boldsymbol{x}^2\rangle$ increases quadratically with time because of particle inertia. At long times, the noise from random motion dominates and the mean-square displacement becomes

$$\langle\boldsymbol{x}\boldsymbol{x}\rangle \to 2k_{\mathrm{B}}T\boldsymbol{A}^{-1}t \tag{1.23}$$

The Stokes−Einstein equation relates the translational and rotational diffusivities to the

resistance tensors in low Reynolds number flow

$$\boldsymbol{D}_T = k_B T \boldsymbol{A}^{-1} \tag{1.24}$$

$$\boldsymbol{D}_R = k_B T \boldsymbol{C}^{-1} \tag{1.25}$$

The Langevin equation can be made inertialess by setting m and $\boldsymbol{I}$ to zero. The inertialess equations are singular since the highest derivatives are 0. The issue is circumvented by adapting suitable integration schemes, such as the midpoint method (Fixman, 1978), or adding a pseudo potential (Grassia and Hinch, 1996).

1.3 Fiber Suspensions

The previous section summarizes the dynamics of a single fiber in suspension. This section includes relevant phenomena for fiber suspensions, in which fiber-fiber and fiber-solvent interactions impact the structure, diffusivities, and rheological properties. Shear-induced diffusion arises from fiber collisions. Rod-like fibers form liquid crystalline phases; flexible fibers entangle and form flocs. The impact of the fiber properties on the structure and rheological properties is summarized with relevant experimental and simulation studies.

1.3.1 *Shear-Induced Diffusion*

Shear-induced diffusion arises from interparticle collisions in shear flow. Shear-induced diffusion dominates over the Brownian diffusivity of the particles at large Péclet number (Koch, 1989), which is the rate of transport from advection over that from molecular diffusion. Shear-induced diffusion contributes to the structure of suspensions of red blood cells (Goldsmith and Marlow, 1979; Tang et al., 2018) and other phenomena such as viscous resuspension (Leighton and Acrivos, 1986), where settled particles in viscous suspensions

are homogenized under shear flow.

Eckstein et al. (1977) first experimentally measured the shear-induced diffusivity for spherical and disk-like particles in Couette flow and found that the lateral dispersion increases linearly with concentration up to a volume fraction of 0.2. Leighton and Acrivos (1987) experimentally determined that the diffusion of spheres in the direction of the shear gradient is proportional to $a^2\dot{\gamma}$, where a is the radius and $\dot{\gamma}$ is the shear rate. Acrivos et al. (1992) derived the diffusivity of spheres under shear flow at vanishing Reynolds number to be $0.267a^2\dot{\gamma}\left[c\ln c^{-1}+\mathcal{O}\left(c\right)\right]$ in the direction of flow, where c is the concentration.

Shear-induced diffusion is more evident in the vorticity and gradient directions; mass transport in the flow direction is dominated by convection. The diffusivities for nonspherical particles are larger than those for spherical particles. When the fiber-fiber repulsion has a range smaller than 10^{-6} of the particle radius, nonspherical particles have shear-induced diffusivites up to 5 times larger than those for spheres (Lopez and Graham, 2007). Wang et al. (2014) simulated linked rigid rod and bead chain models without hydrodynamic interactions. The shear-induced diffusivities increase with concentration and the coefficient of static friction between fibers and are larger for curved fibers than for straight fibers.

1.3.2 *Suspensions of Rigid Rod-like Fibers*

Rod-like colloidal particles, here used interchangeably with rod-like fibers and rod-like polymers, are abundant in nature (Solomon and Spicer, 2010), such as the Tobacco Mosaic Virus (Fraden et al., 1989), cellulose nanocrystals (Klemm et al., 2018), and bimetallic rods (Hong et al., 2007). Rod-like colloidal particles are also synthesized, such as polymethyl methacrylate elipsoids with controlled aspect ratio (Mohraz and Solomon, 2005), which show jamming and orientational ordering. Images of natural and synthetic rod-like particles of different scales are presented in Fig. 1.2. Solutions of rigid rod polymers tend to

form liquid crystalline phases, and thus have complex rheology. This section explores the structure and rheological properties along with relevant simulation studies.

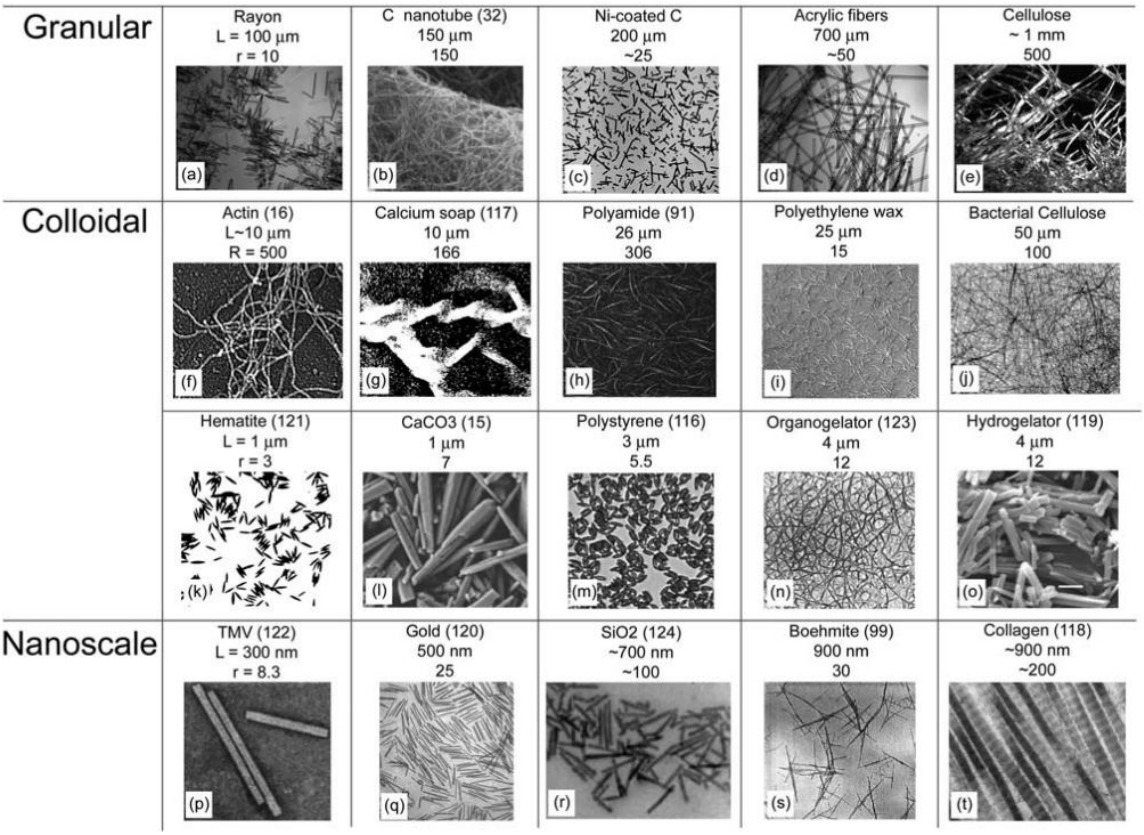


Figure 1.2: Rod-like particles of different scales from Solomon and Spicer (2010). Numbers within the parenthesis refer to references in Solomon and Spicer (2010).

Liquid Crystalline Phases

Rigid rod-like colloidal particles tend to form liquid crystalline suspensions, which combine the properties of fluids and solids. A liquid crystalline suspension flows like a fluid and forms phases that can have orientational and/or translational order under some conditions. Common mesophases, illustrated in Fig. 1.3, include the nematic, cholesteric, and smectic phases (Shibaev and Bobrovsky, 2017; Andrienko, 2018). An isotropic suspension has no orientational or translational order. The nematic phase is characterized by long-range orientational order without translational order. The major axis of the colloid

is aligned with a preferred direction, $\boldsymbol{n}$. Similar to the nematic phase, the cholesteric (or chiral nematic) phase has long-range orientational order without translational order. The cholesteric phase differs from the nematic phase in that the director varies from layer to layer by a helical twist in the cholesteric phase. The smectic phase is characterized by stratification, where the colloidal particles are arranged into layers, and thus have have orientational order. Within a layer, the smectic A phase has no translational order. The smectic B phase contains colloidal particles arranged on a hexagonal lattice, which is referred to as the solid phase in this study. The smectic C phase contains colloidal particles with a preferred direction at an angle from the direction of layer formation.

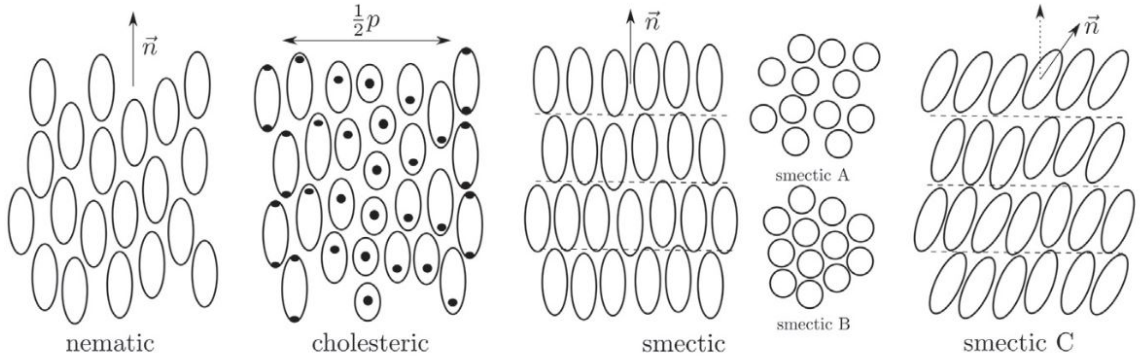


Figure 1.3: Liquid crystalline phases illustrated by Andrienko (2018).

The phase transitions of thermotropic liquid crystals, obtained by melting solids, are controlled by the temperature (Andrienko, 2018). For example, as the temperature is decreased, the suspension moves from the smectic A phase to C then to B. The phase transitions of lytropic liquid crystalline suspensions of rod-like colloidal particles are governed by the concentration. Flory (1956) hypothesized that the separation of anisotropic phases from the isotropic phase can arise from excluded volume interactions. The chiral phases are hypothesized to form from packing of screwlike rods, which are twisted around the major rod axis (Straley, 1976; Orts et al., 1998).

The diversity of structure and properties gives rise to many applications, such as liquid crystalline plating solutions for mesoporous platinum films (Attard et al., 1997),

photocontrollable materials (Shibaev and Bobrovsky, 2017), and liquid crystal displays (Ito et al., 2007). Piezoelectricity is also observed in nematic liquid crystals (Meyer, 1969; Jákli et al., 2009). Clark and Lagerwall (1983) built an electro-optical device by placing chiral smectic C liquid crystals between plates and applying external an electric field.

Rheology of Liquid Crystalline Suspensions

The non-Newtonian rheology of liquid crystalline suspensions is typically described by the three-region flow curve illustrated in Fig. 1.4, first proposed by Onogi and Asada (1980) and reviewed by Wissbrun (1981). Region I is shear thinning at low shear rates. The magnitude of the viscosity strongly depends on the properties of the liquid crystal. Region II is a viscosity plateau, which is narrow for polymer liquid crystals. Region III is shear thinning at high shear rates with power-law dependence. In this region, the viscosity is least dependent on the properties and processing history of the liquid crystals.

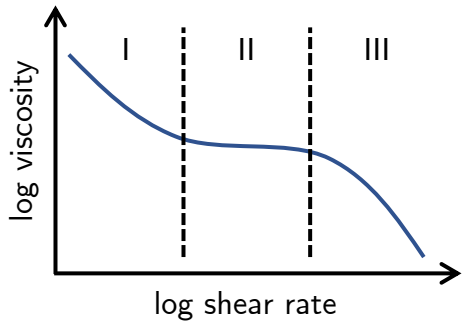


Figure 1.4: Schematic diagram of the three-region flow curve, inspired by Onogi and Asada (1980).

Depending on the type of liquid crystals, the full three-region flow curve might not be accessible experimentally. In lyotropic suspensions of poly(benzyl glutamate) and hydroxypropylcellulose, Burghardt (1998) observed both the partial and full three-region flow curve. Different materials and flow conditions reveal different sections of the potentially universal flow curve.

The three-region flow curve was interpreted using polydomains in Fig 1.5 (Onogi

and Asada, 1980). Region I corresponds to a piled polydomain, which consists of small domains. Region II is the dispersed polydomain, where small domains are suspended. Region III is the monodomain continous phase where all domains align with the flow. The domains for higher concentrations are smaller and require higher shear rates to turn into a monodomain (Asada et al., 1980).

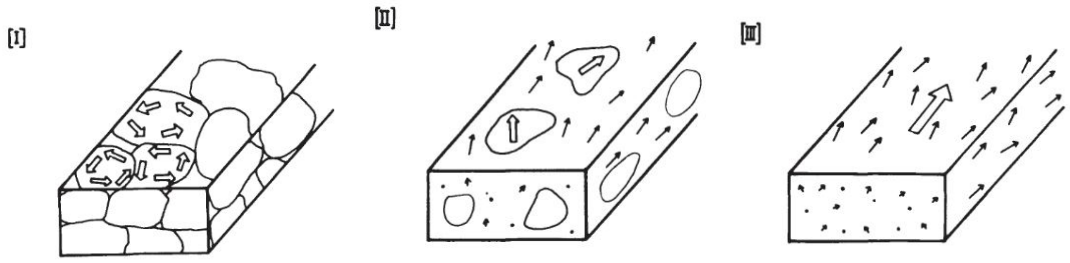


Figure 1.5: Schematic diagram of polydomains in liquid crystalline suspensions by Asada et al. (1980).

Simulations

Simulations can address experimental limitations in concentration. Experimental studies of liquid crystalline suspensions above 10-15 wt% are difficult due to glass transitions (Lagerwall et al., 2014). Rod-like colloidal particles are often modeled as hard spherocylinders for Monte Carlo simulations. McGrother et al. (1996) constructed a phase diagram that contained the isotropic, nematic, smectic A, and solid phases from simulations of hard spherocylinders with $3 \leq r_p \leq 5$, where r_p is the length of the backbone (excluding spherical caps) to diameter ratio, also known as the aspect ratio. All phase transitions were associated with a jump in density. Van Duijneveldt et al. (2000) added a flexible end point to one end of the spherocylinder and a terminal dipole to the other end for spherocylinders with $4 \leq r_p \leq 5$. The addition of dipole stablized the nematic phase and diminished the smectic A phase. Cinacchi et al. (2004) simulated binary mixtures of spherocylinders with different r_p, up to 10. Different smectic A phases were observed, including a smectic A_2 case, where the fibers of one r_p intercalated layers formed by fibers of the other r_p.

Monte Carlo simulations are useful for studying the structure of the suspensions but cannot provide dynamic properties, such as the translational and rotational diffusivities, and rheological properties. Lowen (1994) obtained the translational and rotational diffusivities of suspensions of spherocylinders with $1 \leq r_p \leq 6$ in the isotropic phase. The spherocylinders were moved sequentially with the displacements from the diffusivities of an isolated spherocylinder. Tao et al. (2005) employed Brownian dynamics simulations to obtain the diffusivity for rods with $r_p = 60$ at various volume fractions. The diffusion coefficient was calculated from the restoring torque on a rod by its neighbors and agreed with traditional approaches. Nath and Heussinger (2019) more recently explored the effect of friction on the rheological properties of dense suspensions of soft spherocylinders with $0.001 \leq r_p \leq 2$ through molecular dynamics. The suspensions were shear thickening with the addition of friction forces, which also hindered the alignment of the spherocylinders with the direction of flow. The three-region flow curve described in Sec. 1.3.2 was not observed.

1.3.3 *Suspension of Flexible Fibers*

Flexible fibers, such as nylon (Soszynski and Kerekes, 1988a), polyethylene terephthalate (Chen et al., 2002), carbon (Jiang et al., 2016), and wood fibers (Mason, 1950) flocculate in suspension above critical concentrations in many technical processes (Björkman, 2003). Flocculation impacts the rheological properties and is crucial to the end-product quality (Zhang et al., 2012; Chen et al., 2002). This section explores the mechanism of flocculation and its impact on rheological properties, along with simulation studies.

Floc Formation

Flocculated suspensions contain structure of various degrees of homogeneity, from connected fiber networks to flocs, which are regions of highly entangled fibers. Mason (1950) described floc formation as a dynamic process; the fibers continuously enter and leave the floc with rates equal at equilibrum. Soszynski and Kerekes (1988a) hypothesized that the

relative velocity difference between fibers causes local crowding and leads to floc formation. The crowding number quantifies the number of fibers in a sphere with a diameter equal to the fiber length, L,

$$N_c = \frac{2}{3} C_v \left(\frac{L}{D} \right)^2 \tag{1.26}$$

where C_v is the concentration (volume fraction), and D is the fiber diameter. At $N_c < 1$, the suspension is dilute and the fibers collide by chance (Mason, 1950). Suspensions with $1 < N_c < 60$ are semi-concentrated, where forced collisions occur (Kerekes and Schell, 1992). At $N_c > 60$, the suspension is concentrated, the fibers experience continuous contact, and coherent flocs form. At $N_c = 60$, fibers on average have three contacts per fiber, in agreement with theory developed by Meyer and Wahren (1964). Martinez et al. (2001) further identified $N_c = 16$ as the "gel concentration point", below which the suspension is essentially dilute and above which fibers begin to flocculate.

Meyer and Wahren (1964) hypothesized that at least three fiber contacts are required to entangle fibers in the absence of attractive forces. Fibers come into contact and entangle through flow. Upon cessation of flow, the fibers remain elastically strained, held in place by normal and friction forces. This is known as the elastic interlocking mechanism, which is supported by experimental evidence. When nylon flocs were heated above the glass transition temperature, the fiber stiffness was reduced. The heated flocs dispersed more readily than unheated flocs (Soszynski and Kerekes, 1988b). Thus, the floc was held together elastically.

Zauscher and Klingenberg (2001) measured the coefficient of friction between cellulose surfaces, which decreased with increasing concentration of water-soluble polymers (WSPs). Samaniuk et al. (2012) observed that the same WSPs reduced the yield stress and increased the plastic viscosity of lignocellulosic biomass. The results suggests that friction is important in the rheology and structure of flexible fiber suspensions.

Roller formations are observed in suspensions of wood fibers (Schmid and Klingenberg, 2000a), multiwalled carbon nanotubes (Lin-Gibson et al., 2004), polymer blends

(Hobbie et al., 2002), thixotropic clay gel (Pignon et al., 1997), and microfibrillated cellulose (Karppinen et al., 2012; Sorvari et al., 2013). Rollers are cylindrical flocs whose major axis aligns with the vorticity direction and undergo log-rolling motion under shear flow. Karppinen et al. (2012) observed that rollers form at concentrations near the gel point for microfibrillated cellulose suspensions. Lin-Gibson et al. (2004) hypothesized that roller formation in multiwalled carbon nanotube suspensions is a localized analog of the Weissenberg rod-climbing effect (Weissenberg, 1947), where fluid rises up an immersed rotating rod. Elastic forces acting along a closed streamline induce an inward radial pressure, squeezing the suspension with a normal force up the rod.

Rheology of Fiber Suspensions

The rheological properties of fiber suspensions are highly dependent on the concentration. Dilute suspensions are nearly Newtonian (Chaouche and Koch, 2001); semi-dilute and concentrated suspensions exhibit complex rheological behavior due to flocculation (Derakhshandeh et al., 2011).

A number of experiments were conducted to understand the rheological behavior of semi-dilute and concentrated suspensions (Bibbo et al., 1985; Ganani and Powell, 1985). Bennington et al. (1990) observed that the yield stress of pulp and synthetic fiber suspensions depended on the volume fraction, fiber aspect ratio, and the elastic modulus. Mongruel and Cloitre (1999) observed that semi-dilute suspensions of polyamide fibers in Newtonian suspending fluids exhibited shear thinning at intermediate shear rates. At higher shear rates, the flocs dispersed and fibers aligned with the flow direction. The suspensions then exhibited Newtonian rheological behavior, dependent only on the fiber aspect ratio and concentration.

Petrich et al. (2000) investigated the effect of concentration on the structure and rheological properties of fiber suspensions with aspect ratios of 50 and 72. The viscosity scaled with nL^2d, where n is the number density, L is the fiber length, and d is the diameter. The microstructure, quantified by the period and distribution of the Jeffery orbits, matched

theoretical predictions at dilute concentrations. At semi-dilute concentrations, the period of rotation was less than the Jeffery value due to hydrodynamic interactions. Chaouche and Koch (2001) measured the adhesive force between nylon fibers to be comparable to the hydrodynamic torques that rotate fibers. The shear thinning behavior of fiber suspensions was attributed to floc formation and breakage, which was governed by both the hydrodynamic forces and adhesive forces.

Chen et al. (2002) correlated the non-Newtonian behavior of pulp and polyethylene terephthalate suspensions to the microstructure. At low shear rates, the shear stress increased with increasing shear rate. As the shear rate was increased, discrete jumps in the shear stress were observed as flocs formed. The jumps disappeared when the flocs redisperse at even higher shear rates. Karppinen et al. (2012) imaged microfibrillated cellulose suspensions during flow and correlated regions in the flow curve with floc features. At low shear rates, the suspensions were shear thinning and contained homogeneous fiber networks. At intermediate shear rates, the networks began to break down; a viscosity plateau was observed along with a broad floc size distribution. As the shear rate was increased, voids appeared in the fiber networks and distinct flocs were formed. The suspension became shear thinning while large flocs broke down and the floc size distribution narrowed. At even higher shear rates, the flocs disintegrated.

Semi-dilute and concentrated fiber suspensions are thixotropic, exhibiting time dependent rheology. In a constant stress experiment with a microfibril pulp, Lasseuguette et al. (2008) observed that the shear strain did not begin to increase until 5 seconds into the experiment as the stress broke down the fiber network. In another experiment, the shear rate was increased then decreased twice continuously. The viscosity as a function of shear rate traced out two different loops. The difference was attributed to the break down of some aggregates in the first cycle that were not re-formed in the second.

Theories and Simulations

Sufficiently long fibers are often modeled as slender bodies, which are represented as lines of discrete points that interact with the suspending fluid. Batchelor (1970b) employed slender body theory to represent long rods by placing Stokeslets on a line. The force per unit length was derived for a long slendor body as an asymptotic expansion in the aspect ratio in undisturbed flow (Cox, 1970) and shear flow (Cox, 1971). Hinch (1976) employed slender body theory to model a single thread in Stokes flow and investigated the thread stretching and buckling.

Yamamoto and Matsuoka (1993) and Skjetne et al. (1997) modeled fibers as bead chains. Stiff fibers follow Jeffery orbits with the periodicity that depends only on the fiber aspect ratio. Similar to experimental observations, flexible fibers drift away from Jeffery orbits. The drift depended on the fiber stiffness, initial orientation, and ambient flow field.

Ross and Klingenberg (1997) replaced spheres with prolate spheroids to reduce the computation time. Fibers interacted through repulsive forces. Hydrodynamic interaction and inertia were neglected. The approximation was similarly validated by the agreement with Jeffery orbits for stiff fibers. Oscillations in the viscosity with decaying magnitude with the shear strain were observed, similar to experimental results of Ivanov et al. (1982).

Schmid, Switzer, and Klingenberg employed the linked-spherocylinder model to investigate the effect of friction, attraction, and shape on the structure and rheological properties of the fiber suspensions. Fiber suspensions could flocculate with only friction forces, in the absence of attractive forces (Schmid et al., 2000). The cohesiveness of flocs, quantified by the stored elastic energy, was lower for flocs formed from only attractive forces than for those formed from only friction forces (Schmid and Klingenberg, 2000b). The viscosity for curved fiber suspensions were larger than that for straight fibers (Switzer and Klingenberg, 2003). The shear thinning behavior of fiber suspensions was associated with a competition of hydrodynamic forces and fiber elasticity. Anisotropic bending at

the joints connecting the spherocylinders pushed flocculation to occur at larger values of the coefficient of friction (Switzer and Klingenberg, 2004).

Lindström and Uesaka (2008b) employed the linked-spherocylinder model with normal, friction, and additional lubrication forces to investigate the motion of isolated thread-like fibers in suspension. Inertia and artificial damping in the joints were included. The model was utilized to simulate dewatering by moving two fiber networks towards each other through a fiber suspension. Simulation results qualitatively matched experiments and no large concentration gradients existed in the suspension.

1.4 Cellulose Nanomaterials

Cellulose is the most abundant natural polymer and drives the development of renewable materials at industrial scales (Klemm et al., 2018). Nanofibrillated cellulose (NFC) and cellulose nanocrystals (CNCs) are two classes of nanomaterials (see Fig. 1.6), obtained by breaking down the natural cellulose bundle (Lavoine et al., 2012). Nanofibrillated cellulose fibers are flexible, containing both amorphous and crystalline regions; cellulose nanocrystals are rigid, containing only the crystalline regions. The wide range of fiber properties leads myriad of applications, described below for each material.

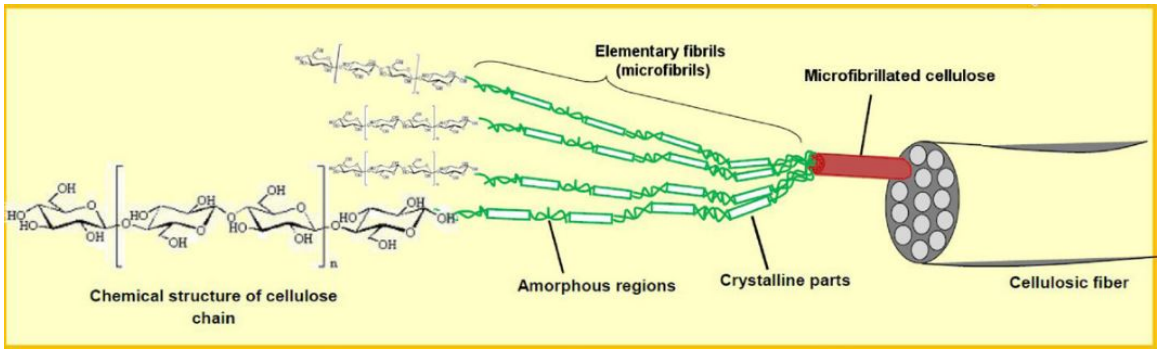


Figure 1.6: Schematic diagram of the structure of cellulose fibers from Lavoine et al. (2012).

1.4.1 *Nanofibrillated Cellulose*

Processing and Applications

Nanofibrillated cellulose (NFC) (or microfibrillated cellulose) fibers are typically 20-60 nm in diameter and up to a few micrometers in length (Lavoine et al., 2012). The most common raw material is wood pulp, such as bleached kraft pulp (Iwamoto et al., 2011) and sulfite pulp (Aulin et al., 2010). Other materials include algae, tunicate, sugar beet, wheat straw, and carrot (Moon et al., 2011; Lavoine et al., 2012). Nanofibrillated cellulose is used in many industrial applications, such as nanocomposites (Siqueira et al., 2010; István and Plackett, 2010; Xu et al., 2013), emulsion stabilization (Turbak et al., 1983), optical films (Nogi et al., 2009), barriers (Aulin et al., 2010), rheological modifiers (Stenstad et al., 2008), coatings (Spence et al., 2011), and petroleum component replacement (Spence et al., 2011). Nanofibrillated cellulose is an attractive material for composite reinforcements for its biodegradability, low raw material cost, wide range of available filler, low energy consumption, and reactive surface (Siqueira et al., 2010).

Turbak et al. (1983) first proposed a method to produce NFC fibers mechanically. Fibers were refined and precut to an average length of 1 mm and suspended in a polar solvent at a concentration of 1-2%. The suspension was passed through a homogenizer that subjected the fibers to a large pressure drop and high shear, which broke down the cellulose bundles. The diameter decreased while the water retention and viscosity increased with the number of passes through the homogenizer. The suspension was cooled to maintain a temperature of 70-80 °C and diluted as needed. A final filtering step eliminated partially fibrillated fibers.

Microfluidization, micro-grinding, and cryocrushing are alternative processes of NFC production (István and Plackett, 2010; Moon et al., 2011; Spence et al., 2011; Abdul Khalil et al., 2014; Lin et al., 2018). Microfluidizers consist of Z-shaped chambers with width ranging from 100 to 400 microns and are more easily cleaned than homogenizers. Micro-grinders consist of a stationary and a rotating disk with grooves of modifiable size. Cry-

ocrushing freezes fiber suspensions with liquid nitrogen then frees fibrils from the cell wall with strong impact forces. The fibers produced are larger in diameter than from other methods.

Chemical and enzymatic treatments (Pääkkö et al., 2007) introduce surface charges and modify the fiber crystallinity. The repulsion between charged fiber surfaces weakens the strength of hydrogen bonds and increases the flowability of the suspension (Moon et al., 2011; Lavoine et al., 2012). The efficiency of cellulose processing is thus increased. Oxidation using (2,2,6,6-Tetramethylpiperidin-1-yl)oxyl (TEMPO) (Saito et al., 2006) and carboxymethylation (Heggset et al., 2017) are the most common chemical treatments, accounting for 88% and 6% of the literature, respectively (Lavoine et al., 2012).

To reduce the impact of NFC production on the environment, hydrolysis using recyclable dicarboxylic acid (Jia et al., 2017) and recovery of cellulose solid residues to reduce cellulose loss (Wang et al., 2012b) have been developed. The production of carboxylated NFC can also be integrated with the production of cellulose nanocrystals (Zhu et al., 2018), described in Sec. 1.4.2 below.

Fibrillated fibers have large specific surface area, which enables high accessibility for fast reactions and high water retention (Herrick et al., 1983; Lavoine et al., 2012). The surfaces can be functionalized for different applications. The surface hydroxyl groups participate in hydrogen bonding and contribute to the formation of strong fiber networks (Eichhorn et al., 2010). The fibrillation process strips away the soft matrix that bundles natural cellulose. Fibrillated fibers thus have large bending stiffness, with Young's modulus larger than 100 GPa (Azizi Samir et al., 2005; Nogi et al., 2009). The flexibility of the fibers arise from the amorphous linkages between stiff regions. The combination of stiff and amorphous regions, and hydrogen bonding enable the formation of strong yet elastic fiber networks.

Rheology

Nanofibrillated cellulose suspensions are shear thinning (Herrick et al., 1983; Turbak et al., 1983; Goussé et al., 2004; Agoda-Tandjawa et al., 2010; Iotti et al., 2011), viscoelastic, and thixotropic (Barnes, 1997). Similar to flexible fibers discussed in Sec. 1.3.3, MFC and NFC fiber suspensions flocculate above critical concentrations (Karppinen et al., 2012). The extent of shear thinning can be quantified via a power-law relation, $\sigma = K\dot{\gamma}^n$, where σ is the shear stress, K is the consistency factor, and n is the shear thinning index (Lasseuguette et al., 2008). The shear thinning behavior is more pronounced as the concentration is increased, with n smaller than the Newtonian value of 1. Three region flow curves, similar to that of liquid crystalline suspensions discussed in Sec. 1.3.2, are observed. The viscosity plateau between two shear thinning region is attributed to the breaking down of the fiber networks (Agoda-Tandjawa et al., 2010; Iotti et al., 2011).

Nanofibrillated fiber suspensions are viscoelastic, characterized by having a larger storage modulus, G', which quantifies the storage of elastic energy, than the loss modulus, G'', which quantifies the loss of energy to viscous dissipation (Lavoine et al., 2012). As the concentration is increased, G' increases faster than G'', and the suspension becomes a gel. At low concentrations, Chen et al. (2013) observed three regions in the order of increasing angular frequencies: visco-fluid with weak fiber linking, visco-elastic gel when fibers aligned, and gel destruction when fibers disentangled.

Nanofibrillated cellulose suspensions exhibit time-dependent rheological behavior, known as thixotropy. In a series of step shear rate experiments, Lasseuguette et al. (2008) observed an increase in viscosity that reached a maximum then decayed over 200 s before reaching a steady value. The shear strain remained constant when constant stress was applied. The stress presumably first broke down fiber networks that hindered flow before the suspension could flow. In another experiment, the flow curve obtained with shear rate cycles was reproducible only beyond the second cycle. The difference between the first and the second cycle was attributed to a faster break-down of microstructures in

the second cycle.

1.4.2 *Cellulose Nanocrystals*

Processing and Applications

Cellulose nanocrystals (CNCs) have become of interest, evident through the large number of reviews and growing number of publications (De Souza Lima and Borsali, 2004; Azizi Samir et al., 2005; Habibi et al., 2010; Eichhorn et al., 2010; Moon et al., 2011; Klemm et al., 2018). Cellulose nanocrystals are obtained through mechanical homogenization, acid hydrolysis, and enzymatic hydrolysis (Moon et al., 2011). The crystals are 6-8 nm in diameter and 150-250 nm in length (Habibi et al., 2008), with the aspect ratio typically less than 70 (Lavoine et al., 2012). The elastic modulus of a cellulose nanocyrstal is 167.7 GPa (Habibi et al., 2010). The specific elastic modulus of CNC films is 92 GPa·cm^3/g, which is higher than that of steel because of the density difference (Sakurada et al., 1962; Eichhorn et al., 2010). Cellulose nanocrystals mechanically reinforce nanocomposites (Kassab et al., 2019; Jardin et al., 2020) for electronics (Kim et al., 2019), electrochemical sensors (Khalilzadeh et al., 2020), and membranes for CO_2 separation from flue gas (Torstensen et al., 2019). The mechanical enhancement of composites is more dramatic beyond the percolation threshold, which decreases with increasing r_p (Garboczi et al., 1995).

Rheology

Cellulose nanocrystal suspensions exhibit the three-region flow curve (Bercea and Navard, 2000; Shafiei-Sabet et al., 2012) typical of lyotropic liquid crystals, which is described in Sec. 1.3.2. Liquid crystalline phases are observed in CNC suspensions, such as the nematic (Shafiei-Sabet et al., 2012; Ishii et al., 2019) and chiral-nematic phases (Dong et al., 2002). The chirality stems from asymmetry in the crystal via the helical twist about its major axis (Orts et al., 1998). Factors that affect the rheology and structure of CNC suspensions include the amount of energy used to redisperse suspensions (Shafiei-Sabet et al., 2012),

temperature (Heggset et al., 2017), processing methods (Li et al., 2015), surface properties (Shafiei-Sabet et al., 2013; Samyn and Taheri, 2016; Moberg et al., 2017), concentration (Wu et al., 2014; Xu et al., 2017), pH (Xu et al., 2017), salts (Dong et al., 2002; Xu et al., 2017; Phan-Xuan et al., 2016; Shafiei-Sabet et al., 2014), aspect ratio (Li et al., 2015; Moberg et al., 2017; Wu et al., 2014), and polymer additives (Bagheriasl et al., 2016).

1.4.3 *Commercialization Challenges*

The main challenge in the commercialization of CNC, NFC, and MFC applications is that the suspensions must be dried for economic transport. However, it is difficult to redisperse the suspensions and achieve the desired properties because of the interparticle forces that drive the particle aggregation (Bagheriasl et al., 2016, 2019). The spacing between cellulose bundles collapses when fibers are dried. The surface hydroxyl groups form strong hydrogen bonds, which are not completely displaced upon rehydration (Moon et al., 2011; Siqueira et al., 2010). Hydrogen bonding and/or van der Waals interactions contribute to the irreversible aggregation (Ureña-Benavides et al., 2011; Shafiei-Sabet et al., 2012, 2014). The aggregation contributes to differences in the end-product properties, such as viscosity and film opacity, compared to never-dried fibers (Herrick et al., 1983; Nogi et al., 2009; Agoda-Tandjawa et al., 2010; Iwamoto et al., 2011). Polymer composites are often heated for molding. However, the cellulosic material begins to degrade above 220 °C (Siqueira et al., 2010). The high moisture adsorption and incompatibility with polymer matrix constrain the application of nanofibers.

1.5 Overview on Approach

To address challenges in commericalization of cellulose nanomaterials described in Sec. 1.4.3, fundamental understanding of how the fiber dimensions, concentration, shear rate, and processing method impact the structure and rheological properties of fiber suspensions is essential. Fiber-level simulations similar to those described in Sec. 1.3.3 are employed

to investigate the relationships, and can provide directions for experimental efforts.

Brownian spherocylinders are employed to model CNCs. The structure of the suspensions at rest are characterized at various aspect ratios and concentrations. The suspensions that are isotropic and nematic at rest are sheared to investigate the relationship between the structure and the three-region flow curve described in Sec. 1.3.2. Non-Brownian linked-spherocylinders are employed to model NFCs. The effect of friction and attractive forces on the structure and viscosity of the suspensions that undergo drying and rehydration are investigated.

2

FIBER MODELS AND SIMULATION METHODS

2.1 Fiber Models

Coarse grained fiber models and simulation methods were developed to investigate the structure and rheology via simulations for a wide range of fibers, from short and stiff fibers to long and flexible fibers. Fibers have radius b, diameter D, length L, and aspect ratio defined as $r_p = L/D$ (Eken et al., 2012). Short and stiff fibers, such as cellulose nanocrystals, are modeled as rigid spherocylinders as illustrated in Fig. 2.1. Brownian motion is incorporated in the model to capture the effect of thermal forces. Long and flexible fibers, such as nanofibrillated cellulose, are modeled as linked spherocylinders as illustrated in Fig. 2.2, with N_{seg} segments connected by ball-and-socket joints. The segments are uniform in length with half-length, l, segment aspect ratio defined as $r_{ps} = l/b$, segment center of mass, $\boldsymbol{r}$, and orientation, $\boldsymbol{p}$. Instead of Brownian motion, friction forces are added to this model since the flexible fibers are are typically longer. For more details and applications of the fiber models, refer to the work by Schmid and Klingenberg (2000a) and Switzer and Klingenberg (2003) on non-Brownian linked rigid spherocylinders and Bette (2003) on Brownian spherocylinders.

The persistence length, l_p, can be calculated to determine the suitability of the models

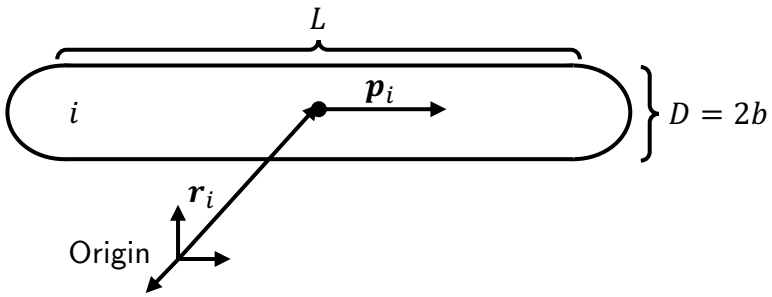


Figure 2.1: Schematic of fiber i modeled as spherocylinders with length L, diameter D, radius b, position $\boldsymbol{r}_i$, and orientation $\boldsymbol{p}_i$.

for materials of interest,

$$l_p = \frac{E_Y I}{k_B T} \tag{2.1}$$

where E_Y is the bending modulus, $I = \pi b^4/4$ is the area moment of inertia, b is the radius, k_B is the Boltzmann constant, and T is the temperature (Solomon and Spicer, 2010). For rigid polymers, $L/l_p \ll 1$; for semi-flexible polymers, $L/l_p \approx 1$. For nanofibrillated cellulose fibers, $E_Y = 78$ GPa and $b = 45$ nm (Guhados et al., 2005). At $T = 298K$, $l_p = 61$ m. For cellulose nanocrystals with $E_Y = 167.5$ (Habibi et al., 2010), and $b = 10$ nm (Lavoine et al., 2012), $l_p = 3 \times 10^{-5}$ m.

2.2 Equations of Motion for Brownian Rigid Spherocylinders

The rigid spherocylinders have mass, m, and moment of inertia, $\boldsymbol{I}_i$. The translational and rotational equations of motion for fiber i with center of mass, $\boldsymbol{r}_i$, and angular velocity, $\boldsymbol{w}_i$ are (Bette, 2003)

$$\boldsymbol{F}_i^{\text{hyd}} + \boldsymbol{F}_i^{\text{Br}} + \sum_j^{N_{Ci}} \boldsymbol{F}_{ij}^{\text{con}} = m\frac{d^2\boldsymbol{r}_i}{dt^2} \tag{2.2}$$

$$\boldsymbol{T}_i^{\text{hyd}} + \boldsymbol{T}_i^{\text{Br}} + \sum_j^{N_{Ci}} \boldsymbol{G}_{ij} \times \boldsymbol{F}_{ij}^{\text{con}} = \boldsymbol{I}_i \cdot \frac{d\boldsymbol{w}_i}{dt} + \tilde{\boldsymbol{w}}_i \cdot (\boldsymbol{I}_i \cdot \boldsymbol{w}_i) \tag{2.3}$$

where $\boldsymbol{F}_i^{\text{hyd}}$ and $\boldsymbol{T}_i^{\text{hyd}}$ are the hydrodynamic force and torque, $\boldsymbol{F}_i^{\text{Br}}$ and $\boldsymbol{T}_i^{\text{Br}}$ are the Brownian force and torque, N_{Ci} is the number of fibers in contact with fiber i, $\boldsymbol{F}_{ij}^{\text{con}}$ is the

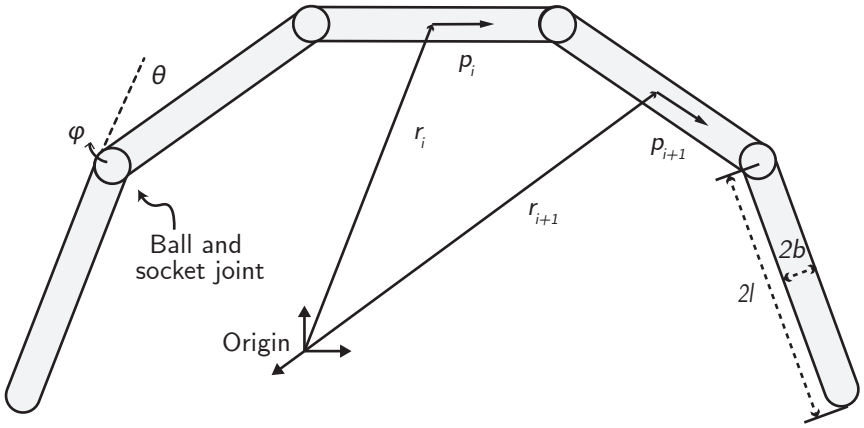


Figure 2.2: Schematic diagram of fibers modeled as linked spherocylinders with radius, b, segment half-length, l, twisting angle, ϕ, and bending angle, θ. Each segment is identified by segment center of mass, $\boldsymbol{r}_i$, and orientation, $\boldsymbol{p}_i$. The angles between segments in this figure is $(\theta, \phi) = (0.6, 0)$.

contact force exerted by fiber j on fiber i, and $\boldsymbol{G}_{ij}$ is the moment arm, used to calculate the associated torque (discussed further in Sec. 2.6).

Neglecting inertia, the translational and rotational equations of motion are

$$\boldsymbol{F}_i^{\text{hyd}} + \boldsymbol{F}_i^{\text{Br}} + \sum_j^{N_{Ci}} \boldsymbol{F}_{ij}^{\text{con}} = 0, \tag{2.4}$$

$$\boldsymbol{T}_i^{\text{hyd}} + \boldsymbol{T}_i^{\text{Br}} + \sum_j^{N_{Ci}} \boldsymbol{G}_{ij} \times \boldsymbol{F}_{ij}^{\text{con}} = 0 \tag{2.5}$$

The forces and torques in these equations are described below. Results from the inertial and inertialess models are compared in Appendix A.

2.3 Equations of Motion for Non-Brownian Linked Rigid Spherocylinders

The translational and rotational equations of motion for a segment i in a linked spherocylinder are

$$\boldsymbol{F}_i^{\text{hyd}} + \sum_j^{N_{Ci}} \boldsymbol{F}_{ij}^{\text{con}} + \boldsymbol{X}_{i+1} - \boldsymbol{X}_i = 0 \tag{2.6}$$

$$\boldsymbol{T}_i^{\text{hyd}} + \sum_j^{N_{Ci}} \boldsymbol{G}_{ij} \times \boldsymbol{F}_{ij}^{\text{con}} + l\boldsymbol{p}_i \times \left[\boldsymbol{X}_{i+1} - \boldsymbol{X}_i\right] + \boldsymbol{Y}_{i+1} - \boldsymbol{Y}_i = 0 \tag{2.7}$$

where $\boldsymbol{X}_i$ is the force applied at joint i to keep the fiber segment inextensible and $\boldsymbol{Y}_i$ is a restoring torque at joint i.

2.4 Hydrodynamic Force and Torque

Hydrodynamic interactions between fibers are neglected. Sundararajakumar and Koch (1997) found that mechanical contacts dominate over hydrodynamic interactions in controlling the fiber orientation distributions when $nL^3/r_p \geq O(1)$, where n is the number density. Throughout this thesis, the ambient translational and angular velocity, $\boldsymbol{U}_i^\infty$ and $\boldsymbol{\Omega}_i^\infty$, are $(\dot{\gamma}z, 0, 0)^{\text{T}}$ and $(0, \dot{\gamma}/2, 0)^{\text{T}}$, where $\dot{\gamma}$ is the shear rate. The rate of strain tensor, $\boldsymbol{E}^\infty$, is defined as $0.5\left[(\nabla \boldsymbol{U}_i^\infty) + (\nabla \boldsymbol{U}_i^\infty)^{\text{T}}\right]$.

For a rigid body in linear flow, the hydrodynamic force and torque are

$$\boldsymbol{F}_i^{\text{hyd}} = \boldsymbol{A}_i \cdot (\boldsymbol{U}_i^\infty - \dot{\boldsymbol{r}}_i) \tag{2.8}$$

$$\boldsymbol{T}_i^{\text{hyd}} = \boldsymbol{C}_i \cdot (\boldsymbol{\Omega}_i^\infty - \boldsymbol{w}_i) + \tilde{\boldsymbol{H}}_i : \boldsymbol{E}^\infty \tag{2.9}$$

where $\boldsymbol{A}_i$, $\boldsymbol{C}_i$, $\tilde{\boldsymbol{H}}_i$ are resistance tensors of an isolated prolate spheroid with equivalent aspect ratio $r_e = 0.7 r_p$ (Kim and Karilla, 1991). The equivalent aspect ratio is chosen so that the spherocylinder and the prolate spheroid have the same Jeffery orbit period

(Jeffery, 1922; Goldsmith and Mason, 1962). The resistance tensors are

$$\boldsymbol{A}_i = 6\pi\eta_0 l \left[Y^A \boldsymbol{\delta} + (X^A - Y^A)\boldsymbol{p}_i\boldsymbol{p}_i \right] \tag{2.10}$$

$$\boldsymbol{C}_i = 8\pi\eta_0 l^3 \left[Y^C \boldsymbol{\delta} + (X^C - Y^C)\boldsymbol{p}_i\boldsymbol{p}_i \right] \tag{2.11}$$

$$\tilde{\boldsymbol{H}}_i = -8\pi\eta_0 l^3 Y^H \boldsymbol{\epsilon} \cdot \boldsymbol{p}_i\boldsymbol{p}_i \tag{2.12}$$

where ϵ is the Levi-Civita permutation tensor. The scalar resistance functions depend on the eccentricity defined as $e = \left(1 - r_e^2\right)^{1/2}$. For linked spherocylinders, r_e refers to the equivalent aspect ratio of a segment.

$$X^A = \frac{8}{3} e^3 \left[-2e + \left(1 + e^2\right) E \right]^{-1} \tag{2.13}$$

$$Y^A = \frac{16}{3} e^3 \left[2e + \left(3e^2 - 1\right) E \right]^{-1} \tag{2.14}$$

$$X^C = \frac{4}{3} e^3 \left(1 - e^2\right) \left[2e - \left(1 - e^2\right) E \right]^{-1} \tag{2.15}$$

$$Y^C = \frac{4}{3} e^3 \left(2 - e^2\right) \left[-2e + \left(1 + e^2\right) E \right]^{-1} \tag{2.16}$$

$$Y^H = \frac{4}{3} e^5 \left[-2e + \left(1 + e^2\right) E \right]^{-1} \tag{2.17}$$

where

$$E = \ln\left(\frac{1+e}{1-e}\right) \tag{2.18}$$

2.5 Brownian Forces and Torque

The Brownian force and torque are random vectors, normally distributed, with zero-mean and variance given by the fluctuation-dissipation therorem (Kubo, 1966)

$$\left\langle \boldsymbol{F}_i^{\mathrm{Br}}(t) \right\rangle = 0, \qquad \left\langle \boldsymbol{T}_i^{\mathrm{Br}}(t) \right\rangle = 0 \tag{2.19}$$

$$\left\langle \boldsymbol{F}_i^{\mathrm{Br}}(t) \boldsymbol{F}_i^{\mathrm{Br}}(t+\tau) \right\rangle = 2k_B T \boldsymbol{A}_i \delta(\tau) \tag{2.20}$$

$$\left\langle \boldsymbol{T}_i^{\mathrm{Br}}(t) \boldsymbol{T}_i^{\mathrm{Br}}(t+\tau) \right\rangle = 2k_B T \boldsymbol{C}_i \delta(\tau) \tag{2.21}$$

where $\delta(\cdot)$ is the Dirac-Delta function.

2.6 Contact Force

Fibers interact with normal and friction forces.

$$\boldsymbol{F}_{ij}^{\mathrm{con}} = \boldsymbol{F}_{ij}^{\mathrm{N}} + \boldsymbol{F}_{ij}^{\mathrm{fric}} \tag{2.22}$$

The friction force is described in Sec. 2.7 below. Figure 2.3 illustrates two fibers in contact. The distance from the center of the fiber ($N_{\mathrm{seg}} = 1$) or segment i to the point of contact with fiber or segment j is s_{ij}. The shortest separation of the major axes of the fibers is g_{ij} in the $\boldsymbol{n}_{ij}$ direction,

$$\boldsymbol{n}_{ij} = \frac{\boldsymbol{r}_j + s_{ji}\boldsymbol{p}_j - \boldsymbol{r}_i - s_{ij}\boldsymbol{p}_i}{g_{ij}} \tag{2.23}$$

The separation of the fiber surfaces is $h_{ij} = g_{ij} - 2b$. The moment arm, $\boldsymbol{G}_{ij}$, points from the center of the fiber to the midpoint of the separation.

$$\boldsymbol{G}_{ij} = s_{ij}\boldsymbol{p}_i + \frac{g_{ij}}{2}\boldsymbol{n}_{ij} \tag{2.24}$$

The definitions of s_{ji} and $\boldsymbol{G}_{ji}$ follow similar forms. For arbitrary distances from the center

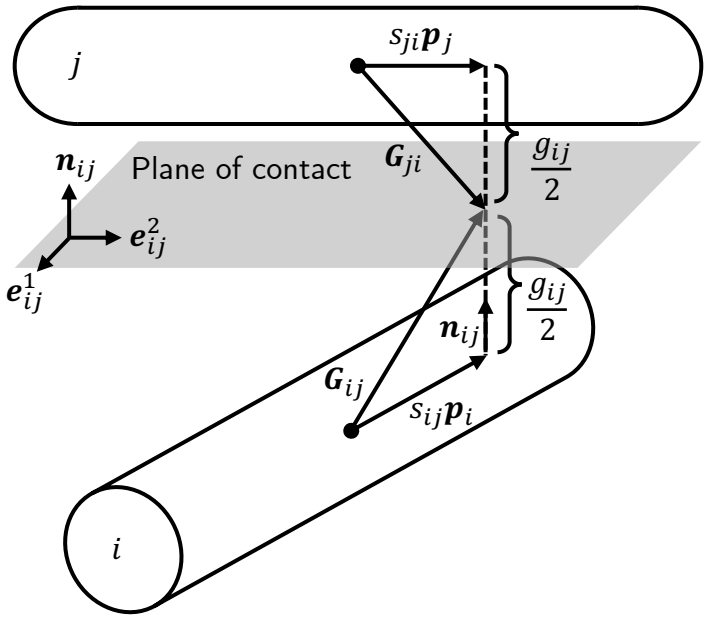


Figure 2.3: Schematic of two fibers in contact.

along the major axes, s'_{ij} and s'_{ji}, the separation is

$$g_{ij} = \|\boldsymbol{r}_j + s'_{ji}\boldsymbol{p}_j - \boldsymbol{r}_i - s'_{ij}\boldsymbol{p}_i\| \tag{2.25}$$

The minimum separation is obtained by from solving $\nabla g_{ij} = 0$.

$$s_{ij} = \frac{\left[(\boldsymbol{r}_i - \boldsymbol{r}_j)\cdot\boldsymbol{p}_j\right](\boldsymbol{p}_i\cdot\boldsymbol{p}_j) - (\boldsymbol{r}_i - \boldsymbol{r}_j)\cdot\boldsymbol{p}_i}{1-(\boldsymbol{p}_i\cdot\boldsymbol{p}_j)^2} \tag{2.26}$$

$$s_{ji} = \frac{\left[(\boldsymbol{r}_j - \boldsymbol{r}_i)\cdot\boldsymbol{p}_i\right](\boldsymbol{p}_i\cdot\boldsymbol{p}_j) - (\boldsymbol{r}_j - \boldsymbol{r}_i)\cdot\boldsymbol{p}_j}{1-(\boldsymbol{p}_i\cdot\boldsymbol{p}_j)^2} \tag{2.27}$$

There are three types of contacts: side-side, end-side, and end-end contacts. The side-side contact occurs when both $|s_{ij}| < l$ and $|s_{ji}| < l$. The end-end contact occurs when both $|s_{ij}| > l$ and $|s_{ji}| > l$. In this case, the distances s_{ij} and s_{ji} are set to $\pm l$ depending on which end of the fiber is in contact. The end-side contact occur when only one of $|s_{ij}|$ and $|s_{ji}|$ exceeds l. The distance s_{ij} or s_{ji} is set to $\pm l$; the remaining distance is adjusted.

For example, if $s_{ij} > l$, s_{ij} is set to l and $s_{ji} = \boldsymbol{p}_j \cdot (\boldsymbol{r}_i + l\boldsymbol{p}_i - \boldsymbol{r}_j)$.

The normal force exerted by fiber/segment j on fiber/segment i is

$$\boldsymbol{F}_{ij}^N = -\left(F^{\text{rep}} e^{-a^{\text{rep}} h_{ij}} - F^{\text{att}} e^{-a^{\text{att}} h_{ij}^2}\right) \boldsymbol{n}_{ij} \tag{2.28}$$

where F^{rep} and F^{att} are the repulsive and attractive force coefficients, and a^{rep} and a^{att} are the repulsive and attractive decay parameters. When the fibers overlap, $h_{ij} < 0$.

2.7 Constraints

2.7.1 *Inextensibility*

The constraint of inextensibility is relevant for fibers with more than one segment. The lengths of the whole fiber and individual segments are kept constant by applying a force $\boldsymbol{X}_i$ at joint i to satisfy the constraint

$$\boldsymbol{r}_i + l\boldsymbol{p}_i - (\boldsymbol{r}_{i+1} - l\boldsymbol{p}_{i+1}) = \boldsymbol{0} \tag{2.29}$$

At joint 1, $\boldsymbol{X}_1 = 0$.

2.7.2 *No Relative Motion*

The constraint of no relative motion is relevant for models involving friction forces. The basis for the plane of contact is $\left(\boldsymbol{e}_{ij}^1, \boldsymbol{e}_{ij}^2, \boldsymbol{n}_{ij}\right)$ in Fig. 2.3. The fibers cannot move relative to each other in the plane of contact.

$$\begin{bmatrix} \Delta\boldsymbol{u}_{ij} \cdot \boldsymbol{e}_{ij}^1 \\ \Delta\boldsymbol{u}_{ij} \cdot \boldsymbol{e}_{ij}^2 \\ \boldsymbol{F}_{ij}^{\text{fric}} \cdot \boldsymbol{n}_{ij} \end{bmatrix} = 0 \tag{2.30}$$

where $\Delta \boldsymbol{u}_{ij}$ is the velocity difference at the point of contact defined as

$$\Delta \boldsymbol{u}_{ij} = \dot{\boldsymbol{r}}_i - \dot{\boldsymbol{r}}_j + \boldsymbol{w}_i \times \boldsymbol{G}_{ij} - \boldsymbol{w}_j \times \boldsymbol{G}_{ji} \tag{2.31}$$

This constraint is solved for $\boldsymbol{F}_{ij}^{\text{fric}}$, which is evaluated according to Coloumb's law of friction, with μ^{stat} and μ^{kin} as the static and kinetic friction coefficients. The contact between fibers is broken (motion decoupled) when $\|\boldsymbol{F}_{ij}^{\text{fric}}\| > \mu^{\text{stat}} \|\boldsymbol{F}_{ij}^{N}\|$ and the fibers slides in the plane of contact with velocity proportional to μ^{kin}.

$$\begin{aligned} \|\boldsymbol{F}_{ij}^{\text{fric}}\| &\leq \mu^{\text{stat}} \|\boldsymbol{F}_{ij}^{N}\| \\ &> \mu^{\text{stat}} \|\boldsymbol{F}_{ij}^{N}\| \rightarrow \boldsymbol{F}_{ij}^{\text{fric}} = \mu^{\text{kin}} \boldsymbol{F}_{ij}^{N} \frac{\Delta \boldsymbol{u}_{ij}}{\|\Delta \boldsymbol{u}_{ij}\|} \end{aligned} \tag{2.32}$$

where $\Delta \boldsymbol{u}_{ij}$ is the relative velocity at the point of contact. When $\|\boldsymbol{F}_{ij}^{\text{fric}}\| > \mu^{\text{stat}} \|\boldsymbol{F}_{ij}^{N}\|$ for multiple contacts in a connected fiber cluster, the contact with the largest $\|\boldsymbol{F}_{ij}^{\text{fric}}\|$ is broken. The frictional forces are resolved and reevaluated. For the work presented in Chapter 4, $\mu^{\text{kin}} = 0$.

2.8 Frames of Reference

The relation between the frames of reference used in this work is illustrated in Fig. 2.4. The laboratory (inertial) frame, used for the equations of motion, has a fixed basis, $\hat{\boldsymbol{e}}_x = (1,0,0)^T$, $\hat{\boldsymbol{e}}_y = (0,1,0)^T$, and $\hat{\boldsymbol{e}}_z = (0,0,1)^T$. The body frame for segment i has the basis $\hat{\boldsymbol{x}}_i$, $\hat{\boldsymbol{y}}_i$, and $\boldsymbol{p}_i$, which is the orientation of the segment. The equilibrium frame for segment i has the basis $\hat{\boldsymbol{x}}_i^{\text{eq}}$, $\hat{\boldsymbol{y}}_i^{\text{eq}}$, and $\boldsymbol{p}_i^{\text{eq}}$.

The vectors $\hat{\boldsymbol{x}}_i$ and $\hat{\boldsymbol{y}}_i$ are chosen at random at the start of the simulation then rotated and translated with the segment. The equilibrium body frame for segment i is fixed relative to the previous segment $i-1$ with the equilibrium bending and twisting angles, θ_i^{eq} and ϕ_i^{eq}. Both the body and equilibrium body frames translate and rotate based on

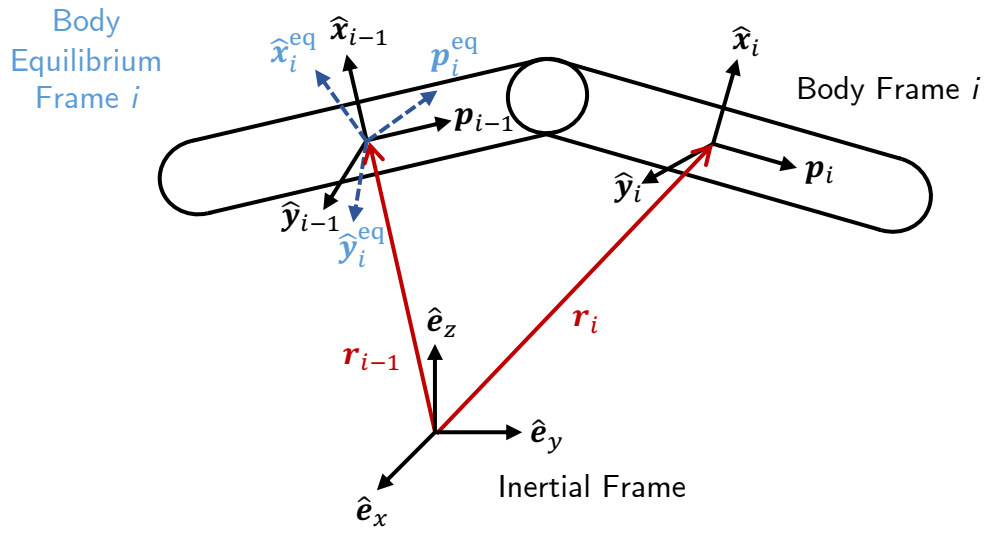


Figure 2.4: Schematic of the frames of reference used, including the inertial frame, body frame for segment i, and the body equilibrium frame of segment i fixed on segment $i-1$.

dynamics; both frames are used to calculate the angle deviations for the restoring torques in Sec. 2.11. Rotation matrices (see Sec. 2.9 and 2.10) are used to convert vectors and matrices from one reference frame to another.

2.9 Equilibrium Rotation Matrix and Euler Angles

Any rotation can be represented by a sequence of rotations about the x, y, and z axes by the Euler angles, (ϕ, θ, ψ), which are the spin, nutation, and precession angles, respectively (Diebel, 2006). The (x, y, z), (z, x, z), and (z, y, z) rotations are the most common of the 12 valid rotation sequences.

In this thesis, the (z, y, z) sequence, *i. e.* rotate (1) about the z axis by ϕ, (2) about the y

axis by θ, and (3) about the z axis by ψ, is employed. The rotation matrix is

$$\boldsymbol{R} = \begin{pmatrix} -\sin\phi\sin\psi + \cos\phi\cos\theta\cos\psi & -\sin\phi\cos\psi - \cos\phi\cos\theta\sin\psi & \cos\phi\sin\theta \\ \cos\phi\sin\psi + \sin\phi\cos\theta\cos\psi & \cos\phi\cos\psi - \sin\phi\cos\theta\sin\psi & \sin\phi\sin\theta \\ -\sin\theta\cos\psi & \sin\theta\sin\psi & \cos\theta \end{pmatrix} \tag{2.33}$$

The equilibrium rotation matrix, $\boldsymbol{R}_i^{\text{eq}}$, transforms vectors in the body frame of segment $i-1$ to the equilibrium body frame of segment i.

$$\hat{\boldsymbol{e}}_i^{\text{eq}} = \boldsymbol{R}_i^{\text{eq}} \cdot \hat{\boldsymbol{e}}_{i-1}^{\text{body}}, \qquad \hat{\boldsymbol{e}}_{i-1}^{\text{body}} = \left(\boldsymbol{R}_i^{\text{eq}}\right)^{\dagger} \cdot \hat{\boldsymbol{e}}_i^{\text{eq}} \tag{2.34}$$

The equilibrium rotation matrix can be obtained from Eq. 2.33 with $(\phi, \theta, \psi) = \left(\phi_i^{\text{eq}}, \theta_i^{\text{eq}}, 0\right)$.

2.10 Rotation Matrix and Euler Parameters

The rotation matrix, $\boldsymbol{R}$, transforms any vector, $\boldsymbol{v}$, and matrix, $\boldsymbol{M}$, from the laboratory frame to the body frame.

$$\boldsymbol{R}_i = \begin{pmatrix} \hat{\boldsymbol{x}}_i \cdot \hat{\boldsymbol{e}}_x & \hat{\boldsymbol{x}}_i \cdot \hat{\boldsymbol{e}}_y & \hat{\boldsymbol{x}}_i \cdot \hat{\boldsymbol{e}}_z \\ \hat{\boldsymbol{y}}_i \cdot \hat{\boldsymbol{e}}_x & \hat{\boldsymbol{y}}_i \cdot \hat{\boldsymbol{e}}_y & \hat{\boldsymbol{y}}_i \cdot \hat{\boldsymbol{e}}_z \\ \boldsymbol{p}_{ix} & \boldsymbol{p}_{iy} & \boldsymbol{p}_{iz} \end{pmatrix} \tag{2.35}$$

$$\boldsymbol{v}^{\text{body}} = \boldsymbol{R} \cdot \boldsymbol{v}^{\text{lab}}, \qquad \boldsymbol{v}^{\text{lab}} = \boldsymbol{R}^{\dagger} \cdot \boldsymbol{v}^{\text{body}} \tag{2.36}$$

$$\boldsymbol{M}^{\text{body}} = \boldsymbol{R} \cdot \boldsymbol{M}^{\text{lab}} \cdot \boldsymbol{R}^{\dagger}, \qquad \boldsymbol{M}^{\text{lab}} = \boldsymbol{R}^{\dagger} \cdot \boldsymbol{M}^{\text{body}} \cdot \boldsymbol{R} \tag{2.37}$$

The rotation matrix is orthogonal, thus $\boldsymbol{R}^{-1} = \boldsymbol{R}^{\dagger}$. Since the laboratory frame is Cartesian, the basis $\hat{\boldsymbol{x}}_i$ and $\hat{\boldsymbol{y}}_i$ for the body frame are stored as the first and second rows of $\boldsymbol{R}$. Diagonal matrices in the body frame, such as the resistance tensors in Eqs. 2.10-2.12, are often operated on before being transformed to the inertial frame for simplicity. For example, a diagonal matrix with elements M_{11}^{body}, M_{11}^{body}, and M_{33}^{body} is raised to an arbitrary power,

p, and then rotated to the inertial frame,

$$\boldsymbol{M}^p = \left(M_{11}^{\text{body}}\right)^p \boldsymbol{\delta} + \left[\left(M_{33}^{\text{body}}\right)^p - \left(M_{11}^{\text{body}}\right)^p\right] \boldsymbol{pp} \tag{2.38}$$

The inverse and the square root of the matrix are obtained when $p = -1$ and $p = 0.5$, respectively.

The rotational equations of motion in Eq. 2.5 and 2.7 are not directly integrable. The angular velocity, $\boldsymbol{w}$, is related to the rotation matrix, which has a time derivative

$$\dot{\boldsymbol{R}}(t) = \boldsymbol{w}(t) \times \boldsymbol{R}(t) \tag{2.39}$$

Both the Euler angles (Sec. 2.9) and the Euler parameters are often used to represent the rotation matrix. With Euler angles, the integration of $\dot{\boldsymbol{R}}$ is prone to numerical errors, such as singularities. Linearization of sine and cosine is only considered valid for angles that are less than 10° (Diebel, 2006). With Euler parameters, the integration of $\dot{\boldsymbol{R}}$ only involves algebraic expressions and is not prone to numerically drift.

The Euler parameters, q_0, q_1, q_2, and q_3, form a unit quaternion, $\boldsymbol{q} = (q_0, q_1, q_2, q_3)^T$, which satisfies

$$\|\boldsymbol{q}\| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^3} = 1 \tag{2.40}$$

The rotation matrix, $\boldsymbol{R}$, expressed in terms of the Euler parameters is

$$\boldsymbol{R} = \begin{pmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2q_1q_2 + 2q_0q_3 & 2q_1q_3 - 2q_0q_2 \\ 2q_1q_2 - 2q_0q_3 & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2q_2q_3 + 2q_0q_1 \\ 2q_1q_3 + 2q_0q_2 & 2q_2q_3 - 2q_0q_1 & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{pmatrix} \tag{2.41}$$

Conversely, the Euler parameters expressed in terms of the elements of $\boldsymbol{R}$ are

$$q_0 = \frac{1}{2}\sqrt{1 + R_{11} + R_{22} + R_{33}} \tag{2.42}$$

$$q_1 = \frac{1}{2}\frac{R_{23} - R_{32}}{\sqrt{1 + R_{11} - R_{22} - R_{33}}} \tag{2.43}$$

$$q_2 = \frac{1}{2}\frac{R_{31} - R_{13}}{\sqrt{1 - R_{11} + R_{22} - R_{33}}} \tag{2.44}$$

$$q_3 = \frac{1}{2}\frac{R_{12} - R_{21}}{\sqrt{1 - R_{11} - R_{22} + R_{33}}} \tag{2.45}$$

The time derivative, $\dot{\boldsymbol{q}}$, of the quaternion, is related to $\boldsymbol{w}$ in the inertial frame by

$$\dot{\boldsymbol{q}} = \frac{1}{2}\boldsymbol{W}\left(\boldsymbol{q}\right)^T \cdot \boldsymbol{w} \tag{2.46}$$

where $\boldsymbol{W}\left(\boldsymbol{q}\right)$, the quaternion rate matrix, is

$$\boldsymbol{W}\left(\boldsymbol{q}\right) = \begin{bmatrix} -q_1 & q_0 & -q_3 & q_2 \\ -q_2 & q_3 & q_0 & -q_1 \\ -q_3 & -q_2 & q_1 & q_0 \end{bmatrix} \tag{2.47}$$

Refer to the summary by Diebel (2006) for more details on Euler angles and Euler parameters.

2.11 Restoring Torque

For the linked spherocylinder model, the fibers have an equilibrium shape prescribed by the equilibrium bending and twisting angles, θ^{eq} and ϕ^{eq}, respectively. A restoring torque, $\boldsymbol{Y}_i$, acts on joint i to correct the deviation from the equilibrium shape. The restoring torque

consists of bending and twisting contributions, $\boldsymbol{\Upsilon}_i^b$ and $\boldsymbol{\Upsilon}_i^t$,

$$\boldsymbol{\Upsilon}_i^b = -k_b \left(\theta_i - \theta_i^{\text{eq}}\right) \boldsymbol{e}_i^b \tag{2.48}$$

$$\boldsymbol{\Upsilon}_i^t = -k_t \left(\phi_i - \phi_i^{\text{eq}}\right) \boldsymbol{e}_i^t \tag{2.49}$$

where θ_i and ϕ_i are the bending and twisting angles; k_b and k_t are the bending and twisting constants; and $\boldsymbol{e}_i^b$ and $\boldsymbol{e}_i^t$ specify the direction of the restoring torques. For small deformations, the bending constant and twisting constants are

$$k_b = \frac{E_{\text{Y}} I}{2l}, \qquad k_t = \frac{G J_0}{2l} \tag{2.50}$$

where $I = \pi b^4/4$ is the area moment of inertia, and $J_0 = 2I$ is the polar moment of inertia. The shear modulus, G, for a linearly elastic isotropic material is related to the Young's modulus, E_{Y}, by

$$G = \frac{E_{\text{Y}}}{2\left(1+\nu\right)} \tag{2.51}$$

where ν is Poisson's ratio. For the results reported in this thesis, $\nu = 0.5$ (Mott and Roland, 2009), and $k_t = 0.67 k_b$.

In simulations, the basis for the body frames, $(\hat{\boldsymbol{x}}_i, \hat{\boldsymbol{y}}_i, \boldsymbol{p}_i)$ and $(\hat{\boldsymbol{x}}_{i-1}, \hat{\boldsymbol{y}}_{i-1}, \boldsymbol{p}_{i-1})$, are stored for segments i and $i-1$, respectively. The equilibrium body frame, $\left(\hat{\boldsymbol{x}}_i^{\text{eq}}, \hat{\boldsymbol{y}}_i^{\text{eq}}, \boldsymbol{p}_i^{\text{eq}}\right)$, is calculated by 1) transforming $(\hat{\boldsymbol{x}}_i, \hat{\boldsymbol{y}}_i, \boldsymbol{p}_i)$ to the body frame, then 2) rotating the basis to the equilibrium frame fixed on segment $i-1$, and 3) transforming the basis back to the inertial frame.

$$\left(\hat{\boldsymbol{x}}_i^{\text{eq}}, \hat{\boldsymbol{y}}_i^{\text{eq}}, \boldsymbol{p}_i^{\text{eq}}\right) = \boldsymbol{R}_{i-1}^{\dagger} \cdot \left(\boldsymbol{R}_i^{\text{eq}}\right)^{\dagger} \cdot \boldsymbol{R}_i \left(\hat{\boldsymbol{x}}_i, \hat{\boldsymbol{y}}_i, \boldsymbol{p}_i\right) \tag{2.52}$$

$$= \boldsymbol{R}_i^{\dagger} \cdot \boldsymbol{R}_i^{\text{eq}} \cdot \boldsymbol{R}_{i-1} \left(\hat{\boldsymbol{x}}_{i-1}, \hat{\boldsymbol{y}}_{i-1}, \boldsymbol{p}_{i-1}\right) \tag{2.53}$$

Alternatively, the calculations for $\left(\hat{\boldsymbol{x}}_i^{\text{eq}}, \hat{\boldsymbol{y}}_i^{\text{eq}}, \boldsymbol{p}_i^{\text{eq}}\right)$ can start from $(\hat{\boldsymbol{x}}_{i-1}, \hat{\boldsymbol{y}}_{i-1}, \boldsymbol{p}_{i-1})$. Step 1 is

not necessary but simplifies computation for $\boldsymbol{p}_i^{\text{eq}}$. From Eq. 2.35, it is clear that $\boldsymbol{R}_i \cdot \boldsymbol{p}_i = (0,0,1)^T$.

The bending angle difference is

$$\theta_i - \theta_i^{\text{eq}} = \cos^{-1}\left(\boldsymbol{p}_i \cdot \boldsymbol{p}_i^{\text{eq}}\right) \tag{2.54}$$

The direction of the bending torque is perpendicular to $\boldsymbol{p}_i$ and $\boldsymbol{p}_i^{\text{eq}}$.

$$\boldsymbol{e}_i^b = \frac{\boldsymbol{p}_i \times \boldsymbol{p}_i^{\text{eq}}}{\|\boldsymbol{p}_i \times \boldsymbol{p}_i^{\text{eq}}\|} \tag{2.55}$$

To calculate the twisting angle difference, $\hat{\boldsymbol{y}}_i$ and $\hat{\boldsymbol{y}}_i^{\text{eq}}$ (or $\hat{\boldsymbol{x}}_i$ and $\hat{\boldsymbol{x}}_i^{\text{eq}}$) are first projected to the plane normal to $\boldsymbol{c}_i = (\boldsymbol{r}_i - \boldsymbol{r}_{i-1}) / \|\boldsymbol{r}_i - \boldsymbol{r}_{i-1}\|$

$$\hat{\boldsymbol{y}}_i^{\perp} = \frac{(\boldsymbol{\delta} - \boldsymbol{c}_i\boldsymbol{c}_i) \cdot \hat{\boldsymbol{y}}_i}{\|(\boldsymbol{\delta} - \boldsymbol{c}_i\boldsymbol{c}_i) \cdot \hat{\boldsymbol{y}}_i\|} \qquad \hat{\boldsymbol{y}}_i^{\text{eq}\perp} = \frac{(\boldsymbol{\delta} - \boldsymbol{c}_i\boldsymbol{c}_i) \cdot \hat{\boldsymbol{y}}_i^{\text{eq}}}{\|(\boldsymbol{\delta} - \boldsymbol{c}_i\boldsymbol{c}_i) \cdot \hat{\boldsymbol{y}}_i^{\text{eq}}\|} \tag{2.56}$$

The twisting angle difference is calculated by

$$\phi_i - \phi_i^{\text{eq}} = \cos^{-1}\left(\hat{\boldsymbol{y}}_i^{\perp} \cdot \hat{\boldsymbol{y}}_i^{\text{eq}\perp}\right) \tag{2.57}$$

The direction of the bending torque is

$$\boldsymbol{e}_i^t = \frac{\hat{\boldsymbol{y}}_i^{\perp} \times \hat{\boldsymbol{y}}_i^{\text{eq}\perp}}{\|\hat{\boldsymbol{y}}_i^{\perp} \times \hat{\boldsymbol{y}}_i^{\text{eq}\perp}\|} \tag{2.58}$$

2.12 Scaling

The time scale, t_s, length scale, l_s, force scale, F_s, and torque scale, T_s are

$$l_s = b,\ F_s = 6\pi\eta_0 b l t_s^{-1},\ T_s = 8\pi\eta_0 l^3 t_s^{-1} \tag{2.59}$$

$$t_s = \begin{cases} 1/D_R & \mathrm{Pe} < 10^{-2} \\ 1/\dot{\gamma} & \mathrm{Pe} \geq 10^{-2} \end{cases} \tag{2.60}$$

where the Péclet number, Pe, is

$$\mathrm{Pe} = \frac{\dot{\gamma}}{D_R} \tag{2.61}$$

and D_R, the rotational diffusivity of an isolated fiber, is

$$D_R = \frac{k_\mathrm{B} T}{8\pi\eta_0 b^3 r_p^3 Y_C} \tag{2.62}$$

where Y_C is the scalar resistance function in Eq. 2.16 (Kim and Karilla, 1991). The scalings for the various variables are summarized in Tab. 2.1. Non-dimensionalized quantities are denoted with superscripted asterisks.

Table 2.1: Scalings used to nondimensionalize the various dimensional variables.

Scale	Dimensional Variables
F_s	$\boldsymbol{F}^{\mathrm{hyd}}$, $\boldsymbol{F}^{\mathrm{Br}}$, $\boldsymbol{F}^{\mathrm{con}}$, $\boldsymbol{X}$, F^{rep}, F^{att}
T_s	$\boldsymbol{T}^{\mathrm{hyd}}$, $\boldsymbol{T}^{\mathrm{Br}}$, $\boldsymbol{T}^{\mathrm{con}}$, $\boldsymbol{Y}$, k_b, k_t
$6\pi\eta_0 l$	$\boldsymbol{A}$
$8\pi\eta_0 l^3$	$\boldsymbol{C}$, $\tilde{\boldsymbol{H}}$
l_s	$\boldsymbol{G}$, h,
$1/l_s$	a^{rep}, a^{att},
$1/t_s$	$\boldsymbol{\Omega}^\infty$, $\boldsymbol{E}^\infty$, $\boldsymbol{w}$, $\dot{\boldsymbol{q}}$
l_s/t_s	$\boldsymbol{U}^\infty$, $\dot{\boldsymbol{r}}$

The non-dimensionalized repulsive component of the normal force in Eq. 2.28 can be expressed by absorbing $F^{\mathrm{rep}*}$ into the exponent.

$$\boldsymbol{F}_{ij}^{\mathrm{rep}*} = -\exp\left[-a^{\mathrm{rep}*}\left(h_{ij}^* - \frac{\ln F^{\mathrm{rep}*}}{a^{\mathrm{rep}*}}\right)\right] \tag{2.63}$$

Increasing $F^{\mathrm{rep}*}$ effectively increases the size of the fibers by $\ln F^{\mathrm{rep}*}/a^{\mathrm{rep}*}$. For consistency, $F^{\mathrm{rep}*} = 150$, $a^{\mathrm{rep}*} = 20$, and $a^{\mathrm{att}*} = 35$ for simulations reported in this work. When $t_s = \dot{\gamma}^{-1}$, the repulsive force, F^{rep} scales with the hydrodynamic force, which is small

compared to thermal forces at small Pe. To prevent excessive fiber overlaps and to have a meaningful time scale for simulations without shear, $t_s = D_R^{-1}$ for Pe$< 10^2$.

The dimensionless bending constant, k_b^*, is

$$k_b^* = \frac{k_b t_s}{8\pi\eta_0 l^3} = \frac{N_{\text{seg}}^4}{\pi} S_{\text{eff}} \tag{2.64}$$

where S_{eff} is the effective stiffness obtained by substituting $k_b = E_Y I/2l$ and $l = L/2N_{\text{seg}}$ in Eq. 2.64.

$$S_{\text{eff}} = \frac{E_Y I t_s}{\eta_0 L^4} \tag{2.65}$$

The effective stiffness quantifies the flexibility of the fibers and is relevant only for the linked rigid spherocylinder model.

2.13 Solution Methods for Brownian Rigid Spherocylinders

The solution methods for the Brownian rigid spherocylinder model (Eq. 2.4 and 2.5) are summarized, including the implementation of the Brownian forces and torques and integration scheme. The non-dimensionalized equations of motion are

$$\dot{\boldsymbol{r}}_i^* = \boldsymbol{U}_i^{\infty *} + (\boldsymbol{A}_i^*)^{-1} \cdot \left(\boldsymbol{F}_i^{\text{Br}*} + \sum_j^{N_{Ri}} \boldsymbol{F}_{ij}^{\text{con}*} \right) \tag{2.66}$$

$$\boldsymbol{w}_i^* = \boldsymbol{\Omega}_i^{\infty *} - \frac{Y^H}{Y^C} (\boldsymbol{E}_i^{\infty *} \cdot \boldsymbol{p}_i) \times \boldsymbol{p}_i + (\boldsymbol{C}_i^*)^{-1} \cdot \left(\boldsymbol{T}_i^{\text{Br}*} + \frac{3}{4r_p^2} \sum_j^{N_{Ri}} \tilde{\boldsymbol{G}}_{ij}^* \cdot \boldsymbol{F}_{ij}^{\text{con}*} \right) \tag{2.67}$$

where N_{Ri} is the number of fibers with separation within the maximum interaction separation, h_{nor}^*, from fiber i. The tilde notation is a shorthand for converting a cross product,

$\boldsymbol{u} \times \boldsymbol{v}$, into a dot product, $\tilde{\boldsymbol{u}} \cdot \boldsymbol{v}$, where

$$\tilde{\boldsymbol{u}} = \begin{bmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_x & 0 \end{bmatrix} \tag{2.68}$$

Inserting Eq. 2.67 into Eq. 2.46 gives an integrable equation for the Euler parameters.

$$\dot{\boldsymbol{q}}_i^* = \frac{1}{2}\boldsymbol{W}\left(\boldsymbol{q}\right)^T \cdot \left[\boldsymbol{\Omega}_i^{\infty *} - \frac{Y^H}{Y^C}\left(\boldsymbol{E}_i^{\infty *} \cdot \boldsymbol{p}_i\right) \times \boldsymbol{p}_i + \left(\boldsymbol{C}_i^*\right)^{-1} \cdot \left(\boldsymbol{T}_i^{\mathrm{Br}*} + \frac{3}{4r_p^2}\sum_j^{N_{Ci}} \tilde{\boldsymbol{G}}_{ij}^* \cdot \boldsymbol{F}_{ij}^{\mathrm{con}*}\right)\right] \tag{2.69}$$

Simulations in this work employ simple shear with the flow, vorticity, and gradient direction in the x, y, and z directions, respectively. Thus

$$\boldsymbol{U}_i^{\infty *} = \left(z^* \dot{\gamma} t_s, 0, 0\right)^T \tag{2.70}$$

$$\boldsymbol{\Omega}_i^{\infty *} = \frac{1}{2}\nabla^* \times \boldsymbol{U}_i^{\infty *} = \left(0, 0.5\dot{\gamma} t_s, 0\right)^T \tag{2.71}$$

$$\boldsymbol{E}_i^{\infty *} = 0.5\left[\left(\nabla^* \boldsymbol{U}_i^{\infty *}\right) + \left(\nabla^* \boldsymbol{U}_i^{\infty *}\right)^{\mathrm{T}}\right] = \begin{pmatrix} 0 & 0 & 0.5\dot{\gamma} t_s \\ 0 & 0 & 0 \\ 0.5\dot{\gamma} t_s & 0 & 0 \end{pmatrix} \tag{2.72}$$

For $\mathrm{Pe} \geq 10^{-2}$, $\dot{\gamma} t_s = 1$; the shear strain, defined as $\gamma = \dot{\gamma} t$, is equivalent to $t^* = t/t_s$. For $\mathrm{Pe} < 10^{-2}$, $\dot{\gamma} t_s = \mathrm{Pe}$ and $\gamma = t^* \mathrm{Pe}$.

The Lees-Edwards periodic boundary condition (Lees and Edwards, 1972), illustrated in Fig. 2.5, is implemented using the algorithm described by Allen and Tildesley (2017). While moving in the gradient direction, the point of re-entrance is corrected in the flow direction by $\pm \dot{\gamma} L_{\mathrm{box}} \Delta t$ when the boundaries at $z = \mp L_{\mathrm{box}}/2$ are crossed, where L_{box} is the simulation box side length, and z is the position in the gradient direction.

The Brownian force and torque are implemented by approximating the Dirac-Delta

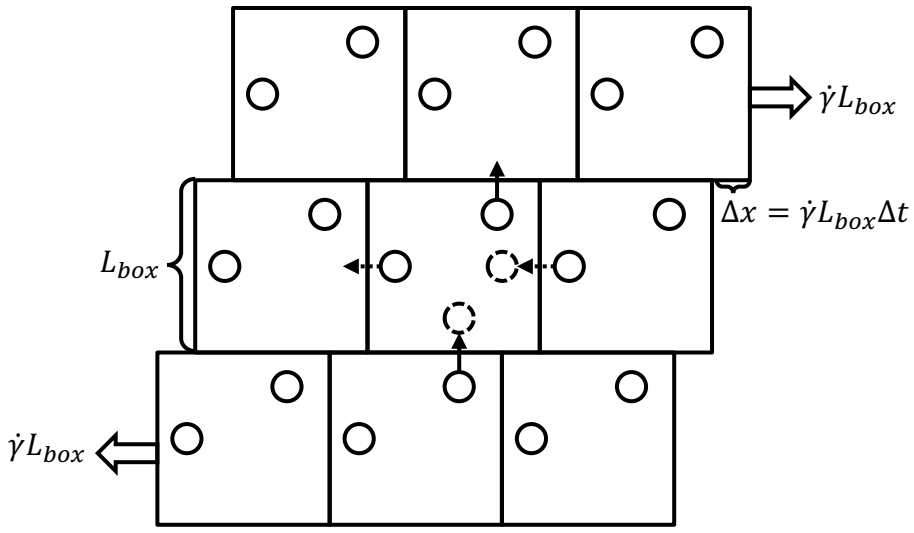


Figure 2.5: Schematic of the Lees-Edwards periodic boundary condition.

function as

$$\delta(t^*) = \frac{1}{\Delta t^*} \tag{2.73}$$

where Δt^* is the simulation time step. The dimensionless Brownian force and torque are

$$\boldsymbol{F}_i^{\mathrm{Br}*} = \sqrt{\frac{k_B T t_s}{3\pi\eta_0 b^2 l \Delta t^*}}\,(\boldsymbol{A}_i^*)^{1/2} \cdot \boldsymbol{\sigma}_{F,i} \tag{2.74}$$

$$\boldsymbol{T}_i^{\mathrm{Br}*} = \sqrt{\frac{k_B T t_s}{4\pi\eta_0 l^3 \Delta t^*}}\,(\boldsymbol{C}_i^*)^{1/2} \cdot \boldsymbol{\sigma}_{T,i} \tag{2.75}$$

where $\boldsymbol{\sigma}_{F,i}$ and $\boldsymbol{\sigma}_{T,i}$ are random vectors with components generated from Gaussian distributions that satisfy

$$\langle \boldsymbol{\sigma}_F \rangle = \mathbf{0}, \qquad \langle \boldsymbol{\sigma}_F \boldsymbol{\sigma}_F \rangle = \boldsymbol{\delta} \tag{2.76}$$

$$\langle \boldsymbol{\sigma}_T \rangle = \mathbf{0}, \qquad \langle \boldsymbol{\sigma}_T \boldsymbol{\sigma}_T \rangle = \boldsymbol{\delta} \tag{2.77}$$

In simulations, the ensemble average, $\langle \cdots \rangle$, is typically approximated by averaging over

fibers and time.

$$\langle \cdots \rangle = \frac{1}{t^* N_{\text{fib}}} \sum_{t^{*\prime}=0}^{t^*} \sum_{i=1}^{N_{\text{fib}}} (\cdots) \tag{2.78}$$

The square root of the resistance tensors are calculated using Eq. 2.38 with $p = 1/2$.

$$(\boldsymbol{A}_i^*)^{1/2} = \sqrt{Y^A}\boldsymbol{\delta} + (\sqrt{X^A} - \sqrt{Y^A})\boldsymbol{p}_i\boldsymbol{p}_i \tag{2.79}$$

$$(\boldsymbol{C}_i^*)^{1/2} = \sqrt{Y^C}\boldsymbol{\delta} + (\sqrt{X^C} - \sqrt{Y^C})\boldsymbol{p}_i\boldsymbol{p}_i \tag{2.80}$$

For $\text{Pe} < 10^{-2}$, where $t_s = D_R^{-1}$, the dimensionless Brownian force and torque simplify to

$$\boldsymbol{F}_i^{\text{Br}*} = \sqrt{\frac{8r_p^2 Y_C}{3\Delta t^*}} (\boldsymbol{A}_i^*)^{1/2} \cdot \boldsymbol{\sigma}_{F,i} \tag{2.81}$$

$$\boldsymbol{T}_i^{\text{Br}*} = \sqrt{\frac{2Y_C}{\Delta t^*}} (\boldsymbol{C}_i^*)^{1/2} \cdot \boldsymbol{\sigma}_{T,i} \tag{2.82}$$

The equations of motion are integrated using a second order Adams-Bashforth algorithm.

$$\boldsymbol{r}_i(t + \Delta t) = \boldsymbol{r}_i(t) + [1.5\dot{\boldsymbol{r}}_i(t) - 0.5\dot{\boldsymbol{r}}_i(t - \Delta t)]\,\Delta t \tag{2.83}$$

$$\boldsymbol{q}_i(t + \Delta t) = \boldsymbol{q}_i(t) + [1.5\dot{\boldsymbol{q}}_i(t) - 0.5\dot{\boldsymbol{q}}_i(t - \Delta t)]\,\Delta t \tag{2.84}$$

After the integration, $\boldsymbol{q}_i$ is renormalized so that $\|\boldsymbol{q}_i\| = 1$. Refer to Appendix B for details on running simulations on the GPU via CUDA.

2.14 Solution Methods for Non-Brownian Linked Rigid Spherocylinders

The solution method in this section follows the approximate method of Switzer (2002). The non-dimensionalized equations of motion are

$$\dot{\boldsymbol{r}}_i^* = \boldsymbol{a}_i^* + (\boldsymbol{A}_i^*)^{-1} \cdot \left(\sum_j^{N_{Ci}} \boldsymbol{F}_{ij}^{\text{fric}*} + \boldsymbol{X}_{i+1}^* - \boldsymbol{X}_i^* \right) \tag{2.85}$$

$$\boldsymbol{w}_i^* = \boldsymbol{b}_i^* + (\boldsymbol{C}_i^*)^{-1} \cdot \left[\frac{3}{4r_{ps}^2} \sum_j^{N_{Ci}} \tilde{\boldsymbol{G}}_{ij}^* \cdot \boldsymbol{F}_{ij}^{\text{fric}*} + \frac{3}{4r_{ps}} \tilde{\boldsymbol{p}}_i \cdot \left(\boldsymbol{X}_{i+1}^* - \boldsymbol{X}_i^* \right) \right] \tag{2.86}$$

$$\boldsymbol{a}_i^* = \boldsymbol{U}_i^{\infty *} + (\boldsymbol{A}_i^*)^{-1} \cdot \sum_j^{N_{Ri}} \boldsymbol{F}_{ij}^{N*} \tag{2.87}$$

$$\boldsymbol{b}_i^* = \boldsymbol{\Omega}_i^{\infty *} - \frac{Y^H}{Y^C} \left(\boldsymbol{E}_i^{\infty *} \cdot \boldsymbol{p}_i \right) \times \boldsymbol{p}_i + (\boldsymbol{C}_i^*)^{-1} \cdot \left(\frac{3}{4r_{ps}^2} \sum_j^{N_{Ri}} \tilde{\boldsymbol{G}}_{ij}^* \cdot \boldsymbol{F}_{ij}^{N*} + \boldsymbol{Y}_{i+1}^* - \boldsymbol{Y}_i^* \right) \tag{2.88}$$

where N_{Ri} is the number of fibers with separation within h_{nor}^* from fiber i; N_{Ci} is the number of fibers with separation within h_{fric}^* from fiber i. The cutoff h_{nor}^* is the maximum interaction distance. The cutoff h_{fric}^* is the maximum distance for friction forces. Typically, $h_{\text{fric}}^* = 0.33$ and $h_{\text{nor}}^* = 2h_{\text{fric}}^*$.

The tilde notation, $\boldsymbol{U}_i^{\infty *}$, $\boldsymbol{\Omega}_i^{\infty *}$, and $\boldsymbol{E}_i^{\infty *}$ are defined in Sec. 2.13. Equation 2.86 is made integrable by applying Eq. 2.46.

$$\dot{\boldsymbol{q}}_i^* = \frac{1}{2} \boldsymbol{W}(\boldsymbol{q})^T \cdot \left\{ \boldsymbol{b}_i^* + (\boldsymbol{C}_i^*)^{-1} \cdot \left[\frac{3}{4r_{ps}^2} \sum_j^{N_{Ci}} \tilde{\boldsymbol{G}}_{ij}^* \cdot \boldsymbol{F}_{ij}^{\text{fric}*} + \frac{3}{4r_{ps}} \tilde{\boldsymbol{p}}_i \cdot \left(\boldsymbol{X}_{i+1}^* - \boldsymbol{X}_i^* \right) \right] \right\} \tag{2.89}$$

The only unknown variables are $\boldsymbol{X}^*$ and $\boldsymbol{F}^{\text{fric}*}$, defined as

$$\boldsymbol{X}^* = \begin{pmatrix} \boldsymbol{X}_2 \\ \boldsymbol{X}_3 \\ \vdots \\ \boldsymbol{X}_{N_{\text{fib}}\left(N_{\text{seg}}-1\right)} \end{pmatrix}, \qquad \boldsymbol{F}^{\text{fric}*} = \begin{pmatrix} \boldsymbol{F}_1^{\text{fric}*} \\ \boldsymbol{F}_2^{\text{fric}*} \\ \vdots \\ \boldsymbol{F}_{N_C}^{\text{fric}*} \end{pmatrix} \tag{2.90}$$

where $N_C = \sum N_{Ci}$ is the total number of contacts in the system.

The system of equations used to solve for $\boldsymbol{X}^*$ and $\boldsymbol{F}^{\text{fric}*}$ is

$$\begin{bmatrix} \boldsymbol{\mathcal{R}}_{\text{X}} & \boldsymbol{\mathcal{Z}}_{\text{F}} \\ \boldsymbol{\mathcal{Z}}_{\text{X}} & \boldsymbol{\mathcal{R}}_{\text{F}} \end{bmatrix} \begin{bmatrix} \boldsymbol{X}^* \\ \boldsymbol{F}^{\text{fric}*} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\mathcal{V}}_{\text{X}} \\ \boldsymbol{\mathcal{V}}_{\text{F}} \end{bmatrix} \tag{2.91}$$

where $\boldsymbol{\mathcal{R}}_{\text{X}}$, $\boldsymbol{\mathcal{Z}}_{\text{X}}$, and $\boldsymbol{\mathcal{V}}_{\text{X}}$ are derived from the inextensibility constraint in Sec. 2.14.1; $\boldsymbol{\mathcal{R}}_{\text{F}}$, $\boldsymbol{\mathcal{Z}}_{\text{F}}$, and $\boldsymbol{\mathcal{V}}_{\text{F}}$ are derived from the no relative motion constraint in Eq. 2.30. Inverting a matrix in $\mathbb{R}^{D\times D}$ is computationally expensive with $\mathcal{O}\left(D^3\right)$ time complexity. In Eq. 2.91, $D = 3\left[N_{\text{fib}}\left(N_{\text{seg}}-1\right)+N_C\right]$, which grows quickly with the scale of the system and the concentration of the suspension. The approximate method speeds up computation by solving Eq. 2.91 through a two step process:

$$\boldsymbol{\mathcal{R}}_{\text{F}}\boldsymbol{F}^{\text{fric}*} = \boldsymbol{\mathcal{V}}_{\text{F}} - \boldsymbol{\mathcal{Z}}_{\text{X}}\boldsymbol{X}^*_{[\text{prev}]} \tag{2.92}$$

$$\boldsymbol{\mathcal{R}}_{\text{X}}\boldsymbol{X}^* = \boldsymbol{\mathcal{V}}_{\text{X}} - \boldsymbol{\mathcal{Z}}_{\text{F}}\boldsymbol{F}^{\text{fric}*} \tag{2.93}$$

where $\boldsymbol{X}^*_{[\text{prev}]}$ contains the inextensibility constraint forces from the previous time step. Once $\boldsymbol{X}^*$ and $\boldsymbol{F}^*_{\text{fric}}$ are known, Eqs. 2.85 and 2.89 are integrated using Eqs. 2.83 and 2.84. The Euler parameters are renormalized so that $\|\boldsymbol{q}_i\| = 1$ for all segments.

2.14.1 *Implementation of Inextensibility Constraint*

The inextensibility constraint is expressed in terms of the velocities by taking the time derivative of Eq. 2.29.

$$\dot{\boldsymbol{r}}_i^* - \dot{\boldsymbol{r}}_{i+1}^* + r_{ps}\left(\boldsymbol{w}_i^* \times \boldsymbol{p}_i + \boldsymbol{w}_{i+1}^* \times \boldsymbol{p}_{i+1}\right) = \boldsymbol{0} \tag{2.94}$$

where $\boldsymbol{w}_i^* \times \boldsymbol{p}_i$ is the time derivative of $\boldsymbol{p}_i$. Substituting Eqs. 2.85 and 2.86 into Eq. 2.94 gives

$$\boldsymbol{\mathcal{S}}_i \cdot \boldsymbol{X}_i^* + \boldsymbol{\mathcal{T}}_i \cdot \boldsymbol{X}_{i+1}^* + \boldsymbol{\mathcal{U}}_i \cdot \boldsymbol{X}_{i+2}^* + \sum_k^{N_{C_i}} \boldsymbol{\mathcal{W}}_{ik}^- \cdot \boldsymbol{F}_{ik}^{\text{fric}*} + \sum_k^{N_{C_{i+1}}} \boldsymbol{\mathcal{W}}_{i+1\ l}^+ \cdot \boldsymbol{F}_{i+1\ l}^{\text{fric}*} +$$

$$\boldsymbol{a}_i^* - \boldsymbol{a}_{i+1}^* - r_p\left(\tilde{\boldsymbol{p}}_i \cdot \boldsymbol{b}_i^* + \tilde{\boldsymbol{p}}_{i+1} \cdot \boldsymbol{b}_{i+1}^*\right) = 0$$

where $\boldsymbol{\mathcal{S}}_i$, $\boldsymbol{\mathcal{T}}_i$, $\boldsymbol{\mathcal{U}}_i$, $\boldsymbol{\mathcal{W}}_i^-$, and $\boldsymbol{\mathcal{W}}_i^+$ are functions that only depend on the orientation and the momentum arm for the contact force,

$$\boldsymbol{\mathcal{S}}_i = -\left(\boldsymbol{A}_i^*\right)^{-1} - \frac{3}{4Y^C}\tilde{\boldsymbol{p}}_i^2, \tag{2.95}$$

$$\boldsymbol{\mathcal{T}}_i = \left(\boldsymbol{A}_i^*\right)^{-1} + \left(\boldsymbol{A}_{i+1}^*\right)^{-1} - \frac{3}{4Y^C}\left(\tilde{\boldsymbol{p}}_i^2 + \tilde{\boldsymbol{p}}_{i+1}^2\right), \tag{2.96}$$

$$\boldsymbol{\mathcal{U}}_i = -\left(\boldsymbol{A}_{i+1}^*\right)^{-1} - \frac{3}{4Y^C}\tilde{\boldsymbol{p}}_{i+1}^2, \tag{2.97}$$

$$\boldsymbol{\mathcal{W}}_{ik}^- = \left(\boldsymbol{A}_i^*\right)^{-1} - \frac{3}{4r_{ps}Y^C}\tilde{\boldsymbol{p}}_i \cdot \tilde{\boldsymbol{G}}_{ik}^*, \tag{2.98}$$

$$\boldsymbol{\mathcal{W}}_{i+1\ l}^+ = -\left(\boldsymbol{A}_{i+1}^*\right)^{-1} - \frac{3}{4r_{ps}Y^C}\tilde{\boldsymbol{p}}_{i+1} \cdot \tilde{\boldsymbol{G}}_{i+1\ l}^*. \tag{2.99}$$

The matrix $\mathcal{R}_X$ is tridiagonal,

$$\mathcal{R}_X = \begin{bmatrix} \mathcal{T}_1 & \mathcal{U}_1 & \mathbf{0} & \dots & & \\ \mathcal{S}_2 & \mathcal{T}_2 & \mathcal{U}_2 & \mathbf{0} & \dots & \\ \mathbf{0} & \mathcal{S}_3 & \mathcal{T}_3 & \mathcal{U}_3 & \mathbf{0} & \dots \\ \vdots & & \ddots & \ddots & \ddots & \\ & & & & & \\ & & \dots & \mathbf{0} & \mathcal{S}_{N_{\text{seg}}-1} & \mathcal{T}_{N_{\text{seg}}-1} \end{bmatrix} \tag{2.100}$$

The quantity $\left(\mathcal{V}_X - \mathcal{Z}_F \cdot \boldsymbol{F}^{\text{fric}*}\right)_i$ is

$$\left(\mathcal{V}_X - \mathcal{Z}_F \cdot \boldsymbol{F}^{\text{fric}*}\right)_i = -\Bigg[\boldsymbol{a}_i^* - \boldsymbol{a}_{i+1}^* - r_p\left(\tilde{\boldsymbol{p}}_i \cdot \boldsymbol{b}_i^* + \tilde{\boldsymbol{p}}_{i+1} \cdot \boldsymbol{b}_{i+1}^*\right) + \sum_k^{N_{C_i}} \mathcal{W}_{ik}^- \cdot \boldsymbol{F}_{ik}^{\text{fric}*} + \sum_k^{N_{C_{i+1}}} \mathcal{W}_{i+1\ l}^+ \cdot \boldsymbol{F}_{i+1\ l}^{\text{fric}*}\Bigg] \tag{2.101}$$

2.14.2 *Implementation of No Relative Motion*

The non-dimensionalized relative velocity difference at the point of contact of segments i and j is

$$\Delta \boldsymbol{u}_{ij}^* = \dot{\boldsymbol{r}}_i^* - \dot{\boldsymbol{r}}_j^* + \boldsymbol{w}_i^* \times \boldsymbol{G}_{ij}^* - \boldsymbol{w}_j^* \times \boldsymbol{G}_{ji}^* \tag{2.102}$$

Substituting in Eqs. 2.85 and 2.86 into the constraint of no relative motion in the plane of contact (Eq. 2.30) gives

$$\left(\boldsymbol{\mathcal{Q}}_i^{\dagger} \cdot \boldsymbol{e}_{ij}^{1 \text{ or } 2}\right) \cdot \boldsymbol{X}_i^* + \left(\boldsymbol{\mathcal{Q}}_i^{'\dagger} \cdot \boldsymbol{e}_{ij}^{1 \text{ or } 2}\right) \cdot \boldsymbol{X}_{i+1}^* + \tag{2.103}$$

$$\left(\boldsymbol{\mathcal{M}}_j^{\dagger} \cdot \boldsymbol{e}_{ij}^{1 \text{ or } 2}\right) \cdot \boldsymbol{X}_j^* + \left(\boldsymbol{\mathcal{M}}_j^{'\dagger} \cdot \boldsymbol{e}_{ij}^{1 \text{ or } 2}\right) \cdot \boldsymbol{X}_{j+1}^* +$$

$$\sum_k^{N_{C_i}} \left(\boldsymbol{\mathcal{J}}_{ik}^{\dagger} \cdot \boldsymbol{e}_{ij}^{1 \text{ or } 2}\right) \cdot \boldsymbol{F}_{ik}^{\text{fric}*} + \sum_l^{N_{C_j}} \left(\boldsymbol{\mathcal{J}}_{jl}^{'\dagger} \cdot \boldsymbol{e}_{ij}^{1 \text{ or } 2}\right) \cdot \boldsymbol{F}_{jl}^{\text{fric}*} +$$

$$\left(\boldsymbol{a}_i^* - \boldsymbol{a}_j^* - \tilde{\boldsymbol{G}}_{ij}^* \cdot \boldsymbol{b}_i^* + \tilde{\boldsymbol{G}}_{ji}^* \cdot \boldsymbol{b}_j^*\right) \cdot \boldsymbol{e}_{ij}^{1 \text{ or } 2} = 0$$

where $\boldsymbol{\mathcal{Q}}_i$, $\boldsymbol{\mathcal{Q}}_i^{'}$, $\boldsymbol{\mathcal{M}}_i$, $\boldsymbol{\mathcal{M}}_i^{'}$, $\boldsymbol{\mathcal{J}}_i$, and $\boldsymbol{\mathcal{J}}_i^{'}$ are functions that only depend on the orientation and momentum arm for the contact force:

$$\boldsymbol{\mathcal{Q}}_i = -\left(\boldsymbol{A}_i^*\right)^{-1} - \frac{3}{4r_{ps}Y^C}\tilde{\boldsymbol{G}}_{ij}^* \cdot \tilde{\boldsymbol{p}}_i, \tag{2.104}$$

$$\boldsymbol{\mathcal{Q}}_i^{'} = \left(\boldsymbol{A}_i^*\right)^{-1} - \frac{3}{4r_{ps}Y^C}\tilde{\boldsymbol{G}}_{ij}^* \cdot \tilde{\boldsymbol{p}}_i, \tag{2.105}$$

$$\boldsymbol{\mathcal{M}}_j = \left(\boldsymbol{A}_j^*\right)^{-1} + \frac{3}{4r_{ps}Y^C}\tilde{\boldsymbol{G}}_{ji}^* \cdot \tilde{\boldsymbol{p}}_j, \tag{2.106}$$

$$\boldsymbol{\mathcal{M}}_j^{'} = -\left(\boldsymbol{A}_j^*\right)^{-1} + \frac{3}{4r_{ps}Y^C}\tilde{\boldsymbol{G}}_{ji}^* \cdot \tilde{\boldsymbol{p}}_j, \tag{2.107}$$

$$\boldsymbol{\mathcal{J}}_{ik} = \left(\boldsymbol{A}_i^*\right)^{-1} - \frac{3}{4r_{ps}^2}\tilde{\boldsymbol{G}}_{ij}^* \cdot \left(\boldsymbol{C}_i^*\right)^{-1} \cdot \tilde{\boldsymbol{G}}_{ik}^*, \tag{2.108}$$

$$\boldsymbol{\mathcal{J}}_{jl}^{'} = -\left(\boldsymbol{A}_j^*\right)^{-1} + \frac{3}{4r_{ps}^2}\tilde{\boldsymbol{G}}_{ji}^* \cdot \left(\boldsymbol{C}_j^*\right)^{-1} \cdot \tilde{\boldsymbol{G}}_{jl}^*. \tag{2.109}$$

The matrix $\boldsymbol{\mathcal{R}}_{\mathrm{F}}$ is constructed for each group of contacting segments of size N_{Cs}

$$\boldsymbol{\mathcal{R}}_{\mathrm{F}} = \begin{bmatrix} \boldsymbol{\mathcal{R}}_{\mathrm{F},11} & \boldsymbol{\mathcal{R}}_{\mathrm{F},12} & \dots & \boldsymbol{\mathcal{R}}_{\mathrm{F},1N_{Cs}} \\ \boldsymbol{\mathcal{R}}_{\mathrm{F},21} & \boldsymbol{\mathcal{R}}_{\mathrm{F},22} & \dots & \boldsymbol{\mathcal{R}}_{\mathrm{F},2N_{Cs}} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{\mathcal{R}}_{\mathrm{F},N_{Cs}1} & \boldsymbol{\mathcal{R}}_{\mathrm{F},N_{Cs}2} & \dots & \boldsymbol{\mathcal{R}}_{\mathrm{F},N_{Cs}N_{Cs}} \end{bmatrix} \tag{2.110}$$

The diagonal elements of $\boldsymbol{\mathcal{R}}_{\mathrm{F}}$ for the contact between segment i and j are

$$\boldsymbol{\mathcal{R}}_{\mathrm{F},ij,ji} = \begin{bmatrix} \left(\boldsymbol{\mathcal{J}}_{ij}^{\dagger} - \boldsymbol{\mathcal{J}}_{ij}^{\prime\dagger}\right) \cdot \boldsymbol{e}_{ij}^{1} \\ \left(\boldsymbol{\mathcal{J}}_{ij}^{\dagger} - \boldsymbol{\mathcal{J}}_{ij}^{\prime\dagger}\right) \cdot \boldsymbol{e}_{ij}^{2} \\ \boldsymbol{n}_{ij} \end{bmatrix} \tag{2.111}$$

which is evident by setting $k = j$ and $l = i$ in Eq. 2.103. The off-diagonal elements of $\boldsymbol{\mathcal{R}}_{\mathrm{F}}$ account for the connectivity of the contacting segments by incorporating the indirect contribution to friction, *i.e.* cases where $k \neq j$ and $l \neq i$ in Eq. 2.103. If segment i is also in contact with segment m, the off-diagonal elements of $\boldsymbol{\mathcal{R}}_{\mathrm{F}}$ are

$$\boldsymbol{\mathcal{R}}_{\mathrm{F},ij,im} = \xi \begin{bmatrix} \boldsymbol{\mathcal{J}}_{im}^{\dagger} \cdot \boldsymbol{e}_{ij}^{1} \\ \boldsymbol{\mathcal{J}}_{im}^{\dagger} \cdot \boldsymbol{e}_{ij}^{2} \\ \boldsymbol{0} \end{bmatrix}. \tag{2.112}$$

If segment j is also in contact with segment m, the off-diagonal elements of $\boldsymbol{\mathcal{R}}_{\mathrm{F}}$ are

$$\boldsymbol{\mathcal{R}}_{\mathrm{F},ij,jm} = \xi \begin{bmatrix} \boldsymbol{\mathcal{J}}_{jm}^{\prime\dagger} \cdot \boldsymbol{e}_{ij}^{1} \\ \boldsymbol{\mathcal{J}}_{jm}^{\prime\dagger} \cdot \boldsymbol{e}_{ij}^{2} \\ \boldsymbol{0} \end{bmatrix}. \tag{2.113}$$

The rule for the sign, $\xi \in \{+1, -1\}$, is summarized in Tab. 2.2, which arises from substitutions utilized when $j > i$,

$$\boldsymbol{F}_{ji}^{\text{fric}*} = -\boldsymbol{F}_{ji}^{\text{fric}*}, \tag{2.114}$$

$$\boldsymbol{n}_{ji} = \boldsymbol{n}_{ij} \tag{2.115}$$

$$g_{ji}^{*} = g_{ij}^{*} \tag{2.116}$$

With the substitutions,

$$\boldsymbol{G}_{ij}^{*} = s_{ij}^{*}\boldsymbol{p}_i + \frac{g_{ij}^{*}}{2}\boldsymbol{n}_{ij} \tag{2.117}$$

$$\boldsymbol{G}_{ji}^{*} = s_{ji}^{*}\boldsymbol{p}_j - \frac{g_{ij}^{*}}{2}\boldsymbol{n}_{ij} \tag{2.118}$$

The corresponding right hand side for the contact between segment i and j is

$$\left(\boldsymbol{\mathcal{V}}_{\text{F}} - \boldsymbol{\mathcal{Z}}_{\text{X}}\boldsymbol{X}_{[\text{prev}]}^{*}\right)_{ij} = \begin{pmatrix} \boldsymbol{\mathcal{I}}_{ij} \cdot \boldsymbol{e}_{ij}^{1} \\ \boldsymbol{\mathcal{I}}_{ij} \cdot \boldsymbol{e}_{ij}^{2} \\ \boldsymbol{0} \end{pmatrix} \tag{2.119}$$

where

$$\begin{aligned} \boldsymbol{\mathcal{I}}_{ij} = -\Big[& \boldsymbol{\mathcal{Q}}_i \cdot \left(\boldsymbol{X}_{[\text{prev}]}^{*}\right)_i + \boldsymbol{\mathcal{Q}}_i^{'} \cdot \left(\boldsymbol{X}_{[\text{prev}]}^{*}\right)_{i+1} \\ & + \boldsymbol{\mathcal{M}}_j \cdot \left(\boldsymbol{X}_{[\text{prev}]}^{*}\right)_j + \boldsymbol{\mathcal{M}}_j^{'} \cdot \left(\boldsymbol{X}_{[\text{prev}]}^{*}\right)_{j+1} \\ & \boldsymbol{a}_i^{*} - \boldsymbol{a}_j^{*} - \tilde{\boldsymbol{G}}_{ij}^{*} \cdot \boldsymbol{b}_i^{*} + \tilde{\boldsymbol{G}}_{ji}^{*} \cdot \boldsymbol{b}_j^{*} \Big]. \end{aligned} \tag{2.120}$$

Refer to Switzer (2002) for more details on how to build $\boldsymbol{\mathcal{R}}_{\text{F}}$.

Table 2.2: Off-diagonal element of $\boldsymbol{\mathcal{R}}_{\mathrm{F}}$.

Case	Friction Force	ξ	$\boldsymbol{\mathcal{J}}$ or $\boldsymbol{\mathcal{J}}^{'}$
$i < m$	$\boldsymbol{F}_{im}^{\mathrm{fric}*}$	1	$\boldsymbol{\mathcal{J}}_{im}$
$i > m$	$\boldsymbol{F}_{mi}^{\mathrm{fric}*}$	-1	$\boldsymbol{\mathcal{J}}_{im}$
$j < m$	$\boldsymbol{F}_{jm}^{\mathrm{fric}*}$	1	$\boldsymbol{\mathcal{J}}^{'}_{im}$
$j > m$	$\boldsymbol{F}_{mj}^{\mathrm{fric}*}$	-1	$\boldsymbol{\mathcal{J}}^{'}_{im}$

2.15 Computational Algorithm

In the previous sections, detailed descriptions for evaluating the constraint forces are described. Here, computational algorithms for initialization, contact identification, regrowing fibers from the center of mass, and postprocessing are included.

2.15.1 *Initialization*

A total of N_{fib} fibers are placed in the simulation box, with side lengths L_x^*, L_y^*, and L_z^*. To avoid self-interaction from periodic boundary condition, the side lengths satisfy $\min\left(L_x^*, L_y^*, L_z^*\right) \geq 3r_p$. Each fiber consists of N_{seg} segments. For linked rigid spherocylinders, N_{seg} is typically set to 5. The fibers are placed with random positions and orientations without overlaps in the simulation box. For the Brownian spherocylinder model, fibers are placed on a rectangular lattice in some cases. The volume fraction is

$$\Phi = \frac{1}{L_x^* L_y^* L_z^*}\left(\frac{4}{3}\pi + 2\pi r_p\right). \tag{2.121}$$

The dimensionless number density is

$$nL^3 = \frac{N_{\mathrm{fib}} L^3}{L_x^* L_y^* L_z^*}. \tag{2.122}$$

The parameters, N_{fib} and side lengths, are chosen based on the desired Φ and available computational resources.

2.15.2 *Contact Identification*

Identifying fiber contacts and the separation distances is required to calculate the contact forces. The naive approach evaluates the distance between every pair of fibers and takes $\mathcal{O}\left(N_{\text{fib}}^2\right)$ time. The Verlet list (neighbor list) and the cell list in Fig. 2.6 are used to speed up computation (Allen and Tildesley, 2017). In this study, the Verlet list is used for the Brownian spherocylinder model; the cell list is used for the linked spherocylinder model. The Verlet lists and cell lists can also be combined to suit shared-memory and parallel multi-core architecture (Gonnet, 2012). The parallel versions of the algorithms are summarized in Appendix B.

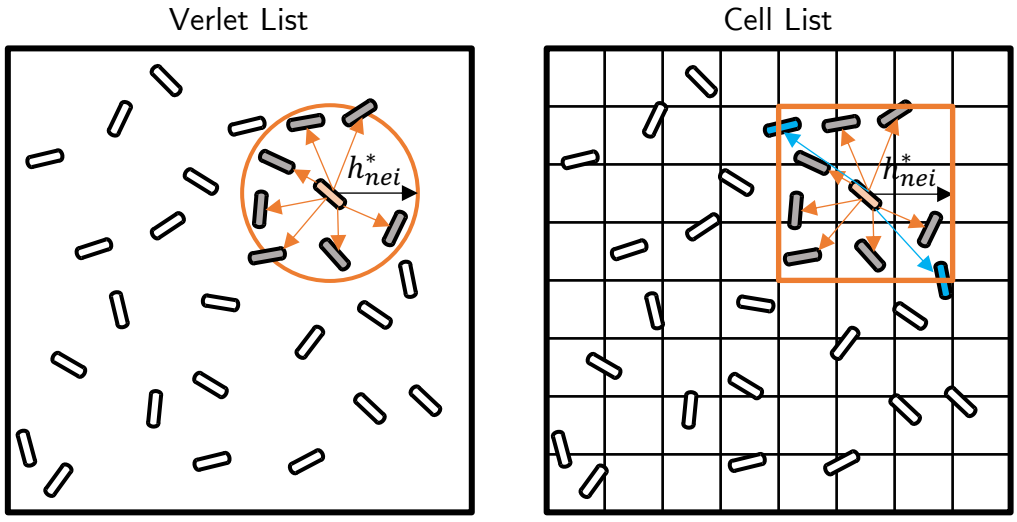


Figure 2.6: Schematic of Verlet and cell lists.

The Verlet list contains fibers within a cutoff, h^*_{ver}, that is larger than h^*_{rep} and is updated every n steps. The parameters, h^*_{ver} and n, can be optimized with the constraint that fibers stay within h^*_{ver} in n moves. Only the separation distances between pairs of fibers in the Verlet list are calculated in a time step. As N_{fib} increases, the $\mathcal{O}\left(N_{\text{fib}}^2\right)$ time distributed over n steps and the demand for large memory become undesirable.

The cell list is constructed by dividing the system into $M \times M \times M$ equal sized bins. Fibers are assigned to the bins based on the coordinates of the fibers. For an example in

1D, a fiber with a position of 3 is assigned to the second bin when the bin lengths are 2. A loop through all the fibers is required, and thus the assignment takes $\mathcal{O}\left(N_{\text{fib}}\right)$ time. A bin contains an average of $C = N_{\text{fib}}/M^3$ fibers, independent of N_{fib}. Separations between segment surfaces are only evaluated for fiber pairs within neighboring bins. The $\mathcal{O}\left(C^2\right)$ computation time scales better for large systems. The traditional cell list contains $3 \times 3 \times 3$ bins. For parallel implementations, $5 \times 5 \times 5$ and $7 \times 7 \times 7$ bins are also employed.

For the linked spherocylinder model, identifying groups of fiber in contact are required to solve for the frictional forces. When fibers are identified to be in contact, *i. e.* the separation distance is less than h^*_{fric}, an algorithm similar to Union Find is implemented. If neither fiber belongs to a group, the fibers are placed in a new group. If only one fiber has a group, the fiber without a group is added to the group of the other fiber. If the fibers belong to different groups, the groups are combined.

The parallel version of the algorithm first compiles a list of fibers, L_1, in direct contact with each fiber. A leader of a group is chosen to be the fiber that has the smallest index and identified via iterative breadth first search. A new list, L_2, is created during iterative search, containing L_1 and other fibers encountered that are indirectly connected to the leader. The leader then establishes the group connectivity going through L_1 for each fiber in its L_2.

2.15.3 *Regrowing Fibers*

For linked fibers, the center of mass of a fiber, $\boldsymbol{R}_{\text{cm}}$ is integrated similarly to Eq. 2.83

$$\boldsymbol{R}^*_{\text{cm}}(t + \Delta t) = \boldsymbol{R}^*_{\text{cm}}(t) + \left[1.5\dot{\boldsymbol{R}}^*_{\text{cm}}(t) - 0.5\dot{\boldsymbol{R}}^*_{\text{cm}}(t - \Delta t)\right] \Delta t \tag{2.123}$$

where the velocity of the center of mass is the average of the segment velocities.

$$\boldsymbol{R}^*_{\text{cm}} = \frac{1}{N_{\text{seg}}} \sum_{i=1}^{N_{\text{seg}}} \dot{\boldsymbol{r}}^*_i \tag{2.124}$$

The Euler parameters for segment i, $\boldsymbol{q}_i$, are integrated according to Eq. 2.84. The orientation, $\boldsymbol{p}_i$ is obtained from the third row of the rotation matrix in Eqs. 2.35 and 2.41. The fiber is then regrown from $\boldsymbol{R}^*_{\text{cm}}$.

$$\boldsymbol{r}^*_1 = \boldsymbol{R}^*_{\text{cm}} - \frac{r_{ps}}{N_{\text{seg}}} \sum_{i=2}^{N_{\text{seg}}} \left(\boldsymbol{p}_1 + \boldsymbol{p}_i + 2 \sum_{j=2}^{i-1} \boldsymbol{p}_j \right), \tag{2.125}$$

$$\boldsymbol{r}^*_{i\neq 1} = \boldsymbol{r}^*_1 + r_{ps}\boldsymbol{p}_1 + 2r_{ps} \sum_{j=2}^{i-1} \boldsymbol{p}_j + r_{ps}\boldsymbol{p}_i. \tag{2.126}$$

This approach eliminates numerical errors and ensures that the fiber are inextensible.

2.15.4 *Postprocessing*

Stress

The total stress is

$$\langle \boldsymbol{\sigma} \rangle = -p\boldsymbol{\delta} + \langle \boldsymbol{\sigma}_P \rangle + 2\eta_0 \boldsymbol{E}^\infty \tag{2.127}$$

where p is the isotropic pressure, and $\boldsymbol{\sigma}_P$ is the particle contribution to stress defined as

$$\langle \boldsymbol{\sigma}_P \rangle = -\frac{1}{V} \sum_{i=1}^{N} \boldsymbol{r}_i \boldsymbol{F}^{\text{tot}}_i + \frac{1}{V} \sum_{i=1}^{N} \boldsymbol{S}_i \tag{2.128}$$

where V is the volume of the suspension, $N = N_{\text{fib}} N_{\text{seg}}$, the total number of segments, and $\boldsymbol{F}^{\text{tot}}_i$ is the sum of non-hydrodynamic forces on segment i (Bossis and Brady, 1989; Wilson and Klingenberg, 2017). The stresslet, $\boldsymbol{S}$, exerted by the fluid on a fiber segment is given by the slender body theory (Bossis and Brady, 1989)

$$\boldsymbol{S} = \frac{1}{2} \int_{-l}^{l} [s\boldsymbol{p}\boldsymbol{F}(s) + s\boldsymbol{F}(s)\boldsymbol{p}]\, ds - \frac{2}{3}\boldsymbol{\delta} \int_{-l}^{l} s\boldsymbol{p} \cdot \boldsymbol{F}(s) ds \tag{2.129}$$

where $\boldsymbol{F}(s)$ is the force per unit length. Batchelor (1970b) derived an expression for $\boldsymbol{F}(s)$ for slender bodies ($r_p \gg 1$)

$$\boldsymbol{F}(s) = \frac{4\pi\eta_0}{ln(2r_{ps})}\left[\boldsymbol{\delta} - \frac{1}{2}\boldsymbol{pp}\right] \cdot [\boldsymbol{u}^{\infty}(s) - \boldsymbol{u}(s)] \tag{2.130}$$

where $\boldsymbol{u}(s)$ is the velocity at position s along the segment axis. For linear flow, $\boldsymbol{u} = \dot{\boldsymbol{r}} + s\dot{\boldsymbol{p}}$. The nondimensionalized particle contribution stress is thus

$$\begin{aligned}\frac{\langle \boldsymbol{\sigma}_P \rangle \, t_s}{\eta_0} = &-\frac{6r_{ps}\pi}{V^*}\sum_{i=1}^{N} \boldsymbol{r}_i^* \boldsymbol{F}_i^{\mathrm{tot}*} \\ &+ \frac{4\pi r_{ps}^3}{3\ln\left(2r_{ps}\right) V^*}\sum_{i=1}^{N}\Big[\left(\boldsymbol{\nabla U}^{\infty *}\right)^T \cdot \boldsymbol{p}_i\boldsymbol{p}_i + \boldsymbol{p}_i\boldsymbol{p}_i \cdot \left(\boldsymbol{\nabla U}^{\infty *}\right. \\ &- \boldsymbol{p}_i\boldsymbol{p}_i\boldsymbol{p}_i\boldsymbol{p}_i : \left(\boldsymbol{\nabla U}^{\infty *}\right) - \left(\boldsymbol{p}_i\dot{\boldsymbol{p}}_i^* + \dot{\boldsymbol{p}}_i^*\boldsymbol{p}_i\right)\Big] + \text{I.T.}\end{aligned} \tag{2.131}$$

where I.T. contains isotropic terms of no interest, *i.e.*, does not contribute to the shear stress.

Stored Elastic Energy

The stored elastic energy that quantifies the deformation of a fiber from its equilibrium shape is

$$E^{\mathrm{elas}} = \frac{1}{2}k_b \sum_{i=2}^{N_{\mathrm{seg}}} \left(\theta_i - \theta_i^{\mathrm{eq}}\right)^2 + k_t \left(\phi_i - \phi_i^{\mathrm{eq}}\right)^2 \tag{2.132}$$

Diffusivity

The diffusivities can be obtained from the fiber positions and orientations as functions of time (Ford, 1981; Lowen, 1994). The translational diffusivites, D_T, is the slope of the mean-square-displacement

$$\left\langle \left(\boldsymbol{r}(t+t') - \boldsymbol{r}(t')\right)^2 \right\rangle = 6D_T t \tag{2.133}$$

The aftereffect operator (Ford, 1981), simplified for the special case of symmetric top (Bette, 2003), for which only diagonal elements of $\langle \boldsymbol{R} \rangle$ are nonzero is

$$\left\langle \boldsymbol{R}_A(t+t') \right\rangle = \left\langle \boldsymbol{p}(t')\boldsymbol{p}(t') \right\rangle^{-1} \cdot \left\langle \boldsymbol{p}(t')\boldsymbol{p}(t+t') \right\rangle \tag{2.134}$$

The rotational diffusivity, D_R, is related to the aftereffect operator by (Ford, 1981)

$$Tr\left(\left\langle \boldsymbol{R}_A(t+t') \right\rangle\right) = 3e^{-2D_R t} \tag{2.135}$$

This form of D_R only accounts for rotation of the major spherocylinder axis.

Order Parameters

The order parameter is used characterize suspensions of Brownian spherocylinders, which have one segment per fiber. The order parameter, S, is the largest eigenvalue of the $\boldsymbol{Q}$ tensor; $\boldsymbol{n}$ is the corresponding eigenvector (Andrienko, 2018),

$$\langle \boldsymbol{Q} \rangle = \frac{1}{N_{\text{fib}}} \sum_i^{N_{\text{fib}}} \left(\boldsymbol{p}_i \boldsymbol{p}_i - \frac{1}{3}\boldsymbol{I} \right) = S\left(\boldsymbol{nn} - \boldsymbol{I}\right) + B\left(\boldsymbol{ll} - \boldsymbol{mm}\right) \tag{2.136}$$

where B is the biaxiality of the fiber distribution; $\boldsymbol{l}$ and $\boldsymbol{m}$ form a local orthonormal basis with $\boldsymbol{n}$. For uniaxial nematic materials, $B = 0$.

The hexatic order parameter for a fiber j quantifies the extent to which its six closest neighbors in a layer are arranged on a triangular lattice (Houssa et al., 1998; Zaluzhnyy et al., 2017),

$$\psi_j = \frac{1}{6} \sum_{k=1}^{6} e^{i6\theta_{kj}} \tag{2.137}$$

where θ_{kj} is the angle between $\Delta \boldsymbol{r}_{kj} = \boldsymbol{r}_k - \boldsymbol{r}_j$ and a reference axis in the plane that defines the layer (see Fig. 2.7). The hexatic order parameter for the suspension is

$$\langle \psi \rangle = \frac{1}{N_{\text{fib}}} \sum_{j=1}^{N_{\text{fib}}} \psi_j \tag{2.138}$$

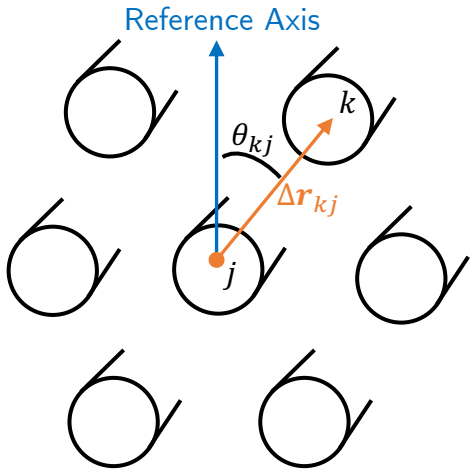


Figure 2.7: Schematic of hexatic order parameter angle.

Pair Distribution

Pair distribution functions quantify the structure of the suspension. The radial pair distribution function, $g(r)$, is proportionalto the probability density of finding a fiber at distance r from another fiber. All directions from the central fiber are treated equally; the histogram used to calculate $g(r)$ is constructed from spherical shells as illustrated in Fig. 2.8. Pair distributions parallel and perpendicular to the director $\boldsymbol{n}$ (see Sec. 2.15.4) are used to distinguish liquid crystalline phases (Houssa et al., 1998). The parallel distribution function, $g_{\parallel}(r)$, is constructed from a histogram of thin discs stacked in the direction of $\boldsymbol{n}$. The perpendicular function, $g_{\perp}(r)$, is constructed from a histogram of cylindrical shell bins with axes aligned with $\boldsymbol{n}$. The length of the cylindrical shells is chosen to be at most r_p to capture only one layer of smectic or solid phases.

Cluster Analysis

A cluster is defined as a group of connected fibers, *i.e.*, each fiber can reach another in the group by a series of fiber contacts. A labeling technique developed by Hoshen and Kopelman (1976) is used to identify clusters. Given the fiber positions, $\boldsymbol{r}_i$, and orientation, $\boldsymbol{p}_i$, a connectivity matrix of size $N_{\text{fib}} \times N_{\text{fib}}$ containing 0s and 1s is generated. A 1 in the

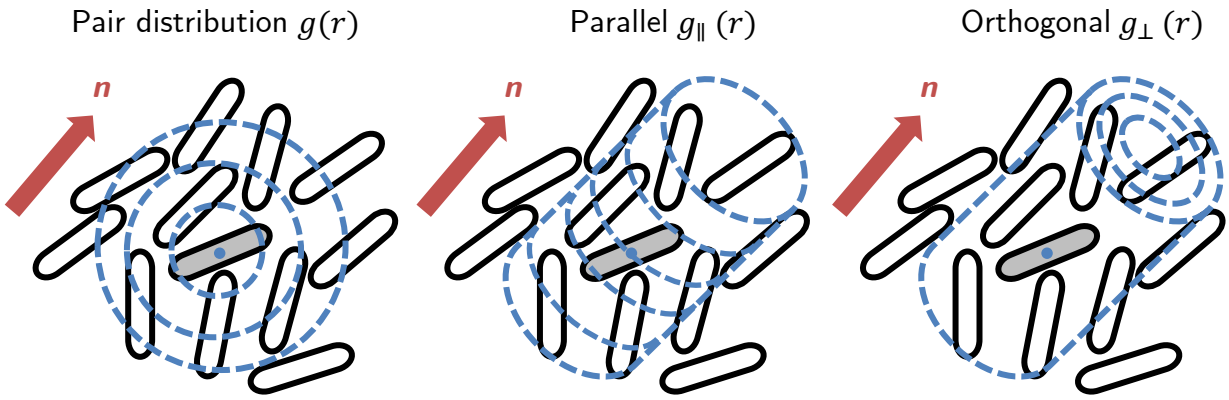


Figure 2.8: Schematic of the bins used for radial, parallel, and orthogonal pair distributions.

(m, n) entry indicates that fibers m and n are in contact through any segment pair. A 0 indicates otherwise. The connectivity matrix is used to iteratively assign a group label to the fibers. The cluster center of mass is

$$\boldsymbol{r}^c_{\mathrm{cm,i}} = \frac{1}{N^c_{\mathrm{fib}}} \sum_i^{N^c_{\mathrm{fib}}} \boldsymbol{r}_{\mathrm{cm,i}} \tag{2.139}$$

where N^c_{fib} is the number of fibers in the cluster.

A cluster, at times, crosses the simulation box, and thus has fibers on opposite ends of the simulation box because of periodic boundaries. The center of mass obtained by directly applying Eq. 2.139 can be incorrect, and even outside of the cluster in some cases. To calculate the correct center of mass under the periodic boundary conditions, the fibers in a cluster are first identified by a pass through the labeling algorithm. The 26 mirror images of those fibers are then included for another pass through the labeling algorithm. Equation 2.139 can then be applied to the cluster with the largest number of fibers. Breaking the cluster identification into two passes reduces the computational time, and ensures that all the clusters identified in the second pass are from the same original cluster.

The size of the clusters is characterized by the gyration tensor

$$\boldsymbol{G}_y = \frac{1}{N_{\text{fib}}^c} \sum_{i}^{N_{\text{fib}}^c} \left(\boldsymbol{r}_{\text{cm,i}} - \boldsymbol{r}_{\text{cm,i}}^c\right) \left(\boldsymbol{r}_{\text{cm,i}} - \boldsymbol{r}_{\text{cm,i}}^c\right) \tag{2.140}$$

When the cluster is compact and take roughly a rectangular shape, the volume of the cluster can be approximated as $\lambda_1\lambda_2\lambda_3$, where the λs are the eigenvalues of $\boldsymbol{G}_y$. The radius of gyration is

$$R_g = \sqrt{\lambda_1^2 + \lambda_2^2 + \lambda_3^2} \tag{2.141}$$

Intensity of Segregation

Danckwerts (1952) defined the intensity of segregation, I, to quantify the extent of mixing in reactors.

$$I = \frac{\sigma_\Phi^2}{\Phi\left(1 - \Phi\right)} \tag{2.142}$$

where σ_Φ^2 is the variance in volume fraction. The system is divided into N_{bin} cubic bins. Fiber segments are placed in the bins based on the centers of mass of the segments. The local volume fraction, Φ_{loc}, is the volume fraction of the bin, and

$$\sigma_\Phi^2 = \frac{1}{N_{\text{bin}}} \sum_{i=1}^{N_{\text{bin}}} \left(\Phi_{\text{loc},i} - \Phi\right)^2 \tag{2.143}$$

The bin side length is $1.5r_{ps}$ for results reported in this study, and can be adjusted to capture structures of different scales.

3

RHEOLOGY AND STRUCTURE OF SUSPENSIONS OF SPHEROCYLINDERS VIA BROWNIAN DYNAMICS SIMULATIONS

3.1 Introduction

Rod-like colloidal particles are abundant in nature and commonly synthesized. Examples include the tobacco mosaic virus (Fraden et al., 1989), cellulose nanocrystals (Klemm et al., 2018), bimetallic rods (Hong et al., 2007), and polymethyl methacrylate ellipsoids with uniform size (Mohraz and Solomon, 2005). At sufficiently high concentrations, colloidal suspensions of rod-like particles can form liquid crystalline phases, such as nematic and cholesteric mesophases and the smectic A phase (Wissbrun, 1981). Flory (1956) showed that particle asymmetry and excluded volume interactions can cause suspensions of sufficiently long rod-like colloidal particles to form an anisotropic (nematic) phase.

Non-Brownian rod-like particles show interesting behavior in shear flow. Jeffery (1922) derived expressions for the periodic rotation of a spheroid in laminar shear flow. For a prolate spheroid, the major fiber axis rotates with a period independent of orientation, and spends more time nearly aligned with the flow direction. Elongated particles of other shapes exhibit the same dynamics, which can be described by Jeffery's equation with an equivalent aspect ratio (Bretherton, 1962; Goldsmith and Mason, 1962; Cox,

1970, 1971). Similar orbits remain for weakly Brownian rod-like particles (Leal and Hinch, 1971).

The rheology of liquid crystalline suspensions of rod-like colloidal particles is characterized by a three-region flow curve, first proposed by Onogi and Asada (1980). The flow curve contains two shear thinning regions that bracket a viscosity plateau at intermediate shear rates. Three structural models are used to interpret the regions: a piled polydomain at low shear rates, a dispersed polydomain at intermediate shear rates, and a continuous monodomain at high shear rates (Asada et al., 1980). The domains for suspensions with higher concentrations are smaller and require a higher shear rate to reach a continuous phase.

Rod-like colloidal particles are often modeled as hard spherocylinders. Monte Carlo simulation studies have examined plain spherocylinders in suspension (McGrother et al., 1996), spherocylinders with flexible end points and terminal dipoles (Van Duijneveldt et al., 2000), mixtures of spherocylinders with different aspect ratios (Cinacchi et al., 2004), and attraction between cylinders from depletion forces by polymer addition (Bolhuis et al., 1997). The effects of friction (Nath and Heussinger, 2019) and aggregates (Kobayashi et al., 2020) on the rheological properties have been explored in simulations. Brownian dynamics simulations, or broadly Stokesian dynamics simulations (Brady and Bossis, 1988), of hard spherocylinders have been employed to obtain dynamic properties, such as the translational and rotational diffusivities (Lowen, 1994; Tao et al., 2005).

Cellulose nanocrystals (CNCs), in particular, have become of interest in the past decade, as evidenced in the large number of reviews and a rapidly growing number of publications (Klemm et al., 2018; De Souza Lima and Borsali, 2004; Azizi Samir et al., 2005; Habibi et al., 2010; Eichhorn et al., 2010; Moon et al., 2011). Cellulose nanocrystal is a biopolymer, with a diameter of 6-8 nm (Habibi et al., 2008), obtained from acid hydrolysis or oxidation of cellulose (Moon et al., 2011). The aspect ratio is usually less than 70 (Lavoine et al., 2012). Applications of CNCs include mechanically reinforced nanocomposites (Kassab et al., 2019; Jardin et al., 2020), electronics (Kim et al., 2019), elec-

trochemical sensors (Khalilzadeh et al., 2020), and membranes for CO_2 separation from flue gas (Torstensen et al., 2019). One of the main challenges in the commercialization of CNC applications is that the CNCs must be dried for economic transport. However, it is very difficult to redisperse the CNCs and achieve the desired properties because the interparticle forces drive particle aggregation (Bagheriasl et al., 2016, 2019).

Similarly to other rod-like polymers, CNC suspensions form liquid crystalline phases (Ishii et al., 2019; Lagerwall et al., 2014; Dong et al., 2002), gel above critical concentration that depends on the aspect ratio, and exhibit the three-region flow curve (Bercea and Navard, 2000; Shafiei-Sabet et al., 2012) described above. Factors that affect the rheology and structure of CNC suspensions include the amount of energy used to redisperse suspensions (Shafiei-Sabet et al., 2012), temperature (Heggset et al., 2017), processing methods (Li et al., 2015), surface properties (Shafiei-Sabet et al., 2013; Samyn and Taheri, 2016; Moberg et al., 2017), concentration (Wu et al., 2014; Xu et al., 2017), pH (Xu et al., 2017), salts (Dong et al., 2002; Xu et al., 2017; Phan-Xuan et al., 2016; Shafiei-Sabet et al., 2014), aspect ratio (Li et al., 2015; Moberg et al., 2017; Wu et al., 2014), and polymer additives (Bagheriasl et al., 2016).

In this work, CNC fibers are modeled as soft spherocylinders. Brownian dynamics simulations are employed to investigate the structure and rheology of suspensions of soft spherocylinders with only repulsive interactions. Translational and rotational diffusivities, and liquid cyrstalline phase diagram qualitatively match results from simulations of hard spherocylinders. This agreement helps to validate the simulation method, and also suggests that the repulsive interaction is sufficiently stiff to mimic a hard spherocylinder interaction. The structure and rheological properties qualitatively agree with previously reported experimental results for CNC suspensions. This somewhat surprising result suggests that the rheological properties, to a large extent, are determined just by the repulsive interactions. Transient rheological properties and fiber domains that arise from shearing suspensions that are isotropic and nematic at rest are also investigated.

3.2 Methods

3.2.1 *Model*

The model employed here for suspensions of fibers is similar to that employed by Schmid and Klingenberg (2000a) and Switzer and Klingenberg (2003), except that here the fibers are not linked, friction is omitted, and Brownian motion is added. Fibers are modeled as spherocylinders with radius b, diameter D, length L, and aspect ratio defined as $r_p = L/D$ (McGrother et al., 1996; Eken et al., 2012) (see Fig. 3.1).

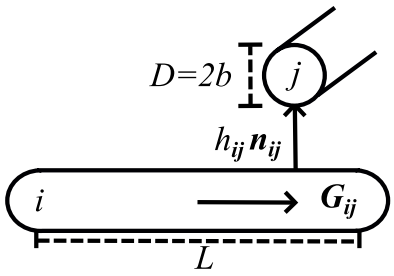


Figure 3.1: Schematic diagram of fibers modeled as spherocylinders with length L, diameter D, and radius b. The shortest separation between fiber surfaces is h_{ij}, and $\boldsymbol{n}_{ij}$ is the direction normal to both surfaces. The moment arm for the torque arising from repulsive interactions is $\boldsymbol{G}_{ij}$, which points from the center of the fiber to the location of the contact along the fiber axis.

Neglecting inertia, the translational and rotational equations of motion are

$$\boldsymbol{F}_i^{\text{hyd}} + \boldsymbol{F}_i^{\text{br}} + \sum_j^{N_{Ci}} \boldsymbol{F}_{ij}^{\text{rep}} = 0, \tag{3.1}$$

$$\boldsymbol{T}_i^{\text{hyd}} + \boldsymbol{T}_i^{\text{br}} + \sum_j^{N_{Ci}} \boldsymbol{G}_{ij} \times \boldsymbol{F}_{ij}^{\text{rep}} = 0 \tag{3.2}$$

The forces and torques in these equations are described below.

For a fiber with linear and angular velocity $\dot{\boldsymbol{x}}_i$ and $\boldsymbol{w}_i$, the hydrodynamic force and torque are $\boldsymbol{F}_i^{\text{hyd}} = \boldsymbol{A}_i \cdot (\boldsymbol{U}_i^{\infty} - \dot{\boldsymbol{x}}_i)$ and $\boldsymbol{T}_i^{\text{hyd}} = \boldsymbol{C}_i \cdot (\boldsymbol{\Omega}_i^{\infty} - \boldsymbol{w}_i) + \tilde{\boldsymbol{H}}_i : \boldsymbol{E}^{\infty}$, where $\boldsymbol{A}_i$, $\boldsymbol{C}_i$, $\tilde{\boldsymbol{H}}_i$ are the resistance tensors of an isolated prolate spheroid with equivalent aspect ratio

$r_e = 0.7r_p$ (Kim and Karilla, 1991). For simulations under shear, the ambient translational and angular velocity $\boldsymbol{U}_i^\infty$ and $\boldsymbol{\Omega}_i^\infty$ are $(\dot{\gamma}z, 0, 0)^{\mathrm{T}}$ and $(0, \dot{\gamma}/2, 0)^{\mathrm{T}}$, where $\dot{\gamma}$ is the shear rate. The rate of strain tensor $\boldsymbol{E}^\infty$ is $0.5\left[(\nabla \boldsymbol{U}_i^\infty) + (\nabla \boldsymbol{U}_i^\infty)^{\mathrm{T}}\right]$.

Hydrodynamic interactions between fibers are neglected. Sundararajakumar and Koch (1997) found that mechanical contacts dominate over hydrodynamic interactions in controlling the fiber orientation distributions when $nL^3/r_p \geq O(1)$, where n is the number density. Hydrodynamic interactions are not included in the model because most simulations in this study fall in the semi-dilute regime, $10^{-1} \leq nL^3/r_p \leq 10^2$.

The Brownian force $\boldsymbol{F}_i^{\mathrm{br}}$ and torque $\boldsymbol{T}_i^{\mathrm{br}}$ are random vectors with zero mean and variance given by Kubo (1966)

$$\left\langle \boldsymbol{F}^{\mathrm{br}}(t)\boldsymbol{F}^{\mathrm{br}}(t+t')\right\rangle = 2k_{\mathrm{B}}T\boldsymbol{A}\delta(t') \tag{3.3}$$

$$\left\langle \boldsymbol{T}^{\mathrm{br}}(t)\boldsymbol{T}^{\mathrm{br}}(t+t')\right\rangle = 2k_{\mathrm{B}}T\boldsymbol{C}\delta(t') \tag{3.4}$$

where k_{B} is the Boltzmann constant, and T is the absolute temperature.

Fibers interact with short-range repulsive forces to mimic a hard spherocylinder interaction. The repulsive force on fiber i due to fiber j is

$$\boldsymbol{F}_{ij}^{\mathrm{rep}} = -F^{\mathrm{rep}}e^{-ah_{ij}}\boldsymbol{n}_{ij} \tag{3.5}$$

and the associated torque on fiber i is $\boldsymbol{G}_{ij} \times \boldsymbol{F}_{ij}^{\mathrm{rep}}$; the moment arm $\boldsymbol{G}_{ij}$ is illustrated in Fig. 3.1. Here $F^{\mathrm{rep}} = 900\pi\eta_0 b^2 r_p t_s^{-1}$ is the repulsive force scale, a is the decay parameter, h_{ij} is the minimum separation between fiber surfaces, and $\boldsymbol{n}_{ij}$ is the unit surface normal, which points from fiber i to j at the point of contact, which is normal to both surfaces. The decay parameter a determines the repulsive stiffness and is set to $a = 20/b$ for results reported here. For this value of a, there are occasional small overlaps between fiber surfaces. Attractive interactions are not included in this study to determine features of the suspension structure and rheological properties that are caused by only repulsive

interactions

The equations of motion are nondimensionalized by the time scale t_s, length scale l_s, force scale F_s, and torque scale T_s,

$$t_s = \begin{cases} 1/D_R & \text{Pe} < 10^{-2} \\ 1/\dot{\gamma} & 10^{-2} \leq \text{Pe} \end{cases} \tag{3.6}$$

$$l_s = b,\ F_s = 6\pi\eta_0 b^2 r_p t_s^{-1},\ T_s = 8\pi\eta_0 (b r_p)^3 t_s^{-1} \tag{3.7}$$

where the Péclet number is

$$\text{Pe} = \frac{\dot{\gamma}}{D_R} \tag{3.8}$$

and D_R is the rotational diffusivity of an isolated spherocylinder,

$$D_R = \frac{k_B T}{8\pi\eta_0 b^3 r_p^3 Y_C} \tag{3.9}$$

where Y_C is the scalar resistance function that relates hydrodynamic torque and angular velocity (Kim and Karilla, 1991). For results reported here, $10^{-5} \leq \text{Pe} \leq 10^6$, which covers a wide range of possible experimental values of $\dot{\gamma}$ and D_R, such as $10^{-2} \leq \dot{\gamma} \leq 10^5$ s^{-1} and $10^2 \leq D_R \leq 10^5$ s^{-1}. Using the parameter values $\eta_0 = 0.001$ Pa·s, $b = 2.8$ nm, and $r_p = 50$, $Y_C = 0.1$, $t_s = D_R^{-1} = 1.5$ ms, the force and torque scale factors are $F_s = 0.05$ pN and $T_s = 46$ zN·m.

Variables made dimensionless with theses scales are indicated with a superscripted asterisk.

3.2.2 *Simulation Methods*

In this paper, x, y, z are the flow, vorticity, and gradient directions, respectively. Fiber positions and orientations are initialized either by (1) random sequential addition or (2) placing fibers on a rectangular lattice with uniform orientation. The latter method is em-

ployed for more highly concentrated suspensions. Periodic Lees-Edwards sliding boundaries (Lees and Edwards, 1972) are employed. The random Brownian forces and torques are generated from Gaussian distributions. To determine the spherocylinders' positions and orientations as a function of time, the equations of motion are integrated using a second-order Adams-Bashforth algorithm on a GPU,

$$\boldsymbol{x}_i(t+\Delta t) = \boldsymbol{x}_i(t) + [1.5\dot{\boldsymbol{x}}_i(t) - 0.5\dot{\boldsymbol{x}}_i(t-\Delta t)]\,\Delta t \tag{3.10}$$

$$\boldsymbol{q}_i(t+\Delta t) = \boldsymbol{q}_i(t) + [1.5\dot{\boldsymbol{q}}_i(t) - 0.5\dot{\boldsymbol{q}}_i(t-\Delta t)]\,\Delta t \tag{3.11}$$

where $\boldsymbol{x}_i = (x_i, y_i, z_i)$ is the vector location of the center of fiber i, and $\boldsymbol{q}_i$ is the vector of Euler parameters of fiber i, which is directly related to the orientation $\boldsymbol{p}_i$ (Wittenburg, 1977; Haug, 1992; Ross and Klingenberg, 1997).

The number of fibers, N, ranges from $1,000$ to $13,000$. To maintain an approximately cubic box that is sufficiently large, a larger value of N is required for larger values of r_p and ϕ. The smallest simulation box length employed is at least $5L$ for simulations to extract diffusivities and viscosities, and at least $3L$ for simulations to generate phase diagrams.

3.2.3 *Rheological and Dynamic Properties*

The total stress of the system $\langle\boldsymbol{\sigma}\rangle$ is

$$\langle\boldsymbol{\sigma}\rangle = -p\boldsymbol{\delta} + \langle\boldsymbol{\sigma}_P\rangle + 2\eta_0\boldsymbol{E}^{\infty} \tag{3.12}$$

where p is an isotropic pressure and $\langle\boldsymbol{\sigma}_P\rangle$ is the particle contribution to stress

$$\langle\boldsymbol{\sigma}_P\rangle = -\frac{1}{V}\sum_{i=1}^{N}\boldsymbol{x}_i\boldsymbol{F}_i^{\text{tot}} + \frac{1}{V}\sum_{i=1}^{N}\boldsymbol{S}_i \tag{3.13}$$

where V is the volume of the simulation box, and $\boldsymbol{F}_i^{\text{tot}}$ is the sum of non-hydrodynamic forces acting on fiber i (Bossis and Brady, 1989; Wilson and Klingenberg, 2017). The angle brackets, $\langle \dots \rangle$, are ensemble averages estimated by averaging over particles and multiple configurations. The stresslet $\boldsymbol{S}$ exerted by the fluid on the fiber is given by slender body theory (Bossis and Brady, 1989; Switzer and Klingenberg, 2003)

$$\boldsymbol{S}_i = \frac{1}{2}\int_{-l}^{l} [s\boldsymbol{p}_i\boldsymbol{F}(s) + s\boldsymbol{F}(s)\boldsymbol{p}_i]\, ds - \frac{2}{3}\boldsymbol{\delta}\int_{-l}^{l} s\boldsymbol{p}_i \cdot \boldsymbol{F}(s) ds \tag{3.14}$$

where l is the fiber half length and $\boldsymbol{F}(s)$ is the force per unit length. Batchelor (1970a) derived an expression for $\boldsymbol{F}(s)$ for slender bodies ($r_p \gg 1$)

$$\boldsymbol{F}(s) = \frac{4\pi\eta_0}{ln(2r_p)}\left[\boldsymbol{\delta} - \frac{1}{2}\boldsymbol{p}_i\boldsymbol{p}_i\right] \cdot [\boldsymbol{u}^{\infty}(s) - \boldsymbol{u}_i(s)] \tag{3.15}$$

where $\boldsymbol{u}_i(s)$ is the velocity at position s along the fiber axis. Substituting $\boldsymbol{u}_i = \dot{\boldsymbol{x}}_i + s\dot{\boldsymbol{p}}_i$, $\boldsymbol{S}_i$ is expressed for linear flows as

$$\begin{aligned} \boldsymbol{S}_i = \quad & \frac{4\pi\eta_0 l^3}{3ln(2r_p)}\Big[(\boldsymbol{\nabla}\boldsymbol{U}^{\infty})^T \cdot \boldsymbol{p}_i\boldsymbol{p}_i + \boldsymbol{p}_i\boldsymbol{p}_i \cdot (\boldsymbol{\nabla}\boldsymbol{U}^{\infty}) \\ & -\boldsymbol{p}_i\boldsymbol{p}_i\boldsymbol{p}_i\boldsymbol{p}_i : (\boldsymbol{\nabla}\boldsymbol{U}^{\infty}) - (\boldsymbol{p}_i\dot{\boldsymbol{p}}_i + \dot{\boldsymbol{p}}_i\boldsymbol{p}_i)\Big] + \text{I.T.} \end{aligned} \tag{3.16}$$

where I.T. is an isotropic term of no interest.

The relative shear viscosity η_r is $\langle \sigma_{xz} \rangle / \eta_0\dot{\gamma}$. The first and second normal stress differences, $\langle N_1 \rangle$ and $\langle N_2 \rangle$, are

$$\langle N_1 \rangle = \langle \sigma_{xx} \rangle - \langle \sigma_{zz} \rangle \tag{3.17}$$

$$\langle N_2 \rangle = \langle \sigma_{zz} \rangle - \langle \sigma_{yy} \rangle \tag{3.18}$$

In addition to rheological properties, the translational and rotational diffusivities, D_T and D_R, are obtained from the fiber positions and orientations as functions of time

(Lowen, 1994; Ford, 1981). The mean-squared-displacement (MSD) is

$$\left\langle (\boldsymbol{x}(t+t') - \boldsymbol{x}(t'))^2 \right\rangle = 6D_T t \tag{3.19}$$

and thus D_T can be obtained from the slope of the MSD as a function of time. The rotational diffusivity is obtained from the time evolution of the fiber orientations. The aftereffect operator simplified for the special case of a symmetric top is

$$\left\langle \boldsymbol{R}(t+t') \right\rangle = \left\langle \boldsymbol{p}(t')\boldsymbol{p}(t') \right\rangle^{-1} \cdot \left\langle \boldsymbol{p}(t')\boldsymbol{p}(t+t') \right\rangle \tag{3.20}$$

for which only diagonal elements of $\langle \boldsymbol{R} \rangle$ are nonzero (Ford, 1981). The rotational diffusivity is related to the aftereffect operator by

$$Tr\left(\left\langle \boldsymbol{R}(t+t') \right\rangle\right) = 3e^{-2D_R t} \tag{3.21}$$

Thus D_R is obtained from the slope of $\ln Tr(\langle \boldsymbol{R}(t+t') \rangle)$ as a function of time. This form of D_R only accounts for rotation of the major spherocylinder axis.

3.2.4 *Structural Measures*

The degree of orientational order is quantified using the $\boldsymbol{Q}$ tensor

$$\langle \boldsymbol{Q} \rangle = \frac{1}{N} \sum_i^N (\boldsymbol{p}_i \boldsymbol{p}_i - \boldsymbol{I}) = S\,(\boldsymbol{nn} - \boldsymbol{I}) \tag{3.22}$$

where the order parameter S is the largest eigenvalue of $\boldsymbol{Q}$ and $\boldsymbol{n}$ is the corresponding eigenvector (Andrienko, 2018). For prolate objects, $0 \leq S \leq 1$, where $S = 1$ indicates complete alignment in one direction. For isotropic systems, S is near zero, whereas S is near 1 for nematic, smectic, and solid phases.

The radial distribution functions, $g_{\parallel}(r)$ and $g_{\perp}(r)$, characterize the structure parallel and perpendicular to the director, $\boldsymbol{n}$, respectively (Houssa et al., 1998). The parallel

distribution function, $g_{\parallel}(r)$, distinguishes a nematic phase from the smectic and solid phases, and the perpendicular function $g_{\perp}(r)$ helps to distinguish the smectic from the solid phase. A histogram of pair center-to-center separations with thin disc bins stacked in the direction of $\boldsymbol{n}$ is used to obtain $g_{\parallel}$. A histogram with cylindrical shell bins with axes aligned with $\boldsymbol{n}$ is used to obtain $g_{\perp}$. To capture only the distribution within one smectic or solid layer, the length of the concentric shells is chosen to be at most r_p.

The hexatic order parameter, ψ_j, quantifies the extent to which fiber j and its six closest neighbors within a layer are arranged on a triangular lattice (Houssa et al., 1998; Zaluzhnyy et al., 2017),

$$\psi_j = \frac{1}{6}\sum_{k=1}^{6} e^{i6\theta_{kj}} \tag{3.23}$$

where θ_{kj} is the angle between $\Delta\boldsymbol{x}_{kj} = \boldsymbol{x}_k - \boldsymbol{x}_j$ and a reference axis in the plane that defines the layer. The average hexatic order parameter, $\langle\psi\rangle$, for the suspension is

$$\langle\psi\rangle = \frac{1}{N}\sum_{j=1}^{N}\psi_j \tag{3.24}$$

The hexatic order parameter is in the range $0 \leq \psi \leq 1$, where $\psi = 1$ indicates a perfect triangular lattice arrangement and helps to distinguish smectic and solid phases.

The distribution of fibers within a layer can also be characterized by the distribution of angles between a fiber and two of its neighbors, $f(\theta)$. The distribution is obtained from a histogram of angles between a central fiber i and two nearest neighbors j and k, where $\cos\theta_{jk} = \Delta\boldsymbol{x}_{ij}\cdot\Delta\boldsymbol{x}_{ik}/\left|\Delta\boldsymbol{x}_{ij}\right|\left|\Delta\boldsymbol{x}_{ik}\right|$.

3.3 Results

3.3.1 *Diffusivities*

Simulations without shear were conducted to investigate the effect of concentration on the diffusivities and to compare our results with previously reported results for hard

spherocylinders, to validate the model and simulation method. Fibers with $r_p = 5$ were placed on a lattice with $3,610 \leq N \leq 7,688$ to maintain cubic simulation boxes with side lengths at least $5L$. Simulations were run with a dimensionless time step of $\Delta t^* = 10^{-6}$ to a dimensionless time of $t^* = 100$. The order parameter, $\langle S \rangle$, decreases to a steady value by $t^* \lesssim 5$. To allow Brownian motion to sufficiently randomize the suspension, the MSD and $\langle \boldsymbol{R} \rangle$ are calculated using Eqs. 3.19-3.21 with $t'^* = 0$ defined at $t^* = 75$. Since the MSD is only nonlinear for $t'^* \ll 1$, D_T is extracted from the slope over the entire range $0 \leq t'^* \leq 25$; D_R is extracted from $t'^* = 0$ to t'^* such that $\mathrm{Tr}\,\langle \boldsymbol{R}(t'^*) \rangle \leq 0.3$. The diffusivities are normalized by the short-time estimates, which are extracted using Eqs. 3.19 and 3.21 for $0 \leq t'^* \leq 10^{-4}$. The normalized translational and rotational diffusivities, D_T^* and D_R^*, were averaged over 3 runs that started with different seeds.

In Fig. 3.2, D_T^* and D_R^* are plotted as functions of ϕ along with results by Lowen (1994) who fit his simulation results with empirical expressions. For this range of ϕ $(r_p = 5)$, the suspensions are isotropic. At the smallest value of ϕ, D_T^* and D_R^* are approximately 1. As ϕ is increased, D_T^* and D_R^* decrease and approach 0. The results agree with those of Lowen (1994) who performed Brownian dynamics simulations of hard spherocylinders without hydrodynamic interactions. Fibers were moved sequentially with random displacements consistent with the diffusivity of an isolated fiber. Displacements were rejected if the fibers overlapped. Despite the fact that the repulsive force used in this study is a short-ranged, continuous function of separation, as opposed to the hard spherocylinder interaction, the diffusivities match closely over the range of ϕ investigated, which helps to validate the model and simulation method. In addition, the repulsive force with the decay parameter $a = 20/b$ is apparently sufficiently stiff to mimic the hard spherocylinder interaction. Larger a values require too small of a time step to obtain simulation results in a reasonable time.

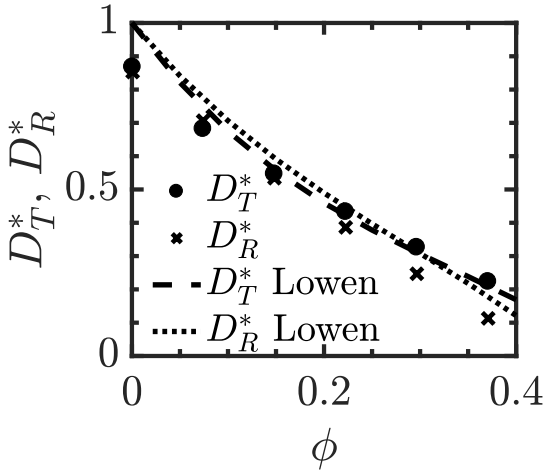
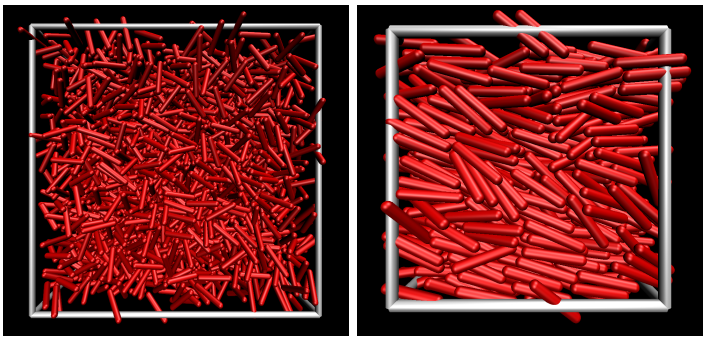


Figure 3.2: Translational and rotational diffusivities normalized by short time estimates, D_T^* and D_R^*, vs. volume fraction, ϕ, for fibers with aspect ratio $r_p = 5$. Also plotted are simulation results obtained by Lowen (1994). The standard deviation over three simulation runs is less than the symbol size.

3.3.2 *Structural Characterization*

Simulations without shear were performed for a range of r_p and ϕ to map the liquid crystalline phases. We compare our results with previously reported phase diagrams for hard spherocylinders, to further validate the model and simulation method. Fibers were initially placed on a rectangular lattice. The simulations were run with $\Delta t^* = 10^{-6}$ to $t^* = 100$.

In Fig. 3.3, simulation snapshots for four liquid crystalline phases with $r_p = 5$ at $t^* = 100$ are presented. (Methods of characterization are illustrated in Fig. 3.4 below.) In Fig. 3.3a, the isotropic phase has no translational or orientational order. The nematic phase is marked by fiber alignment in one direction without translational order as shown in Fig. 3.3b. Both the smectic (Fig. 3.3c) and solid phases (Fig. 3.3d) have translational and orientational order with the fibers organized in layers. In the smectic phase, there is translational disorder within a layer; in the solid phase, there is translational order within a layer. Chiral phases are not observed, possibly because of a lack of asymmetry in the model, *e.g.*, a lack of a helical twist along fiber major axis (Orts et al., 1998).



(a) Isotropic, $\phi = 0.15$ (b) Nematic, $\phi = 0.45$

(c) Smectic, $\phi = 0.52$ (d) Solid, $\phi = 0.70$

Figure 3.3: Simulation snapshots of the four phases observed for aspect ratio $r_p = 5$: (a) the isotropic phase at volume fraction $\phi = 0.15$, (b) the nematic phase at $\phi = 0.45$, (c) the smectic phase at $\phi = 0.52$, (d) the solid phase at $\phi = 0.70$.

The isotropic phase shown in Fig. 3.3a has $\langle S \rangle = 0.015$ for $r_p = 5$. All other phases have $\langle S \rangle \geq 0.75$. The quantitative characterization of the phases described by Eqs. 3.22-3.24 is illustrated in Fig. 3.4. In Fig. 3.4a, $\langle g_{\parallel} \rangle$ is plotted as a function of r/D. The smectic and solid phases are distinguished from the isotropic and nematic phases by the layering that is captured by $\langle g_{\parallel} \rangle$. For the isotropic and nematic phases, $\langle g_{\parallel} \rangle$ is independent of r/D, suggesting that there is no order in the direction parallel to $\boldsymbol{n}$. For the smectic and solid phases, $\langle g_{\parallel} \rangle$ exhibits peaks at $r/D = 0$ and $L/D + 1$. This peak separation shows that regions with a high concentration of fiber centers are one fiber length apart. This observation, along with $\langle S \rangle \geq 0.75$, suggests that fibers are oriented within layers.

The solid phase is distinguished from the smectic phase by degree of order within a layer. In Fig. 3.4b, $\langle g_{\perp} \rangle$ is plotted as a function of r/D. Both the smectic and solid phases exhibit peaks in $\langle g_{\perp} \rangle$ that decay with r/D. The peaks for the solid phase are more

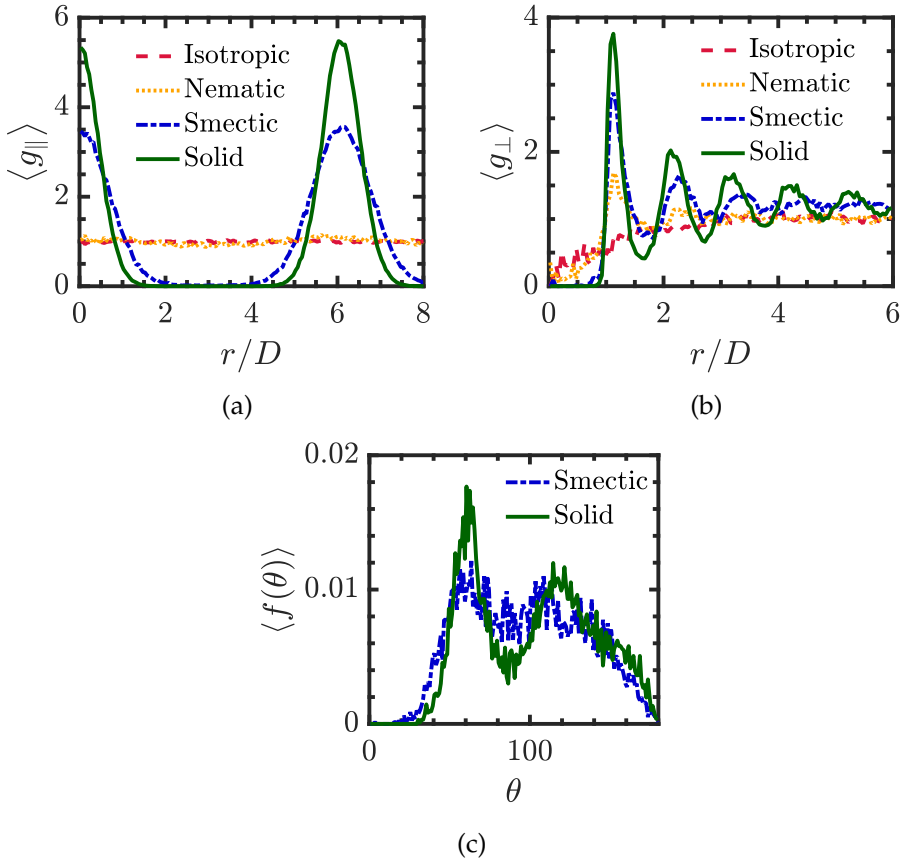


Figure 3.4: Characterization of phases shown in Fig. 3.3: (a) $\langle g_\parallel \rangle$ vs. r/D, (b) $\langle g_\perp \rangle$ vs. r/D, and (c) $\langle f(\theta) \rangle$ vs. θ.

persistent and maintain a larger amplitude with less peak broadening. For the smectic phase, the peaks are difficult to identify beyond $r/D = 4$.

The degree of order of fibers within a layer can be further quantified using $\langle f(\theta) \rangle$ and $\langle \psi \rangle$. In Fig. 3.4c, $\langle f(\theta) \rangle$ is plotted as a function of θ for the smectic and solid phases. The solid phase exhibits peaks at 60° and 120°, indicating the fibers are arranged on a triangular lattice with broadened peaks from Brownian motion. The peaks are less distinct for the smectic phase. The hexatic order parameter, $\langle \psi \rangle$, is 0.37 and 0.79 for the smectic and solid phases in Fig. 3.4c.

A phase diagram was generated using the characterization methods described above for $2 \leq r_p \leq 30$ and $0.001 \leq \phi^* \leq 0.8$, where ϕ^* is the volume fraction normalized by the

close packing volume fraction ϕ_{cp} (Bolhuis and Frenkel, 1997),

$$\phi_{cp} = \frac{2}{\sqrt{2}+\sqrt{3}\frac{L}{D}}\left(\frac{\pi}{6}+\frac{\pi}{4}\frac{L}{D}\right) \tag{3.25}$$

Fibers were initially placed on a rectangular lattice with $1,014 \leq N \leq 24,300$. Simulations without shear were performed with time step $\Delta t^* = 10^{-6}$ to $t^* = 100$.

Figure 3.5a represents a map of phases obtained in the simulation runs. The isotropic phase is identified by $\langle S \rangle \leq 0.22$; the nematic phase has $0.48 \leq \langle S \rangle \leq 1$. The smectic phase is identified by peaks in $\langle g_{\parallel} \rangle$ that are r_p apart, with $\langle g_{\parallel} \rangle$ going to zero in between the peaks. A system that exhibits peaks in $\langle g_{\parallel} \rangle$ without $\langle g_{\parallel} \rangle$ going to zero in between the peaks may be a coexistence of the nematic and smectic phases. The solid phase is identified by $\langle \psi \rangle \geq 0.48$; the smectic phase has $0.35 \leq \langle \psi \rangle \leq 0.42$. Figure 3.5a illustrates that the transitions between the phases are clearly delineated.

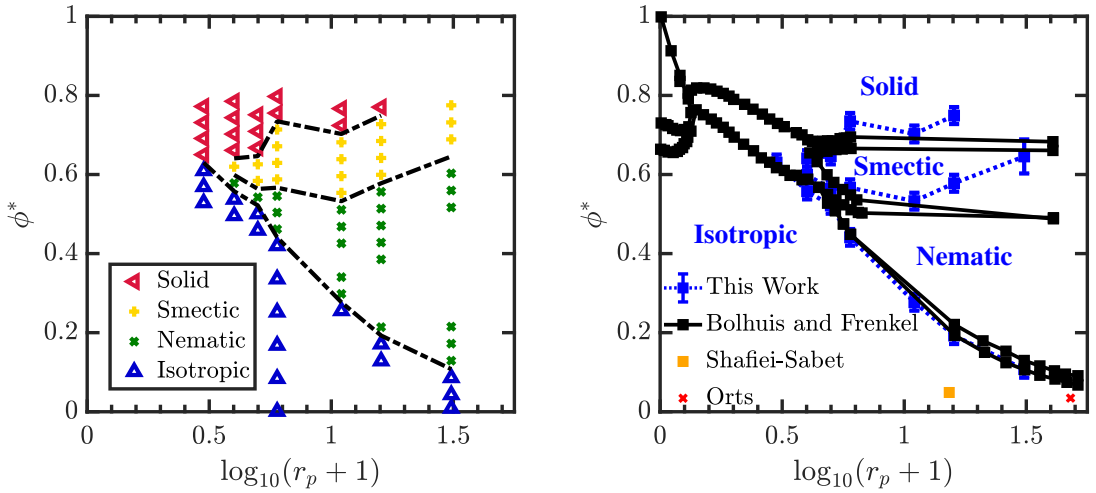


Figure 3.5: Phase diagram of normalized volume fraction, ϕ^*, and the aspect ratio, r_p. (a) Map of the phases obtained in simulations. The dotted lines are estimates of the phase boundaries. (b) The phase diagram plotted along with the results from Bolhuis and Frenkel (1997). The conditions for rheological measurements reported by Shafiei-Sabet et al. (2012) and Orts et al. (1998), which are compared to simulation results below, are also included for reference.

To estimate the value of ϕ^* at the transitions for a given r_p, the values of ϕ^* on each

side of a transition were simply averaged. The uncertainties are half of the ϕ^* differences. The transitions between the phases are illustrated in Fig. 3.5b along with the transitions predicted by Bolhuis and Frenkel (1997) for a true hard spherocylinder system. Here, the value of ϕ^* at the transition for a given r_p is plotted as a function of $\log_{10}(r_p+1)$. The transition from an isotropic to nematic phase occurs at smaller values of ϕ^* as r_p is increased. The value of ϕ^* for the nematic-smectic and smectic-solid phase transition begins to increase with r_p for $\log_{10}(r_p+1) \geq 1$. Bolhuis and Frenkel (1997) performed Monte Carlo simulations of hard spherocylinders and identified phase transitions and coexistence regions using the free energy difference and thermodynamic integration. The nematic-smectic and smectic-solid phase transitions deviate from those obtained by Bolhuis and Frenkel for $\log_{10}(r_p+1) \geq 1.2$. We do not observe a solid for $\log_{10}(r_p+1) = 1.5$, for the range of concentrations investigated. The differences are attributed to the different treatments of excluded volume (the hard spherocylinder interaction versus the continuous repulsive force used here), and are more significant at large values of r_p and ϕ^*. Note that the coexistence regions identified by Bolhuis and Frenkel (1997) are relatively small and on the same order as the uncertainty in ϕ^* at the transitions.

3.3.3 *Rheological Properties and Structure Under Shear*

To investigate the dependence of the rheological properties and structure on Pe for suspensions that are isotropic at rest, simulations under shear were performed for suspensions with $(r_p, \phi) = (50, 0.031)$ and $(14, 0.044)$ over the range $10^{-4} \leq \mathrm{Pe} \leq 10^5$. For each set of parameters, three simulations were performed, employing 3,200 fibers randomly placed in a cubic box with side lengths approximately $3L$. Simulations were run with $\Delta t^* = 10^{-6}$ to $t^* = 100$. As described in Sec. 3.2.1, for $\mathrm{Pe} \geq 10^{-2}$, the time scale is $\dot{\gamma}^{-1}$, and t^* is equivalent to the shear strain. For $\mathrm{Pe} < 10^{-2}$, the time scale is D_R^{-1} and thus the shear strain becomes $\gamma = t^*\mathrm{Pe}$. The relative viscosity, η_r, and order parameter, $\langle S \rangle$, were block averaged with a block size of $\Delta_B t^* = 10$ starting at $t^* = 40$.

In Fig. 3.6, η_r and $\langle S \rangle$ from simulations are plotted as functions of Pe for both $(r_p, \phi) = (50, 0.031)$ and $(14, 0.044)$. For both sets of parameters, η_r decreases with increasing Pe and appears to show flow curves with three regions. Region I (Pe $< 10^{-2}$) is shear thinning. Region II ($10^{-2} \leq$ Pe < 1) shows somewhat of a plateau in the viscosity. The relative viscosity is not constant in this region, similar to that observed experimentally (Shafiei-Sabet et al., 2012; Li et al., 2015), but there is a change in sign of the second derivative of η_r with respect to Pe. Region III (Pe $\geq 10^2$) is again shear thinning, with η_r approaching a large Pe plateau. The variance in η_r grows linearly with Pe^{-1} for small values of Pe. Due to the large uncertainties in the viscosity, it is unclear whether a low Pe viscosity plateau exists (i.e., as Pe approaches 0).

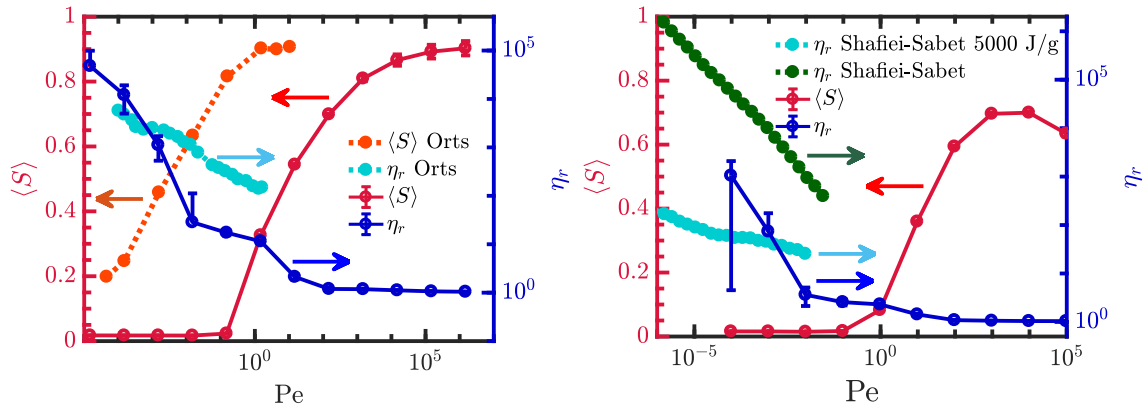


Figure 3.6: Order parameter, $\langle S \rangle$, and relative viscosity, η_r, as a function of Pe for two different values of r_p and ϕ: (a) aspect ratio $r_p = 50$ and volume fraction $\phi = 0.031$, including results reported by Orts et al. (1998), and (b) $r_p = 14$ and $\phi = 0.044$, including results for both non-sonicated and sonicated (5000 J/g) suspensions reported by Shafiei-Sabet et al. (2012). Error bars indicate the 95% confidence interval from block averaging the results for three runs to $t^* = 100$. The block size is $\Delta_B t^* = 10$ and the averaging started at $t^* = 40$.

The order parameter, $\langle S \rangle$, is zero for small values of Pe and increases with Pe for both sets of parameters. At small Pe, thermal forces dominate over hydrodynamic forces and the suspension is isotropic. As Pe is increased, the hydrodynamic forces begin to align fibers in the flow direction and dominate at large Pe. For $1 \leq$ Pe $\leq 10^5$, $\langle S \rangle$ is larger

for $(r_p, \phi) = (50, 0.031)$ (Fig. 3.6a) than for $(r_p, \phi) = (14, 0.044)$ (Fig. 3.6b); this is expected since for dilute suspensions of fibers undergoing Jeffery orbits, fibers with larger values of r_p spend more time roughly oriented in the flow direction. Even at infinite Pe, Jeffery orbits and collisions will produce some degree of disorder, and thus $\langle S \rangle < 1$.

The order and viscosity in simulations are coupled. The shear thinning region at small values of Pe (Region I) occurs with no change in $\langle S \rangle$. The order begins to increase within the viscosity plateau (Region II). The shear thinning region at large values of Pe (Region III) occurs while $\langle S \rangle$ exhibits the largest increase with increasing Pe. Alignment of fibers in the flow direction reduces both the stresslet and the nonhydrodynamic force contributions to the stress.

Orts et al. (1998) measured the order and viscosity of suspensions of fibers with $L =$ 280 nm and $r_p = 47$ at 5 w/w% in water. This data is plotted along with the simulation data in in Fig. 3.6a. This composition and aspect ratio are indicated on the phase diagram in Fig. 3.5b for reference. A cellulose density of 1.6 g/cm^3 (Moon et al., 2013) is used to convert the mass concentration to a volume fraction of $\phi = 0.031$, identical to that for the simulated results in Fig. 3.6a. The behavior of η_r and $\langle S \rangle$ with increasing Pe are similar to that observed in the simulations. The suspensions are shear thinning while $\langle S \rangle$ increases with Pe. The shape of $\langle S \rangle$ as a function of Pe qualitatively agrees with simulations, where both increase with the same slope and reach the same plateau of $\langle S \rangle \approx 0.9$. The experimental data are shifted to smaller Pe by 3 orders of magnitude. The lack of hydrodynamic interactions and attractive forces in the simulations are likely causes for the shift in $\langle S \rangle$. Hydrodynamic interactions can help align the fibers. Attractive forces tend to cause fibers to form a network and can increase the order by the affine deformation of the network.

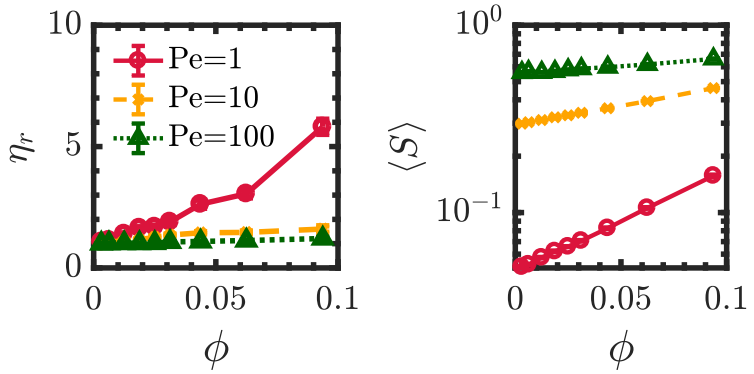
Shafiei-Sabet et al. (2012) measured the viscosity of suspensions of fibers with $L =$ 100 ± 8 nm and $D = 7 \pm 3$ at 7 w/w% in water, which corresponds to $r_p = 14$ and $\phi = 0.044$ for various levels of sonication. The measured viscosity decreased with increasing sonication energy. For all sonication levels at this concentration, Shafiei-Sabet

et al. (2012) reported that the suspension exhibited liquid crystalline domains coexisting with isotropic regions. For this r_p and ϕ, simulations produce isotropic suspensions (Fig. 3.5b). This difference may be attributed to the lack of attractive forces in the simulations. The experimental results for a non-sonicated suspension and a suspension that was sonicated with 5000 J/g are plotted along with simulation results with the same r_p and ϕ in Fig. 3.6b. The experimental systems are both shear thinning over the range of Pe investigated, with the nonsonicated suspensions exhibiting a much larger viscosity. The suspension that was sonicated exhibits the three region flow curve, similar to that observed in the simulations, with a viscosity plateau in the range $5 \times 10^{-5} \leq \text{Pe} \leq 3 \times 10^{-4}$. This agrees qualitatively with the flow curve observed in the simulations, but the plateau in the simulations occurs at larger Pe. It is unclear whether a high Pe viscosity plateau would appear in experiments at larger Pe. The fact that sonication decreases η_r suggests that attractive forces and aggregates were present. Nonetheless, the values of η_r obtained from the simulations are surprisingly similar to those reported for the sonicated suspension, and exhibit similar features in the flow curve, even though only repulsive forces are included in the simulations.

To investigate the dependence of the rheological properties and structure of suspensions with $r_p = 14$ on ϕ, at various Pe, simulations under shear were performed for suspensions with $3 \times 10^{-4} \leq \phi \leq 0.093$ at Pe $= 1$, 10, and 100. All simulations employed $3,200$ fibers, randomly placed in a cubic simulation box. The side lengths were at least $3L$. All suspensions were isotropic at rest. Three simulations were run with $\Delta t^* = 10^{-6}$ to $t^* = 100$ ($\gamma = 100$) for each parameter set. The viscosity and order parameter were blocked averaged with a block size of $\Delta_B t^* = 10$ starting from $t^* = 40$.

In Fig. 3.7, η_r and $\langle S \rangle$ are plotted as a function of ϕ for Pe $= 1$, 10, and 100. Overall, η_r and $\langle S \rangle$ increase with ϕ for all Pe. At the lowest concentration, $\phi = 3 \times 10^{-4}$, $\eta_r \approx 1$ for all Pe, as expected for dilute suspensions (i.e., the particle contribution to the stress is small). As ϕ is increased, η_r increases, more so for smaller Pe; at $\phi = 0.093$, η_r is 5 times larger for Pe $= 1$ than for Pe $= 100$. Similarly, $\langle S \rangle$ increases with ϕ, more so

for smaller Pe. For $\phi = 3 \times 10^{-4}$, $\langle S \rangle$ at Pe $= 100$ is an order of magnitude larger than at Pe $= 1$. For $\phi = 0.093$, $\langle S \rangle$ at Pe $= 100$ is only a factor of 4 larger than at Pe $= 1$. The rheological properties and structure are coupled—the largest increase of η_r with increasing ϕ is associated with the largest relative increase of $\langle S \rangle$.



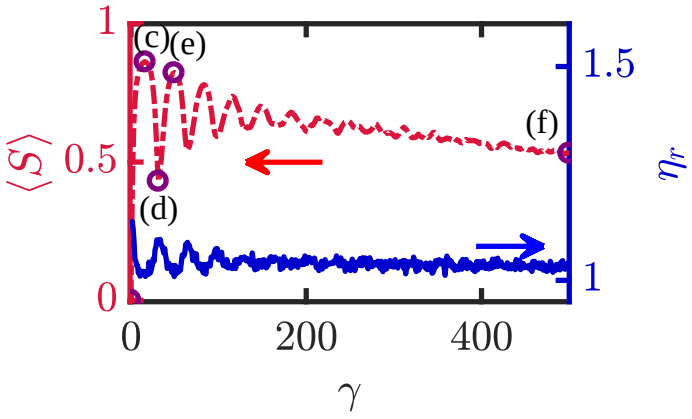
(a) (b)

Figure 3.7: Relative viscosity, η_r, and order parameter, $\langle S \rangle$, as a function of volume fraction, ϕ, for various Pe at aspect ratio $r_p = 14$. Error bars indicate the 95% confidence interval.

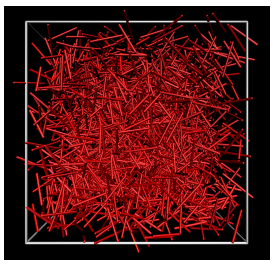
3.3.4 *Transient Properties of Suspensions that are Isotropic at Rest*

The transient rheological properties and structure of the suspensions depicted in Fig. 3.6b ($r_p = 14$, $\phi = 0.044$, $10^{-4} \leq$ Pe $\leq 10^5$) were also investigated. At low Pe (Pe $< 10^2$), η_r and $\langle S \rangle$ rapidly reach steady state with random fluctuations. At intermediate Pe ($10^2 \leq$ Pe $< 10^4$), η_r and $\langle S \rangle$ often exhibit weak overshoots followed by a rapid approach to steady state with fluctuations. At high Pe (Pe $\geq 10^4$), both η_r and $\langle S \rangle$ exhibit damped periodic oscillations. This is illustrated in Fig. 3.8, where η_r and $\langle S \rangle$, blocked averaged with $\Delta_B \gamma = 1$, at Pe $= 10^5$ are plotted as a function of γ along with simulation snapshots. Both η_r and $\langle S \rangle$ exhibit periodic oscillations that decrease in amplitude with increasing strain. The oscillations of η_r and $\langle S \rangle$ are anti-correlated. The oscillations are connected to the changes in the microstructure. The strains for each snapshot are indicated in Fig. 3.8. The suspension starts with random fiber orientations (Fig. 3.8b). The fibers align with

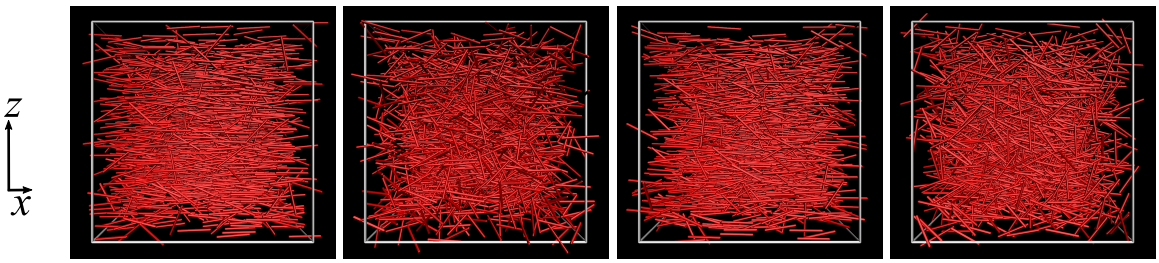
the flow direction (Fig. 3.8c). The suspension again becomes disordered (Fig. 3.8d). The process repeats until the suspension reaches an intermediate structure (Fig. 3.8f).



(a)



(b) Initial configuration



(c) $\gamma = 16.5$ (d) $\gamma = 31.5$ (e) $\gamma = 49.5$ (f) $\gamma = 500$

Figure 3.8: (a) Transient order parameter, $\langle S \rangle$, and relative viscosity, η_r, as a function of the strain defined as $\gamma = \dot{\gamma}t$. The transient data are blocked averaged with $\Delta_B\gamma = 1$. (b)-(f) Simulation snapshots at various γ. The ambient flow direction is $\boldsymbol{U}^\infty = \dot{\gamma}z\boldsymbol{e}_x$. The simulation parameters are aspect ratio $r_p = 14$, volume fraction $\phi = 0.044$, and Pe $= 10^5$.

Ross and Klingenberg (1997) observed similar oscillations in the viscosity in simulations of suspensions of non-Brownian, linked, rigid spherocylinders with $\phi = 0.025$ and $r_p = 25$. Ivanov et al. (1982) performed experiments with suspensions of nylon fibers ($D \leq 0.1$ mm) in Castor Oil ($\eta_0 = 4.9$ Pa·s). Damped oscillations were observed for $4.0 \leq r_e \leq 8.6$, $1.61 \leq \dot{\gamma} \leq 4.01$, $0.0025 \leq \phi \leq 0.023$, and Pe$= O(10^{12})$. Oscillations have not been reported in the rheology of cellulose nanocrystal suspensions, presumably because the nano-scale dimensions result in small values of Pe for experimentally-accessible shear rates.

At large Pe, hydrodynamic forces dominate over thermal forces in determining the

fiber dynamics and suspension structure. Fibers undergo essentially Jeffery orbits, altered somewhat by fiber collisions. The dimensionless period of oscillation in Fig. 3.8a is estimated to be 40, which is slightly larger than half of 62, the Jeffery orbit period calculated using $T\dot{\gamma} = 2\pi\left(r_e + r_e^{-1}\right)$ (Jeffery, 1922; Goldsmith and Mason, 1962) with $r_e = 0.7r_p$ at $r_p = 14$. Fibers collide with other fibers while orbiting. A longer period may result from the confining effect of neighboring fibers.

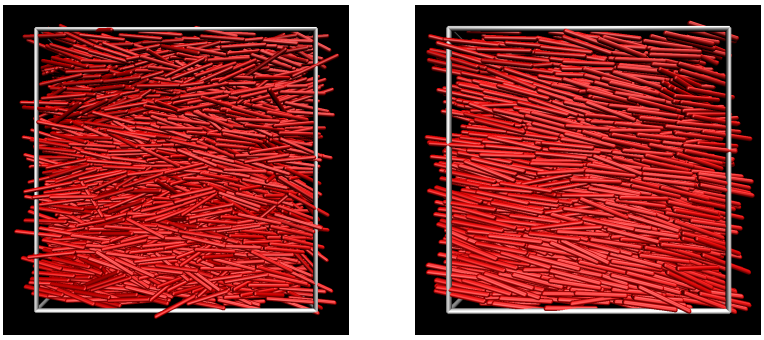
3.3.5 *Transient Properties of Suspensions that are Nematic at Rest*

Shafiei-Sabet et al. (2012) noted that liquid crystalline suspensions of cellulose nanocrsytals and lyotropic liquid crystalline polymers exhibit similar rheology, in which the liquid crystalline structures change with shear rate. The rheological properties of suspensions that are nematic at rest differ from those of suspensions that are isotropic at rest (Bercea and Navard, 2000). To investigate the effect of Pe on the rheological properties and structure of suspensions that are nematic at rest, suspensions with $r_p = 10$ and $\phi = 0.30$ and 0.46 are first examined at rest then subjected to shear.

Fibers were started on lattice with $N = 11,094$ and $9,680$ for $\phi = 0.30$ and 0.46 ($\phi^* = 0.34$ and 0.51), respectively. Simulation boxes were cubic with side lengths at least $5L$. Simulations without shear were run with $\Delta t^* = 10^{-6}$ to $t^* = 100$. The phases of the suspensions were identified to be nematic based on the methods and results described in Secs. 3.2.4 and 3.3.2.

Snapshots at $t^* = 100$ for $r_p = 10$ and $\phi = 0.30$ and 0.46 are shown in Fig. 3.9. The apparently dark voids in the snapshots are not voids, but are filled by groups of fibers sticking out of the opposite end of the simulation box. In Fig. 3.9a, the suspension with $\phi = 0.30$ is near the isotropic-nematic phase transition and exhibits characteristics of the nematic phase, *i.e.*, orientational order without translational order. In Fig. 3.9b, the suspension with $\phi = 0.46$ is near the nematic-smectic phase transition and exhibits some layering. While suspensions of both $\phi = 0.30$ and 0.46 are characterized to be nematic at

rest, the latter could be in coexistence with the smectic phase.



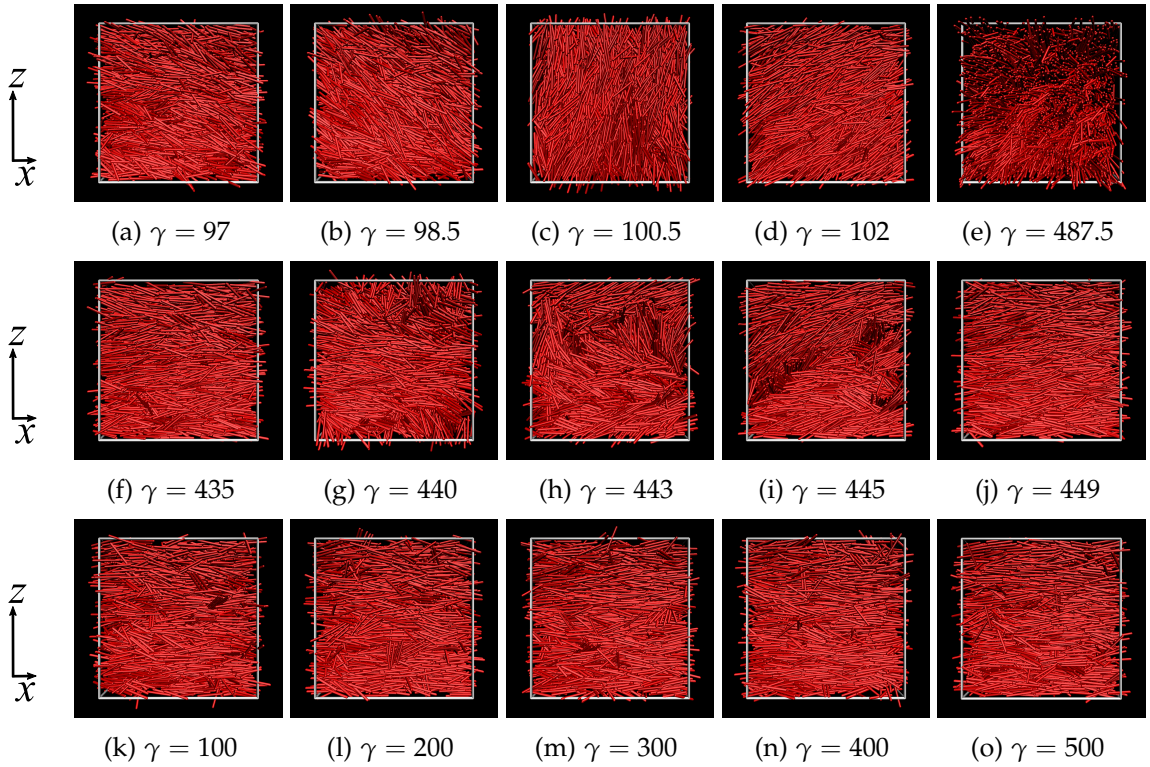
(a) $\phi = 0.30$ (b) $\phi = 0.46$

Figure 3.9: Snapshots of suspensions that are nematic at rest with $r_p = 10$ and (a) $\phi = 0.30$ ($\phi^* = 0.34$), near the isotropic-nematic phase transition, (b) $\phi = 0.46$ ($\phi^* = 0.51$), near nematic-smectic phase transition.

Suspensions with $r_p = 10$ and $\phi = 0.30$ and 0.46 were sheared at Pe $= 0.1$, 1, and 10. Figures 3.10 and 3.11 present simulation snapshots at various strains, γ. Figure 3.12 presents the transient order parameter, $\langle S \rangle$, relative viscosity, η_r, and the first and second normal stress differences, $\langle N_1 \rangle$ and $\langle N_2 \rangle$. The relationships between liquid crystalline structures and the rheological properties are described below.

Nematic suspension near the isotropic-nematic transition

The suspension with $\phi = 0.30$, which is near the isotropic-nematic phase transition, was sheared at Pe $= 0.1$, 1, and 10. At Pe $= 0.1$, all the fibers in the suspension rotate a few times in the flow-gradient plane (Figs. 3.10a-d) before aligning in the vorticity direction (Fig. 3.10e) and begin kayaking. The motions occur largely in concert, *i.e.*, the motions and orientations of all the fibers are nearly identical. In Fig. 3.12a, $\langle S \rangle \approx 0.75$ throughout the simulation; this verifies that motions occur in concert. This high degree of order suggests that the entire system is a single domain. Rotations in the gradient direction cause oscillations in η_r with a period of 15, which is less than half of the Jeffery orbit period for $r_p = 10$ ($T\dot{\gamma} = 45$). Oscillations in $\langle N_1 \rangle$ and $\langle N_2 \rangle$ are also observed. As the domain aligns in the vorticity direction, η_r and $\langle S \rangle$ oscillate about constant values while



(a) $\gamma = 97$ (b) $\gamma = 98.5$ (c) $\gamma = 100.5$ (d) $\gamma = 102$ (e) $\gamma = 487.5$

(f) $\gamma = 435$ (g) $\gamma = 440$ (h) $\gamma = 443$ (i) $\gamma = 445$ (j) $\gamma = 449$

(k) $\gamma = 100$ (l) $\gamma = 200$ (m) $\gamma = 300$ (n) $\gamma = 400$ (o) $\gamma = 500$

Figure 3.10: Simulation snapshots for suspensions with $r_p = 10$ and $\phi = 0.3$ at various strains, γ, (a)-(e) Pe $= 0.1$, (f)-(j) Pe $= 1$, (k)-(o) Pe $= 10$. The suspension is nematic at rest.

the magnitudes of oscillations decrease. The alignment causes $\langle N_1 \rangle$ to decrease to 0 and $\langle N_2 \rangle$ to decrease to a negative plateau.

As Pe is increased to 1, the single domain observed at Pe $= 0.1$ breaks into smaller domains. In Figs. 3.10f-j, simulation snapshots show the rotation of one domain, which causes the rotation of nearby domains. Eventually, all the fibers in the suspension have rotated. The domains re-align with the flow direction at different γ. In Fig. 3.12b, $\langle S \rangle$ oscillates with a larger amplitude and the same periodicity as at Pe $= 0.1$, while the fibers rotate in the flow-gradient plane ($\gamma \leq 120$). Oscillations are also observed for η_r, $\langle N_1 \rangle$, and $\langle N_2 \rangle$. As $\langle S \rangle$ increases, η_r and $\langle N_1 \rangle$ increase, and $\langle N_2 \rangle$ decreases.

As Pe is increased to 10, the domains observed at Pe $= 1$ break up into even smaller clusters of fibers. Coherent rotation of fibers distinguishes a domain from fiber cluster. In Figs. 3.10k-o, simulation snapshots show that the fibers mostly align with the flow except

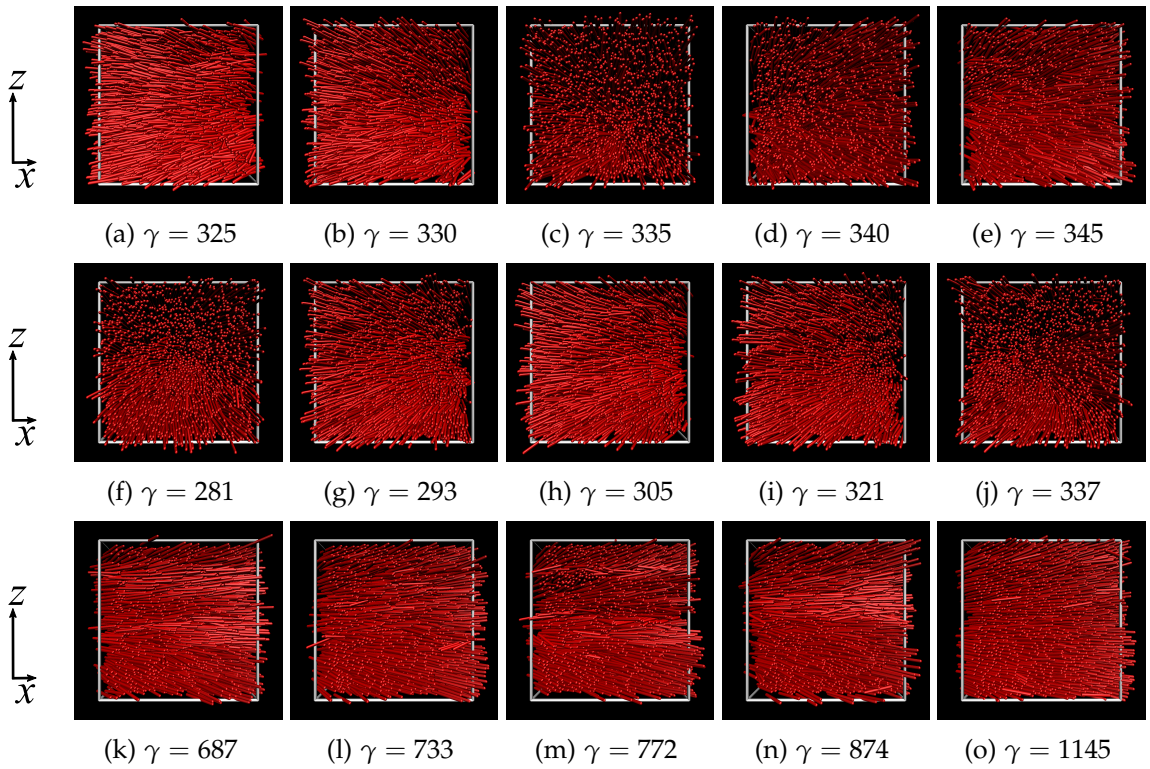
for small clusters of fibers containing at most a dozen of fibers. This structure seems to correspond to the continuous monodomain that Onogi and Asada Onogi and Asada (1980) described. In Fig. 3.12c, $\langle S \rangle \approx 0.8$ throughout the simulation. While η_r and $\langle N_1 \rangle$ fluctuate around a positive value, $\langle N_2 \rangle$ fluctuates around 0.

For the suspension near the isotropic-nematic phase transition, a system-wide domain at Pe $= 0.1$, propagating domains at Pe $= 1$, and small clusters at Pe $= 10$ are observed. The dynamics of the domains and formation of a continuous phase with small clusters seem to explain the interplay between structure and the rheological properties.

Nematic suspension near the nematic-smectic phase transition

The suspension with $\phi = 0.46$, which is near the nematic-smectic phase transition, was also sheared at Pe $= 0.1$, 1, and 10. In Figs. 3.11a-j, simulation snapshots show that the fibers align in the vorticity direction and kayak in concert for both Pe $= 0.1$ and 1. In Figs. 3.12a and 3.12b, $\langle S \rangle \approx 0.95$ for both Pe. As ϕ is increased from 0.3 to 0.46 for Pe $= 0.1$, a higher order is maintained and the system-wide domain is more coherent. Unlike $\phi = 0.3$ at Pe $= 0.1$, rotation in the gradient direction is not observed. The transient rheology is similar; η_r fluctuates around a positive value; $\langle N_1 \rangle$ reaches a plateau near 0 and $\langle N_2 \rangle$ reaches a negative-valued plateau. The oscillations in $\langle N_1 \rangle$ and $\langle N_2 \rangle$ that last for a strain of approximately 60 result from kayaking. The minimum values of $\langle N_1 \rangle$ and $\langle N_2 \rangle$ correspond to alignment in the vorticity direction. Deviation of the fiber orientation from the vorticity direction as the fibers kayak increases both $\langle N_1 \rangle$ and $\langle N_2 \rangle$.

As Pe is increased to 10, layered domains with different orientations coexist. In Figs. 3.11k-o, simulation snapshots show that the structure starts as one coherent domain and evolves to four layers with different orientations. In Fig. 3.12c, $\langle S \rangle$ shows large oscillations with periodicity of approximately 280. The relative viscosity fluctuates around a positive value. The partial alignment of fibers in the vorticity direction causes $\langle N_1 \rangle$ and $\langle N_2 \rangle$ to decrease. For $\phi = 0.3$ at Pe $= 10$, a continuous monodomain is observed, while for $\phi = 0.46$, several domains are observed. This qualitatively matches the observation of



(a) $\gamma = 325$ (b) $\gamma = 330$ (c) $\gamma = 335$ (d) $\gamma = 340$ (e) $\gamma = 345$

(f) $\gamma = 281$ (g) $\gamma = 293$ (h) $\gamma = 305$ (i) $\gamma = 321$ (j) $\gamma = 337$

(k) $\gamma = 687$ (l) $\gamma = 733$ (m) $\gamma = 772$ (n) $\gamma = 874$ (o) $\gamma = 1145$

Figure 3.11: Simulation snapshots for suspensions with $r_p = 10$ and $\phi = 0.46$ at various strains, γ, (a)-(e) Pe $= 0.1$, (f)-(j) Pe $= 1$, (k)-(o) Pe $= 10$. The suspension is nematic at rest.

Onogi and Asada (1980) that a higher shear rate is required to bring a suspension with a higher concentration to a continous phase.

For the suspension near the nematic-smectic phase transition, system-wide domains at Pe $= 0.1$ and 1, and layered domains with varying orientations at Pe $= 10$ are observed. Breakdown of domains into small clusters is not observed for the range of Pe investigated. As ϕ is increased from 0.3 to 0.46, the domains are more coherent and complete. Similar to $\phi = 0.3$, the dynamics of the domains affect the rheological properties.

3.4 Conclusions

The structure and rheological properties of suspensions of rigid spherocylinders with only repulsive interactions were investigated via Brownian dynamics simulations. The sphe-

rocylinder is a model for a cellulose nanocrystal (CNC) fiber. The diffusivities and liquid crystalline phase diagram qualitatively matched results from simulations of suspensions of hard spherocylinders. The viscosity and structure qualitatively matched experimental values for CNC suspensions.

The spherocylinders interact with a repulsive force that decays exponentially with the surface separation. A decay parameter, a, governs the stiffness of the interaction. With a set to $20/b$, where b is the radius, the translational and rotational diffusivities match reported values of suspensions of hard spherocylinders for aspect ratio $r_p = 5$ and volume fraction $\phi \leq 0.4$.

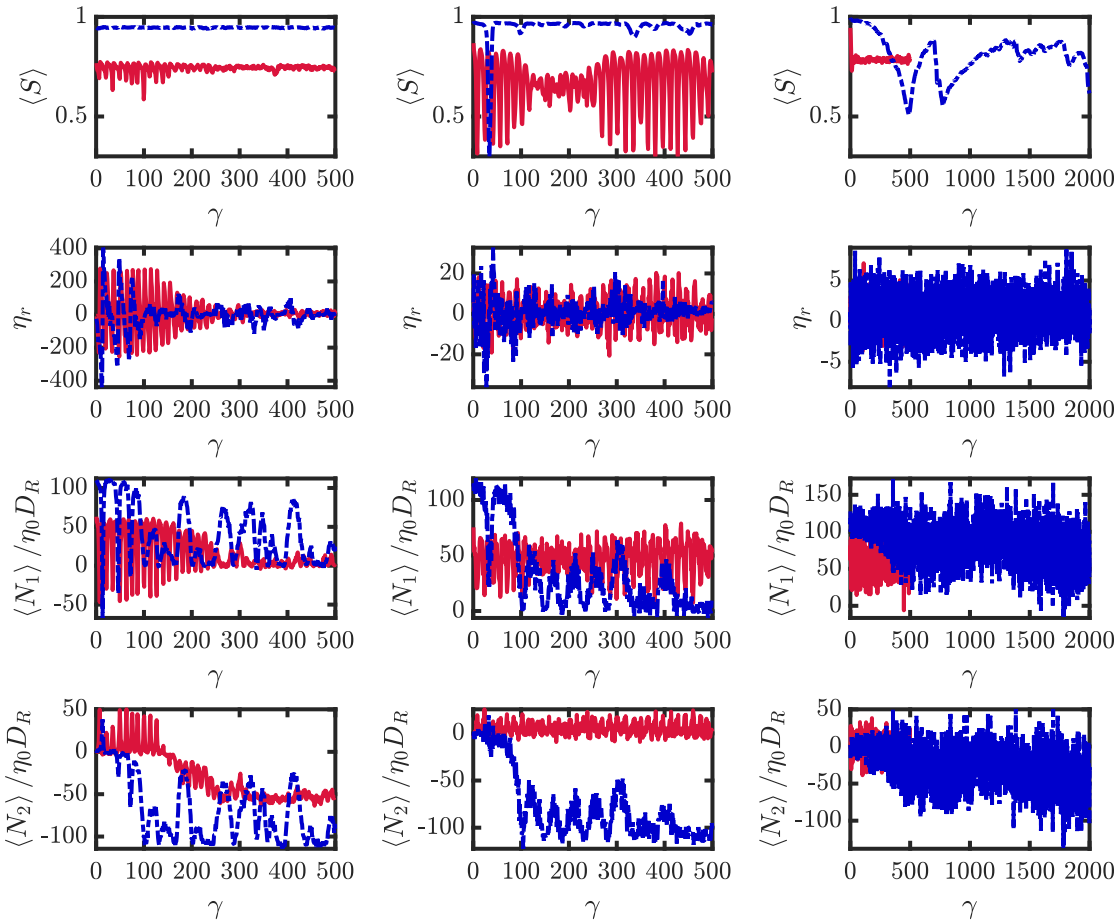
Isotropic, nematic, smectic, and solid phases were identified using the orientational and hexatic order parameters, and pair distribution functions. The phase diagram with $2 \leq r_p \leq 30$ and $0.001 \leq \phi \leq 0.7$ qualitatively agrees with that for hard spherocylinders reported by Bolhuis and Frenkel (1997). Differences between in the phase diagrams was more evident for $r_p \geq 15$ and $\phi \geq 0.45$.

The relative viscosity and order parameter of suspensions with $(r_p, \phi) = (50, 0.031)$ and $(14, 0.044)$ and Péclet number in the range $10^{-4} \leq \mathrm{Pe} \leq 10^5$ were compared to experimental results reported for cellulose nanocrystal suspensions by Orts et al. (1998) and Shafiei-Sabet et al. (2012). The simulation results exhibit the typical three-region flow curve of lyotropic liquid crystalline suspensions, with two shear thinning regions bracketing a viscosity plateau at intermediate shear rates. Large viscosity values, similar in magnitude to those measured experimentally, were obtained despite employing only repulsive interactions. The increase of the order parameter with increasing Pe qualitatively matched experimental values.

For $\mathrm{Pe} \geq 10^4$, isotropic suspensions with $r_p = 14$ and $\phi = 0.044$ exhibit oscillations in the viscosity and order parameter whose amplitudes decrease with strain and reach steady values. This behavior is consistent with previously reported experiments (Ivanov et al., 1982) and simulations of flexible fibers (Ross and Klingenberg, 1997).

Suspensions that are nematic at rest exhibited a variety of changes in the transient

structure and rheological properties with increasing Pe, which depended on the spherocylinder concentration ($r_p = 10$). The results are consistent to those reported by Onogi and Asada (1980), who described the relationship between structure and rheology in terms of domain dynamics.



(a) Pe = 0.1 (b) Pe = 1 (c) Pe = 10

Figure 3.12: Transient rheology for suspensions with $\phi = 0.3$ (solid red curve) and 0.51 (dotted blue curve) sheared with Pe = 0.1, 1, and 10. The order parameter, $\langle S \rangle$, relative viscosity, η_r, and the first and second normal stress differences, $\langle N_1 \rangle / \eta_0 D_R$, and $\langle N_2 \rangle / \eta_0 D_R$ are plotted as a function of the strain, γ. The corresponding simulation snapshots are shown in Figs. 3.10 and 3.11.

4

EFFECT OF DRYING AND REHYDRATION ON THE RHEOLOGICAL PROPERTIES AND STRUCTURE OF SUSPENSIONS OF LINKED SPHEROCYLINDERS

4.1 Introduction

Flexible fibers, such as nylon (Soszynski and Kerekes, 1988a), polyethylene terephthalate (Chen et al., 2002), carbon (Jiang et al., 2016), wood (Mason, 1950), and microfibrillated cellulose (Karppinen et al., 2012) flocculate in suspensions above critical concentrations (Björkman, 2003). Flocculation impacts the rheological properties, and is crucial to the end-product quality (Zhang et al., 2012; Chen et al., 2002).

Microfibrillated cellulose (MFC), or nanofibrillated cellulose (NFC), has gained attention as a renewable material to replace or supplement petroleum-based materials (Moon et al., 2011; Klemm et al., 2018). Mechanical and chemical treatments of cellulose fibers (Turbak et al., 1983; Moon et al., 2011; Lavoine et al., 2012; Zhu et al., 2018; Klemm et al., 2018) are employed to obtain MFC fibers, which are 2-50 nm in diameter and several micrometers in length (Lavoine et al., 2012). The fibers interact through electrostatic repulsion (Pääkkö et al., 2007; Fall et al., 2011), and attraction that arises from hydrogen bonding and van der Waals forces (Notley et al., 2004). Applications include composite

reinforcement (Zimmermann et al., 2010; Xu et al., 2013; Ghanadpour et al., 2015), rheological modification (Sun et al., 2016), transparent films (Nogi et al., 2009), dewatering (Sim et al., 2015), oxygen barriers (Syverud and Stenius, 2009), and emulsion stabilization (Andresen and Stenius, 2007).

Suspensions of MFC fibers are shear thinning (Karppinen et al., 2012), typical of flexible fiber suspensions (Bibbo et al., 1985; Ganani and Powell, 1985), where the viscosity depends on the concentration (Petrich et al., 2000), adhesive forces between fibers (Chaouche and Koch, 2001), elastic modulus of the fibers (Bennington et al., 1990), and the aspect ratio (Mongruel and Cloitre, 1999). The shear thinning behavior is more pronounced as the concentration is increased (Lasseuguette et al., 2008). Three-region flow curves, where two shear thinning region bracket a viscosity plateau at intermediate shear rates, are observed. The viscosity plateau is attributed to the breaking down of fiber networks (Agoda-Tandjawa et al., 2010; Iotti et al., 2011).

Transport and storage of MFC fiber suspensions at high concentrations are crucial to economic commercialization of MFC applications. However, the drying and rehydration process irreversibly modifies the rheological properties (Saito et al., 2006). The reductions in viscosity, storage modulus, and yield stress are attributed to hydrogen bonding and the van der Waals forces in permanent aggregates (Iwamoto et al., 2008; Missoum et al., 2012). Suspensions can be redispersed through sonication (Chen et al., 2013), surface modification (Andresen et al., 2006; Lasseuguette et al., 2008), and addition of polymers and/or electrolytes (Lowys et al., 2001; Missoum et al., 2012), with increased cost of processing.

Fiber-level simulations have been employed to investigate the relationship between structure and rheological properties of flexible fiber suspenisons. Ross and Klingenberg (1997) modeled fibers as linked-spherocylinders that interact through repulsive forces. Oscillations in the viscosity with decaying magnitude with the shear strain were observed, similar to those reported experimentally by Ivanov et al. (1982). In simulations, fiber suspensions flocculated with only friction forces, in the absence of attractive forces

(Schmid et al., 2000). The cohesiveness of flocs, quantified by the stored elastic energy, was lower for flocs formed from only attractive forces than for those formed from only friction forces (Schmid and Klingenberg, 2000b). The viscosities of curved fiber suspensions were larger than those for straight fibers (Switzer and Klingenberg, 2003). The shear thinning behavior of fiber suspensions was associated with flocculation, and a competition of hydrodynamic forces and fiber elasticity.

This chapter focuses on the effect of the drying and rehydration process on the rheological properties and structure of suspensions of flexible fibers, as a model of MFC fiber suspensions. Fiber-level simulations are employed to determine the changes in the rheological properties and structure of the dried and rehydrated suspensions. Fibers are modeled as linked-rigid spherocylinders that interact via friction, repulsion, and short-ranged attraction, as a simple model for hydrogen bonding and/or van der Waals forces. When fibers interact via sufficient attraction and friction, the viscosities of the suspensions that undergo a drying and rehydration process are smaller than those of never-dried suspensions. Cluster statistics reveal that the difference in viscosity is associated with the formation of dense and persistent flocs, as observed experimentally (Missoum et al., 2012).

4.2 Methods

4.2.1 *Fiber Model*

As illustrated in Fig. 4.1, a fiber is modeled as linked-spherocylinders with ball and socket joints, which allow the fiber to bend and twist with angular potentials. The model employed here was also used by Switzer and Klingenberg (2003). Fibers have length L, diameter D, and aspect ratio r_p defined as L/D. Each fiber consists of N_{seg} spherocylinders or segments, with half-length l, and radius b. The segment aspect ratio r_{ps} is l/b, equivalent to r_p/N_{seg}.

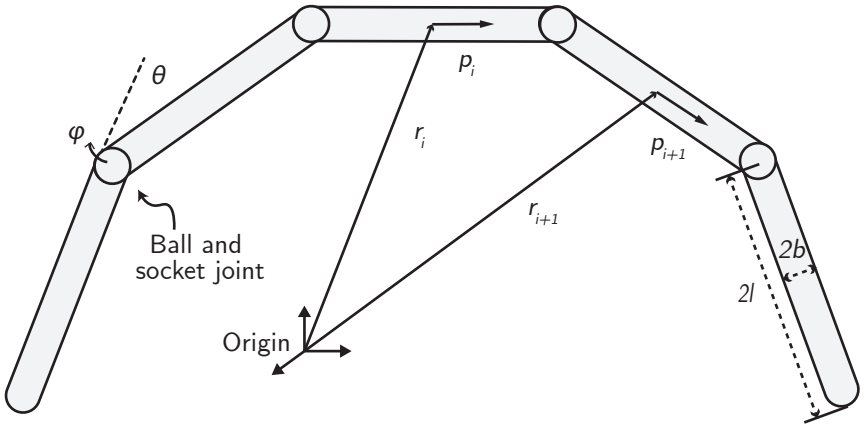


Figure 4.1: Schematic diagram for a fiber modeled as linked-spherocylinders. Bending and twisting angles, θ and ϕ, between segments are 0.6 and 0.0 in this diagram.

Neglecting inertia, the translational and rotational equations of motion for fiber segment i are

$$\boldsymbol{F}_i^{\text{hyd}} + \sum_j^{N_{C_i}} \boldsymbol{F}_{ij}^{\text{con}} + \boldsymbol{X}_{i+1} - \boldsymbol{X}_i = 0 \tag{4.1}$$

$$\boldsymbol{T}_i^{\text{hyd}} + \sum_j^{N_{C_i}} \boldsymbol{G}_{ij} \times \boldsymbol{F}_{ij}^{\text{con}} + l\boldsymbol{p}_i \times (\boldsymbol{X}_{i+1} - \boldsymbol{X}) + \boldsymbol{Y}_{i+1} - \boldsymbol{Y}_i = 0 \tag{4.2}$$

The forces and torques in these equations are described below.

For an isolated segment with linear and angular velocity $\dot{\boldsymbol{r}}_i$ and $\boldsymbol{\omega}_i$, the hydrodynamic force and torque are $\boldsymbol{F}_i^{\text{hyd}} = \boldsymbol{A}_i \cdot (\boldsymbol{U}_i^\infty - \dot{\boldsymbol{r}}_i)$ and $\boldsymbol{T}_i^{\text{hyd}} = \boldsymbol{C}_i \cdot (\boldsymbol{\Omega}_i^\infty - \boldsymbol{\omega}_i) + \tilde{\boldsymbol{H}}_i : \boldsymbol{E}^\infty$, where $\boldsymbol{A}_i$, $\boldsymbol{C}_i$, and $\tilde{\boldsymbol{H}}_i$ are the resistance tensors for an isolated prolate spheroid with equivalent aspect ratio, $r_{es} = 0.7 r_{ps}$ (Kim and Karilla, 1991). For simple shear flow, the ambient translational and angular velocity $\boldsymbol{U}_i^\infty$ and $\boldsymbol{\Omega}_i^\infty$ are $(\dot{\gamma}z, 0, 0)^{\text{T}}$ and $(0, \dot{\gamma}/2, 0)^{\text{T}}$, where $\dot{\gamma}$ is the shear rate. The rate of strain tensor $\boldsymbol{E}^\infty$ is $0.5\left[(\nabla \boldsymbol{U}_i^\infty) + (\nabla \boldsymbol{U}_i^\infty)^{\text{T}}\right]$.

Hydrodynamic interactions between segments are neglected. Sundararajakumar and Koch (1997) found that mechanical contacts dominate over hydrodynamic interactions in controlling the fiber orientation distributions when $nL^3/r_p \geq O(1)$, where n is the number density. Hydrodynamic interactions are not included in the model because most

simulations in this study fall in the semi-dilute regime, $10^{-1} \leq nL^3/r_p \leq 1$.

Fibers are kept inextensible by applying a constraint force $\boldsymbol{X}_i$ at joint i. The associated torque is $l\boldsymbol{p}_i \times (\boldsymbol{X}_{i+1} - \boldsymbol{X}_i)$, where $\boldsymbol{p}_i$ is the orientation vector of segment i. A fiber with bending and twisting angles, θ_i and ϕ_i, is restored to its equilibrium shape by applying a torque $\boldsymbol{Y}_i$ at joint i.

$$\boldsymbol{Y}_i = \kappa_b\left(\theta_i^{\text{eq}} - \theta_i\right)\boldsymbol{e}_i^b + \kappa_t\left(\phi_i^{\text{eq}} - \phi_i\right)\boldsymbol{e}_i^t \tag{4.3}$$

where $\boldsymbol{e}_i^b$ and $\boldsymbol{e}_i^t$ are the directions of bending and twisting in Eqs. 2.55 and 2.58. At joint 1 of each fiber, $\boldsymbol{X}_i = 0$ and $\boldsymbol{Y}_i = 0$. For small deformation of a linearly elastic cylinder, the bending constant κ_b is $E_Y I_{\text{mom}}/2l$, where E_Y is the Young's modulus, and $I_{\text{mom}} = \pi b^4/4$ is the area moment. The twisting constant κ_t is $0.67\kappa_b$. The effective stiffness S_{eff} of the fiber is $E_Y I_{\text{mom}}/\eta_0\dot{\gamma}L^4$ (dimensionless), where η_0 is the viscosity of the suspending fluid.

A segment i interacts with N_{C_i} other segments. The contact force $\boldsymbol{F}_{ij}^{\text{con}}$ on segment i due to segment j is the sum of normal and friction forces, $\boldsymbol{F}_{ij}^N$ and $\boldsymbol{F}_{ij}^{\text{fric}}$. The associated torque on segment i is $\boldsymbol{G}_{ij} \times \boldsymbol{F}_{ij}^{\text{con}}$, where $\boldsymbol{G}_{ij}$ is the vector from the center of segment i to the point of contact with segment j. The normal force as a function of the minimum separation between segment surfaces, h_{ij}, is

$$\boldsymbol{F}_{ij}^N = \begin{cases} \left[-F^{\text{rep}}\exp\left(-a^{\text{rep}}h_{ij}\right) + F^{\text{att}}\exp\left(-a^{\text{att}}h_{ij}^2\right)\right]\boldsymbol{n}_{ij} & h_{ij} > 0 \\ \left[-F^{\text{rep}}\exp\left(-a^{\text{rep}}h_{ij}\right) + F^{\text{att}}\right]\boldsymbol{n}_{ij} & h_{ij} \leq 0 \end{cases} \tag{4.4}$$

where F^{rep} and F^{att} are the repulsive and attractive force coefficients, a^{rep} and a^{att} are the associated decay parameters, and $\boldsymbol{n}_{ij}$ is the direction normal to both segment surfaces.

In the plane of contact, fibers are constrained to no relative motion,

$$\begin{bmatrix} \Delta\boldsymbol{u}_{ij} \cdot \boldsymbol{e}_1^{\text{loc}} \\ \Delta\boldsymbol{u}_{ij} \cdot \boldsymbol{e}_2^{\text{loc}} \\ \boldsymbol{F}_{ij}^{\text{fric}} \cdot \boldsymbol{n}_{ij} \end{bmatrix} = \boldsymbol{0} \tag{4.5}$$

where $\Delta \boldsymbol{u}_{ij}$ is the relative velocity at the point of contact, and $\boldsymbol{e}_1^{\text{loc}}$ and $\boldsymbol{e}_2^{\text{loc}}$ define the plane of contact. Equation 4.5 is solved for $\boldsymbol{F}_{ij}^{\text{fric}}$. For contacts with $\left\|\boldsymbol{F}_{ij}^{\text{fric}}\right\| > \mu \left\|\boldsymbol{F}_{ij}^{N}\right\|$, $\boldsymbol{F}_{ij}^{\text{fric}}$ is replaced by $\mathbf{0}$, where μ is the static friction coefficient. Equation 4.5 is resolved if there are more than two pairs of contacts.

4.2.2 *Drying and Rehydration Processes*

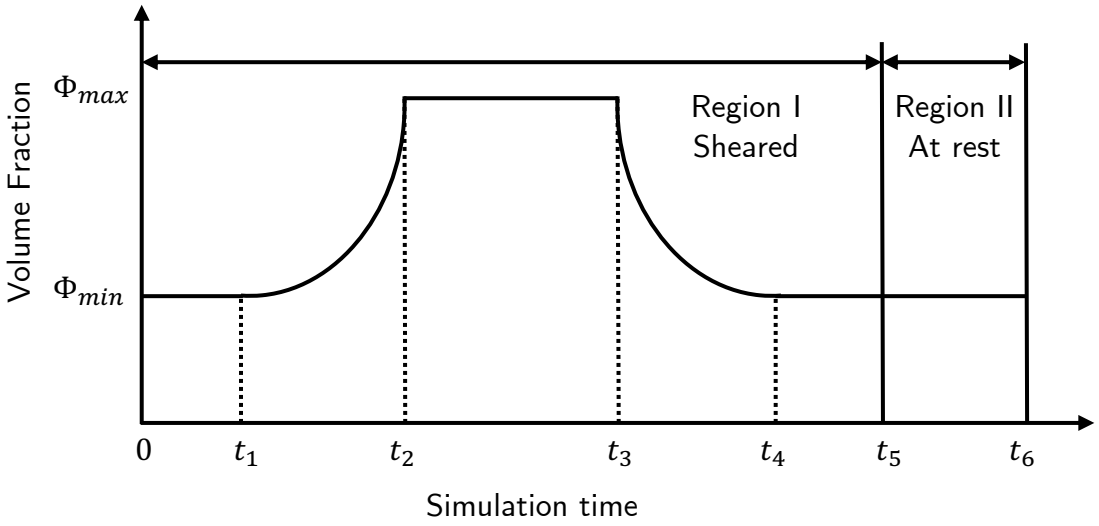


Figure 4.2: Schematic diagram of the drying and rehydration process, where Φ is the volume fraction, and t is the simulation time.

Suspensions are simulated with a fixed number of fibers, typically $1,280$. The volume fraction Φ is varied for suspensions that undergo the drying and rehydration process by changing the size of the cubic simulation box, with side length L_{box}. To avoid overlaps of fibers and let the suspensions reach steady intermediate structures, L_{box} is only changed every dimensionless time interval of 0.1. When L_{box} is decreased during drying, the segment centers of mass are moved affinely with the box after the equations of motion are integrated. During rehydration, fibers are moved by integrating the equations of motion only.

The volume fraction of the suspension as a function of time, t, through the drying and rehydration process is illustrated in Fig. 4.2. The drying and rehydration process is

divided into two regions: region I of $0 \leq t \leq t_5$, and region II of $t_5 \leq t \leq t_6$. Suspensions are sheared in region I and at rest in region II. From $t = 0$ to t_1, the suspension remains at $\Phi = \Phi_{min}$. From $t = t_1$ to t_2, the suspension is dried with rate characterized by the parameter α (see below) to $\Phi = \Phi_{max}$, at which the suspension remains from $t = t_2$ to t_3.

From $t = t_3$ to t_4, the suspension is rehydrated with rate characterized by the parameter β back to $\Phi = \Phi_{min}$, at which the suspension remains from $t = t_4$ to t_6. Throughout the drying and rehydration process, the simulation box side is varied while the number of fibers N_{fib} is kept constant. The volume fraction as a function of t is

$$\Phi = \begin{cases} \Phi_{min} & 0 \leq t < t_1, t_4 \leq t \leq t_6 \\ \Phi_{min}\alpha^{-10(t-t_1)} & t_1 \leq t < t_2 \\ \Phi_{max} & t_2 \leq t < t_3 \\ \Phi_{max}\beta^{10(t-t_3)} & t_3 \leq t < t_4 \end{cases} \tag{4.6}$$

The drying and rehydration ratio, r_Φ is Φ_{max}/Φ_{min}. Suspensions that are never-dried remain at Φ_{min} during simulations.

4.2.3 *Simulation Method*

In this chapter, x, y, z are the flow, vorticity, and gradient directions, respectively. Fibers are added to the simulation box by random sequential addition. The center of mass position of a fiber and the orientation for the first segment of the fiber are randomly initialized. The orientations of the segments are then generated from the specified equilibrium shape. The simulation box side lengths are at least $2.3L$, even when $\Phi = \Phi_{max}$ during the drying and rehydration process. Periodic Lees-Edwards sliding boundaries (Lees and Edwards, 1972) are employed for simulations under simple shear.

A second-order Adams-Bashforth algorithm on a GPU is employed to integrate the

equations of motion

$$\boldsymbol{r}_i(t+\Delta t) = \boldsymbol{r}_i(t) + [1.5\dot{\boldsymbol{r}}_i(t) - 0.5\dot{\boldsymbol{r}}_i(t-\Delta t)]\,\Delta t \tag{4.7}$$

$$\boldsymbol{q}_i(t+\Delta t) = \boldsymbol{q}_i(t) + [1.5\dot{\boldsymbol{q}}_i(t) - 0.5\dot{\boldsymbol{q}}_i(t-\Delta t)]\,\Delta t \tag{4.8}$$

where $\boldsymbol{q}_i$ is the vector of Euler parameters of fiber segment i, which is directly related to the orientation $\boldsymbol{p}_i$ (Wittenburg, 1977; Haug, 1992; Ross and Klingenberg, 1997).

To nondimensionalize the variables and the equations of motion for simulations under shear, the time scale is set to $\dot{\gamma}^{-1}$, the length scale to b, the force scale to $6\pi\eta_0 bl\dot{\gamma}$, and the torque scale to $8\pi\eta_0 l^3\dot{\gamma}$. Under shear, the dimensionsionless time, t^*, is equivalent to the shear strain. For simulations at rest, the time scale is set to D_R^{-1}, where D_R, the rotational diffusivity of an isolated fiber, is $k_\mathrm{B}T/8\pi\eta_0 b^3 r_p^3 Y_C$, and Y_C is the scalar resistance function that relates hydrodynamic torque and angular velocity (Kim and Karilla, 1991).

The microfibrillated cellulose fibers that Iwamoto et al. (2009) measured were 20.3 nm wide and 8.4 nm thick, with lengths up to 10 μm. Using $L = 10$ μm, $D = 20$ nm, $\eta_0 = 0.001$ Pa·s, $E_Y = 145$ GPa, and $\dot{\gamma} = 1$ s^{-1}, $S_\mathrm{eff} = 1.8$. Fibers with such large values of S_eff require simulations with prohibitively small time steps. Switzer and Klingenberg (2004) showed that the flocculation behavior of model structures of stiff fibers can be replicated in less stiff fibers by employing a larger coefficient of friction. For simulation results reported in this chapter, $N_\mathrm{fib} = 1,280$, $N_\mathrm{seg} = 5$, $\theta^\mathrm{eq} = 0.6$, $\phi^\mathrm{eq} = 0$, $r_p = 75$, $S_\mathrm{eff} = 0.05$, and $0 \leq \mu \leq 15$.

For fiber-fiber normal interactions, F^rep is fixed at $900\pi\eta_0 bl\dot{\gamma}$, and F^att is governed by $6\pi\eta_0 bl\dot{\gamma}A_N$, where A_N ranges from 0 to 50. The decay parameters a^rep and a^att are set to $20/b$ and $35/b$. The drying and rehydration process is implemented with α and β fixed at 0.9995, and $\Phi_{min} = 0.003$. The maximum concentration, Φ_{max}, is $4\Phi_{min}$.

Excluding the variables fixed above, the viscosity, η, is a function of variables from the

fiber model, and the drying and rehydration process

$$\eta = f\left(\mu, A_N, t_{1-6}\right) \tag{4.9}$$

4.2.4 *Characterization*

The total stress of the system $\langle\sigma\rangle$ is

$$\langle\boldsymbol{\sigma}\rangle = -p\boldsymbol{\delta} + \langle\boldsymbol{\sigma}_P\rangle + 2\eta_0\boldsymbol{E}^\infty \tag{4.10}$$

where p is the isotropic pressure, and $\boldsymbol{\sigma}_P$ is the particle contribution. Simplified using slender body theory (Batchelor, 1970b; Switzer and Klingenberg, 2003), $\boldsymbol{\sigma}_P$ is

$$\begin{aligned}\langle\boldsymbol{\sigma}_P\rangle =& \frac{4\pi n l^3 \eta_0}{3\ln\left(2r_p\right)}\left\langle \sum_{i=1}^{N_{\text{seg}}}\left\{\boldsymbol{E}^\infty\cdot\boldsymbol{p}_i\boldsymbol{p}_i + \boldsymbol{p}_i\boldsymbol{p}_i\cdot\boldsymbol{E}^\infty - \boldsymbol{p}_i\boldsymbol{p}_i\boldsymbol{p}_i\boldsymbol{p}_i : \boldsymbol{E}^\infty - \left(\boldsymbol{p}_i\dot{\boldsymbol{p}}_i + \dot{\boldsymbol{p}}_i\boldsymbol{p}_i\right)\right.\right. \\ &\left.\left. + \frac{3}{l^2}\left(\left[\boldsymbol{\delta} - \frac{1}{2}\boldsymbol{p}_i\boldsymbol{p}_i\right]\cdot\left(\boldsymbol{U}^\infty - \dot{\boldsymbol{r}}_i\right)\boldsymbol{r}_i + \boldsymbol{r}_i\left[\boldsymbol{\delta} - \frac{1}{2}\boldsymbol{p}_i\boldsymbol{p}_i\right]\cdot\left(\boldsymbol{U}^\infty - \dot{\boldsymbol{r}}_i\right)\right)\right\}\right\rangle + \Gamma\boldsymbol{\delta}\end{aligned}$$

where the angle brackets indicate averaging over fibers and configurations every dimensionless time interval of 0.5, and Γ contains isotropic terms of no interest, *i.e.*, does not contribute to the shear stress. The relative shear viscosity η_r is $\langle\sigma_{xz}\rangle / \eta_0\dot{\gamma}$.

To quantify the deformation of a fiber entangled with other fibers upon cessation of shear, the stored elastic energy E^{elas} for a fiber is

$$E^{\text{elas}} = \frac{1}{2}\sum_{i=2}^{N_{\text{seg}}}\left[\kappa_b\left(\theta_i - \theta_i^{\text{eq}}\right)^2 + \kappa_t\left(\phi_i - \phi_i^{\text{eq}}\right)^2\right] \tag{4.11}$$

Fibers trapped in a floc by interfiber forces will have $E^{\text{elas}} > 0$, whereas non-flocculated fibers will have $E^{\text{elas}} = 0$ at rest. The average of E^{elas} over fibers and configurations is $\left\langle E^{\text{elas}}\right\rangle$.

To quantify the heterogeneity of the suspension, the simulation box is divided into

N_{bin} cubic bins, with side lengths of r_{ps}. The local volume fraction within bin i is $\Phi_{\text{loc},i}$. The intensity of segregation, I, quantifies the variance of the local volume fraction from the average volume fraction Φ for one configuration

$$I = \frac{\sum_{i=1}^{N_{\text{bin}}} \left(\Phi_{\text{loc},i} - \Phi\right)^2}{N_{\text{bin}}\Phi(1-\Phi)} \tag{4.12}$$

The average of I over configurations is $\langle I \rangle$.

In addition to bulk heterogeneity, cluster statistics are obtained from the fiber center of mass, $\boldsymbol{r}_{\text{cm}}$. A cluster is defined as a group of contacting fibers, through any segment pairs. The 26 periodic images around the central simulation box are included when identifying the clusters through a labeling algorithm developed by Hoshen and Kopelman (1976). The cluster center of mass, $\boldsymbol{r}^c_{\text{cm}}$, is the average of $\boldsymbol{r}_{\text{cm}}$ over the S fibers in the cluster. The radius of gyration tensor $\boldsymbol{G}_y$ quantifies the distribution of fibers around r^c_{cm} for one cluster

$$\boldsymbol{G}_y = \frac{1}{S}\sum_{i}^{S} \left(\boldsymbol{r}_{\text{cm},i} - \boldsymbol{r}^c_{\text{cm}}\right)\left(\boldsymbol{r}_{\text{cm},i} - \boldsymbol{r}^c_{\text{cm}}\right)^T \tag{4.13}$$

The radius of gyration of a cluster, R_g, is $\sqrt{\lambda_1^2 + \lambda_2^2 + \lambda_3^2}$, where λ_1, λ_2, and λ_3 are the square roots of the eigenvalues of $\boldsymbol{G}_y$ in order of decreasing magnitude. The average of S over clusters and configurations is $\langle S \rangle$. The average of R_g for clusters with $S \geq 0.8N_{\text{fib}}$ is $\langle R_g \rangle$.

4.3 Results

To investigate the effect of friction and attractive forces on the rheological properties and structure of never-dried (ND) and dried-and-rehydrated (DR) suspensions, simulations under shear were simulated with $\mu = 0$, 5, 10, and 15 and $0 \leq A_N \leq 50$. For both the ND and DR suspensions, the simulation box side lengths were initially set to $4L$.

For the ND suspensions, three simulations at $\Phi = 0.003$ were performed for each set

of μ and A_N. Simulations were run with $\Delta t^* = 10^{-4}$ to $t^* = 1,500$ under shear, and from $t^* = 1,500$ to $t^* = 2,000$ at rest. For each run, the relative shear viscosity, $\eta_{r\mathrm{ND}}$, and the intensity of segregation, I_{ND}, were averaged over configurations for $1,000 \leq t^* \leq 1,500$. The stored elastic energy, $E^{\mathrm{elas}}_{\mathrm{ND}}$, was averaged over configurations for $1,900 \leq t^* \leq 2,000$. Simulation snapshots every dimensionless time interval of 25 for $1,200 \leq t^* \leq 1,500$ were analyzed for cluster statistics.

For the DR suspensions, five simulations at $\Phi_{min} = 0.003$ and $\Phi_{max} = 0.012$ were performed for each set of parameters. The DR suspensions were sheared in region I and at rest in region II (see Fig. 4.2). Simulations were run with $\Delta t^* = 10^{-4}$ and $t^*_{1-6} = \{100, 378, 678, 956, 1256, 1756\}$. For each run, the relative shear viscosity, $\eta_{r\mathrm{DR}}$, and the intensity of segregation, I_{DR}, and cluster properties were averaged over configurations for $956 \leq t^* \leq 1,256$. The stored elastic energy, $E^{\mathrm{elas}}_{\mathrm{ND}}$, was averaged over configurations for $1,656 \leq t^* \leq 1,756$. Simulation snapshots every dimensionless time interval of 25 for $956 \leq t^* \leq 1,256$ were analyzed for cluster statistics.

In this chapter, the subscripts ND and DR indicate values from simulations of ND and DR suspensions, with the error bars indicating the standard deviations. We are primarily interested in the difference in properties between the DR suspensions and the ND suspensions. The symbol Δ, when applied to suspension or cluster properties, refers to the difference between values for the DR suspensions, either time-dependent or time-averaged, and the steady-state values for the ND suspensions, for the same set of parameters. For example, $\Delta\eta_r = \eta_{r,DR} - \eta_{r,ND}$ is the difference between the average relative viscosities of the DR and ND suspensions.

In Fig. 4.3, $\Delta\eta_r(t^*) = \eta_{r,DR}(t^*) - \langle\eta_{r,ND}\rangle$ is plotted as a function of t^* for $\mu = 15$ and several values of A_N. The results are in the region $t^*_4 < t^* \leq t^*_5$, where the DR suspensions have been fully rehydrated to the original concentration. The DR relative viscosity values were block averaged over a dimensionless time interval of $\Delta t^* = 10$. For all values of A_N, $\Delta\eta_r(t^*)$ appears to fluctuate around a constant value that decreases as A_N is increased. This suggests that the simulations have been run sufficiently long so that steady states

have been achieved. For $A_N \gtrsim 20$, $\Delta\eta_r(t^*) < 0$, which means that the viscosities of those DR suspensions are less than that of the equivalent ND suspension, as has been reported experimentally (Missoum et al., 2012).

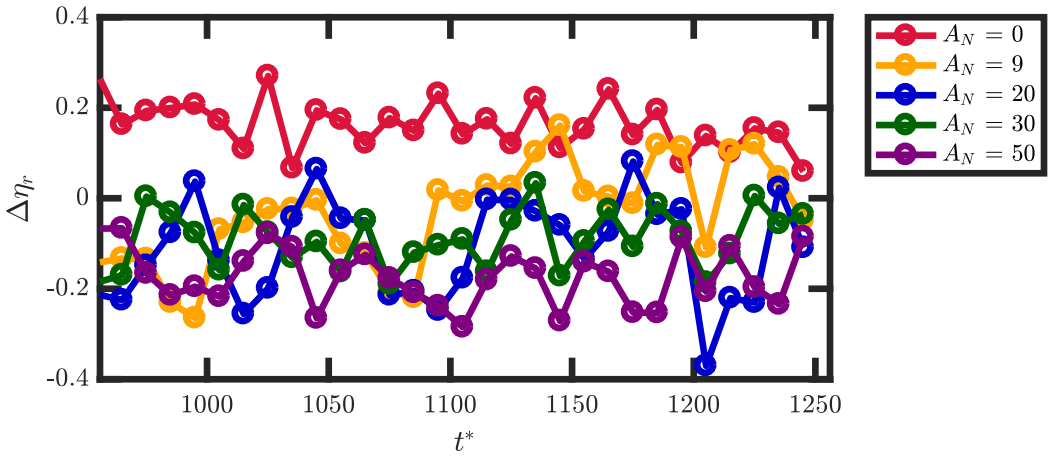


Figure 4.3: Difference of the relative viscosity of the ND and DR suspensions, $\Delta\eta_r$, as a function of dimensionless time, t^*, for $\mu = 15$.

In Fig. 4.4, $\eta_{r,ND}$, $\eta_{r,DR}$, and $\Delta\eta_r$ are plotted as a function of A_N for several values of μ. Here the viscosity values have been averaged over the region $t_4^* < t^* \leq t_5^*$. (Simulations snapshots are presented in Fig. 4.5; the stored elastic energy and the intensity of segregation are presented in Figs. 4.6 and 4.7.) In Figs. 4.4a and b, for $A_N = 0$, $\eta_{r,ND}$ and $\eta_{r,DR}$ increase as μ is increased. For $\mu = 0$, $\eta_{r,ND}$ and $\eta_{r,DR}$ are near 1, and increase slightly for large A_N. For $\mu \geq 5$, $\eta_{r,ND}$ and $\eta_{r,DR}$ increase as A_N is increased from 0 to 20. For $A_N = 20$ to 50, $\eta_{r,ND}$ plateaus while $\eta_{r,DR}$ decreases slightly. For large A_N, $\eta_{r,DR}$ is independent of μ.

The results for the ND suspensions are consistent with results previously reported for simulations of flexible fiber suspensions. For purely repulsive fibers ($A_N = 0$), the suspension viscosity increases as μ is increased (Schmid et al., 2000; Switzer and Klingenberg, 2003). For frictionless fibers ($\mu = 0$), the suspension viscosity increases as the strength of the attractive force is increased (Schmid and Klingenberg, 2000b,a). In both cases, the increase in viscosity is related to the formation of aggregates, or flocs, in the

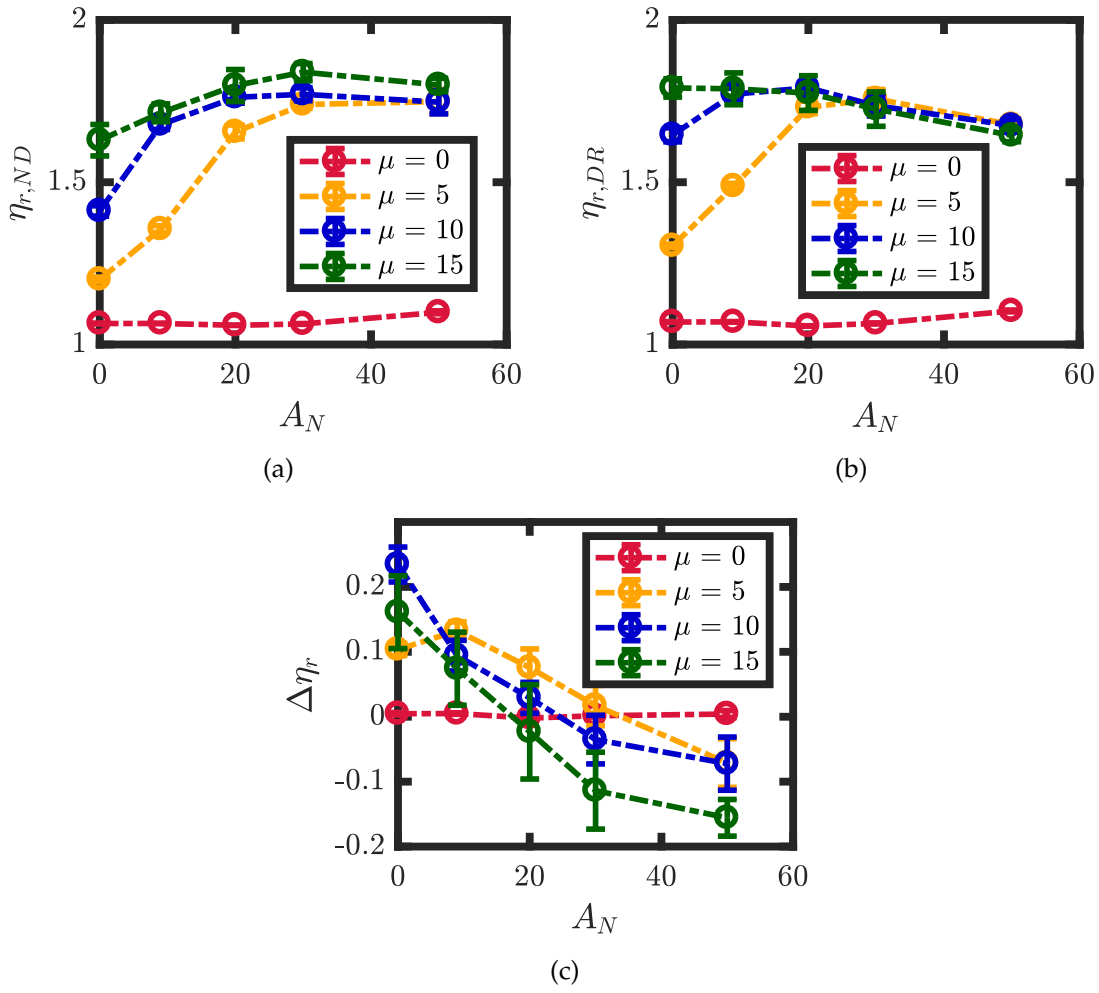


Figure 4.4: Relative viscosity, averaged over configurations, as a function of the attractive force parameter, A_N, with $\mu = 0, 5, 10$, and 15 for (a) ND suspensions, (b) DR suspensions, and (c) the difference between that for ND and DR suspensions.

suspension. Friction-induced flocculation is attributed to the elastic interlocking mechanism (Meyer and Wahren, 1964) that produces flocs and fiber networks. For flocculation via attractive forces, aggregation also occurs as the fibers form networks.

In Fig. 4.4c, $\Delta\eta_r$ is plotted as a function of A_N. Here, $\Delta\eta_r \approx 0$ for all values of A_N with $\mu = 0$. For suspensions with only repulsive interactions between fibers ($A_N = 0$), $\Delta\eta_r > 0$ for all $\mu > 0$. For $\mu \geq 5$, $\Delta\eta_r$ decreases as A_N is increased, and becomes negative for sufficiently large values of A_N. This is one of the major results of this study: the lower viscosity of dried and rehydrated suspensions relative to that of never-dried suspensions

arises from a combination of friction and attractive forces. Neither friction nor attractive forces alone can produce this phenomenon. Furthermore, there is a threshold magnitude of the attractive force that is required for this to occur. Since attractive forces will always be present (e.g., van der Waals forces), these results suggest that it should be possible to prevent the dried and rehydrated fiber suspensions from exhibiting a lower viscosity by reducing the magnitude of the attractive forces, for example, by increasing the magnitude of electrostatic or steric repulsive forces.

In Fig. 4.5, snapshots from the simulations for ND and DR suspensions are presented for four sets of parameters: $(\mu, A_N) = \{(0,0),(0,50),(15,0),(15,50)\}$. The effect of attractive forces alone is illustrated by comparing Figs. 4.5**a** and **b**. The effect of friction alone is illustrated by comparing Figs. 4.5**a** and **c**. The effect of attractive forces and friction is illustrated in Fig. 4.5**d**. For ND suspensions with $\mu = 0$, fibers are uniformly distributed in the simulation box for $A_N = 0$. As A_N is increased to 50, flocs and voids in the microstructure are observed.

For DR suspensions with $\mu = 0$, fibers are uniformly distributed in the simulation box at $t^* = t_3^*$, where $\Phi = \Phi_{\max}$. Rehydration starts at $t^* = t_3^*$, and is complete at $t^* = t_4^*$ where $\Phi = \Phi_{\min}$. The suspension expands, more quickly in the gradient (z) direction than the vorticity (y) direction, and maintains a rectangular shape with a homogeneous fiber distribution. The rectangular shape suggests that the fibers are not entangled elastically, and that the suspension expands because of shear-induced diffusion, which arises from fiber collisions and gradient drift of curved fibers (Koch, 1989; Lopez and Graham, 2007; Wang et al., 2012a). The expansion is slower with the presence of the attractive force. The difference in the shape of the suspension for the ND and DR suspensions for $\mu = 0$ has no impact on $\Delta\eta_r$, as illustrated in Fig. 4.4c.

For ND suspensions with $\mu = 15$, flocs are observed for both $A_N = 0$ and 50. Cylindrical flocs with the major axis aligned with the vorticity direction are observed for $A_N = 50$. For DR suspensions with $\mu = 15$, fibers form flocs in the simulation box at $t^* = t_3^*$, where $\Phi = \Phi_{\max}$. As Φ is decreased from $t^* = t_3^*$ to t_4^* during rehydration, the flocs with voids

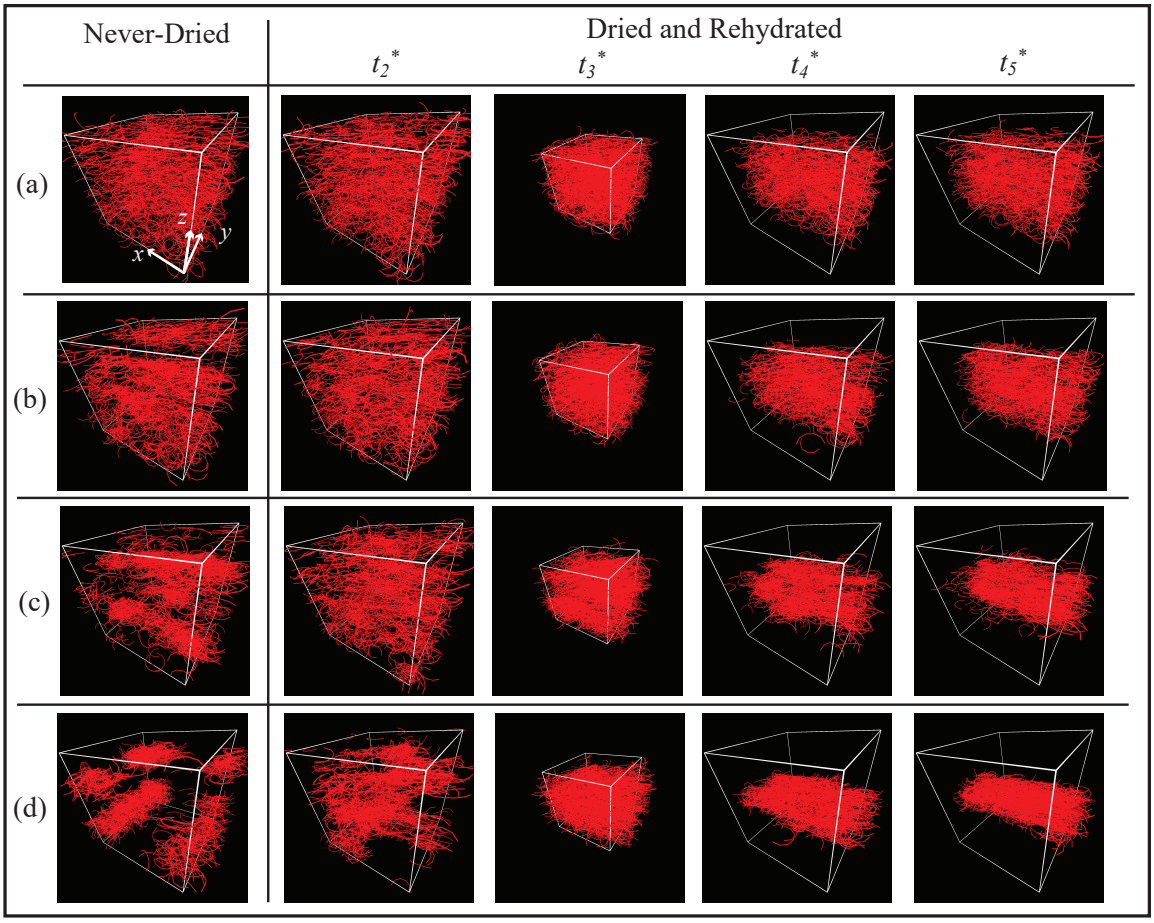


Figure 4.5: Simulation snapshots for ND suspensions at $t^* = 1500$ and DR suspensions at t_1^*, t_3^*, t_4^* and t_5^* (a) $\mu = 0$ and $A_N = 0$, (b) $\mu = 0$ and $A_N = 50$, (c) $\mu = 15$ and $A_N = 0$, and (d) $\mu = 15$ and $A_N = 50$. Here, x, y, and z are the flow, vorticity, and gradient directions, respectively. Periodic images are included for fibers that cross the boundaries.

between them are observed. The network remains intact and homogeneous for $A_N = 50$ and is rectangular, long in the flow direction and short in the gradient direction. From $t^* = t_4^*$ to t_5^*, the structure of the network for both $A_N = 0$ and 50 do not change appreciably. This is consistent with the results in Fig. 4.3, where the relative viscosity is largely independent of time during this period.

For both $\mu = 0$ and 15, the DR fiber networks are more coherent for $A_N = 50$ than for $A_N = 0$. For $\mu = 15$ and $A_N = 50$, a persistent and non-redispersible network is obtained, and is associated with the most negative $\Delta\eta_r$, as plotted in Fig. 4.4c. In contrast, for the

same set of parameters, the ND suspension forms a collection of flocs as opposed to a single network. Thus the negative value of $\Delta\eta_r$ is attributed to a greater degree of fiber aggregation.

At $t^* = t_5^*$, the shear flow is stopped. The segments continue to move as the fibers, deformed by the flow, relax and attempt to reach their equilibrium shapes. If the coefficient of friction is sufficiently large, fibers will come to rest, by $t^* = t_6^*$, locked in elastically-strained, nonequilibrium configurations (Meyer and Wahren, 1964; Switzer and Klingenberg, 2003), leading to a nonzero value of the stored elastic energy. In Fig. 4.6, the stored elastic energies at $t^* = t_6^*$, $\langle E_{ND}^{\text{elas}} \rangle$, $\langle E_{DR}^{\text{elas}} \rangle$, and ΔE^{elas}, are plotted as a function of A_N for several values of μ. For both ND and DR suspensions with $\mu = 0$, $\langle E^{\text{elas}} \rangle = 0$ for all A_N. Without friction, interacting fibers can still slide past one another to reach their equilibrium shapes. For $\mu \geq 5$, $\langle E_{ND}^{\text{elas}} \rangle$ increases as A_N is increased, and $\langle E_{DR}^{\text{elas}} \rangle$ passes through a maximum at $A_N \approx 30$. For both types of suspensions, fibers are elastically interlocked, held in place by friction.

In Fig. 4.6c, $\Delta E^{\text{elas}} > 0$ and exhibits maxima at $A_N \approx 20$ for $\mu \geq 5$, indicating that the DR fibers are more deformed, and in this sense, more strongly aggregate. The change in the stored elastic energy due to drying and rehydration increases as μ is increased, with nearly identical curves for $\mu = 10$ and 15. In Fig. 4.4c, $\Delta\eta_r$ is more negative for $\mu = 15$ than for $\mu = 10$. Thus the difference in the rheological properties is not completely correlated with the fiber deformation alone, and other measures of the structure are required.

In Fig. 4.7, the intensities of segregation, $\langle I_{ND} \rangle$, $\langle I_{DR} \rangle$, and ΔI, are plotted as a function of A_N for several values of μ for the suspensions in Region II. For both the ND and DR suspensions, $\langle I \rangle$ increases as μ or A_N is increased. In Fig. 4.7c, $\Delta I > 0$ for all μ and A_N, quantifying the increase in heterogeneity caused by the drying and rehydration process. As μ is increased, ΔI increases. In contrast to $\Delta\eta_r$, ΔI is not sensitive to changes in A_N.

Cluster statistics were calculated for the largest cluster in each simulation snapshot. In Fig. 4.8, the size of the largest cluster, S, normalized by the number of fibers, N_{fib}, are plotted as a function of A_N for the ND and DR suspensions. For the ND suspensions,

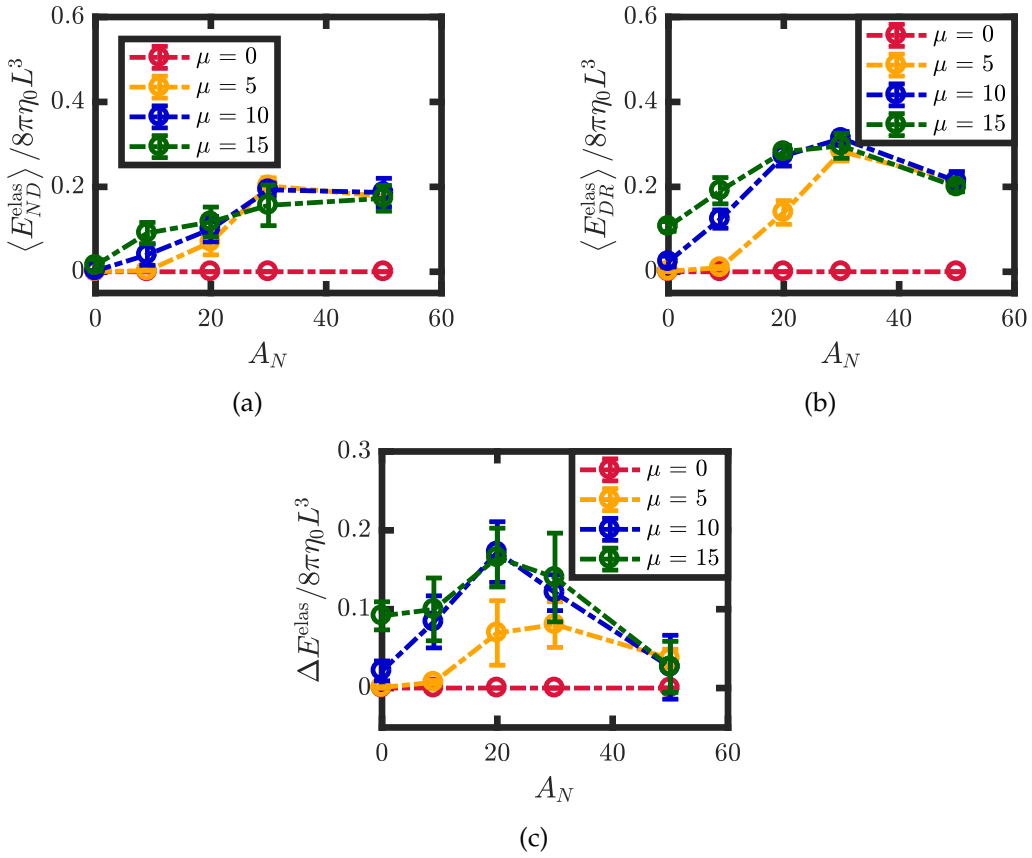


Figure 4.6: Stored elastic energy, averaged over fibers and configurations, as a function of the attractive force parameter, A_N, with $\mu = 0, 5, 10$, and 15 for (a) ND suspensions, (b) DR suspensions, and (c) the difference between that for ND and DR suspensions.

$\langle S_{ND} \rangle$ increases as A_N is increased for $\mu \leq 5$. For $\mu \geq 10$, $\langle S_{ND} \rangle$ is roughly constant at approximately 50% of $N_{\rm fib}$. For the DR suspensions, $\langle S_{DR} \rangle \approx 1$ for $\mu \geq 5$ and $A_N \geq 20$. The drying and rehydration process consolidated fibers into larger clusters.

For DR suspensions, properties of clusters that contain at least 80% of the fibers in the suspension are calculated. In Fig. 4.9a, the radius of gyration, $\langle R_{g,DR} \rangle$, normalized by the simulation box side length, $L_{\rm box}$ is plotted as a function of A_N. For $A_N \geq 30$, $\langle R_{g,DR} \rangle$ is larger for $\mu = 0$ than for $\mu \geq 5$ because frictionless aggregates readily redisperse via shear-induced diffusion upon rehydration. For $\mu = 5$, $\langle R_{g,DR} \rangle$ decreases as A_N is increased from 9 to 50. The radius of gyration for $\mu = 10$ and 15 are indistinguishable, which is consistent with the results for the stored elastic energy (Fig. 4.6), the intensity of

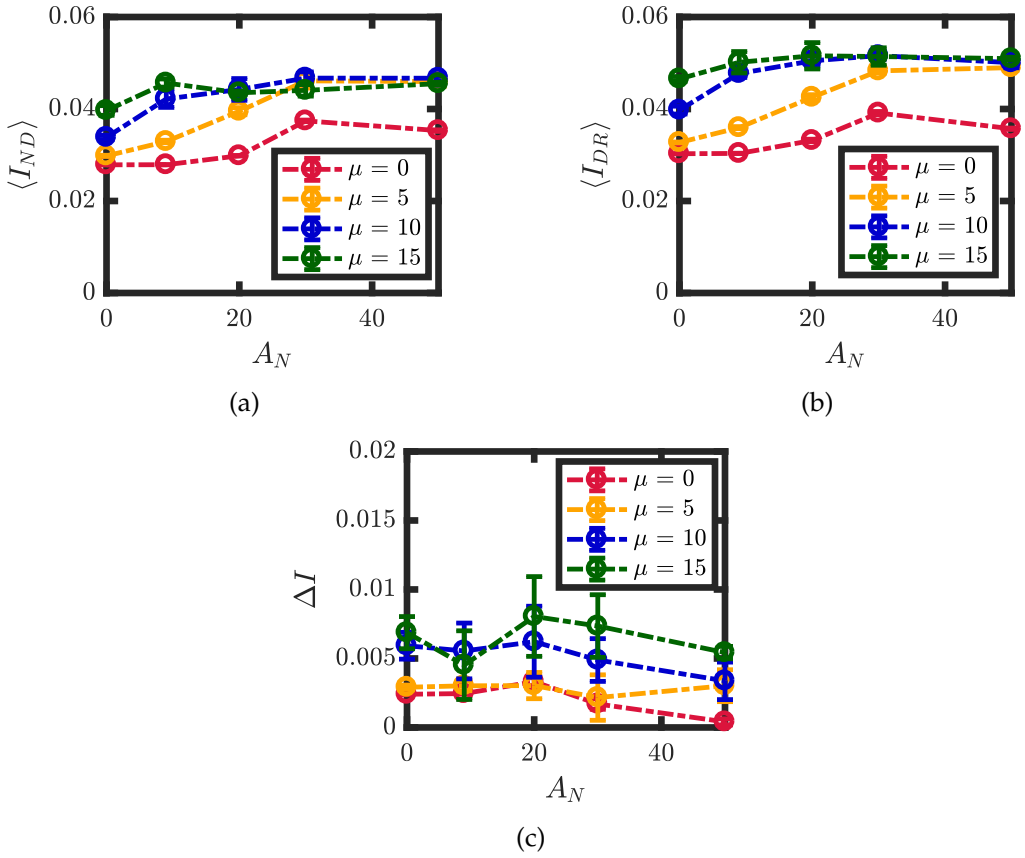


Figure 4.7: Intensity of segregation, averaged over configurations, as a function of the attractive force parameter, A_N, with $\mu = 0, 5, 10$, and 15 for (a) ND suspensions, (b) DR suspensions, and (c) the difference between that for ND and DR suspensions.

segregation (Fig. 4.7), and the number of fibers in the largest cluster (Fig. 4.8). The shear force stretches the flocs in the direction of flow, and thus $\langle R_{g,DR} \rangle \approx L_{\text{box}}$ for all $\mu \geq 5$.

In Fig. 4.9b, the quantity $8\lambda_1^2\lambda_2^2\lambda_3^2$, which is roughly the volume of the cluster, $\langle V_{DR} \rangle$, is plotted as a function of A_N. The angle brackets indicate averaging over clusters with $S \geq N_{\text{fib}}$ and configurations. For all μ, $\langle V_{DR} \rangle$ monotonically decreases as A_N is increased. Along with the observation that $\langle S_{DR} \rangle$ plateaus for $\mu \geq 5$ and $A_N \geq 20$, the formation of the densest flocs for $\mu \geq 10$ and $A_N \geq 30$ correlates with $\Delta\eta_r < 0$.

In Fig. 4.10a, the separation between fibers in contact for DR suspensions, $\langle h_{DR} \rangle$, is plotted as a function of A_N. Angle brackets indicate the average over pairs of fibers in contact and configurations. For all μ, $\langle h_{DR} \rangle$ decreases as A_N is increased. At $\mu = 0$, h_{DR}

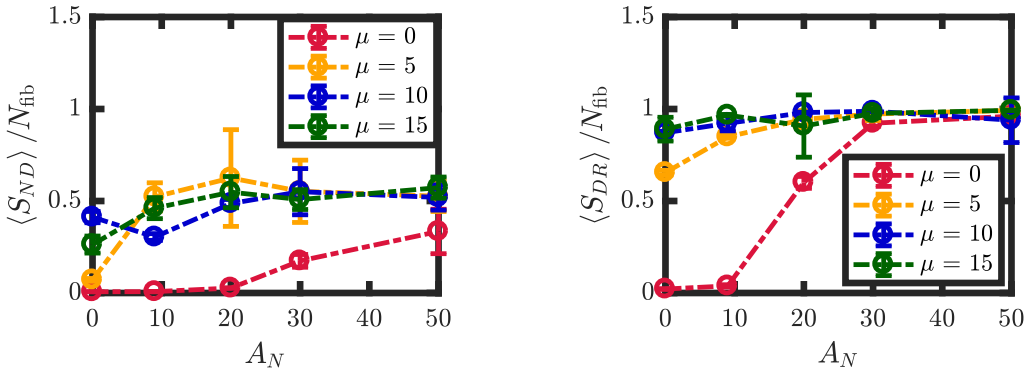


Figure 4.8: The number of fibers in the largest cluster, S, averaged over configurations and normalized by N_{fib}, as a function of the attractive force parameter, A_N, for (a) ND suspensions and (b) DR suspensions.

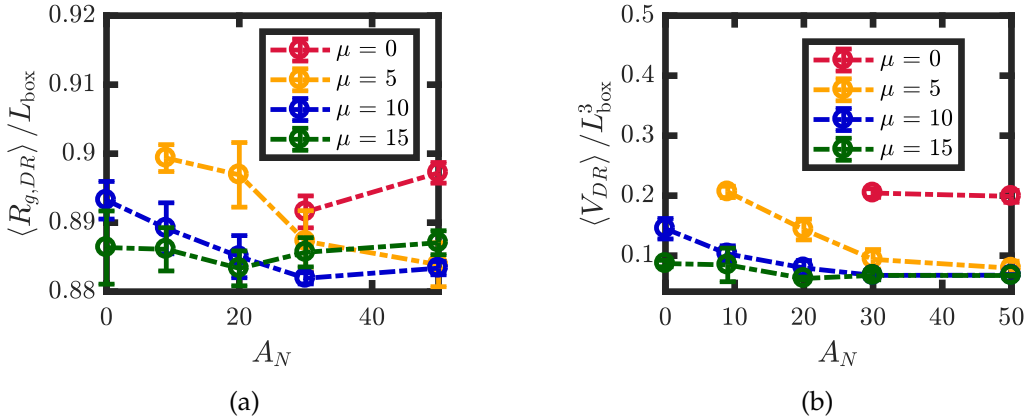
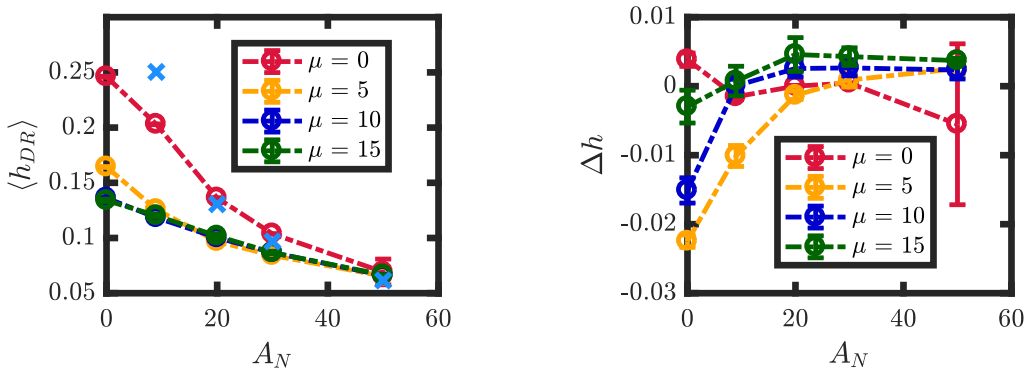


Figure 4.9: Cluster statistics, averaged over configurations, for DR suspensions as a function of the attractive force parameter A_N: (a) radius of gyration, $R_{g,\text{DR}}$, normalized by the simulation box side length, L_{box}, and (b) the volume of the cluster, V_{DR}, normalized by L_{box}^3. Only clusters that contain at least 80% of the fibers are characterized.

is the largest for all A_N. The mechanical equilibrium separations, where the net normal force in Eq. 4.4 is 0, are also plotted as the light blue xs. As expected, the fiber separations are near their equilibrium values for $\mu = 0$. The separations decrease when friction is added, which indicates that elastic interlocking is significant; for $\langle h_{DR} \rangle$ smaller than the equilibrium value, the net normal force is repulsive, which allows friction to contribute to the network strength. At $A_N = 50$, $\langle h_{DR} \rangle$ for all μ are indistinguishable from the

mechanical equilibrium separation. In Fig. 4.10b, Δh is plotted as a function of A_N. For $\mu = 10$ and 15, $\Delta h > 0$ for $A_N \geq 9$ despite the fact that the DR networks are more densely packed. Simulations with $\Delta\eta_r < 0$ also exhibit $\Delta h > 0$.



(a) (b)

Figure 4.10: The separation between fibers in contact, h, averaged over configurations, as a function of the attractive force parameter, A_N for (a) DR suspensions and (b) difference with ND suspensions. Blue crosses are the mechanical equilibrium separations.

4.4 Conclusions

The structure and rheological properties of suspensions of linked-rigid spherocylinders, with friction, attractive, and repulsive normal forces, were investigated via fiber-level simulations. The linked-rigid spherocylinder is a model for microfibrillated cellulose fibers. Never-dried (ND) suspensions are simulated at $\Phi = 0.003$, where Φ is the volume fraction. The volume fraction of dried-and-rehydrated (DR) suspensions was initially 0.003, dried to 0.012, and rehydrated back to 0.003. The simulation box side lengths were changed every 1,000 dimensionless time steps during drying and rehydration. For drying, the fibers were moved affinely with the box after the equations of motion for the time step were integrated. For rehydration, fibers were moved by only integrating the equations of motion.

When sufficient attraction and friction were applied, the relative shear viscosity, η_r

of the DR suspensions are smaller than those of the ND suspensions, with the same parameters. The reduction in η_r is associated with the formation of irredispersible flocs. Floc formation occurs through the elastic interlocking mechanism, indicated by positive stored elastic energy for both the ND and DR suspensions. For a coefficient of friction of $\mu = 10$ and 15, the drying and rehydration process especially compacts all the fibers in the suspension into clusters with small volumes. Despite the compactness, the average separation between fibers in contact are larger for the DR suspensions than for the ND suspensions.

5

CONCLUSIONS AND FUTURE DIRECTIONS

The major goal of this thesis was to gain fundamental understanding of the relationship between rheological properties and structure of fiber suspensions, and thereby address challenges in producing suspensions of cellulose nanocrystals and microfibrillated cellulose fibers.

5.1 Contributions

In Chapter 3, Brownian dynamics simulations were conducted for suspensions of rigid spherocylinders, as a model for rodlike colloidal particles, such as cellulose nanocrystals (CNCs). To mimic a hard spherocylinder interaction, the spherocylinders were modeled to interact through a soft repulsive force that decays exponentially with the surface separation.

The diffusivities matched results from simulations of hard spherocylinders with aspect ratio $r_p = 5$ and volume fraction $\Phi \leq 0.4$. Liquid crystalline phases were identified using orientational and hexatic order parameters, and pair distribution functions. A phase diagram of liquid crystalline phases with $2 \leq r_p \leq 30$ and $0.001 \leq \Phi \leq 0.7$, including isotropic, nematic, smectic, and solid phases, were constructed, and qualitatively

agreed with that for hard spherocylinders reported by Bolhuis and Frenkel (1997).

The typical three-region flow curves observed experimentally for rodlike colloidal suspensions were reproduced in the simulations. A shear thinning region at low Péclet numbers, Pe, was followed by a viscosity plateau at intermediate Pe, which was followed by another shear thinning region at high Pe. The viscosity and order parameter qualitatively matched experimental values for CNC suspensions. Despite that fibers only interact via repulsion, large viscosity values similar in magnitude to experimental measurements were obtained.

The transient rheology and structure of suspensions that were nematic at rest exhibited a variety of behaviors that depended on the dynamics of domains at different Pe and Φ. System-wide domains that align in and kayak about the vorticity direction, domains that rotate locally, and layered domains were observed. Oscillations in the order parameter, viscosity, and the first and second normal stress differences correlated with the structure, consistent with previously reported experiments (Ivanov et al., 1982) and simulations of flexible fibers (Ross and Klingenberg, 1997).

In Chapter 4, fiber-level simulations were employed to investigate the effect of drying and rehydration on the rheological properties and structure of suspensions of non-Brownian flexible fibers. As a model for nanofibrillated cellulose fibers (NFCs), each fiber consisted of 5 rigid spherocylinders, connected by joints with bending and twisting potentials. The fibers were curved with equilibrium bending and twisting angles between fiber segments of 0.6 and 0. Fibers interacted through friction, and normal repulsive and attractive forces, as a model for hydrogen bonding and/or van der Waals interactions.

A drying and rehydration process was implemented such that the volume fraction was first raised 4 fold from the baseline value of 0.003. During drying, simulation box was shrunk every dimensionless time interval of 0.1. The fibers were moved affinely with the box after the equations of motion were integrated. The suspensions were then rehydrated by expanding the simulation box. When the friction and attractive forces were sufficiently large, the relative shear viscosities of the suspensions that were dried

and rehydrated were smaller than those of the never dried suspension with the same parameters. The reduction in viscosity was associated with the formation of dense and persistent aggregates, which have also been observed eperimentally for NFC (Missoum et al., 2012).

5.2 Future research directions

5.2.1 *Experiments*

Experimentally, nanofibrillated (or microfibrillated) cellulose fiber suspensions are typically dried to remove all of the solvent. In Chapter 4, a drying and rehydration process was implemented such that the volume fraction of a dilute suspension was raised only 4 fold. The number of fibers, required to maintain a simulation box side length of at least 1.5 times of the fiber length at the largest values of Φ, grows quickly with the baseline Φ and how much it is increased. To efficiently explore the effect of different parameters, the fiber concentrations in the simulations were limited. For fundamental understanding of the aggregates that form due to drying and rehydration, and to compare with simulation studies, only a portion of the solvent should be removed in experiments, without completely drying the suspension. The suspension should then be rehydrated back to the same concentration.

In Chapter 3, Brownian dynamics simulations revealed that a wide range of domains formed from shearing suspensions that were nematic at rest. Flow visualization experiments using devices, such as a transparent cylinder similar to that used by Karppinen et al. (2012), should be conducted to confirm the existence of the domains and to investigate the relationship between the domain dynamics and oscillations in the rheological properties.

5.2.2 *Simulations*

The rheological properties and microstructure of fiber suspensions depend on many variables. The number of simulations required to understand the effect of each variable quickly grows with the number of independent variables. All of the parameter space was not explored in this thesis. Other regions of the parameter space should be explored to gain a fuller understanding of the systems. Recommendations for investigations of other parameters and suggestions for features to add to the fiber models and characterization techniques are included in this section.

Brownian dynamics simulations of suspensions of spherocylinders. Cinacchi et al. (2004) simulated binary mixtures of suspensions of hard spherocylinders with different aspect ratios. The liquid crystalline phases obtained depended on the composition of the mixture, and the aspect ratios. Different forms of the smectic A phase were observed. In real systems of fiber suspensions, a distribution in the fiber dimensions is typically observed (Elazzouzi-Hafraoui et al., 2008). Similar binary mixtures of spherocylinders or ones with a continuous distribution of aspect ratios should be simulated for realistic systems.

In Chapter 3, fibers only interact via electrostatic repulsion, which accounts for the effect of excluded volume. In suspensions of cellulose nanocrystals, hydrogen bonding and/or van der Waals interactions also contribute to the structure and rheological properties (Ureña-Benavides et al., 2011; Shafiei-Sabet et al., 2012, 2014). To investigate the effect of attractive forces, the form of the short-ranged atttraction that exponentially decays with the separation between fiber surface employed in Chapter 4 can be implemented.

Other forms of attractive force should also be considered. Bolhuis et al. (1997) investigated the effect of employing either nonadditive attraction due to depletion forces or long-ranged pairwise additive attraction on the phase transitions of suspensions of hard spherocylinders. The form of the attractive force has an impact on the phase be-

havior. The angle-dependent attractive force developed by Lintuvuori and Wilson (2008) is another form of attractive force that can be easily implemented. Comparison of the rheological properties and microstructure with both experimental and simulation results could potentially reveal the importance of different types of attractive forces in physical systems.

In Chapter 3, the nematic, smectic, and solid phases were observed in simulations. The chiral nematic phase (Dong et al., 2002) that arises from anisotropy in either the fiber structure or the interactions between fibers (Straley, 1976; Orts et al., 1998) was missing. For realistic simulations, one can introduce anisotropy in the system by adding a dipole on one end of the fiber, such as the spherocylinders modeled by Van Duijneveldt et al. (2000) so that fibers interact anisotropically. The effect of the anisotropy on the domain dynamics and rheological properties can then be compared to simulation results from shearing nematic suspensions presented in Chapter 3.

In Chapter 4, the number of fibers in clusters and the dimension of clusters were calculated for suspensions of flexible fibers, where each fiber in the cluster can reach another through a series of direct contacts between fibers. In Chapter 3, domains arose from shearing suspensions that were nematic at rest. Because the volume fractions were large, most of the fibers in the suspension belonged to the same cluster. Local domains, where fibers rotate coherently or form layers with shear, cannot be distinguished from surrounding fibers using the cluster identification algorithms used for flexible fibers. A new cluster identification algorithm that potentially takes into account both the separation and angle between fibers should be developed. Cluster statistics, such as the distribution of the number of fibers in the clusters, can then be used to quantify the domain dynamics at various Péclet numbers and concentrations.

Fiber-level simulations of suspensions of non-Brownian linked-spherocylinders. In Chapter 4, U-shaped fibers of uniform shape and dimensions were employed to investigate the effect of drying and rehydration on fiber suspensions. Similar studies using

straight fibers and helical fibers can be conducted to determine the effect of shape on the formation of persistent aggregates and rheological properties. Mixtures of straight and curved fibers should be simulated to examine the impact on flocculation of sheared suspensions. Different forms of the attractive force, mentioned above for Brownian dynamics simulations of spherocylinders, can be similarly employed for suspensions of flexible fibers.

In Chapter 4, the drying and rehydration process was implemented such that the fiber suspensions were sheared. In real systems, the suspensions are often at rest during drying or storage. Simulations where the drying and rehydration process does not include shear should be conducted. Settling effects on the redispersibility can also be investigated by including gravity in the equations of motion.

A

INERTIAL MODEL FOR BROWNIAN RIGID SPHEROCYLINDERS

Inertia in the general Langevin equation is often neglected (Hinch, 1994; Córdoba et al., 2012) to increase the computational efficiency of Brownian dynamics simulations (Grassia et al., 1995). The removal of inertia from the equations of motion used to generate the results reported in Chapter 3 is validated here. The translational and rotational diffusivities from simulations employing the inertial and inertialess models are compared. The model and simulation methods closely follow that for Brownian linked-spherocylinders by Bette (2003) with the number of segments set to 1.

a.1 Model

The translational and rotational equations of motion for a Brownian rigid spherocylinder (or fiber) i, with mass, m_i, and moment of inertia tensor, $\boldsymbol{I}_i$, are

$$\boldsymbol{F}_i^{\text{hyd}} + \boldsymbol{F}_i^{\text{Br}} + \sum_j^{N_{Ci}} \boldsymbol{F}_{ij}^{\text{con}} = m_i \frac{d^2 \boldsymbol{r}_i}{dt^2}, \tag{A.1}$$

$$\boldsymbol{T}_i^{\text{hyd}} + \boldsymbol{T}_i^{\text{Br}} + \sum_j^{N_{Ci}} \boldsymbol{G}_{ij} \times \boldsymbol{F}_{ij}^{\text{con}} = \boldsymbol{I}_i \cdot \frac{d\boldsymbol{w}_i}{dt} + \tilde{\boldsymbol{w}}_i \cdot (\boldsymbol{I}_i \cdot \boldsymbol{w}_i) \tag{A.2}$$

where $\boldsymbol{r}_i$ is the center of mass, $\boldsymbol{w}_i$ is the angular velocity, and N_{Ci} is the number of fibers in contact with fiber i. The hydrodynamic force and torque are $\boldsymbol{F}_i^{\text{hyd}}$ and $\boldsymbol{T}_i^{\text{hyd}}$; the Brownian force and torque are $\boldsymbol{F}_i^{\text{Br}}$ and $\boldsymbol{T}_i^{\text{Br}}$. The contact force, $\boldsymbol{F}_{ij}^{\text{con}}$, and the moment arm, $\boldsymbol{G}_{ij}$, are described in Chapter 2. Neglecting inertia, the translational and rotational equations of motion are

$$\boldsymbol{F}_i^{\text{hyd}} + \boldsymbol{F}_i^{\text{Br}} + \sum_j^{N_{Ci}} \boldsymbol{F}_{ij}^{\text{con}} = 0, \tag{A.3}$$

$$\boldsymbol{T}_i^{\text{hyd}} + \boldsymbol{T}_i^{\text{Br}} + \sum_j^{N_{Ci}} \boldsymbol{G}_{ij} \times \boldsymbol{F}_{ij}^{\text{con}} = 0 \tag{A.4}$$

a.2 Simulation Methods

The moment of inertia tensor, $\boldsymbol{I}_i^{\text{body}}$, is a diagonal matrix in the body frame with elements

$$I_{xx} = I_{yy} = \frac{1}{12} m \left(4l^2 + 3b^2\right), \qquad I_{zz} = \frac{1}{2} b^2 m \tag{A.5}$$

where b is the radius and l is the half-length. Using Eq. 2.38, $\boldsymbol{I}_i^{\text{body}}$ and $\left(\boldsymbol{I}_i^{\text{body}}\right)^{-1}$ are transformed to the inertial frame

$$\boldsymbol{I}_i = I_{xx}\boldsymbol{\delta} + \left(I_{zz} - I_{xx}\right) \boldsymbol{p}_i \boldsymbol{p}_i \tag{A.6}$$

$$\boldsymbol{I}_i^{-1} = I_{xx}^{-1}\boldsymbol{\delta} + \left(I_{zz}^{-1} - I_{xx}^{-1}\right) \boldsymbol{p}_i \boldsymbol{p}_i \tag{A.7}$$

where $\boldsymbol{p}_i$ is the fiber orientation. The equations of motion are nondimensionalized using the scales summarized in Sec. 2.12 along with the scales for mass and moment of inertia, $6\pi\eta_0 l t_s$ and $8\pi\eta_0 l^3 t_s$. The dimensionless mass and moment of inertia are thus

$$m^* = \frac{m}{6\pi\eta_0 l t_s}, \qquad \boldsymbol{I}^* = \frac{\boldsymbol{I}}{8\pi\eta_0 l^3 t_s} \tag{A.8}$$

where η_0 is the viscosity of the suspending fluid, and t_s is the time scale. Fibers interact via both repulsive and attractive forces. For the results reported in this appendix, the normal force between fiber surfaces, governed by Eq. 2.28, is set with $F^{\text{rep}} = 150F_s$, $F^{\text{att}} = 100F_s$, $a^{\text{rep}} = 20$, and $a^{\text{att}} = 35$, where F_s is the force scale, F^{rep} and F^{att} are the repulsive and attractive force coefficients, and a^{rep} and a^{att} are the repulsive and attractive decay parameters. The dimensionless equations of motion are

$$\ddot{\boldsymbol{r}}_i^* = \frac{1}{m^*}\left(\boldsymbol{F}_i^{\text{hyd}*} + \boldsymbol{F}_i^{\text{Br}*} + \sum_j^{N_{Ci}} \boldsymbol{F}_{ij}^{\text{con}*}\right), \tag{A.9}$$

$$\dot{\boldsymbol{w}}_i^* = (\boldsymbol{I}_i^*)^{-1} \cdot \left[\boldsymbol{T}_i^{\text{hyd}*} + \boldsymbol{T}_i^{\text{Br}*} + \frac{3}{4r_p^2}\sum_j^{N_{Ci}} \boldsymbol{G}_{ij}^* \times \boldsymbol{F}_{ij}^{\text{con}*} - \tilde{\boldsymbol{w}}_i^* \cdot (\boldsymbol{I}_i^* \cdot \boldsymbol{w}_i^*)\right] \tag{A.10}$$

where the asterick indicates dimensionless quantities, and $r_p = l/b$ is the aspect ratio. The right-hand side of Eqs. A.9 and A.10 are calculated from the fiber positions and orientations for $\ddot{\boldsymbol{r}}_i^*(t^* + \Delta t^*)$ and $\dot{\boldsymbol{w}}_i^*(t^* + \Delta t^*)$, and dimensionless Brownian force and torque in Eqs. 2.74 and 2.75. The equations of motion are numerically integrated using

$$\boldsymbol{r}_i^*\left(t^* + \Delta t^*\right) = \boldsymbol{r}_i^*\left(t^*\right) + \dot{\boldsymbol{r}}_i^*\left(t^*\right)\Delta t^* \tag{A.11}$$

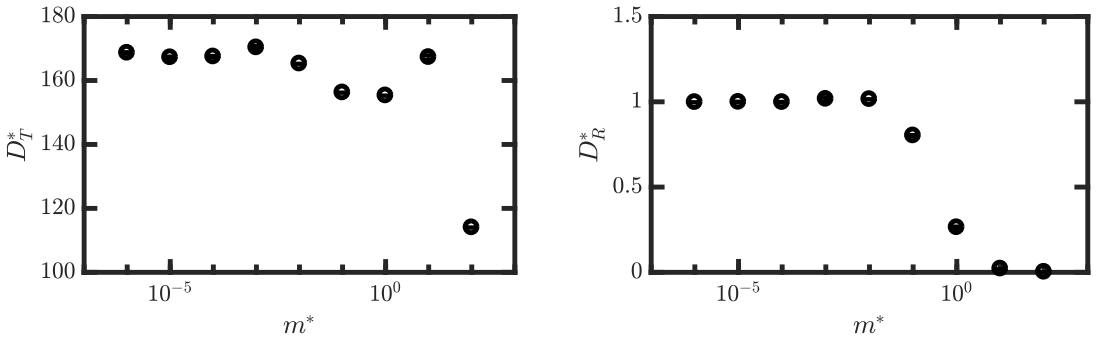
$$\dot{\boldsymbol{r}}_i^*\left(t^* + \Delta t^*\right) = \dot{\boldsymbol{r}}_i^*\left(t^*\right) + \ddot{\boldsymbol{r}}_i^*\left(t^* + \Delta t^*\right)\Delta t^* \tag{A.12}$$

$$\boldsymbol{q}(t^* + \Delta t^*) = \boldsymbol{q}_i^*\left(t^*\right) + \dot{\boldsymbol{q}}_i^*\left(t^*\right)\Delta t^* \tag{A.13}$$

$$\boldsymbol{w}_i^*\left(t^* + \Delta t^*\right) = \boldsymbol{w}_i^*\left(t^*\right) + \dot{\boldsymbol{w}}_i^*\left(t^* + \Delta t^*\right)\Delta t^* \tag{A.14}$$

where $\boldsymbol{q}$ is the vector of Euler parameter, and the form for $\dot{\boldsymbol{q}}$ is given by Eq. 2.69. The simulation methods for the inertialess model and the postprocessing methods for the diffusivities are described in Chapter 2.

a.3 Results



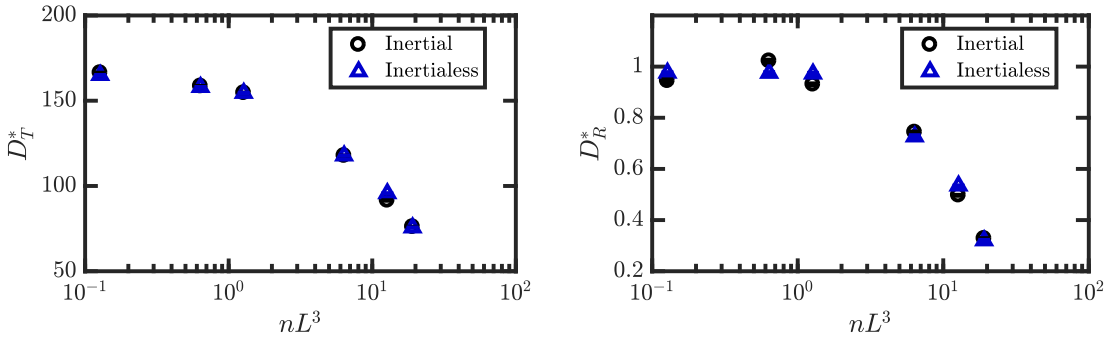
(a) (b)

Figure A.1: Diffusivities as a function of dimensionless mass, m^*: (a) Translational diffusivity, (b) rotational diffusivity. Simulations employed the inertial model. The error bars indicate the 95% confidence interval from linear regressions. The dimensionless time steps used are 10^{-7}, 10^{-6}, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-3}, 10^{-2}, and 10^{-2} as m^* is increased.

Simulations without shear were conducted to investigate the effect of inertia on the diffusivities. Fibers with $r_p = 10$ were placed with random positions and orientations into cubic simulation boxes that have dimensionless side lengths of 500. Using 3,200 fibers, the dimensionless concentration, nL^3 is 0.27, where n is the number density and L is the fiber length. The simulations were run to $t^* = 20$, where t^* is the dimensionless time. As the dimensionless mass, m^*, is increased, the dimensionless time step, Δt^*, was increased to maintain computational efficiency. The translational and rotational diffusivities, D_T^* and D_R^*, nondimensionalized by the short time estimates, were calculated by linear regressions based on Eqs. 2.133 and 2.135.

In Fig. A.1, D_T^* and D_R^* are plotted as functions of m^*. As m^* is increased, D_T^* fluctuates around a fixed value then drops by 30% from $m^* = 10$ to 10^2; D_R^* fluctuates around 1 for $10^{-6} \leq m^* \leq 10^{-2}$ then decreases to small, nonzero values.

Simulations without shear were conducted to investigate the effect of concentration on the translational and rotational diffusivities. Using values typical for cellulose nanocrystals, $r_p = 50$, $b = 10$ nm, fiber density of 10^3 kg·m^{-3} and $\eta_0 = 10^{-3}$ Pa·s, $D_R = \mathcal{O}\left(10^4\right)$ s^{-1}



(a) (b)

Figure A.2: Diffusivities as a function of dimensionless concentration, nL^3, from simulations employing the inertial and inertialess model: (a) Translational diffusivity, (b) rotational diffusivity. The error bars indicate the 95% confidence interval from linear regressions.

and $m^* = \mathcal{O}\left(10^{-6}\right)$. Fibers with $m^* = 10^{-4}$, chosen to be slightly larger than typical values of cellulose nanocrystals to extend of the range that inertia can be safely neglected, and $r_p = 10$ were placed with random positions and orientations into cubic simulation boxes. The box side lengths were adjusted for 3,200 fibers and $0.1 \leq nL^3 \leq 75$. The simulations were run with $\Delta t^* = 5 \times 10^{-6}$ to $t^* = 20$ for both the inertial and inertialess models. In Fig. A.2, D_T^* and D_R^* are plotted as functions of nL^3 for both models. As nL^3 is increased, D_T^* and D_R^* both decrease. For the range of nL^3 investigated, the diffusivities from both models are equivalent.

Combining the findings from Figs. A.1 and A.2, neglecting inertia has no impact on the diffusivities of dilute and semi-dilute suspensions of Brownian rigid spherocylinders for $m^* \leq 10^{-2}$. The majority of results reported in Chapter 3, including the diffusivities and rheological properties, were generated from dilute and semi-dilute suspensions. The typical value for cellulose nanocrystals is far from the limit of $m^* = 10^{-2}$. For the subjects of the studies, inertia can be safely neglected.

B

GPU COMPUTING VIA CUDA

Fiber-level simulation offers a systematic approach to explore the parameter space, which expands quickly with the number of independent variables. The simulation time increases when the system size is increased or the time step is decreased. Thus, computationally efficient programs are required to obtain simulations results for a large number of simulations in reasonable time, *i. e.*, less than a couple of days each.

Programs can be parallelized to run on both the CPU and GPU to increase the computational efficiency via a myriad of programming languages and techniques, such as simultaneous multithreading, CUDA, OpenACC, OpenMP, and MPI. Amdahl's law (Amdahl, 1967) can be used to evaluate the potential speedup, S, under the best case scenario,

$$S = \frac{1}{(1-f) + \frac{f}{n}} \tag{B.1}$$

where f is the fraction of the time a program spends in portions of parallelizable code, and n is the number of threads (or processors) available. For the Brownian rigid spherocylinder program, $f \lessapprox 1$. For the non-Brownian linked-rigid spherocylinder program, $f < 1$ due to the constraint of no relative motion in Eq. 2.30. The contact matrix in Eq. 2.110 is constructed, and the friction forces are solved; the process repeats if a contact

is broken. While the contact matrix can be constructed in parallel, the process of construction and evaluation only occurs serially. The process is especially time-consuming for contact groups with many fibers, wherein multiple contacts may be broken. While Amdahl's law is modified in the multicore era (Hill and Marty, 2008), the concept of evaluating S before deciding on an approach for increasing the computational efficiency still applies.

This appendix specifically focuses on GPU-accelerated programming via CUDA. As the language and architecture are fast-evolving, only key concepts are included. Refer to the official book (Sanders and Kandrot, 2011) and website by NVIDIA for tutorials. The thesis by Wilson (2016) offers another example of parallelizing simulation programs. To demonstrate optimizing and evaluating the performance of a CUDA program, the process of choosing the neighbor detection algorithm for the Brownian rigid spherocylinder program is presented.

b.1 Introduction to CUDA

Prior to the release of CUDA by NVIDIA in 2006, GPUs were employed for graphics programming using languages such as OpenGL, which is an advanced language that is difficult for average users. CUDA enabled the easy use of the GPUs for scientific computing, aided by a plethora of libraries. For example, the `cusparse` and `cublas` libraries were used to solve sparse systems of equations, and the `curand` library was used to generate random numbers for programs used in this thesis.

While a CUDA program is constrained to run only on NVIDIA produced GPUs, it is flexible to the type of GPU, which can have different numbers of streaming multiprocessors (SMs). For example, the same program can run on GTX-1080, with 20 SMs, and Titan RTX, with 72 SMs. The computation load is automatically distributed to the SMs available at runtime without requiring user input. OpenCL, released in 2009, is another framework for cross-platform programming. While OpenCL is not constrained to NVIDIA produced

GPUs, it is harder to learn for beginners in parallel programming.

The CUDA language can be thought of as C with extensions, described below.

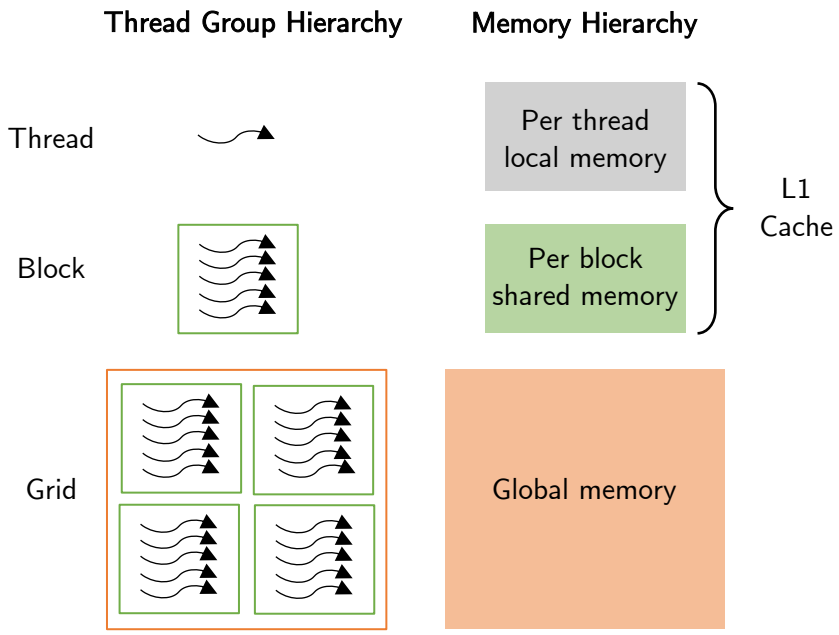


Figure B.1: Schematic diagram of the thread group heirarchy and memory hierarchy, inspired by Sanders and Kandrot (2011).

b.1.1 *Thread Group Hierarchy*

The thread group hierarchy is illustrated in Fig. B.1. Threads are the smallest units to execute instructions. The hardware groups together threads that execute the same instructions into warps, which are the smallest units scheduled to run on the SMs. Warps typically consist of 32 threads. The warps form blocks, which form grids. Multiple blocks can be scheduled to run on a SM. Blocks are required to run independently from other blocks because block to block communication is not available.

The thread group hierarchy is demonstrated in the following program, `helloWorld.cu`, that prints 20 "hello"s and 20 "world"s.

```
1 // helloWorld.cu
```

```
2  #include "cuda_runtime.h"
3  #include "device_launch_parameters.h"
4  #include <stdio.h>
5  #define BLOCK_SIZE 10
6  __global__ void hello(){
7          float localVar = 10.0;
8          __shared__ float sharedVar;
9          __shared__ float sharedArr[BLOCK_SIZE];
10         printf("hello\n");
11 }
12 int main(){
13         float *globalVar;
14         cudaMalloc((void**)&globalVar,100*sizeof(float));
15         hello << < 2, 10 >> >();
16         cudaDeviceSynchronize();
17         for (int i = 0; i < 20; i++){
18                 printf("world\n");
19         }
20         return 0;
21 }
```

The number of blocks and threads are specified by the numbers inside the triple brackets. The `hello` kernel (equivalent to functions in C and subroutines in Fortran) is launched with 2 blocks each containing 10 threads. Each of the 20 threads execute the print statement for "hello" in line 10. To process 2D data arrays, 2D thread and block sizes are useful. Two-dimensional sizes can be specified using `dim3`. Within a kernel, a thread can identify its thread and block indices using built-in variables, such as `threadIdx.x`, `threadIdx.y`, `blockIdx.x`, `blockDim.x`, and `gridDim.x`. The hardware typically limits the number of blocks to $65,535$ in each direction and a total of $1,024$ threads per block.

In CUDA terminology, the CPU is referred to as the host and the GPU as the device. Functions run on the host, and kernels run on the device. Kernels are distinguished from

functions by `__global__` in front of kernel definitions. The `global` keyword indicates that the kernel can be launched from both the host and device. When it is replaced by `device`, the kernel can only be launched from the device similarly to how functions are called on a CPU (without specifiying the number of blocks and threads). To launch a kernel with multiple threads for recursion or fine-grained parallelism, dynamic parallelism can be used at the cost of significant overhead.

b.1.2 *Memory Types*

The host and device memories are separate. The built-in function, `cudaMemcpy`, is used to copy data from the host memory to device memory, and vice versa. The data transfer between the host and device is time-consuming. While the use of unified memory, released in 2014, eliminates the need to explicitly call `cudaMemcpy`, the data transfer done implicitly behind the scenes is the same.

Corresponding to the thread group hierarchy, there are three types of device memories: local, shared, and global, as illustrated in Fig. B.1. A variable that resides in local memory, such as `localVar` in line 7 of `helloWorld.cu`, is accessible only to the thread that created it. Local variables can also be declared using `malloc` inside a kernel. A variable that resides in global memory, such as `globalVar` in line 13 of `helloWorld.cu`, is allocated by `cudaMalloc` in `main()` and accessible to all threads in all kernels.

A variable that resides in shared memory, such as `sharedArr` and `sharedVar` in lines 8 and 9 of `helloWorld.cu`, is accessible to threads in the block of the shared memory declaration. Access to shared memory is 100x faster than to global memory. Copying global variables to shared memory before computations can reduce the computation time.

A variable that resides in constant memory remains constant during kernel execution. Constant variables are declared by the keyword `__constant__` outside of `main` and kernels. Data from the host can be copied to constant memory by calling `cudaMemcpyToSymbol`. Since the performance for most CUDA programs is limited by the memory bandwidth

rather than computation speed, the use of constant memory, which is cached, can potentially speed up the program by reducing the memory traffic.

Each GPU has multiple SMs. The L1 cache, typically $16 - 48$ kB, for each SM is partitioned into local memory and shared memory. Variables that are declared as shared are pushed to the L2 cache when shared memory runs out. The L2 cache, $512 - 768$ kB, and GPU memory, 8 GB for GTX-1080, are used for global memory.

b.1.3 *Streams*

A stream in a CUDA program is a sequence of operations that occur in the order the operations are issued. Using multiple streams allow kernels to execute concurrently, and thus enhance the GPU utilization. Order across operations in different streams is not guaranteed. As mentioned in Sec. B.1.2, the data transfer between the host and device is computationally expensive. Streams can be employed to hide the latency from data transfer by issuing memory transfers on one stream via `cudaMemcpyAsync` and kernel execution on another. Because there is no guarantee in the order of memory transfer and kernel execution, one should check that the memory transferred is not modified in the kernel on a different stream.

Kernel chunkification can be implemented to potentially increase the kernel concurrency. A call to a kernel is chunkified or broken into multiple calls to the same kernel. Programmers can provide offsets to the kernels to ensure that the right blocks of data are accessed. The kernel calls are launched on different streams and executed in parallel when there is sufficient computational resource.

b.1.4 *Synchronization*

Multiple levels of asynchronous execution exist in a CUDA program. Instructions on the CPU and GPU are executed asynchronously. To synchronize the CPU and GPU, `cudaDeviceSynchronize` can be called, such as in line 16 of `helloWorld.cu`, to block CPU

execution until the GPU finishes printing "hello". If line 16 is removed, the CPU simply launches the `hello` kernel and immediately moves on to print "world". Without synchronization, the order of "world" and "hello" is not guaranteed. Some built-in functions, such as `cudaMemcpy`, implicitly block the host.

When multiple streams are employed, the call to `cudaDeviceSynchronize` also synchronizes across streams. To block the execution of the host until individual streams complete execution, `cudaStreamSynchronize` can be called instead. Threads in a block also execute asynchronously. For synchronization within a block, `__syncthreads` can be called within kernels.

b.1.5 *Atomic Operations*

Multiple threads might modify the data at same memory location during kernel execution. Race conditions, as illustrated in Fig. B.2, can occur. Modification of a variable is broken into three steps: read in data from memory, compute new value, and write to memory. A variable `t` starts with value of 2. Each of the two threads execute `t=t+1`. In case 1, thread 2 accesses the value that thread 1 modified. In case 2, both threads access the original value of 2. Two final answers for `t` are possible depending on the timing of the three steps.

Variable t starts at 2, each thread execute t = t + 1

Case	Thread id	Read	Compute	Write	Final answer
1	1	2	3	3	4
	2	3	4	4	
2	1	2	3	3	3
	2	2	3	3	

Figure B.2: Example of race condition.

Atomic operations are employed to avoid race conditions by only granting access to specified memory locations to 1 thread. Other threads that modify the same variable wait in queue. An example of atomic operation is provided in Sec. B.2. Care should be taken to ensure that collision of requests to modify the same variable is minimal. When a large number of threads atomically modify the same variable, the wait time can slow down the program.

b.1.6 *Debugging and Profiling*

The CUDA debugger in Visual Studio via Nsight and command line tools, `cuda-gdb` and `cuda-memcheck`, can be employed for debugging. Race conditions can be explicitly checked. Calling `cudaDeviceSynchronize` after every kernel is recommended for trouble shooting.

To profile the performance of CUDA programs, Nsight and the command line tool, `nvprof`, can be employed. The outputs include the amount of time a program spends in each kernel and automated analysis for ways to improve individual kernels. Timelines to visualize executions done on each stream to check for kernel concurrency can also be generated.

b.1.7 *Overview of Example*

The rest of the appendix includes an example of selecting the neighbor detection algorithms for a CUDA program for simulating Brownian rigid spherocylinders. The conventional neighbor list and cell list algorithms are parallelized. The program with the neighbor list and constant memory has the lowest simulation time for three out of four initial configurations tested.

In addition to algorithmic improvements, the random number generation via `curand` and `thrust` are compared and found to be similar. To further improve the performance, `nvprof` outputs are generated. To address low kernel concurrency, the top two most

time consuming kernels are chunkified and launched in different streams. To investigate the effect of the number of available SMs, the results are generated using the GTX-1080 and Titan RTX GPUs. By combining repetitive computations and incorporating other instructional parallelism techniques, the main kernel responsible for low global memory load efficiency is shifted from a user written kernel to a cuda library call.

b.2 Methods

The two conventional neighbor detection algorithms, described in Sec. 2.15.2, were parallelized to run on the GPU. Both the neighbor (Verlet) list and cell list were stored in a 2D array `potCon[nfib, npcn]`, where `nfib` is the number of fibers, and `npcn` is maximum number of potential contacts (size of a list) for any fiber. The number of fibers in a list is maintained by `potConSize[nfib]`. The programs that employ the cell list and neighbor list (NL) are denoted as the baseline and NL programs, respectively. The baseline program regenerates the cell list every time step while the NL program regenerates the neighbor list every `neighbor_check` steps.

The list, `potCon`, was subsequently passed to a kernel, `contact`, which calculates the separations between fibers and those in the lists. The `contact` kernel was launched with `nfib` blocks and `nThreads` each.

```
1  contact <<< nfib, nThreads >>> (potCon, potConSize, contact_cutoff, ...);
```

where `nThreads` was optimized and set to 64 for both programs.

Interfiber forces and torques were only calculated for fiber pairs with separation smaller than the contact cutoff. The forces and torques were atomically added (see Sec. B.1.5) to global variables.

```
1  atomicAdd (pointer to global variable, calculations);
```

Results from calculations, such as multiplying the force by the moment arm, can be stored in a local variable before `atomicAdd`.

```
float dummy = calculations;
atomicAdd (pointer to global variable, dummy);
```

The neighbor detection algorithms used to generate the neighbor and cell lists, along with other steps taken for optimization, are described below. Only parts of the code that were modified to test different algorithms and techniques are highlighted. For the complete code, refer to Appendix C.

B.2.1 *Parallel Cell List Implementation*

Similar to the serial counterpart, the parallel cell list algorithm divides the simulation box into cubic bins with sides chosen based on the contact cutoff. Fibers were assigned to the bins based on positions. For example, the x index of the bin for a fiber with position `rx` is obtained by

```
xbin = int(floorf(rx/dx)) + nx/2
```

where `xbin` is the x index of the bin, `dx` is the bin side length, and `nx` is the number of bins in the x direction. Since a loop through the fibers is required, the assignment is an $\mathcal{O}(\texttt{nfib})$ operation for a serial program. In a CUDA program, a kernel can be launched with `nfib` total threads and reduce the assignment to $\mathcal{O}(1)$ time complexity in parallel.

The cell list for a fiber only contains fibers from neighboring bins with larger indices and center of mass separation within $2r_p + 2$, where r_p is the aspect ratio. This cutoff value was chosen to first eliminate a fraction of fibers from consideration by the `contact` kernel, which calculates the separation at the point of contact and is more computatinally intensive. A block cannot terminate until all threads complete computations. Thus, despite some repetitive code, having smaller sets of fibers in the `contact` kernel reduces the computational cost. The cell list for a fiber was obtained by running the user-written kernel, `link`, with `nfib` blocks, 125 threads each,

```
link <<< nfib, 125 >>> (...);
```

As described in Sec. 2.15.2, a fiber and fibers in $5 \times 5 \times 5 = 125$ neighboring bins were considered for potential fiber contacts. The list is then sent to the `contact` kernel.

Because only the center of mass separations were considered for fibers to be placed in `potCon`, the value for `npcn` is large. The large memory requirement limits the system size. The system size was limited by memory rather than the number of blocks a kernel can launch.

b.2.2 *Parallel Neighbor List Implementation*

The neighbor list for each fiber contains all other fibers within `neighbor_cutoff`, chosen based on `neighbor_check`, from itself. The neighbor list for a fiber only contains fibers with indices greater than itself to eliminate repetitive computations. To generate the neighbor list, a user-written kernel, `neighbor_list`, was launched with `nBlocks` blocks, 128 threads each,

```
1 dim3 nBlocks (A, nfib/A);
2 neighbor_list <<< nBlocks, 128 >>> (...);
```

where A is a number set to 32 in this study. Fibers were atomically added to the neighbor list, which was sent downstream to the `contact` kernel. The neighbor list and cell list were compared for validation.

b.2.3 *Constant Memory*

More than 60 of the same variables from the baseline and NL programs were declared as constant memory in a separate file `consts.cuh`. For the compilation to be successful, relocatable memory is required. For compilation through `CMake`, this is achieved by adding the following line to `CMakeLists.txt`,

```
1 set(CMAKE_CUDA_FLAGS ${CMAKE_CUDA_FLAGS} "-rdc=true")
```

B.2.4 *Kernel Chunkification*

Based on performance discussed below, the NL program with constant memory was selected to experiment with kernel chunkification. The calls to the most time consuming kernels, `neighbor_list` and `contact`, were chunkified into `nSm` calls. Each chunk was launched on a different stream, stored in `streamArr`. For example,

```
int nSm = 10;
cudaStream_t *streamArr = (cudaStream_t*)malloc(nSm*sizeof(cudaStream_t));
for (int i=0; i<nSm; i++){
        cudaStreamCreate(&streamArr[i]);
}
```

An offset, which depends on the dataset and method, was provided to the kernel calls to use the correct chunk of data. For kernel chunkification studies, the number of threads for the kernels was chosen to be 32.

```
for (int i=0; i<nSm; i++){
        offset = 8*nfib/nSm;
        neighbor_list <<< nfib/nSm, 32, 0, streamArr[i]>>> (..., offset);
}
```

B.2.5 *Random Number Generation with `curand` and `thrust`*

For the baseline and NL programs, random numbers were generated using the `curand` library.

```
curandGenerator_t gen;
float *randN;
cudaMalloc((void**)&randN,N*sizeof(float));
curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
curandSetPseudoRandomGeneratorSeed(gen, random seed);
curandSetStream(gen, stream number);
curandGenerateNormal(gen, randN, N, 0.0, 1.0);
```

Given that all computations in the program were executed on the device, the use of the `thrust` library was not warranted.

In the interest of learning about the library and testing its performance, the call to the `curand` library was replaced by the `thrust` library. Since there is no built-in function in `thrust` to generate an array of random numbers, a functor is written to be used with `thrust::transform`.

```
struct randFunctor{
        __device__ float operator()(int ind){
        thrust::default_random_engine rng;
        thrust::normal_distribution<float> dist(0.f, 1.f);
        rng.discard(ind);
        return dist(rng);
    }
};
```

In the main function, the functor was used with an iterator that was updated every time step.

```
int randNum = rand() % 1000;
thrust::counting_iterator<int> first(randNum);
thrust::counting_iterator<int> last = first + N;
thrust::transform(first, last, randT.begin(), randFunctor());
```

b.3 Results

b.3.1 *Baseline*

The performance of the baseline program, which utilized the parallel cell list program described in Sec. B.2.1, was established with various initial configurations, concentrations, and numbers of fibers, which is a measure of the system size. Fibers were placed into the simulation box either with random positions and orientations, indicated by `isotropic`

in the discussion below, or on a rectangular lattice, indicated by `ordered`. Four sets of initial configurations based on the method of fiber placement and the volume fraction, ϕ, were utilized: `isotropic 5%`, `isotropic 15%`, `ordered 5%`, and `ordered 50%`. The sizes of the simulation boxes were adjusted based on ϕ and the number of fibers, N_{fib}, which ranged from $1,600$ to $32,770$. Simulations were run for 10^5 steps. While suspensions with `ordered` initial configurations become disordered as the simulations progressed, the time spent in simulating ordered structures can still offer insight to the performance of simulating phases with layers, such as the smectic phase in Sec. 1.3.2. Configuration files were not generated to exclude time from file I/O, which is easily hidden by asynchronous execution of the CPU and GPU.

In Fig. B.3, the simulation time, T, per 10^5 time steps for the baseline program are plotted as functions of N_{fib} for the four sets of initial configurations. The simulation time increases as N_{fib} is increased for all four sets. As N_{fib} is increased 20 fold from $1,600$, T increased at most by 15 fold. The parallel baseline program scales better with N_{fib} than serial programs that typically employ the Verlet neighbor list, which has a time complexity of $\mathcal{O}\left(N_{\text{fib}}^2\right)$.

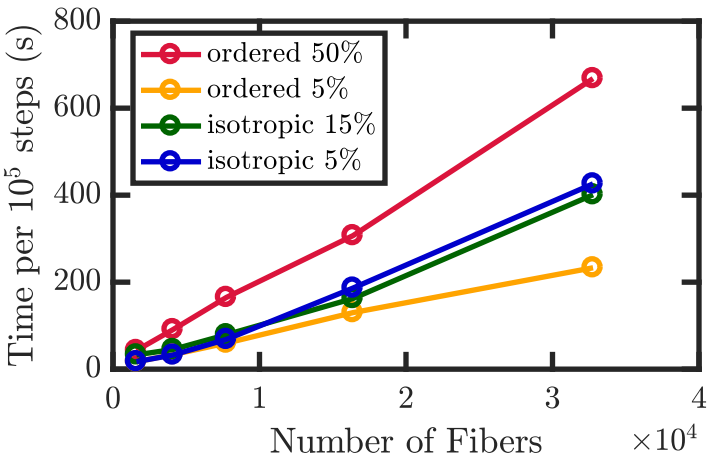


Figure B.3: Simulation time per 10^5 time steps as a function of the number of fibers for the baseline program for four sets of initial configurations.

As described in Sec. B.2.1, the cell list algorithm was parallelized such that each thread

calculated the separation between a fiber and fibers in one of its neighboring bins. As expected, T for `ordered 50%` is the largest for all N_{fib} because more fibers occupy each bin. Since a block cannot terminate until all threads in the block complete computation, the execution time scales with the most number of fibers in bins. The homogeneity of the fiber distribution thus impacts the performance. For `isotropic 5%` and `isotropic 15%`, T is close for all N_{fib} despite having different ϕ. This suggests that the maximum number of fibers in a bin are close. The simulation time for `isotropic 5%` is larger than for `ordered 5%` despite having the same ϕ. Fibers are more evenly distributed for `ordered` cases compared to `isotropic` cases. Bins with more fibers than average control the performance.

B.3.2 *Neighbor List*

The cell lists used in the baseline program were replaced by neighbor lists (NLs). The performance of the NL program was established to compare with that of the baseline program with the same parameter values. Additional large system sizes, goverend by N_{fib}, were simulated because the NL program requires less memory compared to the baseline program. In Fig. B.4, T per 10^5 time steps for the program with neighbor lists (NL) implemented is plotted as a function of N_{fib} for the four sets of initial configurations. Similar to the baseline program, T increases as N_{fib} is increased. As N_{fib} is increased 20 fold from $N_{\text{fib}} = 1,600$, T increases at most by 33 fold. While the scaling is worse than the baseline program, the performance for the NL program is still superior to that for serial programs.

The simulation time does not depend strongly on the initial configurations in contrast to the baseline program. A block of 128 threads is utilized for each fiber to calculate the separation with all other fibers. The number of pairs a thread calculated the separation for was $N_{\text{fib}}/128$. While T of the baseline program depends on the fiber distribution, the performance for the NL program scales similarly with N_{fib} regardless of the initial

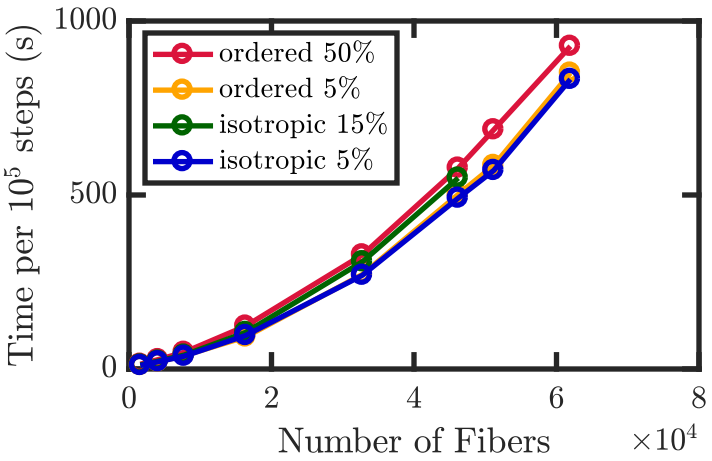


Figure B.4: Simulation time per 10^5 time steps as a function of the number of fibers for the program using neighbor lists for four sets of initial configurations.

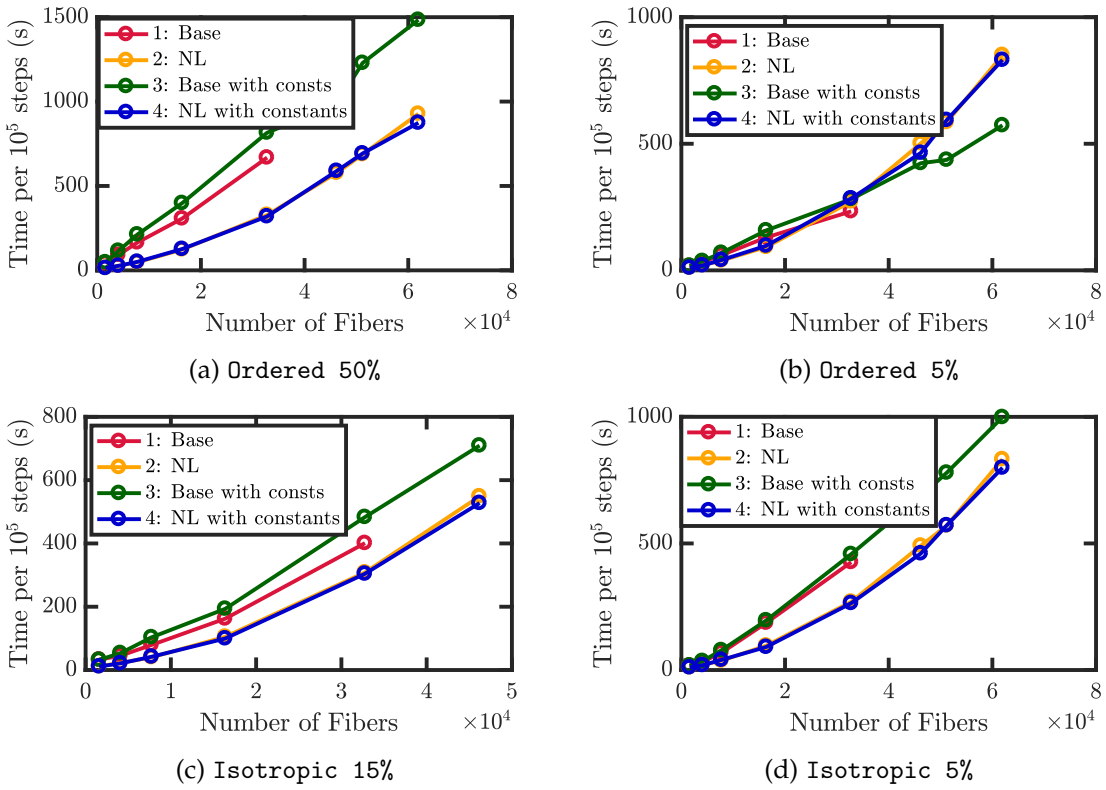
configuration.

B.3.3 *Constant Memory*

To investigate the effect of using constant memory, the performance of the baseline and NL program with and without the use of constant memory was established. In Fig. B.5, T per 10^5 time steps for the four programs is plotted as a function of N_{fib} for the four sets of initial configurations. For both the baseline and NL program, the use of constant memory does not change the performance appreciably. The simulation time increases slightly with incorporation of constant memory for the baseline program with initial configurations of higher concentrations, `ordered 50%` and `isotropic 15%`. Except for `ordered 5%`, the NL program with constant memory is faster than the baseline program and thus chosen to test other optimization techniques.

B.3.4 *Kernel Chunkification and GPU Type*

To investigate the effect of kernel chunkification and GPU type, the performance of the NL program with constant memory and the `isotropic 5%` initial configuration was es-



(a) `Ordered 50%` (b) `Ordered 5%`

(c) `Isotropic 15%` (d) `Isotropic 5%`

Figure B.5: Simulation time per 10^5 time steps as a function of the number of fibers for four sets of initial configurations: `ordered 50%`, `ordered 5%`, `isotropic 15%`, and `isotropic 5%`. Case 1 is the baseline program; case 2 is the NL program; case 3 is the baseline program with constant memory; case 4 is the NL program with constant memory.

tablished. In Fig. B.6, T per 100 time steps on the GTX-1080 and Titan RTX GPU for $N_{\text{fib}} = 1,600$ and $46,208$ are plotted as a function of the number of chunks, N_{chunk}, a kernel is divided into. For $N_{\text{fib}} = 1,600$, T increases as N_{chunk} is increased; for $N_{\text{fib}} = 46,208$, T is relatively constant. This suggests that, without chunkification, each kernel launches enough blocks to occupy all of the SMs, even for `TitanRTX`, which has 72 SMs. Launching extra kernels only adds to the overhead, which is a larger chunk of the execution time for smaller systems. For both system sizes, programs are faster when run on Titan RTX than GTX-1080.

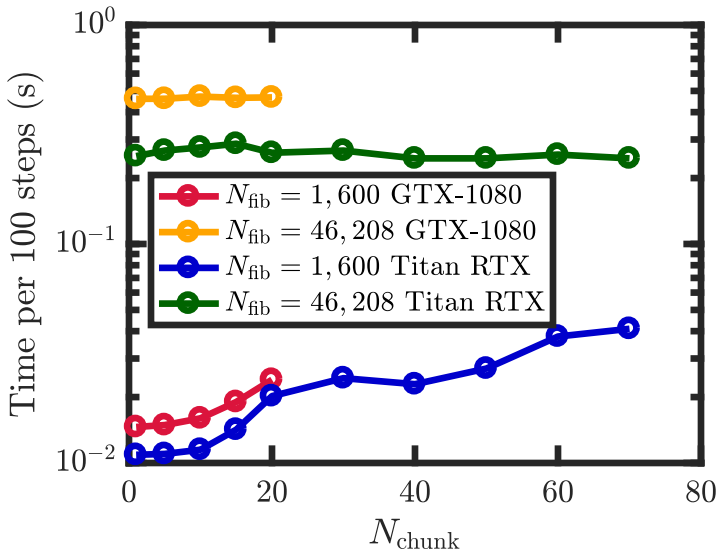


Figure B.6: Simulation time per 100 time steps as a function of the number of chunks a kernel is divided into using the NL program with constant memory for two system sizes: $N_{\text{fib}} = 1,600$ and $46,208$. Simulations were run on two types of GPUs: GTX-1080 and Titan RTX.

b.3.5 *Random Number Generation via* `curand` *and* `thrust`

The performance of the NL program with constant memory using the `curand` and `thrust` libraries for random number generation was investigated. In Fig. B.7, T per 10^5 steps is plotted as a function of N_{fib} for the `isotropic 15%` initial configuration. Replacing the `curand` random number generator by calls to the `thrust` library has no impact on the performance. One can choose the library based on convenience without impacting the performance.

b.3.6 *Other Optimizations*

An `nvprof` report was generated for the NL program with constant memory that ran for 50 simulation steps. The outputs, summarized in Tab. B.1, suggests that memory loading and storage is inefficient. Both kernel concurrency and compute utilization are at 0% for the `nvprof` output; this is consistent with the finding of kernel chunkification.

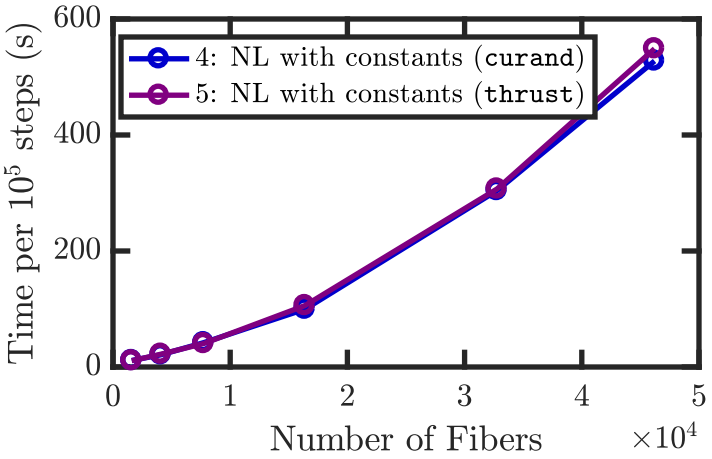


Figure B.7: Simulation time per 10^5 time steps as a function of the number of fibers for the NL program with constant memory. For random number generation, case 4 employs the `curand` library while case 5 employs the `thrust` library.

Two steps were taken to further optimize the program. First, computation results were stored in local variables before atomic addition (see the beginning of Sec. B.2). Second, repetitive variable assignments were moved outside of a while loop. The `nvprof` output for the optimized program is indicated by the Opt case in Tab. B.1. The kernel responsible for low global memory load efficiency shifted from the user-written kernel, `contact`, to `curandGenerateNormal`, which is a library call.

Table B.1: `nvprof` output for the performance as various speedup measures, were taken.

Cases	`nvprof` output category	`nvprof` output	kernel responsible
Initial Case	Low kernel concurrency	0%	
	Low compute utilization	0%	
	Low global memory load efficiency	26.9% avg	`contact`
	Low global memory store efficiency	64.3% avg	`curandGenerateNormal`
	Low shared memory efficiency	4.3% avg	`contact`
	Low warp execution efficiency	41.9% avg	`contact`
Opt	Low kernel concurrency	0%	
	Low compute utilization	0%	
	Low global memory load efficiency	31.5% avg	`curandGenerateNormal`
	Low global memory store efficiency	64.3% avg	`curandGenerateNormal`
	Low shared memory efficiency	4% avg	`contact`
	Low warp execution efficiency	43.2% avg	`contact`

More variables were assigned to shared memory to further optimize the performance.

However, the simulation time increases, suggesting that the shared memory had been exhausted and the assignments overflowed to global memory. For kernels that slowed down due to additional shared memory, a second pass was conducted to remove variables that appear less frequently from the shared memory.

b.4 Conclusions and Future Directions

A review of CUDA concepts was presented, along with the procedure and results for selecting a neighbor detection algorithm and optimizing the performance of the CUDA program for Brownian dynamics simulations of rigid spherocylinders. The sub-quadratic scaling of the simulation time with the number of fibers in the suspension for the baseline program and the NL program is superior to typical serial programs. Adding 60 variables to constant memory has minimal impact on the performance. For most simulations, the NL program with constant memory outperforms the baseline program, and was thus chosen for further optimization.

For the NL program with constant memory, kernel concurrency via chunkification and `cudastreams` was not observed because the system sizes were sufficiently large that all SMs were occupied without chunkification. The simulation times from running programs using the random number generator from the `curand` and `thrust` libraries were equivalent. Instruction level parallelism was developed based on outputs from `nvprof`. An optimized program, for which the kernel responsible for low global memory load efficiency became a library call instead of user-defined kernel, was obtained. From testing the usage of shared memory, only variables that are used at least twice are recommended to be declared as shared memory so as to not exhaust the available amount of memory.

More tests to examine the effect of splitting the L1 cache differently for more allotment of shared memory can be tested. Future directions include exploring the use of texture memory and optimizing shared memory usage by distributing the L1 cache differently. Since the programs were written for a physical system, most vectors were stored as three

arrays, one for each of the x, y, and z components. One can experiment with texture memory for potential speedups by combining the three arrays into a two-dimensional array.

C

SIMULATION CODES

Simulation codes that are written in CUDA for simulating suspensions of Brownian rigid fibers and non-Brownian flexible fibers are provided in this appendix. For both programs, the function `main()` is contained in `main.cu`, which calls the kernels described below. Header files and `CMakeLists.txt` that are used to compile the programs are also included.

c.1 Brownian Fiber Code

`main.cu` - This file contains the `main()` function and `orientCalc`, which calculates the order parameter of the suspension on the host.

`contact.cu` - This file contains the `contact` kernel, which calculates the separation between fibers in the neighbor list and updates the normal forces. The file also contains `parallelSepCon`, which calculates the separation between parallel fibers.

`hydroRes.cu` - This file contains the `hydroRes` kernel, which calculates the hydrodynamic resistance tensors every time step.

`initialize.cu` - This file contains the `initialize` kernel, which initializes variables, such as rotation matrices, from input files.

`less_edwards.cu` - This file contains the `less_edwards` kernel, which updates variables for the Lees-Ewards boundary condition.

`neighbor_list.cu` - This file contains the `neighbor_list` kernel, which generates the neighbor list every `neighbor_check` steps. The file also contains `parallelSep`, which calculates the separation between parallel fibers.

`setVarArray.cu` - This file contains the `setVarArray` kernel, which stores pointers to device variables into arrays of double pointers.

`stress.cu` - This file contains the `stress` kernel, which calculates the stress of the suspension every `stress_write` steps.

`zeroVar.cu` - This file contains the `zeroVar` kernel, which zeroes variables every time step.

`updateBod.cu` - This file contains the `updateBod` kernel, which updates the body forces.

`updatePos.cu` - This file contains the `updatePos` kernel, which updates the fiber positions and orientations.

`updateVel.cu` - This file contains the `updateVel` kernel, which sums up forces and torques, and updates the linear and angular velocities.

`Parameters.in` - This is an input file that contains parameters of the simulation run.

`Centers_of_Mass.in` - This is an input file that contains the centers of mass of the fibers.

`Euler_Parameters.in` - This is an input file that contains the Euler parameters of the fiber segments.

`Physical_Parameters.in` - This is an input file that contains parameters used to calculate the magnitude of the Brownian force and torque.

main.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cmath>
#include <time.h>
#include <cuda_runtime_api.h>
#include <curand.h>
// user-written header files
#include "setVarArray.h"
#include "initialize.h"
#include "zeroVar.h"
#include "hydroRes.h"
#include "updateVel.h"
#include "updatePos.h"
#include "updateBod.h"
#include "lees_edwards.h"
#include "contact.h"
#include "stress.h"
#include "neighbor_list.h"
#include "consts.cuh"
#define pi 3.14159265
using namespace std;
// declare variables constant
__device__ __constant__ int nfib,npcn,fac;
__device__ __constant__ float E11,E12,E13,E22,E23,E33;
__device__ __constant__ float duxdx,duxdy,duxdz;
__device__ __constant__ float duydx,duydy,duydz;
__device__ __constant__ float duzdx,duzdy,duzdz;
__device__ __constant__ float dx,dy,dz,dt,elf;
__device__ __constant__ float C0,C1,C2,C3,C4,C5;
__device__ __constant__ float C6,C7,C8,C9,C10;
__device__ __constant__ float Omega_x,Omega_y,Omega_z;
__device__ __constant__ float contact_cutoff,over_cut,rep_cutoff;
__device__ __constant__ float rp,sidex,sidey,sidez;
__device__ __constant__ float fstar,fact,Astar,decatt;
// error checking function
#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) {
      getchar();
      exit(code);
    }
  }
}
// calculates order parameter on host
void orientCalc(float *px,float *py,float *pz,int h_nfib,float *P2);
// main body
int main(void) {
  ////////////////////////////////////////////////////////
  const int numVar = 198;
const int numIntVar = 36;
int h_npcn = 2000;
float neighbor_cutoff = 10.0*10.0;
int neighbor_check = 50;
// numVar-maximum number of float variables on device
// numIntVar-maximum number of integer variables on device
// h_npcn-number of potential contacts
// neighbor_cutoff-cutoff separation for neighbor list
// neighbor_check-how often neighbor list is generated
float **var;
int **intVar;
gpuErrchk(cudaMalloc((void**)&var,numVar*sizeof(float*)));
gpuErrchk(cudaMalloc((void**)&intVar,numIntVar*sizeof(int*)));
gpuErrchk(cudaPeekAtLastError());
////////////////////////////////////////////////////////
// input and output files
FILE *Parameters,*Physical_Parameters;
FILE *Centers_of_Mass,*Euler_Parameters;
FILE *resist_func,*README,*center_mass;
FILE *Stress_tensor;
FILE *q0file,*q1file,*q2file,*q3file;
FILE *pxfile,*pyfile,*pzfile;
FILE *rxfile,*ryfile,*rzfile;
FILE *wxfile,*wyfile,*wzfile;
FILE *uxfile,*uyfile,*uzfile;
FILE *orientfile;
// open files
Parameters = fopen("Parameters.in","r");
Centers_of_Mass = fopen("Centers_of_Mass.in","r");
Euler_Parameters = fopen("Euler_Parameters.in","r");
Physical_Parameters = fopen("Physical_Parameters.in","r");
README = fopen("README.txt","w");
orientfile = fopen("orientation.txt","w");
resist_func = fopen("resist_func.txt","w");
Stress_tensor = fopen("Stress_tensor.txt","w");
center_mass = fopen("center_mass.txt","wb");
pxfile = fopen("px.txt","wb");
pyfile = fopen("py.txt","wb");
pzfile = fopen("pz.txt","wb");
rxfile = fopen("rx.txt","wb");
ryfile = fopen("ry.txt","wb");
rzfile = fopen("rz.txt","wb");
q0file = fopen("q0.txt","wb");
q1file = fopen("q1.txt","wb");
q2file = fopen("q2.txt","wb");
q3file = fopen("q3.txt","wb");
wxfile = fopen("wx.txt","wb");
wyfile = fopen("wy.txt","wb");
wzfile = fopen("wz.txt","wb");
uxfile = fopen("ux.txt","wb");
uyfile = fopen("uy.txt","wb");
uzfile = fopen("uz.txt","wb");
```

```
////////////////////////////////////////////////////
// Read parameters
// variables on the host,indicated by h_
int h_nfib,nseg,config_write;
int stress_write,randSeed,orient_write;
int h_fac,fiberPerBlock,contact_write;
float h_rp,h_contact_cutoff,h_rep_cutoff,h_over_cut;
float h_dt,strain,h_sidex,h_sidey,h_sidez;
float h_fstar,h_fact,h_Astar,h_decatt,h_elf,delta_rx;
float h_duxdx,h_duydx,h_duzdx,h_duxdy,h_duydy;
float h_duzdy,h_duxdz,h_duydz,h_duzdz;
float fraction_rp;
// nfib-number of fibers
// nseg-number of segments
// config_write-number of time steps between configuration writes
// fac-factor of fluid velocity
// fiberPerBlock-number of fiber per block
// rp-segment aspect ratio
// contact_cutoff-cutoff for contacts
// rep_cutoff-cutoff distance for calculating repulsive forces
// over_cut-limit to the overlapping of two fibers
// dt,strain-time step,total time of run
// sidex,sidey,sidez-simulation box dimensions (x and y directions)
// fstar,fact-force prefactor,force expfonential factor (decay length
    )
// Astar-Prefactor of the attractive force
// decatt-decay length of the attractive force
// elf-electric field factor
// delta_rx-shift variable for sliding periodic images
// duxdx...-velocity gradient tensor
// fraction_rp-effective aspect ratio factor
// Read in Parameters.in //
fscanf(Parameters,"%d",&h_nfib);
fscanf(Parameters,"%*[^\n]%d",&nseg);
fscanf(Parameters,"%*[^\n]%f",&h_rp);
fscanf(Parameters,"%*[^\n]%f",&h_contact_cutoff);
fscanf(Parameters,"%*[^\n]%f",&h_rep_cutoff);
fscanf(Parameters,"%*[^\n]%f",&h_over_cut);
fscanf(Parameters,"%*[^\n]%f",&h_dt);
fscanf(Parameters,"%*[^\n]%f",&strain);
fscanf(Parameters,"%*[^\n]%f",&h_sidex);
fscanf(Parameters," %f",&h_sidey);
fscanf(Parameters," %f",&h_sidez);
fscanf(Parameters,"%*[^\n]%f",&fraction_rp);
fscanf(Parameters,"%*[^\n]%d",&config_write);
fscanf(Parameters,"%*[^\n]%d",&contact_write);
fscanf(Parameters,"%*[^\n]%f",&h_fstar);
fscanf(Parameters,"%*[^\n]%f",&h_fact);
fscanf(Parameters,"%*[^\n]%f",&h_Astar);
fscanf(Parameters,"%*[^\n]%f",&h_decatt);
fscanf(Parameters,"%*[^\n]%f",&delta_rx);
fscanf(Parameters,"%*[^\n]%f",&h_duxdx);
fscanf(Parameters,"%*[^\n]%f",&h_duydx);
fscanf(Parameters,"%*[^\n]%f",&h_duzdx);
fscanf(Parameters,"%*[^\n]%f",&h_duxdy);
fscanf(Parameters,"%*[^\n]%f",&h_duydy);
fscanf(Parameters,"%*[^\n]%f",&h_duzdy);
fscanf(Parameters,"%*[^\n]%f",&h_duxdz);
fscanf(Parameters,"%*[^\n]%f",&h_duydz);
fscanf(Parameters,"%*[^\n]%f",&h_duzdz);
fscanf(Parameters,"%*[^\n]%d",&h_fac);
fscanf(Parameters,"%*[^\n]%f",&h_elf);
fscanf(Parameters,"%*[^\n]%d",&fiberPerBlock);
fscanf(Parameters,"%*[^\n]%d",&stress_write);
fscanf(Parameters,"%*[^\n]%d",&randSeed);
fscanf(Parameters,"%*[^\n]%d",&orient_write);
float h_ts,h_T,h_b,h_eta0;
fscanf(Physical_Parameters,"%f",&h_ts);
fscanf(Physical_Parameters,"%*[^\n]%f",&h_T);
fscanf(Physical_Parameters,"%*[^\n]%f",&h_b);
fscanf(Physical_Parameters,"%*[^\n]%f",&h_eta0);
// ts-time scale
// T-temperature
// b-radius
// eta0-suspending fluid viscosity
////////////////////////////////////////////////////
// declare variables on host and device
// variables on host
float *rcmx,*rcmy,*rcmz,*rx,*ry,*rz;
float *q0,*q1,*q2,*q3,*px,*py,*pz;
float *ux,*uy,*uz,*wx,*wy,*wz;
cudaMallocHost((void**)&rcmx,h_nfib*sizeof(float));
cudaMallocHost((void**)&rcmy,h_nfib*sizeof(float));
cudaMallocHost((void**)&rcmz,h_nfib*sizeof(float));
cudaMallocHost((void**)&rx,h_nfib*nseg*sizeof(float));
cudaMallocHost((void**)&ry,h_nfib*nseg*sizeof(float));
cudaMallocHost((void**)&rz,h_nfib*nseg*sizeof(float));
cudaMallocHost((void**)&q0,h_nfib*nseg*sizeof(float));
cudaMallocHost((void**)&q1,h_nfib*nseg*sizeof(float));
cudaMallocHost((void**)&q2,h_nfib*nseg*sizeof(float));
cudaMallocHost((void**)&q3,h_nfib*nseg*sizeof(float));
cudaMallocHost((void**)&px,h_nfib*nseg*sizeof(float));
cudaMallocHost((void**)&py,h_nfib*nseg*sizeof(float));
cudaMallocHost((void**)&pz,h_nfib*nseg*sizeof(float));
cudaMallocHost((void**)&ux,h_nfib*nseg*sizeof(float));
cudaMallocHost((void**)&uy,h_nfib*nseg*sizeof(float));
cudaMallocHost((void**)&uz,h_nfib*nseg*sizeof(float));
cudaMallocHost((void**)&wx,h_nfib*nseg*sizeof(float));
cudaMallocHost((void**)&wy,h_nfib*nseg*sizeof(float));
cudaMallocHost((void**)&wz,h_nfib*nseg*sizeof(float));
float h_C0,h_C1,h_C2,h_C3,h_C4,h_C5;
float h_C6,h_C7,h_C8,h_C9,h_C10;
float h_Omega_x,h_Omega_y,h_Omega_z;
float h_E11,h_E12,h_E13,h_E22,h_E23,h_E33;
float Xa,Ya,Xc,Yc,Yh,re,ecc;
float nL3,volfrac,consistency;
```

```
int time_steps;
// re-effective segment aspect ratio
// ecc-eccentricity
// time_steps-number of time steps
// nL3-dimensionless concentration ("n-L-cubed")
// volfrac-volume fraction
// consist-consistency = volfrac/2.6
int *d_nseg;
float *d_Xa,*d_Xc,*d_delta_rx;
cudaMalloc((void**)&d_nseg,sizeof(int));
cudaMalloc((void**)&d_Xa,sizeof(float));
cudaMalloc((void**)&d_Xc,sizeof(float));
cudaMalloc((void**)&d_delta_rx,sizeof(float));
// floating point variables on device
float *d_rcmx,*d_rcmy,*d_rcmz,*d_rx,*d_ry,*d_rz;
float *d_q0,*d_q1,*d_q2,*d_q3;
float *d_q0dot,*d_q1dot,*d_q2dot,*d_q3dot;
float *d_qe0,*d_qe1,*d_qe2,*d_qe3;
float *d_px,*d_py,*d_pz;
float *d_ucmx,*d_ucmy,*d_ucmz;
float *d_ucox,*d_ucoy,*d_ucoz;
float *d_ux,*d_uy,*d_uz;
float *d_uxfl,*d_uyfl,*d_uzfl;
float *d_wx,*d_wy,*d_wz;
// rcmx,rcmy,rcmz-fiber center of mass
// rx,ry,rz-segment centers
// q0,q1,q2,q3-segment euler parameters
// q0dot...-time derivatives of Euler Parameters
// qe1,...-time derivatives of Euler Parameters from previous time
// px,py,pz-segment orientational vectors
// ucmx,ucmy,ucmz-fiber center of mass velocity
// ucox,ucoy,ucoz- fiber center of mass velocity from previous time
// ux,uy,uz-segment velocity
// uxfl,uyfl,uzfl-ambient flow field velocity
// wx,wy,wz-segment angular velocity
cudaMalloc((void**)&d_rcmx,h_nfib*sizeof(float));
cudaMalloc((void**)&d_rcmy,h_nfib*sizeof(float));
cudaMalloc((void**)&d_rcmz,h_nfib*sizeof(float));
cudaMalloc((void**)&d_rx,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_ry,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_rz,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_q0,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_q1,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_q2,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_q3,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_q0dot,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_q1dot,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_q2dot,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_q3dot,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_qe0,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_qe1,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_qe2,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_qe3,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_px,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_py,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_pz,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_ucmx,h_nfib*sizeof(float));
cudaMalloc((void**)&d_ucmy,h_nfib*sizeof(float));
cudaMalloc((void**)&d_ucmz,h_nfib*sizeof(float));
cudaMalloc((void**)&d_ucox,h_nfib*sizeof(float));
cudaMalloc((void**)&d_ucoy,h_nfib*sizeof(float));
cudaMalloc((void**)&d_ucoz,h_nfib*sizeof(float));
cudaMalloc((void**)&d_ux,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_uy,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_uz,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_uxfl,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_uyfl,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_uzfl,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_wx,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_wy,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_wz,h_nfib*nseg*sizeof(float));
float *d_R11,*d_R12,*d_R13,*d_R21,*d_R22,*d_R23;
float *d_fcx,*d_fcy,*d_fcz,*d_tcx,*d_tcy,*d_tcz;
float *d_fbx,*d_fby,*d_fbz,*d_tbx,*d_tby,*d_tbz;
float *d_frx,*d_fry,*d_frz;
// R11,R12,...-segment rotation matrix
// fbx,fby,fbz-segment body force (i.e. gravity,etc.)
// tbx,tby,tbz-segment body torque
// fcx,fcy,fcz-segment colloidal repulsive force
// tcx,tcy,tcz-segment colloidal repulsive torque
cudaMalloc((void**)&d_R11,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_R12,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_R13,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_R21,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_R22,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_R23,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_fcx,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_fcy,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_fcz,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_tcx,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_tcy,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_tcz,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_fbx,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_fby,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_fbz,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_tbx,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_tby,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_tbz,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_frx,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_fry,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_frz,h_nfib*nseg*sizeof(float));
float *d_D1,*d_D2,*d_D3;
float *d_A11,*d_A12,*d_A13,*d_A23,*d_A22,*d_A33;
float *d_C11,*d_C12,*d_C13,*d_C23,*d_C22,*d_C33;
// D1,D2,D3-"Jeffery" term in hydrodynamic torque
// A11,...-A inverse term in notes
```

```
// C11,...-C inverse term in notes
// C0,C1...-constant groupings
// Omega_...-ambient flow field angular velocity
cudaMalloc((void**)&d_D1,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_D2,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_D3,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_A11,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_A12,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_A13,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_A23,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_A22,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_A33,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_C11,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_C12,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_C13,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_C23,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_C22,h_nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_C33,h_nfib*nseg*sizeof(float));
float *d_neighb_cutoff,*d_Ya,*d_Yc,*d_Yh;
// E11,E12,...-rate of strain tensor
// Xa,Ya,Xc,Yc,Yh-scalar resistance functions
cudaMalloc((void**)&d_neighb_cutoff,sizeof(float));
cudaMalloc((void**)&d_Ya,sizeof(float));
cudaMalloc((void**)&d_Yc,sizeof(float));
cudaMalloc((void**)&d_Yh,sizeof(float));
float P2 = 0.0;
// P2: order parameter
int *d_step,*potCon,*potConSize;
// step-time step
cudaMalloc((void**)&d_step,sizeof(int));
cudaMalloc((void**)&potCon,h_nfib*nseg*h_npcn*sizeof(int));
cudaMalloc((void**)&potConSize,h_nfib*nseg*sizeof(int));
//////////////////////////////////////////////////////
// generate random number
curandGenerator_t gen;
float *randN;
cudaMalloc((void**)&randN,6*h_nfib*nseg*sizeof(float));
curandCreateGenerator(&gen,CURAND_RNG_PSEUDO_DEFAULT);
curandSetPseudoRandomGeneratorSeed(gen,randSeed);
gpuErrchk(cudaPeekAtLastError());
if (cudaSuccess != cudaGetLastError()){return 1;}
// cuda streams
cudaStream_t stream0,stream1,stream2,stream3,stream4;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaStreamCreate(&stream3);
cudaStreamCreate(&stream4);
// kernel launch parameters
if (h_nfib < fiberPerBlock) fiberPerBlock = h_nfib;
int numThreads = fiberPerBlock*nseg;
int numBlocks = (h_nfib*nseg+numThreads-1)/numThreads;
dim3 nBlocks (numThreads,numBlocks);
```

```
// numThreads-number of threads per block
// numBlocks -number of blocks
// stress
float *d_Stress,*Stress;
cudaMalloc((void**)&d_Stress,6*sizeof(float));
Stress = (float*)calloc(6,sizeof(float));
// temporary variables
int idum1,idum2,step,m,i,mi;
float tprint;
gpuErrchk(cudaPeekAtLastError());
//////////////////////////////////////////////////////
// Read in Centers of Mass,Euler Parameters,Equilibrium Angles //
for (m = 0; m < h_nfib; m++){
  fscanf(Centers_of_Mass,"%d %f %f %f ",
    &idum1,rcmx+m,rcmy+m,rcmz+m);
  for (i = 0; i < nseg; i++){
    mi = m*nseg+i;
    fscanf(Euler_Parameters,"%d %d %f %f %f %f",
      &idum1,&idum2,q0+mi,q1+mi,q2+mi,q3+mi);
  }
}
fclose(Parameters);
fclose(Centers_of_Mass);
fclose(Euler_Parameters);
if (cudaSuccess != cudaGetLastError()){ printf("after reading input
    files!\n"); getchar(); }
//////////////////////////////////////////////////////
// calculate constants
h_elf = 0.0;
// Rate of strain tensor for the flow field
h_E11 = 0.5*(h_duxdx+h_duxdx);
h_E12 = 0.5*(h_duydx+h_duxdy);
h_E13 = 0.5*(h_duzdx+h_duxdz);
h_E22 = 0.5*(h_duydy+h_duydy);
h_E23 = 0.5*(h_duzdy+h_duydz);
h_E33 = 0.5*(h_duzdz+h_duzdz);
// Fluid vorticity vector
h_Omega_x = 0.5*(h_duzdy-h_duydz);
h_Omega_y = 0.5*(h_duxdz-h_duzdx);
h_Omega_z = 0.5*(h_duydx-h_duxdy);
// Resistance functions
re = fraction_rp*h_rp;
ecc = sqrtf(re*re-1.0)/re;
Xa = (8.0*powf(ecc,3.0)/3.0)/(-2.0*ecc+(1.0+ecc*ecc)*logf((1.0+ecc)
    /(1.0-ecc)));
Ya = (16.0*powf(ecc,3.0)/3.0)/(2.0*ecc+(3.0*ecc*ecc-1.0)*logf((1.0+
    ecc)/(1.0-ecc)));
Xc = (4.0*powf(ecc,3.0)/3.0)*(1.0-ecc*ecc)/(2.0*ecc-(1.0-ecc*ecc)*
    logf((1.0+ecc)/(1.0-ecc)));
Yc = (4.0*powf(ecc,3.0)/3.0)*(2.0-ecc*ecc)/(-2.0*ecc+(1.0+ecc*ecc)*
    logf((1.0+ecc)/(1.0-ecc)));
Yh = (4.0*powf(ecc,5.0)/3.0)/(-2.0*ecc+(1.0+ecc*ecc)*logf((1.0+ecc)
    /(1.0-ecc)));
```

```
fprintf(resist_func,"Xa = %.14E\nYa = %.14E\nXc = %.14E\nYc = %.14E\
    nYh = %.14E\n",Xa,Ya,Xc,Yc,Yh);
fclose(resist_func);
// Other constants //
time_steps = int(strain/h_dt); // number of time steps
h_C0 = 3.0/(4.0*h_rp);
h_C1 = 3.0/(4.0*h_rp*h_rp);
h_C2 = 3.0/(4.0*Yc);
h_C3 = 1.0/Xa-1.0/Ya;
h_C4 = 1.0/Xc-1.0/Yc;
h_C5 = 100.0*sqrtf(1.38*h_T*h_ts/(3.0*pi*h_eta0*powf(h_b,3.0)*h_rp*
    h_dt));
h_C6 = 100.0*sqrtf(1.38*h_T*h_ts/(4.0*pi*h_eta0*powf(h_rp*h_b,3.0)*
    h_dt));
h_C7 = sqrtf(Ya);
h_C8 = sqrtf(Yc);
h_C9 = sqrtf(Xa)-sqrtf(Ya);
h_C10 = sqrtf(Xc)-sqrtf(Yc);
h_contact_cutoff = powf((h_contact_cutoff+2.0),2.0);
h_rep_cutoff = powf((h_rep_cutoff+2.0),2.0);
nL3 = float(h_nfib)*powf((float(2.0)*float(nseg)*h_rp),3.0)/(h_sidex*
    h_sidey*h_sidez);
volfrac = float(h_nfib)*(float(nseg)*pi*(float(2.0)*h_rp)+4.0/3.0*pi)
    / (h_sidex*h_sidey*h_sidez);
consistency = volfrac/float(2.6000);
printf("nL3 %10f\nvolfrac %10f\nconsistency %10f\n",nL3,volfrac,
    consistency);
////////////////////////////////////////////////////
// Make README
fprintf(README,"Number of Fibers: %d\n",h_nfib);
fprintf(README,"Number of Segments: %d\n",nseg);
fprintf(README,"Aspect Ratio of a Segment: %.15f\n",h_rp);
fprintf(README,"Aspect Ratio of a Fiber: %.15f\n",h_rp*nseg);
fprintf(README,"Time Step: %.15E\n",h_dt);
fprintf(README,"Total Strain: %.15E\n",strain);
fprintf(README,"Box Side X: %.15f\n",h_sidex);
fprintf(README,"Box Side Y: %.15f\n",h_sidey);
fprintf(README,"Box Side Z: %.15f\n",h_sidez);
fprintf(README,"concentration,nL3: %.15f\n",nL3);
fprintf(README,"Volume fraction: %.15E\n",volfrac);
fprintf(README,"Consistency: %.15E\n",consistency);
fclose(README);
////////////////////////////////////////////////////
// copy data to constant memory
cudaMemcpyToSymbol(npcn,&h_npcn,sizeof(int),0,cudaMemcpyHostToDevice)
    ;
cudaMemcpyToSymbol(nfib,&h_nfib,sizeof(int),0,cudaMemcpyHostToDevice)
    ;
cudaMemcpyToSymbol(rp,&h_rp,sizeof(float),0,cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(sidex,&h_sidex,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(sidey,&h_sidey,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(sidez,&h_sidez,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(fstar,&h_fstar,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(fact,&h_fact,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(Astar,&h_Astar,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(decatt,&h_decatt,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(fac,&h_fac,sizeof(int),0,cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(over_cut,&h_over_cut,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(contact_cutoff,&h_contact_cutoff,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(rep_cutoff,&h_rep_cutoff,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(C1,&h_C1,sizeof(float),0,cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(C5,&h_C5,sizeof(float),0,cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(C6,&h_C6,sizeof(float),0,cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(C7,&h_C7,sizeof(float),0,cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(C8,&h_C8,sizeof(float),0,cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(C9,&h_C9,sizeof(float),0,cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(C10,&h_C10,sizeof(float),0,cudaMemcpyHostToDevice)
    ;
cudaMemcpyToSymbol(Omega_x,&h_Omega_x,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(Omega_y,&h_Omega_y,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(Omega_z,&h_Omega_z,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(elf,&h_elf,sizeof(float),0,cudaMemcpyHostToDevice)
    ;
cudaMemcpyToSymbol(dt,&h_dt,sizeof(float),0,cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(duxdx,&h_duxdx,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(duxdy,&h_duxdy,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(duxdz,&h_duxdz,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(duydx,&h_duydx,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(duydy,&h_duydy,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(duydz,&h_duydz,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(duzdx,&h_duzdx,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(duzdy,&h_duzdy,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(duzdz,&h_duzdz,sizeof(float),0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(C0,&h_C0,sizeof(float),0,cudaMemcpyHostToDevice);
```

```
cudaMemcpyToSymbol(C2,&h_C2,sizeof(float),0,cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(C3,&h_C3,sizeof(float),0,cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(C4,&h_C4,sizeof(float),0,cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(E11,&h_E11,sizeof(float),0,cudaMemcpyHostToDevice)
    ;
cudaMemcpyToSymbol(E12,&h_E12,sizeof(float),0,cudaMemcpyHostToDevice)
    ;
cudaMemcpyToSymbol(E13,&h_E13,sizeof(float),0,cudaMemcpyHostToDevice)
    ;
cudaMemcpyToSymbol(E22,&h_E22,sizeof(float),0,cudaMemcpyHostToDevice)
    ;
cudaMemcpyToSymbol(E23,&h_E23,sizeof(float),0,cudaMemcpyHostToDevice)
    ;
cudaMemcpyToSymbol(E33,&h_E33,sizeof(float),0,cudaMemcpyHostToDevice)
    ;
// copy to global memory
cudaMemcpy(d_nseg,&nseg,sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(d_delta_rx,&delta_rx,sizeof(float),cudaMemcpyHostToDevice)
    ;
cudaMemcpy(d_Ya,&Ya,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_Yc,&Yc,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_Yh,&Yh,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_rcmx,rcmx,h_nfib*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_rcmy,rcmy,h_nfib*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_rcmz,rcmz,h_nfib*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_q0,q0,h_nfib*nseg*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_q1,q1,h_nfib*nseg*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_q2,q2,h_nfib*nseg*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_q3,q3,h_nfib*nseg*sizeof(float),cudaMemcpyHostToDevice);
//////////////////////////////////////////////////
// stores device pointers into double pointer arrays
setVarArray << <1,1 >> >(var,intVar,d_nseg,d_step,potCon,potConSize,
  d_rcmx,d_rcmy,d_rcmz,d_rx,d_ry,d_rz,d_q0,d_q1,d_q2,d_q3,d_q0dot,
      d_q1dot,
  d_q2dot,d_q3dot,d_qe0,d_qe1,d_qe2,d_qe3,d_px,d_py,d_pz,d_ucmx,
      d_ucmy,
  d_ucmz,d_ucox,d_ucoy,d_ucoz,d_ux,d_uy,d_uz,d_uxfl,d_uyfl,d_uzfl,
      d_wx,d_wy,
  d_wz,d_R11,d_R12,d_R13,d_R21,d_R22,d_R23,d_fcx,d_fcy,d_fcz,
  d_tcx,d_tcy,d_tcz,d_fbx,d_fby,d_fbz,d_tbx,d_tby,d_tbz,d_D1,d_D2,
      d_D3,
  d_A11,d_A12,d_A13,d_A23,d_A22,d_A33,d_C11,d_C12,d_C13,d_C23,d_C22,
      d_C33,
  d_neighb_cutoff,d_Ya,d_Yc,d_Yh,d_delta_rx,
  d_Stress,d_frx,d_fry,d_frz);
gpuErrchk(cudaDeviceSynchronize());
// initialize variables
gpuErrchk(cudaPeekAtLastError());
initialize << <numBlocks,numThreads >> >(var,intVar);
gpuErrchk(cudaPeekAtLastError());
gpuErrchk(cudaDeviceSynchronize());
if (cudaSuccess != cudaGetLastError()){return 1;}
// copy data to host
step = 0;
cudaMemcpy(rcmx,d_rcmx,h_nfib*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(rcmy,d_rcmy,h_nfib*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(rcmz,d_rcmz,h_nfib*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(rx,d_rx,h_nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(ry,d_ry,h_nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(rz,d_rz,h_nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(px,d_px,h_nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(py,d_py,h_nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(pz,d_pz,h_nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(ux,d_ux,h_nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(uy,d_uy,h_nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(uz,d_uz,h_nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(wx,d_wx,h_nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(wy,d_wy,h_nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(wz,d_wz,h_nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
// print configurations
tprint = 0.0;
fwrite(&tprint,sizeof(float),1,center_mass);
fwrite(rcmx,sizeof(float),h_nfib,center_mass);
fwrite(rcmy,sizeof(float),h_nfib,center_mass);
fwrite(rcmz,sizeof(float),h_nfib,center_mass);
fwrite(&tprint,sizeof(float),1,uxfile);
fwrite(&tprint,sizeof(float),1,uyfile);
fwrite(&tprint,sizeof(float),1,uzfile);
fwrite(&tprint,sizeof(float),1,wxfile);
fwrite(&tprint,sizeof(float),1,wyfile);
fwrite(&tprint,sizeof(float),1,wzfile);
fwrite(&tprint,sizeof(float),1,pxfile);
fwrite(&tprint,sizeof(float),1,pyfile);
fwrite(&tprint,sizeof(float),1,pzfile);
fwrite(&tprint,sizeof(float),1,rxfile);
fwrite(&tprint,sizeof(float),1,ryfile);
fwrite(&tprint,sizeof(float),1,rzfile);
fwrite(&tprint,sizeof(float),1,q0file);
fwrite(&tprint,sizeof(float),1,q1file);
fwrite(&tprint,sizeof(float),1,q2file);
fwrite(&tprint,sizeof(float),1,q3file);
fwrite(px,sizeof(float),h_nfib*nseg,pxfile);
fwrite(py,sizeof(float),h_nfib*nseg,pyfile);
fwrite(pz,sizeof(float),h_nfib*nseg,pzfile);
fwrite(rx,sizeof(float),h_nfib*nseg,rxfile);
fwrite(ry,sizeof(float),h_nfib*nseg,ryfile);
fwrite(rz,sizeof(float),h_nfib*nseg,rzfile);
fwrite(q0,sizeof(float),h_nfib*nseg,q0file);
fwrite(q1,sizeof(float),h_nfib*nseg,q1file);
fwrite(q2,sizeof(float),h_nfib*nseg,q2file);
fwrite(q3,sizeof(float),h_nfib*nseg,q3file);
fwrite(ux,sizeof(float),h_nfib*nseg,uxfile);
fwrite(uy,sizeof(float),h_nfib*nseg,uyfile);
fwrite(uz,sizeof(float),h_nfib*nseg,uzfile);
fwrite(wx,sizeof(float),h_nfib*nseg,wxfile);
fwrite(wy,sizeof(float),h_nfib*nseg,wyfile);
```

```
fwrite(wz,sizeof(float),h_nfib*nseg,wzfile);
// zeroes variables
zeroVar << <numBlocks,numThreads >> >(var,intVar);
gpuErrchk(cudaPeekAtLastError());
gpuErrchk(cudaDeviceSynchronize());
// set up constant groups
hydroRes << <numBlocks,numThreads >> >(var,intVar);
gpuErrchk(cudaPeekAtLastError());
gpuErrchk(cudaDeviceSynchronize());
// generates neighbor lists
neighbor_list <<< nBlocks,128 >>> (var,intVar,neighbor_cutoff);
gpuErrchk(cudaPeekAtLastError());
gpuErrchk(cudaDeviceSynchronize());
// calculates separation between fibers in neighbor lists
contact << <nBlocks,32 >> >(var,intVar);
gpuErrchk(cudaPeekAtLastError());
gpuErrchk(cudaDeviceSynchronize());
// generate random numbers
curandSetStream(gen,stream3);
curandGenerateNormal(gen,randN,6*h_nfib*nseg,0.0,1.0);
  // calculate order parameter
  orientCalc(px,py,pz,h_nfib,&P2);
  fprintf(orientfile,"0.0000\t%.4f\n",P2);
// calculates the stress of the suspension
  stress << <numBlocks,numThreads >> >(var,intVar);
  gpuErrchk(cudaDeviceSynchronize());
  fprintf(Stress_tensor,"%7.5f %15.9f %15.9f %15.9f %15.9f %15.9f
       %15.9f\n",
0.0,Stress[0],Stress[1],Stress[2],Stress[3],Stress[4],Stress[5]);
////////////////////////////////////////////////////
// main time loop
for (step = 0; step < time_steps +1; step++){
  // copy the current step to device,also blocks cpu
  cudaMemcpy(d_step,&step,sizeof(int),cudaMemcpyHostToDevice);
  updateVel << <numBlocks,numThreads >> >(var,intVar,randN);
  if (h_duxdz != 0.0)
    lees_edwards << <1,1,0,stream1 >> >(var,intVar);
  updatePos << <numBlocks,numThreads >> >(var,intVar);
  if ((step+1) % orient_write == 0 || (step+1) % config_write == 0){
        tprint = (step+1)*h_dt;
    cudaMemcpyAsync(px,d_px,h_nfib*nseg*sizeof(float),
         cudaMemcpyDeviceToHost,stream3);
        cudaMemcpyAsync(py,d_py,h_nfib*nseg*sizeof(float),
             cudaMemcpyDeviceToHost,stream3);
        cudaMemcpyAsync(pz,d_pz,h_nfib*nseg*sizeof(float),
             cudaMemcpyDeviceToHost,stream3);
      cudaStreamSynchronize(stream3);
      if ((step+1) % orient_write == 0) {
      orientCalc(px,py,pz,h_nfib,&P2);
      fprintf(orientfile,"%.4f\t%.4f\n",tprint,P2);
    }
  }
  zeroVar << <numBlocks,numThreads,0,stream1 >> >(var,intVar);
  if ((step+1) % config_write == 0){
    cudaMemcpyAsync(ux,d_ux,h_nfib*nseg*sizeof(float),
         cudaMemcpyDeviceToHost,stream3);
    cudaMemcpyAsync(uy,d_uy,h_nfib*nseg*sizeof(float),
         cudaMemcpyDeviceToHost,stream3);
    cudaMemcpyAsync(uz,d_uz,h_nfib*nseg*sizeof(float),
         cudaMemcpyDeviceToHost,stream3);
    cudaMemcpyAsync(wx,d_wx,h_nfib*nseg*sizeof(float),
         cudaMemcpyDeviceToHost,stream3);
    cudaMemcpyAsync(wy,d_wy,h_nfib*nseg*sizeof(float),
         cudaMemcpyDeviceToHost,stream3);
    cudaMemcpyAsync(wz,d_wz,h_nfib*nseg*sizeof(float),
         cudaMemcpyDeviceToHost,stream3);
  }
  gpuErrchk(cudaDeviceSynchronize());
  hydroRes << <numBlocks,numThreads,0,stream0 >> >(var,intVar);
  updateBod << < numBlocks,numThreads,0,stream1 >> >(var,intVar);
  if ((step+1) % stress_write == 0){
    stress << <numBlocks,numThreads,0,stream4 >> >(var,intVar);
  }
  if ((step+1) % config_write == 0){
    cudaMemcpyAsync(rx,d_rx,h_nfib*nseg*sizeof(float),
         cudaMemcpyDeviceToHost,stream3);
    cudaMemcpyAsync(ry,d_ry,h_nfib*nseg*sizeof(float),
         cudaMemcpyDeviceToHost,stream3);
    cudaMemcpyAsync(rz,d_rz,h_nfib*nseg*sizeof(float),
         cudaMemcpyDeviceToHost,stream3);
    cudaMemcpyAsync(rcmx,d_rcmx,h_nfib*sizeof(float),
         cudaMemcpyDeviceToHost,stream3);
    cudaMemcpyAsync(rcmy,d_rcmy,h_nfib*sizeof(float),
         cudaMemcpyDeviceToHost,stream3);
    cudaMemcpyAsync(rcmz,d_rcmz,h_nfib*sizeof(float),
         cudaMemcpyDeviceToHost,stream3);
    cudaMemcpyAsync(q0,d_q0,h_nfib*nseg*sizeof(float),
         cudaMemcpyDeviceToHost,stream3);
    cudaMemcpyAsync(q1,d_q1,h_nfib*nseg*sizeof(float),
         cudaMemcpyDeviceToHost,stream3);
    cudaMemcpyAsync(q2,d_q2,h_nfib*nseg*sizeof(float),
         cudaMemcpyDeviceToHost,stream3);
    cudaMemcpyAsync(q3,d_q3,h_nfib*nseg*sizeof(float),
         cudaMemcpyDeviceToHost,stream3);
    fwrite(&tprint,sizeof(float),1,uxfile);
    fwrite(&tprint,sizeof(float),1,uyfile);
    fwrite(&tprint,sizeof(float),1,uzfile);
    fwrite(&tprint,sizeof(float),1,wxfile);
    fwrite(&tprint,sizeof(float),1,wyfile);
    fwrite(&tprint,sizeof(float),1,wzfile);
    fwrite(ux,sizeof(float),h_nfib*nseg,uxfile);
    fwrite(uy,sizeof(float),h_nfib*nseg,uyfile);
    fwrite(uz,sizeof(float),h_nfib*nseg,uzfile);
    fwrite(wx,sizeof(float),h_nfib*nseg,wxfile);
    fwrite(wy,sizeof(float),h_nfib*nseg,wyfile);
    fwrite(wz,sizeof(float),h_nfib*nseg,wzfile);
```

```
}
if ((step+1) % neighbor_check == 0){
  neighbor_list <<< nBlocks,128 >>> (var,intVar,neighbor_cutoff);
}
cudaMemcpy(Stress,d_Stress,6*sizeof(float),cudaMemcpyDeviceToHost);
if ((step+1) % config_write == 0){
  fwrite(&tprint,sizeof(float),1,center_mass);
  fwrite(rcmx,sizeof(float),h_nfib,center_mass);
  fwrite(rcmy,sizeof(float),h_nfib,center_mass);
  fwrite(rcmz,sizeof(float),h_nfib,center_mass);
  fwrite(&tprint,sizeof(float),1,pxfile);
  fwrite(&tprint,sizeof(float),1,pyfile);
  fwrite(&tprint,sizeof(float),1,pzfile);
  fwrite(&tprint,sizeof(float),1,rxfile);
  fwrite(&tprint,sizeof(float),1,ryfile);
  fwrite(&tprint,sizeof(float),1,rzfile);
  fwrite(&tprint,sizeof(float),1,q0file);
  fwrite(&tprint,sizeof(float),1,q1file);
  fwrite(&tprint,sizeof(float),1,q2file);
  fwrite(&tprint,sizeof(float),1,q3file);
  fwrite(px,sizeof(float),h_nfib*nseg,pxfile);
  fwrite(py,sizeof(float),h_nfib*nseg,pyfile);
  fwrite(pz,sizeof(float),h_nfib*nseg,pzfile);
  fwrite(rx,sizeof(float),h_nfib*nseg,rxfile);
  fwrite(ry,sizeof(float),h_nfib*nseg,ryfile);
  fwrite(rz,sizeof(float),h_nfib*nseg,rzfile);
  fwrite(q0,sizeof(float),h_nfib*nseg,q0file);
  fwrite(q1,sizeof(float),h_nfib*nseg,q1file);
  fwrite(q2,sizeof(float),h_nfib*nseg,q2file);
  fwrite(q3,sizeof(float),h_nfib*nseg,q3file);
}
contact << <nBlocks,64,0,stream0 >> >(var,intVar);
curandGenerateNormal(gen,randN,6*h_nfib*nseg,0.0,1.0);
if ((step+1) % stress_write == 0){
  cudaStreamSynchronize(stream0);
  fprintf(Stress_tensor,"%7.5f %15.9f %15.9f %15.9f %15.9f %15.9f
       %15.9f\n",
    (step+1)*h_dt,Stress[0],Stress[1],Stress[2],Stress[3],Stress
        [4],Stress[5]);
}
// flush files occasionally in case of server crashes
if ((step+1) % 50000 == 0){
  fflush(stdout);
  fflush(Stress_tensor);
  fflush(rxfile); fflush(ryfile);
  fflush(rzfile); fflush(pxfile);
  fflush(pyfile); fflush(pzfile);
  fflush(wxfile); fflush(wyfile);
  fflush(wzfile); fflush(uxfile);
  fflush(uyfile); fflush(uzfile);
  fflush(q0file); fflush(q1file);
  fflush(q2file); fflush(q3file);
  fflush(center_mass); fflush(orientfile);
    }
  }
  // close files
  cudaDeviceSynchronize();
  fclose(center_mass);
  fclose(pxfile); fclose(pyfile); fclose(pzfile);
  fclose(rxfile); fclose(ryfile); fclose(rzfile);
  fclose(uxfile); fclose(uyfile); fclose(uzfile);
  fclose(wxfile); fclose(wyfile); fclose(wzfile);
  fclose(Stress_tensor); fclose(orientfile);
  fclose(q0file); fclose(q1file); fclose(q2file); fclose(q3file);
  return 0;
}
void orientCalc(float *px,float *py,float *pz,int h_nfib,float *P2){
    float Q11 = 0.0,Q12 = 0.0,Q13 = 0.0;
    float Q22 = 0.0,Q23 = 0.0,Q33 = 0.0;
    // Q: orientational order tensor
  float d1,d2,p;
    float B11,B12,B13,B22,B23,B33;
    float trace,r,phi;
    *P2 = 0.0;
    for (int i = 0; i < h_nfib; i++){
        Q11 += 3.0*px[i]*px[i]-1.0;
        Q12 += 3.0*px[i]*py[i];
        Q13 += 3.0*px[i]*pz[i];
        Q22 += 3.0*py[i]*py[i]-1.0;
        Q23 += 3.0*py[i]*pz[i];
        Q33 += 3.0*pz[i]*pz[i]-1.0;
    }
  //largest eigenvalue of matrix Q
    d1 = Q12*Q12+Q13*Q13+Q23*Q23;
    if (fabsf(d1) <= 1.0e-6){
        *P2 = Q11;
        if (*P2 < Q22){
            *P2 = Q22;
        }
        if (*P2 < Q33){
            *P2 = Q33;
        }
    }
    else{
        trace = (Q11+Q22+Q33)/3.0;
        d2 = pow(Q11-trace,2.0)+pow(Q22-trace,2.0)+pow(Q33-trace,2.0)
             +2.0*d1;
        p = sqrt(d2/6.0);
        B11 = (Q11-trace)/p;
        B12 = Q12/p;
        B13 = Q13/p;
        B22 = (Q22-trace)/p;
        B23 = Q23/p;
        B33 = (Q33-trace)/p;
        r = 0.5*(B11*(B22*B33- B23*B23)-B12*(B12*B33-B23*B13)+B13*(B12*
             B23-B22*B13));
```

```
        if (r <= -1.0){
            phi = pi/3.0;
        }
        else if (r >= 1.0){
            phi = 0.0;
        }
        else{
            phi = acos(r)/3.0;
        }
        *P2 = trace+2.0*p*cos(phi);
    }
    *P2 /= (2.0*h_nfib);
}
```

contact.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime_api.h>
#include <math.h>
#include "contact.h"
#include "consts.cuh"
using namespace std;
__device__ void parallelSepCon(int mi,int nj,float sx,float sy,float
     sz,float pxmi,float pymi,float pzmi,
  float pxnj,float pynj,float pznj,float pdotp,
  float *xmin,float *ymin);
__global__ void contact(float **var,int **intVar){
  int mi=blockIdx.x+gridDim.x*blockIdx.y;
  if (mi >= nfib) return;
  int tid=threadIdx.x;
  float rxnj,rynj,rznj,pxnj,pynj,pznj;
  float sxx,syy,szz,corx,cory,corz;
  float rxmi_shift,rymi_shift,rzmi_shift;
  float pdotp,xmin,ymin,ddx,ddy,ddz,sep;
  float xi[9],yj[9],gij,nijx,nijy,nijz,forc;
  float sep_tmp,hij;
  int nj,ipos,ith;
  __shared__ float rxmi,rymi,rzmi,pxmi,pymi,pzmi;
  __shared__ float delta_rx;
  __shared__ float *rx,*ry,*rz,*px,*py,*pz;
  __shared__ float *fcx,*fcy,*fcz,*tcx,*tcy,*tcz;
  __shared__ int nPair;
  int *potCon=intVar[32];
  int *potConSize=intVar[33];
  if (tid == 0){
    rx=var[3]; ry=var[4];
    rz=var[5]; px=var[18];
    py=var[19]; pz=var[20];
    fcx=var[66]; fcy=var[67];
    fcz=var[68]; tcx=var[69];
    tcy=var[70]; tcz=var[71];
    rxmi=rx[mi]; rymi=ry[mi]; rzmi=rz[mi];
    pxmi=px[mi]; pymi=py[mi]; pzmi=pz[mi];
    nPair=potConSize[mi];
    delta_rx=*var[138];
  }
  __syncthreads();
  while (tid < nPair){
    nj=potCon[mi*npcn+tid];
    rxnj=rx[nj]; rynj=ry[nj]; rznj=rz[nj];
    pxnj=px[nj]; pynj=py[nj]; pznj=pz[nj];
    // find minimum image (for shear flow system)
    sxx=rxnj-rxmi;
    syy=rynj-rymi;
    szz=rznj-rzmi;
    cory=roundf(syy / sidey);
    corz=roundf(szz / sidez);
    sxx=sxx-corz*delta_rx;
    corx=roundf(sxx / sidex);
    sxx=sxx-corx*sidex;
    syy=syy-cory*sidey;
    szz=szz-corz*sidez;
    rxmi_shift=rxnj-sxx;
    rymi_shift=rynj-syy;
    rzmi_shift=rznj-szz;
    pdotp=pxmi*pxnj+pymi*pynj+pzmi*pznj;
    xmin=(-(pxnj*sxx+pynj*syy+pznj*szz)* pdotp
     +(pxmi*sxx+pymi*syy+pzmi*szz))
      / (1.0-pdotp*pdotp);
    ymin=((pxmi*sxx+pymi*syy+pzmi*szz)* pdotp
     -(pxnj*sxx+pynj*syy+pznj*szz))
      / (1.0-pdotp*pdotp);
    ddx=rxnj+ymin*pxnj-rxmi_shift-xmin*pxmi;
    ddy=rynj+ymin*pynj-rymi_shift-xmin*pymi;
    ddz=rznj+ymin*pznj-rzmi_shift-xmin*pzmi;
    sep=ddx*ddx+ddy*ddy+ddz*ddz;
    ipos=8;
    yj[0]=rp;
    xi[0]=pxmi*sxx+pymi*syy+pzmi*szz+yj[0]*pdotp;
    yj[1]=-rp;
    xi[1]=pxmi*sxx+pymi*syy+pzmi*szz+yj[1]*pdotp;
    xi[2]=rp;
    yj[2]=-(pxnj*sxx+pynj*syy+pznj*szz)+xi[2]*pdotp;
    xi[3]=-rp;
    yj[3]=-(pxnj*sxx+pynj*syy+pznj*szz)+xi[3]*pdotp;
    xi[4]=rp; yj[4]=rp;
    xi[5]=rp; yj[5]=-rp;
    xi[6]=-rp; yj[6]=rp;
    xi[7]=-rp; yj[7]=-rp;
    xi[8]=xmin; yj[8]=ymin;
    // Check if segments are parallel
    if (fabsf(pdotp*pdotp-1.0) <= 1.0e-6) {
      parallelSepCon(mi,nj,sxx,syy,szz,pxmi,pymi,pzmi,
        pxnj,pynj,pznj,pdotp,&xmin,&ymin);
      sep=(sxx+ymin*pxnj-xmin*pxmi)*(sxx+ymin*pxnj-xmin*pxmi) +
```

```
      (syy+ymin*pynj-xmin*pymi)*(syy+ymin*pynj-xmin*pymi) +
      (szz+ymin*pznj-xmin*pzmi)*(szz+ymin*pznj-xmin*pzmi);
  }
  else if (sep < rep_cutoff && (fabsf(xmin) >= rp || fabsf(ymin) >=
       rp)){
    sep=1000.0;
    // check which end-side or end-end separation
    // is the smallest
    for (ith=0; ith < 8; ith++){
      sep_tmp=(sxx+yj[ith]*pxnj-xi[ith]*pxmi)*(sxx+yj[ith]*pxnj-xi[
          ith]*pxmi) +
        (syy+yj[ith]*pynj-xi[ith]*pymi)*(syy+yj[ith]*pynj-xi[ith]*
            pymi) +
        (szz+yj[ith]*pznj-xi[ith]*pzmi)*(szz+yj[ith]*pznj-xi[ith]*
            pzmi);
      if (sep_tmp < sep && fabsf(xi[ith]) <= rp && fabsf(yj[ith]) <=
          rp){
        sep=sep_tmp;
        ipos=ith;
      }
    }
    xmin=xi[ipos];
    ymin=yj[ipos];
  }
  gij=sqrtf(sep);
  nijx=(sxx+ymin*pxnj-xmin*pxmi) / gij;
  nijy=(syy+ymin*pynj-xmin*pymi) / gij;
  nijz=(szz+ymin*pznj-xmin*pzmi) / gij;
  hij=gij-2.0;
  if (gij < over_cut){
    hij=over_cut-2.0;
  }
  forc=fstar*expf(-fact*hij)-Astar*expf(-decatt*hij*hij);
  if (sep < rep_cutoff){
    atomicAdd(fcx+mi,-nijx*forc);
    atomicAdd(fcy+mi,-nijy*forc);
    atomicAdd(fcz+mi,-nijz*forc);
    atomicAdd(tcx+mi,-forc*xmin*(pymi*nijz-pzmi*nijy));
    atomicAdd(tcy+mi,-forc*xmin*(pzmi*nijx-pxmi*nijz));
    atomicAdd(tcz+mi,-forc*xmin*(pxmi*nijy-pymi*nijx));
    atomicAdd(fcx+nj,nijx*forc);
    atomicAdd(fcy+nj,nijy*forc);
    atomicAdd(fcz+nj,nijz*forc);
    atomicAdd(tcx+nj,forc*ymin*(pynj*nijz-pznj*nijy));
    atomicAdd(tcy+nj,forc*ymin*(pznj*nijx-pxnj*nijz));
    atomicAdd(tcz+nj,forc*ymin*(pxnj*nijy-pynj*nijx));
  }
  tid += blockDim.x;
 }
}
__device__ void parallelSepCon(int mi,int nj,float sx,float sy,float
     sz,float pxmi,float pymi,float pzmi,
  float pxnj,float pynj,float pznj,float pdotp,
  float *xmin,float *ymin){
  float posneg,pn2,dist,sijp,sijm,sjip,sjim;
  // The different end point to fiber contact points
  sijp=pxmi*sx+pymi*sy+pzmi*sz+rp*pdotp;
  sijm=pxmi*sx+pymi*sy+pzmi*sz-rp*pdotp;
  sjip=-(pxnj*sx+pynj*sy+pznj*sz)+rp*pdotp;
  sjim=-(pxnj*sx+pynj*sy+pznj*sz)-rp*pdotp;
  // for fiber i
  if (fabsf(sijp) < fabsf(sijm)){
    *xmin=sijp;
    posneg=1.0;
  }
  else if (fabsf(sijp) > fabsf(sijm)){
    *xmin=sijm;
    posneg=-1.0;
  }
  else{
    *xmin=0.0;
    posneg=0.0;
  }
  if (*xmin >= rp){
    *xmin=rp;
  }
  if (*xmin <= -rp){
    *xmin=-rp;
  }
  // for fiber j
  if (fabsf(sjip) < fabsf(sjim)){
    *ymin=sjip;
  }
  else if (fabsf(sjip) > fabsf(sjim)){
    *ymin=sjim;
  }
  else{
    *ymin=0.0;
    posneg=0.0;
  }
  if (*ymin >= rp){
    *ymin=rp;
  }
  if (*ymin <= -rp){
    *ymin=-rp;
  }
  if (fabsf(*xmin) < rp && fabsf(*ymin) < rp){
    if (pdotp > 0.0){
      pn2=1.0;
    }
    else{
      pn2=-1.0;
    }
    dist=(rp+posneg**xmin) / 2.0;
    *xmin=*xmin-posneg*dist;
    *ymin=*ymin+posneg*pn2*dist;
```

```
  }
}
```

hydroRes.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime_api.h>
#include "hydroRes.h"
#include "consts.cuh"
using namespace std;
__global__ void hydroRes(float **var, int **intVar){
  int mi=threadIdx.x+blockDim.x * blockIdx.x;
  if (mi >= nfib) return;
  float Ya=*var[131]; float Yc=*var[132];
  float Yh=*var[133]; float *rx=var[3];
  float *ry=var[4]; float *rz=var[5];
  float *px=var[18]; float *py=var[19];
  float *pz=var[20]; float *uxfl=var[30];
  float *uyfl=var[31]; float *uzfl=var[32];
  float *D1=var[78]; float *D2=var[79];
  float *D3=var[80]; float *A11=var[81];
  float *A12=var[82]; float *A13=var[83];
  float *A23=var[84]; float *A22=var[85];
  float *A33=var[86]; float *C11=var[87];
  float *C12=var[88]; float *C13=var[89];
  float *C23=var[90]; float *C22=var[91];
  float *C33=var[92]; float rxmi, rymi, rzmi;
  float pxmi, pymi, pzmi;
  pxmi=px[mi]; pymi=py[mi]; pzmi=pz[mi];
  // Calculate inverse of the resistance tensors
  A11[mi]=1.0/Ya+C3*pxmi*pxmi;
  A12[mi]=C3*pxmi*pymi;
  A13[mi]=C3*pxmi*pzmi;
  A22[mi]=1.0/Ya+C3*pymi*pymi;
  A23[mi]=C3*pymi*pzmi;
  A33[mi]=1.0/Ya+C3*pzmi*pzmi;
  C11[mi]=1.0/Yc+C4*pxmi*pxmi;
  C12[mi]=C4*pxmi*pymi;
  C13[mi]=C4*pxmi*pzmi;
  C22[mi]=1.0/Yc+C4*pymi*pymi;
  C23[mi]=C4*pymi*pzmi;
  C33[mi]=1.0/Yc+C4*pzmi*pzmi;
  D1[mi]=-Yh*((E12*pxmi+E22*pymi+E23*pzmi)*pzmi -
    (E13*pxmi+E23*pymi+E33*pzmi)*pymi);
  D2[mi]=-Yh*((E13*pxmi+E23*pymi+E33*pzmi)*pxmi -
    (E11*pxmi+E12*pymi+E13*pzmi)*pzmi);
  D3[mi]=-Yh*((E11*pxmi+E12*pymi+E13*pzmi)*pymi -
    (E12*pxmi+E22*pymi+E23*pzmi)*pxmi);
  // ambient velocity field
  rxmi=rx[mi]; rymi=ry[mi]; rzmi=rz[mi];
  uxfl[mi]=fac*(duxdx*rxmi+duxdy*rymi+duxdz*rzmi);
  uyfl[mi]=fac*(duydx*rxmi+duydy*rymi+duydz*rzmi);
  uzfl[mi]=fac*(duzdx*rxmi+duzdy*rymi+duzdz*rzmi);
}
```

initialize.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cuda_runtime_api.h>
#include "initialize.h"
#include "consts.cuh"
using namespace std;
__global__ void initialize(float **var,int **intVar){
  // calculates thread index
  int mi=threadIdx.x+blockIdx.x*blockDim.x;
  if (mi >= nfib) return;
  // get variables
  float *q0=var[6]; float *q1=var[7];
  float *q2=var[8]; float *q3=var[9];
  float *q0dot=var[10]; float *q1dot=var[11];
  float *q2dot=var[12]; float *q3dot=var[13];
  float *qe0=var[14]; float *qe1=var[15];
  float *qe2=var[16]; float *qe3=var[17];
  float *px=var[18]; float *py=var[19];
  float *pz=var[20]; float *ux=var[27];
  float *uy=var[28]; float *uz=var[29];
  float *uxfl=var[30]; float *uyfl=var[31];
  float *uzfl=var[32]; float *wx=var[33];
  float *wy=var[34]; float *wz=var[35];
  float *R11=var[36]; float *R12=var[37];
  float *R13=var[38]; float *R21=var[39];
  float *R22=var[40]; float *R23=var[41];
  float *fcx=var[66]; float *fcy=var[67];
  float *fcz=var[68]; float *tcx=var[69];
  float *tcy=var[70]; float *tcz=var[71];
  float *fbx=var[72]; float *fby=var[73];
  float *fbz=var[74]; float *tbx=var[75];
  float *tby=var[76]; float *tbz=var[77];
  float *rcmx=var[0]; float *rcmy=var[1];
  float *rcmz=var[2]; float *rx=var[3];
  float *ry=var[4]; float *rz=var[5];
  // local variables
  float q0mi,q1mi,q2mi,q3mi;
  float pxmi,pymi,pzmi,dum;
  // zero varaibles
  ux[mi]=0.0; uy[mi]=0.0; uz[mi]=0.0;
  wx[mi]=0.0; wy[mi]=0.0; wz[mi]=0.0;
  q0dot[mi]=0.0; q1dot[mi]=0.0; q2dot[mi]=0.0; q3dot[mi]=0.0;
  qe0[mi]=0.0; qe1[mi]=0.0; qe2[mi]=0.0; qe3[mi]=0.0;
  uxfl[mi]=0.0; uyfl[mi]=0.0; uzfl[mi]=0.0;
  fcx[mi]=0.0; fcy[mi]=0.0; fcz[mi]=0.0;
  tcx[mi]=0.0; tcy[mi]=0.0; tcz[mi]=0.0;
```

```
  // Euler parameters
  q0mi=q0[mi]; q1mi=q1[mi]; q2mi=q2[mi]; q3mi=q3[mi];
  dum=sqrtf(q0mi*q0mi+q1mi*q1mi+q2mi*q2mi+q3mi*q3mi);
  q0mi /= dum; q1mi /= dum;
  q2mi /= dum; q3mi /= dum;
  q0[mi]=q0mi; q1[mi]=q1mi; q2[mi]=q2mi; q3[mi]=q3mi;
  // Rotation matrix
  R11[mi]=2.0*(q0mi*q0mi+q1mi*q1mi) - 1.0;
  R12[mi]=2.0*(q1mi*q2mi+q0mi*q3mi);
  R13[mi]=2.0*(q1mi*q3mi - q0mi*q2mi);
  R21[mi]=2.0*(q1mi*q2mi - q0mi*q3mi);
  R22[mi]=2.0*(q0mi*q0mi+q2mi*q2mi) - 1.0;
  R23[mi]=2.0*(q3mi*q2mi+q0mi*q1mi);
  pxmi=2.0*(q1mi*q3mi+q0mi*q2mi);
  pymi=2.0*(q3mi*q2mi - q0mi*q1mi);
  pzmi=2.0*(q0mi*q0mi+q3mi*q3mi) - 1.0;
  dum=sqrtf(pxmi*pxmi+pymi*pymi+pzmi*pzmi);
  pxmi /= dum; pymi /= dum; pzmi /= dum;
  px[mi]=pxmi; py[mi]=pymi; pz[mi]=pzmi;
  // Body forces and torques
  fbx[mi]=0.0; fby[mi]=0.0; fbz[mi]=0.0;
  tbx[mi]=elf*pzmi*pymi;
    tby[mi]=-elf*pzmi*pxmi;
  tbz[mi]=0.0;
  // segment center of mass and
  // fiber center of mass is the same
  rx[mi]=rcmx[mi];
  ry[mi]=rcmy[mi];
  rz[mi]=rcmz[mi];
}
```

lees_edwards.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime_api.h>
#include <math.h>
#include "lees_edwards.h"
#include "consts.cuh"
using namespace std;
__global__ void lees_edwards(float **var, int **intVar){
  float *delta_rx = var[138];
  *delta_rx = *delta_rx + duxdz*sidex*dt;
  *delta_rx = *delta_rx - lroundf(*delta_rx / sidex)*sidex;
}
```

neighbor_list.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime_api.h>
#include <math.h>
#include "neighbor_list.h"
#include "consts.cuh"
using namespace std;
// calculates the separation between parallel fibers
__device__ void parallelSep(int mi,int nj,float sx,float sy,float sz,
    float pxmi,float pymi,float pzmi,
  float pxnj,float pynj,float pznj,float pdotp,
  float *xmin,float *ymin);
// generates neighbor lists
__global__ void neighbor_list(float **var,int **intVar,float
    neighbor_cutoff){
  int mi = blockIdx.x+blockIdx.y*gridDim.x;
  if (mi >= nfib) return;
  int nj = threadIdx.x+mi+1;
  float rxnj,rynj,rznj,pxnj,pynj,pznj;
  float sxx,syy,szz,corx,cory,corz;
  float rxmi_shift,rymi_shift,rzmi_shift;
  float pdotp,xmin,ymin,ddx,ddy,ddz,sep;
  float xi[9],yj[9];
  float sep_tmp;
  int ipos,ith,nPair;
  __shared__ float rxmi,rymi,rzmi,pxmi,pymi,pzmi;
  __shared__ float delta_rx;
  __shared__ float *rx,*ry,*rz,*px,*py,*pz;
  int *potCon = intVar[32];
  int *potConSize = intVar[33];
  if (threadIdx.x == 0){
    rx = var[3]; ry = var[4]; rz = var[5];
    px = var[18]; py = var[19]; pz = var[20];
    rxmi = rx[mi]; rymi = ry[mi]; rzmi = rz[mi];
    pxmi = px[mi]; pymi = py[mi]; pzmi = pz[mi];
    delta_rx = *var[138];
    potConSize[mi] = 0;
  }
  __syncthreads();
  // each fiber considers fibers with larger ids
  // fibers within cutoff are added to potCon(nfib,npcn)
  // counters are updated to potConSize(nfib,1)
  while (nj < nfib){
    rxnj = rx[nj]; rynj = ry[nj]; rznj = rz[nj];
    pxnj = px[nj]; pynj = py[nj]; pznj = pz[nj];
    // find minimum image (for shear flow system)
    sxx = rxnj-rxmi;
    syy = rynj-rymi;
    szz = rznj-rzmi;
    cory = roundf(syy/sidey);
    corz = roundf(szz/sidez);
    sxx = sxx-corz*delta_rx;
    corx = roundf(sxx/sidex);
    sxx = sxx-corx*sidex;
    syy = syy-cory*sidey;
    szz = szz-corz*sidez;
```

```
rxmi_shift = rxnj-sxx;
rymi_shift = rynj-syy;
rzmi_shift = rznj-szz;
pdotp = pxmi*pxnj+pymi*pynj+pzmi*pznj;
xmin = (-(pxnj*sxx+pynj*syy+pznj*szz)* pdotp
  + (pxmi*sxx+pymi*syy+pzmi*szz))
  / (1.0-pdotp*pdotp);
ymin = ((pxmi*sxx+pymi*syy+pzmi*szz)* pdotp
  - (pxnj*sxx+pynj*syy+pznj*szz))
  / (1.0-pdotp*pdotp);
ddx = rxnj+ymin*pxnj-rxmi_shift-xmin*pxmi;
ddy = rynj+ymin*pynj-rymi_shift-xmin*pymi;
ddz = rznj+ymin*pznj-rzmi_shift-xmin*pzmi;
sep = ddx*ddx+ddy*ddy+ddz*ddz;
ipos = 8;
yj[0] = rp;
xi[0] = pxmi*sxx+pymi*syy+pzmi*szz+yj[0]*pdotp;
yj[1] = -rp;
xi[1] = pxmi*sxx+pymi*syy+pzmi*szz+yj[1]*pdotp;
xi[2] = rp;
yj[2] = -(pxnj*sxx+pynj*syy+pznj*szz)+xi[2]*pdotp;
xi[3] = -rp;
yj[3] = -(pxnj*sxx+pynj*syy+pznj*szz)+xi[3]*pdotp;
xi[4] = rp; yj[4] = rp;
xi[5] = rp; yj[5] = -rp;
xi[6] = -rp; yj[6] = rp;
xi[7] = -rp; yj[7] = -rp;
xi[8] = xmin; yj[8] = ymin;
// Check if segments are parallel
if (fabsf(pdotp*pdotp-1.0) <= 1.0e-6) {
  parallelSep(mi,nj,sxx,syy,szz,pxmi,pymi,pzmi,
    pxnj,pynj,pznj,pdotp,&xmin,&ymin);
  sep = (sxx+ymin*pxnj-xmin*pxmi)*(sxx+ymin*pxnj-xmin*pxmi) +
    (syy+ymin*pynj-xmin*pymi)*(syy+ymin*pynj-xmin*pymi) +
    (szz+ymin*pznj-xmin*pzmi)*(szz+ymin*pznj-xmin*pzmi);
}
else if (sep < neighbor_cutoff && (fabsf(xmin) >= rp || fabsf(ymin)
     >= rp)){
  sep = 1000.0;
  // check which end-side or end-end separation
  // is the smallest
  for (ith = 0; ith < 8; ith++){
    sep_tmp = (sxx+yj[ith]*pxnj-xi[ith]*pxmi)*(sxx+yj[ith]*pxnj-xi[
        ith]*pxmi) +
      (syy+yj[ith]*pynj-xi[ith]*pymi)*(syy+yj[ith]*pynj-xi[ith]*
          pymi) +
      (szz+yj[ith]*pznj-xi[ith]*pzmi)*(szz+yj[ith]*pznj-xi[ith]*
          pzmi);
    if (sep_tmp < sep && fabsf(xi[ith]) <= rp && fabsf(yj[ith]) <=
        rp){
      sep = sep_tmp;
      ipos = ith;
    }
  }
  xmin = xi[ipos];
  ymin = yj[ipos];
 }
 if (sep < neighbor_cutoff){
  nPair = atomicAdd(potConSize+mi,1);
  if (nPair >= npcn){
    printf("err: allocate more space for neighbor list: npcn: %d\n"
        ,npcn);
    return;
  }
  potCon[mi*npcn+nPair] = nj;
 }
 nj += blockDim.x;
 }
}
__device__ void parallelSep(int mi,int nj,float sx,float sy,float sz,
    float pxmi,float pymi,float pzmi,
  float pxnj,float pynj,float pznj,float pdotp,
  float *xmin,float *ymin){
  float posneg,pn2,dist,sijp,sijm,sjip,sjim;
  // The different end point to fiber contact points
  sijp = pxmi*sx+pymi*sy+pzmi*sz+rp*pdotp;
  sijm = pxmi*sx+pymi*sy+pzmi*sz-rp*pdotp;
  sjip = -(pxnj*sx+pynj*sy+pznj*sz)+rp*pdotp;
  sjim = -(pxnj*sx+pynj*sy+pznj*sz)-rp*pdotp;
  // for fiber i
  if (fabsf(sijp) < fabsf(sijm)){
    *xmin = sijp;
    posneg = 1.0;
  }
  else if (fabsf(sijp) > fabsf(sijm)){
    *xmin = sijm;
    posneg = -1.0;
  }
  else{
    *xmin = 0.0;
    posneg = 0.0;
  }
  if (*xmin >= rp){
    *xmin = rp;
  }
  if (*xmin <= -rp){
    *xmin = -rp;
  }
  // for fiber j
  if (fabsf(sjip) < fabsf(sjim)){
    *ymin = sjip;
  }
  else if (fabsf(sjip) > fabsf(sjim)){
    *ymin = sjim;
  }
  else{
```

```
    *ymin = 0.0;
    posneg = 0.0;
  }
  if (*ymin >= rp){
    *ymin = rp;
  }
  if (*ymin <= -rp){
    *ymin = -rp;
  }
  if (fabsf(*xmin) < rp && fabsf(*ymin) < rp){
    if (pdotp > 0.0){
      pn2 = 1.0;
    }
    else{
      pn2 = -1.0;
    }
    dist = (rp+posneg**xmin)/2.0;
    *xmin = *xmin-posneg*dist;
    *ymin = *ymin+posneg*pn2*dist;
  }
}
```

setVarArray.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <cuda_runtime_api.h>
#include "setVarArray.h"
using namespace std;
__global__ void setVarArray(float **var,int **intVar,
  int *nseg,int *step,int *potCon,int *potConSize,
  float *rcmx,float *rcmy,float *rcmz,
  float *rx,float *ry,float *rz,float *q0,float *q1,float *q2,float *q3
       ,float *q0dot,
  float *q1dot,float *q2dot, float *q3dot,float *qe0,float *qe1,float *
       qe2,float *qe3,
  float *px,float *py,float *pz,float *ucmx,float *ucmy,float *ucmz,
       float *ucox,float *ucoy,
  float *ucoz,float *ux,float *uy,float *uz,float *uxfl,float *uyfl,
       float *uzfl,float *wx,
  float *wy,float *wz,float *R11,float *R12,float *R13,float *R21,float
        *R22,float *R23,
  float *fcx,float *fcy,float *fcz,float *tcx,float *tcy,float *tcz,
       float *fbx,
  float *fby,float *fbz, float *tbx,float *tby,float *tbz,float *D1,
       float *D2,float *D3,
  float *A11, float *A12, float *A13, float *A23, float *A22, float *
       A33, float *C11, float *C12,
  float *C13, float *C23, float *C22, float *C33,
  float *neighb_cutoff, float *Ya,float *Yc,float *Yh,
  float *delta_rx,float *Stress,float *frx,float *fry,float *frz){
  // integer variables
  intVar[7]=nseg; intVar[9]=step;
  intVar[32]=potCon; intVar[33]=potConSize;
  // floating point variables
  var[0]=rcmx; var[1]=rcmy; var[2]=rcmz;
  var[3]=rx; var[4]=ry; var[5]=rz; var[6]=q0;
  var[7]=q1; var[8]=q2; var[9]=q3; var[10]=q0dot;
  var[11]=q1dot; var[12]=q2dot;var[13]=q3dot;var[14]=qe0;
  var[15]=qe1; var[16]=qe2; var[17]=qe3; var[18]=px;
  var[19]=py; var[20]=pz; var[21]=ucmx; var[22]=ucmy;
  var[23]=ucmz; var[24]=ucox; var[25]=ucoy; var[26]=ucoz;
  var[27]=ux; var[28]=uy; var[29]=uz; var[30]=uxfl;
  var[31]=uyfl; var[32]=uzfl; var[33]=wx; var[34]=wy;
  var[35]=wz; var[36]=R11; var[37]=R12; var[38]=R13;
  var[39]=R21; var[40]=R22; var[41]=R23; var[66]=fcx;
  var[67]=fcy; var[68]=fcz; var[69]=tcx; var[70]=tcy;
  var[71]=tcz; var[72]=fbx; var[73]=fby; var[74]=fbz;
  var[75]=tbx; var[76]=tby; var[77]=tbz; var[78]=D1;
  var[79]=D2; var[80]=D3; var[81]=A11; var[82]=A12;
  var[83]=A13; var[84]=A23; var[85]=A22; var[86]=A33;
  var[87]=C11; var[88]=C12; var[89]=C13; var[90]=C23;
  var[91]=C22; var[92]=C33; var[130]=neighb_cutoff; var[131]=Ya;
  var[132]=Yc; var[133]=Yh; var[138]=delta_rx;
  var[165]=Stress; var[194]=frx; var[195]=fry; var[196]=frz;
}
```

stress.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime_api.h>
#include <math.h>
#include "stress.h"
#include "consts.cuh"
using namespace std;
__global__ void stress(float **var, int **intVar){
  int mi=threadIdx.x+blockDim.x * blockIdx.x;
  if (mi >= nfib) return;
  float *rx=var[3]; float *ry=var[4];
  float *rz=var[5]; float *px=var[18];
  float *py=var[19]; float *pz=var[20];
  float *wx=var[33]; float *wy=var[34];
  float *wz=var[35]; float *fcx=var[66];
  float *fcy=var[67]; float *fcz=var[68];
  float *Stress=var[165];
  float Stress11, Stress12, Stress13;
  float Stress22, Stress23, Stress33;
  float p1, p2;
  float e10, e11, e12;
  float e20, e21, e22;
  float prefactor, dotProd;
  float pxmi, pymi, pzmi;
  float rxmi, rymi, rzmi;
  float wxmi, wymi, wzmi;
```

```
float fcxmi, fcymi, fczmi;
int tid;
__shared__ float StressShared[6];
tid=threadIdx.x;
if (tid < 6){
  StressShared[tid]=0.0;
}
__syncthreads();
e10=0.0; e11=0.0; e12=1.0;
e20=duxdz; e21=0.0; e22=0.0;
rxmi=rx[mi]; rymi=ry[mi]; rzmi=rz[mi];
pxmi=px[mi]; pymi=py[mi]; pzmi=pz[mi];
wxmi=wx[mi]; wymi=wy[mi]; wzmi=wz[mi];
p1=pxmi*e10+pymi*e11+pzmi*e12;
p2=pxmi*e20+pymi*e21+pzmi*e22;
prefactor=-9.0*log(2.0*rp)/(2.0*rp*rp);
Stress11=0.0; Stress12=0.0; Stress13=0.0;
Stress22=0.0; Stress23=0.0; Stress33=0.0;
fcxmi=fcx[mi]; fcymi=fcy[mi]; fczmi=fcz[mi];
dotProd=rxmi*fcxmi+rymi*fcymi+rzmi*fczmi;
// Extra particle stress
Stress11=prefactor*(0.5*(rxmi*fcxmi+rxmi*fcxmi)
  -1.0/3.0*dotProd)
  + p1*(e20*pxmi+pxmi*e20) - p1*p2*pxmi*pxmi
  - (wymi*pzmi-wzmi*pymi)*pxmi
  - pxmi*(wymi*pzmi-wzmi*pymi);
Stress12=prefactor*0.5*(rxmi*fcymi+rymi*fcxmi)
  + p1*(e20*pymi+pxmi*e21) - p1*p2*pxmi*pymi
  - (wymi*pzmi-wzmi*pymi)*pymi
  - pxmi*(wzmi*pxmi-wxmi*pzmi);
Stress13= prefactor*0.5*(rxmi*fczmi+rzmi*fcxmi)
  + p1*(e20*pzmi+pxmi*e22) - p1*p2*pxmi*pzmi
  - (wymi*pzmi-wzmi*pymi)*pzmi
  - pxmi*(wxmi*pymi-wymi*pxmi);
Stress22=prefactor*(0.5*(rymi*fcymi+rymi*fcymi)
  -1.0/3.0*dotProd)
  + p1*(e21*pymi+pymi*e21) - p1*p2*pymi*pymi
  - (wzmi*pxmi-wxmi*pzmi)*pymi
  - pymi*(wzmi*pxmi-wxmi*pzmi);
Stress23=prefactor*0.5*(rymi*fczmi+rzmi*fcymi)
  + p1*(e21*pzmi+pymi*e22) - p1*p2*pymi*pzmi
  - (wzmi*pxmi-wxmi*pzmi)*pzmi
  - pymi*(wxmi*pymi-wymi*pxmi);
Stress33=prefactor*(0.5*(rzmi*fczmi+rzmi*fczmi)
  -1.0/3.0*dotProd)
  + p1*(e22*pzmi+pzmi*e22) - p1*p2*pzmi*pzmi
  - (wxmi*pymi-wymi*pxmi)*pzmi
  - pzmi*(wxmi*pymi-wymi*pxmi);
Stress11 /= sidex*sidey*sidez;
Stress12 /= sidex*sidey*sidez;
Stress13 /= sidex*sidey*sidez;
Stress22 /= sidex*sidey*sidez;
Stress23 /= sidex*sidey*sidez;
  Stress33 /= sidex*sidey*sidez;
  atomicAdd(StressShared+0, Stress11);
  atomicAdd(StressShared+1, Stress12);
  atomicAdd(StressShared+2, Stress13);
  atomicAdd(StressShared+3, Stress22);
  atomicAdd(StressShared+4, Stress23);
  atomicAdd(StressShared+5, Stress33);
  __syncthreads();
  if (tid == 0){
    atomicAdd(Stress+0, StressShared[0]);
    atomicAdd(Stress+1, StressShared[1]);
    atomicAdd(Stress+2, StressShared[2]);
    atomicAdd(Stress+3, StressShared[3]);
    atomicAdd(Stress+4, StressShared[4]);
    atomicAdd(Stress+5, StressShared[5]);
  }
}
```

updateBod.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime_api.h>
#include "updateBod.h"
#include "consts.cuh"
using namespace std;
__global__ void updateBod(float **var, int **intVar){
  float *px = var[18];
  float *py = var[19];
  float *pz = var[20];
  float *tbx = var[75];
  float *tby = var[76];
  int mi = threadIdx.x + blockDim.x * blockIdx.x;
  if (mi >= nfib) return;
  tbx[mi] = elf*pz[mi]*py[mi];
  tby[mi] = -elf*pz[mi]*px[mi];
}
```

updatePos.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime_api.h>
#include <math.h>
#include "updatePos.h"
#include "consts.cuh"
using namespace std;
__global__ void updatePos(float **var,int **intVar){
  int mi=threadIdx.x + blockDim.x*blockIdx.x;
  if (mi >= nfib) return;
  int step=*intVar[9];
```

```
float delta_rx=*var[138];
float *rcmx=var[0]; float *rcmy=var[1];
float *rcmz=var[2]; float *rx=var[3];
float *ry=var[4]; float *rz=var[5];
float *q0=var[6]; float *q1=var[7];
float *q2=var[8]; float *q3=var[9];
float *q0dot=var[10]; float *q1dot=var[11];
float *q2dot=var[12]; float *q3dot=var[13];
float *qe0=var[14]; float *qe1=var[15];
float *qe2=var[16]; float *qe3=var[17];
float *px=var[18]; float *py=var[19];
float *pz=var[20]; float *ucmx=var[21];
float *ucmy=var[22]; float *ucmz=var[23];
float *ucox=var[24]; float *ucoy=var[25];
float *ucoz=var[26]; float *ux=var[27];
float *uy=var[28]; float *uz=var[29];
float *wx=var[33]; float *wy=var[34];
float *wz=var[35]; float *R11=var[36];
float *R12=var[37]; float *R13=var[38];
float *R21=var[39]; float *R22=var[40];
float *R23=var[41];
float wbx,wby,wbz,dum;
float q0mi,q1mi,q2mi,q3mi;
float q0dotmi,q1dotmi,q2dotmi,q3dotmi;
float pxmi,pymi,pzmi,wxmi,wymi,wzmi;
// wbx,wby,wbz-body frame coordinates of the ang. velocity
float rcmxm,rcmym,rcmzm;
float corx,cory,corz;
// Find the time derivative of the Euler parameters
wxmi=wx[mi]; wymi=wy[mi]; wzmi=wz[mi];
wbx=R11[mi]*wxmi + R12[mi]*wymi + R13[mi]*wzmi;
wby=R21[mi]*wxmi + R22[mi]*wymi + R23[mi]*wzmi;
wbz=px[mi]*wxmi + py[mi]*wymi + pz[mi]*wzmi;
q0mi=q0[mi]; q1mi=q1[mi]; q2mi=q2[mi]; q3mi=q3[mi];
q0dotmi=0.5*(-wbx*q1mi-wby*q2mi-wbz*q3mi);
q1dotmi=0.5*(wbx*q0mi + wbz*q2mi-wby*q3mi);
q2dotmi=0.5*(wby*q0mi-wbz*q1mi + wbx*q3mi);
q3dotmi=0.5*(wbz*q0mi + wby*q1mi-wbx*q2mi);
q0dot[mi]=q0dotmi; q1dot[mi]=q1dotmi;
q2dot[mi]=q2dotmi; q3dot[mi]=q3dotmi;
// Integrate the Euler parameters (Adams-Bashforth)
if (step == 0){
  q0mi += q0dotmi*dt;
  q1mi += q1dotmi*dt;
  q2mi += q2dotmi*dt;
  q3mi += q3dotmi*dt;
}
else{
  q0mi += (1.5*q0dotmi-0.5*qe0[mi])*dt;
  q1mi += (1.5*q1dotmi-0.5*qe1[mi])*dt;
  q2mi += (1.5*q2dotmi-0.5*qe2[mi])*dt;
  q3mi += (1.5*q3dotmi-0.5*qe3[mi])*dt;
}
qe0[mi]=q0dotmi; qe1[mi]=q1dotmi;
qe2[mi]=q2dotmi; qe3[mi]=q3dotmi;
dum=sqrtf(q0mi*q0mi + q1mi*q1mi + q2mi*q2mi + q3mi*q3mi);
q0mi /= dum; q1mi /= dum;
q2mi /= dum; q3mi /= dum;
q0[mi]=q0mi; q1[mi]=q1mi; q2[mi]=q2mi; q3[mi]=q3mi;
// Calculate the new rotation matrix and p-vectors
R11[mi]=2.0*(q0mi*q0mi + q1mi*q1mi)-1.0;
R12[mi]=2.0*(q1mi*q2mi + q0mi*q3mi);
R13[mi]=2.0*(q1mi*q3mi-q0mi*q2mi);
R21[mi]=2.0*(q1mi*q2mi-q0mi*q3mi);
R22[mi]=2.0*(q0mi*q0mi + q2mi*q2mi)-1.0;
R23[mi]=2.0*(q3mi*q2mi + q0mi*q1mi);
pxmi=2.0*(q1mi*q3mi + q0mi*q2mi);
pymi=2.0*(q3mi*q2mi-q0mi*q1mi);
pzmi=2.0*(q0mi*q0mi + q3mi*q3mi)-1.0;
dum=sqrtf(pxmi*pxmi + pymi*pymi + pzmi*pzmi);
pxmi /= dum; pymi /= dum; pzmi /= dum;
px[mi]=pxmi; py[mi]=pymi; pz[mi]=pzmi;
// the following part is combined to this kernel
// because there is only one segment
rcmxm=rcmx[mi]; rcmym=rcmy[mi]; rcmzm=rcmz[mi];
ucmx[mi]=ux[mi]; ucmy[mi]=uy[mi]; ucmz[mi]=uz[mi];
if (step == 0){
  rcmxm += ucmx[mi]*dt;
  rcmym += ucmy[mi]*dt;
  rcmzm += ucmz[mi]*dt;
}
else{
  rcmxm += (1.5*ucmx[mi]-0.5*ucox[mi])*dt;
  rcmym += (1.5*ucmy[mi]-0.5*ucoy[mi])*dt;
  rcmzm += (1.5*ucmz[mi]-0.5*ucoz[mi])*dt;
}
ucox[mi]=ucmx[mi]; ucoy[mi]=ucmy[mi]; ucoz[mi]=ucmz[mi];
// Check periodic boundaries for a fiber leaving the box
cory=roundf(rcmym/sidey);
corz=roundf(rcmzm/sidez);
rcmxm -= corz*delta_rx;
corx=roundf(rcmxm/sidex);
rcmxm -= corx*sidex;
rcmym -= cory*sidey;
rcmzm -= corz*sidez;
rcmx[mi]=rcmxm; rcmy[mi]=rcmym; rcmz[mi]=rcmzm;
rx[mi]=rcmxm; ry[mi]=rcmym; rz[mi]=rcmzm;
// Check if the time step was too big
if (rcmxm*rcmxm > sidex*sidex ||
  rcmym*rcmym > sidey*sidey ||
  rcmzm*rcmzm > sidez*sidez){
  // or output to a file
  printf("time step too big\n");
  printf("rcmx[%5d]=%10f\n",mi,rcmxm);
  printf("rcmy[%5d]=%10f\n",mi,rcmym);
  printf("rcmz[%5d]=%10f\n",mi,rcmzm);
```

```
  }
}
```

updateVel.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime_api.h>
#include <math.h>
#include "updateVel.h"
#include "consts.cuh"
using namespace std;
__global__ void updateVel(float **var,int **intVar,float *randN){
  int mi=threadIdx.x+blockDim.x * blockIdx.x;
  if (mi >= nfib) return;
  float *px=var[18]; float *py=var[19];
  float *pz=var[20]; float *ux=var[27];
  float *uy=var[28]; float *uz=var[29];
  float *uxfl=var[30]; float *uyfl=var[31];
  float *uzfl=var[32]; float *wx=var[33];
  float *wy=var[34]; float *wz=var[35];
  float *fcx=var[66]; float *fcy=var[67];
  float *fcz=var[68]; float *tcx=var[69];
  float *tcy=var[70]; float *tcz=var[71];
  float *fbx=var[72]; float *fby=var[73];
  float *fbz=var[74]; float *tbx=var[75];
  float *tby=var[76]; float *tbz=var[77];
  float *D1=var[78]; float *D2=var[79];
  float *D3=var[80]; float *A11=var[81];
  float *A12=var[82]; float *A13=var[83];
  float *A23=var[84]; float *A22=var[85];
  float *A33=var[86]; float *C11=var[87];
  float *C12=var[88]; float *C13=var[89];
  float *C23=var[90]; float *C22=var[91];
  float *C33=var[92];
  int offset;
  float fxSum,fySum,fzSum,xSum,ySum,zSum;
  float r1,r2,r3,t1,t2,t3; // random vector
  float a11,a12,a13,a21,a22,a23,a31,a32,a33;
  float c11,c12,c13,c21,c22,c23,c31,c32,c33;
  float pxmi,pymi,pzmi;
  float Frx,Fry,Frz,Trx,Try,Trz;
  offset=gridDim.x * blockDim.x;
  pxmi=px[mi]; pymi=py[mi]; pzmi=pz[mi];
  // normalize random vectors
  r1=randN[mi];
  r2=randN[mi+offset];
  r3=randN[mi+2*offset];
  t1=randN[mi+3*offset];
  t2=randN[mi+4*offset];
  t3=randN[mi+5*offset];
  a11=C7+C9*pxmi*pxmi;
  a12=C9*pxmi*pymi;
  a13=C9*pxmi*pzmi;
  a21=C9*pymi*pxmi;
  a22=C7+C9*pymi*pymi;
  a23=C9*pymi*pzmi;
  a31=C9*pzmi*pxmi;
  a32=C9*pzmi*pymi;
  a33=C7+C9*pzmi*pzmi;
  c11=C8+C10*pxmi*pxmi;
  c12=C10*pxmi*pymi;
  c13=C10*pxmi*pzmi;
  c21=C10*pymi*pxmi;
  c22=C8+C10*pymi*pymi;
  c23=C10*pymi*pzmi;
  c31=C10*pzmi*pxmi;
  c32=C10*pzmi*pymi;
  c33=C8+C10*pzmi*pzmi;
  // random force and torque
  Frx=C5*(a11*r1+a12*r2+a13*r3);
  Fry=C5*(a21*r1+a22*r2+a23*r3);
  Frz=C5*(a31*r1+a32*r2+a33*r3);
  Trx=C6*(c11*t1+c12*t2+c13*t3);
  Try=C6*(c21*t1+c22*t2+c23*t3);
  Trz=C6*(c31*t1+c32*t2+c33*t3);
  // sum up forces
  fxSum=fcx[mi]+fbx[mi]+Frx;
  fySum=fcy[mi]+fby[mi]+Fry;
  fzSum=fcz[mi]+fbz[mi]+Frz;
  xSum=D1[mi]+tbx[mi]+C1*tcx[mi]+Trx;
  ySum=D2[mi]+tby[mi]+C1*tcy[mi]+Try;
  zSum=D3[mi]+tbz[mi]+C1*tcz[mi]+Trz;
  // update velocity
  ux[mi]=uxfl[mi]+A11[mi]*fxSum+A12[mi]*fySum+A13[mi]*fzSum;
  uy[mi]=uyfl[mi]+A12[mi]*fxSum+A22[mi]*fySum+A23[mi]*fzSum;
  uz[mi]=uzfl[mi]+A13[mi]*fxSum+A23[mi]*fySum+A33[mi]*fzSum;
  wx[mi]=Omega_x+C11[mi]*xSum+C12[mi]*ySum+C13[mi]*zSum;
  wy[mi]=Omega_y+C12[mi]*xSum+C22[mi]*ySum+C23[mi]*zSum;
  wz[mi]=Omega_z+C13[mi]*xSum+C23[mi]*ySum+C33[mi]*zSum;
}
```

zeroVar.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime_api.h>
#include "zeroVar.h"
#include "consts.cuh"
using namespace std;
__global__ void zeroVar(float **var, int **intVar){
  int tid=threadIdx.x + blockDim.x * blockIdx.x;
  if (tid >= nfib) return;
  float *fcx=var[66];
  float *fcy=var[67];
```

```
  float *fcz=var[68];
  float *tcx=var[69];
  float *tcy=var[70];
  float *tcz=var[71];
  float *Stress=var[165];
  fcx[tid]=0.0; fcy[tid]=0.0;
  fcz[tid]=0.0; tcx[tid]=0.0;
  tcy[tid]=0.0; tcz[tid]=0.0;
  if (tid < 6){
    Stress[tid]=0.0;
  }
}
```

consts.cuh

```
extern __device__ __constant__ int nfib, npcn, fac;
extern __device__ __constant__ float E11, E12, E13, E22, E23, E33;
extern __device__ __constant__ float duxdx, duxdy, duxdz;
extern __device__ __constant__ float duydx, duydy, duydz;
extern __device__ __constant__ float duzdx, duzdy, duzdz;
extern __device__ __constant__ float dx, dy, dz, dt, elf;
extern __device__ __constant__ float C0, C2, C3, C4;
extern __device__ __constant__ float C1, C5, C6, C7, C8, C9, C10;
extern __device__ __constant__ float Omega_x, Omega_y, Omega_z;
extern __device__ __constant__ float contact_cutoff, over_cut,
    rep_cutoff;
extern __device__ __constant__ float rp, sidex, sidey, sidez;
extern __device__ __constant__ float fstar, fact, Astar, decatt;
```

contact.h

```
#ifndef __flexfric__contact__
#define __flexfric__contact__
#include <stdio.h>
#include <cuda_runtime_api.h>
#include <cublas_v2.h>
__global__ void contact(float **var, int **intVar);
#endif /* defined(__flexfric__contact__) */
```

hydroRes.h

```
#ifndef __flexfric__hydroRes__
#define __flexfric__hydroRes__
#include <stdio.h>
#include <cuda_runtime_api.h>
__global__ void hydroRes(float **var, int **intVar);
#endif /* defined(__flexfric__hydroRes__) */
```

initialize.h

```
#ifndef __flexfric__initialize__
#define __flexfric__initialize__
#include <stdio.h>
#include <cuda_runtime_api.h>
#include <curand_kernel.h>
__global__ void initialize(float **var, int **intVar);
#endif /* defined(__flexfric__initialize__) */
```

lees_edwards.h

```
#ifndef __flexfric__lees_edwards__
#define __flexfric__lees_edwards__
#include <stdio.h>
#include <cuda_runtime_api.h>
__global__ void lees_edwards(float **var, int **intVar);
#endif /* defined(__flexfric__lees_edwards__) */
```

neighbor_list.h

```
#ifndef __flexfric__neighbor_list__
#define __flexfric__neighbor_list__
#include <stdio.h>
#include <cuda_runtime_api.h>
__global__ void neighbor_list(float **var, int **intVar, float
    neighbor_cutoff);
#endif /* defined(__flexfric__neighbor_list__) */
```

setVarArray.h

```
#ifndef __flexfric__setVarArray__
#define __flexfric__setVarArray__
#include <stdio.h>
#include <cuda_runtime_api.h>
__global__ void setVarArray(float **var, int **intVar,
       int *nseg, int *step, int *potCon, int *potConSize,
       float *rcmx, float *rcmy, float *rcmz,
       float *rx, float *ry, float *rz, float *q0, float *q1, float *
            q2, float *q3, float *q0dot,
       float *q1dot, float *q2dot, float *q3dot, float *qe0, float *
            qe1, float *qe2, float *qe3,
       float *px, float *py, float *pz, float *ucmx, float *ucmy,
            float *ucmz, float *ucox, float *ucoy,
       float *ucoz, float *ux, float *uy, float *uz, float *uxfl,
            float *uyfl, float *uzfl, float *wx,
       float *wy, float *wz, float *R11, float *R12, float *R13, float
             *R21, float *R22, float *R23,
       float *fcx, float *fcy, float *fcz, float *tcx, float *tcy,
            float *tcz, float *fbx,
       float *fby, float *fbz, float *tbx, float *tby, float *tbz,
            float *D1, float *D2, float *D3,
```

```
       float *A11, float *A12, float *A13, float *A23, float *A22,
            float *A33, float *C11, float *C12,
       float *C13, float *C23, float *C22, float *C33,
       float *neighb_cutoff, float *Ya, float *Yc, float *Yh,
       float *delta_rx, float *Stress,
       float *frx, float *fry, float *frz);
#endif /* defined(__flexfric__setVarArray__) */
```

stress.h

```
#ifndef __flexfric__stress__
#define __flexfric__stress__
#include <stdio.h>
#include <cuda_runtime_api.h>
__global__ void stress(float **var, int **intVar);
#endif /* defined(__flexfric__stress__) */
```

updateBod.h

```
#ifndef __flexfric__updateBod__
#define __flexfric__updateBod__
#include <stdio.h>
#include <cuda_runtime_api.h>
__global__ void updateBod(float **var, int **intVar);
#endif /* defined(__flexfric__updateBod__) */
```

updatePos.h

```
#ifndef __flexfric__updatePos__
#define __flexfric__updatePos__
#include <stdio.h>
#include <cuda_runtime_api.h>
__global__ void updatePos(float **var, int **intVar);
#endif /* defined(__flexfric__updatePos__) */
```

updateVel.h

```
#ifndef __flexfric__updateVel__
#define __flexfric__updateVel__
#include <stdio.h>
#include <cuda_runtime_api.h>
#include <curand_kernel.h>
__global__ void updateVel(float **var, int **intVar, float *randN);
#endif /* defined(__flexfric__updateVel__) */
```

zeroVar.h

```
#ifndef __flexfric__zeroVar__
#define __flexfric__zeroVar__
#include <stdio.h>
#include <cuda_runtime_api.h>
__global__ void zeroVar(float **var, int **intVar);
#endif /* defined(__flexfric__zeroVar__) */
```

CMakeLists.txt

```
cmake_minimum_required (VERSION 3.8 FATAL_ERROR)
set(CMAKE_CUDA_HOST_COMPILER $ENV{CU_CCBIN} CACHE PATH "Cuda host
    compiler dir")
set(CMAKE_CUDA_FLAGS ${CMAKE_CUDA_FLAGS} "-rdc=true")

project(brownfric LANGUAGES C CUDA)

add_executable(brownfric main.cu contact.cu hydroRes.cu
               zeroVar.cu initialize.cu lees_edwards.cu updateBod.cu
               updatePos.cu setVarArray.cu stress.cu updateVel.cu
               neighbor_list.cu)

# For linking ...
# Specify target & libraries to link it with
target_link_libraries( brownfric -lcurand -lm )
```

Parameters.in

```
32  :nfib, Number of fibers
1  :nseg, number of segments in each fiber
14  :rp, aspect ratio of a segment
0.33   :contact_cutoff, cutoff for contacts.
0.66  :rep_cutoff, repulsive force cutoff
1.85  :overlap
1.00E-06  :dt, time step
100  :strain
146.52 146.52 146.52 :side, length of a side in the simulation box
0.7  :fraction_rp, effective aspect ratio factor
500000  :config_write, how often the configuration is written
500000  :contact_write, how often the configuration is written
1.500E+02  :fstar, prefactor for the force
20.0  :fact, exponential factor in force
0  :Astar
35.0  :decatt
0.0  :delta_rx
0.0  :duxdx
0.0  :duydx
0.0  :duzdx
0.0  :duxdy
0.0  :duydy
0.0  :duzdy
1.0  :duxdz
0.0  :duydz
```

```
0.0  :duzdz
1  :fac, flow field velocity factor
0.0  :elf, electric field factor
32  :fiberPerBlock, number of fiber in each block
5000 :stress_write, how often the stress is written
19  :randseed
5000 :orient_write, how often the order parameter is written
```

Physical_Parameters.in

```
1.00E-04 :ts, time scale s
298.15 :T, temperature K
3.5 :b, radius nm
0.001 :eta0, solvent viscosity in Pa s
```

Centers_of_Mass.in

```
 1 9.367852195389567E+00 2.483213600423184E+00 -5.077006546977908E+01
 2 6.062545895067976E+01 1.126986118560657E+01 -4.228234238438304E+00
 3 -5.895515814576298E+01 1.173482941780239E+01 2.004846310699359E+01
 4 8.283633328713478E+00 3.998867873350159E+01 -1.313328337093816E+01
 5 -2.535835313307122E+01 -1.333354242771864E+01 1.370566348683089E+01
 6 -4.148097715850919E+01 -2.797295240554959E+00 3.359858332531527E+01
 7 -7.200722003510224E+01 -7.111534753913061E+01 3.347009832786397E+01
 8 -6.350865206737070E+01 5.333777211757377E+01 -5.308772788072005E+01
 9 5.121762187628076E+01 1.023505046725273E+01 -4.329999508777634E+01
10 1.877829676570371E+01 7.357038569189612E+00 2.971877937024460E+01
11 1.048673908380792E+01 -5.434879046766088E+01 6.012709891442213E+00
12 -3.234975004909560E+01 6.139345864053816E+01 -5.842597156779841E+01
13 -5.251129695916549E+01 2.590076775334776E+01 -6.575053901089355E+01
14 -2.416915221109987E+01 2.164575709657743E+01 -5.187376867689193E+01
15 2.883812066914514E+01 1.261830391701311E+01 1.024485875107345E+00
16 -4.504214923376217E+01 1.423709008097649E+01 -6.587082059981303E+01
17 -1.203560148233548E+01 -4.162907169604674E+01 -3.899335250493139E
     +01
18 2.268357110548764E+01 3.629822324864566E+01 -1.056019470747557E+00
19 -3.712685518423096E+01 1.657433010097591E+00 -1.228585163338110E+01
20 -4.822468857178465E+01 3.028617176407948E+01 -7.183395354514941E+01
21 7.324649406222628E+01 -4.902088336765485E+01 -1.501605146069080E+01
22 -4.288875660924241E+01 3.762405471866951E+01 2.753314821124079E+00
23 -4.337553373340518E+01 -5.583053787130864E+01 -5.003714406078682E
     +01
24 1.641343845354393E+01 -5.335581151077524E+01 1.793545975293966E+00
25 1.351275306953117E+01 6.239407109169290E+01 2.212541019011289E+01
26 1.767604034800083E+01 1.775352596757934E+01 6.978264107320459E+01
27 -3.784098361412063E+01 -2.072670037938282E+01 1.064551734751090E+01
28 -2.738768839286641E+01 4.990752796808258E+01 6.986318518035114E+01
29 5.167453461974860E+01 -3.052173411995173E+01 4.294274686004967E+01
30 -1.221050143539905E+01 -1.226616622298955E+01 -3.661236622637138E
     +01
31 -4.281273756539450E+01 -4.095933831134812E+01 2.499760873071850E+01
32 6.038830017987635E+00 4.218243845416234E+01 -6.940145637821402E+00
```

Euler_Parameters.in

```
 1 1 9.376079918231921E-01 -2.075646060226267E-01 -2.723474543780942E
     -01 -6.029139316456939E-02
 2 1 6.171730720337827E-01 1.155670932115783E-01 7.649978835054101E-01
     -1.432476329830712E-01
 3 1 3.442133694510844E-01 5.625145599244140E-02 9.249354719076421E-01
     -1.511532427599518E-01
 4 1 4.034813776251057E-01 -2.266718955632482E-01 -7.728566881990075E
     -01 -4.341833359049751E-01
 5 1 6.803041161592106E-01 -9.184140226141458E-03 7.328056300075984E-01
      9.892913337776515E-03
 6 1 8.320683149738113E-01 5.414480772234824E-01 -1.009156268075302E-01
      6.566837255239012E-02
 7 1 1.257040277842942E-01 1.076928166789604E-02 -9.883887543748959E-01
      8.467697559786634E-02
 8 1 4.101722175075643E-01 1.796235358715743E-01 8.190498677926658E-01
     -3.586801519665356E-01
 9 1 7.888592248982641E-01 -5.855699206592316E-01 1.498092376681776E-01
      1.112033435708229E-01
10 1 3.919332398341809E-01 -1.846961512568854E-01 -8.152738915078146E
     -01 -3.841928539804979E-01
11 1 9.918087378988356E-01 -5.791048156658977E-02 1.136563403532962E
     -01 6.636252688093344E-03
12 1 6.691748653932571E-01 -1.418291929605098E-01 7.135929510000096E
     -01 1.512434455875183E-01
13 1 1.869694975243120E-01 -6.967883425328154E-02 -9.182009478770268E
     -01 -3.421904241359477E-01
14 1 5.460661229150648E-01 1.821822168490910E-01 -7.756631685776603E
     -01 2.587819123905770E-01
15 1 1.594369975378437E-01 2.430499406065587E-02 -9.756376711779788E
     -01 1.487287654028009E-01
16 1 7.594420981396069E-01 -5.964788681853942E-02 6.458457392975958E
     -01 5.072583368005133E-02
17 1 7.939435326941048E-01 3.354778994337732E-01 -4.670730928531490E
     -01 1.973600056174041E-01
18 1 7.742625526497322E-01 5.978872562578690E-02 6.281638983166177E-01
      -4.850695521299139E-02
19 1 1.104785845900591E-01 -6.394578654055398E-02 -8.583986602246086E
     -01 -4.968472188261496E-01
20 1 3.984095417826138E-01 -1.639643404862599E-01 -8.345240239752468E
     -01 -3.434460444868098E-01
21 1 4.810018257089818E-02 -3.662534807407864E-02 -7.941550504259652E
     -01 -6.047005144682311E-01
22 1 9.951003698198458E-01 -8.261292016676212E-02 5.413098863035838E
     -02 4.493937674927512E-03
23 1 6.997773390902797E-01 7.749823782567471E-03 7.142751863018358E-01
      -7.910383084562319E-03
24 1 2.004179254039366E-01 5.294381376190277E-02 -9.458332782624153E
     -01 2.498579946539636E-01
```

```
25 1 4.551836383940345E-01 1.382912823818994E-01 8.416083743824653E-01
      -2.556926293909209E-01
26 1 5.130582797997216E-01 2.307646398349346E-01 7.539942906475645E-01
      -3.391334430599256E-01
27 1 5.502549697226650E-01 -2.668812164583417E-01 -7.118847133336776E
     -01 -3.452738616215429E-01
28 1 1.598591276610887E-01 -2.428800849778875E-02 -9.756444102804878E
     -01 -1.482333856966279E-01
29 1 5.176906753415033E-01 1.641295955863227E-01 8.004132842826933E-01
      -2.537644869971385E-01
30 1 4.611710624553010E-01 -1.258657394138171E-03 -8.873070507606350E
     -01 -2.421694835675111E-03
31 1 6.738928731619214E-01 1.330460868280252E-01 7.129883863693892E-01
      -1.407646801711799E-01
32 1 5.650723332452512E-01 3.752458346329214E-01 6.120973100572965E-01
      -4.064736373081140E-01
```

c.2 Flexible Fiber Code

`main.cu` - This file contains the `main()` function and `orientCalc`, which calculates the order parameter of the suspension on the host. A boolean variable, `driedRehyd`, at the beginning of `main.cu`, indicates whether the simulation program is simulating never-dried or dried-and-rehydrated suspensions.

`bnei_set.cu` - The simulation box is divided into cubic bins. This file contains the `bnei_set` kernel, which stores the indices of all neighboring bins of each bin.

`boxSize.cu` - This file contains the `boxSize` kernel, which changes the simulation box side length and moves fiber affinely every `box_write` steps.

`cell.cu` - This file contains the `cell` kernel, which places fiber segments into bins based on positions.

`contact.cu` - This file contains the `contact` kernel, which calculates the separation between fiber pairs in the cell list and updates the normal forces. The kernel `parallel_sort_para` calculates the separation between fibers in parallel.

`csr_set.cu` - This file contains the `csr_set` kernel, which sets the matrix format in csr through arrays `csrRow` and `csrCol` to use for the solver, `cusolverSpXcsrqrAnalysisBatched`, for the $\boldsymbol{X}$ forces.

`cublas_free.cu` - This file contains the `cublas_free` kernel, which frees resources from using the `cublas` library.

`cublas_initialize.cu` - This file contains the `cublas_initialize` kernel, which allocates

resources for using the `cublas` library. The file also includes the `cublasGetErrorString` kernel, which interprets error messages from `cublas`.

`delta_twist_torque.cu` - This file contains the `delta_twist_torque` kernel, which calculates the restoring torque $\boldsymbol{Y}$.

`delta_twist_torque2.cu` - This file contains the `delta_twist_torque2` kernel, which calculates the inverse of the hydrodynamic resistance tensors and updates the ambient velocity.

`delta_twist_zero.cu` - This file contains the `delta_twist_zero` kernel, which zeroes the variables, such as the normal and friction forces, every time step.

`friction_3x3.cu` - This file contains the `friction_3x3` kernel, which solves for the friction forces, $\boldsymbol{F}^{\text{fric}}$, for contact group that only contain two fiber segments.

`friction_update.cu` - This file contains the `friction_update` kernel, which atomically sums up the friction force as it is solved. This kernel is called by the `lead_list` kernel described below.

`group.cu` - This file contains the `group` kernel, in which a leader fiber of a group generates a list containing pairs of fiber in direct contact. A leader of the group is chosen as the fiber with the smallest index and identified in the `lead` kernel described below.

`initialize.cu` - This file contains the `initialize` kernel, which initializes variables, such as orientations and rotation matrices, from input files.

`lead.cu` - This file contains the `lead` kernel, which generates lists of fibers in a group

from lists of fibers in direct contact. A leader of the group, the segment with the smallest index, is also identified in the process with the `status` tag.

`lead_list.cu` - This file contains the `lead_list` kernel, which solves for the friction forces $\boldsymbol{F}^{\text{fric}}$ for groups with more than 2 segments, and calls the `friction`, `Jmat`, and `Jdiag` device kernels to construct $\boldsymbol{\mathcal{R}}_F$ matrices. This file calls the `friction_update` kernel in `friction_update.cu`. Solvers from `cublas` library are called from within the kernel. This requires compilation with CUDA version less than 10.0.

`lees_edwards.cu` - This file contains the `lees_edwards` kernel, which updates variables for the Lees-Ewards boundary condition.

`link.cu` - This file contains the `link` kernel, which adds segments in neighboring bins to lists containing potential contacts `potCon`.

`regrowFib.cu` - This file contains the `regrowFib` kernel, which generates the centers of mass of all segments except the first of each fiber from the center of mass of the first segment and orientations of all segments. This kernel is launched with `nfib(nseg-1)` threads.

`regrowSeg.cu` - This file contains the `regrowSeg` kernel, which calculates the centers of mass of the first segment in each fiber from the segment orientations and fiber center of mass. This kernel is launched with `nfib` threads.

`rhs.cu` - This file contains the `rhs` kernel, which calculates commonly used constants used in the `lead_list` and `friction_3x3` kernels.

`setVarArray.cu` - This file contains the `setVarArray` kernel, which stores pointers to de-

vice variables into arrays of double pointers.

`stress.cu` - This file contains the `stress` kernel, which calculates the stress of the suspension every `stress_write` steps.

`total_contacts_zero.cu` - This file contains the `total_contacts_zero` kernel, which zeroes the total number of contacts and the number of overlaps every simulation step.

`updateBod.cu` - This file contains the `updateBod` kernel, which updates the segment body forces.

`updateCen.cu` - This file contains the `updateCen` kernel, which updates the fiber center of mass.

`updateOri.cu` - This file contains the `updateOri` kernel, which updates the rotation matrices and segment orientations.

`x_forces_presolve.cu` - This file contains the `x_forces_presolve` kernel, which calculates the linear and angular velocities from summing forces before solving for the $\boldsymbol{X}$ forces.

`x_forces_para.cu` - This file contains the `x_forces_para` kernel, which constructs the matrix $\boldsymbol{\mathcal{R}}_F$ and array $\boldsymbol{\mathcal{V}}_F$.

`x_forces_postsolve.cu` - This file contains the `x_forces_postsolve` kernel, which updates the linear and angular velocities after solving for the $\boldsymbol{X}$ forces.

`x_forces_udpate.cu` - This file contains the `x_forces_update` kernel, which updates the

X forces from results of calling the `cusolver` library.

`Parameters.in` - This is an input file that contains parameters of the simulation run.

`Centers_of_Mass.in` - This is an input file that contains the centers of mass of the fibers.

`Euler_Parameters.in` - This is an input file that contains the Euler parameters of the fiber segments.

`Equilibrium_Angles.in` - This is an input file that contains the equlibrium angles at each joint.

`box.gen` - This is an input file that contains parameters on the box side length during drying and rehydration.

`box.cpp` - This is a program that generates `Lbox.txt` used from `box.gen`.

`Lbox.txt` - This is an input file that contains the box side length during the simulation.

main.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <cmath>
#include <time.h>
#include <cuda.h>
#include <cusparse_v2.h>
#include <cusolverSp.h>
#include <cublas_v2.h>
#include "setVarArray.h"
#include "initialize.h"
#include "regrowSeg.h"
#include "regrowFib.h"
#include "csr_set.h"
#include "cublas_free.h"
#include "cublas_initialize.h"
#include "bnei_set.h"
#include "cell.h"
#include "link.h"
#include "lead.h"
#include "lead_list.h"
#include "delta_twist_zero.h"
#include "delta_twist_torque.h"
#include "delta_twist_torque2.h"
#include "x_forces_presolve.h"
#include "x_forces_para.h"
#include "x_forces_update.h"
#include "x_forces_postsolve.h"
#include "total_contacts_zero.h"
#include "updateOri.h"
#include "updateCen.h"
#include "updateBod.h"
#include "lees_edwards.h"
#include "contact.h"
#include "group.h"
#include "rhs.h"
#include "friction_3x3.h"
#include "stress.h"
#include "boxSize.h"
using namespace std;
#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) {
      getchar();
      exit(code);
    }
  }
}
int main(void) {
  const int npcn = 2500;
  bool driedRehyd = true;
  const int numVar = 166;
  const int numIntVar = 35;
  float **var;
  int **intVar;
  gpuErrchk(cudaMalloc((void**)&var,numVar*sizeof(float*)));
  gpuErrchk(cudaMalloc((void**)&intVar,numIntVar*sizeof(int*)));
  // npcn - number of potential contact
  // driedRehyd - true for drying and rehydration
  // - false for never-dried
  // numVar - number of float variables
  // numIntVar - number of integer variables
  // var - array of pointers to floating point variables
  // intVar - ... interger ...
  ////////////////////////////////////////
  // input and output files
  FILE *Parameters,*Centers_of_Mass,*Euler_Parameters,*
       Equilibrium_Angles;
  FILE *resist_func,*README,*center_mass;
  FILE *Number_of_Contacts,*Stress_tensor;
  FILE *pxfile,*pyfile,*pzfile,*rxfile,*ryfile,*rzfile;
  FILE *q0file,*q1file,*q2file,*q3file;
  FILE *wxfile,*wyfile,*wzfile,*uxfile,*uyfile,*uzfile;
  Parameters = fopen("Parameters.in","r");
  Centers_of_Mass = fopen("Centers_of_Mass.in","r");
  Euler_Parameters = fopen("Euler_Parameters.in","r");
  Equilibrium_Angles = fopen("Equilibrium_Angles.in","r");
  resist_func = fopen("resist_func.txt","w");
  README = fopen("README.txt","w");
  center_mass = fopen("center_mass.txt","wb");
  Number_of_Contacts = fopen("Number_of_Contacts.txt","w");
  Stress_tensor = fopen("Stress_tensor.txt","w");
  pxfile = fopen("px.txt","wb");
  pyfile = fopen("py.txt","wb");
  pzfile = fopen("pz.txt","wb");
  rxfile = fopen("rx.txt","wb");
  ryfile = fopen("ry.txt","wb");
  rzfile = fopen("rz.txt","wb");
  q0file = fopen("q0.txt","wb");
  q1file = fopen("q1.txt","wb");
  q2file = fopen("q2.txt","wb");
  q3file = fopen("q3.txt","wb");
  wxfile = fopen("wx.txt","wb");
  wyfile = fopen("wy.txt","wb");
  wzfile = fopen("wz.txt","wb");
  uxfile = fopen("ux.txt","wb");
  uyfile = fopen("uy.txt","wb");
  uzfile = fopen("uz.txt","wb");
  // variables used in Parameters.in
  int nfib,nseg,config_write,maxCon,maxGr,maxBin;
  int fac,bdimx,bdimy,bdimz,fiberPerBlock,contact_write;
```

```
float rp,kb,mu_stat,mu_kin;
float contact_cutoff,rep_cutoff,over_cut;
float dt,strain,sidex,sidey,sidez;
float fstar,fact,Astar,decatt,elf,delta_rx;
float duxdx,duydx,duzdx,duxdy,duydy;
float duzdy,duxdz,duydz,duzdz;
float fraction_rp,dx,dy,dz;
int nfibGrid,nfibBlock,blasGrid,blasBlock;
int stress_write;
// nfib - number of fibers
// nseg - number of segments
// config_write - number of time steps between configuration writes
// maxCon - max fiber contacting one fiber
// maxGr - max number of fibers in a group
// maxBin - max number of fibers in a bin
// fac - factor of fluid velocity
// bdimx,bdimy,bdimz - number of bins to look for contacts
// fiberPerBlock - number of fiber per block
// rp - segment aspect ratio
// kb - bending constant
// mu_stat/mu_kin - static/kinetic coefficient of friction
// contact_cutoff - cutoff for contacts
// rep_cutoff - cutoff distance for calculating repulsive forces
// over_cut - limit to the overlapping of two fibers
// dt,strain - time step,total time of run
// sidex,sidey,sidez - simulation box dimensions (x and y directions)
// fstar,fact - force prefactor,force expfonential factor (decay
    length)
// Astar - Prefactor of the attractive force
// decatt - decay length of the attractive force
// elf - electric field factor
// delta_rx - shift variable for sliding periodic images
// duxdx... - velocity gradient tensor
// fraction_rp - effective aspect ratio factor
// dx,dy,dz - size of cell
// nfibGrid,nfibBlock - grid and block size with total nfib threads
// blasGrid,blasBlock - grid and block dimension for kernels
    involving
// cublas functions
fscanf(Parameters,"%d",&nfib);
fscanf(Parameters,"%*[^\n]%d",&nseg);
fscanf(Parameters,"%*[^\n]%f",&rp);
fscanf(Parameters,"%*[^\n]%f",&kb);
fscanf(Parameters,"%*[^\n]%f",&mu_stat);
fscanf(Parameters,"%*[^\n]%f",&mu_kin);
fscanf(Parameters,"%*[^\n]%f",&contact_cutoff);
fscanf(Parameters,"%*[^\n]%f",&rep_cutoff);
fscanf(Parameters,"%*[^\n]%f",&over_cut);
fscanf(Parameters,"%*[^\n]%f",&dt);
fscanf(Parameters,"%*[^\n]%f",&strain);
fscanf(Parameters,"%*[^\n]%f",&sidex);
fscanf(Parameters," %f",&sidey);
fscanf(Parameters," %f",&sidez);
```

```
fscanf(Parameters,"%*[^\n]%f",&fraction_rp);
fscanf(Parameters,"%*[^\n]%d",&config_write);
fscanf(Parameters,"%*[^\n]%d",&contact_write);
fscanf(Parameters,"%*[^\n]%f",&fstar);
fscanf(Parameters,"%*[^\n]%f",&fact);
fscanf(Parameters,"%*[^\n]%f",&Astar);
fscanf(Parameters,"%*[^\n]%f",&decatt);
fscanf(Parameters,"%*[^\n]%f",&delta_rx);
fscanf(Parameters,"%*[^\n]%f",&duxdx);
fscanf(Parameters,"%*[^\n]%f",&duydx);
fscanf(Parameters,"%*[^\n]%f",&duzdx);
fscanf(Parameters,"%*[^\n]%f",&duxdy);
fscanf(Parameters,"%*[^\n]%f",&duydy);
fscanf(Parameters,"%*[^\n]%f",&duzdy);
fscanf(Parameters,"%*[^\n]%f",&duxdz);
fscanf(Parameters,"%*[^\n]%f",&duydz);
fscanf(Parameters,"%*[^\n]%f",&duzdz);
fscanf(Parameters,"%*[^\n]%d",&fac);
fscanf(Parameters,"%*[^\n]%f",&elf);
fscanf(Parameters,"%*[^\n]%f",&dx);
fscanf(Parameters,"%*[^\n]%f",&dy);
fscanf(Parameters,"%*[^\n]%f",&dz);
fscanf(Parameters,"%*[^\n]%d",&bdimx);
fscanf(Parameters,"%*[^\n]%d",&bdimy);
fscanf(Parameters,"%*[^\n]%d",&bdimz);
fscanf(Parameters,"%*[^\n]%d",&fiberPerBlock);
fscanf(Parameters,"%*[^\n]%d",&maxCon);
fscanf(Parameters,"%*[^\n]%d",&maxGr);
fscanf(Parameters,"%*[^\n]%d",&maxBin);
fscanf(Parameters,"%*[^\n]%d",&nfibGrid);
fscanf(Parameters,"%*[^\n]%d",&nfibBlock);
fscanf(Parameters,"%*[^\n]%d",&blasGrid);
fscanf(Parameters,"%*[^\n]%d",&blasBlock);
fscanf(Parameters,"%*[^\n]%d",&stress_write);
// variables used when driedRehyd is true
int box_write,nLbox,*d_box_write;
float *Lx,*Ly,*Lz,*dLx,*dLy,*dLz;
float *d_dx_fixed,*d_dy_fixed,*d_dz_fixed;
float *d_Lx,*d_Ly,*d_Lz,*d_dLx,*d_dLy,*d_dLz;
float afactor,dum;
// when simulating dried and rehydration
if (driedRehyd) {
  cudaMalloc((void**)&d_dx_fixed,sizeof(float));
  cudaMalloc((void**)&d_dy_fixed,sizeof(float));
  cudaMalloc((void**)&d_dz_fixed,sizeof(float));
  cudaMemcpy(d_dx_fixed,&dx,sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(d_dy_fixed,&dy,sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(d_dz_fixed,&dz,sizeof(float),cudaMemcpyHostToDevice);
  // read input parameters
  FILE *input;
  input = fopen("box.gen","r");
  fscanf(input,"%f",&dum);
  fscanf(input,"%*[^\n]%f",&dum);
```

```
  fscanf(input,"%*[^\n]%f",&dum);
  fscanf(input,"%*[^\n]%f",&afactor);
  fscanf(input,"%*[^\n]%d",&box_write);
  fclose(input);
  // read box info
  FILE *LboxFile;
  LboxFile = fopen("Lbox.txt","r");
  nLbox = strain/(dt *float(box_write))+1;
  Lx = (float*)malloc(nLbox*sizeof(float));
  Ly = (float*)malloc(nLbox*sizeof(float));
  Lz = (float*)malloc(nLbox*sizeof(float));
  dLx = (float*)malloc(nLbox*sizeof(float));
  dLy = (float*)malloc(nLbox*sizeof(float));
  dLz = (float*)malloc(nLbox*sizeof(float));
  for (int box = 0; box < nLbox; box++){
    fscanf(LboxFile,"%f %f %f %f %f %f %f",
      &dum,Lx+box,Ly+box,Lz+box,
      dLx+box,dLy+box,dLz+box);
  }
  fclose(LboxFile);
  cudaMalloc((void**)&d_box_write,sizeof(int));
  cudaMalloc((void**)&d_Lx,nLbox*sizeof(float));
  cudaMalloc((void**)&d_Ly,nLbox*sizeof(float));
  cudaMalloc((void**)&d_Lz,nLbox*sizeof(float));
  cudaMalloc((void**)&d_dLx,nLbox*sizeof(float));
  cudaMalloc((void**)&d_dLy,nLbox*sizeof(float));
  cudaMalloc((void**)&d_dLz,nLbox*sizeof(float));
  cudaMemcpy(d_box_write,&box_write,sizeof(float),
       cudaMemcpyHostToDevice);
  cudaMemcpy(d_Lx,Lx,nLbox*sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(d_Ly,Ly,nLbox*sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(d_Lz,Lz,nLbox*sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(d_dLx,dLx,nLbox*sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(d_dLy,dLy,nLbox*sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(d_dLz,dLz,nLbox*sizeof(float),cudaMemcpyHostToDevice);
}
// variables on the device
int *d_nfib,*d_nseg,*d_maxCon,*d_maxGr,*d_fac;
int *d_bdimx,*d_bdimy,*d_bdimz,*d_maxBin;
int *d_blasGrid,*d_blasBlock;
float *d_rp,*d_kb,*d_mu_stat,*d_mu_kin;
float *d_contact_cutoff,*d_rep_cutoff,*d_over_cut;
float *d_dt,*d_sidex,*d_sidey,*d_sidez;
float *d_fstar,*d_fact,*d_Astar,*d_decatt,*d_elf;
float *d_delta_rx,*d_duxdx,*d_duxdy,*d_duxdz,*d_duydx;
float *d_duydy,*d_duydz,*d_duzdx,*d_duzdy,*d_duzdz;
float *d_dx,*d_dy,*d_dz;
cudaMalloc((void**)&d_nfib,sizeof(int));
cudaMalloc((void**)&d_nseg,sizeof(int));
cudaMalloc((void**)&d_maxCon,sizeof(int));
cudaMalloc((void**)&d_maxGr,sizeof(int));
cudaMalloc((void**)&d_maxBin,sizeof(int));
cudaMalloc((void**)&d_fac,sizeof(int));
cudaMalloc((void**)&d_bdimx,sizeof(int));
cudaMalloc((void**)&d_bdimy,sizeof(int));
cudaMalloc((void**)&d_bdimz,sizeof(int));
cudaMalloc((void**)&d_blasGrid,sizeof(int));
cudaMalloc((void**)&d_blasBlock,sizeof(int));
cudaMalloc((void**)&d_rp,sizeof(float));
cudaMalloc((void**)&d_kb,sizeof(float));
cudaMalloc((void**)&d_mu_stat,sizeof(float));
cudaMalloc((void**)&d_mu_kin,sizeof(float));
cudaMalloc((void**)&d_contact_cutoff,sizeof(float));
cudaMalloc((void**)&d_rep_cutoff,sizeof(float));
cudaMalloc((void**)&d_over_cut,sizeof(float));
cudaMalloc((void**)&d_dt,sizeof(float));
cudaMalloc((void**)&d_sidex,sizeof(float));
cudaMalloc((void**)&d_sidey,sizeof(float));
cudaMalloc((void**)&d_sidez,sizeof(float));
cudaMalloc((void**)&d_fstar,sizeof(float));
cudaMalloc((void**)&d_fact,sizeof(float));
cudaMalloc((void**)&d_Astar,sizeof(float));
cudaMalloc((void**)&d_decatt,sizeof(float));
cudaMalloc((void**)&d_elf,sizeof(float));
cudaMalloc((void**)&d_delta_rx,sizeof(float));
cudaMalloc((void**)&d_duxdx,sizeof(float));
cudaMalloc((void**)&d_duxdy,sizeof(float));
cudaMalloc((void**)&d_duxdz,sizeof(float));
cudaMalloc((void**)&d_duydx,sizeof(float));
cudaMalloc((void**)&d_duydy,sizeof(float));
cudaMalloc((void**)&d_duydz,sizeof(float));
cudaMalloc((void**)&d_duzdx,sizeof(float));
cudaMalloc((void**)&d_duzdy,sizeof(float));
cudaMalloc((void**)&d_duzdz,sizeof(float));
cudaMalloc((void**)&d_dx,sizeof(float));
cudaMalloc((void**)&d_dy,sizeof(float));
cudaMalloc((void**)&d_dz,sizeof(float));
// variables on host
float pi = 3.14159265;
float *rcmx,*rcmy,*rcmz,*rx,*ry,*rz;
float *q0,*q1,*q2,*q3,*px,*py,*pz;
float *ux,*uy,*uz,*wx,*wy,*wz;
int *num_groups,*total_contacts;
cudaMallocHost((void**)&rcmx,nfib*sizeof(float));
cudaMallocHost((void**)&rcmy,nfib*sizeof(float));
cudaMallocHost((void**)&rcmz,nfib*sizeof(float));
cudaMallocHost((void**)&rx,nfib*nseg*sizeof(float));
cudaMallocHost((void**)&ry,nfib*nseg*sizeof(float));
cudaMallocHost((void**)&rz,nfib*nseg*sizeof(float));
cudaMallocHost((void**)&q0,nfib*nseg*sizeof(float));
cudaMallocHost((void**)&q1,nfib*nseg*sizeof(float));
cudaMallocHost((void**)&q2,nfib*nseg*sizeof(float));
cudaMallocHost((void**)&q3,nfib*nseg*sizeof(float));
cudaMallocHost((void**)&px,nfib*nseg*sizeof(float));
cudaMallocHost((void**)&py,nfib*nseg*sizeof(float));
cudaMallocHost((void**)&pz,nfib*nseg*sizeof(float));
```

```
cudaMallocHost((void**)&ux,nfib*nseg*sizeof(float));
cudaMallocHost((void**)&uy,nfib*nseg*sizeof(float));
cudaMallocHost((void**)&uz,nfib*nseg*sizeof(float));
cudaMallocHost((void**)&wx,nfib*nseg*sizeof(float));
cudaMallocHost((void**)&wy,nfib*nseg*sizeof(float));
cudaMallocHost((void**)&wz,nfib*nseg*sizeof(float));
cudaMallocHost((void**)&num_groups,sizeof(int));
cudaMallocHost((void**)&total_contacts,sizeof(int));
float C0,C1,C2,C3,C4;
float Omega_x,Omega_y,Omega_z,kt;
float E11,E12,E13,E22,E23,E33;
float Xa,Ya,Xc,Yc,Yh;
float *thetaeq = (float*)malloc(nfib*nseg*sizeof(float));
float *phieq = (float*)malloc(nfib*nseg*sizeof(float));
float re,ecc;
int time_steps,overs;
float nL3,volfrac,consistency;
// re - effective segment aspect ratio
// ecc - eccentricity
// time_steps - number of time steps
// num_groups - number of groups
// overs - number of overlapping contacts
// nL3 - dimensionless concentration ("n-L-cubed")
// volfrac - volume fraction
// consist - consistency = volfrac/2.6
// floating point variables on device
float *d_rcmx,*d_rcmy,*d_rcmz,*d_rx,*d_ry,*d_rz;
float *d_q0,*d_q1,*d_q2,*d_q3;
float *d_q0dot,*d_q1dot,*d_q2dot,*d_q3dot;
float *d_qe0,*d_qe1,*d_qe2,*d_qe3;
float *d_px,*d_py,*d_pz;
float *d_ucmx,*d_ucmy,*d_ucmz;
float *d_ucox,*d_ucoy,*d_ucoz;
float *d_ux,*d_uy,*d_uz;
float *d_uxfl,*d_uyfl,*d_uzfl;
float *d_wx,*d_wy,*d_wz;
// rcmx,rcmy,rcmz - fiber center of mass
// rx,ry,rz - segment centers
// q0,q1,q2,q3 - segment euler parameters
// q0dot... - time derivatives of Euler Parameters
// qe1,... - time derivatives of Euler Parameters from previous time
// px,py,pz - segment orientational vectors
// ucmx,ucmy,ucmz - fiber center of mass velocity
// ucox,ucoy,ucoz - fiber center of mass velocity from previous time
// ux,uy,uz - segment velocity
// uxfl,uyfl,uzfl - ambient flow field velocity
// wx,wy,wz - segment angular velocity
cudaMalloc((void**)&d_rcmx,nfib*sizeof(float));
cudaMalloc((void**)&d_rcmy,nfib*sizeof(float));
cudaMalloc((void**)&d_rcmz,nfib*sizeof(float));
cudaMalloc((void**)&d_rx,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_ry,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_rz,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_q0,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_q1,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_q2,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_q3,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_q0dot,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_q1dot,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_q2dot,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_q3dot,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_qe0,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_qe1,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_qe2,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_qe3,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_px,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_py,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_pz,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_ucmx,nfib*sizeof(float));
cudaMalloc((void**)&d_ucmy,nfib*sizeof(float));
cudaMalloc((void**)&d_ucmz,nfib*sizeof(float));
cudaMalloc((void**)&d_ucox,nfib*sizeof(float));
cudaMalloc((void**)&d_ucoy,nfib*sizeof(float));
cudaMalloc((void**)&d_ucoz,nfib*sizeof(float));
cudaMalloc((void**)&d_ux,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_uy,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_uz,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_uxfl,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_uyfl,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_uzfl,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_wx,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_wy,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_wz,nfib*nseg*sizeof(float));
float *d_R11,*d_R12,*d_R13,*d_R21,*d_R22,*d_R23;
float *d_R11eq,*d_R12eq,*d_R13eq,*d_R21eq,*d_R22eq,*d_R23eq;
float *d_R31eq,*d_R32eq,*d_R33eq;
float *d_Xx,*d_Xy,*d_Xz,*d_Yx,*d_Yy,*d_Yz,*d_Ybx,*d_Yby,*d_Ybz;
float *d_fx,*d_fy,*d_fz,*d_tx,*d_ty,*d_tz;
float *d_fcx,*d_fcy,*d_fcz,*d_tcx,*d_tcy,*d_tcz;
float *d_fbx,*d_fby,*d_fbz,*d_tbx,*d_tby,*d_tbz;
// R11,R12,... - segment rotation matrix
// R11eq,... - equilibrium rotation matrix
// Xx,Xy,Xz - intrafiber constraint forces
// Yx,Yy,Yz - restoring bending/twisting torques
// fx,fy,fz - frictional forces on segment
// tx,ty,tz - frictional torques on segment
// fbx,fby,fbz - segment body force (i.e. gravity,etc.)
// tbx,tby,tbz - segment body torque
// fcx,fcy,fcz - segment colloidal repulsive force
// tcx,tcy,tcz - segment colloidal repulsive torque
cudaMalloc((void**)&d_R11,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_R12,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_R13,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_R21,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_R22,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_R23,nfib*nseg*sizeof(float));
```

```
cudaMalloc((void**)&d_R11eq,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_R12eq,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_R13eq,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_R21eq,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_R22eq,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_R23eq,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_R31eq,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_R32eq,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_R33eq,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_Xx,(nfib*nseg+nfib)*sizeof(float));
cudaMalloc((void**)&d_Xy,(nfib*nseg+nfib)*sizeof(float));
cudaMalloc((void**)&d_Xz,(nfib*nseg+nfib)*sizeof(float));
cudaMalloc((void**)&d_Yx,(nfib*nseg+nfib)*sizeof(float));
cudaMalloc((void**)&d_Yy,(nfib*nseg+nfib)*sizeof(float));
cudaMalloc((void**)&d_Yz,(nfib*nseg+nfib)*sizeof(float));
cudaMalloc((void**)&d_Ybx,(nfib*nseg+nfib)*sizeof(float));
cudaMalloc((void**)&d_Yby,(nfib*nseg+nfib)*sizeof(float));
cudaMalloc((void**)&d_Ybz,(nfib*nseg+nfib)*sizeof(float));
cudaMalloc((void**)&d_fx,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_fy,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_fz,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_tx,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_ty,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_tz,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_fcx,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_fcy,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_fcz,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_tcx,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_tcy,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_tcz,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_fbx,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_fby,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_fbz,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_tbx,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_tby,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_tbz,nfib*nseg*sizeof(float));
float *d_D1,*d_D2,*d_D3;
float *d_A11,*d_A12,*d_A13,*d_A23,*d_A22,*d_A33;
float *d_C11,*d_C12,*d_C13,*d_C23,*d_C22,*d_C33;
float *d_C0,*d_C1,*d_C2,*d_C3,*d_C4;
float *d_g,*d_Omega_x,*d_Omega_y,*d_Omega_z;
// D1,D2,D3 - "Jeffery" term in hydrodynamic torque
// A11,... - A inverse term in notes
// C11,... - C inverse term in notes
// C0,C1... - constant groupings
// g - gap between fiber centers at contact point
// Omega_... - ambient flow field angular velocity
cudaMalloc((void**)&d_D1,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_D2,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_D3,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_A11,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_A12,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_A13,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_A23,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_A22,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_A33,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_C11,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_C12,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_C13,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_C23,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_C22,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_C33,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_C0,sizeof(float));
cudaMalloc((void**)&d_C1,sizeof(float));
cudaMalloc((void**)&d_C2,sizeof(float));
cudaMalloc((void**)&d_C3,sizeof(float));
cudaMalloc((void**)&d_C4,sizeof(float));
cudaMalloc((void**)&d_g,1*sizeof(float));
cudaMalloc((void**)&d_Omega_x,sizeof(float));
cudaMalloc((void**)&d_Omega_y,sizeof(float));
cudaMalloc((void**)&d_Omega_z,sizeof(float));
float *d_kt,*d_E11,*d_E12,*d_E13,*d_E22,*d_E23,*d_E33;
float *d_neighb_cutoff,*d_Ya,*d_Yc,*d_Yh;
float *A_fric,*P_fric,*d_thetaeq,*d_phieq;
// kt - twisting constant
// E11,E12,... - rate of strain tensor
// Xa,Ya,Xc,Yc,Yh - scalar resistance functions
// A_fric,P_fric - matrix and RHS for friction contact
// thetaeq/phieq - angles associated with the equil. configuration
cudaMalloc((void**)&d_kt,sizeof(float));
cudaMalloc((void**)&d_E11,sizeof(float));
cudaMalloc((void**)&d_E12,sizeof(float));
cudaMalloc((void**)&d_E13,sizeof(float));
cudaMalloc((void**)&d_E22,sizeof(float));
cudaMalloc((void**)&d_E23,sizeof(float));
cudaMalloc((void**)&d_E33,sizeof(float));
cudaMalloc((void**)&d_neighb_cutoff,sizeof(float));
cudaMalloc((void**)&d_Ya,sizeof(float));
cudaMalloc((void**)&d_Yc,sizeof(float));
cudaMalloc((void**)&d_Yh,sizeof(float));
cudaMalloc((void**)&A_fric,blasBlock*blasGrid*maxGr*maxGr*9*sizeof(
    float));
cudaMalloc((void**)&P_fric,blasBlock*blasGrid* maxGr*3*sizeof(float))
    ;
cudaMalloc((void**)&d_thetaeq,nfib*nseg*sizeof(float));
cudaMalloc((void**)&d_phieq,nfib*nseg*sizeof(float));
float *csrValTot,*VTot,*Vsol;
// csrValTot - stores value in csr format for all fibers
// VTot - stores value of rhs
// Vsol - stores solution
// Gij...,Gji... - vector from r to contact point
cudaMalloc((void**)&csrValTot,nfib*9*(4+3*(nseg-3))*sizeof(float));
cudaMalloc((void**)&VTot,nfib*3*(nseg-1)*sizeof(float));
cudaMalloc((void**)&Vsol,nfib*3*(nseg-1)*sizeof(float));
float *nxV,*nyV,*nzV,*gV;
float *GijxV,*GijyV,*GijzV;
```

```
float *GjixV,*GjiyV,*GjizV;
// matrices to store contact info for each pair of fibers in contact
cudaMalloc((void**)&nxV,nfib*nseg*maxCon*sizeof(float));
cudaMalloc((void**)&nyV,nfib*nseg*maxCon*sizeof(float));
cudaMalloc((void**)&nzV,nfib*nseg*maxCon*sizeof(float));
cudaMalloc((void**)&gV,nfib*nseg*maxCon*sizeof(float));
cudaMalloc((void**)&GijxV,nfib*nseg*maxCon*sizeof(float));
cudaMalloc((void**)&GijyV,nfib*nseg*maxCon*sizeof(float));
cudaMalloc((void**)&GijzV,nfib*nseg*maxCon*sizeof(float));
cudaMalloc((void**)&GjixV,nfib*nseg*maxCon*sizeof(float));
cudaMalloc((void**)&GjiyV,nfib*nseg*maxCon*sizeof(float));
cudaMalloc((void**)&GjizV,nfib*nseg*maxCon*sizeof(float));
float *aix,*aiy,*aiz,*bix,*biy,*biz;
// constants used to generate matrices for friction forces
cudaMalloc((void**)&aix,nfib*nseg*sizeof(float));
cudaMalloc((void**)&aiy,nfib*nseg*sizeof(float));
cudaMalloc((void**)&aiz,nfib*nseg*sizeof(float));
cudaMalloc((void**)&bix,nfib*nseg*sizeof(float));
cudaMalloc((void**)&biy,nfib*nseg*sizeof(float));
cudaMalloc((void**)&biz,nfib*nseg*sizeof(float));
// calculate bin parameters
int nxbin,nybin,nzbin;
nxbin = int(floorf(sidex/dx));
nybin = int(floorf(sidey/dy));
nzbin = int(floorf(sidez/dz));
printf("dx dy dz %10f %10f %10f\n",dx,dy,dz);
printf("nxbin nybin nzbin %4d %4d %4d\n",nxbin,nybin,nzbin);
// only use even number of bins in each dimension
if (nxbin % 2 != 0){
  nxbin--;
}
if (nybin % 2 != 0){
  nybin--;
}
if (nzbin % 2 != 0){
  nzbin--;
}
if (!driedRehyd){
  dx = sidex/float(nxbin);
  dy = sidey/float(nybin);
  dz = sidez/float(nzbin);
  printf("dx dy dz %10f %10f %10f\n",dx,dy,dz);
}
// integer variables on device
int *d_ifiber,*d_ncnt,*d_ncpf,*d_num_groups,*d_overs;
int *d_total_contacts,*d_step,*csrRow,*csrCol;
int *bin,*list,*bnei,*d_nxbin,*d_nybin,*d_nzbin;
int *clist,*status,*lead_clist,*nc,*clist_pos;
int *bnum,*potCon,*potConSize,*groupId;
// ifiber - index of fibers in contact
// ncnt - number of contacts in groups
// ncpf - number of fibers contacting fiber
// num_groups - number of groups in contact
// overs - number of overlapping contacts
// total_contacts - number of contacts in system
// step - time step
// csrRow - indices to first nonzero value in ith block row
// csrCol - col indices of corresponding value in bsrVal
// bin - number of fibers in bin[x]
// list - list of fibers in each bin
// bnei - neighbor of each bin
// nxbin - number of bins in x direction
// clist - list of fibers in contact
// status - marks whether a fiber is in charge of solving
// the system of contacts
// lead_clist - list of all fibers in contacting group
// nc - number of contacts in groups
// clist_pos - position of contact in clist
cudaMalloc((void**)&d_ifiber,nfib*nseg*2*maxGr*sizeof(int));
cudaMalloc((void**)&d_ncnt,nfib*nseg*sizeof(int));
cudaMalloc((void**)&d_ncpf,(nfib*nseg)*sizeof(int));
cudaMalloc((void**)&d_num_groups,sizeof(int));
cudaMalloc((void**)&d_overs,sizeof(int));
cudaMalloc((void**)&d_total_contacts,sizeof(int));
cudaMalloc((void**)&d_step,sizeof(int));
cudaMalloc((void**)&csrRow,(3*(nseg-1)+1)*sizeof(int));
cudaMalloc((void**)&csrCol,9*(4+3*(nseg-3))*sizeof(int));
cudaMalloc((void**)&bin,nxbin*nybin*nzbin*sizeof(int));
cudaMalloc((void**)&list,maxBin*nxbin*nybin*nzbin*sizeof(int));
cudaMalloc((void**)&bnei,bdimx*bdimy*bdimz*nxbin*nybin*nzbin*sizeof(
    int));
cudaMalloc((void**)&d_nxbin,sizeof(int));
cudaMalloc((void**)&d_nybin,sizeof(int));
cudaMalloc((void**)&d_nzbin,sizeof(int));
cudaMalloc((void**)&clist,nfib*nseg*maxCon*sizeof(int));
cudaMalloc((void**)&status,nfib*nseg*sizeof(int));
cudaMalloc((void**)&lead_clist,nfib*nseg*maxGr*sizeof(int));
cudaMalloc((void**)&nc,nfib*nseg*sizeof(int));
cudaMalloc((void**)&clist_pos,nfib*nseg*maxGr*sizeof(int));
cudaMalloc((void**)&bnum,nfib*nseg*sizeof(int));
cudaMalloc((void**)&potCon,nfib*nseg*npcn*sizeof(int));
cudaMalloc((void**)&potConSize,nfib*nseg*sizeof(int));
cudaMalloc((void**)&groupId,nfib*nseg*sizeof(int));
// recalculate once max arrays are allocated
if (driedRehyd) {
  sidex = Lx[0];
  sidey = Ly[0];
  sidez = Lz[0];
  nxbin = int(floorf(sidex/dx));
  nybin = int(floorf(sidey/dy));
  nzbin = int(floorf(sidez/dz));
  printf("dx dy dz %10f %10f %10f\n",dx,dy,dz);
  printf("nxbin nybin nzbin %4d %4d %4d\n",nxbin,nybin,nzbin);
  if (nxbin % 2 != 0){
    nxbin--;
  }
```

```
  if (nybin % 2 != 0){
    nybin--;
  }
  if (nzbin % 2 != 0){
    nzbin--;
  }
  dx = sidex/float(nxbin);
  dy = sidey/float(nybin);
  dz = sidez/float(nzbin);
}
// variables on device for library calls
float *Workspace,**d_AA,**d_PP;
int *d_Pivot,*d_info;
cublasHandle_t *handle_cublas;
cublasStatus_t *status_cublas;
cudaMalloc((void**)&Workspace,36*200*sizeof(int));
cudaMalloc((void**)&d_AA,blasBlock*blasGrid*sizeof(float*));
cudaMalloc((void**)&d_PP,blasBlock*blasGrid*sizeof(float*));
cudaMalloc((void**)&d_Pivot,blasBlock*blasGrid*maxGr*3*sizeof(int));
cudaMalloc((void**)&d_info,blasBlock*blasGrid*sizeof(int));
cudaMalloc((void**)&handle_cublas,blasBlock*blasGrid*sizeof(
     cublasHandle_t));
cudaMalloc((void**)&status_cublas,blasBlock*blasGrid*sizeof(
     cublasStatus_t));
// cusolver parameters
int batchSize = nfib;
int ma = 3*(nseg-1);
int nnz = 9*(4+3*(nseg-3));
// nnz - number of nonzero blocks in Ab and A
// ma - number of rows of A
// batchSize - number of systems solved
cusparseHandle_t handle_sparse;
cusolverSpHandle_t handle_solver;
cusparseMatDescr_t descrA = NULL;
cusolverStatus_t status_solver;
cusparseStatus_t status_sparse;
cudaStream_t streamId = 0;
void *pBuffer = NULL;
csrqrInfo_t info = NULL;
size_t internalDataInBytes,workspaceInBytes;
// cuda streams
cudaStream_t stream2,stream3,stream4,stream5,stream6;
cudaStreamCreate(&stream2);
cudaStreamCreate(&stream3);
cudaStreamCreate(&stream4);
cudaStreamCreate(&stream5);
cudaStreamCreate(&stream6);
// kernel launch parameters
if (nfib < fiberPerBlock) fiberPerBlock = nfib;
int numThreads = fiberPerBlock*nseg;
int numBlocks = (nfib*nseg)/numThreads;
// numThreads - number of threads per block
// numBlocks - number of blocks
```

```
// stress Storage
float *d_Stress,*Stress;
cudaMalloc((void**)&d_Stress,6*sizeof(float));
Stress = (float*)malloc(6*sizeof(float));
// temporary variables
int idum1,idum2,step,m,i,mi;
float tprint,t;
// Read in Centers of Mass,Euler Parameters,Equilibrium Angles
for (m = 0; m < nfib; m++){
  fscanf(Centers_of_Mass,"%d %f %f %f ",
    &idum1,rcmx+m,rcmy+m,rcmz+m);
  for (i = 0; i < nseg; i++){
    mi = m*nseg+i;
    fscanf(Euler_Parameters,"%d %d %f %f %f %f",
      &idum1,&idum2,q0+mi,q1+mi,q2+mi,q3+mi);
    if (i > 0){
      fscanf(Equilibrium_Angles,"%d %d %f %f",&idum1,&idum2,thetaeq+
           mi,phieq+mi);
    }
  }
}
fclose(Parameters); fclose(Centers_of_Mass);
fclose(Euler_Parameters); fclose(Equilibrium_Angles);
// initialize constants
elf = 0.0;
// Rate of strain tensor for the flow field
E11 = 0.5*(duxdx+duxdx);
E12 = 0.5*(duydx+duxdy);
E13 = 0.5*(duzdx+duxdz);
E22 = 0.5*(duydy+duydy);
E23 = 0.5*(duzdy+duydz);
E33 = 0.5*(duzdz+duzdz);
// Fluid vorticity vector
Omega_x = 0.5*(duzdy-duydz);
Omega_y = 0.5*(duxdz-duzdx);
Omega_z = 0.5*(duydx-duxdy);
// Resistance functions
re = fraction_rp*rp;
ecc = sqrtf(re*re-1.0)/re;
Xa = (8.0*powf(ecc,3.0)/3.0)/(-2.0*ecc+(1.0+ecc*ecc)*logf((1.0+ecc)
     /(1.0-ecc)));
Ya = (16.0*powf(ecc,3.0)/3.0)/(2.0*ecc+(3.0*ecc*ecc-1.0)*logf((1.0+
     ecc)/(1.0-ecc)));
Xc = (4.0*powf(ecc,3.0)/3.0)*(1.0-ecc*ecc)/(2.0*ecc-(1.0-ecc*ecc)*
     logf((1.0+ecc)/(1.0-ecc)));
Yc = (4.0*powf(ecc,3.0)/3.0)*(2.0-ecc*ecc)/(-2.0*ecc+(1.0+ecc*ecc)*
     logf((1.0+ecc)/(1.0-ecc)));
Yh = (4.0*powf(ecc,5.0)/3.0)/(-2.0*ecc+(1.0+ecc*ecc)*logf((1.0+ecc)
     /(1.0-ecc)));
fprintf(resist_func,"Xa = %.14E\nYa = %.14E\nXc = %.14E\nYc = %.14E\
     nYh = %.14E\n",Xa,Ya,Xc,Yc,Yh);
fclose(resist_func);
// Other constants //
```

```
time_steps = int(strain/dt); // number of time steps
kt = 0.67*kb; // twisting constant
C0 = 3.0/(4.0*rp);
C1 = 3.0/(4.0*rp*rp);
C2 = 3.0/(4.0*Yc);
C3 = 1.0/Xa-1.0/Ya;
C4 = 1.0/Xc-1.0/Yc;
contact_cutoff = powf((contact_cutoff+2.0),2.0);
rep_cutoff = powf((rep_cutoff+2.0),2.0);
nL3 = float(nfib)*powf((float(2.0)*float(nseg)*rp),3.0)/(sidex*sidey*
    sidez);
volfrac = float(nfib)*float(nseg)*pi*(float(2.0)*rp)/(sidex*sidey*
    sidez);
consistency = volfrac/float(2.6000);
// generate README
fprintf(README,"Number of Fibers: %d\n",nfib);
fprintf(README,"Number of Segments: %d\n",nseg);
fprintf(README,"Aspect Ratio of a Segment: %.15f\n",rp);
fprintf(README,"Aspect Ratio of a Fiber: %.15f\n",rp*nseg);
fprintf(README,"Time Step: %.15E\n",dt);
fprintf(README,"Total Strain: %.15E\n",strain);
fprintf(README,"Box Side X: %.15f\n",sidex);
fprintf(README,"Box Side Y: %.15f\n",sidey);
fprintf(README,"Box Side Z: %.15f\n",sidez);
fprintf(README,"Coefficients of Friction: %.15f %.15f\n",mu_stat,
    mu_kin);
fprintf(README,"Bending Constant: %.15f\n",kb);
fprintf(README,"Twisting Constant: %.15f\n",kt);
fprintf(README,"concentration,nL3: %.15f\n",nL3);
fprintf(README,"Volume fraction: %.15E\n",volfrac);
fprintf(README,"Consistency: %.15E\n",consistency);
if (driedRehyd) fprintf(README,"Max concentration: %.15E\n",afactor*
    volfrac);
fclose(README);
// copy memory to device
cudaMemcpy(d_blasGrid,&blasGrid,sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(d_blasBlock,&blasBlock,sizeof(int),cudaMemcpyHostToDevice)
    ;
cudaMemcpy(d_dx,&dx,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_dy,&dy,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_dz,&dz,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_nxbin,&nxbin,sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(d_nybin,&nybin,sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(d_nzbin,&nzbin,sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(d_bdimx,&bdimx,sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(d_bdimy,&bdimy,sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(d_bdimz,&bdimz,sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(d_maxCon,&maxCon,sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(d_maxGr,&maxGr,sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(d_maxBin,&maxBin,sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(d_nfib,&nfib,sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(d_nseg,&nseg,sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(d_rp,&rp,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_kb,&kb,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_mu_stat,&mu_stat,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_mu_kin,&mu_kin,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_over_cut,&over_cut,sizeof(float),cudaMemcpyHostToDevice)
    ;
cudaMemcpy(d_dt,&dt,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_sidex,&sidex,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_sidey,&sidey,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_sidez,&sidez,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_fstar,&fstar,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_fact,&fact,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_Astar,&Astar,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_decatt,&decatt,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_delta_rx,&delta_rx,sizeof(float),cudaMemcpyHostToDevice)
    ;
cudaMemcpy(d_duxdx,&duxdx,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_duxdy,&duxdy,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_duxdz,&duxdz,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_duydx,&duydx,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_duydy,&duydy,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_duydz,&duydz,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_duzdx,&duzdx,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_duzdy,&duzdy,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_duzdz,&duzdz,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_fac,&fac,sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(d_elf,&elf,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_E11,&E11,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_E12,&E12,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_E13,&E13,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_E22,&E22,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_E23,&E23,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_E33,&E33,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_Omega_x,&Omega_x,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_Omega_y,&Omega_y,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_Omega_z,&Omega_z,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_Ya,&Ya,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_Yc,&Yc,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_Yh,&Yh,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_kt,&kt,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_C0,&C0,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_C1,&C1,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_C2,&C2,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_C3,&C3,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_C4,&C4,sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_contact_cutoff,&contact_cutoff,sizeof(float),
    cudaMemcpyHostToDevice);
cudaMemcpy(d_rep_cutoff,&rep_cutoff,sizeof(float),
    cudaMemcpyHostToDevice);
cudaMemcpy(d_rcmx,rcmx,nfib*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_rcmy,rcmy,nfib*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_rcmz,rcmz,nfib*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_q0,q0,nfib*nseg*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_q1,q1,nfib*nseg*sizeof(float),cudaMemcpyHostToDevice);
```

```
cudaMemcpy(d_q2,q2,nfib*nseg*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_q3,q3,nfib*nseg*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_thetaeq,thetaeq,nfib*nseg*sizeof(float),
     cudaMemcpyHostToDevice);
cudaMemcpy(d_phieq,phieq,nfib*nseg*sizeof(float),
     cudaMemcpyHostToDevice);
// stores device pointers into double pointer arrays
setVarArray << <1,1 >> >(var,intVar,d_ifiber,d_ncnt,d_ncpf,
  d_num_groups,d_overs,d_total_contacts,d_nfib,d_nseg,d_fac,d_step,
       csrRow,csrCol,
  bin,list,bnei,d_nxbin,d_nybin,d_nzbin,
  clist,status,lead_clist,nc,d_bdimx,d_bdimy,d_bdimz,d_maxCon,
       clist_pos,d_maxGr,d_maxBin,
  d_blasGrid,d_blasBlock,bnum,potCon,potConSize,groupId,
  d_rcmx,d_rcmy,d_rcmz,d_rx,d_ry,d_rz,d_q0,d_q1,d_q2,d_q3,d_q0dot,
       d_q1dot,
  d_q2dot,d_q3dot,d_qe0,d_qe1,d_qe2,d_qe3,d_px,d_py,d_pz,d_ucmx,
       d_ucmy,
  d_ucmz,d_ucox,d_ucoy,d_ucoz,d_ux,d_uy,d_uz,d_uxfl,d_uyfl,d_uzfl,
       d_wx,d_wy,
  d_wz,d_R11,d_R12,d_R13,d_R21,d_R22,d_R23,d_R11eq,d_R12eq,d_R13eq,
       d_R21eq,
  d_R22eq,d_R23eq,d_R31eq,d_R32eq,d_R33eq,d_Xx,d_Xy,d_Xz,d_Yx,d_Yy,
       d_Yz,
  d_Ybx,d_Yby,d_Ybz,d_fx,d_fy,d_fz,d_tx,d_ty,d_tz,d_fcx,d_fcy,d_fcz,
  d_tcx,d_tcy,d_tcz,d_fbx,d_fby,d_fbz,d_tbx,d_tby,d_tbz,d_D1,d_D2,
       d_D3,
  d_A11,d_A12,d_A13,d_A23,d_A22,d_A33,d_C11,d_C12,d_C13,d_C23,d_C22,
       d_C33,
  d_C0,d_C1,d_C2,d_C3,d_C4,d_g,d_duxdx,d_duxdy,d_duxdz,d_duydx,
       d_duydy,
  d_duydz,d_duzdx,d_duzdy,d_duzdz,d_dx,d_dy,d_dz,
  d_Omega_x,d_Omega_y,d_Omega_z,d_rp,d_kb,
  d_kt,d_dt,d_over_cut,d_sidex,d_sidey,d_sidez,d_E11,d_E12,d_E13,
       d_E22,d_E23,
  d_E33,d_contact_cutoff,d_rep_cutoff,d_neighb_cutoff,d_Ya,d_Yc,d_Yh,
       A_fric,
  P_fric,d_thetaeq,d_phieq,d_delta_rx,d_mu_stat,d_mu_kin,csrValTot,
       VTot,Vsol,
  d_fstar,d_fact,d_Astar,d_decatt,d_elf,GijxV,GijyV,GijzV,GjixV,
  GjiyV,GjizV,nxV,nyV,nzV,gV,aix,aiy,aiz,bix,biy,biz,d_Stress);
gpuErrchk(cudaDeviceSynchronize());
// Initialization for cusolver
status_solver = cusolverSpCreate(&handle_solver);
if (status_solver != CUSOLVER_STATUS_SUCCESS) printf("[
     cusolverSpCreate status %d]\n",status_solver);
status_solver = cusolverSpSetStream(handle_solver,streamId);
if (status_solver != CUSOLVER_STATUS_SUCCESS) printf("[
     cusolverSpSetStream status %d]\n",status_solver);
status_sparse = cusparseCreate(&handle_sparse);
if (status_sparse != CUSPARSE_STATUS_SUCCESS) printf("cusparseCreate
     Error\n");
// 1. create a descriptor
cusparseCreateMatDescr(&descrA);
cusparseSetMatIndexBase(descrA,CUSPARSE_INDEX_BASE_ZERO);
cusparseSetMatType(descrA,CUSPARSE_MATRIX_TYPE_GENERAL);
// 2. create empty info structure
status_solver = cusolverSpCreateCsrqrInfo(&info);
if (status_solver != CUSOLVER_STATUS_SUCCESS) printf("[
     cusolverSpCreateCsrqrInfo status %d]\n",status_solver);
// set matrix format in csr-only when nseg > 1
if (nseg > 1) csr_set << <1,1 >> >(intVar);
gpuErrchk(cudaDeviceSynchronize());
if (cudaSuccess != cudaGetLastError()){ printf("after csr set!\n");
     getchar(); }
// 3. symbolic analysis
status_solver = cusolverSpXcsrqrAnalysisBatched(handle_solver,ma,ma,
     nnz,descrA,csrRow,csrCol,info);
if (status_solver != CUSOLVER_STATUS_SUCCESS) {
  printf("[cusolverSpXcsrqrAnalysisBatched status %d]\n",
       status_solver);
  getchar();
}
gpuErrchk(cudaDeviceSynchronize());
// 4. allocate working space
status_solver = cusolverSpScsrqrBufferInfoBatched(handle_solver,ma,ma
     ,nnz,descrA,csrValTot,csrRow,csrCol,batchSize,info,&
     internalDataInBytes,&workspaceInBytes);
if (status_solver != CUSOLVER_STATUS_SUCCESS){
  printf("[cusolverSpDcsrqrBufferInfoBatched status %d]\n",
       status_solver);
  getchar();
}
cudaMalloc(&pBuffer,workspaceInBytes);
gpuErrchk(cudaDeviceSynchronize());
// cublas constructors
cublas_initialize << <blasGrid,blasBlock >> >(handle_cublas);
// initialize variables
initialize << <numBlocks,numThreads >> >(var,intVar);
gpuErrchk(cudaPeekAtLastError());
gpuErrchk(cudaDeviceSynchronize());
// get position of first segment of each fiber
regrowSeg << <nfibGrid,nfibBlock >> >(var,intVar);
gpuErrchk(cudaDeviceSynchronize());
// get position of all segments except the first of each fiber
regrowFib << <nfib/fiberPerBlock,(nseg-1)*fiberPerBlock >> >(var,
     intVar);
gpuErrchk(cudaDeviceSynchronize());
// copy memory back to host for printing
step = 0; overs = 0; tprint = 0.0;
cudaMemcpy(rcmx,d_rcmx,nfib*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(rcmy,d_rcmy,nfib*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(rcmz,d_rcmz,nfib*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(rx,d_rx,nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(ry,d_ry,nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(rz,d_rz,nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
```

```
cudaMemcpy(px,d_px,nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(py,d_py,nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(pz,d_pz,nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(ux,d_ux,nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(uy,d_uy,nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(uz,d_uz,nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(wx,d_wx,nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(wy,d_wy,nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(wz,d_wz,nfib*nseg*sizeof(float),cudaMemcpyDeviceToHost);
fwrite(&tprint,sizeof(float),1,center_mass);
fwrite(rcmx,sizeof(float),nfib,center_mass);
fwrite(rcmy,sizeof(float),nfib,center_mass);
fwrite(rcmz,sizeof(float),nfib,center_mass);
fwrite(&tprint,sizeof(float),1,uxfile);
fwrite(&tprint,sizeof(float),1,uyfile);
fwrite(&tprint,sizeof(float),1,uzfile);
fwrite(&tprint,sizeof(float),1,wxfile);
fwrite(&tprint,sizeof(float),1,wyfile);
fwrite(&tprint,sizeof(float),1,wzfile);
fwrite(&tprint,sizeof(float),1,pxfile);
fwrite(&tprint,sizeof(float),1,pyfile);
fwrite(&tprint,sizeof(float),1,pzfile);
fwrite(&tprint,sizeof(float),1,rxfile);
fwrite(&tprint,sizeof(float),1,ryfile);
fwrite(&tprint,sizeof(float),1,rzfile);
fwrite(&tprint,sizeof(float),1,q0file);
fwrite(&tprint,sizeof(float),1,q1file);
fwrite(&tprint,sizeof(float),1,q2file);
fwrite(&tprint,sizeof(float),1,q3file);
fwrite(px,sizeof(float),nfib*nseg,pxfile);
fwrite(py,sizeof(float),nfib*nseg,pyfile);
fwrite(pz,sizeof(float),nfib*nseg,pzfile);
fwrite(rx,sizeof(float),nfib*nseg,rxfile);
fwrite(ry,sizeof(float),nfib*nseg,ryfile);
fwrite(rz,sizeof(float),nfib*nseg,rzfile);
fwrite(q0,sizeof(float),nfib*nseg,q0file);
fwrite(q1,sizeof(float),nfib*nseg,q1file);
fwrite(q2,sizeof(float),nfib*nseg,q2file);
fwrite(q3,sizeof(float),nfib*nseg,q3file);
fwrite(ux,sizeof(float),nfib*nseg,uxfile);
fwrite(uy,sizeof(float),nfib*nseg,uyfile);
fwrite(uz,sizeof(float),nfib*nseg,uzfile);
fwrite(wx,sizeof(float),nfib*nseg,wxfile);
fwrite(wy,sizeof(float),nfib*nseg,wyfile);
fwrite(wz,sizeof(float),nfib*nseg,wzfile);
// set list of neighbor for each bin
bnei_set << < nzbin*nybin,nxbin >> > (var,intVar);
// 1. zero sorting,friction,X forces variables
delta_twist_zero << <numBlocks,numThreads >> >(var,intVar);
// 2. set up constant groups
delta_twist_torque2 << <numBlocks,numThreads >> >(var,intVar);
// 3. solve for restoring torque
delta_twist_torque << <nfib/fiberPerBlock,(nseg-1)*fiberPerBlock >>
    >(var,intVar);
// 5. Linked cell sort
// a. put fibers into bins
cell << < numBlocks,numThreads >> > (var,intVar);
// b. link with fibers in neighbor bins
link << < nfib*nseg,bdimx*bdimy*bdimz >> > (var,intVar);
// c. get separation between fiber pairs
contact << <numBlocks,numThreads >> >(var,intVar);
// d. generate contact list for fibers
lead << <numBlocks,numThreads >> >(var,intVar);
// 4. find X forces
// a. update u and w
x_forces_presolve << <numBlocks,numThreads >> >(var,intVar);
// b. prepare matrix and RHS
x_forces_para << < numBlocks,numThreads >> >(var,intVar);
// c. solver
status_solver = cusolverSpScsrqrsvBatched(handle_solver,ma,ma,nnz,
  descrA,csrValTot,csrRow,csrCol,VTot,Vsol,batchSize,info,pBuffer);
if (status_solver != CUSOLVER_STATUS_SUCCESS)
  printf("time = %f\n [cusolverSpDcsrqrsvBatched %d]\n",dt*(step+1),
       status_solver);
//// d. update X forces
x_forces_update << <nfib/fiberPerBlock,fiberPerBlock*(nseg-1) >> >(
    var,intVar);
//// e. update u and w
x_forces_postsolve << <numBlocks,numThreads >> >(var,intVar);
// generates lists of fiber a contact group from direct contacts
group << <numBlocks,numThreads,0,stream3 >> >(var,intVar);
// calculate common constants to solve for friction forces
rhs << <numBlocks,numThreads,0,stream4 >> >(var,intVar);
step = 0;
cudaMemcpy(d_step,&step,sizeof(int),cudaMemcpyHostToDevice);
// main loop
for (step = 0; step < time_steps +1; step++){
  cudaMemcpy(d_step,&step,sizeof(int),cudaMemcpyHostToDevice);
  friction_3x3 << <numBlocks,numThreads>> >(var,intVar);
  lead_list << <blasGrid,blasBlock >> >(var,intVar,d_AA,d_PP,d_Pivot,
       d_info,handle_cublas,status_cublas);
  x_forces_presolve << <numBlocks,numThreads >> >(var,intVar);
  x_forces_para << < numBlocks,numThreads,0,stream2 >> >(var,intVar);
  cudaMemcpyAsync(total_contacts,d_total_contacts,sizeof(int),
       cudaMemcpyDeviceToHost,stream3);
  gpuErrchk(cudaDeviceSynchronize());
  status_solver = cusolverSpScsrqrsvBatched(handle_solver,ma,ma,nnz,
       descrA,csrValTot,csrRow,csrCol,VTot,Vsol,batchSize,info,
       pBuffer);
  gpuErrchk(cudaDeviceSynchronize());
  if (status_solver != CUSOLVER_STATUS_SUCCESS){
    tprint = (step+1)*dt;
    printf("time = %f\n [cusolverSpScsrqrsvBatched %d]\n",tprint,
         status_solver);
    gpuErrchk(cudaDeviceSynchronize());
  }
```

```
x_forces_update << <nfib/fiberPerBlock,fiberPerBlock*(nseg-1) >> >(
    var,intVar);
x_forces_postsolve << <numBlocks,numThreads >> >(var,intVar);
updateOri << <numBlocks,numThreads,0,stream2 >> >(var,intVar);
updateCen << <nfibGrid,nfibBlock,0,stream3 >> >(var,intVar);
delta_twist_zero << <numBlocks,numThreads,0,stream4 >> >(var,intVar
    );
if (driedRehyd && (step+1) % box_write == 0){
  boxSize << <nfibGrid,nfibBlock >> >(var,intVar,
    d_Lx,d_Ly,d_Lz,d_dLx,d_dLy,d_dLz,d_box_write,
    d_dx_fixed,d_dy_fixed,d_dz_fixed);
  cudaMemcpy(&nxbin,d_nxbin,sizeof(int),cudaMemcpyDeviceToHost);
  cudaMemcpy(&nybin,d_nybin,sizeof(int),cudaMemcpyDeviceToHost);
  cudaMemcpy(&nzbin,d_nzbin,sizeof(int),cudaMemcpyDeviceToHost);
  bnei_set << < nzbin*nybin,nxbin >> > (var,intVar);
}
regrowSeg << <nfibGrid,nfibBlock >> >(var,intVar);
regrowFib << <nfib/fiberPerBlock,(nseg-1)*fiberPerBlock,0,stream2
    >> >(var,intVar);
updateBod << < numBlocks,numThreads,0,stream3 >> >(var,intVar);
lees_edwards << <1,1,0,stream4 >> >(var,intVar);
delta_twist_torque << <nfib/fiberPerBlock,(nseg-1)*fiberPerBlock,0,
    stream5 >> >(var,intVar);
delta_twist_torque2 << <numBlocks,numThreads,0,stream6 >> >(var,
    intVar);
cell << < numBlocks,numThreads >> > (var,intVar);
link << < nfib*nseg,bdimx*bdimy*bdimz,0,stream2 >> > (var,intVar);
contact << <numBlocks,numThreads,0,stream2 >> >(var,intVar);
if ((step+1) % stress_write == 0){
  stress << <numBlocks,numThreads,0,stream4 >> >(var,intVar);
  cudaMemcpy(Stress,d_Stress,6*sizeof(float),cudaMemcpyDeviceToHost
      );
  fprintf(Stress_tensor,"%7.5f %15.6f %15.6f %15.6f %15.6f %15.6f
      %15.6f\n",
    (step+1)*dt,Stress[0],Stress[1],Stress[2],Stress[3],Stress[4],
        Stress[5]);
}
if ((step+1) % config_write == 0){
  cudaMemcpyAsync(ux,d_ux,nfib*nseg*sizeof(float),
      cudaMemcpyDeviceToHost,stream3);
  cudaMemcpyAsync(uy,d_uy,nfib*nseg*sizeof(float),
      cudaMemcpyDeviceToHost,stream3);
  cudaMemcpyAsync(uz,d_uz,nfib*nseg*sizeof(float),
      cudaMemcpyDeviceToHost,stream3);
  cudaMemcpyAsync(wx,d_wx,nfib*nseg*sizeof(float),
      cudaMemcpyDeviceToHost,stream3);
  cudaMemcpyAsync(wy,d_wy,nfib*nseg*sizeof(float),
      cudaMemcpyDeviceToHost,stream3);
  cudaMemcpyAsync(wz,d_wz,nfib*nseg*sizeof(float),
      cudaMemcpyDeviceToHost,stream3);
  cudaMemcpyAsync(rx,d_rx,nfib*nseg*sizeof(float),
      cudaMemcpyDeviceToHost,stream3);
  cudaMemcpyAsync(ry,d_ry,nfib*nseg*sizeof(float),
      cudaMemcpyDeviceToHost,stream3);
  cudaMemcpyAsync(rz,d_rz,nfib*nseg*sizeof(float),
      cudaMemcpyDeviceToHost,stream3);
  cudaMemcpyAsync(px,d_px,nfib*nseg*sizeof(float),
      cudaMemcpyDeviceToHost,stream3);
  cudaMemcpyAsync(py,d_py,nfib*nseg*sizeof(float),
      cudaMemcpyDeviceToHost,stream3);
  cudaMemcpyAsync(pz,d_pz,nfib*nseg*sizeof(float),
      cudaMemcpyDeviceToHost,stream3);
  cudaMemcpyAsync(rcmx,d_rcmx,nfib*sizeof(float),
      cudaMemcpyDeviceToHost,stream3);
  cudaMemcpyAsync(rcmy,d_rcmy,nfib*sizeof(float),
      cudaMemcpyDeviceToHost,stream3);
  cudaMemcpyAsync(rcmz,d_rcmz,nfib*sizeof(float),
      cudaMemcpyDeviceToHost,stream3);
  cudaMemcpyAsync(q0,d_q0,nfib*nseg*sizeof(float),
      cudaMemcpyDeviceToHost,stream3);
  cudaMemcpyAsync(q1,d_q1,nfib*nseg*sizeof(float),
      cudaMemcpyDeviceToHost,stream3);
  cudaMemcpyAsync(q2,d_q2,nfib*nseg*sizeof(float),
      cudaMemcpyDeviceToHost,stream3);
  cudaMemcpyAsync(q3,d_q3,nfib*nseg*sizeof(float),
      cudaMemcpyDeviceToHost,stream3);
}
lead << <numBlocks,numThreads >> >(var,intVar);
group << <numBlocks,numThreads,0,stream2 >> >(var,intVar);
rhs << <numBlocks,numThreads,0,stream4 >> >(var,intVar);
cudaStreamSynchronize(stream3);
// Rewrite configurations every config_write steps
if ((step+1) % config_write == 0){
  tprint = (step+1)*dt;
  fwrite(&tprint,sizeof(float),1,center_mass);
  fwrite(rcmx,sizeof(float),nfib,center_mass);
  fwrite(rcmy,sizeof(float),nfib,center_mass);
  fwrite(rcmz,sizeof(float),nfib,center_mass);
  fwrite(&tprint,sizeof(float),1,pxfile);
  fwrite(&tprint,sizeof(float),1,pyfile);
  fwrite(&tprint,sizeof(float),1,pzfile);
  fwrite(&tprint,sizeof(float),1,rxfile);
  fwrite(&tprint,sizeof(float),1,ryfile);
  fwrite(&tprint,sizeof(float),1,rzfile);
  fwrite(&tprint,sizeof(float),1,uxfile);
  fwrite(&tprint,sizeof(float),1,uyfile);
  fwrite(&tprint,sizeof(float),1,uzfile);
  fwrite(&tprint,sizeof(float),1,wxfile);
  fwrite(&tprint,sizeof(float),1,wyfile);
  fwrite(&tprint,sizeof(float),1,wzfile);
  fwrite(&tprint,sizeof(float),1,q0file);
  fwrite(&tprint,sizeof(float),1,q1file);
  fwrite(&tprint,sizeof(float),1,q2file);
  fwrite(&tprint,sizeof(float),1,q3file);
  fwrite(px,sizeof(float),nfib*nseg,pxfile);
  fwrite(py,sizeof(float),nfib*nseg,pyfile);
```

```
    fwrite(pz,sizeof(float),nfib*nseg,pzfile);
    fwrite(rx,sizeof(float),nfib*nseg,rxfile);
    fwrite(ry,sizeof(float),nfib*nseg,ryfile);
    fwrite(rz,sizeof(float),nfib*nseg,rzfile);
    fwrite(q0,sizeof(float),nfib*nseg,q0file);
    fwrite(q1,sizeof(float),nfib*nseg,q1file);
    fwrite(q2,sizeof(float),nfib*nseg,q2file);
    fwrite(q3,sizeof(float),nfib*nseg,q3file);
    fwrite(ux,sizeof(float),nfib*nseg,uxfile);
    fwrite(uy,sizeof(float),nfib*nseg,uyfile);
    fwrite(uz,sizeof(float),nfib*nseg,uzfile);
    fwrite(wx,sizeof(float),nfib*nseg,wxfile);
    fwrite(wy,sizeof(float),nfib*nseg,wyfile);
    fwrite(wz,sizeof(float),nfib*nseg,wzfile);
  }
  gpuErrchk(cudaDeviceSynchronize());
  if ((step+1) % contact_write == 0){
    cudaMemcpyAsync(num_groups,d_num_groups,sizeof(int),
        cudaMemcpyDefault,stream2);
    cudaMemcpy(&overs,d_overs,sizeof(int),cudaMemcpyDeviceToHost);
    fprintf(Number_of_Contacts,"%8.5f %6d %6d %6.3f %5d\n",
      (step+1)*dt,*num_groups,*total_contacts,float(*total_contacts)/
          float(nfib),overs);
  }
  gpuErrchk(cudaDeviceSynchronize());
  if ((step+1) % 10 == 0){
// if ((step+1) % 10000 == 0{)
    t = dt*float(step+1);
    printf("%8.5f %6d %6d %6.3f %5d\n",
      t,*num_groups,*total_contacts,float(*total_contacts)/float(nfib
          ),overs);
  }
  printf("%f\n",float(step)*dt);
  if ((step+1) % 50000 == 0){
    fflush(stdout); fflush(Number_of_Contacts);
    fflush(Stress_tensor); fflush(rxfile);
    fflush(ryfile); fflush(rzfile);
    fflush(pxfile); fflush(pyfile);
    fflush(pzfile); fflush(wxfile);
    fflush(wyfile); fflush(wzfile);
    fflush(uxfile); fflush(uyfile);
    fflush(uzfile); fflush(q0file);
    fflush(q1file); fflush(q2file);
    fflush(q3file); fflush(center_mass);
  }
}
// close files
cudaDeviceSynchronize();
fclose(center_mass); fclose(Number_of_Contacts);
fclose(pxfile); fclose(pyfile); fclose(pzfile);
fclose(rxfile); fclose(ryfile); fclose(rzfile);
fclose(q0file); fclose(q1file);
fclose(q2file); fclose(q3file);
  fclose(uxfile); fclose(uyfile); fclose(uzfile);
  fclose(wxfile); fclose(wyfile); fclose(wzfile);
  fclose(Stress_tensor);
  // release resources used by cusolver
  cusolverSpDestroyCsrqrInfo(info);
  status_solver = cusolverSpDestroy(handle_solver);
  if (status_solver == CUSOLVER_STATUS_NOT_INITIALIZED) printf("
      CUSOLVER_STATUS_NOT_INITIALIZED\n");
  status_sparse = cusparseDestroyMatDescr(descrA);
  if (status_sparse != CUSPARSE_STATUS_SUCCESS) printf("[
      cusparseDestroyMatDescr status %d]\n",status_sparse);
  status_sparse = cusparseDestroy(handle_sparse);
  if (status_sparse == CUSPARSE_STATUS_NOT_INITIALIZED) printf("
      CUSPARSE_STATUS_NOT_INITIALIZED\n");
  cudaDeviceSynchronize();
  cublas_free << <blasGrid,blasBlock >> >(handle_cublas);
  gpuErrchk(cudaDeviceSynchronize());
  return 0;
}
```

bnei_set.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <math.h>
#include "bnei_set.h"
using namespace std;
__global__ void bnei_set(float **var, int **intVar){
  int *bnei=intVar[14]; int nxbin=*intVar[15];
  int nybin=*intVar[16]; int nzbin=*intVar[17];
  int bdimx=*intVar[22]; int bdimy=*intVar[23];
  int bdimz=*intVar[24];
  int tid=threadIdx.x +blockIdx.x*blockDim.x;
  int xbin, ybin, zbin, xcen, ycen, zcen;
  int xpos, ypos, zpos, xind, yind, zind;
  int xdiff, ydiff, zdiff, ind, tid2;
  // id of the central bin
  xcen=(bdimx-1)/2;
  ycen=(bdimy-1)/2;
  zcen=(bdimz-1)/2;
  zbin=tid/(nxbin*nybin);
  ybin=(tid-zbin*nxbin*nybin)/nxbin;
  xbin=tid-ybin*nxbin-zbin*nxbin*nybin;
  // find id of neighboring bins
  for (tid2=0; tid2 < bdimx*bdimy*bdimz; tid2++){
    zpos=tid2/(bdimx*bdimy);
    ypos=(tid2-zpos*bdimx*bdimy)/bdimx;
    xpos=tid2-ypos*bdimx-zpos*bdimx*bdimy;
    xdiff=xpos-xcen;
    xind=xbin +xdiff;
    if (xind < 0){
```

```
      xind += nxbin;
    }
    if (xind >= nxbin){
      xind -= nxbin;
    }
    ydiff=ypos-ycen;
    yind=ybin +ydiff;
    if (yind < 0){
      yind += nybin;
    }
    if (yind >= nybin){
      yind -= nybin;
    }
    zdiff=zpos-zcen;
    zind=zbin +zdiff;
    if (zind < 0){
      zind += nzbin;
    }
    if (zind >= nzbin){
      zind -= nzbin;
    }
    ind=xind +yind*nxbin +zind*nxbin*nybin;
    if ((xbin +ybin*nxbin +zbin*nxbin*nybin +tid2*nxbin*nybin*nzbin) >=
         0 && (xbin +ybin*nxbin +zbin*nxbin*nybin +tid2*nxbin*nybin*
        nzbin) < nxbin*nybin*nzbin*bdimx*bdimy*bdimz){
      bnei[xbin +ybin*nxbin +zbin*nxbin*nybin +tid2*nxbin*nybin*nzbin]=
           ind;
    }
    else{
      printf("error: tid2 %4d\n", tid2);
    }
  }
}
```

boxSize.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <math.h>
#include "boxSize.h"
using namespace std;
__global__ void boxSize(float **var, int **intVar,
  float *Lx, float *Ly, float *Lz,
  float *dLx, float *dLy, float *dLz, int *box_write,
  float *dx_fixed, float *dy_fixed, float *dz_fixed){
  int step=*intVar[9];
  int *nxbin=intVar[15];
  int *nybin=intVar[16];
  int *nzbin=intVar[17];
  float *rcmx=var[0];
  float *rcmy=var[1];
  float *rcmz=var[2];
  float *dx=var[108];
  float *dy=var[109];
  float *dz=var[110];
  float *sidex=var[119];
  float *sidey=var[120];
  float *sidez=var[121];
  float *delta_rx=var[138];
  int m=threadIdx.x + blockIdx.x*blockDim.x;
  int boxInd;
  boxInd=int(float(step + 1)/float(*box_write));
  // have one thread in charge of updating box size info
  if (m == 0){
    *delta_rx=*delta_rx*dLx[boxInd];
    *sidex=Lx[boxInd];
    *sidey=Ly[boxInd];
    *sidez=Lz[boxInd];
    *nxbin=int(floorf(*sidex / *dx_fixed));
    *nybin=int(floorf(*sidey / *dy_fixed));
    *nzbin=int(floorf(*sidez / *dz_fixed));
    if (*nxbin % 2 != 0) (*nxbin)--;
    if (*nybin % 2 != 0) (*nybin)--;
    if (*nzbin % 2 != 0) (*nzbin)--;
    *dx=Lx[boxInd]/float(*nxbin);
    *dy=Ly[boxInd]/float(*nybin);
    *dz=Lz[boxInd]/float(*nzbin);
  }
  rcmx[m] *= dLx[boxInd];
  rcmy[m] *= dLy[boxInd];
  rcmz[m] *= dLz[boxInd];
}
```

cell.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <math.h>
#include "cell.h"
using namespace std;
__global__ void cell(float **var,int **intVar){
  int *bin=intVar[12]; int *list=intVar[13];
  int nxbin=*intVar[15]; int nybin=*intVar[16];
  int nzbin=*intVar[17]; int maxBin=*intVar[28];
  int *bnum=intVar[31]; float *rx=var[3];
  float *ry=var[4]; float *rz=var[5];
  float sidex=*var[119]; float sidey=*var[120];
  float sidez=*var[121]; float delta_rx=*var[138];
  float dx=*var[108]; float dy=*var[109];
  float dz=*var[110];
  int xbin,ybin,zbin,old,corx,cory,corz;
  float rxmi,rymi,rzmi;
  int mi=threadIdx.x + blockDim.x*blockIdx.x;
  cory=roundf(ry[mi]/sidey);
```

```
  corz=roundf(rz[mi]/sidez);
  rxmi=rx[mi]-corz*delta_rx;
  corx=roundf(rxmi/sidex);
  rymi=ry[mi]-cory*sidey;
  rzmi=rz[mi]-corz*sidez;
  rxmi=rxmi-corx*sidex;
  xbin=int(floorf(rxmi/dx)) + nxbin/2;
  ybin=int(floorf(rymi/dy)) + nybin/2;
  zbin=int(floorf(rzmi/dz)) + nzbin/2;
  if (xbin == nxbin){
    xbin--;
  }
  if (ybin == nybin){
    ybin--;
  }
  if (zbin == nzbin){
    zbin--;
  }
  if (xbin == -1){
    xbin++;
  }
  if (ybin == -1){
    ybin++;
  }
  if (zbin == -1){
    zbin++;
  }
  if (xbin < 0 || ybin < 0 || zbin < 0 || xbin >= nxbin || ybin >=
       nybin || zbin >= nzbin){
    printf("error in cell: bin index out of bound\n");
    printf("mi=%4d (%3d %3d %3d) rx ry rz %10f %10f %10f \n",mi,xbin,
         ybin,zbin,rx[mi],ry[mi],rz[mi]);
  }
  old=atomicAdd(bin + xbin + ybin*nxbin + zbin*nxbin*nybin,1);
  list[xbin + ybin*nxbin + zbin*nxbin*nybin + old*nxbin*nybin*nzbin]=mi
       ;
  bnum[mi]=xbin + ybin*nxbin + zbin*nxbin*nybin;
  if (old > maxBin){
    printf("error in cell: number of segments per bin %4d exceeds
         maxBin\n",old);
  }
}
```

contact.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <math.h>
#include "contact.h"
using namespace std;
__device__ void parallel_sort_para(int mi,int nj,float sx,
  float sy,float sz,float pxmi,float pymi,float pzmi,
  float pxnj,float pynj,float pznj,float pdotp,float rp,
  float *xmin,float *ymin);
__global__ void contact(float **var,int **intVar){
  const int npcn=2500;
  int mi=threadIdx.x+blockIdx.x*blockDim.x;
  int *potConSize=intVar[33];
  int nPair=potConSize[mi];
  if (nPair == 0) return;
  int *ncpf=intVar[2]; int *clist=intVar[18];
  int maxCon=*intVar[25]; int *potCon=intVar[32];
  float *rx=var[3]; float *ry=var[4];
  float *rz=var[5]; float *px=var[18];
  float *py=var[19]; float *pz=var[20];
  float *fcx=var[66]; float *fcy=var[67];
  float *fcz=var[68]; float *tcx=var[69];
  float *tcy=var[70]; float *tcz=var[71];
  float rp=*var[114]; float over_cut=*var[118];
  float sidex=*var[119]; float sidey=*var[120];
  float sidez=*var[121]; float contact_cutoff=*var[128];
  float rep_cutoff=*var[129]; float delta_rx=*var[138];
  float fstar=*var[144]; float fact=*var[145];
  float Astar=*var[146]; float decatt=*var[147];
  float *GijxV=var[149]; float *GijyV=var[150];
  float *GijzV=var[151]; float *GjixV=var[152];
  float *GjiyV=var[153]; float *GjizV=var[154];
  float *nxV=var[155]; float *nyV=var[156];
  float *nzV=var[157]; float *gV=var[158];
  // local variables
  float rxmi,rymi,rzmi,rxnj,rynj,rznj;
  float pxmi,pymi,pzmi,pxnj,pynj,pznj;
  float sxx,syy,szz,corx,cory,corz;
  float rxmi_shift,rymi_shift,rzmi_shift;
  float pdotp,xmin,ymin,dx,dy,dz,sep;
  float xi[9],yj[9],gij,nijx,nijy,nijz,forc;
  float Gijx,Gijy,Gijz,Gjix,Gjiy,Gjiz,sep_tmp;
  int nP,nj,ipos,ith,oldmi,oldnj;
  rxmi=rx[mi]; rymi=ry[mi]; rzmi=rz[mi];
  pxmi=px[mi]; pymi=py[mi]; pzmi=pz[mi];
  // each fiber goes through fibers in its cell list
  for (nP=0; nP < nPair; nP++){
    // index of the neighbor being considered
    nj=potCon[mi+npcn+nP];
    rxnj=rx[nj]; rynj=ry[nj]; rznj=rz[nj];
    pxnj=px[nj]; pynj=py[nj]; pznj=pz[nj];
    // find minimum image (for shear flow system)
    sxx=rxnj-rxmi;
    syy=rynj-rymi;
    szz=rznj-rzmi;
    cory=roundf(syy/sidey);
    corz=roundf(szz/sidez);
    sxx=sxx-corz*delta_rx;
    corx=roundf(sxx/sidex);
    sxx=sxx-corx*sidex;
```

```
syy=syy-cory*sidey;
szz=szz-corz*sidez;
rxmi_shift=rxnj-sxx;
rymi_shift=rynj-syy;
rzmi_shift=rznj-szz;
pdotp=pxmi*pxnj+pymi*pynj+pzmi*pznj;
xmin=(-(pxnj+sxx+pynj+syy+pznj+szz)* pdotp
  + (pxmi+sxx+pymi+syy+pzmi+szz))
  / (1.0-pdotp*pdotp);
ymin=((pxmi+sxx+pymi+syy+pzmi+szz)* pdotp
  - (pxnj+sxx+pynj+syy+pznj+szz))
  / (1.0-pdotp*pdotp);
dx=rxnj+ymin*pxnj-rxmi_shift-xmin*pxmi;
dy=rynj+ymin*pynj-rymi_shift-xmin*pymi;
dz=rznj+ymin*pznj-rzmi_shift-xmin*pzmi;
sep=dx*dx+dy*dy+dz*dz;
// find the type of contact
ipos=8;
yj[0]=rp;
xi[0]=pxmi*sxx+pymi*syy+pzmi*szz+yj[0]+pdotp;
yj[1]=-rp;
xi[1]=pxmi*sxx+pymi*syy+pzmi*szz+yj[1]+pdotp;
xi[2]=rp;
yj[2]=-(pxnj*sxx+pynj*syy+pznj*szz)+xi[2]+pdotp;
xi[3]=-rp;
yj[3]=-(pxnj*sxx+pynj*syy+pznj*szz)+xi[3]+pdotp;
xi[4]=rp; yj[4]=rp;
xi[5]=rp; yj[5]=-rp;
xi[6]=-rp; yj[6]=rp;
xi[7]=-rp; yj[7]=-rp;
xi[8]=xmin; yj[8]=ymin;
// Check if segments are parallel
if (fabsf(pdotp*pdotp-1.0) <= 1.0e-6) {
  parallel_sort_para(mi,nj,sxx,syy,szz,pxmi,pymi,pzmi,
    pxnj,pynj,pznj,pdotp,rp,&xmin,&ymin);
  sep=(sxx+ymin*pxnj-xmin*pxmi)*(sxx+ymin*pxnj-xmin*pxmi) +
    (syy+ymin*pynj-xmin*pymi)*(syy+ymin*pynj-xmin*pymi) +
    (szz+ymin*pznj-xmin*pzmi)*(szz+ymin*pznj-xmin*pzmi);
}
else if (sep < rep_cutoff && (fabsf(xmin) >= rp || fabsf(ymin) >=
     rp)){
  sep=1000.0;
  // check which end-side or end-end separation
  // is the smallest
  for (ith=0; ith < 8; ith++){
    sep_tmp=(sxx+yj[ith]+pxnj-xi[ith]+pxmi)*(sxx+yj[ith]+pxnj-xi[
        ith]+pxmi) +
      (syy+yj[ith]+pynj-xi[ith]+pymi)*(syy+yj[ith]+pynj-xi[ith]+
          pymi) +
      (szz+yj[ith]+pznj-xi[ith]+pzmi)*(szz+yj[ith]+pznj-xi[ith]+
          pzmi);
    if (sep_tmp < sep && fabsf(xi[ith]) <= rp && fabsf(yj[ith]) <=
        rp){
      sep=sep_tmp;
      ipos=ith;
    }
  }
  xmin=xi[ipos];
  ymin=yj[ipos];
}
gij=sqrtf(sep);
nijx=(sxx+ymin*pxnj-xmin*pxmi)/gij;
nijy=(syy+ymin*pynj-xmin*pymi)/gij;
nijz=(szz+ymin*pznj-xmin*pzmi)/gij;
Gijx=xmin*pxmi+gij*nijx/2.0;
Gijy=xmin*pymi+gij*nijy/2.0;
Gijz=xmin*pzmi+gij*nijz/2.0;
Gjix=ymin*pxnj-gij*nijx/2.0;
Gjiy=ymin*pynj-gij*nijy/2.0;
Gjiz=ymin*pznj-gij*nijz/2.0;
forc=fstar*expf(-fact*(gij-2.0))-Astar*expf(-decatt*(gij-2.0)*(gij
    -2.0));
if (gij < 2.0){
  atomicAdd(intVar[4],1); // overs
  forc=fstar*expf(-fact*(gij-2.0))-Astar;
}
if (gij < over_cut){
  gij=over_cut;
  forc=fstar*expf(-fact*(gij-2.0))-Astar;
}
if (sep < rep_cutoff){
  atomicAdd(fcx+mi,-nijx*forc);
  atomicAdd(fcy+mi,-nijy*forc);
  atomicAdd(fcz+mi,-nijz*forc);
  atomicAdd(tcx+mi,-forc*xmin*(pymi*nijz-pzmi*nijy));
  atomicAdd(tcy+mi,-forc*xmin*(pzmi*nijx-pxmi*nijz));
  atomicAdd(tcz+mi,-forc*xmin*(pxmi*nijy-pymi*nijx));
  atomicAdd(fcx+nj,nijx*forc);
  atomicAdd(fcy+nj,nijy*forc);
  atomicAdd(fcz+nj,nijz*forc);
  atomicAdd(tcx+nj,forc*ymin*(pynj*nijz-pznj*nijy));
  atomicAdd(tcy+nj,forc*ymin*(pznj*nijx-pxnj*nijz));
  atomicAdd(tcz+nj,forc*ymin*(pxnj*nijy-pynj*nijx));
}
if (sep < contact_cutoff){
  oldmi=atomicAdd(ncpf+mi,1);
  oldnj=atomicAdd(ncpf+nj,1);
  clist[mi*maxCon+oldmi]=nj;
  clist[nj*maxCon+oldnj]=mi;
  nxV[mi*maxCon+oldmi]=nijx;
  nyV[mi*maxCon+oldmi]=nijy;
  nzV[mi*maxCon+oldmi]=nijz;
  gV[mi*maxCon+oldmi]=gij;
  GijxV[mi*maxCon+oldmi]=Gijx;
  GijyV[mi*maxCon+oldmi]=Gijy;
  GijzV[mi*maxCon+oldmi]=Gijz;
```

```
      GjixV[mi*maxCon+oldmi]=Gjix;
      GjiyV[mi*maxCon+oldmi]=Gjiy;
      GjizV[mi*maxCon+oldmi]=Gjiz;
      if (oldmi >= maxCon || oldnj >= maxCon){
        printf("link: mi nj %7d %7d exceeding maxCon,allocate more
             space\n",mi,nj);
      }
    }
  }
}
__device__ void parallel_sort_para(int mi,int nj,float sx,float sy,
  float sz,float pxmi,float pymi,float pzmi,float pxnj,float pynj,
  float pznj,float pdotp,float rp,float *xmin,float *ymin){
  float posneg,pn2,dist,sijp,sijm,sjip,sjim;
  // The different end point to fiber contact points
  sijp=pxmi*sx+pymi*sy+pzmi*sz+rp*pdotp;
  sijm=pxmi*sx+pymi*sy+pzmi*sz-rp*pdotp;
  sjip=-(pxnj*sx+pynj*sy+pznj*sz)+rp*pdotp;
  sjim=-(pxnj*sx+pynj*sy+pznj*sz)-rp*pdotp;
  // for fiber i
  if (fabsf(sijp) < fabsf(sijm)){
    *xmin=sijp;
    posneg=1.0;
  }
  else if (fabsf(sijp) > fabsf(sijm)){
    *xmin=sijm;
    posneg=-1.0;
  }
  else{
    *xmin=0.0;
    posneg=0.0;
  }
  if (*xmin >= rp){
    *xmin=rp;
  }
  if (*xmin <= -rp){
    *xmin=-rp;
  }
  // for fiber j
  if (fabsf(sjip) < fabsf(sjim)){
    *ymin=sjip;
  }
  else if (fabsf(sjip) > fabsf(sjim)){
    *ymin=sjim;
  }
  else{
    *ymin=0.0;
    posneg=0.0;
  }
  if (*ymin >= rp){
    *ymin=rp;
  }
  if (*ymin <= -rp){
    *ymin=-rp;
  }
  if (fabsf(*xmin) < rp && fabsf(*ymin) < rp){
    if (pdotp > 0.0){
      pn2=1.0;
    }
    else{
      pn2=-1.0;
    }
    dist=(rp+posneg**xmin)/2.0;
    *xmin=*xmin-posneg*dist;
    *ymin=*ymin+posneg*pn2*dist;
  }
}
```

csr_set.cu

```
#include <stdio.h>
#include <cuda.h>
#include "csr_set.h"
using namespace std;
__global__ void csr_set(int **intVar){
  int nseg = *intVar[7]; int *csrRow = intVar[10];
  int *csrCol = intVar[11];
  int blockDimA = 3;
  int ma = (nseg-1)*blockDimA;
  int nnz = (4+3*(nseg-3))*blockDimA*blockDimA;
  int j, k;
  if (nseg == 2){
    for (j = 0; j < 3; j++){
      csrCol[3*j+0] = 0;
      csrCol[3*j+1] = 1;
      csrCol[3*j+2] = 2;
      csrRow[j] = 3*j;
    }
    csrRow[ma] = nnz;
    return;
  }
  csrRow[0] = 0;
  csrRow[1] = 6;
  csrRow[2] = 12;
  csrRow[3] = 18;
  for (k = 4; k < ma-2; k++){
    csrRow[k] = csrRow[k-1]+9;
  }
  for (k = ma-2; k < ma; k++){
    csrRow[k] = csrRow[k-1]+6;
  }
  csrRow[ma] = nnz;

  for (j = 0; j < 3;j++){
    for (k = 0; k < 6; k++){
      csrCol[j*6+k] = k;
```

```
    }
  }
  for (j = 0; j < (nseg-3); j++){
    for (k = 0; k < 9; k++){
      csrCol[18+j*27+k] = k+3*j;
      csrCol[27+j*27+k] = k+3*j;
      csrCol[36+j*27+k] = k+3*j;
    }
  }
  for (j = 0; j < 3; j++){
    for (k = 0; k < 6; k++){
      csrCol[(nnz-18)+6*j+k] = k+3*(nseg-3);
    }
  }
}
```

cublas_initialize.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cublas_v2.h>
#include "cublas_initialize.h"
__device__ const char* cublasGetErrorString(cublasStatus_t status);
__global__ void cublas_initialize(cublasHandle_t *handle_cublas){
  cublasStatus_t status = CUBLAS_STATUS_SUCCESS;
  int mi = threadIdx.x + blockIdx.x*blockDim.x;
  status = cublasCreate_v2(handle_cublas + mi);
  if (status != CUBLAS_STATUS_SUCCESS)
    printf("failure intialize cublas handle mi status %6d %s\n", mi,
        cublasGetErrorString(status));
}
__device__ const char* cublasGetErrorString(cublasStatus_t status){
  switch(status){
    case CUBLAS_STATUS_SUCCESS: return "CUBLAS_STATUS_SUCCESS";
    case CUBLAS_STATUS_NOT_INITIALIZED: return "
        CUBLAS_STATUS_NOT_INITIALIZED";
    case CUBLAS_STATUS_ALLOC_FAILED: return "CUBLAS_STATUS_ALLOC_FAILED
        ";
    case CUBLAS_STATUS_INVALID_VALUE: return "
        CUBLAS_STATUS_INVALID_VALUE";
    case CUBLAS_STATUS_ARCH_MISMATCH: return "
        CUBLAS_STATUS_ARCH_MISMATCH";
    case CUBLAS_STATUS_MAPPING_ERROR: return "
        CUBLAS_STATUS_MAPPING_ERROR";
    case CUBLAS_STATUS_EXECUTION_FAILED: return "
        CUBLAS_STATUS_EXECUTION_FAILED";
    case CUBLAS_STATUS_INTERNAL_ERROR: return "
        CUBLAS_STATUS_INTERNAL_ERROR";
  }
  return "unknown error";
}
```

cublas_free.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cublas_v2.h>
#include "cublas_free.h"
__global__ void cublas_free(cublasHandle_t *handle_cublas){
       int mi = threadIdx.x + blockIdx.x*blockDim.x;
       cublasDestroy_v2(handle_cublas[mi]);
}
```

delta_twist_torque.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <math.h>
#include "delta_twist_torque.h"
using namespace std;
__global__ void delta_twist_torque(float **var,int **intVar){
  int nseg=*intVar[7]; float *R21eq=var[45];
  float *R22eq=var[46]; float *R23eq=var[47];
  float *R31eq=var[48]; float *R32eq=var[49];
  float *R33eq=var[50]; float kb=*var[115];
  float kt=*var[116]; float *rx=var[3];
  float *ry=var[4]; float *rz=var[5];
  float *px=var[18]; float *py=var[19];
  float *pz=var[20]; float *R11=var[36];
  float *R12=var[37]; float *R13=var[38];
  float *R21=var[39]; float *R22=var[40];
  float *R23=var[41]; float *Yx=var[54];
  float *Yy=var[55]; float *Yz=var[56];
  float cx,cy,cz,ang_theta,ang_phi;
  // cx,cy,cz-unit vector of (sxx,syy,szz)
  // ang_theta,ang_phi-amount the joint is deformed from equil.
  float zeqx,zeqy,zeqz,zdotz,dirx,diry,dirz,dum;
  float yitx,yity,yitz,yieqx,yieqy,yieqz;
  float Ybex,Ybey,Ybez,Ytx,Yty,Ytz;
  float pxmi,pymi,pzmi,pxmi_1,pymi_1,pzmi_1;
  float R21mi,R22mi,R23mi;
  float R11mi_1,R12mi_1,R13mi_1,R21mi_1,R22mi_1,R23mi_1;
  float R21eqmi,R22eqmi,R23eqmi,R31eqmi,R32eqmi,R33eqmi;
  int m,i,mi,mi2;
  int tid=threadIdx.x+blockDim.x*blockIdx.x;
  m=tid/(nseg-1);
  i=tid-m * (nseg-1)+1;
  mi=m*nseg+i;
  mi2=m*(nseg+1)+i;
  // Calculate joint torques
  // Calculate bending and twisting torques,both isotropic
```

```
// Find the bending component first
// peq vecotr
R11mi_1=R11[mi-1]; R12mi_1=R12[mi-1]; R13mi_1=R13[mi-1];
R21mi_1=R21[mi-1]; R22mi_1=R22[mi-1]; R23mi_1=R23[mi-1];
R31eqmi=R31eq[mi]; R32eqmi=R32eq[mi]; R33eqmi=R33eq[mi];
pxmi_1=px[mi-1]; pymi_1=py[mi-1]; pzmi_1=pz[mi-1];
zeqx=R11mi_1 * R31eqmi+R21mi_1 * R32eqmi+pxmi_1*R33eqmi;
zeqy=R12mi_1 * R31eqmi+R22mi_1 * R32eqmi+pymi_1*R33eqmi;
zeqz=R13mi_1 * R31eqmi+R23mi_1 * R32eqmi+pzmi_1*R33eqmi;
pxmi=px[mi]; pymi=py[mi]; pzmi=pz[mi];
zdotz=pxmi*zeqx+pymi*zeqy+pzmi*zeqz;
dirx=pymi*zeqz-pzmi*zeqy;
diry=pzmi*zeqx-pxmi*zeqz;
dirz=pxmi*zeqy-pymi*zeqx;
dum=sqrtf(dirx*dirx+diry*diry+dirz*dirz);
if (zdotz > 1.0-1.0E-6){
  Ybex=0.0; Ybey=0.0; Ybez=0.0;
}
else{
  if (zdotz < -1.0){
    zdotz=-1.0;
  }
  ang_theta=acosf(zdotz);
  Ybex=-kb*ang_theta*(dirx/dum);
  Ybey=-kb*ang_theta*(diry/dum);
  Ybez=-kb*ang_theta*(dirz/dum);
}
// Find the twisting component
// first find the c vector
R21eqmi=R21eq[mi]; R22eqmi=R22eq[mi]; R23eqmi=R23eq[mi];
cx=rx[mi]-rx[mi-1];
cy=ry[mi]-ry[mi-1];
cz=rz[mi]-rz[mi-1];
dum=sqrtf(cx*cx+cy*cy+cz*cz);
cx=cx/dum;
cy=cy/dum;
cz=cz/dum;
zeqx=R11mi_1*R21eqmi+R21mi_1*R22eqmi+pxmi_1*R23eqmi;
zeqy=R12mi_1*R21eqmi+R22mi_1*R22eqmi+pymi_1*R23eqmi;
zeqz=R13mi_1*R21eqmi+R23mi_1*R22eqmi+pzmi_1*R23eqmi;
// identify the perpendicular components of y and yeq
R21mi=R21[mi]; R22mi=R22[mi]; R23mi=R23[mi];
yitx=R21mi-cx*(cx*R21mi+cy*R22mi+cz*R23mi);
yity=R22mi-cy*(cx*R21mi+cy*R22mi+cz*R23mi);
yitz=R23mi-cz*(cx*R21mi+cy*R22mi+cz*R23mi);
dum=sqrtf(yitx*yitx+yity*yity+yitz*yitz);
yitx=yitx/dum;
yity=yity/dum;
yitz=yitz/dum;
yieqx=zeqx-cx*(cx*zeqx+cy*zeqy+cz*zeqz);
yieqy=zeqy-cy*(cx*zeqx+cy*zeqy+cz*zeqz);
yieqz=zeqz-cz*(cx*zeqx+cy*zeqy+cz*zeqz);
dum=sqrtf(yieqx*yieqx+yieqy*yieqy+yieqz*yieqz);
  yieqx=yieqx/dum;
  yieqy=yieqy/dum;
  yieqz=yieqz/dum;
  zdotz=yitx*yieqx+yity*yieqy+yitz*yieqz;
  dirx=yity*yieqz-yitz*yieqy;
  diry=yitz*yieqx-yitx*yieqz;
  dirz=yitx*yieqy-yity*yieqx;
  dum=sqrtf(dirx*dirx+diry*diry+dirz*dirz);
  if (zdotz > (1.0-1.0E-6)){
    Ytx=0.0; Yty=0.0; Ytz=0.0;
  }
  else{
    if (zdotz < -1.0){
      zdotz=-1.0;
    }
    ang_phi=acosf(zdotz);
    Ytx=-kt*ang_phi*(dirx/dum);
    Yty=-kt*ang_phi*(diry/dum);
    Ytz=-kt*ang_phi*(dirz/dum);

  }
  // Add the bending and twisting torques together
  Yx[mi2]=Ybex+Ytx;
  Yy[mi2]=Ybey+Yty;
  Yz[mi2]=Ybez+Ytz;
}
```

delta_twist_torque.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <math.h>
#include "delta_twist_torque.h"
using namespace std;
__global__ void delta_twist_torque(float **var,int **intVar){
  int nseg=*intVar[7]; float *R21eq=var[45];
  float *R22eq=var[46]; float *R23eq=var[47];
  float *R31eq=var[48]; float *R32eq=var[49];
  float *R33eq=var[50]; float kb=*var[115];
  float kt=*var[116]; float *rx=var[3];
  float *ry=var[4]; float *rz=var[5];
  float *px=var[18]; float *py=var[19];
  float *pz=var[20]; float *R11=var[36];
  float *R12=var[37]; float *R13=var[38];
  float *R21=var[39]; float *R22=var[40];
  float *R23=var[41]; float *Yx=var[54];
  float *Yy=var[55]; float *Yz=var[56];
  float cx,cy,cz,ang_theta,ang_phi;
  // cx,cy,cz-unit vector of (sxx,syy,szz)
  // ang_theta,ang_phi-amount the joint is deformed from equil.
  float zeqx,zeqy,zeqz,zdotz,dirx,diry,dirz,dum;
```

```
float yitx,yity,yitz,yieqx,yieqy,yieqz;
float Ybex,Ybey,Ybez,Ytx,Yty,Ytz;
float pxmi,pymi,pzmi,pxmi_1,pymi_1,pzmi_1;
float R21mi,R22mi,R23mi;
float R11mi_1,R12mi_1,R13mi_1,R21mi_1,R22mi_1,R23mi_1;
float R21eqmi,R22eqmi,R23eqmi,R31eqmi,R32eqmi,R33eqmi;
int m,i,mi,mi2;
int tid=threadIdx.x+blockDim.x*blockIdx.x;
m=tid/(nseg-1);
i=tid-m * (nseg-1)+1;
mi=m*nseg+i;
mi2=m*(nseg+1)+i;
// Calculate joint torques
// Calculate bending and twisting torques,both isotropic
// Find the bending component first
// peq vecotr
R11mi_1=R11[mi-1]; R12mi_1=R12[mi-1]; R13mi_1=R13[mi-1];
R21mi_1=R21[mi-1]; R22mi_1=R22[mi-1]; R23mi_1=R23[mi-1];
R31eqmi=R31eq[mi]; R32eqmi=R32eq[mi]; R33eqmi=R33eq[mi];
pxmi_1=px[mi-1]; pymi_1=py[mi-1]; pzmi_1=pz[mi-1];
zeqx=R11mi_1 * R31eqmi+R21mi_1 * R32eqmi+pxmi_1*R33eqmi;
zeqy=R12mi_1 * R31eqmi+R22mi_1 * R32eqmi+pymi_1*R33eqmi;
zeqz=R13mi_1 * R31eqmi+R23mi_1 * R32eqmi+pzmi_1*R33eqmi;
pxmi=px[mi]; pymi=py[mi]; pzmi=pz[mi];
zdotz=pxmi*zeqx+pymi*zeqy+pzmi*zeqz;
dirx=pymi*zeqz-pzmi*zeqy;
diry=pzmi*zeqx-pxmi*zeqz;
dirz=pxmi*zeqy-pymi*zeqx;
dum=sqrtf(dirx*dirx+diry*diry+dirz*dirz);
if (zdotz > 1.0-1.0E-6){
  Ybex=0.0; Ybey=0.0; Ybez=0.0;
}
else{
  if (zdotz < -1.0){
    zdotz=-1.0;
  }
  ang_theta=acosf(zdotz);
  Ybex=-kb*ang_theta*(dirx/dum);
  Ybey=-kb*ang_theta*(diry/dum);
  Ybez=-kb*ang_theta*(dirz/dum);
}
// Find the twisting component
// first find the c vector
R21eqmi=R21eq[mi]; R22eqmi=R22eq[mi]; R23eqmi=R23eq[mi];
cx=rx[mi]-rx[mi-1];
cy=ry[mi]-ry[mi-1];
cz=rz[mi]-rz[mi-1];
dum=sqrtf(cx*cx+cy*cy+cz*cz);
cx=cx/dum;
cy=cy/dum;
cz=cz/dum;
zeqx=R11mi_1*R21eqmi+R21mi_1*R22eqmi+pxmi_1*R23eqmi;
zeqy=R12mi_1*R21eqmi+R22mi_1*R22eqmi+pymi_1*R23eqmi;
zeqz=R13mi_1*R21eqmi+R23mi_1*R22eqmi+pzmi_1*R23eqmi;
// identify the perpendicular components of y and yeq
R21mi=R21[mi]; R22mi=R22[mi]; R23mi=R23[mi];
yitx=R21mi-cx*(cx*R21mi+cy*R22mi+cz*R23mi);
yity=R22mi-cy*(cx*R21mi+cy*R22mi+cz*R23mi);
yitz=R23mi-cz*(cx*R21mi+cy*R22mi+cz*R23mi);
dum=sqrtf(yitx*yitx+yity*yity+yitz*yitz);
yitx=yitx/dum;
yity=yity/dum;
yitz=yitz/dum;
yieqx=zeqx-cx*(cx*zeqx+cy*zeqy+cz*zeqz);
yieqy=zeqy-cy*(cx*zeqx+cy*zeqy+cz*zeqz);
yieqz=zeqz-cz*(cx*zeqx+cy*zeqy+cz*zeqz);
dum=sqrtf(yieqx*yieqx+yieqy*yieqy+yieqz*yieqz);
yieqx=yieqx/dum;
yieqy=yieqy/dum;
yieqz=yieqz/dum;
zdotz=yitx*yieqx+yity*yieqy+yitz*yieqz;
dirx=yity*yieqz-yitz*yieqy;
diry=yitz*yieqx-yitx*yieqz;
dirz=yitx*yieqy-yity*yieqx;
dum=sqrtf(dirx*dirx+diry*diry+dirz*dirz);
if (zdotz > (1.0-1.0E-6)){
  Ytx=0.0; Yty=0.0; Ytz=0.0;
}
else{
  if (zdotz < -1.0){
    zdotz=-1.0;
  }
  ang_phi=acosf(zdotz);
  Ytx=-kt*ang_phi*(dirx/dum);
  Yty=-kt*ang_phi*(diry/dum);
  Ytz=-kt*ang_phi*(dirz/dum);

}
// Add the bending and twisting torques together
Yx[mi2]=Ybex+Ytx;
Yy[mi2]=Ybey+Yty;
Yz[mi2]=Ybez+Ytz;
}
```

---

delta_twist_zero.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <math.h>
#include "delta_twist_zero.h"
using namespace std;
__global__ void delta_twist_zero(float **var, int **intVar){
  int *ncnt = intVar[1]; int *ncpf = intVar[2];
  int nfib = *intVar[6]; int nseg = *intVar[7];
```

```
  int *bin = intVar[12]; int nxbin = *intVar[15];
  int nybin = *intVar[16]; int nzbin = *intVar[17];
  int *status = intVar[19]; int *nc = intVar[21];
  float *Yx = var[54]; float *Yy = var[55];
  float *Yz = var[56]; float *fx = var[60];
  float *fy = var[61]; float *fz = var[62];
  float *tx = var[63]; float *ty = var[64];
  float *tz = var[65]; float *fcx = var[66];
  float *fcy = var[67]; float *fcz = var[68];
  float *tcx = var[69]; float *tcy = var[70];
  float *tcz = var[71]; float *Stress = var[165];
  int tid = threadIdx.x + blockDim.x * blockIdx.x;
  int tid2 = threadIdx.x + blockDim.x * blockIdx.x;
  int tid5 = threadIdx.x + blockDim.x * blockIdx.x;
  int tid8 = threadIdx.x + blockDim.x * blockIdx.x;
  status[tid5] = 0; fx[tid5] = 0.0;
  fy[tid5] = 0.0; fz[tid5] = 0.0;
  tx[tid5] = 0.0; ty[tid5] = 0.0;
  tz[tid5] = 0.0; fcx[tid5] = 0.0;
  fcy[tid5] = 0.0; fcz[tid5] = 0.0;
  tcx[tid5] = 0.0; tcy[tid5] = 0.0;
  tcz[tid5] = 0.0; nc[tid5] = 0;
  ncpf[tid5] = 0; ncnt[tid5] = 0;
  while (tid2 < nfib*(nseg + 1)){
    Yx[tid2] = 0.0;
    Yy[tid2] = 0.0;
    Yz[tid2] = 0.0;
    tid2 += blockDim.x*gridDim.x;
  }
  while (tid8 < nxbin*nybin*nzbin){
    bin[tid8] = 0;
    tid8 += blockDim.x*gridDim.x;
  }
  if (tid == 0){
    *intVar[3] = 0; // num_groups
    *intVar[4] = 0; // overs
    *intVar[5] = 0; // total_contacts
  }
  if (tid < 6)
    Stress[tid] = 0.0;
}
```

friction_3x3.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <math.h>
#include "friction_3x3.h"
using namespace std;
__global__ void friction_3x3(float **var, int **intVar){
  int *ifiber=intVar[0]; int *ncnt=intVar[1];
  int num_groups=*intVar[3]; int maxCon=*intVar[25];
  int maxGr=*intVar[27]; int *groupId=intVar[34];
  float *uxfl=var[30]; float sidex=*var[119];
  float *fx=var[60]; float *fy=var[61];
  float *fz=var[62]; float *tx=var[63];
  float *ty=var[64]; float *tz=var[65];
  float *A11=var[81]; float *A12=var[82];
  float *A13=var[83]; float *A23=var[84];
  float *A22=var[85]; float *A33=var[86];
  float *C11=var[87]; float *C12=var[88];
  float *C13=var[89]; float *C23=var[90];
  float *C22=var[91]; float *C33=var[92];
  float C1=*var[94]; float mu_stat=*var[139];
  float fstar=*var[144]; float fact=*var[145];
  float Astar=*var[146]; float decatt=*var[147];
  float *GijxV=var[149]; float *GijyV=var[150];
  float *GijzV=var[151]; float *GjixV=var[152];
  float *GjiyV=var[153]; float *GjizV=var[154];
  float *nxV=var[155]; float *nyV=var[156];
  float *nzV=var[157]; float *gV=var[158];
  float *aix=var[159]; float *aiy=var[160];
  float *aiz=var[161]; float *bix=var[162];
  float *biy=var[163]; float *biz=var[164];
  int tid_iter=threadIdx.x+blockIdx.x *blockDim.x;
  int tid, mi, nj;
  float Gijx, Gijy, Gijz, Gjix, Gjiy, Gjiz;
  float ex1S, ex2S, ex3S, ey1S, ey2S, ey3S;
  float nijx, nijy, nijz, qxS, qyS, qzS, gij;
  float C12mi, C12nj, C13mi, C13nj, C23mi, C23nj;
  float Q[9], A[3][3], P[3], fric0, fric1, fric2;
  float det, forc, fric_sq, dum, fux;
  while (tid_iter < num_groups){
    tid=groupId[tid_iter];
    if (ncnt[tid] < 2){
      mi=ifiber[tid*2*maxGr+0*maxGr];
      nj=ifiber[tid*2*maxGr+1*maxGr];
      nijx=nxV[mi*maxCon];
      nijy=nyV[mi*maxCon];
      nijz=nzV[mi*maxCon];
      Gijx=GijxV[mi*maxCon];
      Gijy=GijyV[mi*maxCon];
      Gijz=GijzV[mi*maxCon];
      Gjix=GjixV[mi*maxCon];
      Gjiy=GjiyV[mi*maxCon];
      Gjiz=GjizV[mi*maxCon];
      gij=gV[mi*maxCon];
      ex1S=1-nijx*nijx;
      ex2S=-nijy*nijx;
      ex3S=-nijz*nijx;
      dum=sqrtf(ex1S*ex1S+ex2S*ex2S+ex3S*ex3S);
      ex1S=ex1S / dum;
      ex2S=ex2S / dum;
      ex3S=ex3S / dum;
```

```
ey1S=nijy*ex3S-nijz*ex2S;
ey2S=nijz*ex1S-nijx*ex3S;
ey3S=nijx*ex2S-nijy*ex1S;
dum=sqrtf(ey1S*ey1S+ey2S*ey2S+ey3S*ey3S);
ey1S=ey1S / dum;
ey2S=ey2S / dum;
ey3S=ey3S / dum;
fux=sidex*roundf((uxfl[mi]-uxfl[nj]) / sidex);
qxS=-(aix[mi]-aix[nj]+fux+(biy[mi]*Gijz-biz[mi]*Gijy)
  - (biy[nj]*Gjiz-biz[nj]*Gjiy));
qyS=-(aiy[mi]-aiy[nj]+(biz[mi]*Gijx-bix[mi]*Gijz)
  - (biz[nj]*Gjix-bix[nj]*Gjiz));
qzS=-(aiz[mi]-aiz[nj]+(bix[mi]*Gijy-biy[mi]*Gijx)
  - (bix[nj]*Gjiy-biy[nj]*Gjix));
C12mi=C12[mi]; C12nj=C12[nj];
C13mi=C13[mi]; C13nj=C13[nj];
C23mi=C23[mi]; C23nj=C23[nj];
Q[0]=A11[mi]+A11[nj] -
  C1*(-Gijz*C22[mi]*Gijz+Gijz*C23mi*Gijy+Gijy*C23mi*Gijz-Gijy*C33
       [mi]*Gijy) -
  C1*(-Gjiz*C22[nj]*Gjiz+Gjiz*C23nj*Gjiy+Gjiy*C23nj*Gjiz-Gjiy*C33
       [nj]*Gjiy);
Q[1]=A12[mi]+A12[nj] -
  C1*(Gijz *C12mi*Gijz-Gijz*C23mi*Gijx-Gijy*C13mi*Gijz+Gijy*C33[
       mi]*Gijx) -
  C1*(Gjiz*C12nj*Gjiz-Gjiz*C23nj*Gjix-Gjiy *C13nj*Gjiz+Gjiy*C33[
       nj]*Gjix);
Q[2]=A13[mi]+A13[nj] -
  C1*(-Gijz *C12mi*Gijy+Gijz*C22[mi]*Gijx+Gijy*C13mi*Gijy-Gijy*
       C23mi*Gijx) -
  C1*(-Gjiz*C12nj*Gjiy+Gjiz*C22[nj]*Gjix+Gjiy*C13nj*Gjiy-Gjiy*
       C23nj*Gjix);
Q[4]=A22[mi]+A22[nj] -
  C1*(-Gijz*C11[mi]*Gijz+Gijz*C13mi*Gijx+Gijx*C13mi* Gijz-Gijx*
       C33[mi]*Gijx) -
  C1*(-Gjiz*C11[nj]*Gjiz+Gjiz*C13nj*Gjix+Gjix*C13nj*Gjiz-Gjix*C33
       [nj]*Gjix);
Q[5]=A23[mi]+A23[nj] -
  C1*(Gijz*C11[mi]*Gijy-Gijz *C12mi* Gijx-Gijx*C13mi*Gijy+Gijx*
       C23mi*Gijx) -
  C1*(Gjiz*C11[nj]*Gjiy-Gjiz *C12nj*Gjix-Gjix*C13nj*Gjiy+Gjix*C23
       [nj]*Gjix);
Q[8]=A33[mi]+*(A33+nj) -
  C1*(-Gijy*C11[mi]*Gijy+Gijy*C12mi*Gijx+Gijx*C12mi*Gijy-Gijx*C22
       [mi]*Gijx) -
  C1*(-Gjiy*C11[nj]*Gjiy+Gjiy*C12nj*Gjix+Gjix*C12nj*Gjiy-Gjix*C22
       [nj]*Gjix);
// constrict to plane of contact
A[0][0]=Q[0]*ex1S+Q[1]*ex2S+Q[2]*ex3S;
A[0][1]=Q[1]*ex1S+Q[4]*ex2S+Q[5]*ex3S;
A[0][2]=Q[2]*ex1S+Q[5]*ex2S+Q[8]*ex3S;
A[1][0]=Q[0]*ey1S+Q[1]*ey2S+Q[2]*ey3S;
A[1][1]=Q[1]*ey1S+Q[4]*ey2S+Q[5]*ey3S;
A[1][2]=Q[2]*ey1S+Q[5]*ey2S+Q[8]*ey3S;
A[2][0]=nijx;
A[2][1]=nijy;
A[2][2]=nijz;
P[0]=qxS*ex1S+qyS*ex2S+qzS*ex3S;
P[1]=qxS*ey1S+qyS*ey2S+qzS*ey3S;
P[2]=0.0;
det=A[0][0]*(A[1][1]*A[2][2]-A[1][2]*A[2][1])
  - A[0][1]*(A[1][0]*A[2][2]-A[1][2]*A[2][0])
  + A[0][2]*(A[1][0]*A[2][1]-A[1][1]*A[2][0]);
fric0=1 / det*(
  P[0]*(A[1][1]*A[2][2]-A[1][2]*A[2][1])
  - A[0][1]*(P[1]*A[2][2]-A[1][2]*P[2])
  + A[0][2]*(P[1]*A[2][1]-A[1][1]*P[2]));
fric1=1 / det*(
  A[0][0]*(P[1]*A[2][2]-A[1][2]*P[2])
  - P[0]*(A[1][0]*A[2][2]-A[1][2]*A[2][0])
  + A[0][2]*(A[1][0]*P[2]-P[1]*A[2][0]));
fric2=1 / det*(
  A[0][0]*(A[1][1]*P[2]-P[1]*A[2][1])
  - A[0][1]*(A[1][0]*P[2]-P[1]*A[2][0])
  + P[0]*(A[1][0]*A[2][1]-A[1][1]*A[2][0]));
if (!isfinite(fric0) || !isfinite(fric1) || !isfinite(fric2)){
  printf("f3x3 %4d %4d det %15.10f fric %15.10f %15.10f %15.10f\n
       ", mi, nj, det, fric0, fric1, fric2);
  printf("f3x3 %4d %4d %15.10f %15.10f %15.10f\n%15.10f %15.10f
       %15.10f\n%15.10f %15.10f %15.10f\n",
    mi, nj, A[0][0], A[0][1], A[0][2], A[1][0], A[1][1], A[1][2],
         A[2][0], A[2][1], A[2][2]);
}
forc=fstar*expf(-fact*(gij-2.0)) -
  Astar*expf(-decatt*(gij-2.0))*(gij-2.0);
fric_sq=fric0 *fric0+fric1 *fric1+fric2 *fric2;
if ((fric_sq-(mu_stat*forc)*(mu_stat*forc)) > 1.0e-6){
  ncnt[tid]--;
  break;
}
fx[mi]=fric0;
fy[mi]=fric1;
fz[mi]=fric2;
fx[nj]=-fric0;
fy[nj]=-fric1;
fz[nj]=-fric2;
tx[mi]=Gijy*fric2-Gijz*fric1;
ty[mi]=Gijz*fric0-Gijx*fric2;
tz[mi]=Gijx*fric1-Gijy*fric0;
tx[nj]=-Gjiy*fric2+Gjiz*fric1;
ty[nj]=-Gjiz*fric0+Gjix*fric2;
tz[nj]=-Gjix*fric1+Gjiy*fric0;
atomicAdd(intVar[5], 1); // total_contacts
}
tid_iter += blockDim.x*gridDim.x;
}
```

```
}
```

friction_update.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda.h>
#include "friction_update.h"
__global__ void friction_update(int tid,int blasId,float **var,int **
    intVar){
  int maxCon = *intVar[25]; int *clist_pos = intVar[26];
  int maxGr = *intVar[27]; int *ifiber = intVar[0];
  float *fx = var[60]; float *fy = var[61];
  float *fz = var[62]; float *tx = var[63];
  float *ty = var[64]; float *tz = var[65];
  float *P_fric = var[135]; float *GijxV = var[149];
  float *GijyV = var[150]; float *GijzV = var[151];
  float *GjixV = var[152]; float *GjiyV = var[153];
  float *GjizV = var[154];
  float Gijx,Gijy,Gijz,Gjix,Gjiy,Gjiz;
  float P_fric0,P_fric1,P_fric2;
  int mi,nj;
  int ic = threadIdx.x;
  int pos = clist_pos[tid*maxGr+ic];
  mi = ifiber[tid*2*maxGr+0*maxGr+ic];
  nj = ifiber[tid*2*maxGr+1*maxGr+ic];
  P_fric0 = P_fric[blasId*3*maxGr+3*ic+0];
  P_fric1 = P_fric[blasId*3*maxGr+3*ic+1];
  P_fric2 = P_fric[blasId*3*maxGr+3*ic+2];
  Gijx = GijxV[mi*maxCon+pos];
  Gijy = GijyV[mi*maxCon+pos];
  Gijz = GijzV[mi*maxCon+pos];
  Gjix = GjixV[mi*maxCon+pos];
  Gjiy = GjiyV[mi*maxCon+pos];
  Gjiz = GjizV[mi*maxCon+pos];
  atomicAdd(fx+mi,P_fric0);
  atomicAdd(fy+mi,P_fric1);
  atomicAdd(fz+mi,P_fric2);
  atomicAdd(fx+nj,-P_fric0);
  atomicAdd(fy+nj,-P_fric1);
  atomicAdd(fz+nj,-P_fric2);
  atomicAdd(tx+mi,Gijy*P_fric2 - Gijz*P_fric1);
  atomicAdd(ty+mi,Gijz*P_fric0 - Gijx*P_fric2);
  atomicAdd(tz+mi,Gijx*P_fric1 - Gijy*P_fric0);
  atomicAdd(tx+nj,-Gjiy*P_fric2+Gjiz*P_fric1);
  atomicAdd(ty+nj,-Gjiz*P_fric0+Gjix*P_fric2);
  atomicAdd(tz+nj,-Gjix*P_fric1+Gjiy*P_fric0);
  if (!isfinite(P_fric0) || !isfinite(P_fric1) || !isfinite(P_fric2)){
    printf("f update %6d %4d %4d %15.10f %15.10f %15.10f\n",tid,mi,nj,
        P_fric0,P_fric1,P_fric2);
  }
```

```
}
```

group.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <math.h>
#include "group.h"
using namespace std;
__global__ void group(float **var, int **intVar){
  int *ifiber=intVar[0]; int *ncnt=intVar[1];
  int *ncpf=intVar[2]; int nseg=*intVar[7];
  int *clist=intVar[18]; int *status=intVar[19];
  int *lead_clist=intVar[20]; int *nc=intVar[21];
  int maxCon=*intVar[25]; int *clist_pos=intVar[26];
  int maxGr=*intVar[27]; int *groupId=intVar[34];
  int pos, fib, N, ii, jj, ic;
  int segold1, segold2, segnew1, segnew2;
  bool add;
  int tid=threadIdx.x+blockIdx.x*blockDim.x;
  if (ncpf[tid] != 0 && status[tid] == 0){
    pos=atomicAdd(intVar[3], 1); // num_groups
    groupId[pos]=tid;
    // boss fiber compiles list
    // find total contacts in group and create list of contacts
    N=ncpf[tid];
    for (ii=0; ii < N; ii++){
      add=true;
      for (ic=0; ic < ncnt[tid]; ic++){
        segnew1=clist[tid*maxCon+ii];
        segold1=clist[tid*maxCon+ic];
        // if at joints, don't add to group
        if (segnew1 == segold1+1 && segnew1%nseg != 0){
          add=false;
          break;
        }
        if (segold1 == segnew1+1 && segold1%nseg != 0){
          add=false;
          break;
        }
      }
      if (add){
        ifiber[tid*2*maxGr+0*maxGr+ncnt[tid]]=tid;
        ifiber[tid*2*maxGr+1*maxGr+ncnt[tid]]=clist[tid*maxCon+ii];
        clist_pos[tid*maxGr+ncnt[tid]]=ii;
        ncnt[tid]++;
      }
    }
    for (ii=0; ii < nc[tid]; ii++){
      fib=lead_clist[tid*maxGr+ii];
      segnew1=fib;
      N=ncpf[fib];
```

```
for (jj=0; jj < N; jj++){
  add=true;
  if (fib < clist[fib*maxCon+jj]){
    segnew2=clist[fib*maxCon+jj];
    for (ic=0; ic < ncnt[tid]; ic++){
      segold1=ifiber[tid*2*maxGr+0*maxGr+ic];
      segold2=ifiber[tid*2*maxGr+1*maxGr+ic];
      if (segold2 == segnew2){
        if (segnew1 == segold1+1 && segnew1%nseg != 0){
          add=false;
          break;
        }
        if (segold1 == segnew1+1 && segold1%nseg != 0){
          add=false;
          break;
        }
      }
      else if (segold1 == segnew1){
        if (segnew2 == segold2+1 && segnew2%nseg != 0){
          add=false;
          break;
        }
        if (segold2 == segnew2+1 && segold2%nseg != 0){
          add=false;
          break;
        }
      }
      else if (segold1 == segnew2){
        if (segnew1 == segold2+1 && segnew1%nseg != 0){
          add=false;
          break;
        }
        if (segold2 == segnew1+1 && segold2%nseg != 0){
          add=false;
          break;
        }
      }
      else if (segold2 == segnew1){
        if (segnew2 == segold1+1 && segnew2%nseg != 0){
          add=false;
          break;
        }
        if (segold1 == segnew2+1 && segold1%nseg != 0){
          add=false;
          break;
        }
      }
    }
    if (add){
      ifiber[tid*2*maxGr+0*maxGr+ncnt[tid]]=fib;
      ifiber[tid*2*maxGr+1*maxGr+ncnt[tid]]=clist[fib*maxCon+jj];
      clist_pos[tid*maxGr+ncnt[tid]]=jj;
      ncnt[tid]++;
        }
      }
    }
    if (ncnt[tid] >= maxGr){
      printf("error: increase maxGr ncnt[%8d]=%3d\n", tid, ncnt[tid]);
    }
  }
}
```

---

initialize.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cuda.h>
#include "initialize.h"
using namespace std;
__global__ void initialize(float **var, int **intVar){
  int nfib=*intVar[6]; int nseg=*intVar[7];
  float *q0=var[6]; float *q1=var[7];
  float *q2=var[8]; float *q3=var[9];
  float *q0dot=var[10]; float *q1dot=var[11];
  float *q2dot=var[12]; float *q3dot=var[13];
  float *qe0=var[14]; float *qe1=var[15];
  float *qe2=var[16]; float *qe3=var[17];
  float *px=var[18]; float *py=var[19];
  float *pz=var[20]; float *ux=var[27];
  float *uy=var[28]; float *uz=var[29];
  float *uxfl=var[30]; float *uyfl=var[31];
  float *uzfl=var[32]; float *wx=var[33];
  float *wy=var[34]; float *wz=var[35];
  float *R11=var[36]; float *R12=var[37];
  float *R13=var[38]; float *R21=var[39];
  float *R22=var[40]; float *R23=var[41];
  float *R11eq=var[42]; float *R12eq=var[43];
  float *R13eq=var[44]; float *R21eq=var[45];
  float *R22eq=var[46]; float *R23eq=var[47];
  float *R31eq=var[48]; float *R32eq=var[49];
  float *R33eq=var[50]; float *Xx=var[51];
  float *Xy=var[52]; float *Xz=var[53];
  float *Yx=var[54]; float *Yy=var[55];
  float *Yz=var[56]; float *Ybx=var[57];
  float *Yby=var[58]; float *Ybz=var[59];
  float *tx=var[63]; float *ty=var[64];
  float *tz=var[65]; float *fcx=var[66];
  float *fcy=var[67]; float *fcz=var[68];
  float *tcx=var[69]; float *tcy=var[70];
  float *tcz=var[71]; float *fbx=var[72];
  float *fby=var[73]; float *fbz=var[74];
  float *tbx=var[75]; float *tby=var[76];
  float *tbz=var[77]; float elf=*var[148];
  float *thetaeq=var[136]; float *phieq=var[137];
```

```
float q0mi, q1mi, q2mi, q3mi;
float pxmi, pymi, pzmi, thetaeqmi, phieqmi;
int mi=threadIdx.x+blockIdx.x*blockDim.x;
int tid=threadIdx.x+blockIdx.x*blockDim.x;
int m, i;
float dum;
// temporary variables
m=mi / nseg;
i=mi - m*nseg;
// zero varaibles
ux[mi]=0.0; uy[mi]=0.0; uz[mi]=0.0;
wx[mi]=0.0; wy[mi]=0.0; wz[mi]=0.0;
q0dot[mi]=0.0; q1dot[mi]=0.0; q2dot[mi]=0.0; q3dot[mi]=0.0;
qe0[mi]=0.0; qe1[mi]=0.0; qe2[mi]=0.0; qe3[mi]=0.0;
uxfl[mi]=0.0; uyfl[mi]=0.0; uzfl[mi]=0.0;
fcx[mi]=0.0; fcy[mi]=0.0; fcz[mi]=0.0;
tx[mi]=0.0; ty[mi]=0.0; tz[mi]=0.0;
tcx[mi]=0.0; tcy[mi]=0.0; tcz[mi]=0.0;
q0mi=q0[mi]; q1mi=q1[mi]; q2mi=q2[mi]; q3mi=q3[mi];
// Normalize Euler Parameters
dum=sqrtf(q0mi*q0mi+q1mi*q1mi+q2mi*q2mi+q3mi*q3mi);
q0mi /= dum; q1mi /= dum;
q2mi /= dum; q3mi /= dum;
q0[mi]=q0mi; q1[mi]=q1mi; q2[mi]=q2mi; q3[mi]=q3mi;
// Find rotation matrix
R11[mi]=2.0*(q0mi*q0mi+q1mi*q1mi) - 1.0;
R12[mi]=2.0*(q1mi*q2mi+q0mi*q3mi);
R13[mi]=2.0*(q1mi*q3mi - q0mi*q2mi);
R21[mi]=2.0*(q1mi*q2mi - q0mi*q3mi);
R22[mi]=2.0*(q0mi*q0mi+q2mi*q2mi) - 1.0;
R23[mi]=2.0*(q3mi*q2mi+q0mi*q1mi);
pxmi=2.0*(q1mi*q3mi+q0mi*q2mi);
pymi=2.0*(q3mi*q2mi - q0mi*q1mi);
pzmi=2.0*(q0mi*q0mi+q3mi*q3mi) - 1.0;
dum=sqrtf(pxmi*pxmi+pymi*pymi+pzmi*pzmi);
pxmi /= dum; pymi /= dum; pzmi /= dum;
px[mi]=pxmi; py[mi]=pymi; pz[mi]=pzmi;
// Assign body forces and torques
fbx[mi]=0.0; fby[mi]=0.0; fbz[mi]=0.0;
tbx[mi]=elf*pzmi*pymi;
tby[mi]=-elf*pzmi*pxmi;
tbz[mi]=0.0;
// Make the equilibrium rotation matrix
if (i > 0) {
  thetaeqmi=thetaeq[mi]; phieqmi=phieq[mi];
  R11eq[mi]=cosf(thetaeqmi)*cosf(phieqmi);
  R12eq[mi]=cosf(thetaeqmi)*sinf(phieqmi);
  R13eq[mi]=-sinf(thetaeqmi);
  R21eq[mi]=-sinf(phieqmi);
  R22eq[mi]=cosf(phieqmi);
  R23eq[mi]=0.0;
  R31eq[mi]=sinf(thetaeqmi)*cosf(phieqmi);
  R32eq[mi]=sinf(thetaeqmi)*sinf(phieqmi);
  R33eq[mi]=cosf(thetaeqmi);
 }
 while (tid < nfib*(nseg+1)){
  Xx[tid]=0.0; Xy[tid]=0.0; Xz[tid]=0.0;
  Yx[tid]=0.0; Yy[tid]=0.0; Yz[tid]=0.0;
  Ybx[tid]=0.0; Yby[tid]=0.0; Ybz[tid]=0.0;
  tid += blockDim.x * gridDim.x;
 }
}
```

---

lead.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <math.h>
#include "lead.h"
using namespace std;
__global__ void lead(float **var, int **intVar){
 int tid=threadIdx.x+blockIdx.x*blockDim.x;
 int *ncpf=intVar[2];
 // ncpf[tid] - number of segments in direct contact with segment tid
 int *clist=intVar[18];
 // clist[tid*maxCon+i] - the i th fiber in contact with segment tid
 int *status=intVar[19];
 // status[tid] - whether the list of fiber tid has been accessed
 int *lead_clist=intVar[20];
 // lead_clist[tid*maxGr+k] - the k th fiber in group led by tid
 int *nc=intVar[21];
 // nc[tid] - number of fiber in group led by tid
 int maxCon=*intVar[25];
 // maxCon - maximum number of fibers contacting one fiber
 int maxGr=*intVar[27];
 // maxGr - maximum number of fibers in a group
 int pos, fiber, locfiber, i, j, k;
 bool add;
 pos=0;
 add=false;
 // goes through segments in the lead fiber's list of direct contact
 for (i=0; i < ncpf[tid]; i++){
  fiber=clist[tid*maxCon+i];
  if (fiber < tid){
   status[tid]=1;
   return;
  }
  lead_clist[maxGr*tid+nc[tid]]=clist[tid*maxCon+i];
  nc[tid]++;
 }
 // goes through the list of direct contact for all fibers in
      lead_clist
 while (pos != nc[tid]){
  fiber=lead_clist[maxGr*tid+pos];
  for (j=0; j < ncpf[fiber]; j++){
```

```
      locfiber=clist[fiber*maxCon+j];
      if (locfiber < tid){
        status[tid]=1;
        return;
      }
      if (locfiber == tid){
        continue;
      }
      add=true;
      for (k=0; k < nc[tid]; k++){
        if (locfiber == lead_clist[maxGr*tid+k]){
          add=false;
          break;
        }
      }
      if (add){
        lead_clist[maxGr*tid+nc[tid]]=locfiber;
        nc[tid]++;
        if (nc[tid] >= maxGr){
          printf("error in lead: increase maxGr\n");
        }
      }
    }
    pos++;
  }
}
```

lead_list.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cublas_v2.h>
#include "lead_list.h"
#include "cublas_initialize.h"
#include "friction_update.h"
// kernels to build contact matrix for groups with more than two
     segments
__global__ void friction(int tid,int gc,float **var,int **intVar);
__device__ void Jmat(int tid,float **var,int **intVar,float *Asub,int
     mi,int nj,float nijx,float nijy,float nijz,
  float Gijx,float Gijy,float Gijz,float Gjix,float Gjiy,float Gjiz,
  float ex1,float ex2,float ex3,float ey1,float ey2,float ey3);
__device__ void Jdiag(int tid,float **var,int **intVar,float *Asub,int
      rs,
  float G1x,float G1y,float G1z,float G2x,float G2y,float G2z,
  float ex1,float ex2,float ex3,float ey1,float ey2,float ey3,float
       posneg);
// solves for friction forces
__global__ void lead_list(float **var,int **intVar,float **AA,float **
     PP,int *Pivot,int *info,cublasHandle_t *handle_cublas,
     cublasStatus_t *status_cublas){
int *ifiber=intVar[0]; int *ncnt=intVar[1];
int num_groups=*intVar[3]; int maxCon=*intVar[25];
int *clist_pos=intVar[26]; int maxGr=*intVar[27];
int *groupId=intVar[34]; float *A_fric=var[134];
float *P_fric=var[135]; float mu_stat=*var[139];
float fstar=*var[144]; float fact=*var[145];
float Astar=*var[146]; float decatt=*var[147];
float *gV=var[158];
float fric_max,fric_sq,forc,gij;
bool break_contact;
int imax,mi,ic,bossFiber,gc,blasId;
gc=threadIdx.x+blockIdx.x *blockDim.x;
blasId=threadIdx.x+blockIdx.x*blockDim.x;
// each thread process a group
while (gc < num_groups){
  bossFiber=groupId[gc];
  // exclude groups where all contact are broekn
  if (ncnt[bossFiber] < 2) {
    gc += blockDim.x*gridDim.x;
    continue;
  }
  // build matrix,solve for and evaluate friction forces
  break_contact=true;
  while (break_contact && ncnt[bossFiber] != 0){
    // build matrix
    friction << <ncnt[bossFiber],ncnt[bossFiber] >> >(bossFiber,
         blasId,var,intVar);
    break_contact=false;
    // identifies which chunk of memory to be use for solve
    AA[blasId]=A_fric+blasId*9*maxGr*maxGr;
    PP[blasId]=P_fric+blasId*3*maxGr;
    status_cublas[blasId]=CUBLAS_STATUS_SUCCESS;
    cudaDeviceSynchronize();
    // solves by calling cublas
    status_cublas[blasId]=cublasSgetrfBatched(handle_cublas[blasId
         ],3*ncnt[bossFiber],AA+blasId,3*ncnt[bossFiber],Pivot+blasId
         *3*maxGr,info+blasId,1);
    cudaDeviceSynchronize();
    if (status_cublas[blasId] != CUBLAS_STATUS_SUCCESS){
      printf("bossFiber %d blasId %d ncnt %d sgetrf stats=%s info %d\
           n",bossFiber,blasId,ncnt[bossFiber],cublasGetErrorString(
           status_cublas[blasId]),info[blasId]);
    }
    status_cublas[blasId]=cublasSgetrsBatched(handle_cublas[blasId],
         CUBLAS_OP_N,3*ncnt[bossFiber],1,(const float**)(AA+blasId)
         ,3*ncnt[bossFiber],Pivot+blasId*3*maxGr,PP+blasId,3*ncnt[
         bossFiber],info+blasId,1);
    cudaDeviceSynchronize();
         if (status_cublas[blasId] != CUBLAS_STATUS_SUCCESS){
      printf("bossFiber %d blasId %d ncnt %d sgetrs stats=%s info %d\
           n",bossFiber,blasId,ncnt[bossFiber],cublasGetErrorString(
           status_cublas[blasId]),info[blasId]);
    }
```

```
    // Loop over all the contacts to calculate force magnitudes
    fric_max=0.0;
    imax=-1;
    for (ic=0; ic < ncnt[bossFiber]; ic++){
      mi=ifiber[bossFiber* 2*maxGr+0*maxGr+ic];
      gij=gV[mi*maxCon+clist_pos[bossFiber*maxGr+ic]];
      // Square of the normal force
      forc=fstar*expf(-fact*(gij-2.0)) -
        Astar*expf(-decatt*(gij-2.0))*(gij-2.0);
      // Square of the friction force
      fric_sq=P_fric[blasId*3*maxGr+3*ic+0]*P_fric[blasId*3*maxGr+3*
           ic+0] +
        P_fric[blasId*3*maxGr+3*ic+1]*P_fric[blasId*3*maxGr+3*ic+1] +
        P_fric[blasId*3*maxGr+3*ic+2]*P_fric[blasId*3*maxGr+3*ic+2];
      // check to see if the magnitude is too great
      if ((fric_sq-(mu_stat*forc)*(mu_stat*forc)) > 1.0e-6 ){
        if ((fric_sq-fric_max) > 1.0e-6) {
          fric_max=fric_sq;
          imax=ic;
        }
      }
    }
    if (imax != -1){
      break_contact=true;
      for (ic=imax; ic < (ncnt[bossFiber]-1); ic++){
        ifiber[bossFiber*2*maxGr+0*maxGr+ic]=ifiber[bossFiber*2*maxGr
            +0*maxGr+ic+1];
        ifiber[bossFiber*2*maxGr+1*maxGr+ic]=ifiber[bossFiber*2*maxGr
            +1*maxGr+ic+1];
        clist_pos[bossFiber*maxGr+ic]=clist_pos[bossFiber*maxGr+ic
            +1];
      }
      ncnt[bossFiber]--;
    }
  }
  // Assign the final friction force to the apporpriate fiber
  if (ncnt[bossFiber] != 0) {
    friction_update << <1,ncnt[bossFiber] >> >(bossFiber,blasId,var,
        intVar);
    atomicAdd(intVar[5],ncnt[bossFiber]); // total_contacts
  }
  gc += blockDim.x*gridDim.x;
  cudaDeviceSynchronize();
 }
}
__global__ void friction(int tid,int blasId,float **var,int **intVar){
#include "math.h"
#include <cublas_v2.h>
 int *ifiber=intVar[0]; int *ncnt=intVar[1];
 int maxCon=*intVar[25]; int *clist_pos=intVar[26];
 int maxGr=*intVar[27]; float *uxfl=var[30];
 float sidex=*var[119]; float *A_fric=var[134];
 float *P_fric=var[135]; float *GijxV=var[149];
 float *GijyV=var[150]; float *GijzV=var[151];
 float *GjixV=var[152]; float *GjiyV=var[153];
 float *GjizV=var[154]; float *nxV=var[155];
 float *nyV=var[156]; float *nzV=var[157];
 float *aix=var[159]; float *aiy=var[160];
 float *aiz=var[161]; float *bix=var[162];
 float *biy=var[163]; float *biz=var[164];
 int row=blockIdx.x;
 int col=threadIdx.x;
 int mi,nj,a,b,size,pos,rs;
 float G1x,G1y,G1z,G2x,G2y,G2z;
 float Asub[9],Gijx,Gijy,Gijz,Gjix,Gjiy,Gjiz;
 float nijx,nijy,nijz,ex1S,ex2S,ex3S;
 float ey1S,ey2S,ey3S,fux,qxS,qyS,qzS,dum;
 int posneg,mark,ii;
 int disp=blasId*9*maxGr*maxGr;
 int dispRHS=blasId*3*maxGr;
 int ind=row;
 size=3*ncnt[tid];
 // obtain parameters
 mi=ifiber[tid*2*maxGr+0*maxGr+row];
 nj=ifiber[tid*2*maxGr+1*maxGr+row];
 a=ifiber[tid*2*maxGr+0*maxGr+col];
 b=ifiber[tid*2*maxGr+1*maxGr+col];
 pos=clist_pos[tid*maxGr+ind];
 nijx=nxV[mi*maxCon+pos];
 nijy=nyV[mi*maxCon+pos];
 nijz=nzV[mi*maxCon+pos];
 Gijx=GijxV[mi*maxCon+pos];
 Gijy=GijyV[mi*maxCon+pos];
 Gijz=GijzV[mi*maxCon+pos];
 Gjix=GjixV[mi*maxCon+pos];
 Gjiy=GjiyV[mi*maxCon+pos];
 Gjiz=GjizV[mi*maxCon+pos];
 ex1S=1-nijx*nijx;
 ex2S=-nijy*nijx;
 ex3S=-nijz*nijx;
 dum=sqrtf(ex1S*ex1S+ex2S*ex2S+ex3S*ex3S);
 ex1S=ex1S/dum; ex2S=ex2S/dum; ex3S=ex3S/dum;
 ey1S=nijy*ex3S-nijz*ex2S;
 ey2S=nijz*ex1S-nijx*ex3S;
 ey3S=nijx*ex2S-nijy*ex1S;
 dum=sqrtf(ey1S*ey1S+ey2S*ey2S+ey3S*ey3S);
 ey1S=ey1S/dum; ey2S=ey2S/dum; ey3S=ey3S/dum;
 // Periodic boundaries correction term
 fux=sidex*roundf((uxfl[mi]-uxfl[nj])/sidex);
 qxS=-(aix[mi]-aix[nj]+fux+(biy[mi]*Gijz-biz[mi]*Gijy)
   - (biy[nj]*Gjiz-biz[nj]*Gjiy));
 qyS=-(aiy[mi]-aiy[nj]+(biz[mi]*Gijx-bix[mi]*Gijz)
   - (biz[nj]*Gjix-bix[nj]*Gjiz));
 qzS=-(aiz[mi]-aiz[nj]+(bix[mi]*Gijy-biy[mi]*Gijx)
   - (bix[nj]*Gjiy-biy[nj]*Gjix));
 // Fill in the actual rhs vector
```

```
if (col == 0){
  P_fric[dispRHS+3*row+0]=qxS*ex1S+qyS*ex2S+qzS*ex3S;
  P_fric[dispRHS+3*row+1]=qxS*ey1S+qyS*ey2S+qzS*ey3S;
  P_fric[dispRHS+3*row+2]=0.0;
}
// build matrix
mark=0; posneg=1;
if (row == col){
  Jmat(tid,var,intVar,Asub,mi,nj,nijx,nijy,nijz,Gijx,Gijy,Gijz,Gjix,
       Gjiy,Gjiz,ex1S,ex2S,ex3S,ey1S,ey2S,ey3S);
  mark=1;
}
else{
  pos=clist_pos[tid*maxGr+col];
  if (mi == a){
    rs=mi;
    posneg=-1.0;
    G1x=Gijx; G1y=Gijy; G1z=Gijz;
    G2x=GijxV[a*maxCon+pos];
    G2y=GijyV[a*maxCon+pos];
    G2z=GijzV[a*maxCon+pos];
    mark=1;
  }
  else if (mi == b){
    rs=mi;
    posneg=1.0;
    G1x=Gijx; G1y=Gijy; G1z=Gijz;
    G2x=GjixV[a*maxCon+pos];
    G2y=GjiyV[a*maxCon+pos];
    G2z=GjizV[a*maxCon+pos];
    mark=1;
  }
  else if (nj == a){
    rs=nj;
    posneg=1.0;
    G1x=Gjix; G1y=Gjiy; G1z=Gjiz;
    G2x=GijxV[a*maxCon+pos];
    G2y=GijyV[a*maxCon+pos];
    G2z=GijzV[a*maxCon+pos];
    mark=1;
  }
  else if (nj == b){
    rs=nj;
    posneg=-1.0;
    G1x=Gjix; G1y=Gjiy; G1z=Gjiz;
    G2x=GjixV[a*maxCon+pos];
    G2y=GjiyV[a*maxCon+pos];
    G2z=GjizV[a*maxCon+pos];
    mark=1;
  }
  if (mark == 1){
    Jdiag(tid,var,intVar,Asub,rs,G1x,G1y,G1z,G2x,
      G2y,G2z,ex1S,ex2S,ex3S,ey1S,ey2S,ey3S,posneg);
  }
  else{
    // zero this block
    for (ii=0; ii < 9; ii++){
      Asub[ii]=0;
    }
  }
}
// each block update shared memory,cublas is column major
pos=3*col*size+3*row;
A_fric[disp+pos+0*size+0]=Asub[0];
A_fric[disp+pos+0*size+1]=Asub[3];
A_fric[disp+pos+0*size+2]=Asub[6];
A_fric[disp+pos+1*size+0]=Asub[1];
A_fric[disp+pos+1*size+1]=Asub[4];
A_fric[disp+pos+1*size+2]=Asub[7];
A_fric[disp+pos+2*size+0]=Asub[2];
A_fric[disp+pos+2*size+1]=Asub[5];
A_fric[disp+pos+2*size+2]=Asub[8];
}
__device__ void Jdiag(int tid,float **var,int **intVar,float *Asub,int
      rs,
  float G1x,float G1y,float G1z,float G2x,float G2y,float G2z,
  float ex1,float ex2,float ex3,float ey1,float ey2,float ey3,float
       posneg){
  float *A11=var[81]; float *A12=var[82];
  float *A13=var[83]; float *A23=var[84];
  float *A22=var[85]; float *A33=var[86];
  float *C11=var[87]; float *C12=var[88];
  float *C13=var[89]; float *C23=var[90];
  float *C22=var[91]; float *C33=var[92];
  float C1=*var[94];
  float Q[9],C11rs,C12rs,C13rs;
  float C23rs,C22rs,C33rs;
  C11rs=C11[rs]; C12rs=C12[rs]; C13rs=C13[rs];
  C23rs=C23[rs]; C22rs=C22[rs]; C33rs=C33[rs];
  posneg=-1.0*posneg;
  Q[0]=posneg*(A11[rs]-C1*(-G1z*C22rs*G2z +
    G1z*C23rs*G2y+G1y*C23rs*G2z-G1y*C33rs*G2y));
  Q[1]=posneg*(A12[rs]-C1*(G1z*C12rs*G2z -
    G1z*C23rs*G2x-G1y*C13rs*G2z+G1y*C33rs*G2x));
  Q[2]=posneg*(A13[rs]-C1*(-G1z*C12rs*G2y +
    G1z*C22rs*G2x+G1y*C13rs*G2y-G1y*C23rs*G2x));
  Q[3]=posneg*(A12[rs]-C1*(G1z*C12rs*G2z -
    G1z*C13rs*G2y-G1x*C23rs*G2z+G1x*C33rs*G2y));
  Q[4]=posneg*(A22[rs]-C1*(-G1z*C11rs*G2z +
    G1z*C13rs*G2x+G1x*C13rs*G2z-G1x*C33rs*G2x));
  Q[5]=posneg*(A23[rs]-C1*(G1z*C11rs*G2y -
    G1z*C12rs*G2x-G1x*C13rs*G2y+G1x*C23rs*G2x));
  Q[6]=posneg*(A13[rs]-C1*(-G1y*C12rs*G2z +
    G1y*C13rs*G2y+G1x*C22rs*G2z-G1x*C23rs*G2y));
  Q[7]=posneg*(A23[rs]-C1*(G1y*C11rs*G2z -
    G1y*C13rs*G2x-G1x*C12rs*G2z+G1x*C23rs*G2x));
```

```
  Q[8]=posneg*(A33[rs]-C1*(-G1y*C11rs*G2y +
    G1y*C12rs*G2x+G1x*C12rs*G2y-G1x*C22rs*G2x));
  Asub[0]=Q[0]*ex1+Q[3]*ex2+Q[6]*ex3;
  Asub[1]=Q[1]*ex1+Q[4]*ex2+Q[7]*ex3;
  Asub[2]=Q[2]*ex1+Q[5]*ex2+Q[8]*ex3;
  Asub[3]=Q[0]*ey1+Q[3]*ey2+Q[6]*ey3;
  Asub[4]=Q[1]*ey1+Q[4]*ey2+Q[7]*ey3;
  Asub[5]=Q[2]*ey1+Q[5]*ey2+Q[8]*ey3;
  Asub[6]=0; Asub[7]=0; Asub[8]=0;
}
__device__ void Jmat(int tid,float **var,int **intVar,float *Asub,int
     mi,int nj,float nijx,float nijy,float nijz,
  float Gijx,float Gijy,float Gijz,float Gjix,float Gjiy,float Gjiz,
       float ex1,float ex2,float ex3,float ey1,float ey2,float ey3){
  float *A11=var[81]; float *A12=var[82];
  float *A13=var[83]; float *A23=var[84];
  float *A22=var[85]; float *A33=var[86];
  float *C11=var[87]; float *C12=var[88];
  float *C13=var[89]; float *C23=var[90];
  float *C22=var[91]; float *C33=var[92];
  float C1=*var[94];
  float Q[9],C12mi,C12nj,C13mi,C13nj,C23mi,C23nj;
  C12mi=C12[mi]; C12nj=C12[nj];
  C13mi=C13[mi]; C13nj=C13[nj];
  C23mi=C23[mi]; C23nj=C23[nj];
  Q[0]=A11[mi]+A11[nj] -
    C1*(-Gijz*C22[mi]*Gijz+Gijz*C23mi*Gijy+Gijy*C23mi*Gijz-Gijy*C33[mi
         ]*Gijy) -
    C1*(-Gjiz*C22[nj]*Gjiz+Gjiz*C23nj*Gjiy+Gjiy*C23nj*Gjiz-Gjiy*C33[nj
         ]*Gjiy);
  Q[1]=A12[mi]+A12[nj] -
    C1*(Gijz *C12mi*Gijz-Gijz*C23mi*Gijx-Gijy*C13mi*Gijz+Gijy*C33[mi]*
         Gijx) -
    C1*(Gjiz*C12nj*Gjiz-Gjiz*C23nj*Gjix-Gjiy *C13nj*Gjiz+Gjiy*C33[nj]*
         Gjix);
  Q[2]=A13[mi]+A13[nj] -
    C1*(-Gijz *C12mi*Gijy+Gijz*C22[mi]*Gijx+Gijy*C13mi*Gijy-Gijy*C23mi*
         Gijx) -
    C1*(-Gjiz*C12nj*Gjiy+Gjiz*C22[nj]*Gjix+Gjiy*C13nj*Gjiy-Gjiy*C23nj*
         Gjix);
  Q[4]=A22[mi]+A22[nj] -
    C1*(-Gijz*C11[mi]*Gijz+Gijz*C13mi*Gijx+Gijx*C13mi* Gijz-Gijx*C33[mi
         ]*Gijx) -
    C1*(-Gjiz*C11[nj]*Gjiz+Gjiz*C13nj*Gjix+Gjix*C13nj*Gjiz-Gjix*C33[nj
         ]*Gjix);
  Q[5]=A23[mi]+A23[nj] -
    C1*(Gijz*C11[mi]*Gijy-Gijz *C12mi* Gijx-Gijx*C13mi*Gijy+Gijx*C23mi*
         Gijx) -
    C1*(Gjiz*C11[nj]*Gjiy-Gjiz *C12nj*Gjix-Gjix*C13nj*Gjiy+Gjix*C23[nj
         ]*Gjix);
  Q[8]=A33[mi]+*(A33+nj) -
    C1*(-Gijy*C11[mi]*Gijy+Gijy*C12mi*Gijx+Gijx*C12mi*Gijy-Gijx*C22[mi
         ]*Gijx) -
    C1*(-Gjiy*C11[nj]*Gjiy+Gjiy*C12nj*Gjix+Gjix*C12nj*Gjiy-Gjix*C22[nj
         ]*Gjix);
  // constrict to plane of contact
  Asub[0]=Q[0]*ex1+Q[1]*ex2+Q[2]*ex3;
  Asub[1]=Q[1]*ex1+Q[4]*ex2+Q[5]*ex3;
  Asub[2]=Q[2]*ex1+Q[5]*ex2+Q[8]*ex3;
  Asub[3]=Q[0]*ey1+Q[1]*ey2+Q[2]*ey3;
  Asub[4]=Q[1]*ey1+Q[4]*ey2+Q[5]*ey3;
  Asub[5]=Q[2]*ey1+Q[5]*ey2+Q[8]*ey3;
  Asub[6]=nijx; Asub[7]=nijy; Asub[8]=nijz;
}
```

lees_edwards.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <math.h>
#include "lees_edwards.h"
using namespace std;
__global__ void lees_edwards(float **var, int **intVar){
  float *delta_rx = var[138]; float sidex = *var[119];
  float dt = *var[117];
  *delta_rx = *delta_rx + sidex*dt;
  *delta_rx = *delta_rx - lroundf(*delta_rx / sidex)*sidex;
}
```

link.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <math.h>
#include "link.h"
using namespace std;
__global__ void link(float **var, int **intVar){
  const int npcn=2500;
  int nseg=*intVar[7]; int *bin=intVar[12];
  int *list=intVar[13]; int *bnei=intVar[14];
  int nxbin=*intVar[15]; int nybin=*intVar[16];
  int nzbin=*intVar[17]; int *bnum=intVar[31];
  int *potCon=intVar[32]; int *potConSize=intVar[33];
  int mi=blockIdx.x;
  int tid=threadIdx.x;
  int miBin=bnum[mi];
  int nextBin=bnei[miBin+tid*nxbin*nybin*nzbin];
  int nextTot=bin[nextBin];
  int nj, i, pos;
  // zeroes the number of potential contacts
  if (tid == 0){
    potConSize[mi]=0;
```

```
  }
  __syncthreads();
  // add segments in neighboring bins to potCon
  for (i=0; i < nextTot; i++){
    nj=list[nextBin+i*nxbin*nybin*nzbin];
    // skip when segments are connected
    if (mi >= nj){
      continue;
    }
    if ((mi-(mi/nseg)*nseg) != 0 && nj == mi-1){
      continue;
    }
    if ((nj-(nj/nseg)*nseg) != 0 && mi == nj-1){
      continue;
    }
    if ((mi-(mi/nseg)*nseg) != nseg-1 && nj == mi+1){
      continue;
    }
    if ((nj-(nj/nseg)*nseg) != nseg-1 && mi == nj+1){
      continue;
    }
    pos=atomicAdd(potConSize+mi, 1);
    if (pos >= npcn-1) printf("allocate more space for potCon pos %4d\n
        ", pos);
    potCon[mi*npcn+pos]=nj;
  }
}
```

regrowFib.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <cuda.h>
#include "regrowFib.h"
using namespace std;
__global__ void regrowFib(float **var, int **intVar){
  int nseg = *intVar[7]; float rp = *var[114];
  float *rx = var[3]; float *ry = var[4];
  float *rz = var[5]; float *px = var[18];
  float *py = var[19]; float *pz = var[20];
  float pjx, pjy, pjz;
  int mi, m, i, mth;
  int tid = threadIdx.x+blockDim.x*blockIdx.x;
  // calculate fiber and segment indices
  m = tid / (nseg-1);
  i = tid - m * (nseg-1)+1;
  mi = m*nseg+i;
  // Regrow the fibers
  pjx = 0.0; pjy = 0.0; pjz = 0.0;
  for (mth = 1; mth<(i); mth++){
    pjx = pjx+px[m*nseg+mth];
    pjy = pjy+py[m*nseg+mth];
    pjz = pjz+pz[m*nseg+mth];
  }
  rx[mi] = rx[m*nseg]+rp*px[m*nseg]+2.0*rp*pjx+rp*px[mi];
  ry[mi] = ry[m*nseg]+rp*py[m*nseg]+2.0*rp*pjy+rp*py[mi];
  rz[mi] = rz[m*nseg]+rp*pz[m*nseg]+2.0*rp*pjz+rp*pz[mi];
}
```

regrowSeg.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <cuda.h>
#include "regrowSeg.h"
using namespace std;
__global__ void regrowSeg(float **var, int **intVar){
  int nseg = *intVar[7]; float rp = *var[114];
  float *rcmx = var[0]; float *rcmy = var[1];
  float *rcmz = var[2]; float *rx = var[3];
  float *ry = var[4]; float *rz = var[5];
  float *px = var[18]; float *py = var[19];
  float *pz = var[20];
  float pjx, pjy, pjz, pkx, pky, pkz;
  int mi, m, k, j;
  m = threadIdx.x + blockDim.x * blockIdx.x;
  mi = m*nseg;
  // Regrow the fibers
  pjx = 0.0; pjy = 0.0; pjz = 0.0;
  for (j = 1; j<nseg; j++){
    pkx = 0.0; pky = 0.0; pkz = 0.0;
    for (k = 1; k<(j); k++){
      pkx = pkx + px[m*nseg + k];
      pky = pky + py[m*nseg + k];
      pkz = pkz + pz[m*nseg + k];
    }
    pjx = pjx+(px[m*nseg]+px[m*nseg+j]+2*pkx);
    pjy = pjy+(py[m*nseg]+py[m*nseg+j]+2*pky);
    pjz = pjz+(pz[m*nseg]+pz[m*nseg+j]+2*pkz);
  }
  rx[mi] = rcmx[m]-(rp/(float)nseg)*pjx;
  ry[mi] = rcmy[m]-(rp/(float)nseg)*pjy;
  rz[mi] = rcmz[m]-(rp/(float)nseg)*pjz;
}
```

rhs.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <math.h>
#include "rhs.h"
using namespace std;
```

```
__global__ void rhs(float **var, int **intVar){
  int nseg=*intVar[7]; float *px=var[18];
  float *py=var[19]; float *pz=var[20];
  float *uxfl=var[30]; float *uyfl=var[31];
  float *uzfl=var[32]; float *Xx=var[51];
  float *Xy=var[52]; float *Xz=var[53];
  float *Yx=var[54]; float *Yy=var[55];
  float *Yz=var[56]; float *fcx=var[66];
  float *fcy=var[67]; float *fcz=var[68];
  float *tcx=var[69]; float *tcy=var[70];
  float *tcz=var[71]; float *fbx=var[72];
  float *fby=var[73]; float *fbz=var[74];
  float *tbx=var[75]; float *tby=var[76];
  float *tbz=var[77]; float *D1=var[78];
  float *D2=var[79]; float *D3=var[80];
  float *A11=var[81]; float *A12=var[82];
  float *A13=var[83]; float *A23=var[84];
  float *A22=var[85]; float *A33=var[86];
  float *C11=var[87]; float *C12=var[88];
  float *C13=var[89]; float *C23=var[90];
  float *C22=var[91]; float *C33=var[92];
  float C0=*var[93]; float C1=*var[94];
  float Omega_x=*var[111]; float Omega_y=*var[112];
  float Omega_z=*var[113]; float *aix=var[159];
  float *aiy=var[160]; float *aiz=var[161];
  float *bix=var[162]; float *biy=var[163];
  float *biz=var[164]; float dirx, diry, dirz;
  float fcxmi, fcymi, fczmi;
  float fbxmi, fbymi, fbzmi;
  float Xxmi2, Xymi2, Xzmi2, Xxmi2_p, Xymi2_p, Xzmi2_p;
  int mi, m, i, mi2;
  mi=threadIdx.x+blockIdx.x*blockDim.x;
  m=mi / nseg;
  i=mi-m*nseg;
  mi2=m*(nseg+1)+i;
  fcxmi=fcx[mi]; fcymi=fcy[mi]; fczmi=fcz[mi];
  fbxmi=fbx[mi]; fbymi=fby[mi]; fbzmi=fbz[mi];
  Xxmi2=Xx[mi2]; Xymi2=Xy[mi2]; Xzmi2=Xz[mi2];
  Xxmi2_p=Xx[mi2+1]; Xymi2_p=Xy[mi2+1]; Xzmi2_p=Xz[mi2+1];
  aix[mi]=uxfl[mi]+A11[mi]*(fcxmi+fbxmi+Xxmi2_p-Xxmi2)
    + A12[mi]*(fcymi+fbymi+Xymi2_p-Xymi2)
    + A13[mi]*(fczmi+fbzmi+Xzmi2_p-Xzmi2);
  aiy[mi]=uyfl[mi]+A12[mi]*(fcxmi+fbxmi+Xxmi2_p-Xxmi2)
    + A22[mi]*(fcymi+fbymi+Xymi2_p-Xymi2)
    + A23[mi]*(fczmi+fbzmi+Xzmi2_p-Xzmi2);
  aiz[mi]=uzfl[mi]+A13[mi]*(fcxmi+fbxmi+Xxmi2_p-Xxmi2)
    + A23[mi]*(fcymi+fbymi+Xymi2_p-Xymi2)
    + A33[mi]*(fczmi+fbzmi+Xzmi2_p-Xzmi2);
  dirx=D1[mi]+Yx[mi2+1]-Yx[mi2]+C1*tcx[mi]+tbx[mi]
    + C0*(py[mi]*(Xzmi2_p+Xzmi2)-pz[mi]*(Xymi2_p+Xymi2));
  diry=D2[mi]+Yy[mi2+1]-Yy[mi2]+C1*tcy[mi]+tby[mi]
    + C0*(pz[mi]*(Xxmi2_p+Xxmi2)-px[mi]*(Xzmi2_p+Xzmi2));
  dirz=D3[mi]+Yz[mi2+1]-Yz[mi2]+C1*tcz[mi]+tbz[mi]
    + C0*(px[mi]*(Xymi2_p+Xymi2)-py[mi]*(Xxmi2_p+Xxmi2));
  bix[mi]=Omega_x+C11[mi]*dirx+C12[mi]*diry+C13[mi]*dirz;
  biy[mi]=Omega_y+C12[mi]*dirx+C22[mi]*diry+C23[mi]*dirz;
  biz[mi]=Omega_z+C13[mi]*dirx+C23[mi]*diry+C33[mi]*dirz;
}
```

setVarArray.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <cuda.h>
#include "setVarArray.h"
using namespace std;
__global__ void setVarArray(float **var,int **intVar,int *ifiber,
  int *ncnt,int *ncpf,int *num_groups,int *overs,int *total_contacts,
  int *nfib,int *nseg,int *fac,int *step,int *csrRow,int *csrCol,
  int *bin,int *list, int *bnei,int *nxbin,int *nybin,
  int *nzbin,int *clist,int *status,int *lead_clist,int *nc,
  int *bdimx,int *bdimy,int *bdimz,int *maxCon,int *clist_pos,
  int *maxGr,int *maxBin, int *blasGrid,int *blasBlock,int *bnum,
  int *potCon,int *potConSize,int *groupId,float *rcmx,
  float *rcmy,float *rcmz,float *rx,float *ry,float *rz,
  float *q0,float *q1,float *q2,float *q3,float *q0dot,float *q1dot,
  float *q2dot,float *q3dot,float *qe0,float *qe1,float *qe2,float *qe3
      ,
  float *px,float *py,float *pz,float *ucmx,float *ucmy,float *ucmz,
  float *ucox,float *ucoy,float *ucoz,float *ux,float *uy,
  float *uz,float *uxfl,float *uyfl,float *uzfl,float *wx,float *wy,
  float *wz,float *R11,float *R12,float *R13,float *R21,float *R22,
  float *R23,float *R11eq,float *R12eq,float *R13eq,float *R21eq,
  float *R22eq,float *R23eq,float *R31eq,float *R32eq,float *R33eq,
  float *Xx,float *Xy,float *Xz,float *Yx,float *Yy,float *Yz,
  float *Ybx,float *Yby,float *Ybz,float *fx,float *fy,float *fz,
  float *tx,float *ty,float *tz,float *fcx,float *fcy,float *fcz,
  float *tcx,float *tcy,float *tcz,float *fbx,float *fby,float *fbz,
  float *tbx,float *tby,float *tbz,float *D1,float *D2,float *D3,
  float *A11,float *A12,float *A13,float *A23,float *A22,float *A33,
  float *C11,float *C12,float *C13,float *C23,float *C22,float *C33,
  float *C0,float *C1,float *C2,float *C3,float *C4,float *g,
  float *duxdx,float *duxdy,float *duxdz,float *duydx,
  float *duydy,float *duydz,float *duzdx,float *duzdy,float *duzdz,
  float *dx,float *dy,float *dz,float *Omega_x,float *Omega_y,
  float *Omega_z,float *rp,float *kb,float *kt,float *dt,
  float *over_cut,float *sidex,float *sidey,float *sidez,
  float *E11,float *E12,float *E13,float *E22,float *E23,float *E33,
  float *contact_cutoff,float *rep_cutoff,float *neighb_cutoff,
  float *Ya,float *Yc,float *Yh,float *A_fric,float *P_fric,
  float *thetaeq,float *phieq,float *delta_rx,float *mu_stat,
  float *mu_kin,float *csrValTot,float *VTot,float *Vsol,
  float *fstar,float *fact,float *Astar,float *decatt,float *elf,
  float *GijxV,float *GijyV,float *GijzV,float *GjixV,float *GjiyV,
  float *GjizV,float *nxV,float *nyV,float *nzV,float *gV,
```

```
float *aix,float *aiy,float *aiz,float *bix,float *biy,
float *biz,float *Stress){
intVar[0]=ifiber; intVar[1]=ncnt;
intVar[2]=ncpf; intVar[3]=num_groups;
intVar[4]=overs; intVar[5]=total_contacts;
intVar[6]=nfib; intVar[7]=nseg;
intVar[8]=fac; intVar[9]=step;
intVar[10]=csrRow; intVar[11]=csrCol;
intVar[12]=bin; intVar[13]=list;
intVar[14]=bnei; intVar[15]=nxbin;
intVar[16]=nybin; intVar[17]=nzbin;
intVar[18]=clist; intVar[19]=status;
intVar[20]=lead_clist; intVar[21]=nc;
intVar[22]=bdimx; intVar[23]=bdimy;
intVar[24]=bdimz; intVar[25]=maxCon;
intVar[26]=clist_pos; intVar[27]=maxGr;
intVar[28]=maxBin; intVar[29]=blasGrid;
intVar[30]=blasBlock; intVar[31]=bnum;
intVar[32]=potCon; intVar[33]=potConSize;
intVar[34]=groupId; var[0]=rcmx;
var[1]=rcmy; var[2]=rcmz; var[3]=rx;
var[4]=ry; var[5]=rz; var[6]=q0;
var[7]=q1; var[8]=q2; var[9]=q3;
var[10]=q0dot; var[11]=q1dot; var[12]=q2dot;
var[13]=q3dot; var[14]=qe0; var[15]=qe1;
var[16]=qe2; var[17]=qe3; var[18]=px;
var[19]=py; var[20]=pz; var[21]=ucmx;
var[22]=ucmy; var[23]=ucmz; var[24]=ucox;
var[25]=ucoy; var[26]=ucoz; var[27]=ux;
var[28]=uy; var[29]=uz; var[30]=uxfl;
var[31]=uyfl; var[32]=uzfl; var[33]=wx;
var[34]=wy; var[35]=wz; var[36]=R11;
var[37]=R12; var[38]=R13; var[39]=R21;
var[40]=R22; var[41]=R23; var[42]=R11eq;
var[43]=R12eq; var[44]=R13eq; var[45]=R21eq;
var[46]=R22eq; var[47]=R23eq; var[48]=R31eq;
var[49]=R32eq; var[50]=R33eq; var[51]=Xx;
var[52]=Xy; var[53]=Xz; var[54]=Yx;
var[55]=Yy; var[56]=Yz; var[57]=Ybx;
var[58]=Yby; var[59]=Ybz; var[60]=fx;
var[61]=fy; var[62]=fz; var[63]=tx;
var[64]=ty; var[65]=tz; var[66]=fcx;
var[67]=fcy; var[68]=fcz; var[69]=tcx;
var[70]=tcy; var[71]=tcz; var[72]=fbx;
var[73]=fby; var[74]=fbz; var[75]=tbx;
var[76]=tby; var[77]=tbz; var[78]=D1;
var[79]=D2; var[80]=D3; var[81]=A11;
var[82]=A12; var[83]=A13; var[84]=A23;
var[85]=A22; var[86]=A33; var[87]=C11;
var[88]=C12; var[89]=C13; var[90]=C23;
var[91]=C22; var[92]=C33; var[93]=C0;
var[94]=C1; var[95]=C2; var[96]=C3;
var[97]=C4; var[98]=g; var[99]=duxdx;
var[100]=duxdy; var[101]=duxdz; var[102]=duydx;
var[103]=duydy; var[104]=duydz; var[105]=duzdx;
var[106]=duzdy; var[107]=duzdz; var[108]=dx;
var[109]=dy; var[110]=dz; var[111]=Omega_x;
var[112]=Omega_y; var[113]=Omega_z; var[114]=rp;
var[115]=kb; var[116]=kt; var[117]=dt;
var[118]=over_cut; var[119]=sidex; var[120]=sidey;
var[121]=sidez; var[122]=E11; var[123]=E12;
var[124]=E13; var[125]=E22; var[126]=E23;
var[127]=E33; var[128]=contact_cutoff;
var[129]=rep_cutoff; var[130]=neighb_cutoff;
var[131]=Ya; var[132]=Yc; var[133]=Yh;
var[134]=A_fric; var[135]=P_fric; var[136]=thetaeq;
var[137]=phieq; var[138]=delta_rx; var[139]=mu_stat;
var[140]=mu_kin; var[141]=csrValTot; var[142]=VTot;
var[143]=Vsol; var[144]=fstar; var[145]=fact;
var[146]=Astar; var[147]=decatt; var[148]=elf;
var[149]=GijxV; var[150]=GijyV; var[151]=GijzV;
var[152]=GjixV; var[153]=GjiyV; var[154]=GjizV;
var[155]=nxV; var[156]=nyV; var[157]=nzV;
var[158]=gV; var[159]=aix; var[160]=aiy;
var[161]=aiz; var[162]=bix; var[163]=biy;
var[164]=biz; var[165]=Stress;
}
```

---

stress.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <math.h>
#include "stress.h"
using namespace std;
__global__ void stress(float **var,int **intVar){
  int nfib=*intVar[6]; int nseg=*intVar[7];
  float *rx=var[3]; float *ry=var[4];
  float *rz=var[5]; float *px=var[18];
  float *py=var[19]; float *pz=var[20];
  float *ux=var[27]; float *uy=var[28];
  float *uz=var[29]; float *uxfl=var[30];
  float *uyfl=var[31]; float *uzfl=var[32];
  float *wx=var[33]; float *wy=var[34];
  float *wz=var[35]; float rp=*var[114];
  float *Stress=var[165];
  float Stress11,Stress12,Stress13;
  float Stress22,Stress23,Stress33;
  float fh1,fh2,fh3,p1,p2,ITterm;
  float e11,e12,e13,e21,e22,e23;
  float pxmi,pymi,pzmi,rxmi,rymi,rzmi;
  float wxmi,wymi,wzmi,uxmi,uymi,uzmi;
  float uxflmi,uyflmi,uzflmi;
  int mi,m,i,tid;
  __shared__ float StressShared[6];
```

```
tid=threadIdx.x;
mi=threadIdx.x+blockDim.x * blockIdx.x;
m=mi/nseg;
i=mi-m*nseg;
if (tid < 6){
  StressShared[tid]=0.0;
}
__syncthreads();
Stress11=0.0; Stress12=0.0; Stress13=0.0;
Stress22=0.0; Stress23=0.0; Stress33=0.0;
e11=0.0; e12=0.0; e13=1.0;
e21=1.0; e22=0.0; e23=0.0;
rxmi=rx[mi]; rymi=ry[mi]; rzmi=rz[mi];
pxmi=px[mi]; pymi=py[mi]; pzmi=pz[mi];
wxmi=wx[mi]; wymi=wy[mi]; wzmi=wz[mi];
uxmi=ux[mi]; uymi=uy[mi]; uzmi=uz[mi];
uxflmi=uxfl[mi]; uyflmi=uyfl[mi]; uzflmi=uzfl[mi];
// Hydrodynamic force
fh1=(1.0-0.5*pxmi*pxmi)*(uxflmi-uxmi) +
  (-0.5*pxmi*pymi)*(uyflmi-uymi) +
  (-0.5*pxmi*pzmi)*(uzflmi-uzmi);
fh2=(-0.5*pymi*pxmi)*(uxflmi-uxmi) +
  (1.0-0.5*pymi*pymi)*(uyflmi-uymi) +
  (-0.5*pymi*pzmi)*(uzflmi-uzmi);
fh3=(-0.5*pzmi*pxmi)*(uxflmi-uxmi) +
  (-0.5*pzmi*pymi)*(uyflmi-uymi) +
  (1.0-0.5*pzmi*pzmi)*(uzflmi-uzmi);
p1=pxmi*e11+pymi*e12+pzmi*e13;
p2=pxmi*e21+pymi*e22+pzmi*e23;
// Isotropic term
ITterm=p1*p2/3.0+(fh1*rxmi+fh2*rymi +
  fh3*rzmi)/(rp*rp);
// Extra particle stress
Stress11=p1*(e21*pxmi +
  pxmi*e21)-p1*p2*pxmi*pxmi -
  (wymi*pzmi-wzmi*pymi)*pxmi -
  pxmi*(wymi*pzmi-wzmi*pymi) +
  (3.0/(rp*rp))*(fh1*rxmi+rxmi*fh1)-ITterm;
Stress12=p1*(e21*pymi +
  pxmi*e22)-p1*p2*pxmi*pymi -
  (wymi*pzmi-wzmi*pymi)*pymi -
  pxmi*(wzmi*pxmi-wxmi*pzmi) +
  (3.0/(rp*rp))*(fh1*rymi+rxmi*fh2);
Stress13=p1*(e21*pzmi +
  pxmi*e23)-p1*p2*pxmi*pzmi -
  (wymi*pzmi-wzmi*pymi)*pzmi -
  pxmi*(wxmi*pymi-wymi*pxmi) +
  (3.0/(rp*rp))*(fh1*rzmi+rxmi*fh3);
Stress22=p1*(e22*pymi +
  pymi*e22)-p1*p2*pymi*pymi -
  (wzmi*pxmi-wxmi*pzmi)*pymi -
  pymi*(wzmi*pxmi-wxmi*pzmi) +
  (3.0/(rp*rp))*(fh2*rymi+rymi*fh2)-ITterm;
  Stress23=p1*(e22*pzmi +
    pymi*e23)-p1*p2*pymi*pzmi -
    (wzmi*pxmi-wxmi*pzmi)*pzmi -
    pymi*(wxmi*pymi-wymi*pxmi) +
    (3.0/(rp*rp))*(fh2*rzmi+rymi*fh3);
  Stress33=p1*(e23*pzmi +
    pzmi*e23)-p1*p2*pzmi*pzmi -
    (wxmi*pymi-wymi*pxmi)*pzmi -
    pzmi*(wxmi*pymi-wymi*pxmi) +
    (3.0/(rp*rp))*(fh3*rzmi+rzmi*fh3)-ITterm;
  atomicAdd(StressShared+0,Stress11);
  atomicAdd(StressShared+1,Stress12);
  atomicAdd(StressShared+2,Stress13);
  atomicAdd(StressShared+3,Stress22);
  atomicAdd(StressShared+4,Stress23);
  atomicAdd(StressShared+5,Stress33);
  __syncthreads();
  if (tid == 0){
    atomicAdd(Stress+0,StressShared[0]/float(nfib));
    atomicAdd(Stress+1,StressShared[1]/float(nfib));
    atomicAdd(Stress+2,StressShared[2]/float(nfib));
    atomicAdd(Stress+3,StressShared[3]/float(nfib));
    atomicAdd(Stress+4,StressShared[4]/float(nfib));
    atomicAdd(Stress+5,StressShared[5]/float(nfib));
  }
}
```

total_contacts_zero.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include "total_contacts_zero.h"
using namespace std;
__global__ void total_contacts_zero(float **var, int **intVar){
  int *overs = intVar[4];
  int *total_contacts = intVar[5];
  *total_contacts = 0;
  *overs = 0;
}
```

updateBod.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include "updateBod.h"
using namespace std;
__global__ void updateBod(float **var, int **intVar){
  float *px = var[18]; float *py = var[19];
  float *pz = var[20]; float *tbx = var[75];
```

```
  float *tby = var[76]; float elf = *var[148];
  int mi = threadIdx.x + blockDim.x * blockIdx.x;
  // Assign new body forces and torques.
  tbx[mi] = elf*pz[mi]*py[mi];
  tby[mi] = -elf*pz[mi]*px[mi];
}
```

updateCen.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <math.h>
#include "updateCen.h"
using namespace std;
__global__ void updateCen(float **var,int **intVar){
  int nseg=*intVar[7]; int step=*intVar[9];
  float dt=*var[117]; float sidex=*var[119];
  float sidey=*var[120]; float sidez=*var[121];
  float delta_rx=*var[138]; float *rcmx=var[0];
  float *rcmy=var[1]; float *rcmz=var[2];
  float *ucmx=var[21]; float *ucmy=var[22];
  float *ucmz=var[23]; float *ucox=var[24];
  float *ucoy=var[25]; float *ucoz=var[26];
  float *ux=var[27]; float *uy=var[28];
  float *uz=var[29];
  int ii;
  float ucmxm,ucmym,ucmzm,rcmxm,rcmym,rcmzm;
  float corx,cory,corz,nsegF;
  // wbx,wby,wbz - body frame coordinates of the ang. velocity
  int m=threadIdx.x+blockDim.x * blockIdx.x;
  ucmxm=0.0; ucmym=0.0; ucmzm=0.0;
  rcmxm=rcmx[m]; rcmym=rcmy[m]; rcmzm=rcmz[m];
  // Integrate each fiber individually
  for (ii=0; ii < nseg; ii++){
    ucmxm += ux[m*nseg+ii];
    ucmym += uy[m*nseg+ii];
    ucmzm += uz[m*nseg+ii];
  }
  // Integrate the fiber centers of mass
  nsegF=(float)nseg;
  ucmxm /= nsegF; ucmym /= nsegF; ucmzm /= nsegF;
  ucmx[m]=ucmxm; ucmy[m]=ucmym; ucmz[m]=ucmzm;
  if (step == 0){
    rcmxm += ucmxm*dt;
    rcmym += ucmym*dt;
    rcmzm += ucmzm*dt;
  }
  else{
    rcmxm += (1.5*ucmxm - 0.5*ucox[m])*dt;
    rcmym += (1.5*ucmym - 0.5*ucoy[m])*dt;
    rcmzm += (1.5*ucmzm - 0.5*ucoz[m])*dt;
  }
  ucox[m]=ucmxm; ucoy[m]=ucmym; ucoz[m]=ucmzm;
  // Check periodic boundaries for a fiber leaving the box
  cory=roundf(rcmym / sidey);
  corz=roundf(rcmzm / sidez);
  rcmxm -= corz*delta_rx;
  corx=roundf(rcmxm / sidex);
  rcmxm -= corx*sidex;
  rcmym -= cory*sidey;
  rcmzm -= corz*sidez;
  rcmx[m]=rcmxm; rcmy[m]=rcmym; rcmz[m]=rcmzm;
  // Check if the time step was too big
  if (rcmxm*rcmxm > sidex*sidex ||
    rcmym*rcmym > sidey*sidey ||
    rcmzm*rcmzm > sidez*sidez){
    // or output to a file
    printf("time step too big\n");
    printf("rcmx[%5d]=%10f\n",m,rcmxm);
    printf("rcmy[%5d]=%10f\n",m,rcmym);
    printf("rcmz[%5d]=%10f\n",m,rcmzm);
  }
}
```

updateOri.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <math.h>
#include "updateOri.h"
using namespace std;
__global__ void updateOri(float **var,int **intVar){
  int step=*intVar[9]; float dt=*var[117];
  float *q0=var[6]; float *q1=var[7];
  float *q2=var[8]; float *q3=var[9];
  float *q0dot=var[10]; float *q1dot=var[11];
  float *q2dot=var[12]; float *q3dot=var[13];
  float *qe0=var[14]; float *qe1=var[15];
  float *qe2=var[16]; float *qe3=var[17];
  float *px=var[18]; float *py=var[19];
  float *pz=var[20]; float *wx=var[33];
  float *wy=var[34]; float *wz=var[35];
  float *R11=var[36]; float *R12=var[37];
  float *R13=var[38]; float *R21=var[39];
  float *R22=var[40]; float *R23=var[41];
  // New and dummy variables
  float wbx,wby,wbz,dum,q0mi,q1mi,q2mi,q3mi;
  float q0dotmi,q1dotmi,q2dotmi,q3dotmi;
  float pxmi,pymi,pzmi,wxmi,wymi,wzmi;
  // wbx,wby,wbz-body frame coordinates of the ang. velocity
  int mi=threadIdx.x+blockDim.x*blockIdx.x;
  // Find the time derivative of the Euler parameters
```

```
  wxmi=wx[mi]; wymi=wy[mi]; wzmi=wz[mi];
  wbx=R11[mi]*wxmi+R12[mi]*wymi+R13[mi]*wzmi;
  wby=R21[mi]*wxmi+R22[mi]*wymi+R23[mi]*wzmi;
  wbz=px[mi]*wxmi+py[mi]*wymi+pz[mi]*wzmi;
  q0mi=q0[mi]; q1mi=q1[mi]; q2mi=q2[mi]; q3mi=q3[mi];
  q0dotmi=0.5*(-wbx*q1mi-wby*q2mi-wbz*q3mi);
  q1dotmi=0.5*(wbx*q0mi+wbz*q2mi-wby*q3mi);
  q2dotmi=0.5*(wby*q0mi-wbz*q1mi+wbx*q3mi);
  q3dotmi=0.5*(wbz*q0mi+wby*q1mi-wbx*q2mi);
  q0dot[mi]=q0dotmi; q1dot[mi]=q1dotmi; q2dot[mi]=q2dotmi; q3dot[mi]=
       q3dotmi;
  // Integrate the Euler parameters (Adams-Bashforth)
  if (step == 0){
    q0mi += q0dotmi*dt;
    q1mi += q1dotmi*dt;
    q2mi += q2dotmi*dt;
    q3mi += q3dotmi*dt;
  }
  else{
    q0mi += (1.5*q0dotmi-0.5*qe0[mi])*dt;
    q1mi += (1.5*q1dotmi-0.5*qe1[mi])*dt;
    q2mi += (1.5*q2dotmi-0.5*qe2[mi])*dt;
    q3mi += (1.5*q3dotmi-0.5*qe3[mi])*dt;
  }
  qe0[mi]=q0dotmi; qe1[mi]=q1dotmi;
  qe2[mi]=q2dotmi; qe3[mi]=q3dotmi;
  dum=sqrtf(q0mi*q0mi+q1mi*q1mi+q2mi*q2mi+q3mi*q3mi);
  q0mi /= dum; q1mi /= dum;
  q2mi /= dum; q3mi /= dum;
  q0[mi]=q0mi; q1[mi]=q1mi; q2[mi]=q2mi; q3[mi]=q3mi;
  // Calculate the new rotation matrix and p-vectors
  R11[mi]=2.0*(q0mi*q0mi+q1mi*q1mi)-1.0;
  R12[mi]=2.0*(q1mi*q2mi+q0mi*q3mi);
  R13[mi]=2.0*(q1mi*q3mi-q0mi*q2mi);
  R21[mi]=2.0*(q1mi*q2mi-q0mi*q3mi);
  R22[mi]=2.0*(q0mi*q0mi+q2mi*q2mi)-1.0;
  R23[mi]=2.0*(q3mi*q2mi+q0mi*q1mi);
  pxmi=2.0*(q1mi*q3mi+q0mi*q2mi);
  pymi=2.0*(q3mi*q2mi-q0mi*q1mi);
  pzmi=2.0*(q0mi*q0mi+q3mi*q3mi)-1.0;
  dum=sqrtf(pxmi*pxmi+pymi*pymi+pzmi*pzmi);
  pxmi /= dum; pymi /= dum; pzmi /= dum;
  px[mi]=pxmi; py[mi]=pymi; pz[mi]=pzmi;
}
```

x_forces_para.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include "x_forces_para.h"
using namespace std;

__global__ void x_forces_para(float **var,int **intVar){
  int nseg=*intVar[7]; float rp=*var[114];
  float Ya=*var[131]; float C2=*var[95];
  float C3=*var[96]; float *csrValTot=var[141];
  float *VTot=var[142]; float *px=var[18];
  float *py=var[19]; float *pz=var[20];
  float *ux=var[27]; float *uy=var[28];
  float *uz=var[29]; float *wx=var[33];
  float *wy=var[34]; float *wz=var[35];
  int m,i,k1,k,mi,offset,offset_V;
  float pxmi,pymi,pzmi,pxmi_1,pymi_1,pzmi_1;
  float pxmi_p,pymi_p,pzmi_p;
  float wxmi,wymi,wzmi,wxmi_1,wymi_1,wzmi_1;
  float T[9],U[9],S[9];
  const int blockDimA=3;
  const int nnz=(4+3*(nseg-3))*blockDimA*blockDimA;
  // blockDim-block dimension of matrix
  // nnz-number of nonzero values
  // T,U,S-temp stores corresponding block matrix
  mi=threadIdx.x+blockDim.x*blockIdx.x;
  pxmi=px[mi]; pymi=py[mi]; pzmi=pz[mi];
  pxmi_1=px[mi-1]; pymi_1=py[mi-1]; pzmi_1=pz[mi-1];
  pxmi_p=px[mi+1]; pymi_p=py[mi+1]; pzmi_p=pz[mi+1];
  wxmi=wx[mi]; wymi=wy[mi]; wzmi=wz[mi];
  wxmi_1=wx[mi-1]; wymi_1=wy[mi-1]; wzmi_1=wz[mi-1];
  m=mi/nseg;
  i=mi-m*nseg;
  // Start calculate for 2nd joint (X_1=0)
  k1=3*(i+1)-2-1;
  offset_V=m*3*(nseg-1);
  if (i > 0){
    VTot[offset_V+k1-3]=ux[mi-1]-ux[mi] +
      rp*(wymi*pzmi-wzmi*pymi+wymi_1*pzmi_1-wzmi_1*pymi_1);
    VTot[offset_V+k1-2]=uy[mi-1]-uy[mi] +
      rp*(wzmi*pxmi-wxmi*pzmi+wzmi_1*pxmi_1-wxmi_1*pzmi_1);
    VTot[offset_V+k1-1]=uz[mi-1]-uz[mi] +
      rp*(wxmi*pymi-wymi*pxmi+wxmi_1*pymi_1-wymi_1*pxmi_1);
    if (i < (nseg-1)){
      // S matrix grouping
      S[0]=1.0/Ya-C2+(C3+C2)*pxmi*pxmi;
      S[1]=(C3+C2)*pxmi*pymi;
      S[2]=(C3+C2)*pxmi*pzmi;
      S[4]=1.0/Ya-C2+(C3+C2)*pymi*pymi;
      S[5]=(C3+C2)*pymi*pzmi;
      S[8]=1.0/Ya-C2+(C3+C2)*pzmi*pzmi;
      S[3]=S[1]; S[6]=S[2]; S[7]=S[5];
      // U matrix grouping
      U[0]=1.0/Ya-C2+(C3+C2)*pxmi*pxmi;
      U[1]=(C3+C2)*pxmi*pymi;
      U[2]=(C3+C2)*pxmi*pzmi;
      U[4]=1.0/Ya-C2+(C3+C2)*pymi*pymi;
      U[5]=(C3+C2)*pymi*pzmi;
      U[8]=1.0/Ya-C2+(C3+C2)*pzmi*pzmi;
```

```
    U[3]=U[1]; U[6]=U[2]; U[7]=U[5];
  }
}
if (i < (nseg-1)){
  // T matrix grouping
  T[0]=-2.0*(1.0/Ya+C2) +
    (-C3+C2)*(pxmi*pxmi+pxmi_p*pxmi_p);
  T[1]=(-C3+C2)*(pxmi*pymi+pxmi_p*pymi_p);
  T[2]=(-C3+C2)*(pxmi*pzmi+pxmi_p*pzmi_p);
  T[3]=T[1];
  T[4]=-2.0*(1.0/Ya+C2) +
    (-C3+C2)*(pymi*pymi+pymi_p*pymi_p);
  T[5]=(-C3+C2)*(pymi*pzmi+pymi_p*pzmi_p);
  T[6]=T[2];
  T[7]=T[5];
  T[8]=-2.0*(1.0/Ya+C2) +
    (-C3+C2)*(pzmi*pzmi+pzmi_p*pzmi_p);
}
// put matrix into csr format
offset=m *nnz;
// special case nseg == 2 and nseg == 3
if (nseg == 2){
  if (i == 0){
    for (k=0; k < 9; k += 3){
      csrValTot[offset+0+3*k/3]=T[k];
      csrValTot[offset+1+3*k/3]=T[k+1];
      csrValTot[offset+2+3*k/3]=T[k+2];
    }
  }
}
else if (nseg == 3){
  if (i == 0){
    for (k=0; k < 9; k += 3){
      csrValTot[offset+0+6*k/3]=T[k];
      csrValTot[offset+1+6*k/3]=T[k+1];
      csrValTot[offset+2+6*k/3]=T[k+2];
    }
  }
  if (i == 1){
    for (k=0; k < 9; k += 3){
      csrValTot[offset+3+6*k/3]=U[k];
      csrValTot[offset+4+6*k/3]=U[k+1];
      csrValTot[offset+5+6*k/3]=U[k+2];
      csrValTot[offset+18+6*k/3]=S[k];
      csrValTot[offset+19+6*k/3]=S[k+1];
      csrValTot[offset+20+6*k/3]=S[k+2];
      csrValTot[offset+21+6*k/3]=T[k];
      csrValTot[offset+22+6*k/3]=T[k+1];
      csrValTot[offset+23+6*k/3]=T[k+2];
    }
  }

}
```

```
  else if (nseg > 3){
    // first two segments
    if (i == 0){
      for (k=0; k < 9; k += 3){
        csrValTot[offset+0+6*k/3]=T[k];
        csrValTot[offset+1+6*k/3]=T[k+1];
        csrValTot[offset+2+6*k/3]=T[k+2];
      }
    }
    if (i == 1){
      for (k=0; k < 9; k += 3){
        csrValTot[offset+3+6*k/3]=U[k];
        csrValTot[offset+4+6*k/3]=U[k+1];
        csrValTot[offset+5+6*k/3]=U[k+2];
        csrValTot[offset+18+9*k/3]=S[k];
        csrValTot[offset+19+9*k/3]=S[k+1];
        csrValTot[offset+20+9*k/3]=S[k+2];
        csrValTot[offset+21+9*k/3]=T[k];
        csrValTot[offset+22+9*k/3]=T[k+1];
        csrValTot[offset+23+9*k/3]=T[k+2];
      }
    }
    // middle section
    if (i != 0 && i != 1 && i != nseg-2 && i != nseg-1 && nseg){
      for (k=0; k < 9; k += 3){
        csrValTot[offset+9*(2+3*(i-2))+6+0+9*k/3]=U[k];
        csrValTot[offset+9*(2+3*(i-2))+6+1+9*k/3]=U[k+1];
        csrValTot[offset+9*(2+3*(i-2))+6+2+9*k/3]=U[k+2];
        csrValTot[offset+9*(2+3*(i-1))+0+9*k/3]=S[k];
        csrValTot[offset+9*(2+3*(i-1))+1+9*k/3]=S[k+1];
        csrValTot[offset+9*(2+3*(i-1))+2+9*k/3]=S[k+2];
        csrValTot[offset+9*(2+3*(i-1))+3+0+9*k/3]=T[k];
        csrValTot[offset+9*(2+3*(i-1))+3+1+9*k/3]=T[k+1];
        csrValTot[offset+9*(2+3*(i-1))+3+2+9*k/3]=T[k+2];
      }
    }
    // last segment
    if (i == nseg-2){
      for (k=0; k < 9; k += 3){
        csrValTot[offset+9*(2+3*(i-2))+6+0+9*k/3]=U[k];
        csrValTot[offset+9*(2+3*(i-2))+6+1+9*k/3]=U[k+1];
        csrValTot[offset+9*(2+3*(i-2))+6+2+9*k/3]=U[k+2];
        csrValTot[offset+9*(2+3*(i-1))+0+6*k/3]=S[k];
        csrValTot[offset+9*(2+3*(i-1))+1+6*k/3]=S[k+1];
        csrValTot[offset+9*(2+3*(i-1))+2+6*k/3]=S[k+2];
        csrValTot[offset+9*(2+3*(i-1))+3+0+6*k/3]=T[k];
        csrValTot[offset+9*(2+3*(i-1))+3+1+6*k/3]=T[k+1];
        csrValTot[offset+9*(2+3*(i-1))+3+2+6*k/3]=T[k+2];
      }
    }
  }
}
```

x_forces_postsolve.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include "x_forces_postsolve.h"
using namespace std;
__global__ void x_forces_postsolve(float **var, int **intVar){
  int nseg=*intVar[7]; float *px=var[18];
  float *py=var[19]; float *pz=var[20];
  float *ux=var[27]; float *uy=var[28];
  float *uz=var[29]; float *wx=var[33];
  float *wy=var[34]; float *wz=var[35];
  float *Xx=var[51]; float *Xy=var[52];
  float *Xz=var[53]; float *A11=var[81];
  float *A12=var[82]; float *A13=var[83];
  float *A23=var[84]; float *A22=var[85];
  float *A33=var[86]; float *C11=var[87];
  float *C12=var[88]; float *C13=var[89];
  float *C23=var[90]; float *C22=var[91];
  float *C33=var[92]; float C0=*var[93];
  float pxmi, pymi, pzmi, A12mi, A13mi, A23mi;
  float C12mi, C13mi, C23mi, Xxmi2, Xymi2, Xzmi2;
  float Xxmi2_p, Xymi2_p, Xzmi2_p;
    int m, i, mi, mi2;
  mi=threadIdx.x+blockIdx.x*blockDim.x;
  m=mi / nseg;
  i=mi-m*nseg;
  mi2=m*(nseg+1)+i;
  Xxmi2=Xx[mi2]; Xymi2=Xy[mi2]; Xzmi2=Xz[mi2];
  Xxmi2_p=Xx[mi2+1]; Xymi2_p=Xy[mi2+1]; Xzmi2_p=Xz[mi2+1];
  A12mi=A12[mi]; A13mi=A13[mi]; A23mi=A23[mi];
  ux[mi] += A11[mi]*(Xxmi2_p-Xxmi2) +
    A12mi*(Xymi2_p-Xymi2)+A13mi*(Xzmi2_p-Xzmi2);
  uy[mi] += A12mi*(Xxmi2_p-Xxmi2) +
    A22[mi]*(Xymi2_p-Xymi2)+A23mi*(Xzmi2_p-Xzmi2);
  uz[mi] += A13mi*(Xxmi2_p-Xxmi2) +
    A23mi*(Xymi2_p-Xymi2)+A33[mi]*(Xzmi2_p-Xzmi2);
  pxmi=px[mi]; pymi=py[mi]; pzmi=pz[mi];
  C12mi=C12[mi]; C13mi=C13[mi]; C23mi=C23[mi];
  wx[mi] += C0*(
    C11[mi]*(pymi*(Xzmi2_p+Xzmi2)-pzmi*(Xymi2_p+Xymi2))
    + C12mi*(pzmi*(Xxmi2_p+Xxmi2)-pxmi*(Xzmi2_p+Xzmi2))
    + C13mi*(pxmi*(Xymi2_p+Xymi2)-pymi*(Xxmi2_p+Xxmi2)));
  wy[mi] += C0*(
    C12mi*(pymi*(Xzmi2_p+Xzmi2)-pzmi*(Xymi2_p+Xymi2))
    + C22[mi]*(pzmi*(Xxmi2_p+Xxmi2)-pxmi*(Xzmi2_p+Xzmi2))
    + C23mi*(pxmi*(Xymi2_p+Xymi2)-pymi*(Xxmi2_p+Xxmi2)));
  wz[mi] += C0*(
    C13mi*(pymi*(Xzmi2_p+Xzmi2)-pzmi*(Xymi2_p+Xymi2))
    + C23mi*(pzmi*(Xxmi2_p+Xxmi2)-pxmi*(Xzmi2_p+Xzmi2))
    + C33[mi]*(pxmi*(Xymi2_p+Xymi2)-pymi*(Xxmi2_p+Xxmi2)));
}
```

x_forces_presolve.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <math.h>
#include "x_forces_presolve.h"
using namespace std;
__global__ void x_forces_presolve(float **var, int **intVar){
  int nseg = *intVar[7]; float Omega_x = *var[111];
  float Omega_y = *var[112]; float Omega_z = *var[113];
  float C1 = *var[94]; float *ux = var[27];
  float *uy = var[28]; float *uz = var[29];
  float *uxfl = var[30]; float *uyfl = var[31];
  float *uzfl = var[32]; float *wx = var[33];
  float *wy = var[34]; float *wz = var[35];
  float *Yx = var[54]; float *Yy = var[55];
  float *Yz = var[56]; float *fx = var[60];
  float *fy = var[61]; float *fz = var[62];
  float *tx = var[63]; float *ty = var[64];
  float *tz = var[65]; float *fcx = var[66];
  float *fcy = var[67]; float *fcz = var[68];
  float *tcx = var[69]; float *tcy = var[70];
  float *tcz = var[71]; float *fbx = var[72];
  float *fby = var[73]; float *fbz = var[74];
  float *tbx = var[75]; float *tby = var[76];
  float *tbz = var[77]; float *D1 = var[78];
  float *D2 = var[79]; float *D3 = var[80];
  float *A11 = var[81]; float *A12 = var[82];
  float *A13 = var[83]; float *A23 = var[84];
  float *A22 = var[85]; float *A33 = var[86];
  float *C11 = var[87]; float *C12 = var[88];
  float *C13 = var[89]; float *C23 = var[90];
  float *C22 = var[91]; float *C33 = var[92];
  int m, i, mi, mi2;
  float fxSum, fySum, fzSum, xSum, ySum, zSum;
  mi = threadIdx.x+blockDim.x*blockIdx.x;
  m = mi / nseg;
  i = mi-m*nseg;
  mi2 = m*(nseg+1)+i;
  fxSum = fcx[mi]+fbx[mi]+fx[mi];
  fySum = fcy[mi]+fby[mi]+fy[mi];
  fzSum = fcz[mi]+fbz[mi]+fz[mi];
  xSum = D1[mi]+Yx[mi2+1]-Yx[mi2]+tbx[mi]+C1*(tcx[mi]+tx[mi]);
  ySum = D2[mi]+Yy[mi2+1]-Yy[mi2]+tby[mi]+C1*(tcy[mi]+ty[mi]);
  zSum = D3[mi]+Yz[mi2+1]-Yz[mi2]+tbz[mi]+C1*(tcz[mi]+tz[mi]);
  ux[mi] = uxfl[mi]+A11[mi]*fxSum+A12[mi]*fySum+A13[mi]*fzSum;
  uy[mi] = uyfl[mi]+A12[mi]*fxSum+A22[mi]*fySum+A23[mi]*fzSum;
  uz[mi] = uzfl[mi]+A13[mi]*fxSum+A23[mi]*fySum+A33[mi]*fzSum;
  wx[mi] = Omega_x+C11[mi]*xSum+C12[mi]*ySum+C13[mi]*zSum;
  wy[mi] = Omega_y+C12[mi]*xSum+C22[mi]*ySum+C23[mi]*zSum;
  wz[mi] = Omega_z+C13[mi]*xSum+C23[mi]*ySum+C33[mi]*zSum;
```

```
}
```

x_forces_update.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda.h>
#include "x_forces_update.h"
using namespace std;
__global__ void x_forces_update(float **var, int **intVar){
  int nseg = *intVar[7]; float *Xx = var[51];
  float *Xy = var[52]; float *Xz = var[53];
  float *Vsol = var[143];
  int m, i, k1, mi2, offset_V;
  int tid = threadIdx.x+blockDim.x*blockIdx.x;
  m = tid / (nseg-1);
  i = tid-m * (nseg-1);
  mi2 = m*(nseg+1)+i;
  k1 = 3 * (i+1)-3;
  offset_V = m * 3 * (nseg-1);
  Xx[mi2+1] = Vsol[offset_V+k1];
  Xy[mi2+1] = Vsol[offset_V+k1+1];
  Xz[mi2+1] = Vsol[offset_V+k1+2];
}
```

bnei_set.h

```
#ifndef __flexfric__bnei_set__
#define __flexfric__bnei_set__
#include <stdio.h>
#include <cuda.h>
__global__ void bnei_set(float **var, int **intVar);
#endif /* defined(__flexfric__bnei_set__) */
```

boxSize.h

```
#ifndef __flexfric__boxSize__
#define __flexfric__boxSize__
#include <stdio.h>
#include <cuda.h>
__global__ void boxSize(float **var, int **intVar,
       float *Lx, float *Ly, float *Lz,
       float *dLx, float *dLy, float *dLz, int *box_write,
       float *dx_fixed, float *dy_fixed, float *dz_fixed);
#endif /* defined(__flexfric__boxSize__) */
```

cell.h

```
#ifndef __flexfric__cell__
#define __flexfric__cell__
#include <stdio.h>
#include <cuda.h>
__global__ void cell(float **var, int **intVar);
#endif /* defined(__flexfric__cell__) */
```

contact.h

```
#ifndef __flexfric__contact__
#define __flexfric__contact__
#include <stdio.h>
#include <cuda.h>
#include <cublas_v2.h>
__global__ void contact(float **var, int **intVar);
#endif /* defined(__flexfric__contact__) */
```

csr_set.h

```
#ifndef __flexfric__csr_set__
#define __flexfric__csr_set__
#include <stdio.h>
#include <cuda.h>
__global__ void csr_set(int **intVar);
#endif /* defined(__flexfric__csr_set__) */
```

cublas_initialize.h

```
#ifndef __flexfric__cublas_initialize__
#define __flexfric__cublas_initialize__
#include <stdio.h>
#include <cuda.h>
#include <cublas_v2.h>
__global__ void cublas_initialize(cublasHandle_t *handle_cublas);
__device__ const char* cublasGetErrorString(cublasStatus_t status);
#endif /* defined(__flexfric__cublas_initialize__) */
```

cublas_free.h

```
#ifndef __flexfric__cublas_free__
#define __flexfric__cublas_free__
#include <stdio.h>
#include <cuda.h>
#include <cublas_v2.h>
__global__ void cublas_free(cublasHandle_t *handle_cublas);
#endif /* defined(__flexfric__cublas_free__) */
```

delta_twist_torque.h

```
#ifndef __flexfric__delta_twist_torque__
#define __flexfric__delta_twist_torque__
#include <stdio.h>
#include <cuda.h>
__global__ void delta_twist_torque(float **var, int **intVar);
#endif /* defined(__flexfric__delta_twist_torque__) */
```

delta_twist_torque.h

```
#ifndef __flexfric__delta_twist_torque__
#define __flexfric__delta_twist_torque__
#include <stdio.h>
#include <cuda.h>
__global__ void delta_twist_torque(float **var, int **intVar);
#endif /* defined(__flexfric__delta_twist_torque__) */
```

delta_twist_zero.h

```
#ifndef __flexfric__delta_twist_zero__
#define __flexfric__delta_twist_zero__
#include <stdio.h>
#include <cuda.h>
__global__ void delta_twist_zero(float **var, int **intVar);
#endif /* defined(__flexfric__delta_twist_zero__) */
```

friction_3x3.h

```
#ifndef __flexfric__friction_3x3__
#define __flexfric__friction_3x3__
#include <stdio.h>
#include <cuda.h>
__global__ void friction_3x3(float **var, int **intVar);
#endif /* defined(__flexfric__friction_3x3__) */
```

friction_update.h

```
#ifndef __flexfric__friction_update__
#define __flexfric__friction_update__
#include <stdio.h>
#include <cuda.h>
__global__ void friction_update(int tid, int blasId, float **var, int
    **intVar);
#endif /* defined(__flexfric__friction_update__) */
```

group.h

```
#ifndef __flexfric__group__
#define __flexfric__group__
#include <stdio.h>
#include <cuda.h>
#include <cublas_v2.h>
__global__ void group(float **var, int **intVar);
#endif /* defined(__flexfric__group__) */
```

initialize.h

```
#ifndef __flexfric__initialize__
#define __flexfric__initialize__
#include <stdio.h>
#include <cuda.h>
__global__ void initialize(float **var, int **intVar);
#endif /* defined(__flexfric__initialize__) */
```

lead.h

```
#ifndef __flexfric__lead__
#define __flexfric__lead__
#include <stdio.h>
#include <cuda.h>
__global__ void lead(float **var, int **intVar);
#endif /* defined(__flexfric__lead__) */
```

lead_list.h

```
#ifndef __flexfric__lead_list__
#define __flexfric__lead_list__
#include <stdio.h>
#include <cuda.h>
__global__ void lead_list(float **var, int **intVar, float **AA,
       float **PP, int *Pivot, int *info,
       cublasHandle_t *handle_cublas, cublasStatus_t *status_cublas);
#endif /* defined(__flexfric__lead_list__) */
```

lees_edwards.h

```
#ifndef __flexfric__lees_edwards__
#define __flexfric__lees_edwards__
#include <stdio.h>
#include <cuda.h>
__global__ void lees_edwards(float **var, int **intVar);
#endif /* defined(__flexfric__lees_edwards__) */
```

link.h

```
#ifndef __flexfric__link__
#define __flexfric__link__
#include <stdio.h>
#include <cuda.h>
__global__ void link(float **var, int **intVar);
#endif /* defined(__flexfric__link__) */
```

regrowFib.h

```
#ifndef __flexfric__regrowFib__
#define __flexfric__regrowFib__
#include <stdio.h>
#include <cuda.h>
__global__ void regrowFib(float **var, int **intVar);
#endif /* defined(__flexfric__regrowFib__) */
```

regrowSeg.h

```
#ifndef __flexfric__regrowSeg__
#define __flexfric__regrowSeg__
#include <stdio.h>
#include <cuda.h>
__global__ void regrowSeg(float **var, int **intVar);
#endif /* defined(__flexfric__regrowSeg__) */
```

rhs.h

```
#ifndef __flexfric__rhs__
#define __flexfric__rhs__
#include <stdio.h>
#include <cuda.h>
#include <cublas_v2.h>
__global__ void rhs(float **var, int **intVar);
#endif /* defined(__flexfric__rhs__) */
```

setVarArray.h

```
#ifndef __flexfric__setVarArray__
#define __flexfric__setVarArray__
#include <stdio.h>
#include <cuda.h>
__global__ void setVarArray(float **var,int **intVar,int *ifiber,
       int *ncnt,int *ncpf,int *num_groups,int *overs,int *
            total_contacts,
       int *nfib,int *nseg,int *fac,int *step,int *csrRow,int *csrCol,
       int *bin,int *list, int *bnei,int *nxbin,int *nybin,
       int *nzbin,int *clist,int *status,int *lead_clist,int *nc,
       int *bdimx,int *bdimy,int *bdimz,int *maxCon,int *clist_pos,
       int *maxGr,int *maxBin, int *blasGrid,int *blasBlock,int *bnum,
       int *potCon,int *potConSize,int *groupId,float *rcmx,
       float *rcmy,float *rcmz,float *rx,float *ry,float *rz,
       float *q0,float *q1,float *q2,float *q3,float *q0dot,float *
            q1dot,
       float *q2dot,float *q3dot,float *qe0,float *qe1,float *qe2,
            float *qe3,
       float *px,float *py,float *pz,float *ucmx,float *ucmy,float *
            ucmz,
       float *ucox,float *ucoy,float *ucoz,float *ux,float *uy,
       float *uz,float *uxfl,float *uyfl,float *uzfl,float *wx,float *
            wy,
       float *wz,float *R11,float *R12,float *R13,float *R21,float *
            R22,
       float *R23,float *R11eq,float *R12eq,float *R13eq,float *R21eq,
       float *R22eq,float *R23eq,float *R31eq,float *R32eq,float *
            R33eq,
       float *Xx,float *Xy,float *Xz,float *Yx,float *Yy,float *Yz,
       float *Ybx,float *Yby,float *Ybz,float *fx,float *fy,float *fz,
       float *tx,float *ty,float *tz,float *fcx,float *fcy,float *fcz,
       float *tcx,float *tcy,float *tcz,float *fbx,float *fby,float *
            fbz,
       float *tbx,float *tby,float *tbz,float *D1,float *D2,float *D3,
       float *A11,float *A12,float *A13,float *A23,float *A22,float *
            A33,
       float *C11,float *C12,float *C13,float *C23,float *C22,float *
            C33,
       float *C0,float *C1,float *C2,float *C3,float *C4,float *g,
       float *duxdx,float *duxdy,float *duxdz,float *duydx,
       float *duydy,float *duydz,float *duzdx,float *duzdy,float *
            duzdz,
       float *dx,float *dy,float *dz,float *Omega_x,float *Omega_y,
       float *Omega_z,float *rp,float *kb,float *kt,float *dt,
       float *over_cut,float *sidex,float *sidey,float *sidez,
       float *E11,float *E12,float *E13,float *E22,float *E23,float *
            E33,
       float *contact_cutoff,float *rep_cutoff,float *neighb_cutoff,
       float *Ya,float *Yc,float *Yh,float *A_fric,float *P_fric,
       float *thetaeq,float *phieq,float *delta_rx,float *mu_stat,
       float *mu_kin,float *csrValTot,float *VTot,float *Vsol,
       float *fstar,float *fact,float *Astar,float *decatt,float *elf,
       float *GijxV,float *GijyV,float *GijzV,float *GjixV,float *
            GjiyV,
       float *GjizV,float *nxV,float *nyV,float *nzV,float *gV,
       float *aix,float *aiy,float *aiz,float *bix,float *biy,
       float *biz,float *Stress);
#endif /* defined(__flexfric__setVarArray__) */
```

stress.h

```
#ifndef __flexfric__stress__
#define __flexfric__stress__
#include <stdio.h>
```

```
#include <cuda.h>
#include <cublas_v2.h>
__global__ void stress(float **var, int **intVar);
#endif /* defined(__flexfric__stress__) */
```

total_contacts_zero.h

```
#ifndef __flexfric__total_contacts_zero__
#define __flexfric__total_contacts_zero__
#include <stdio.h>
#include <cuda.h>
__global__ void total_contacts_zero(float **var, int **intVar);
#endif /* defined(__flexfric__total_contacts_zero__) */
```

updateBod.h

```
#ifndef __flexfric__updateBod__
#define __flexfric__updateBod__
#include <stdio.h>
#include <cuda.h>
__global__ void updateBod(float **var, int **intVar);
#endif /* defined(__flexfric__updateBod__) */
```

updateCen.h

```
#ifndef __flexfric__updateCen__
#define __flexfric__updateCen__
#include <stdio.h>
#include <cuda.h>
__global__ void updateCen(float **var, int **intVar);
#endif /* defined(__flexfric__updateCen__) */
```

updateOri.h

```
#ifndef __flexfric__updateOri__
#define __flexfric__updateOri__
#include <stdio.h>
#include <cuda.h>
__global__ void updateOri(float **var, int **intVar);
#endif /* defined(__flexfric__updateOri__) */
```

x_forces_para.h

```
#ifndef __flexfric__x_forces_para__
#define __flexfric__x_forces_para__
#include <stdio.h>
#include <cuda.h>
__global__ void x_forces_para(float **var, int **intVar);
#endif /* defined(__flexfric__x_forces_para__) */
```

x_forces_postsolve.h

```
#ifndef __flexfric__x_forces_postsolve__
#define __flexfric__x_forces_postsolve__
#include <stdio.h>
#include <cuda.h>
__global__ void x_forces_postsolve(float **var, int **intVar);
#endif /* defined(__flexfric__x_forces_postsolve__) */
```

x_forces_presolve.h

```
#ifndef __flexfric__x_forces_presolve__
#define __flexfric__x_forces_presolve__
#include <stdio.h>
#include <cuda.h>
__global__ void x_forces_presolve(float **var, int **intVar);
#endif /* defined(__flexfric__x_forces_presolve__) */
```

x_forces_update.h

```
#ifndef __flexfric__x_forces_update__
#define __flexfric__x_forces_update__
#include <stdio.h>
#include <cuda.h>
__global__ void x_forces_update(float **var, int **intVar);
#endif /* defined(__flexfric__x_forces_update__) */
```

Makefile

```
SRCS := $(wildcard *.cu)
OBJS := $(patsubst %.cu,%.o,$(SRCS))

COMPILER = nvcc
NVCCFLAGS = -g -G -arch=sm_60 -rdc=true
LFLAGS = -lcudadevrt -lcublas -lcublas_device -lcusparse -lcusolver -
    lm
EXE = flexfric

$(EXE): $(OBJS)
	$(COMPILER) $(NVCCFLAGS) $(LFLAGS) -o $@ $^

%.o: %.cu
	$(COMPILER) $(NVCCFLAGS) $(LFLAGS) -dc $<

clean:
	-rm -f $(OBJS) $(EXE)
```

Parameters.in

```
32  :nfib, Number of fibers
5  :nseg, number of segments in each fiber
15.0  :rp, aspect ratio of a segment
10.0   :kb, bending constant
10  :mu_stat, static coefficient of friction
0.00  :mu_kin, kinetic coefficient of friction
0.33   :contact_cutoff, cutoff for contacts.
0.66  :rep_cutoff, repulsive force cutoff
1.85  :overlap
0.0001  :dt, time step
0.1  :strain
600.0 600.0 600.0 :side, length of a side in the simulation box
0.7  :fraction_rp, effective aspect ratio factor
5000  :config_write, how often the configuration is written
5000  :contact_write, how often the configuration is written
150.0  :fstar, prefactor for the force
20.0  :fact, exponential factor in force
50  :Astar
35.0  :decatt
0.0  :delta_rx
0.0  :duxdx
0.0  :duydx
0.0  :duzdx
0.0  :duxdy
0.0  :duydy
0.0  :duzdy
1.0  :duxdz
0.0  :duydz
0.0  :duzdz
1  :fac, flow field velocity factor
0.0  :elf, electric field factor
18.0  :dx
18.0  :dy
18.0  :dz
5  :bdimx
5  :bdimy
5  :bdimz
32  :fiberPerBlock, number of fiber in each block
20  :maxCon, maximum number of fibers contacting any fiber
640  :maxGr, maximum number of fibers in a group
400  :maxBin, maximum number of fibers in a bin
40  :nfibGrid, grid size with total number of thread nfib
32   :nfibBlock, block size with total number of thread nfib
8  :blasGrid, grid size for kernel involving cublas
32  :blasBlock, block size for kernel involving cublas
2000  :stress_write
```

box.gen

```
600 :sidex of initial box
600 :sidey of initial box
600 :sidez of initial box
4  :a, max_cont / init_cont
1000 :box_write
0.0001 :dt
0.0005 :change in volume every gamma = dt*box_write
100 :gamma_init, shearing before concentration
300 :gamma_high, shearing at high conc
300 :gamma_low, shearing at low conc
```

boxGen.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <math.h>
using namespace std;
int main(){
	// read input parameters
	FILE *input;
	input = fopen("box.gen", "r");
	float sidex, sidey, sidez, a, dt, rate;
	float gamma_init, gamma_high, gamma_low;
	int box_write;
	fscanf(input, "%f", &sidex);
	fscanf(input, "%*[^\n]%f", &sidey);
	fscanf(input, "%*[^\n]%f", &sidez);
	fscanf(input, "%*[^\n]%f", &a);
	fscanf(input, "%*[^\n]%d", &box_write);
	fscanf(input, "%*[^\n]%f", &dt);
	fscanf(input, "%*[^\n]%f", &rate);
	fscanf(input, "%*[^\n]%f", &gamma_init);
	fscanf(input, "%*[^\n]%f", &gamma_high);
	fscanf(input, "%*[^\n]%f", &gamma_low);
	fclose(input);
	float N, region1, region2, region3, region4, region5, gammaTot;
	// N = number of concentrating/redispersing steps
	// region1 = shearing before concetrating
	// region2 = concentrating
	// region3 = shearing at high concentration
	// region4 = redispersing
	// region5 = shearing at low concentration
	// gammaTot = total strain required to finish cycle
	N = ceil(-logf(a) / logf(1.0 - rate)*dt*float(box_write));
	region1 = gamma_init;
	region2 = region1 + N;
	region3 = region2 + gamma_high;
	region4 = region3 + N;
	region5 = region4 + gamma_low;
	gammaTot = region5;
	int ind;
```

```
int nStep = gammaTot / (dt*box_write) + 1;
float power, n;
float *gamma = (float*)malloc(nStep*sizeof(float));
float *Lx = (float*)malloc(nStep*sizeof(float));
float *Ly = (float*)malloc(nStep*sizeof(float));
float *Lz = (float*)malloc(nStep*sizeof(float));
n = 0.0;
for (int step = 0; step < nStep; step++){
       gamma[step] = dt*float(step*box_write);
       if (step <= region1 / (dt*box_write)){
              Lx[step] = sidex;
              Ly[step] = sidey;
              Lz[step] = sidez;
              ind = step;
              n = 1.0;
              continue;
       }
       if (step < region2 / (dt*box_write)){
              power = 1.0 / 3.0*n;
              Lx[step] = sidex*powf(1.0 - rate, power);
              Ly[step] = sidey*powf(1.0 - rate, power);
              Lz[step] = sidez*powf(1.0 - rate, power);
              n += 1.0;
              continue;
       }
       if (step < region3 / (dt*box_write)){
              Lx[step] = Lx[step - 1];
              Ly[step] = Ly[step - 1];
              Lz[step] = Lz[step - 1];
              ind = step;
              n = 1.0;
              continue;
       }
       if (step < region4 / (dt*box_write)){
              power = -1.0 / 3.0 *n;
              Lx[step] = Lx[ind] * powf(1.0 - rate, power);
              Ly[step] = Ly[ind] * powf(1.0 - rate, power);
              Lz[step] = Lz[ind] * powf(1.0 - rate, power);
              n += 1.0;
              continue;
       }
       if (step < region5 / (dt*box_write)){
              Lx[step] = sidex;
              Ly[step] = sidey;
              Lz[step] = sidez;
              continue;
       }
}
FILE *Box;
Box = fopen("Lbox.txt", "w");
// print to file
fprintf(Box, "%8.4f %10.6f %10.6f %10.6f %10.6f %10.6f %10.6f\n
     ",
              gamma[0], Lx[0], Ly[0], Lz[0], 1.0, 1.0, 1.0);
       for (int step = 1; step < nStep; step++){
              if (step < region2 / (dt*box_write)){
                     fprintf(Box, "%8.4f %10.6f %10.6f %10.6f %10.6f
                          %10.6f %10.6f\n",
                            gamma[step], Lx[step], Ly[step], Lz[step],
                            Lx[step]/Lx[step-1], Ly[step]/Ly[step-1],
                                 Lz[step]/Lz[step-1]);
              }
              else{
                     fprintf(Box, "%8.4f %10.6f %10.6f %10.6f %10.6f
                          %10.6f %10.6f\n",
                            gamma[step], Lx[step], Ly[step], Lz[step],
                            1.0, 1.0, 1.0);
              }
       }
       fclose(Box);
       free(gamma); free(Lx); free(Ly); free(Lz);
       return 0;
}
```

Centers_of_Mass.in

```
 1 5.040038600564003E+01 2.448210655711591E+02 -1.538462913595140E+02
 2 -9.235137719660997E+01 2.598946575075388E+02 -1.335962402634323E+02
 3 -2.308242768980563E+02 -1.164982309564948E+01 -9.091114467009902E+01
 4 -1.520135007798672E+02 -2.455248607322574E+02 2.563884702511132E+02
 5 8.089570328593254E+00 -2.077001340687275E+02 -1.985792074352503E+02
 6 2.953810882754624E+02 8.079679766669869E+01 1.403132957406342E+02
 7 -2.895301714539528E+02 -1.417347083799541E+02 1.600830730050802E+02
 8 2.699496097862720E+02 2.133250515908003E+01 -1.615726640447974E+01
 9 -1.621227278374135E+02 -1.784843606874347E+01 -2.238238519057631E+01
10 1.930033776909113E+02 -2.331433279439807E+02 -1.966903799213469E+02
11 1.384096367284656E+02 -1.352471988648176E+02 -7.964221201837063E+00
12 1.924957903102040E+01 5.076323077082634E+01 -1.088077280670404E+02
13 -2.996747271157801E+02 8.939085127785802E+01 -2.209482788108289E+02
14 -1.569085570983589E+02 2.077680137939751E+02 2.149081371724606E+02
15 -2.732164557091892E+02 -9.075224474072456E+01 1.487241561524570E+02
16 2.217436429113150E+02 1.512949450872838E+02 -2.321943970397115E+02
17 2.158168293535709E+01 2.861989857628942E+01 1.708509695716202E+02
18 -2.453531309030950E+02 9.119287133216858E-01 -6.023624287918210E+01
19 -2.313890224322677E+02 8.079658281058073E+01 -9.098763288930058E+01
20 -2.726489413529634E+02 -1.213239890523255E+02 6.641149828210473E+01
21 -1.077554203569889E+02 1.553806476294994E-01 -1.804321959614754E+02
22 -2.195296011865139E+02 -1.092294716276228E+02 -1.450295384041965E
     +02
23 1.706808107905090E+02 -1.905368895269930E+02 1.084501957520843E+02
24 -2.259394311346114E+02 1.730886256322265E+02 -1.839899500831962E+02
25 -2.289024710655212E+02 2.994726000353694E+02 -2.165159013122320E+02
26 1.105605524964631E+02 3.616466252133250E+01 2.432707180269063E+02
27 -1.253523663617671E+02 6.608784869313240E+01 -2.215582424774766E+02
28 -3.573524951934814E+00 8.090516394004226E+01 -8.660799544304609E+00
```

```
29 1.295789779163897E+02 1.598404842428863E+02 -1.768618874251842E+02
30 1.384711548686028E+02 -1.280533921904862E+02 -1.217936469241977E+02
31 6.418943786993623E+01 1.637953670695424E+02 -8.005794379860163E+01
32 -1.176043826155365E+02 7.786423461511731E+01 -8.083046358078718E+01
```

Euler_Parameters.in

```
 1 1 7.903623423185169E-01 5.998583267857329E-01 -9.916230100591358E
     -02 7.526083769013715E-02
 1 2 7.977876322511802E-01 3.946973109514766E-01 4.085312430055185E
     -01 -2.021166743321670E-01
 1 3 8.075155416855277E-01 1.454853841791198E-01 5.625655393998048E
     -01 -1.013541652148009E-01
 1 4 7.653390810285925E-01 -8.476552427566746E-02 6.341434833042877E
     -01 7.023488824854318E-02
 1 5 6.408242941436458E-01 -2.670912850632121E-01 6.643331975159074E
     -01 2.768896389482111E-01
 2 1 4.283152485238017E-01 -1.082488220195714E-01 -8.697744595525971E
     -01 -2.198195394485142E-01
 2 2 6.946169725699465E-01 -1.418714161060245E-01 -6.909863381789316E
     -01 -1.411298804644308E-01
 2 3 8.778008426795028E-01 -7.613033756315854E-02 -4.711687658888756E
     -01 -4.086375343049613E-02
 2 4 9.730168358979627E-01 5.267119317048816E-02 -2.243134906915581E
     -01 1.214250233199515E-02
 2 5 9.825481119232519E-01 1.777496011609129E-01 5.393589533320558E
     -02 -9.757368384707551E-03
 3 1 3.672841968267754E-01 -1.075522512278937E-01 8.866366713795942E
     -01 2.596348300633712E-01
 3 2 5.018715533721832E-02 -1.178777130082049E-02 9.722132783326084E
     -01 2.283498178687789E-01
 3 3 2.312199798680812E-01 -2.281702344405922E-02 -9.679324822739037E
     -01 -9.5516564584559536E-02
 3 4 4.694419103319791E-01 2.887497272894604E-02 -8.808264172404856E
     -01 5.417888394065204E-02
 3 5 6.776400480952114E-01 1.405667183379682E-01 -7.067882044899202E
     -01 1.466130857295106E-01
 4 1 9.675617688721295E-01 -2.505802008454694E-01 -3.112566929970798E
     -02 -8.060959739719984E-03
 4 2 9.351668043348383E-01 -1.955121175358319E-01 2.891101375837372E
     -01 6.044326524215830E-02
 4 3 8.280208352515146E-01 -7.339257422608839E-02 5.537024215846426E
     -01 4.907804773169980E-02
 4 4 6.560850242656890E-01 3.628889408127586E-02 7.526635607276004E
     -01 -4.163077531704031E-02
 4 5 4.157160221014575E-01 7.694870147597477E-02 8.910967709591385E
     -01 -1.649412959070538E-01
 5 1 5.171643677881870E-01 -9.970082018224778E-02 -8.346899254142183E
     -01 -1.609145473760493E-01
 5 2 7.564861910671191E-01 -1.189201650850853E-01 -6.353050752961739E
     -01 -9.987040784847628E-02
 5 3 9.174442208898131E-01 -6.207589721515198E-02 -3.920954608831238E
     -01 -2.652987176125193E-02
 5 4 9.914281525856019E-01 4.194423506737898E-02 -1.236266306770823E
     -01 5.230257426303883E-03
 5 5 9.768709652794615E-01 1.365078187544871E-01 1.630027488380318E
     -01 -2.277798244162131E-02
 6 1 7.352016460259595E-01 2.304964483718748E-01 6.082628618425072E
     -01 -1.906992865005907E-01
 6 2 4.922785561989979E-01 1.232495576224756E-01 8.358686076219178E
     -01 -2.092726461931671E-01
 6 3 2.448010690007033E-01 2.559847533974959E-02 9.639793262334463E
     -01 -1.008018515251844E-01
 6 4 4.188086159293775E-04 -2.727762469875677E-05 -9.978855818977923E
     -01 -6.499376412390412E-02
 6 5 2.615644657042799E-01 -5.774077022241030E-02 -9.408066526230128E
     -01 -2.076845591642287E-01
 7 1 7.122797045604182E-01 3.935228072479787E-03 7.018739912531577E
     -01 -3.877738220026137E-03
 7 2 4.730386146284184E-01 2.156960994551463E-03 8.810298960511711E
     -01 -4.017319225215614E-03
 7 3 1.915543635383786E-01 3.834753313093649E-04 9.814799632017195E
     -01 -1.964838321142577E-03
 7 4 1.070314544948806E-01 -1.343477892196210E-04 -9.942548426733550E
     -01 -1.248006398347659E-03
 7 5 3.960627170235085E-01 -1.613528660246427E-03 -9.182144235515146E
     -01 -3.740734042795756E-03
 8 1 1.028905632075637E-01 -2.259864162450478E-02 9.712841507729653E
     -01 2.133305694381324E-01
 8 2 2.122666705555816E-01 -3.787657241208288E-02 -9.612934213255374E
     -01 -1.715318744426323E-01
 8 3 4.869240856372664E-01 -3.718519293380033E-02 -8.701188132671881E
     -01 -6.644883032295303E-02
 8 4 7.054786594915827E-01 3.366119945501048E-02 -7.071268681861680E
     -01 3.373984206292707E-02
 8 5 8.665670529353403E-01 1.372483574813204E-01 -4.739099375562969E
     -01 7.505865853467908E-02
 9 1 1.034236878417560E-01 2.180399627357269E-02 -9.730102632436383E
     -01 2.051320407987502E-01
 9 2 4.081241040395658E-01 6.999087452626616E-02 -8.971425416010866E
     -01 1.538546497005964E-01
 9 3 6.579277407568627E-01 4.838247876902524E-02 -7.495014240556138E
     -01 5.511659486345600E-02
 9 4 8.386563429378979E-01 -3.854275818308427E-02 -5.427226504679423E
     -01 -2.494231165555907E-02
 9 5 9.481335334398061E-01 -1.443966176007922E-01 -2.799546422210628E
     -01 -4.263587563631702E-02
10 1 9.971029918364369E-01 7.605951879392479E-02 -7.549538102966491E
     -04 5.758825717396146E-05
10 2 9.528758439075875E-01 5.987869648267998E-02 2.968068320004967E
     -01 -1.865133461087838E-02
10 3 8.243459841589432E-01 2.267092707831284E-02 5.654183634973783E
     -01 -1.554997383855330E-02
10 4 6.231124777173344E-01 -1.073993420898968E-02 7.819423574929434E
     -01 1.347751774360412E-02
10 5 3.651893960578922E-01 -2.048689166265097E-02 9.292467057457774E
```

-01 5.213014614879759E-02
11 1 8.431789720368571E-01 3.531007543820395E-01 -3.739577827209373E
-01 1.566034964876067E-01
11 2 9.505347941708523E-01 3.090636553905422E-01 -2.951545621275506E
-02 9.596865726089265E-03
11 3 9.642778053273972E-01 1.266398677695658E-01 2.306791006476105E
-01 -3.029538857144713E-02
11 4 8.978719647308642E-01 -7.316526024257032E-02 4.327002871067072E
-01 3.525962537730727E-02
11 5 7.390025470873959E-01 -2.098042968493272E-01 6.158626068324983E
-01 1.748446222432373E-01
12 1 5.458424270045507E-01 6.320606178393247E-02 8.299547416356876E
-01 -9.610497110964893E-02
12 2 2.707803711528560E-01 2.576734161839628E-02 9.579686381317267E
-01 -9.115987637270696E-02
12 3 2.270918660829369E-02 9.418731982773213E-04 -9.988828939826030E
-01 4.142909397363492E-02
12 4 3.100114786046148E-01 -8.053598332983414E-03 -9.503780624917075E
-01 -2.468928961675320E-02
12 5 5.720895403838419E-01 -4.854973116644360E-02 -8.158205549403275E
-01 -6.923368778222538E-02
13 1 9.511607847997465E-02 -1.170596432328475E-02 9.879436321668791E
-01 1.215865192970151E-01
13 2 2.084888019905605E-01 -2.107428192972869E-02 -9.728403597616028E
-01 -9.833579462537735E-02
13 3 4.869006975963371E-01 -2.143103482287069E-02 -8.723497683191984E
-01 -3.839665532389774E-02
13 4 7.175798616302195E-01 1.978049133205886E-02 -6.959309297534254E
-01 1.918372638332633E-02
13 5 8.860076471683271E-01 7.984996155725631E-02 -4.548996296025256E
-01 4.099707045674164E-02
14 1 9.058491441733668E-01 2.048904495851298E-01 -3.616174303396834E
-01 8.179282207936479E-02
14 2 9.820467714299637E-01 1.802829790420128E-01 -5.460493936844647E
-02 1.002430986603379E-02
14 3 9.691100682767575E-01 7.602778348815391E-02 2.338990003200156E
-01 -1.834964173475843E-02
14 4 8.72010438336464 6E-01 -4.273391921513095E-02 4.870338197211422E
-01 2.386767748641251E-02
14 5 6.914532566010806E-01 -1.126239688088524E-01 7.043068343068848E
-01 1.147175607050518E-01
15 1 8.931101878555595E-02 1.379911511283162E-02 9.842296173433024E
-01 -1.520696770886562E-01
15 2 2.172058488562003E-01 2.748537164503799E-02 -9.680193390908203E
-01 1.224938068408545E-01
15 3 4.939042183552709E-01 2.708851248598231E-02 -8.677900643277637E
-01 4.759453578071097E-02
15 4 7.199453346730114E-01 -2.471290024283274E-02 -6.931823873341888E
-01 -2.379423320530148E-02
15 5 8.845145072331696E-01 -9.967473775847860E-02 -4.528737046170304E
-01 -5.103372231464564E-02
16 1 8.074349465331383E-01 -4.963412881747241E-01 2.716710570489779E
-01 1.669999087783408E-01
16 2 6.257671071122759E-01 -2.748694471693187E-01 6.683405047921183E
-01 2.935698968276261E-01
16 3 5.137013016227157E-01 -8.533641118521758E-02 8.421733691972330E
-01 1.399024154621241E-01
16 4 3.950364540092338E-01 4.046184345162281E-02 9.129973253871301E
-01 -9.351429336873528E-02
16 5 2.151302897014847E-01 8.074164604265070E-02 9.111803354869825E
-01 -3.419797380973171E-01
17 1 9.932593124275897E-01 -4.636055876955696E-02 -1.061230491497333E
-01 -4.953312589525271E-03
17 2 9.804034430398177E-01 -3.774152436218570E-02 1.932080395357111E
-01 7.437719627440175E-03
17 3 8.801779174852811E-01 -1.485842157103846E-02 4.743436953285682E
-01 8.007470370186781E-03
17 4 7.016151921058168E-01 7.425075178966088E-03 7.124774652091589E
-01 -7.540028782186545E-03
17 5 4.600174974153364E-01 1.581713512747662E-02 8.872446419033536E
-01 -3.050681435155141E-02
18 1 2.447938074436471E-01 -1.605601405648212E-01 7.995479081607871E
-01 5.244230884074035E-01
18 2 2.025253318516516E-01 -9.272556115985661E-02 -8.863898223975842E
-01 -4.058306826940837E-01
18 3 3.942230965392341E-01 -6.742824644941613E-02 -9.034183475472819E
-01 -1.545214258629311E-01
18 4 5.067234450046999E-01 5.338464457055725E-02 -8.557184199710114E
-01 9.015218094400107E-02
18 5 5.955938875367222E-01 2.319653066479208E-01 -7.166310893286774E
-01 2.791055346069361E-01
19 1 7.720382853576208E-01 -3.047362662261537E-01 5.188040385828619E
-01 2.047805252916401E-01
19 2 5.416943877490223E-01 -1.672338888749670E-01 7.871152815474082E
-01 2.430011321938884E-01
19 3 3.234441675648169E-01 -4.065817724291441E-02 9.379915855243139E
-01 1.179091539159300E-01
19 4 1.087768238148519E-01 8.491598965828403E-03 9.910148649031644E
-01 -7.736299431077191E-02
19 5 1.374355119498715E-01 3.713698091082101E-02 -9.555440534594448E
-01 2.582012521313649E-01
20 1 7.482324201961513E-01 -2.828864125330619E-02 -6.623600941456650E
-01 -2.504204118672217E-02
20 2 9.110364009759523E-01 -2.841474859711503E-02 -4.111457625079357E
-01 -1.282342117825051E-02
20 3 9.922148412763675E-01 -1.357624952527699E-02 -1.237841893914412E
-01 -1.693710850263592E-03
20 4 9.846750343830929E-01 8.446804601028890E-03 1.741881035709198E
-01 -1.494231927601520E-03
20 5 8.890887563106934E-01 2.477061326852961E-02 4.568868221040480E
-01 -1.272917545913941E-02
21 1 3.733754757097301E-01 1.563051242415598E-02 -9.267369274929994E
-01 3.879572709366883E-02
21 2 6.313783589276835E-01 2.180242644389263E-02 -7.747066317284940E
-01 2.675176320984878E-02
21 3 8.323954970193430E-01 1.260764288025045E-02 -5.539750727844229E

-01 8.390626699971752E-03
21 4 9.587178119963355E-01 -9.103518543561253E-03 -2.842006338857523E
-01 -2.698631138690785E-03
21 5 9.994605111395030E-01 -3.082797915326324E-02 1.132256178764594E
-02 3.492401099000137E-04
22 1 7.360499673801321E-01 6.213834891831691E-01 2.051948942056190E
-01 -1.732283472247057E-01
22 2 5.265604200960388E-01 2.709240969111380E-01 7.165321794466504E
-01 -3.686677277964547E-01
22 3 5.107071224726305E-01 9.433017258523120E-02 8.403498013105417E
-01 -1.552168323124108E-01
22 4 4.710989089165792E-01 -5.343781829576422E-02 8.748499252787560E
-01 9.923621230756750E-02
22 5 3.769442068127097E-01 -1.624581828757685E-01 8.374133339115589E
-01 3.609145493800009E-01
23 1 7.826587205697425E-01 3.648499609192040E-02 -6.207067899714340E
-01 2.893532546324986E-02
23 2 9.318230454832063E-01 3.582652389237394E-02 -3.608736099469226E
-01 1.387478778461234E-02
23 3 9.974252270606316E-01 1.681668810716765E-02 -6.970462155459448E
-02 1.175226822532046E-03
23 4 9.738358053174236E-01 -1.029308857792146E-02 2.270068712318865E
-01 2.399379670195016E-03
23 5 8.630992282158736E-01 -2.963951843559278E-02 5.038668680912771E
-01 1.730319161186933E-02
24 1 1.822324884041035E-01 6.262502281905151E-02 -9.279908126201899E
-01 3.189082601305299E-01
24 2 5.052439374592572E-01 1.376841293526744E-01 -8.219491095466649E
-01 2.239895209613763E-01
24 3 7.336375685364596E-01 8.274680094156178E-02 -6.702344319272796E
-01 7.559557675529693E-02
24 4 8.751898541488117E-01 -6.141721995447796E-02 -4.786879966633498E
-01 -3.359235238076975E-02
24 5 9.454952240869683E-01 -2.265976151288404E-01 -2.274237644793641E
-01 -5.450443465159887E-02
25 1 7.362950547318129E-01 1.613542229249926E-01 6.419083717904178E
-01 -1.406700015892623E-01
25 2 4.985686151304764E-01 8.877016090559037E-02 8.489414842036282E
-01 -1.511540635835346E-01
25 3 2.351915384652335E-01 1.792386187760919E-02 9.689739771631097E
-01 -7.384515549753312E-02
25 4 3.598908241504489E-02 -1.713657199820929E-03 -9.982197296519906E
-01 -4.753125981356938E-02
25 5 3.127918070488200E-01 -4.943405620015077E-02 -9.369059998927924E
-01 -1.480699392031390E-01
26 1 9.433284356000428E-01 -8.832087981318673E-02 -3.184990555767624E
-01 -2.982006663492064E-02
26 2 9.968134638454417E-01 -7.681139035535053E-02 -2.145218098808135E
-02 -1.653039317398944E-03
26 3 9.612462667605995E-01 -3.236490335731568E-02 2.736297636458553E
-01 9.213040572763908E-03
26 4 8.403838908935394E-01 1.772937103995982E-02 5.415810566830742E
-01 -1.142560156875272E-02
26 5 6.432959182325915E-01 4.423529677175734E-02 7.625379935373420E
-01 -5.243480253463822E-02
27 1 3.276844589027857E-01 -2.023908581783765E-01 7.851656095227039E
-01 4.849492772881000E-01
27 2 9.161758934540849E-02 -4.037056870410952E-02 -9.105004662367039E
-01 -4.012048547659668E-01
27 3 2.951427987989098E-01 -4.914007877428830E-02 -9.412319320196537E
-01 -1.567112986410164E-01
27 4 4.255740978332214E-01 4.368482318937610E-02 -8.991438733474487E
-01 9.229636232322376E-02
27 5 5.417670984831460E-01 2.039111203302674E-01 -7.631537194820572E
-01 2.872369517445659E-01
28 1 9.258950019300451E-01 -1.114992539245159E-01 3.583629292571585E
-01 4.315521648036301E-02
28 2 7.762412449471859E-01 -7.679171352268303E-02 6.227019372908397E
-01 6.160243234912281E-02
28 3 5.613634441744095E-01 -2.419049893941207E-02 8.264487705634886E
-01 3.561366226330747E-02
28 4 2.998741735394000E-01 8.093288879005719E-03 9.535971931810173E
-01 -2.573658634063387E-02
28 5 8.705615127078474E-03 7.679316468141227E-04 9.960939204910445E
-01 -8.786651300087207E-02
29 1 5.672793508308321E-01 -2.610971624377614E-01 -7.094961523471840E
-01 -3.265541604273938E-01
29 2 8.317371316998818E-01 -2.930974450323371E-01 -4.446915275103703E
-01 -1.567057013246909E-01
29 3 9.631319787843612E-01 -1.351614057369034E-01 -2.303544232241239E
-01 -3.232685483040856E-02
29 4 9.956090920167042E-01 8.656280123693848E-02 -3.549497642175796E
-02 3.086095349614286E-03
29 5 9.375001513777785E-01 2.872023113688085E-01 1.878717553752267E
-01 -5.755433991704783E-02
30 1 9.199498565206979E-01 -9.201336906498740E-02 3.791930288780247E
-01 3.792688032474203E-02
30 2 7.650525601623563E-01 -6.295296951946472E-02 6.387246379879611E
-01 5.255797413193955E-02
30 3 5.447187038304241E-01 -1.957193839738205E-02 8.378497523613219E
-01 3.010424210544926E-02
30 4 2.782264230624269E-01 6.263173027509474E-03 9.602518243339955E
-01 -2.161629100351843E-02
30 5 1.514491810704739E-02 1.111897215056471E-03 -9.972008052631104E
-01 7.321167340666053E-02
31 1 5.584546245204914E-01 -3.415201771742409E-01 6.449318982887767E
-01 3.944049283467266E-01
31 2 2.017237881966568E-01 -8.830152846283960E-02 8.935921904978201E
-01 3.911564270570357E-01
31 3 1.570500020147196E-02 -2.602471871457638E-03 9.864215688431847E
-01 1.634596850290135E-01
31 4 1.237209847095429E-01 1.264189989234563E-02 -9.870968054836058E
-01 1.008622670460803E-01
31 5 2.790331355902165E-01 1.044001768542662E-01 -8.940596738206834E
-01 3.345122000213785E-01
32 1 8.863028540330694E-01 -4.540049155232727E-01 8.131336196933862E

```
     -02 4.165242824596101E-02
 32 2 8.084272935884596E-01 -3.107425336630780E-01 4.666018194266670E
     -01 1.793519747915081E-01
 32 3 7.132253183669763E-01 -1.072398776818520E-01 6.849829323818974E
     -01 1.029933794988961E-01
 32 4 5.835274155061845E-01 5.426198866399944E-02 8.067978792499050E
     -01 -7.502382272822816E-02
 32 5 3.782887861034087E-01 1.256838803514378E-01 8.703365341698036E
     -01 -2.891634033163771E-01
```

Equilibrium_Angles.in

```
 1 2 6.0E-01 0.0E+00
 1 3 6.0E-01 0.0E+00
 1 4 6.0E-01 0.0E+00
 1 5 6.0E-01 0.0E+00
 2 2 6.0E-01 0.0E+00
 2 3 6.0E-01 0.0E+00
 2 4 6.0E-01 0.0E+00
 2 5 6.0E-01 0.0E+00
 3 2 6.0E-01 0.0E+00
 3 3 6.0E-01 0.0E+00
 3 4 6.0E-01 0.0E+00
 3 5 6.0E-01 0.0E+00
 4 2 6.0E-01 0.0E+00
 4 3 6.0E-01 0.0E+00
 4 4 6.0E-01 0.0E+00
 4 5 6.0E-01 0.0E+00
 5 2 6.0E-01 0.0E+00
 5 3 6.0E-01 0.0E+00
 5 4 6.0E-01 0.0E+00
 5 5 6.0E-01 0.0E+00
 6 2 6.0E-01 0.0E+00
 6 3 6.0E-01 0.0E+00
 6 4 6.0E-01 0.0E+00
 6 5 6.0E-01 0.0E+00
 7 2 6.0E-01 0.0E+00
 7 3 6.0E-01 0.0E+00
 7 4 6.0E-01 0.0E+00
 7 5 6.0E-01 0.0E+00
 8 2 6.0E-01 0.0E+00
 8 3 6.0E-01 0.0E+00
 8 4 6.0E-01 0.0E+00
 8 5 6.0E-01 0.0E+00
 9 2 6.0E-01 0.0E+00
 9 3 6.0E-01 0.0E+00
 9 4 6.0E-01 0.0E+00
 9 5 6.0E-01 0.0E+00
10 2 6.0E-01 0.0E+00
10 3 6.0E-01 0.0E+00
10 4 6.0E-01 0.0E+00
10 5 6.0E-01 0.0E+00
11 2 6.0E-01 0.0E+00
11 3 6.0E-01 0.0E+00
11 4 6.0E-01 0.0E+00
11 5 6.0E-01 0.0E+00
12 2 6.0E-01 0.0E+00
12 3 6.0E-01 0.0E+00
12 4 6.0E-01 0.0E+00
12 5 6.0E-01 0.0E+00
13 2 6.0E-01 0.0E+00
13 3 6.0E-01 0.0E+00
13 4 6.0E-01 0.0E+00
13 5 6.0E-01 0.0E+00
14 2 6.0E-01 0.0E+00
14 3 6.0E-01 0.0E+00
14 4 6.0E-01 0.0E+00
14 5 6.0E-01 0.0E+00
15 2 6.0E-01 0.0E+00
15 3 6.0E-01 0.0E+00
15 4 6.0E-01 0.0E+00
15 5 6.0E-01 0.0E+00
16 2 6.0E-01 0.0E+00
16 3 6.0E-01 0.0E+00
16 4 6.0E-01 0.0E+00
16 5 6.0E-01 0.0E+00
17 2 6.0E-01 0.0E+00
17 3 6.0E-01 0.0E+00
17 4 6.0E-01 0.0E+00
17 5 6.0E-01 0.0E+00
18 2 6.0E-01 0.0E+00
18 3 6.0E-01 0.0E+00
18 4 6.0E-01 0.0E+00
18 5 6.0E-01 0.0E+00
19 2 6.0E-01 0.0E+00
19 3 6.0E-01 0.0E+00
19 4 6.0E-01 0.0E+00
19 5 6.0E-01 0.0E+00
20 2 6.0E-01 0.0E+00
20 3 6.0E-01 0.0E+00
20 4 6.0E-01 0.0E+00
20 5 6.0E-01 0.0E+00
21 2 6.0E-01 0.0E+00
21 3 6.0E-01 0.0E+00
21 4 6.0E-01 0.0E+00
21 5 6.0E-01 0.0E+00
22 2 6.0E-01 0.0E+00
22 3 6.0E-01 0.0E+00
22 4 6.0E-01 0.0E+00
22 5 6.0E-01 0.0E+00
23 2 6.0E-01 0.0E+00
23 3 6.0E-01 0.0E+00
23 4 6.0E-01 0.0E+00
23 5 6.0E-01 0.0E+00
24 2 6.0E-01 0.0E+00
```

```
24 3 6.0E-01 0.0E+00
24 4 6.0E-01 0.0E+00
24 5 6.0E-01 0.0E+00
25 2 6.0E-01 0.0E+00
25 3 6.0E-01 0.0E+00
25 4 6.0E-01 0.0E+00
25 5 6.0E-01 0.0E+00
26 2 6.0E-01 0.0E+00
26 3 6.0E-01 0.0E+00
26 4 6.0E-01 0.0E+00
26 5 6.0E-01 0.0E+00
27 2 6.0E-01 0.0E+00
27 3 6.0E-01 0.0E+00
27 4 6.0E-01 0.0E+00
27 5 6.0E-01 0.0E+00
28 2 6.0E-01 0.0E+00
28 3 6.0E-01 0.0E+00
28 4 6.0E-01 0.0E+00
28 5 6.0E-01 0.0E+00
```

```
29 2 6.0E-01 0.0E+00
29 3 6.0E-01 0.0E+00
29 4 6.0E-01 0.0E+00
29 5 6.0E-01 0.0E+00
30 2 6.0E-01 0.0E+00
30 3 6.0E-01 0.0E+00
30 4 6.0E-01 0.0E+00
30 5 6.0E-01 0.0E+00
31 2 6.0E-01 0.0E+00
31 3 6.0E-01 0.0E+00
31 4 6.0E-01 0.0E+00
31 5 6.0E-01 0.0E+00
32 2 6.0E-01 0.0E+00
32 3 6.0E-01 0.0E+00
32 4 6.0E-01 0.0E+00
32 5 6.0E-01 0.0E+00
```

BIBLIOGRAPHY

Abdul Khalil, H., Davoudpour, Y., Islam, M. N., Mustapha, A., Sudesh, K., Dungani, R., and Jawaid, M. Production and modification of nanofibrillated cellulose using various mechanical processes: A review. *Carbohydr. Polym.*, 99:649–665, 2014.

Acrivos, A., Batchelor, G. K., Hinch, E. J., Koch, D. L., and Mauri, R. Longitudinal shear-induced diffusion of spheres in a dilute suspension. *J. Fluid Mech.*, 240:651–657, 1992.

Agoda-Tandjawa, G., Durand, S., Berot, S., Blassel, C., Gaillard, C., Garnier, C., and Doublier, J. L. Rheological characterization of microfibrillated cellulose suspensions after freezing. *Carbohydr. Polym.*, 80(3):677–686, 2010.

Allen, M. P. and Tildesley, D. J. *Computer Simulation of Liquids*. Second edi edition, 2017.

Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conf. Proc. - 1967 Spring Jt. Comput. Conf. AFIPS 1967*, 1967.

Andresen, M. and Stenius, P. Water-in-oil Emulsions Stabilized by Hydrophobized Microfibrillated Cellulose. *J. Dispers. Sci. Technol.*, 28(6):837–844, 2007.

Andresen, M., Johansson, L. S., Tanem, B. S., and Stenius, P. Properties and characterization of hydrophobized microfibrillated cellulose. *Cellulose*, 13(6):665–677, 2006.

Andrienko, D. Introduction to liquid crystals. *J. Mol. Liq.*, 267:520–541, 2018.

Asada, T., Muramatsu, H., Watanabe, R., and Onogi, S. Rheooptical studies of racemic poly(γ-benzyl glutamate) liquid crystals. *Macromolecules*, 13(4):867–871, 1980.

Attard, G. S., Bartlett, P. N., Coleman, N. R., Elliott, J. M., Owen, J. R., and Wang, J. H. Mesoporous platinum films from lyotropic liquid crystalline phases. *Science*, 278(5339): 838–840, 1997.

Aulin, C., Gällstedt, M., and Lindström, T. Oxygen and oil barrier properties of microfibrillated cellulose films and coatings. *Cellulose*, 17(3):559–574, 2010.

Avalos, J. B. Dynamics of semiflexible and rigid particles. I. The velocity distribution and the Smoluchowski equation. *Phys. Rev. E*, 54(4):3955–3970, 1996.

Azizi Samir, M. A. S., Alloin, F., and Dufresne, A. Review of recent research into cellulosic whiskers, their properties and their application in nanocomposite field. *Biomacromolecules*, 6(2):612–626, 2005.

Bagheriasl, D., Carreau, P. J., Riedl, B., Dubois, C., and Hamad, W. Y. Shear rheology of polylactide (PLA)-cellulose nanocrystal (CNC) nanocomposites. *Cellulose*, 23(3):1885–1897, 2016.

Bagheriasl, D., Safdari, F., Carreau, P. J., Dubois, C., and Riedl, B. Development of cellulose nanocrystal-reinforced polylactide: A comparative study on different preparation methods. *Polym. Compos.*, 40:342–349, 2019.

Barnes, H. A. Thixotropy - A review. *J. Non-Newton Fluid*, 70(97):1–33, 1997.

Batchelor, G. K. The stress system in a suspension of force-free particles. *J . Fluid Mech*, 41(3):545–570, 1970a.

Batchelor, G. K. Slender-body theory for particles of arbitrary cross-section in Stokes flow. *J. Fluid Mech.*, 44(03):419–440, 1970b.

Bennington, C. P. J., Kerekes, R. J., and Grace, J. R. The yield stress of fibre suspensions. *Can. J. Chem. Eng.*, 68(5):748–757, 1990.

Bercea, M. and Navard, P. Shear dynamics of aqueous suspensions of cellulose whiskers. *Macromolecules*, 33(16):6011–6016, 2000.

Bette, H. *Brownian Dynamics Simulation of Flexible Fibers*. PhD thesis, University of Wisconsin-Madison, 2003.

Bibbo, M. A., Dinh, S. M., and Armstrong, R. C. Shear Flow Properties of Semiconcentrated Fiber Suspensions. *J. Rheol.*, 29(6):905–929, 1985.

Björkman, U. Break-up of suspended fibre networks. *Nord. Pulp Pap. Res. J.*, 18(1):32–37, 2003.

Bolhuis, P. and Frenkel, D. Tracing the phase boundaries of hard spherocylinders. *J. Chem. Phys.*, 106(2):666–687, 1997.

Bolhuis, P. G., Stroobants, A., Frenkel, D., and Lekkerkerker, H. N. Numerical study of the phase behavior of rodlike colloids with attractive interactions. *J. Chem. Phys.*, 107 (5):1551–1564, 1997.

Bossis, G. and Brady, J. F. The rheology of Brownian suspensions. *J. Chem. Phys.*, 91(3): 1866–1874, 1989.

Brady, J. and Bossis, G. Stokesian Dynamics. *Annu. Rev. Fluid Mech.*, 20(1):111–157, 1988.

Bretherton, F. P. The motion of rigid particles in a shear flow at low Reynolds number. *J. Fluid Mech.*, 14(2):284–304, 1962.

Burghardt, W. R. Molecular orientation and rheology in sheared lyotropic liquid crystalline polymers. *Macromol. Chem. Phys.*, 199:471–488, 1998.

Chaouche, M. and Koch, D. L. Rheology of non-Brownian rigid fiber suspensions with adhesive contacts. *J. Rheol.*, 45(2):369–382, 2001.

Chen, B., Tatsumi, D., and Matsumoto, T. Floc Structure and Flow Properties of Pulp Fiber Suspensions. *J. Soc. Rheol. Japan*, 30(1):19–25, 2002.

Chen, J. C. and Kim, A. S. Brownian dynamics, molecular dynamics, and monte carlo modeling of colloidal systems. *Adv. Colloid Interface Sci.*, 112(1-3):159–173, 2004.

Chen, P., Yu, H., Liu, Y., Chen, W., Wang, X., and Ouyang, M. Concentration effects on the isolation and dynamic rheological behavior of cellulose nanofibers via ultrasonic processing. *Cellulose*, 20(1):149–157, 2013.

Cinacchi, G., Mederos, L., and Velasco, E. Liquid-crystal phase diagrams of binary mixtures of hard spherocylinders. *J. Chem. Phys.*, 121(8):3854–3863, 2004.

Clark, N. A. and Lagerwall, S. T. Chiral Smectic C or H Liquid Crystal Electro-optical device, 1983.

Córdoba, A., Indei, T., and Schieber, J. D. Elimination of inertia from a Generalized Langevin Equation: Applications to microbead rheology modeling and data analysis. *J. Rheol.*, 56(1):185–212, 2012.

Cox, R. G. The motion of long slender bodies in a viscous fluid Part 1. General theory. *J. Fluid Mech.*, 44(04):791–810, 1970.

Cox, R. G. The motion of long slender bodies in a viscous fluid Part 2. Shear flow. *J. Fluid Mech.*, 45(4):625–657, 1971.

Danckwerts, P. V. The definition and measurement of some characteristics of mixtures. *Appl. Sci. Res. Sect. A*, 3(4):279–296, 1952.

De Souza Lima, M. M. and Borsali, R. Rodlike cellulose microcrystals: Structure, properties, and applications. *Macromol. Rapid Commun.*, 25(7):771–787, 2004.

Derakhshandeh, B., Kerekes, R. J., Hatzikiriakos, S. G., and Bennington, C. P. J. Rheology of pulp fibre suspensions: A critical review. *Chem. Eng. Sci.*, 66(15):3460–3470, 2011.

Diebel, J. Representing attitude: Euler angles, unit quaternions, and rotation vectors. *Matrix*, 58:1–35, 2006.

Dong, X. M., Kimura, T., Revol, J.-F., and Gray, D. G. Effects of ionic strength on the isotropic-chiral nematic phase transition of suspensions of cellulose crystallites. *Langmuir*, 12(8):2076–2082, 2002.

Eckstein, E. C., Bailey, D. G., and Shapiro, A. H. Self-diffusion of particles in shear flow of a suspension. *J. Fluid Mech.*, 79(1):191–208, 1977.

Eichhorn, S. J., Dufresne, A., Aranguren, M., Marcovich, N. E., Capadona, J. R., Rowan, S. J., Weder, C., Thielemans, W., Roman, M., Renneckar, S., Gindl, W., Veigel, S., Keckes, J., Yano, H., Abe, K., Nogi, M., Nakagaito, A. N., Mangalam, A., Simonsen, J., Benight, A. S., Bismarck, A., Berglund, L. A., and Peijs, T. Review: Current international research into cellulose nanofibres and nanocomposites. *J. Mater. Sci.*, 45(1):1–33, 2010.

Eken, A. E., Tozzi, E. J., Klingenberg, D. J., and Bauhofer, W. Combined effects of nanotube aspect ratio and shear rate on the carbon nanotube/polymer composites. *Polymer*, 53 (20):4493–4500, 2012.

Elazzouzi-Hafraoui, S., Nishiyama, Y., Putaux, J. L., Heux, L., Dubreuil, F., and Rochas, C. The shape and size distribution of crystalline nanoparticles prepared by acid hydrolysis of native cellulose. *Biomacromolecules*, 9(1):57–65, 2008.

Fall, A. B., Lindström, S. B., Sundman, O., Ödberg, L., and Wågberg, L. Colloidal stability of aqueous nanofibrillated cellulose dispersions. *Langmuir*, 27(18):11332–11338, 2011.

Fixman, M. Simulation of polymer dynamics. I. General theory. *J. Chem. Phys.*, 69(4): 1527–1537, 1978.

Flory, P. J. Phase equilibria in solutions of rod-like particles. *Proc. R. Soc. A- Math. Phys. Eng. Sci.*, 234(1196):73–89, 1956.

Ford, G. W. Rotational Brownian motion of an asymmetric top. *Phys. Lett. A*, 77(3): 249–254, 1981.

Ford, G. W., Lewis, J. T., and McConnell, J. Rotational Brownian motion of an asymmetric top. *Phys. Lett. A*, 63(3):207–208, 1977.

Forgacs, O. and Mason, S. Particle motions in sheared suspensions. *J. Colloid Sci.*, 14(5): 473–491, 1959.

Fraden, S., Maret, G., Caspar, D. L. D., and Meyer, R. B. Isotropic-nematic phase transition and angular correlations in isotropic suspensions of tobacco mosaic virus. *Phys. Rev. Lett.*, 63(19):2068–2071, 1989.

Ganani, E. and Powell, R. L. Suspensions of rodlike particles: Literature review and data correlations. *J. Compos. Mater.*, 19(3):194–215, 1985.

Garboczi, E. J., Snyder, K. A., Douglas, J. F., and Thorpe, M. F. Geometrical percolation threshold of overlapping ellipsoids. *Phys. Rev. E*, 52(1):819–828, 1995.

Ghanadpour, M., Carosio, F., Larsson, P. T., and Wågberg, L. Phosphorylated Cellulose Nanofibrils: A Renewable Nanomaterial for the Preparation of Intrinsically Flame-Retardant Materials. *Biomacromolecules*, 16(10):3399–3410, 2015.

Goldsmith, H. L. and Marlow, J. C. Flow behavior of erythrocytes. II. Particle motions in concentrated suspensions of ghost cells. *J. Colloid Interface Sci.*, 71(2):383–407, 1979.

Goldsmith, H. L. and Mason, S. G. The flow of suspensions through tubes. I. Single spheres, rods, and discs. *J. Colloid Sci.*, 17(5):448–476, 1962.

Gonnet, P. Pairwise verlet lists: Combining cell lists and verlet lists to improve memory locality and parallelism. *J. Comput. Chem.*, 33(1):76–81, 2012.

Goussé, C., Chanzy, H., Cerrada, M. L., and Fleury, E. Surface silylation of cellulose microfibrils: Preparation and rheological properties. *Polymer*, 45(5):1569–1575, 2004.

Graham, M. D. *Microhydrodynamics, Brownian Motion, and Complex Fluids*. Cambridge University Press, 2018.

Grassia, P. and Hinch, E. J. Computer simulations of polymer chain relaxation via Brownian motion. *J. Fluid Mech.*, 308(1996):255–288, 1996.

Grassia, P. S., Hinch, E. J., and Nitsche, L. C. Computer simulations of Brownian motion of complex systems. *J. Fluid Mech.*, 282:373–403, 1995.

Guhados, G., Wan, W., and Hutter, J. L. Measurement of the elastic modulus of single bacterial cellulose fibers using atomic force microscopy. *Langmuir*, 21(14):6642–6646, 2005.

Habibi, Y., Goffin, A. L., Schiltz, N., Duquesne, E., Dubois, P., and Dufresne, A. Bionanocomposites based on poly(ϵ-caprolactone)-grafted cellulose nanocrystals by ring-opening polymerization. *J. Mater. Chem.*, 18(41):5002–5010, 2008.

Habibi, Y., Lucia, L. A., and Rojas, O. J. Cellulose nanocrystals: Chemistry, self-assembly, and applications. *Chem. Rev.*, 110(6):3479–3500, 2010.

Haug, E. J. *Intermediate Dynamics*. Prentice Hall, 1992.

Heggset, E. B., Chinga-Carrasco, G., and Syverud, K. Temperature stability of nanocellulose dispersions. *Carbohydr. Polym.*, 157:114–121, 2017.

Herrick, F. W., Casebier, R. L., Hamilton, K. J., and Sandberg, K. R. Microfibrillated Cellulose: Morphology and Accessibility. *J. Appl. Polym. Sci. Appl. Polym. Symp.*, 37: 797–813, 1983.

Hill, M. D. and Marty, M. R. Amdahl's law in the multicore era. *Computer*, pages 33–38, 2008.

Hinch, E. J. The distortion of a flexible inextensible thread in a shearing flow. *J. Fluid Mech.*, 74(2):317–333, 1976.

Hinch, E. J. Brownian motion with stiff bonds and rigid constraints. *J. Fluid Mech.*, 271: 219–234, 1994.

Hinch, E. J. and Leal, L. G. Rotation of small non-axisymmetric particles in a simple shear flow. *J. Fluid Mech.*, 92(3):591–607, 1979.

Hobbie, E. K., Jeon, H. S., Wang, H., Kim, H., Stout, D. J., and Han, C. C. Shear-induced structure in polymer blends with viscoelastic asymmetry. *J. Chem. Phys.*, 117(13):6350–6359, 2002.

Hong, Y., Blackman, N. M., Kopp, N. D., Sen, A., and Velegol, D. Chemotaxis of nonbiological colloidal rods. *Phys. Rev. Lett.*, 99(17):1–4, 2007.

Hoshen, J. and Kopelman, R. Percolation and cluster distribution. I. Cluster multiple labeling technique and critical concentration algorithm. *Phys. Rev. B*, 14(8):3438–3445, 1976.

Houssa, M., Rull, L. F., and McGrother, S. C. Effect of dipolar interactions on the phase behavior of the Gay-Berne liquid crystal model. *J. Chem. Phys.*, 109(21):9529–9542, 1998.

Hubbard, P. S. Rotational brownian motion. *Phys. Rev. A*, 6(6):2421–2433, 1972.

Iotti, M., Gregersen, Ø. W., Moe, S., and Lenes, M. Rheological studies of microfibrillar cellulose water dispersions. *J. Polym. Environ.*, 19(1):137–145, 2011.

Ishii, H., Sugimura, K., and Nishio, Y. Thermotropic liquid crystalline properties of (hydroxypropyl)cellulose derivatives with butyryl and heptafluorobutyryl substituents. *Cellulose*, 26(1):399–412, 2019.

István, S. and Plackett, D. Microfibrillated cellulose and new nanocomposite materials: A review. *Cellulose*, 17(3):459–494, 2010.

Ito, M., Sekine, N., Ishizaki, M., Kina, O., and Matsubara, R. Structure, transmission yype liquid crystal display, reflection type display and manufacturing method thereof, 2007.

Ivanov, Y., van de Ven, T. G. M., and Mason, S. G. Damped oscillations in the viscosity of suspensions of rigid rods. I. Monomodal suspensions. *J. Rheol.*, 26(2):213–230, 1982.

Iwamoto, S., Abe, K., and Yano, H. The effect of hemicelluloses on wood pulp nanofibrillation and nanofiber network characteristics. *Biomacromolecules*, 9:1022–1026, 2008.

Iwamoto, S., Kai, W., Isogai, A., and Iwata, T. Elastic modulus of single cellulose microfibrils from tunicate measured by atomic force microscopy. *Biomacromolecules*, 10(1): 2571–2576, 2009.

Iwamoto, S., Isogai, A., and Iwata, T. Structure and mechanical properties of wet-spun fibers made from natural cellulose nanofibers. *Biomacromolecules*, 12(3):831–836, 2011.

Jákli, A., Pintre, I. C., Serrano, J. L., Ros, M. B., and De La Fuente, M. R. Piezoelectric and electric-field-induced properties of a ferroelectric bent-core liquid crystal. *Adv. Mater.*, 21(37):3784–3788, 2009.

Jardin, J. M., Zhang, Z., Hu, G., Tam, K. C., and Mekonnen, T. H. Reinforcement of rubber nanocomposite thin sheets by percolation of pristine cellulose nanocrystals. *Int. J. Biol. Macromol.*, 152:428–436, 2020.

Jeffery, G. B. The motion of ellipsoidal particles immersed in a viscous fluid. *Proc. R. Soc. London, Ser. A*, 102(715):161–179, 1922.

Jia, C., Chen, L., Shao, Z., Agarwal, U. P., Hu, L., and Zhu, J. Y. Using a fully recyclable dicarboxylic acid for producing dispersible and thermally stable cellulose nanomaterials from different cellulosic sources. *Cellulose*, 24(6):2483–2498, 2017.

Jiang, G., Turner, T. A., and Pickering, S. J. The shear viscosity of carbon fibre suspension and its application for fibre length measurement. *Rheol. Acta*, 55(1):1–10, 2016.

Karppinen, A., Saarinen, T., Salmela, J., Laukkanen, A., Nuopponen, M., and Seppälä, J. Flocculation of microfibrillated cellulose in shear flow. *Cellulose*, 19(6):1807–1819, 2012.

Kassab, Z., Aziz, F., Hannache, H., Ben Youcef, H., and El Achaby, M. Improved mechanical properties of k-carrageenan-based nanocomposite films reinforced with cellulose nanocrystals. *Int. J. Biol. Macromol.*, 123:1248–1256, 2019.

Kerekes, R. J. and Schell, C. J. Characterization of fibre flocculation regimes by a crowding factor. *J. pulp Pap. Sci.*, 18(1):J32–J38, 1992.

Khalilzadeh, M. A., Tajik, S., Beitollahi, H., and Venditti, R. A. Green synthesis of magnetic nanocomposite with iron oxide deposited on cellulose nanocrystals with copper (Fe3O4@CNC/Cu): Investigation of catalytic activity for the development of a venlafaxine electrochemical sensor. *Ind. Eng. Chem. Res.*, 59(10):4219–4228, 2020.

Kim, J. W., Park, H., Lee, G., Jeong, Y. R., Hong, S. Y., Keum, K., Yoon, J., Kim, M. S., and Ha, J. S. Paper-like, thin, foldable, and self-healable electronics based on PVA/CNC nanocomposite film. *Adv. Funct. Mater.*, 29(50):1–14, 2019.

Kim, S. and Karilla, S. J. *Microhydrodynamics : Principles and Selected Applications*. Butterworth-Heinemann, 1991.

Kim, Y. J. and Rae, W. J. Separation of screw-sensed particles in a homogeneous shear field. *Int. J. Multiph. Flow*, 17(6):717–744, 1991.

Klemm, D., Cranston, E. D., Fischer, D., Gama, M., Kedzior, S. A., Kralisch, D., Kramer, F., Kondo, T., Lindström, T., Nietzsche, S., Petzold-Welcke, K., and Rauchfuß, F. Nanocellulose as a natural source for groundbreaking applications in materials science: Today's state. *Mater. Today*, 21(7):720–748, 2018.

Kobayashi, Y., Arai, N., and Nikoubashman, A. Structure and dynamics of amphiphilic Janus spheres and spherocylinders under shear. *Soft Matter*, 16(2):476–486, 2020.

Koch, D. L. On hydrodynamic diffusion and drift in sheared suspensions. *Phys. Fluids A*, 1(10):1742–1744, 1989.

Kubo, R. Fluctuation-dissipation theorem. *Rep. Prog. Phys.*, 29(3):255–284, 1966.

Kubo, R., Yokota, M., and Nakajima, S. Statistical-mechanical theory of irreversible processes. II. Response to thermal disturbance. *J. Phys. Soc. Japan*, 12(11):1203–1211, 1957.

Lagerwall, J. P., Schütz, C., Salajkova, M., Noh, J., Park, J. H., Scalia, G., and Bergström, L. Cellulose nanocrystal-based materials: From liquid crystal self-assembly and glass formation to multifunctional thin films. *NPG Asia Mater.*, 6(1):1–12, 2014.

Lasseuguette, E., Roux, D., and Nishiyama, Y. Rheological properties of microfibrillar suspension of TEMPO-oxidized pulp. *Cellulose*, 15(3):425–433, 2008.

Lavoine, N., Desloges, I., Dufresne, A., and Bras, J. Microfibrillated cellulose - Its barrier properties and applications in cellulosic materials: A review. *Carbohydr. Polym.*, 90(2): 735–764, 2012.

Leal, L. G. and Hinch, E. J. The effect of weak Brownian rotations on particles in shear flow. *J. Fluid Mech.*, 46(4):685–703, 1971.

Lees, A. W. and Edwards, S. F. The computer study of transport processes under extreme conditions. *J. Phys. C Solid State Phys.*, 5(15):1921–1928, 1972.

Leighton, D. and Acrivos, A. Viscous resuspension. *Chem. Eng. Sci.*, 41(6):1377–1384, 1986.

Leighton, D. and Acrivos, A. Measurement of shear induced self diffusion in concentrated suspensions of spheres. *J. Fluid Mech.*, 177:109–131, 1987.

Li, M. C., Wu, Q., Song, K., Lee, S., Qing, Y., and Wu, Y. Cellulose nanoparticles: Structure-morphology-rheology relationships. *ACS Sustain. Chem. Eng.*, 3(5):821–832, 2015.

Lin, C., Ma, Q., Su, Q., Bian, H., and Zhu, J. Y. Facile synthesis of highly hydrophobic cellulose nanoparticles through post-esterification microfluidization. *Fibers*, 6(2), 2018.

Lin-Gibson, S., Pathak, J. A., Grulke, E. A., Wang, H., and Hobbie, E. K. Elastic flow instability in nanotube suspensions. *Phys. Rev. Lett.*, 92(4):048302, 2004.

Lindström, S. B. and Uesaka, T. Simulation of semidilute suspensions of non-Brownian fibers in shear flow. *J. Chem. Phys.*, 128(2), 2008a.

Lindström, S. B. and Uesaka, T. Particle-level simulation of forming of the fiber network in papermaking. *Int. J. Eng. Sci.*, 46(9):858–876, 2008b.

Lintuvuori, J. S. and Wilson, M. R. A new anisotropic soft-core model for the simulation of liquid crystal mesophases. *J. Chem. Phys.*, 128(4), 2008.

Lopez, M. and Graham, M. D. Shear-induced diffusion in dilute suspensions of spherical or nonspherical particles: Effects of irreversibility and symmetry breaking. *Phys. Fluids*, 19(7), 2007.

Lowen, H. Brownian dynamics of hard spherocylinders. *Phys. Rev. E*, 50(2):1232–1242, 1994.

Lowys, M. P., Desbrières, J., and Rinaudo, M. Rheological characterization of cellulosic microfibril suspensions. Role of polymeric additives. *Food Hydrocoll.*, 15(1):25–32, 2001.

Lundell, F. The effect of particle inertia on triaxial ellipsoids in creeping shear: From drift toward chaos to a single periodic solution. *Phys. Fluids*, 23(1):1–5, 2011.

Martinez, D., Buckley, K., Jivan, S., Lindström, A., Thiruvengadaswamy, R., Olson, J., Ruth, T., and Kerekes, R. Characterizing the mobility of papermaking fibres during sedimentation. *12th Fundam. Res. Symp.*, 16:225–254, 2001.

Mason, S. The flocculation of pulp suspensions and the formation of paper. *Tappi J.*, 33 (9):440–444, 1950.

McGrother, S. C., Williamson, D. C., and Jackson, G. A re-examination of the phase diagram of hard spherocylinders. *J. Chem. Phys.*, 104(17):6755–6771, 1996.

Meyer, R. and Wahren, D. On the Elastic Properties of Three-Dimensional Fibre networks. *Sven. Papperstidning*, 67(10):432–436, 1964.

Meyer, R. B. Piezoelectric effects in liquid crystals. *Phys. Rev. Lett.*, 22(18):918–921, 1969.

Missoum, K., Bras, J., and Belgacem, M. N. Water redispersible dried nanofibrillated cellulose by adding sodium chloride. *Biomacromolecules*, 13(12):4118–4125, 2012.

Moberg, T., Sahlin, K., Yao, K., Geng, S., Westman, G., Zhou, Q., Oksman, K., and Rigdahl, M. Rheological properties of nanocellulose suspensions: effects of fibril/particle dimensions and surface characteristics. *Cellulose*, 24(6):2499–2510, 2017.

Mohraz, A. and Solomon, M. J. Direct visualization of colloidal rod assembly by confocal microscopy. *Langmuir*, 21(12):5298–5306, 2005.

Mongruel, A. and Cloitre, M. Shear viscosity of suspensions of aligned non-Brownian fibres. *Rheol. Acta*, 38(5):451–457, 1999.

Moon, R. J., Beck, S., and Rudie, A. W. Cellulose nanocrystals- a material with unique properties and many potential applications. In Postek, M. T., Moon, R. J., Rudie, A. W., and Bilodeau, M. A., editors, *Prod. Appl. Cellul. Nanomater.*, chapter 1, pages 9–12. TAPPI Press, Peachtree Corners, GA, 2013.

Moon, R. J., Martini, A., Nairn, J., Simonsen, J., and Youngblood, J. Cellulose nanomaterials review: structure, properties and nanocomposites. *Chem Soc Rev*, 40(7):3941–3994, 2011.

Mori, H. Transport, Collective Motion, and Brownian Motion. *Prog. Theor. Phys.*, 33(3): 423–455, 1965.

Mott, P. H. and Roland, C. M. Limits to Poisson's ratio in isotropic materials. *Phys. Rev. B - Condens. Matter Mater. Phys.*, 80(13):1–4, 2009.

Nath, T. and Heussinger, C. Rheology in dense assemblies of spherocylinders: Frictional vs. frictionless. *Eur. Phys. J. E*, 42(12):1–8, 2019.

Nogi, M., Iwamoto, S., Nakagaito, A. N., and Yano, H. Optically transparent nanofiber paper. *Adv. Mater.*, 21(16):1595–1598, 2009.

Notley, S. M., Pettersson, B., and Wågberg, L. Direct measurement of attractive van der Waals' forces between regenerated cellulose surfaces in an aqueous environment. *J. Am. Chem. Soc.*, 126(43):13930–13931, 2004.

Onogi, S. and Asada, T. Rheology and rheo-optics of polymer liquid crystals. In Astarita, G., Marrucci, G., and Nicolais, L., editors, *Rheology*, pages 127–147. Plenum Press, New York, 1980.

Orts, W. J., Godbout, L., Marchessault, R. H., and Revol, J. F. Enhanced ordering of liquid crystalline suspensions of cellulose microfibrils: A small angle neutron scattering study. *Macromolecules*, 31(17):5717–5725, 1998.

Pääkkö, M., Ankerfors, M., Kosonen, H., Nykänen, A., Ahola, S., Österberg, M., Ruokolainen, J., Laine, J., Larsson, P. T., Ikkala, O., and Lindström, T. Enzymatic hydrolysis combined with mechanical shearing and high-pressure homogenization for nanoscale cellulose fibrils and strong gels. *Biomacromolecules*, 8(6):1934–1941, 2007.

Petrich, M. P., Koch, D. L., and Cohen, C. Experimental determination of the stress-microstructure relationship in semi-concentrated fiber suspensions. *J. Non-Newton Fluid*, 95(2-3):101–133, 2000.

Phan-Xuan, T., Thuresson, A., Skepö, M., Labrador, A., Bordes, R., and Matic, A. Aggregation behavior of aqueous cellulose nanocrystals: the effect of inorganic salts. *Cellulose*, 23(6):3653–3663, 2016.

Pignon, F., Magnin, A., and Piau, J.-M. Butterfly Light Scattering Pattern and Rheology of a Sheared Thixotropic Clay Gel. *Phys. Rev. Lett.*, 79(23):4689–4692, 1997.

Ross, R. F. and Klingenberg, D. J. Dynamic simulation of flexible fibers composed of linked rigid bodies. *J. Chem. Phys.*, 106(7):2949–2960, 1997.

Saito, T., Nishiyama, Y., Putaux, J. L., Vignon, M., and Isogai, A. Homogeneous suspen-

sions of individualized microfibrils from TEMPO-catalyzed oxidation of native cellulose. *Biomacromolecules*, 7(6):1687–1691, 2006.

Sakurada, I., Nukushina, Y., and Ito, T. Experimental determination of the elastic modulus of crystalline regions in oriented polymers. *J. Polym. Sci.*, 57(165):651–660, 1962.

Samaniuk, J. R., Scott, C. T., Root, T. W., and Klingenberg, D. J. Rheological modification of corn stover biomass at high solids concentrations. *J. Rheol.*, 56(3):649–665, 2012.

Samyn, P. and Taheri, H. Rheology of fibrillated cellulose suspensions after surface modification by organic nanoparticle deposits. *J. Mater. Sci.*, 51(21):9830–9848, 2016.

Sanders, J. and Kandrot, E. *CUDA by example: An introduction to general-purpose GPU programming*. Addison-Wesley, Upper Saddle Riber, NJ, 2011.

Schmid, C. F. and Klingenberg, D. J. Mechanical flocculation in flowing fiber suspensions. *Phys. Rev. Lett.*, 84(2):290–293, 2000a.

Schmid, C. F. and Klingenberg, D. J. Properties of fiber flocs with frictional and attractive interfiber forces. *J. Colloid Interface Sci.*, 226(1):136–144, 2000b.

Schmid, C. F., Switzer, L. H., and Klingenberg, D. J. Simulations of fiber flocculation: Effects of fiber properties and interfiber friction. *J. Rheol.*, 44(4):781–809, 2000.

Shafiei-Sabet, S., Hamad, W. Y., and Hatzikiriakos, S. G. Ionic strength effects on the microstructure and shear rheology of cellulose nanocrystal suspensions. *Cellulose*, 21: 3347–3359, 2014.

Shafiei-Sabet, S., Hamad, W. Y., and Hatzikiriakos, S. G. Rheology of nanocrystalline cellulose aqueous suspensions. *Langmuir*, 28:17124–17133, 2012.

Shafiei-Sabet, S., Hamad, W. Y., and Hatzikiriakos, S. G. Influence of degree of sulfation on the rheology of cellulose nanocrystal suspensions. *Rheol. Acta*, 52(8-9):741–751, 2013.

Shibaev, V. P. and Bobrovsky, A. Y. Liquid crystalline polymers: development trends and photocontrollable materials. *Russ. Chem. Rev.*, 86(11):1024–1072, 2017.

Sim, K., Lee, J., Lee, H., and Youn, H. J. Flocculation behavior of cellulose nanofibrils under different salt conditions and its impact on network strength and dewatering ability. *Cellulose*, 22(6):3689–3700, 2015.

Siqueira, G., Bras, J., and Dufresne, A. Cellulosic bionanocomposites: A review of preparation, properties and applications. *Polymers (Basel).*, 2(4):728–765, 2010.

Skjetne, P., Ross, R. F., and Klingenberg, D. J. Simulation of single fiber dynamics. *J. Chem. Phys.*, 107(6):2108–2121, 1997.

Solomon, M. J. and Spicer, P. T. Microstructural regimes of colloidal rod suspensions, gels, and glasses. *Soft Matter*, 6(7):1391–1400, 2010.

Sorvari, A., Saarinen, T., Haavisto, S., Salmela, J., and Seppälä, J. Flow Behavior and Flocculation of MFC Suspension Measured in Rotational Rheometer. *Annu. Trans. Nord. Rheol. Soc.*, 21:29–34, 2013.

Soszynski, R. M. and Kerekes, R. J. Elastic interlocking of nylon fibers suspended in liquid Part 2. Process of interlocking. *Nord. Pulp Pap. Res. J.*, 3(4):180–184, 1988a.

Soszynski, R. M. and Kerekes, R. J. Elastic interlocking of nylon fibers suspended in liquid Part 1. Nature of cohesion among fibers. *Nord. Pulp Pap. Res. J.*, 3(4):172–179, 1988b.

Spence, K. L., Venditti, R. A., Rojas, O. J., Habibi, Y., and Pawlak, J. J. A comparative study of energy consumption and physical properties of microfibrillated cellulose produced by different processing methods. *Cellulose*, 18(4):1097–1111, 2011.

Stenstad, P., Andresen, M., Tanem, B. S., and Stenius, P. Chemical surface modifications of microfibrillated cellulose. *Cellulose*, 15(1):35–45, 2008.

Straley, J. P. Theory of piezoelectricity in nematic liquid crystals, and of the cholesteric ordering. *Phys. Rev. A*, 14(5):1835–1841, 1976.

Sun, X., Wu, Q., Lee, S., Qing, Y., and Wu, Y. Cellulose Nanofibers as a Modifier for Rheology, Curing and Mechanical Performance of Oil Well Cement. *Sci. Rep.*, 6(1): 31654, 2016.

Sundararajakumar, R. R. and Koch, D. L. Structure and properties of sheared fiber suspensions with mechanical contacts. *J. Non-Newton Fluid*, 73(3):205–239, 1997.

Switzer, L. H. and Klingenberg, D. J. Flocculation in simulations of sheared fiber suspensions. *Int. J. Multiph. Flow*, 30(1):67–87, 2004.

Switzer, L. H. *Simulating Systems of Flexible Fibers*. PhD thesis, University of Wisconsin-Madison, 2002.

Switzer, L. H. and Klingenberg, D. J. Rheology of sheared flexible fiber suspensions via fiber-level simulations. *J. Rheol.*, 47(3):759–778, 2003.

Syverud, K. and Stenius, P. Strength and barrier properties of MFC films. *Cellulose*, 16(1): 75–85, 2009.

Tang, J., Erdener, S. E., Li, B., Fu, B., Sakadzic, S., Carp, S. A., Lee, J., and Boas, D. A. Shear-induced diffusion of red blood cells measured with dynamic light scattering-optical coherence tomography. *J. Biophotonics*, 11(2):1–10, 2018.

Tao, Y. G., Den Otter, W. K., Padding, J. T., Dhont, J. K., and Briels, W. J. Brownian dynamics simulations of the self- and collective rotational diffusion coefficients of rigid long thin rods. *J. Chem. Phys.*, 122(24):244903, 2005.

Torstensen, J., Helberg, R. M., Deng, L., Gregersen, Ø. W., and Syverud, K. PVA/nanocellulose nanocomposite membranes for CO2 separation from flue gas. *Int. J. Greenh. Gas Control*, 81(October 2018):93–102, 2019.

Trevelyan, B. J. and Mason, S. G. Particle motions in sheared suspensions. I. Rotations. *J. Colloid Sci.*, 6(4):354–367, 1951.

Turbak, A. F., Snyder, F. W., and Sandberg, K. R. Microfibrillated cellulose, a new cellulose product: properties, uses, and commercial potential. *J Poly Sc*, 37:815–827, 1983.

Ureña-Benavides, E. E., Ao, G., Davis, V. A., and Kitchens, C. L. Rheology and phase behavior of lyotropic cellulose nanocrystal suspensions. *Macromolecules*, 44(22):8990–8998, 2011.

Van Duijneveldt, J. S., Gil-Villegas, A., Jackson, G., and Allen, M. P. Simulation study of the phase behavior of a primitive model for thermotropic liquid crystals: Rodlike molecules with terminal dipoles and flexible tails. *J. Chem. Phys.*, 112(20):9092–9104, 2000.

Wang, J., Tozzi, E. J., Graham, M. D., and Klingenberg, D. J. Flipping, scooping, and spinning: Drift of rigid curved nonchiral fibers in simple shear flow. *Phys. Fluids*, 24 (12), 2012a.

Wang, J., Graham, M. D., and Klingenberg, D. J. Shear-induced diffusion in dilute curved fiber suspensions in simple shear flow. *Phys. Fluids*, 26(3), 2014.

Wang, Q. Q., Zhu, J. Y., Reiner, R. S., Verrill, S. P., Baxa, U., and McNeil, S. E. Approaching zero cellulose loss in cellulose nanocrystal (CNC) production: Recovery and characterization of cellulosic solid residues (CSR) and CNC. *Cellulose*, 19(6):2033–2047, 2012b.

Weissenberg, K. A continuum theory of rheological phenomena. *Nature*, 159(4035):310–311, 1947.

Wilson, B. T. *Engineering Improved Magnetorheological Fluids*. PhD thesis, University of Wisconsin-Madison, 2016.

Wilson, B. T. and Klingenberg, D. J. A jamming-like mechanism of yield-stress increase caused by addition of nonmagnetizable particles to magnetorheological suspensions. *J. Rheol.*, 61(4):601–611, 2017.

Wissbrun, K. F. Rheology of rod-like polymers in the liquid crystalline state. *J. Rheol.*, 25 (6):619–662, 1981.

Wittenburg, J. *Dynamics of systems of rigid bodies*. Leitfäden der angewandten Mathematik und Mechanik. Teubner, 1977.

Wu, Q., Meng, Y., Wang, S., Li, Y., Fu, S., Ma, L., and Harper, D. Rheological behavior of cellulose nanocrystal suspension : Influence of concentration and aspect ratio. *J. Appl. Polym. Sci.*, 131(40525):1–8, 2014.

Xu, X., Liu, F., Jiang, L., Zhu, J. Y., Haagenson, D., and Wiesenborn, D. P. Cellulose nanocrystals vs. Cellulose nanofibrils: A comparative study on their microstructures and effects as polymer reinforcing agents. *ACS Appl. Mater. Interfaces*, 5(8):2999–3009, 2013.

Xu, Y., Atrens, A. D., and Stokes, J. R. Rheology and microstructure of aqueous suspensions of nanocrystalline cellulose rods. *J. Colloid Interface Sci.*, 496:130–140, 2017.

Yamamoto, S. and Matsuoka, T. A method for dynamic simulation of rigid and flexible fibers in a flow field. *J. Chem. Phys.*, 98(1):644–650, 1993.

Yarin, A. L., Gottlieb, O., and Roisman, I. V. Chaotic rotation of triaxial ellipsoids in simple shear flow. *J. Fluid Mech.*, 340(June):83–100, 1997.

Zaluzhnyy, I. A., Kurta, R. P., Sulyanova, E. A., Gorobtsov, O. Y., Shabalin, A. G., Zozulya, A. V., Menushenkov, A. P., Sprung, M., Krówczyński, A., Górecka, E., Ostrovskii, B. I., and Vartanyants, I. A. Structural studies of the bond-orientational order and hexatic-smectic transition in liquid crystals of various compositions. *Soft Matter*, 13(17):3240–3252, 2017.

Zauscher, S. and Klingenberg, D. J. Friction between cellulose surfaces measured with colloidal probe microscopy. *Colloids Surfaces A Physicochem. Eng. Asp.*, 178(1-3):213–229, 2001.

Zhang, L., Batchelor, W., Varanasi, S., Tsuzuki, T., and Wang, X. Effect of cellulose nanofiber dimensions on sheet forming through filtration. *Cellulose*, 19(2):561–574, 2012.

Zhu, J. Y., Chen, L., and Gleisner, R. Integrating the production of carboxylated cellulose nanofibrils and cellulose nanocrystals using recyclable organic acids, 2018.

Zimmermann, T., Bordeanu, N., and Strub, E. Properties of nanofibrillated cellulose from different raw materials and its reinforcement potential. *Carbohydr. Polym.*, 79(4):1086–1093, 2010.

Zwanzig, R. *Brownian motion and Langevin equations*. 2001.