Modelling the Dynamics of Actomyosin Networks

Dissertation

for the award of the degree
"Doctor rerum naturalium"
of the Georg-August-Universität Göttingen

within the doctoral program
Physics of Biological and Complex Systems
of the Georg-August University School of Science (GAUSS)

---

submitted by

Ilyas Patrick Kuhlemann

from
Steinheim, Germany

Göttingen – October 12, 2022

Thesis Advisory Committee

Prof. Dr. Burkhard Geil (1st referee)
Institute of Physical Chemistry
Georg-August-Universität Göttingen

Prof. Dr. Bert de Groot
Computational Biomolecular Dynamics Group
Max Planck Institute for Biophysical Chemistry

Prof. Dr. Sarah Köster (2nd referee)
Institute for X-Ray Physics
Georg-August-Universität Göttingen

Further Members of the Examination Board

Prof. Dr. Timo Betz
Third Institute of Physics - Biophysics
Georg-August-Universität Göttingen

Dr. Claus Heussinger
Institute for theoretical Physics
Georg-August-Universität Göttingen

Prof. Dr. Marcus Müller
Institute for theoretical Physics
Georg-August-Universität Göttingen

Date of the Oral Exam: January 14, 2022

Contents

Chapter 1

Introduction

The cytoskeleton is the major determinant of the cell's mechanical properties and its shape [1, 2]. The cytoskeleton is composed of three kinds of large filaments – microtubules, intermediate filaments and (filamentous) actin – plus a vast number of different proteins that interact with these filaments [2]. Especially actin has been identified to be a prominent factor in cell motility, cell division, and in adaptation, changes and conservation of the cell shape [1, 3–7] (a microscopy image of actin in a cell is shown in figure 1.1). As such, actin is the subject of many studies with the aim to gain more insights into its mechanical properties, and the interplay between actin and the vast family of actin binding proteins [1, 8].

Increasingly, experimental studies are being complemented by computer simulations of actin and actin networks [9–16]. For example, simulations aided in better understanding the effect of myosin motors (proteins that can bind to and actively exert forces onto actin filaments) on filament orientation [17], the influence of cross-links on viscoelastic properties of actin networks [12, 18], and how the interplay of cross-links and myosin motors impact actin filament bundles [19]. The majority of simulation studies uses coarse-grained particle simulations to replicate the dynamics of actin filaments and networks [10–12, 14–17]. In contrast to (molecular dynamic) simulations that explicitly contain coordinates of all atoms of simulated proteins, in coarse-grained simulations one particle represents tens to thousands of atoms. Many of these actin simulation studies employ frameworks like cytosim [11], MEDYAN [14] or AFINES [15]. The means of distributing open-source software are fairly standardized[1], and simulation softwares can nowadays be installed and run on widely available personal computers. Thus, versatile simulation frameworks have become research tools that can easily be shared, used and improved across many departments and laboratories. Furthermore, public availability of simulation code is beneficial for the reproducibility of results.

Since actin in its natural environment in living cells is only one out of a magnitude of components with complex biochemical interactions, actin is often studied *in vitro* in isolated model systems (see example in figure 1.1). These model systems include motility assays [20, 21], reconstituted homogeneous networks

[1]Commonly available on collaboration platforms like github.com, gitlab.com, or via programming-language specific package managers.

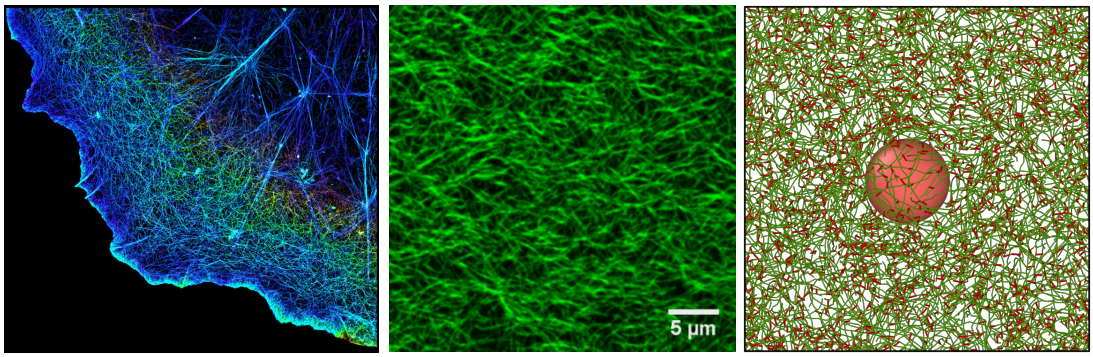

Figure 1.1: Actin is studied in different systems: *in vivo* in living cells (left), *in vitro* in reconstituted networks (center) and *in silico* in simulations (right). Image credits: (Left) Xiaowei Zhuang and National Insistutes of Health (NIH), licensed with CC BY-NC-SA 2.0. (Center) Image kindly provided by Helen Nöding.

[22–24], cortex-like networks attached to membranes [25, 26] and networks confined in droplets [27, 28]. To probe mechanical and viscoelastic properties of actin networks in these model systems, some common techniques are rheometry [22, 23], microrheology (rheological measurements performed with microspheres embedded in the networks) [26], atomic force microscopy [25], and active and passive measurements with beads in optical [24] and magnetic traps [29].
While the aforementioned simulation frameworks can freely be configured to replicate different systems of actin filaments, networks and binding proteins, they lack this flexibility in regard to "external" (non-actin) components and externally imposed potentials. Flexibility in this domain, however, would be required to replicate commonly used actin model systems and measurement techniques that include objects like microspheres (used in microrheology and optical/magnetical traps) or cantilevers of atomic force microscopes.
Further pre-requisites to understand actin network dynamics, are to reach relevant time and length scales in simulations. As a guideline, figure 1.2 depicts scales that would be sufficient for microrheology (MR) simulations. The highlighted time span arises from the upper limit of lag times that were used in MR experiments [26, 30, 31]. The highlighted length scale in figure 1.2 covers reported lengths of *in vitro* actin filaments [32–35]. Furthermore, figure 1.2 shows lengths and simulation times that were reached in simulations of three-dimensional (3D) actin networks with the cytosim and MEDYAN frameworks (AFINES is not shown here since it lacks the ability to simulate networks in 3D). MEDYAN is designed to mimic actin and actin binding protein dynamics in great detail on short length scales. No applications with length scales sufficient for the highlighted ranges in figure 1.2 were reported. The cytosim framework, on the other hand, was used for applications that fall into the highlighted ranges. When tested for MR simulations, however, the performance drastically decreased due to the relatively large size of the microsphere and simulations over desired length and time scales were not feasible (discussed in section 6.4).

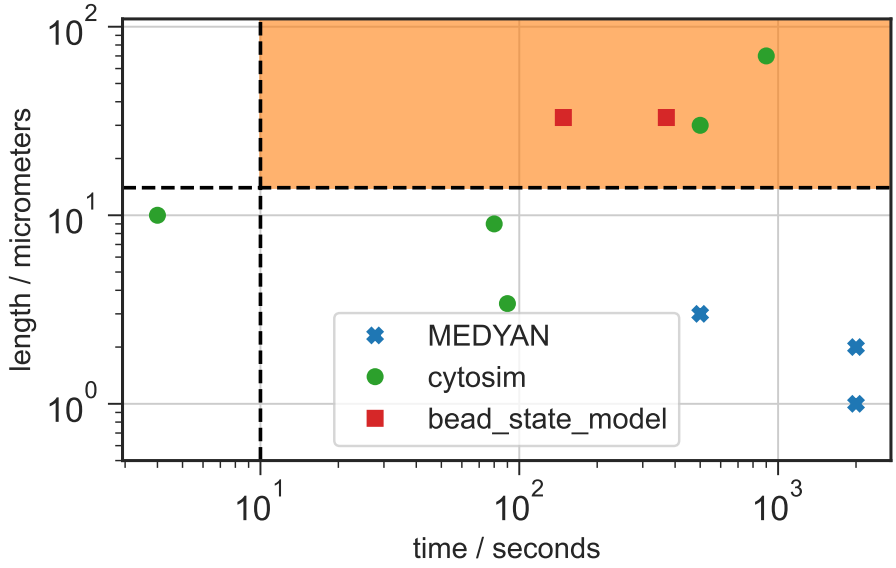

Figure 1.2: Time and length scales in 3D simulations performed with different frameworks. Data points were gathered from simulation box edge length (or equivalent) and total simulated time of various studies (for MEDYAN [14, 36] and cytosim [17, 37–40]) and the simulations in chapter 6 (for `bead_state_model`). The highlighted area indicates lag times in MR studies [26, 30, 31] and reported lengths of *in vitro* actin filaments [32–35].

To overcome these gaps in regard to flexible configuration of external components and coverage of relevant time and length scales, I developed the simulation framework `bead_state_model`. Its technical details, examplary applications and its capabilities and limits are the subject of this thesis. One core concept in order to achieve the performance to simulate relevant length and time scales, is that cross-linking proteins and motor proteins are not explicitly included in the simulations. Rather, they are included as states of the beads of the bead-chains that represent actin polymers, hence the name `bead_state_model`.
In chapters 2 and 3, the theories and methods underlying the simulations and analyses of the example applications are specified. In chapter 4 the `bead_state_model` implementation and the model on which it is based are presented. Chapter 5 analyzes and discusses results of simulated isolated filaments, to confirm filaments exhibit characteristics of the worm-like chain model. To demonstrate the framework's capabilities to perform MR simulations, in chapter 6 I compare MR data from simulations to expectations from theory as well as results from experiments (time and length scales of MR simulations are shown in figure 1.2). Chapter 7 deals with simulations where a microsphere is used to deform a dense actin network layer, a setup that was inspired by indentation experiments with atomic force microscopes. The relaxation after deformation is analyzed for different networks. Good availability and reproducibility of all data presented in chapters 5, 6 and 7 in accordance with good scientific practice were of prime concern. Details for different data sets can be found in section A1 in the appendix.

Chapter 2

Theory

2.1 Overdamped Langevin Dynamics

The equation governing the diffusive motion of particles in this thesis is the Langevin equation,

$$m\ddot{\mathbf{x}} = -\nabla E - \gamma\dot{\mathbf{x}} + \sqrt{2\gamma kT}\mathbf{y}, \tag{2.1}$$

with position of the particle $\mathbf{x}$, drag coefficient $\gamma = kT/D$, Boltzmann constant k, temperature T, diffusion coefficient D, and energy gradient ∇E. The noise term $\mathbf{y}$ is a Wiener process, where increments are independent,

$$\langle \mathbf{y}(t)\mathbf{y}^{\intercal}(t')\rangle = \mathbf{I}\delta(t-t'),$$

and $\mathbf{y}$ has a Gaussian probability distribution. $\mathbf{I}$ is the identity matrix.
We study systems at time scales where

$$t \gg \frac{m}{\gamma} = \frac{mD}{kT},$$

and neglect inertia in the Langevin equation [15, 41, 42], $m\ddot{\mathbf{x}} = 0$, leading to the overdamped Langevin equation:

$$\begin{aligned}\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} &= -\frac{1}{\gamma}\nabla E + \sqrt{\frac{2kT}{\gamma}}\mathbf{y}, \\ &= -\frac{D}{kT}\nabla E + \sqrt{2D}\mathbf{y}.\end{aligned} \tag{2.2}$$

Neglecting inertia is quite common in simulations of biopolymers [12, 14, 15, 19, 43].

2.1.1 Nondimensionalization

To derive a dimensionless form of equation 2.2, we introduce the dimensionless variables

$$\begin{aligned}\boldsymbol{\xi} &= \mathbf{x}/x_0, \\ \tau &= t/t_0, \text{ and} \\ \epsilon &= E/kT.\end{aligned}$$

The dimensionless differential operator becomes $\nabla' = \nabla \cdot x_0$. Inserting the dimensionless variables in equation 2.2 yields

$$\frac{x_0}{t_0}\frac{\mathrm{d}\boldsymbol{\xi}}{\mathrm{d}\tau} = -\frac{D}{kT}\frac{\nabla'}{x_0}(kT\epsilon) + \sqrt{2D}\mathbf{y},$$
$$\Leftrightarrow \frac{\mathrm{d}\boldsymbol{\xi}}{\mathrm{d}\tau} = -D\frac{t_0}{x_0^2}\nabla'\epsilon + \sqrt{2D\frac{t_0}{x_0^2}}\mathbf{y}.$$

In the last step, multiplication with t_0 leads to a factor $\sqrt{t_0}$ for the Wiener term. This is due to a fundamental property of the Wiener process,

$$\langle \Delta y_i^2 \rangle = \Delta t,$$

where $\Delta y_i = y_i(t + \Delta t) - y_i(t)$, for any component y_i of $\mathbf{y}$. Choosing $t_0 = x_0^2/D$ leads to

$$\frac{\mathrm{d}\boldsymbol{\xi}}{\mathrm{d}\tau} = -\nabla'\epsilon + \sqrt{2}\mathbf{y}, \tag{2.3}$$

a dimensionless form of equation 2.2.

2.2 Polymer Theory

In this section, fundamental polymer models are shortly summarized. These models were studied with two recognized text books in this field [44, 45], and the formulas, deduction and the choice of terms were in large parts directly inspired by these books.

2.2.1 Ideal Chains

Important features and properties in the study of polymers are best explained with ideal chains. Ideal chains are a class of mathematical models describing linear chains consisting of jointed segments, where it is assumed that segments that are far apart along the chain do not interact with each other [45]. The segments' endpoints can also be viewed as beads, and the chains consisting of linearly connected beads.

Freely Jointed Chain

A freely jointed chain (FJC) consists of N segments of identical length b_0 [45]. With the positions of the joints $\mathbf{R}_n$, $n \in 0, 1, \ldots, N$, and the segment vectors $\mathbf{r_i} = \mathbf{R}_i - \mathbf{R}_{i-1}$, $i \in 1, \ldots, N$, we can define the end-to-end vector $\mathbf{R}$:

$$\mathbf{R} = \mathbf{R}_N - \mathbf{R}_0 = \sum_{i=1}^{N} \mathbf{r}_i. \tag{2.4}$$

An illustration is shown in figure 2.1.

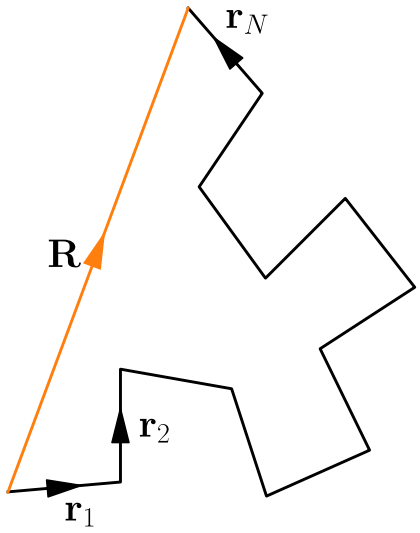

Figure 2.1: Sketch of a freely jointed chain (FJC), consisting of N segments. Adding all segment vectors $\mathbf{r}_i$ to $\mathbf{r}_N$ leads to the end-to-end vector $\mathbf{R}$.

There are no constraints or forces acting on the joints, hence the angular correlation is zero when averaging over all configurations,

$$\langle \mathbf{r}_i \mathbf{r}_j \rangle = \langle \|\mathbf{r}_i\| \|\mathbf{r}_j\| \cos\theta_{ij} \rangle = 0,$$

for any pairs $i \neq j$. The same is true for averages of the end-to-end vector,

$$\langle \mathbf{R} \rangle = 0.$$

A more sensible statistical property for chains is the (non-zero) average end-to-end distance,

$$\overline{R} = \sqrt{\langle \mathbf{R}^2 \rangle}.$$

For the FJC, the average end-to-end distance can be computed as follows [45]:

$$\begin{aligned}
\langle \mathbf{R}^2 \rangle &= \sum_{i=1}^{N} \sum_{j=1}^{N} \langle \mathbf{r}_i \mathbf{r}_j \rangle \\
&= \sum_{i=1}^{N} \langle \underbrace{\mathbf{r}_n^2}_{b_0^2} \rangle + 2 \sum_{i>j} \underbrace{\langle \mathbf{r}_i \mathbf{r}_j \rangle}_{=0} \\
&= N \cdot b_0^2 \\
\Rightarrow \overline{R} &= \sqrt{\langle \mathbf{R}^2 \rangle} = \sqrt{N} b_0
\end{aligned}$$

The maximum end-to-end vector $R_{\max}$ a FJC can assume is its contour length. The contour length l_c is the sum of all segment lengths,

$$l_c = \sum_{i=1}^{N} ||r_i|| \overset{\text{FJC}}{=} \sum_{i=1}^{N} b_0 = Nb_0. \tag{2.5}$$

Equivalent Freely Jointed Chain

Many real polymers can not be described by the FJC model, since no free rotation of the joints between monomers or identical length of segments can be assumed. However, when describing *very long* polymers, their properties can often accurately be captured by an equivalent freely jointed chain. Here, long means that local structural properties can be neglected, and many monomers be represented by a segment of length b of a FJC, where b is the so called Kuhn length [45]. The equivalent FJC then consists of N segments of length b, where N and b are chosen such that the contour length $R_{\max} = N \cdot b$ and mean-square end-to-end distance $\langle \mathbf{R}^2 \rangle = N \cdot b^2$ are identical to $R_{\max}$ and $\langle \mathbf{R}^2 \rangle$ of the real polymer [45].

Freely Rotating Chain

A freely rotating chain (FRC) is defined like the FJC, except that the angle between consecutive segments has a fixed value θ,

$$\mathbf{r}_i \mathbf{r}_{i+1} = b_0^2 \cos\theta.$$

As a consequence, the average $\langle \mathbf{r}_i \mathbf{r}_j \rangle$ does not vanish. It rather decreases rapidly with $i - j$ (assuming $i > j$):

$$\begin{aligned} \langle \mathbf{r}_i \mathbf{r}_j \rangle &= \cos\theta \langle \mathbf{r}_{i-1} \mathbf{r}_j \rangle \\ &= (\cos\theta)^2 \langle \mathbf{r}_{i-2} \mathbf{r}_j \rangle \\ &= \ \ldots \\ &= b_0^2 (\cos\theta)^{i-j} \end{aligned}$$

Applying a logarithmic and exponential function on $(\cos\theta)^{i-j}$,

$$(\cos\theta)^{i-j} = e^{(i-j)\ln(\cos\theta)},$$

and introducing the definition of the persistence length [45],

$$l_p := -\frac{b_0}{\ln\cos\theta},$$

leads to the following exponential form:

$$\frac{\langle \mathbf{r}_i \mathbf{r}_j \rangle}{b_0^2} = (\cos\theta)^{i-j} = \exp\left(-\frac{(i-j)b_0}{l_p}\right) \tag{2.6}$$

2.2.2 Worm-like Chain

The worm-like chain (WLC) is a limit case of the FJC, with

$N \to \infty$, infinitely many segments,

$b_0 \to 0$, infinitely small segments, and

$\theta \to 0$, infinitely small angles,

while keeping persistence length and contour length constant. The WLC model is often used for semi-flexible polymers (polymers with $l_c \approx l_p$) like actin (see section 2.3).

Taking the limits of the WLC into account, we transition the discrete description of the polymer to a continous description, and change the sum over segments into integration along the contour s of the polymer:

$$\begin{aligned} \sum_{i=1}^{N} b_0 &\to \int_0^{l_c} \mathrm{d}s, \\ \mathbf{r}_i &\to \mathbf{r}(s), \\ \mathbf{r}_i / b_0 &\to \mathbf{t}(s) \text{ (unit tangent vector).} \end{aligned}$$

Equation 2.6 in its continous form can be interpreted as the exponentially decaying correlation between tangent vectors along the contour (see figure 2.2):

$$\langle \mathbf{t}(s)\mathbf{t}(s+\Delta s)\rangle = \exp\left(-\frac{\Delta s}{l_p}\right). \tag{2.7}$$

The average end-to-end distance $\overline{R}$ of WLCs depends on its persistence length

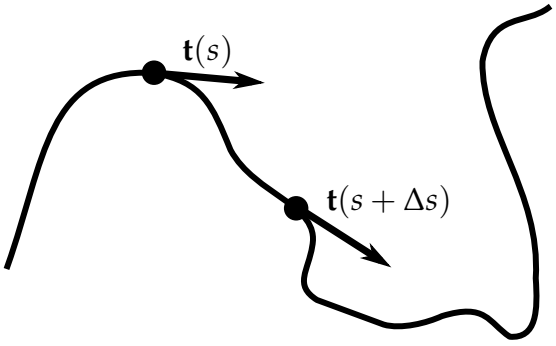

Figure 2.2: Two tangents with distance Δs along the contour of a filament.

l_p and contour length l_c [45]:

$$\overline{R}^2 = \langle \mathbf{R}^2 \rangle = 2 l_p l_c - 2 l_p^2 \left(1 - \exp\left(-\frac{l_c}{l_p}\right)\right). \tag{2.8}$$

2.2.3 Rouse Model

The Rouse model is one of the simplest models and was one of the first to successfully describe dynamics of bead chains [45, 46]. In this thesis, the model's predictions regarding the mean squared displacement (MSD) of Rouse chains serves as a guideline of what MSDs we expect for simulated polymers.
The polymer in the Rouse model is a a bead-spring chain with mean segment length [45]

$$\langle ||r_i|| \rangle = b.$$

For a bead chain of N monomers with monomer friction coefficient ζ the total friction coefficient of the chain is $\zeta_R = N \cdot \zeta$ [45]. Using the Einstein relation, the diffusion coefficient of the Rouse chain can be computed [45]:

$$D_R = \frac{kT}{\zeta_R} = \frac{kT}{N\zeta}. \tag{2.9}$$

A chain with N beads diffuses a distance on the order of its size during the so called Rouse time [45],

$$\tau_R \approx \frac{\zeta}{kT} N R^2, \tag{2.10}$$

where R the radius of individual beads. The Rouse time can be expressed in terms of the time τ_0 it would take a free bead to travel the distance of its own radius [45]:

$$\begin{aligned} \tau_0 &= \frac{\zeta R^2}{kT} \\ \Rightarrow \tau_R &= \tau_0 N^2. \end{aligned}$$

For the MSD of a bead embedded in the chain we expect 3 distinct regimes, as depicted in a hypothetical curve in figure 2.3:

I The high frequency regime, where we see almost unhindered free diffusion of beads.

II The intermdediate frequency regime, where we see subdiffusive motion.

III The low frequency regime, where we see the motion of the chain as if it were one particle diffusing freely.

The transitions are roughly at τ_0 and τ_R, the slopes

$$p = \frac{\mathrm{d} \log MSD}{\mathrm{d} \log \tau} \tag{2.11}$$

(i.e. the exponents of the relation $MSD \sim t^p$) should be very close to, but never exceeding, unity for phases I and III. An estimate for p via relaxation modes [45] puts the slope in phase II at $p_{\mathrm{II}} = 1/2$ for ideal chains.

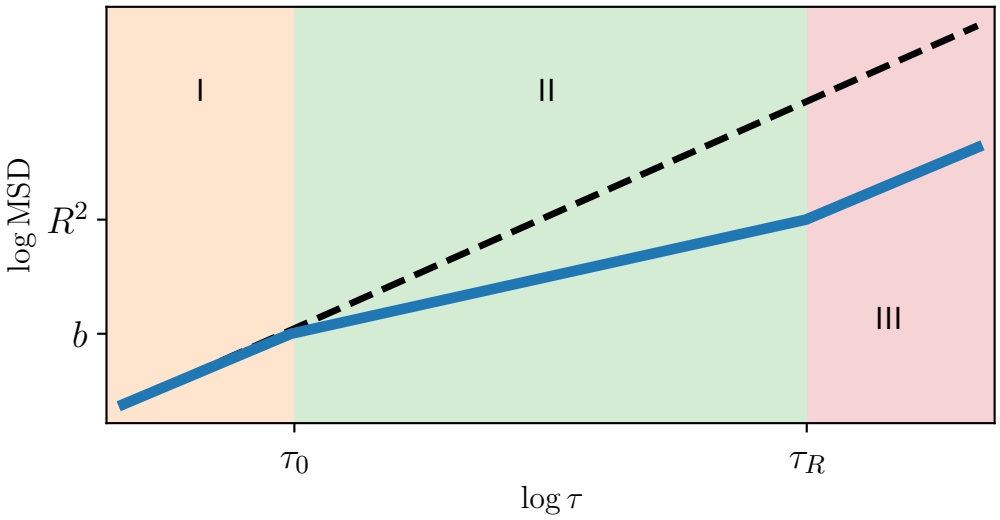

Figure 2.3: Hypothetical MSD of bead embedded in an ideal chain on double logarithmic scales, according to the Rouse model [45]. The black dashed line with slope 1 (on logarithmic scales) is the MSD of a bead that is not part of a chain and diffuses freely. At low lag times $\tau < \tau_0$ (I) and at large lag times $\tau > \tau_R$ (III), the slope is nearly 1. At intermediate lag times $\tau_0 \leq \tau \leq \tau_R$, the slope is $1/2$.

2.2.4 Entangled Networks

In solutions where the concentration of polymers is sufficiently large, polymers interact with each other and form entangled networks [8, 45]. A parameter to characterize the typical spacing between polymers is the mesh size ξ.
To estimate the order of the mesh size, we can use the following approximation by Broedersz and MacKintosh [8]:

$$\xi \approx r_{\text{polymer}} / \sqrt{\phi}, \tag{2.12}$$

where r_{polymer} is the radius of the polymers and ϕ is the volume fraction occupied by the polymers.
In polymer solutions where polymers form entangled networks, the networks usually dominate the properties of the solution (even if most of the volume is occupied by the solvent) [45]. These polymer solutions show viscoelastic behaviour. I.e. their response to mechanical stress and strain is a combination of viscous flow and elastic response [45].

2.2.5 Reptation

In order to quantitatively describe the motion of polymers in entangled networks, de Gennes developed the reptation model [47]. In the model's ansatz de Gennes focuses on the motion of a single polymer that moves in a three-dimensional network, e.g. a polymeric gel [47]. A depiction is shown in figure 2.4.

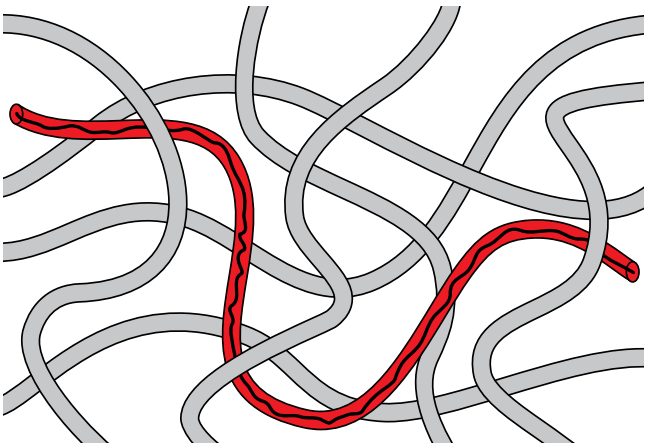

Figure 2.4: The reptation model describes the motion of a single polymer (black line) through a three-dimensional network. The network can be a polymeric gel, as depicted here, but could also be any other set of obstacles that confines the polymer to a tube (shown in red) [47]. The image was published by Carl Handerson under CC0 1.0 license.

In the reptation model, the polymer is confined to a tube by the surrounding network. The polymer can travel along the tube through defects that propagate along the polymer [47].
In the short overview of key formulas and characteristica of the reptation model in this section follows the notation of and line of argumentation of Rubinstein and Colby [45]. The tube diameter a can be estimated via ideal chain statistics to be [45]

$$a \approx b\sqrt{N_e},$$

where N_e is the number of monomers in a so called entanglement strand. The tube can be treated as consisting of N/N_e segments of length a, each containing N_e monomers. This yields the average length $\langle L \rangle$ of the tube [45]:

$$\langle L \rangle \approx a\frac{N}{N_e}.$$

Assuming the Rouse diffusion coefficient D_R (equation 2.9), one can estimate the reptation time τ_{rep}, which is the time it takes for the chain to diffuse out of the tube of length $\langle L \rangle$:

$$\tau_{\text{rep}} \approx \frac{\langle L \rangle^2}{D_R} \approx \frac{\zeta b^2}{kT}\frac{N^3}{N_e} = \frac{\zeta b^2}{kT}N_e^2\left(\frac{N}{N_e}\right)^3.$$

The first part of the equation is the Rouse time (equation 2.10) of an entanglement strand with N_e monomers,

$$\tau_e := \frac{\zeta b^2}{kT}N_e^2.$$

Looking at the mean-squared displacements of the chain monomers due to defects travelling along the chain, de Gennes derived the relation [45, 47]

$$MSD \sim t^{1/4}.$$

This relation is valid in the range $\tau_e < t < \tau_R$. On rather short time scales $\tau_0 < t < \tau_e$ and rather long time scales $\tau_R < t < \tau_{\text{rep}}$, the chain behaves approximately as a free Rouse chain, leading to $MSD \sim t^{1/2}$ [45].

2.3 Actin

Actin is a globular protein (G-actin) that can polymerize into a helical filament (F-actin) with a diameter of about 8 nm and has been found to span lengths of around 14 µm to 30 µm *in vitro* [1, 32, 35, 48]. Actin forms one of the cytoskeletal filaments and plays an essential role for cell motility, a cell's response to mechanical stress and strain, and processes where the shape of the cell is actively altered (like the formation of filopodia or during cytokinesis [5, 7]).
Assembly of monomers into polymers is thermodynamically unfavorable for the first two and three monomers [1]. Once this so-called nucleation is overcome, however, further elongation is energetically favorable, and polymers can rapidly grow to lengths of multiple micrometers [1].
In solutions of polymerized actin, depolymerization and polymerization lead to a steady-state concentration of G-actin monomers [49]. In this steady-state, F-actin exhibits treadmilling, a process where the filament polymerizes at one end and depolymerizes at the other [49]. These ends are referred to as the plus (or the barbed) end and the minus (or pointed) end, respectively. As a consequence of the treadmilling, the filament effectively moves into the direction of the plus end.
These processes can be influenced by many different regulatory proteins [14, 49]. For example, formin is considered a nucleator that can "assist" the nucleation process [14, 49]. Another nucleator is the Arp2/3 protein complex that binds to F-actin and can also trigger nucleation of a new actin filament[14, 36, 49, 50]. This can lead to branched networks of relatively short actin filaments [36, 50].
Under the incfluence of the complex interplay of many regulatory proteins, networks of various different architectures can form [1]. In cells, the architectures of actin networks vary fundamentally across different locations [1]. Figure 2.5 shows a overview sketch of actin in a migrating cell. At the leading edge (towards the bottom in figure 2.5), actin in the lamellipodium is involved in pushing the membrane into the direction of motion by polymerizing against the membrane [4, 50]. From the leading edge, the cell extends finger-like filopodia, which are stabilized by parallel (orientation of plus and minus ends align across filaments) bundles of actin filaments [1]. Parallel to the direction of motion, cell-spanning stressfibers form, consisting of anti-parallel (orientation of plus and minus ends is inverted for neighbors in a bundle) bundles [1]. Close to the membrane, actin forms the cellular cortex, which is anchored to the membrane, for example with the help of the protein ezrin [1].
Some components that were found to be essential in shaping the differently structured networks are mentioned in the boxes in figure 2.5. Among these components the proteins are α-actinin, filamin, fimbrin and fascin which belong to the class of cross-linking proteins (cross-links). Cross-links can attach to

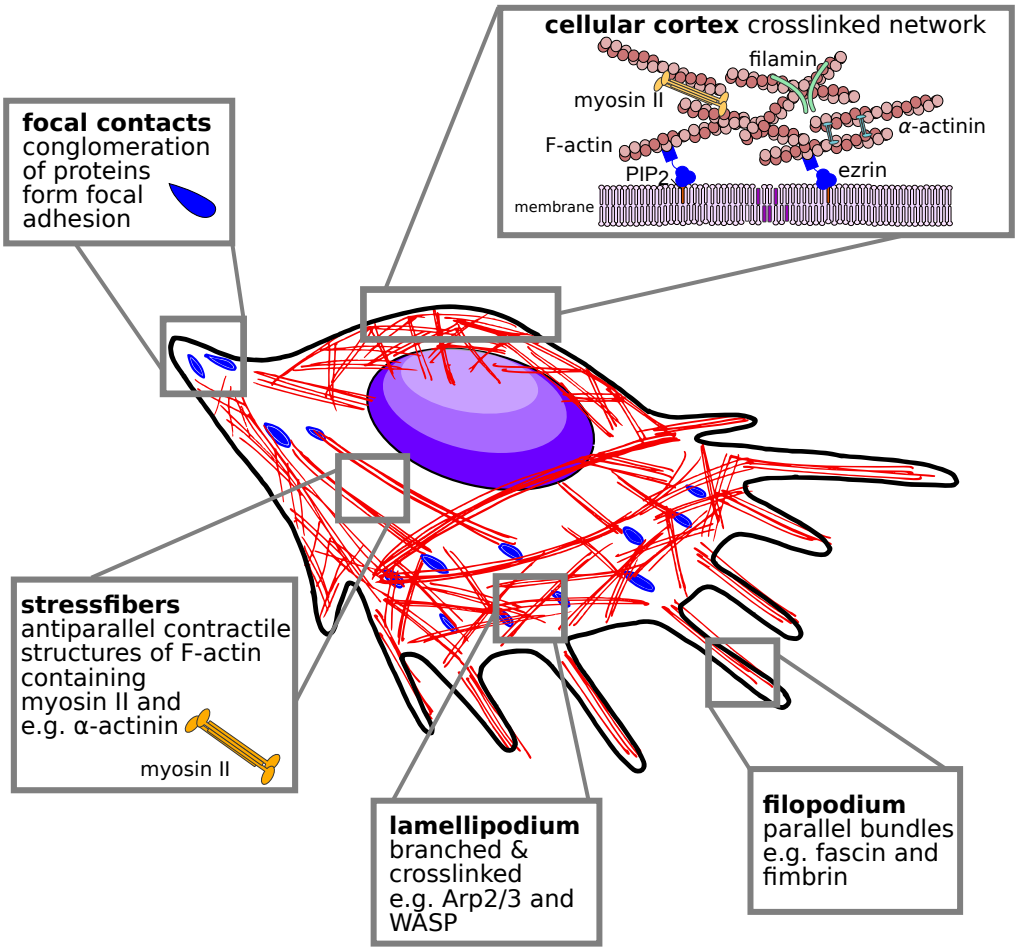

Figure 2.5: Sketch of a migrating cell. Actin cytoskeleton is shown in red, and composition of actin networks at different locations depicted or briefly described. Figure was adapted from [35].

filaments and form a link between two filaments. They are key components for large stable structures like stressfibers and the parallel bundles in filopodia [1, 4]. The distance and the angle at which they bind two filaments together varies across cross-links, which can give rise to different network architectures [1].
One important family of proteins that is essential for the active processes actin is involved in are myosin motors [51]. Like cross-links, myosin motors can attach to filaments and can form a link between two filaments. In addition, they can actively exert forces on the filaments by tugging on them [8]. When active myosin is present, the networks are sometimes referred to as actomyosin networks.
The tugging of motors on filaments has been shown to siginificantly affect mechanical and visoelastic properties of networks [23]. In cross-linked anti-parallel bundles (like in stressfibers and actin rings during cytokinesis), the presence of myosin motors is required for contractility of these bundles [38, 51].

2.4 Microrheology

A common property to characterize the response of a viscoelastic material (like entangled polymer networks) to mechanical strain is the stress relaxation modulus $G(t)$. For stress induced to a viscoelastic material through a step strain at $t = 0$, $G(t)$ describes the stress relaxation of the material [45].
When stress response is studied through rheometry, where oscillatory shear is applied to a viscoelastic material [45, 52–54], the fourier transform $G^*(\omega)$ of $G(t)$ is commonly used [55] rather than $G(t)$ itself. The real and the imaginary parts of $G^*(\omega)$ are referred to as the storage modulus $G'(\omega)$ and loss modulus $G''(\omega)$ [45]. The stress $\sigma(t)$ oscillates with the same frequency ω as the applied strain $\gamma(t)$, but is offset by a phase angle $\delta \in [0, \pi/2]$ [45],

$$\begin{aligned}\sigma(t) &= \sigma_0 \sin(\omega t + \delta) \\ &= \gamma_0 \left[G'(\omega) \sin(\omega t) + G''(\omega) \cos(\omega t) \right].\end{aligned}$$

Since $\sigma(t)$ is comprised only of the in-phase part (G') for solids and only of the out-of-phase part (G'') for liquids, the two components G' and G'' are often associated with the elastic and the viscous parts (respectively) of a viscoelastic material.

To infer viscoelastic properties of materials via the motion of microspheres embedded in the material, Mason and Weitz [56] established a method for which the term microrheology was later coined [55, 57]. Mason and Weitz retrieved mean-squared displacements (MSDs, see section 3.2) of passive microspheres in complex fluids comprised of polymers through diffusing wave spectroscopy. The complex shear moduli G^* computed from the MSDs were validated by comparing them to G^* retrieved by established mechanical rheometry methods. Soon after that, this passive microrheology method was applied to solutions of biopolymers (e.g. DNA [58] and Actin [59, 60]), and was also extended to trajectories of microspheres recorded via particle tracking [58].
To relate the MSD to G^*, Mason and Weitz [56] derived a generalized form of the Stokes-Einstein relation in the Laplace domain,

$$\widetilde{G}(s) = \frac{kT}{\pi a s \langle \Delta \widetilde{r}^2(s) \rangle}, \tag{2.13}$$

where s is the frequency in the Laplace domain, a is the radius of the microsphere, and $\langle \Delta \widetilde{r}^2(s) \rangle$ is the Laplace transform of the MSD. To obtain $G^*(\omega)$, $\widetilde{G}(s)$ was fitted with an empirical model and s substituted by $i\omega$.
In later additions to the method [61], $G^*(\omega)$ was directly computed via the Fourier transform of the MSD, $\mathcal{F}\left\{\langle \Delta r^2(t) \rangle\right\}$, and the identity

$$G^*(\omega) = \frac{kT}{\pi a i \omega \mathcal{F}\left\{\langle \Delta r^2(t) \rangle\right\}}.$$

Chapter 3

Methods

3.1 Molecular Dynamics

In molecular dynamics (MD), trajectories of particles (representing atoms/molecules) are simulated by numerically integrating the particles' equation of motion (EoM). A classical EoM would for example be Newton's second law,

$$F = m\ddot{x}. \tag{3.1}$$

To illustrate key concepts of numerical integration of an EoM, a one-dimensional harmonic oscillator will be used here:

$$F = -k \cdot x.$$

With velocity v, we split the harmonic oscillator's EoM into two first-order differential equations,

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t} &= v, \\ \frac{\mathrm{d}v}{\mathrm{d}t} &= -\frac{k}{m}x. \end{aligned}$$

In particle simulations, we compute time-discrete trajectories $x(t_i) = x_i$, usually with fixed time step size $\Delta t = t_{i+1} - t_i$. The discrete EoM can be written as

$$\begin{aligned} \frac{\Delta x_i}{\Delta t} &= v_i, \\ \frac{\Delta v_i}{\Delta t} &= -\frac{k}{m}x_i. \end{aligned}$$

To compute the oscillator's position x_{i+1} and velocity v_{i+1} at the next point in time, numerical integration schemes are used to integrate the discrete EoM. In this example the first-order Euler integration scheme will be used. Applying it to the discrete EoM yields

$$x_{i+1} = x_i + \Delta x_i = x_i + \Delta t \cdot v_i, \tag{3.2}$$

$$v_{i+1} = v_i + \Delta v_i = v_i - \Delta t \frac{k}{m} x_i. \tag{3.3}$$

The numerical integration leads to deviations in the trajectory compared to analytical solutions. These errors are influenced by the choice of Δt and the integration scheme, which is the reason why often small time steps and higher-order, more expensive integration schemes are employed. In the harmonic oscillator's example, we know the analytical solutions and can compare the deviations due to numerical integration. Example trajectories with different time steps are shown in figure 3.1. The numerically integrated EoM leads to ever increasing

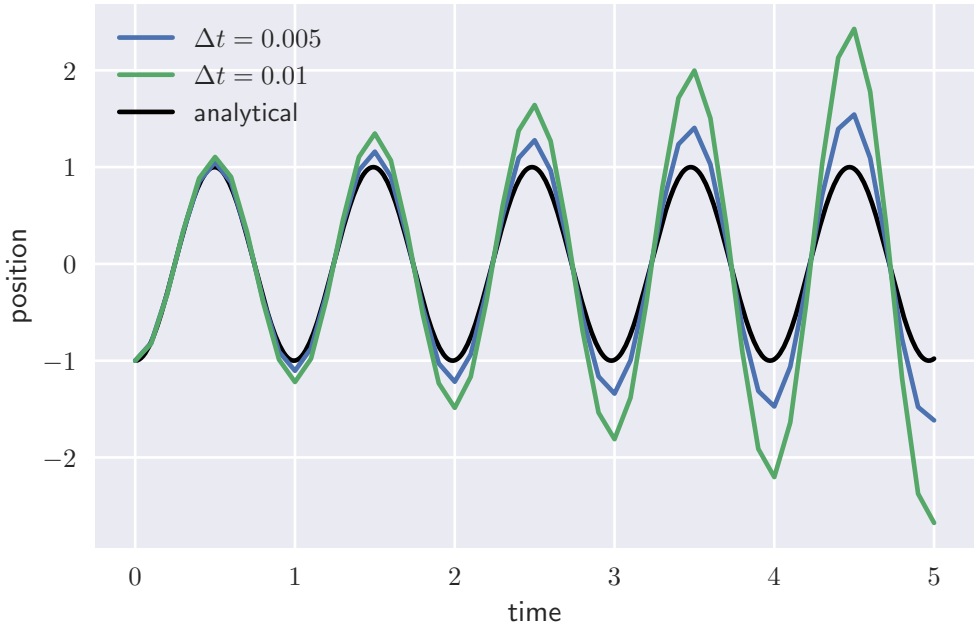

Figure 3.1: Time evolution of simulated trajectories and analytically computed trajectory of a one-dimensional harmonic oscillator. Initial conditions were set to $x(0) = -1, v(0) = 0$. Two simulations with different time steps were conducted. EoMs were integrated using the Euler integration scheme.

amplitudes. The larger the time step, the faster this deviation grows.
In the example of the 1D harmonic oscillator, the numerical integration leads to errors that lead to ever-increasing energies which destabilizes the system. In simulations of many particles, more errors due to numerics and discretization might occur, for example overlapping particles that lead to unphysically large forces, which often leads to unstable systems as well. One technique that is often employed to make systems more stable is the introduction of a thermostat to maintain a fixed average temperature [62, 63].

3.1.1 Thermodynamic Ensemble and Thermostat

When Newton's second law is used as EoM to compute the trajectories of particles, one would expect that total energy E is conserved, and that a simulated system with constant particle number N and constant volume V runs in the microcanonical NVE ensemble. However, we have seen in the example above, that energy is not conserved due to inaccuracies in the numerical integration. In

MD simulations with more particles, other factors can give rise to violation of energy conservation as well.[1] To stabilize the system (and also for other practical reasons, a list of which was given for example by Soddemann et al. [62]), the system in an MD simulation is often coupled to a heat bath via a thermostat [62, 63]. Employing a thermostat leads to constant (average) temperature T and adapts the system to the canonical NVT ensemble [62].
Thermostats are introduced as modifications to the EoM [62]. One example is the Langevin thermostat, where a dissipative term $-\gamma v$ and a stochastic force term $\sqrt{2\gamma kT}y$ are introduced, transforming the Newtonian EoM (equation 3.1) into the Langevin equation (equation 2.1) [63].

Throughout this thesis, the overdamped Langevin equation (equation 2.2) was used exclusively as EoM.

3.1.2 Brownian Dynamics

In the particle simulations presented in this thesis, as well as in many other biopolymer simulations [11, 12, 14–16, 43], the EoM is the overdamped Langevin equation (equation 2.2). Molecular dynamics simulations of this kind are often referred to as Brownian dynamics (BD) simulations. Temperature coupling to maintain a constant average temperature T is achieved via the stochastic force term of the Langevin equation.
The integration scheme used throughout this thesis is similar to the Euler integration scheme used on the ordinary differential equations (see equations 3.2 and 3.3). If at time step τ_i the position of a particle is $\boldsymbol{\xi}_i$, the position $\boldsymbol{\xi}_{i+1}$ at the next time step $\tau_{i+1} = \tau_i + \Delta\tau$ is computed as

$$\begin{aligned}\boldsymbol{\xi}_{i+1} &= \boldsymbol{\xi}_i + \Delta\boldsymbol{\xi}_i \\ &= \boldsymbol{\xi}_i + \Delta\tau\mathbf{f}_i + \sqrt{2\Delta\tau}\mathbf{y}_i,\end{aligned}$$

where $\mathbf{f}_i = -\nabla'\epsilon$, and components of vector $\mathbf{y}_i$ are drawn from a normal distribution $\mathcal{N}(0,1)$. This integration scheme is sometimes referred to as Euler integration as well [19], sometimes as the Euler-Maruyama integration scheme [64].

3.1.3 Choice of the Time Step

The choice of the time step in MD simulations is very critical. On the one hand, it is desirable to choose the time step as large as possible, such that every simulated step advances the simulated time as much as possible, and larger samples of the phase space can be achieved [65, 66]. On the other hand, the larger the time step the larger the errors due to numerical integration of the EoM, which can lead to unphysical behaviour [65–67].
There are some useful guidelines how to estimate an upper limit for the time step. E.g., Kim [66] states that the step size is typically around 0.0333 to 0.01

[1] A comprehensive list can be found in the manual of the GROMACS simulation framework, e.g. at `https://manual.gromacs.org/documentation/2019-rc1/user-guide/terminology.html#energy-conservation` (accessed 2021/11/20).

of the smallest vibrational period in the simulation. However, this and similar guidelines typically stem from empirical studies and experience [65, 66]. Hence, simulations with different time steps should be conducted in advance to long simulations to empirically estimate the largest time step suitable for the simulated system. In 1986, Fincham [65] reported such an empirical approach for simulated liquid argon, and showed that larger time steps than typically being used at that time would reliably lead to stable thermodynamic quantities. Schlick and Beard [67] argue that the BD integrators allow for much larger time steps, e.g. around 100 times larger than time steps suitable for the Verlet integrator [68]. Section 5.4 describes the empirical search for suitable time steps for the simulations presented in this thesis. As a criterion for what is "suitable", the independence of the mean-squared displacements of the fastest fluctuating particles was used.
As mentioned in 3.1.2, the first-order Euler-Maruyama integration scheme is use throughout the thesis. This is the default integration scheme for the ReaDDy simulation framework [64], on which most simulations in this thesis are based, as well as the cytosim simulation framework, which has been used for comparison in a few cases. A further increase of the time step could be achieved with a higher-order integration scheme, as demonstrated for example by Iniesta and de la Torre [69]. However, a second-order integration scheme comes with the drawback of being computationally more expensive, and an overall speed-up of simulations even with larger time step is uncertain. Furthermore, the usage of the Euler-Maruyama integration scheme is the de-facto standard for BD simulations of cytoskeletal polymers, and is employed in most of the recent publications and the related simulation frameworks [11, 12, 15].

3.1.4 Chemical Reactions

The ReaDDy simulation framework, which serves as underlying engine for the `bead_state_model` framework that was developed during this thesis (see section 4.1), allows the user to define chemical reactions alongside the mechanical interactions between particles [64]. This feature was used extensively for the implementation of the `bead_state_model` framework, and this section will briefly describe how ReaDDy handles chemical reactions. For a more detailed description the reader is referred to the publication on ReaDDy by Hoffmann et al. [64] and the ReaDDy software manual.
In ReaDDy, single particles and pairs of particles can undergo several different kinds of reactions [64]:

1. A single particle can decay or spawn: $\mathrm{A} \underset{\lambda_2}{\overset{\lambda_1}{\rightleftharpoons}} \varnothing$
2. A particle can convert into a different type: $\mathrm{A} \xrightarrow{\lambda} \mathrm{B}$
3. Pairs of particles can fuse: $\mathrm{A} + \mathrm{B} \xrightarrow{\lambda} \mathrm{C}$
4. A particle can undergo fission, turning into new single or multiple particles: $\mathrm{A} \xrightarrow{\lambda} \mathrm{B} + \mathrm{C}$

5. Pairs of particles can form a bond, turning them into a connected structure that ReaDDy calls a topology: A + B $\xrightarrow{\lambda}$ AB
6. A particle can form a bond with another particle that is part of an existing topology, hence increasing the number of particles in the topology: A + BC $\xrightarrow{\lambda}$ ABC

Associated with each reaction is a rate λ with dimensionality 1/time, defining the probability for a possible reaction at each simulation step. A reaction between pairs of particles is only considered possible if the distance between the centers of the particles is within a cut-off range r. A reaction of a single particle is not subject to any spatial restrictions, and is always considered possible. In order to determine whether the reaction occurs, the probability p of the reaction at a discrete time increment Δt is computed as [64]

$$p = 1 - e^{-\lambda \Delta t}. \tag{3.4}$$

A topology $\mathcal{T}$ can consist of any number of particles that form a connected graph, where a bond is considered to be an edge of that graph. For particles that are part of a topology, a new set of possible reactions is available. ReaDDy allows to define so called recipes that can apply different kind of changes to a topology. E.g. breaking of bonds between particles, creation of new bonds between particles, and changing of particles types of one or more particles. Any information on the topology, including the position of the topology particles, is made available to the recipe. This allows for all kinds of different conditions (number of particles of type A, number of bonds between particles of types B and C, shortest path in number of bonds between two particles, spatial distance between two particles, etc.) to determine the outcome of a reaction. The same is true for the reaction rate $\lambda(\mathcal{T})$, which is a function of the state of the topology $\mathcal{T}$, and is determined dynamically at each simulation step.
Topology reactions were used extensively for the `bead_state_model`, the framework that was developed for this thesis and is based on ReaDDy. For examples of what is possible with ReaDDy's reaction recipes, the reader is referred to chapter 4, especially the bottom half of figure 4.2 and section 4.1.2, and the ReaDDy publication by Hoffmann et al. [64].

3.1.5 Linked Cells

In simulations with N interacting particles, it can be computationally expensive to iterate over all $N \cdot (N-1)$ particle pairs, immensely impacting the time t it takes for simulations to finish. For large N, the exponent p in the run time proportionality

$$t \sim N^p \tag{3.5}$$

would be $p \approx 2$. For short-range interactions, which can be neglected for particles pairs with distance d larger than a cut-off distance d_c, a common way to cut down the number of particle pairs that a simulation algorithm needs to iterate over by introducing linked cells [70, 71]. In the algorithm of the linked cells, each particle is assigned to a cell based on its position (see figure 3.2).

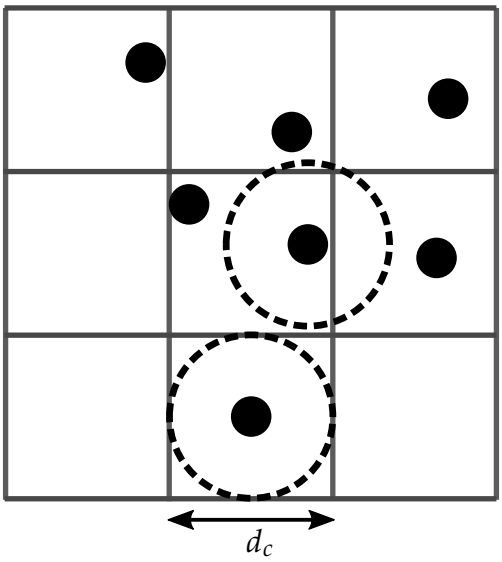

Figure 3.2: Sketch of particle assignment to linked cells based on their position. Edge length of cell is chosen equal to longest interaction range d_c (indicated as particle diameter by dashed line). Interactions only have to be considered across particles within the same cell or direct neighbor cells.

The cells' edge lengths are chosen with respect to d_c of the interaction, such that iterations over particle pairs have to be performed only for particles that are in the same cell or direct neighbor cells. Usually the edge length is set identical to d_c, which has been found to reduce the exponent p in equation 3.5 close to 1 for monodisperse systems (systems where all particles belong to the same particle species) [71].

In ReaDDy, the cut-off distance d_c is determined as the maximum across the cut-off distances of all interaction potentials and spatial reactions. An excerpt of the code responsible for that is shown in code listing L1 in section A7 in the appendix. The distance d_c can be increased by the user via the `skin` parameter, a distance that gets added to the automatically determined maximum. Since systems simulated with ReaDDy can have reactions that change locations and the number of particles, at every simulation step the list of which particle is in which box gets updated twice: once before the reactions are handled and then again, taking all changes due to reactions into account, before the forces are computed [64].

Hierarchical Grids

In case of bidisperse or polydisperse systems, with different d_c for different combinations of species, the exponent p in equation 3.5 usually becomes siginificantly larger than 1, if the cell edge length is set to the largest d_c [70, 71]. One method to efficiently decrease p in polydisperse simulations, is to use hierarchical grids. The principle of hierarchical grids is well explained by Ogarko and Luding [71]. As hierarchical grids were not used in the simulations for this thesis, they will not be described in more details here. They were mentioned here, however, due to their potential to drastically decrease run times of the

simulations in chapters 6 and 7.

3.2 Mean-squared Displacement

For a time discrete trajectory of a particle,

$$\mathbf{r}_{t_j},$$

with $j \in [1, 2, \ldots, N]$, the mean-squared displacement (MSD) with a lag time of $\tau \in [1, 2, \ldots, N-1]$ is

$$\mathrm{MSD}(\tau) = \langle (\underbrace{\mathbf{r}_{t_{j+\tau}} - \mathbf{r}_{t_j}}_{=:\Delta\mathbf{r}_{j,j+\tau}})^2 \rangle_j = \langle \Delta\mathbf{r}^2_{j,j+\tau} \rangle_j.$$

At a given value of τ, the average $\langle \cdot \rangle_j$ is computed over $N - \tau$ eligible values of

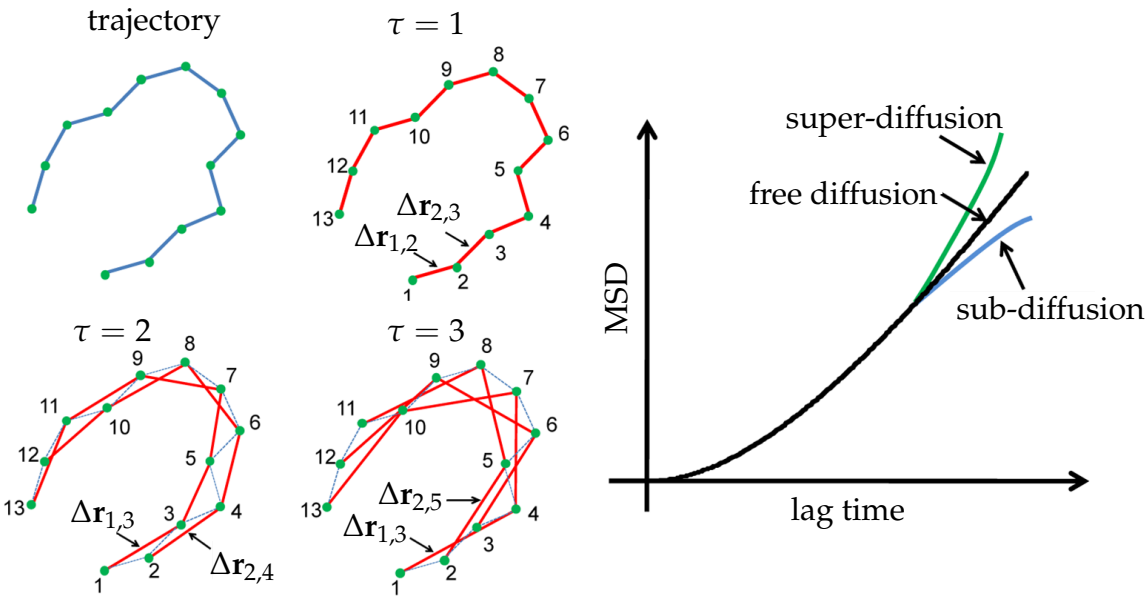

Figure 3.3: **(left)** Sketch of a particle's time-discrete trajectory, consisting of 13 positions, and difference vectors $\Delta\mathbf{r}_{j,j+\tau}$ at $\tau = 1, 2, 3$. **(right)** Hypothetical MSD of a particle, drawn with 3 different branches at higher lag times that illustrate super-diffusion, free diffusion and sub-diffusion. Figure was adapted from Wang et al. [72].

j. This is illustrated in the sketches in figure 3.3, where for an example trajectory with $N = 13$, 12 difference vectors $\Delta\mathbf{r}_{j,j+\tau}$ can be formed for $\tau = 1$, 11 for $\tau = 2$, and so on. As a consequence, at large lag times the MSD is an average over only a few values (or exactly one value in the case of $\tau = N - 1$), which can lead to large fluctuations in the MSD.
For a particle performing a three-dimensional random walk, the MSD is proportional to the lag time [45],

$$\mathrm{MSD}(\tau) = 6D\tau,$$

where D is the diffusion coefficient. A particle for which the MSD is linearly related to the lag time, $MSD(\tau) \sim \tau^p$ with $p = 1$, the particle is said to show free diffusion. For $p < 1$ or $p < 1$, the particle is said to show sub-diffusion or super-diffusion, respectively.

3.3 Computation of Complex Shear Modulus G^*

Two methods to compute the complex shear modulus G^* from microrheology data are presented in this section. The first one, the Mason method, is more commonly used [26, 55, 56, 73, 74], but difficult to implement (small differences in numerical methods can have large impact on result) and sensitive to small changes in the parameters.
The second one, the Paust method, is less widespread so far, but was found to yield similar results (see section 6.2.1).

3.3.1 Mason Method

In the context of this thesis, the Mason method refers to the extension by Dasgupta et al. [55] of the original method by Mason [61]. In the original method [61], $\widetilde{G}(s)$ (in the Laplace domain) is computed as follows:

1. The MSD is expanded around frequency $s = 1/t$ with a power law, and the leading term is retained [55]:

$$\langle \Delta r^2(t) \rangle \approx \langle \Delta r^2(1/s) \rangle (st)^{\alpha(s)}, \tag{3.6}$$

   with

$$\alpha(s = 1/t) = \frac{\mathrm{d} \ln \langle \Delta r^2(t) \rangle}{\mathrm{d} \ln t}. \tag{3.7}$$

2. The approximation in equation 3.6 is Laplace transformed and solved for $s\langle \Delta \widetilde{r}^2(s) \rangle$:

$$s\langle \Delta \widetilde{r}^2(s) \rangle = \langle \Delta r^2(1/s) \rangle \Gamma \left[1 + \alpha(s)\right],$$

   where Γ is the gamma function.

3. This is inserted into equation 2.13, yielding

$$\widetilde{G}(s) = \frac{kT}{\pi a \langle \Delta r^2(1/s) \rangle \Gamma \left[1 + \alpha(s)\right]},$$

Analogous, this procedure can be performed in the Fourier domain [61], yielding

$$\begin{aligned} G^*(\omega) &= \frac{kT i^{\alpha(\omega)}}{\pi a \langle \Delta r^2(1/\omega) \rangle \Gamma \left[1 + \alpha(\omega)\right]} \\ &= \frac{kT}{\pi a \langle \Delta r^2(1/\omega) \rangle \Gamma \left[1 + \alpha(\omega)\right]} \left(\cos\left(\frac{\pi \alpha(\omega)}{2}\right) + i \cdot \sin\left(\frac{\pi \alpha(\omega)}{2}\right) \right). \end{aligned} \tag{3.8}$$

Dasgupta et al. [55] extended this method in the following way:

1. A second-order polynomial is fitted to $\ln\langle\Delta r^2(t)\rangle$ locally, where values around t are weighted with a Gaussian kernel.

2. Instead of computing α as in equation 3.7, the first-order logarithmic derivative $\alpha(\omega)$ and second-order logarithmic derivative $\beta(\omega)$ are computed from the polynomial fits to $\ln\langle\Delta r^2(t)\rangle$.

3. Equation 3.8 was empirically adapted to

$$G'(\omega) = G(\omega)\left\{1/\left[1+\beta'(\omega)\right]\right\}\cdot\cos\left[\frac{\pi\alpha'(\omega)}{2} - \beta'(\omega)\alpha'(\omega)\left(\frac{\pi}{2}-1\right)\right]$$
$$G''(\omega) = G(\omega)\left\{1/\left[1+\beta'(\omega)\right]\right\}\cdot\sin\left[\frac{\pi\alpha'(\omega)}{2} - \beta'(\omega)\left[1-\alpha'(\omega)\right]\left(\frac{\pi}{2}-1\right)\right],$$

   where

$$G(\omega) = \frac{kT}{\pi a\langle\Delta r^2(1/\omega)\rangle\Gamma\left[1+\alpha(\omega)\right]\left[1+\beta(\omega)/2\right]},$$
$$\alpha'(\omega) = \frac{\mathrm{d}G(\omega)}{\mathrm{d}\omega},$$
$$\beta'(\omega) = \frac{\mathrm{d}\alpha'(\omega)}{\mathrm{d}\omega}.$$

As the Mason method consists of many steps that can be realized in many different ways in the numerical analysis. Implementations of the method which can serve as templates have been made publicly available in the IDL programming language by Crocker and Hoffmann [73], in the MATLAB programming language by Daniel Seara[2] and in the Python programming language by Maier and Haraszti[75].

3.3.2 Paust Method

Paust developed a method to in which the MSD of a microrheological probe bead is fitted with a model with the goal to determine the complex modulus G^* of a viscoelastic medium [76].

Paust's model for the MSD $\langle\Delta r^2(\tau)\rangle$ is a linear combination of 3 parts:

1. Elastic part, constant MSD:

$$\langle\Delta r^2(\tau)\rangle = \frac{kT}{K} = A,$$

   with elasticity of the medium K, combined into constant A.

2. Viscous part, linear increase:

$$\langle\Delta r^2(\tau)\rangle = 2nD\tau = 2n\frac{kT}{6\pi\eta a} = B\tau,$$

   with viscosity of the medium η, radius of the particle a and number of dimensions n.

[2] see `https://github.com/dsseara/microrheology`

3. Confinement to a cage:

$$\langle \Delta r^2(\tau) \rangle = C \left(1 - \exp\left(-\frac{\tau}{D}\right)\right),$$

where D is the relaxation time.

Combined, these terms yield the fit model with fit parameters A, B, C, D:

$$\langle \Delta r^2(\tau) \rangle = A + B\tau + C \left(1 - \exp\left(-\frac{\tau}{D}\right)\right). \tag{3.9}$$

Paust applied an $1/\tau$ weighing to the fits, i.e. deviations from lower lag time data were penalized more by the fitting algorithm. All 3 terms can be analytically Laplace transformed to determine G^* via equation 2.13 and coordinate transformation $s = i\omega$:

$$G^*(\omega) = \frac{-kT\omega^2(1 + iD\omega)}{\pi a i\omega \left(B + iBD\omega + i\omega \left(A + C + iAD\omega\right)\right)} \tag{3.10}$$

To demonstrate the effect of the different parameters, a set of curves with various values are shown in figure 3.4.

For more details see chapter 5 of Paust's dissertation [76].

3.4 Plateau Modulus

Entangled networks of polymers can exhibit a wide region in frequency where the storage modulus $G'(\omega)$ is almost constant [45, 77]. The value of G' in the plateau region is referred to as the plateau modulus G_0.
For which frequency ω_0 exactly the plateau value $G_0 = G'(\omega_0)$ is determined is not too relevant, due to G' being almost constant in a wide range. A common method is to choose $\omega_0 = \omega_{\text{min}}$, where ω_{min} is the minimum of the loss modulus G'' [77]. For the microrheology data in chapter 6, this method could not consistently be applied. Instead, G_0 was evaluated at $\omega_0 = \omega_{\text{saddle}}$, where ω_{saddle} is the frequency of the saddle point of G'. Numerical application of this method was consistently possible and the saddle point was found to lie in the plateau region for all data in chapter 6.

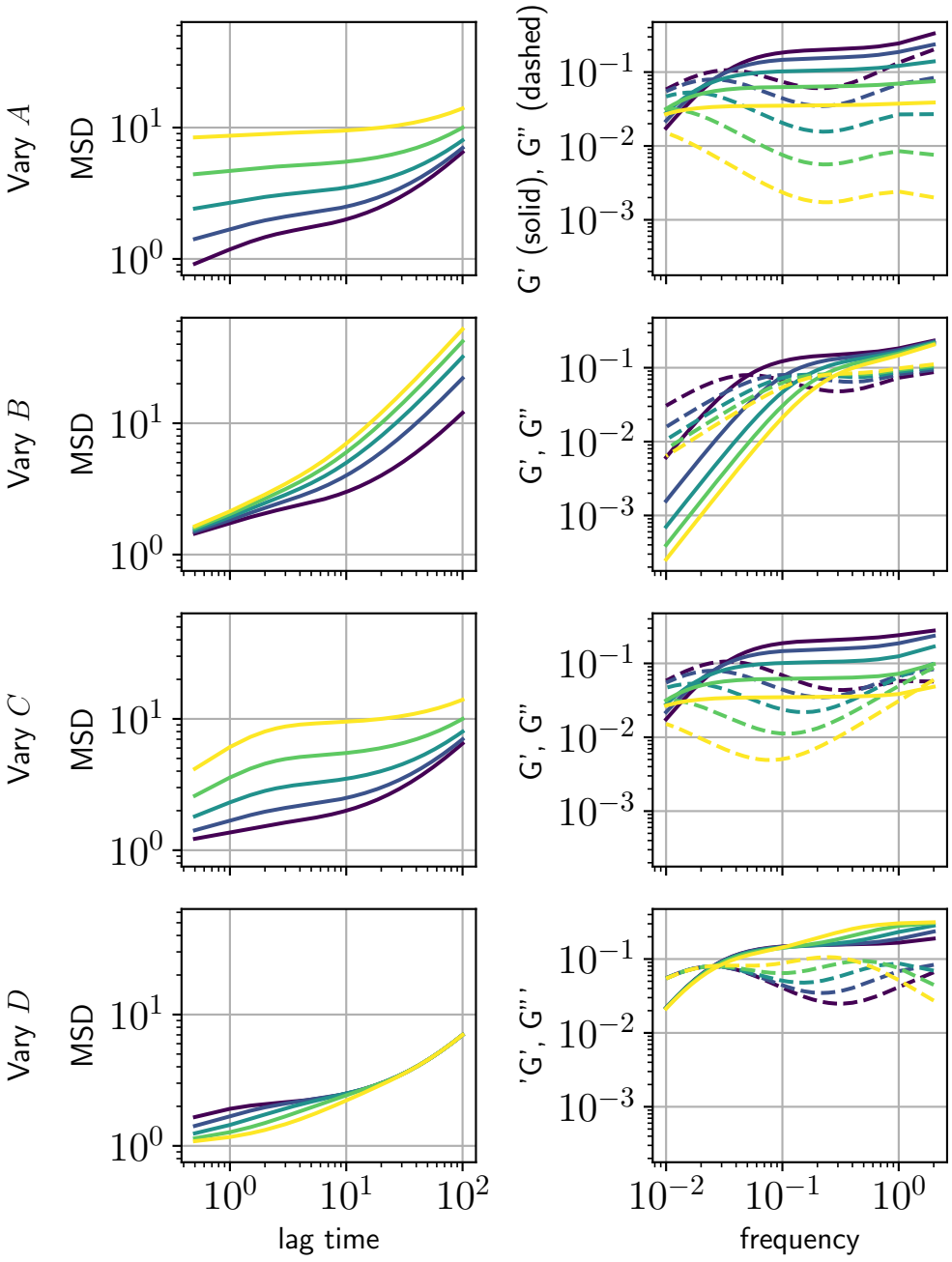

Figure 3.4: Paust model for MSDs (equation 3.9) and G^* (equation 3.10) drawn for various values of parameters A, B, C, D. Each row shows curves with one value varied and the remaining 3 values fixed. Line colors indicate size of value, starting with dark purple for the lowest and going to bright yellow for the largest value. Energy kT and MR bead radius a were set to 1 (unitless). Fixed parameters were set to $A = 1, B = 0.05, C = 1, D = 1$. In the respective rows, varied values were $A, C, D \in \{0.5, 1, 2, 4, 8\}$ and $B \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$.

3.5 actomyosin_analyser Python Package

For the analysis of simulation data produced during my thesis I developed the python package `actomyosin_analyser`. This package was created for two reasons:

1. To provide one unified programming interface for analyses of all kinds of different actomyosin simulation data, regardless with wich simulation framework the data was produced,

2. and to have generalized analysis code in one central place to avoid repetition.

In order to achieve the first item, the programmer has to implement the abstract base class `DataReader` for her data. The `bead_state_model` simulation framework (see chapter 4), which is the main subject of this thesis, contains a `DataReader` implementation that is compliant with the abstract `DataReader` of the `actomyosin_analyser` package, and was used throughout all analyses in this thesis.
The project page is hosted on gitlab.com [3]. Install instructions and more details can be found in the project's manual[4].

3.5.1 DataReader Class

In the `actomyosin_analyser` package, all analyses of the data of a single actomyosin simulation are handled by the `Analyser` class. To access the data of the simulation, the `Analyser` uses implementations of the abstract `DataReader` class (see diagram in figure 3.5). I.e., to analyse actomyosin data with the `actomyosin_analyser` package, the user has to provide a `DataReader` for her data. The `DataReader` has to handle reading the simulation data and e.g. provide the filament trajectories in a compatible format via the method `DataReader.get_filament_coordinates`. For the framework `bead_state_model` that was used for simulations throughout this thesis, an implementation of the `DataReader` is included in the `bead_state_model` package.

An instance of a `DataReader` has to be passed along as argument, when an instance of `Analyser` is created. See a code example in listing 3.1.

The implementation of the `DataReader` of the `bead_state_model` package used in listing 3.1 does not technically inherit from the abstract base class `DataReader`, to avoid making `actomyosin_analyser` a dependency of `bead_state_model`. However, it implements all the defined methods and, since Python does not check types at run time, works well with the `Analyser` class.

3.5.2 Analyser Class

The `Analyser` class is the heart of the `actomyosin_analyser` package. It provides many analysis methods useful for actomyosin data. A list of the provided

[3] Full URL to project page: `https://gitlab.com/ilyas.k/actomyosin_analyser`
[4] Full URL to manual: `http://akbg.uni-goettingen.de/docs/actomyosin_analyser/`

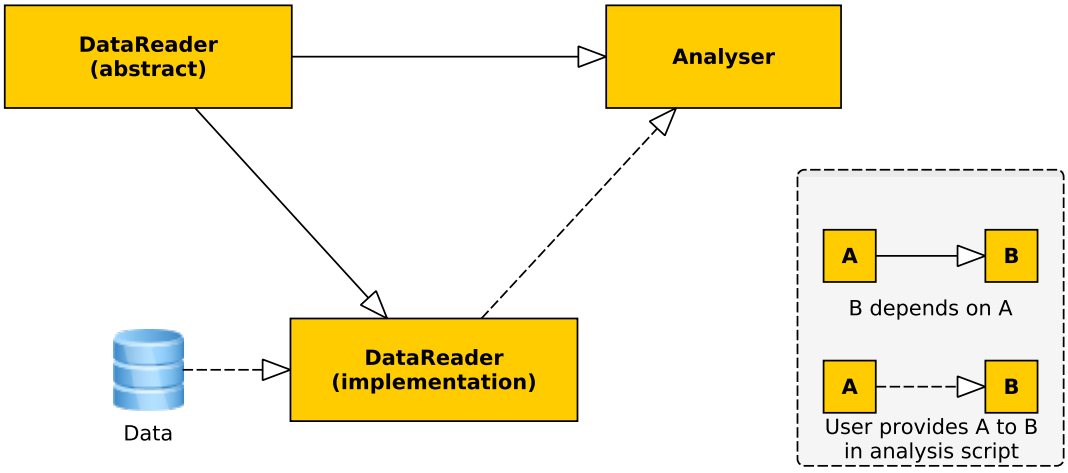

Figure 3.5: In order to make the `Analyser` class usable for any kind of actin simulation data (regardless with which framework the data was produced), the `Analyser` depends on an abstract `DataReader` class. This abstract class has to be implemented for the user's data, and the implementation has to be provided/injected to the `Analyser` class in the user's analysis scripts.

```
from bead_state_model.data_reader import DataReader
from actomyosin_analyser.analysis.analyser import Analyser

dr = DataReader('data.h5')
a = Analyser(dr, 'analysis.h5')
```

Listing 3.1: Provide a `DataReader` instance (`dr`) when creating an `Analyser` instance. The file `'data.h5'` has to exist and contain output of a `bead_state_model` simulation (HDF5 file format). The file `'analysis.h5'` will be created if it does not exist, it will store some of the final or interrim results computed by the `Analyser` instance `a`.

methods can be found in the documentation of the `actomyosin_analyser` package. To make the `Analyser` independent of the format of the raw data, reading of raw data is externalized to the aforementioned `DataReader` class. An instance of a `DataReader` has to be provided, when an `Analyser` is created, as shown in the example in listing 3.1.
For many data that take a lot of time for computing or converting in desired formats, the `Analyser` is designed to compute/convert those data only once, then store them in cache files, and read them upon next request of that data. The user need only use the `get` methods for that, internally the reading is delegated to a matching private `_read` method if the requested data is present, or to a `_compute` method if it is not. This caching pattern is depicted in the diagram in figure 3.6. In this caching pattern where many data are computed only once, compute times can be heavily reduced, but the trade-off is that more disk space will be occupied. For some aggregated results like MSDs (see section 3.2),

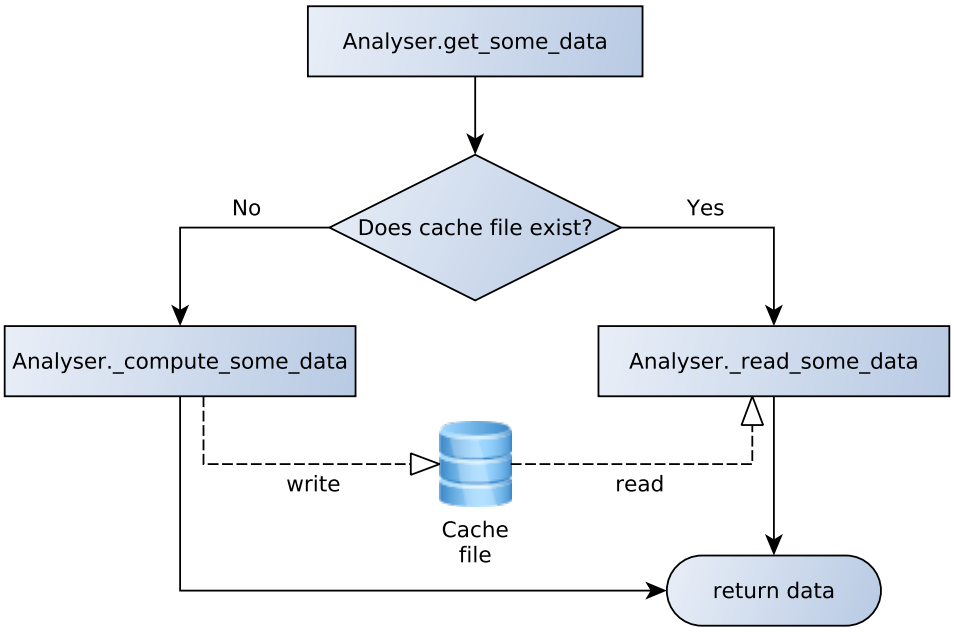

Figure 3.6: Diagram of the caching pattern that is employed by most `get` methods of the `Analyser` class. This is illustrated here with a toy method named `get_some_data`, but the principle is the same for actual methods like `get_trajectories_of_filaments` and `get_filament_center_of_mass_msds`. On first execution, `get_some_data` fails to find a cached result, and will call the method `_compute_some_data`. The computed data will be written to a cache file and returned. Upon consecutive execution, the cache file is present, and rather than computing the result again, the resulting data will be loaded from the cache file via the `_read_some_data` method and returned.

the benefit of saving time outweighs the draw-back that additional disk space is occupied by far. In other cases, the benefits are debatable. Nonetheless, in these cases the choice was often made to accept larger files for saving compute time. For example, the `Analyser` will store a copy of all trajectories in the analysis file when they are retrieved from the `DataReader` for the first time. This can up to double the disk space per simulation. But for all simulation frameworks for which compatible `DataReaders` were implemented (cytosim, `bead_state_model`), this was found to decrease the compute times a lot.

Some standard computations that the `Analyser` can perform include:

- Computation of ensemble MSDs of the center of mass of filaments (filaments can have varying number of beads/coordinates).
- Computation of MSDs of individual particles, which were for example used to compute MSDs of all outer beads and all central beads of filaments with fixed identical number of beads.
- The polymer density estimated via kernel density estimation (see section 7.4 for an in-depth example application).

- The number of cross-links/motors.
- The contour lengths, end-to-end vectors and end-to-end distances of filaments.

A more exhaustive list of analyses that can be performed by the `Analyser` class can be found in the online documentation.

Examples

Basic examples how to analyse simulation data with the `actomyosin_analyser` package and the `Analyser` class can be found in section 4.1.5, and expecially in the corresponding code listing L6 in section A7 in the appendix which demonstrates how to read polymer coordinates at different time steps.
More advanced examples can be found among the analysis scripts in the "experiment" folders. See section A1 in the appendix for instructions how to access these folders.

3.5.3 ExperimentIterator and Pipelines

To make best use of the `actomyosin_analyser` package, "experiments" should be structured as described in section A1.2. This allows the user to make use of the package's `ExperimentIterator` class in combination with several predefined analysis pipelines. The `ExperimentIterator` reads an experiment's simulation index table, and automatically groups simulations by varied parameters. Color gradients for plots can automatically be generated by one of the parameters varied in the experiment. For some analyses that were performed for many experiments, like applying the Paust method (see section 3.3.2) to MR experiments, pre-defined pipelines exist in the `actomyosin_analyser` package that automize these analyses to a high degree. Example usage of pipelines can be found among the publicly available datasets of this thesis, e.g. in the script `"perform_paust_method.py"` that is part of experiments E25, E34, E36 and E37 (see section A1.3 in the appendix).

3.5.4 Exporting to XYZ Format

Visualization of particle coordinates is often the first and most intuitive method to inspect simulation results for any errors, artifacts, or other undesirable outcomes. For visualizations with external tools, ReaDDy and `actomyosin_analyser` provide methods to export particle coordinates to XYZ. The XYZ file format is a loosely defined data format for particle trajectories of 3D simulations. For visualization with ovito [78], the format accepts more than the typically found 4 columns for particle type and 3 cartesian coordinates. The `actomyosin_analyser` package uses additional columns to define orientation of cylindrical segments (rather than using only the coordinates of start and end positions of segments), to give the displayed filaments a more filamentous look. This is demonstrated for one filament in figure 3.7. For single filaments, both the visualization using beads (coordinates exported to XYZ with default ReaDDy method) as well

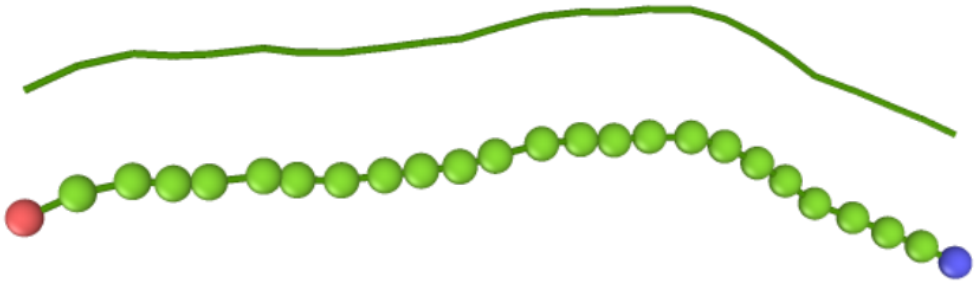

Figure 3.7: Identical bead chain configuration in 2 different visualizations using ovito. **(Top)** Bead chain drawn as sequence of cylindrical segments, where one segment connects two consecutive beads (XYZ file generated with `actomyosin_analyser`). **(Bottom)** The same cylindrical segment visualization overlayed with beads of the bead chain. XYZ file of bead coordinates was generated with default export method of ReaDDy.

as the cylindrical visualization (exported with `actomyosin_analyser`) are fine. However, for crowded simulation boxes, the bead version often fails to visualize any information (see for example figure 4.4).

Tubes at 3D Periodic Boundaries

When periodic boundary conditions are used, segments of discretized filaments can cross up to 3 boundaries. If boundary crossings occur, intersection of the segment with the bounding box and its periodic projection are computed. The tube is drawn from start point of the segment to the intersection, and another tube is drawn starting from the periodic projection of the intersection. A case with two boundary crossings in 2D is illustrated in the sketch in figure 3.8.

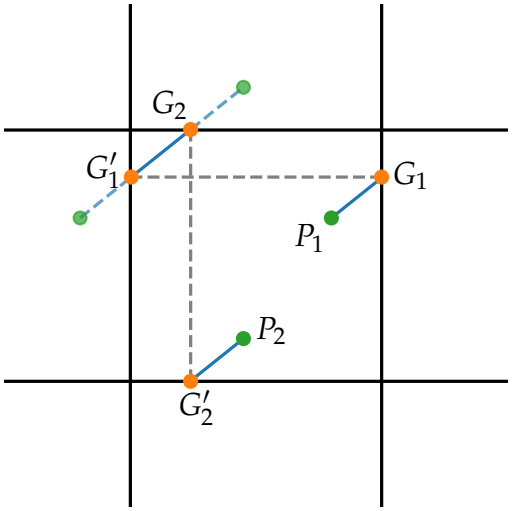

Figure 3.8: Sketch to illustrate how a segment that crosses periodic boundaries is visualized. The segment stretches from point P_1 to point P_2. Along the way from P_1 to P_2, two periodic boundaries are crossed. At the first intersection with the right-hand side boundary, ghost points G_1 and its projection G_1' are inserted. The segment is drawn from P_1 to G_1, and continues from G_1'. The continued segment intersects with the top boundary, where ghost point G_2 is inserted. The segment then continues from the projection G_2' to P_2.

Chapter 4

Bead-state Model

The central idea of the bead-state model is to implicitly include actin binding proteins by assigning different states to segments of filaments. This way the number of particles can be reduced in numerical simulations. Since number of particles is usually one of the main drivers of simulation time in particle simulations, this could lead to a decrease in simulation time.

In the BSM, filaments are modeled as bead chains (see depiction in figure 4.1).

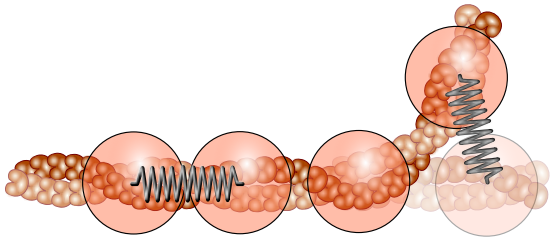

Figure 4.1: bead chain. Image was adapted by templates provided by Burkhard Geil.

Beads impose harmonic interaction potentials for bending and stretching on their neighbors, leading to following bending and stretching energies for a polymer of n beads:

$$U_{\text{bend}} = \frac{k_{\text{bend}}}{2} \sum_{i=1}^{n-2} \theta_i^2, \tag{4.1}$$

$$U_{\text{stretch}} = \frac{k_{\text{stretch}}}{2} \sum_{i=1}^{n-1} (l_0 - l_i)^2. \tag{4.2}$$

To unambiguously mark the order of the beads, the last and the first bead are assigned states "`tail`" and "`head`" respectively (figure 4.2, top). In unbound polymers, all other beads inbetween are in state "`core`".

Polymers can bind via passive (cross-links, CL) and active binding proteins (motors). These binding proteins are not included as distinct components explicitly. Rather, they are modeled implicitly as states of beads. Two `core` beads (b_i, b_j) of two distinct polymers can bind, if their distance r_{ij} is smaller than the

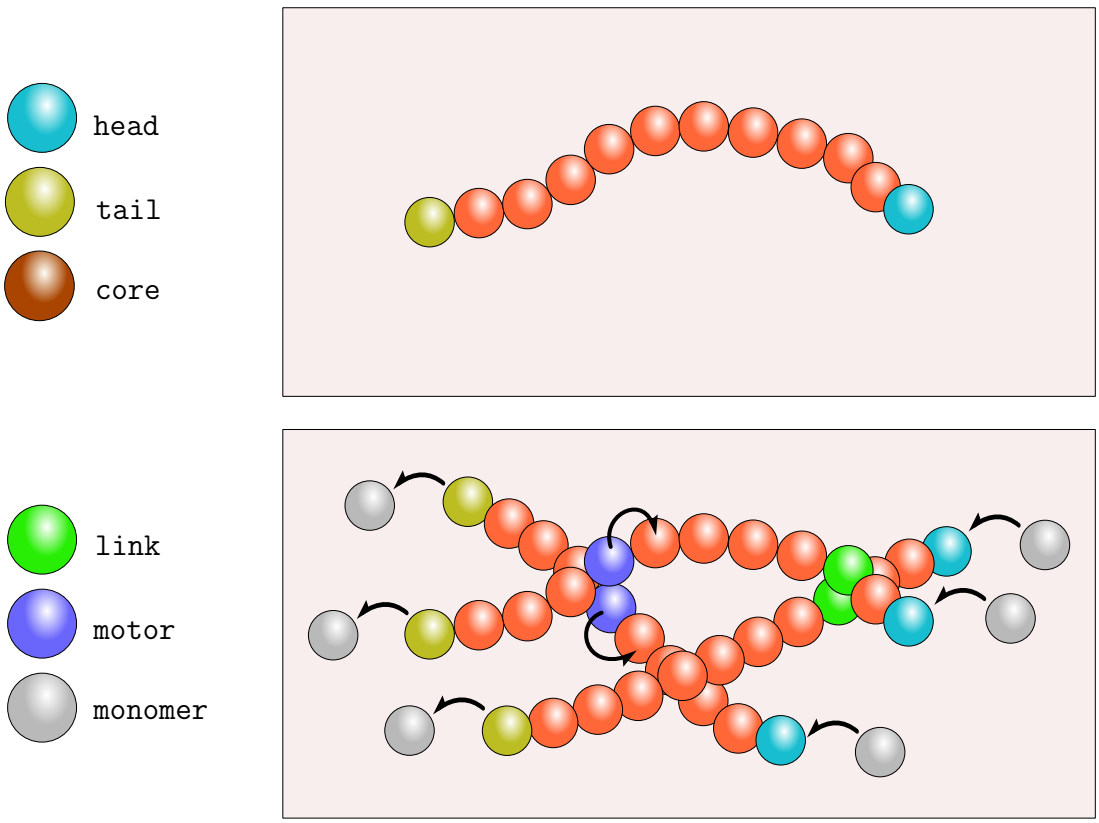

Figure 4.2: Top: isolated bead chain (polymer). First and last beads are in state `tail` and `head`. All other beads are in state `core`. Bottom: network of bead chains (polymers). Two `core` beads of two distinct polymers can form bonds by transitioning to `link` or `motor` states. Polymers can assemble/disassemble, i.e. sinlge beads (monomers) can be added at the head of a polymer and removed at its tail. Images were adapted from templates provided by Burkhard Geil.

binding range r_{bind}. After binding, the state of the beads changes from `core` to `link` or `motor`. `Motor-motor` and `link-link` pairs enact a harmonic potential on each other,

$$U_{\text{link},ij} = \frac{k_{\text{link}}}{2}(l_0 - r_{ij})^2.$$

Binding occurs with a rate of $k_{\text{on, link}}, k_{\text{on, motor}}$, unbinding with a rate of $k_{\text{off, link}}, k_{\text{off, motor}}$. In addition, motors can perform a step towards a polymer's `head` with rate k_{step}. When performing a step, a `motor` bead passes its state along the polymer to the following `core` bead. The `motor` bead transitions from state `motor` to core, while the following bead transitions from `core` to `motor`. If the following bead is in a state different than `core`, the step cannot be peformed.

4.1 Python Implementation: bead_state_model

An implementation of the bead-state model in the Python programming language is being developed as free software on gitlab[1], it is referred to as `bead_state_model` in this thesis. It is based on the simulation engine ReaDDy [64], a general-purpose particle simulation framework. ReaDDy is written in C++, but provides a user interface in Python.

The Python interface allows users to freely configure particle simulations of different particle species and reactions including transitions from one species to another, and bond formation and bond breaking between particles. In this way, multiple particles can form a connected graph, where each bond is an edge in the graph. These graphs are called topologies in ReaDDy.

The Python package `bead_state_model` acts as a wrapper around ReaDDy's Python user interface and handles the definitions of the polymer topology, the topology particle species tail, core, head and motor (cross-links are implemented as motors with $k_{\text{step}} = 0$, since we did not use both motors and cross-links simultaneously yet). For the integration of the equation of motion (equation 2.2), `bead_state_model` relies on the Euler-Maruyama integration scheme, which is the default integrator employed by ReaDDy [64] (see also section 3.1.2).
In order to use ReaDDy as the base for `bead_state_model`, C++ as well as Python source code of ReaDDy had to be adapted. Adaptations to the ReaDDy source code were contributed to the open-source project through ReaDDy's project page[2]. More details of these contributions can be found in the following sections.
Many of the following technical details can also be found in the project's documentation [3].

4.1.1 Potentials

In the simulations, forces are results of potentials defined between pairs of beads. The potentials within a polymer defined in equations 4.1 and 4.2 are defined as topology potentials in the context of ReaDDy. These are based on bonds between topology particles. How this is done in `bead_state_model` can best be described by a short excerpt of the package's code:

```
tuples = [
  ("head", "core"),
  ("core", "core"),
  ("core", "tail"),
  ("core", "motor"),
  ("motor", "motor"),
  ("head", "motor"),
  ("motor", "tail")
]
```

[1] Project page: `https://gitlab.com/ilyas.k/bead_state_model`
[2] ReaDDy project page: `https://github.com/readdy/readdy`
[3] Documentation: `http://akbg.uni-goettingen.de/docs/bead_state_model/`

```
for (t1, t2) in tuples:
    system.topologies.configure_harmonic_bond(
        t1, t2, force_constant=k_stretch, length=1.0
    )

triplets = [
  ("head", "core", "core"),
  ("core", "core", "core"),
  ("core", "core", "tail"),
  ("head", "core", "tail"),
  ("head", "motor", "core"),
  ("head", "core", "motor"),
  ("head", "motor", "tail"),
  ("core", "motor", "core"),
  ("core", "core", "motor"),
  ("core", "motor", "tail"),
  ("motor", "core", "tail")
]

for (t1, t2, t3) in triplets:
    system.topologies.configure_harmonic_angle(
        t1, t2, t3, force_constant=k_bend,
        equilibrium_angle=np.pi
    )
```

Potentials defined above also apply on two beads of two distinct polymers, if a bond forms between their beads (i.e. when a pair of `core` beads turn into a pair of `motor` beads).

In addition to the two potentials defined above, a steric interaction between beads is defined to keep polymers from moving through each other. Beads of a polymer repell each other at a distance smaller than their diameter d,

$$V_{\text{repulsion, ij}} = \frac{k_{\text{repulsion}}}{2}(d - r_{ij})^2 \text{ for } r_{ij} \leq d.$$

The fact that the resting length of the stretch potential is set to be equal to the diameter of the beads, will make it statistically very unlikely that large gaps between beads of a polymer will remain.

4.1.2 Reactions

The reactions of the bead-state model (link binding, link unbinding, motor step, attachment of monomer, detachment of monomer) were defined via the user interface of ReaDDy [64]. ReaDDy discriminates between spatial reactions, which can occur based on the distance between particles, and structural reactions, that involve particles within the same topology (polymer particles are part of a polymer topology). While spatial reactions are defined via a descriptive text, topol-

ogy reactions have to be defined via so called *recipes* (ReaDDy's modular concept to define reaction processes in code).
In `bead_state_model`, link binding is the only spatial reaction, while the other 4 reactions are structural reactions. Excerpts of the source code of `bead_state_model` where link binding and the recipe for motor steps are defined are shown in listings L2 and L3 in section A7 in the appendix.

Self-fusion in ReaDDy Topologies

In networks of polymers with cross-links or motors, multiple polymers can become one large connected graph, which will become one large topology of type polymer in ReaDDy.

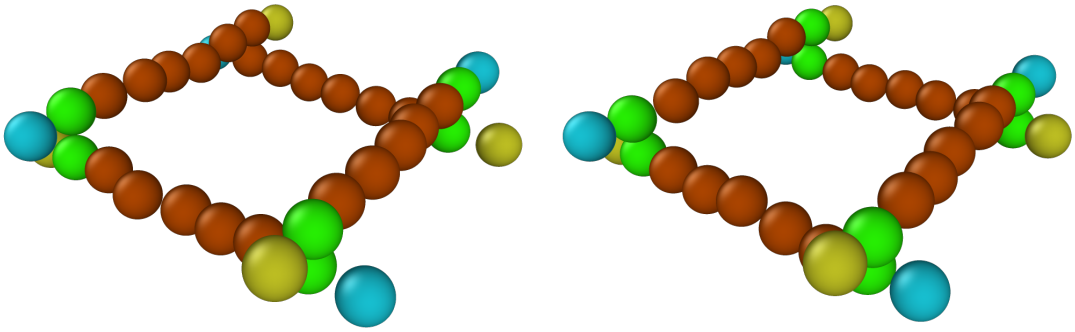

Figure 4.3: 4 filaments forming a square, providing 4 potential binding sites for motors, simulated with two different settings. Same color coding as in figure 4.2 was used, with motors shown in green. **(left)** No self-reaction allowed. Only 3 bonds can form (left, right and bottom corners). The last one (at the top corner) cannot form, because polymers belong to the same topology. **(right)** Same setup, but now with self-reactions allowed if at least 6 edges are between the two particles in the topology's graph. All 4 motor bonds can form.

Consider the four filaments in figure 4.3 (left), where a bead of polymer p_1 connects to a bead of p_2, and similarly p_2 connects to p_3, p_3 connects to p_4. Now, for polymers p_1 and p_4 to connect, which are part of the same topology in ReaDDy, we would have to allow self-reaction for topologies:

```
system.topologies.add_spatial_reaction(
  "link: polymer(core) + polymer(core)"
  "-> polymer(motor--motor) [self=true]", ...
)
```

This would, however, allow two adjacent cores of the same polymer to form an undesired motor-motor bond as well. To circumvent this, a feature was contributed [4] to the ReaDDy simulation framework on their github project page, which allows us to define a minimum distance in the topology graph for self-binding to be allowed:

[4] Pull request #171: `https://github.com/readdy/readdy/pull/171`

```
system.topologies.add_spatial_reaction(
  "link: polymer(core) + polymer(core)"
  "-> polymer(motor--motor) [self=true, distance>6]",
  ...
)
```

Choosing a value that is larger than 1 will prohibit already connected core-core pairs to turn into motor-motor pairs. Choosing the minimum distance smaller than the number of beads in 2 to 3 polymers will allow polymers p_1 and p_4 in figure 4.3 (right) to bind.

`FilamentHandler` Class

The major fraction of particles in `bead_state_model` simulations are the beads that represent the coarse-grained polymers. This means, that we mostly deal with two topologies: isolated polymers, or linked polymers where two or more isolated polymer topologies merge into one single topology. A class named `FilamentHandler` was implemented to keep track of all bonds between polymer beads, this includes bonds whithin a polymer, as well as bonds (links) across polymers. The motivation for this class is twofold:

1. When two or more polymers are linked, it is not always possible to determine which beads belong to which polymer from the information ReaDDy stores for these merged topologies alone. Keeping track of that has to be handled separately.

2. Determining the direction for motor steps (they only move from tail to head) can be computationally costly, when done based on a topology's graph alone. Furthermore, in some cases, determining which particles belong to which filament is not possible from a topology's graph alone, which makes it necessary to keep track of that information separately.

All reactions defined for polymers update the `FilamentHandler`. When occurance of a reaction is being determined, the `FilamentHandler` provides information on eligible reactant beads, and passes it to the functions that dynamically determine the rates of the reactions.

Reaction Rates

The user needs to define rates for the polymeric reactions: attachment of beads at the head, detachment of the tail bead, binding/unbinding of motors, and motor steps (along a polymer, in the direction from tail to head).
Rates in ReaDDy are specified in units of inverse time. A rate λ leads to a probability of $p = 1 - e^{-\lambda \Delta t}$ for the reaction to occur at each integration step of size Δt [64]. The probability p comes to bear for each reaction candidate. ReaDDy allows to specify these rates dynamically for topology reactions by specifying a rate function when the reaction is defined. In `bead_state_model` the topology reactions are attach, detach, motor step, motor unbinding. When a topology is checked for occurance of a reaction at the current integration step, the total rate

has to be determined by the dynamic rate function. In case of motor unbinding for example, the total rate is the number of motor pairs in the topology (i.e. number of motors divided by 2) multiplied by the individual rate. This is done in conjunction with the `FilamentHandler`, which keeps a list of motors for each topology.
Motor binding, by contrast, is a spatial reaction rather than a topology reaction. The number of reaction candidates depends on the spatial composition of all polymers. Two core beads can form a link and become a bond motor pair, if their distance is less than a threshold specified via parameter `reaction_radius_motor_binding` (attribute of the `Parameters` class). In addition, the self-reaction distance (attribute name: `min_network_distance`) comes into play for polymers belonging to the same topology (see figure 4.3).
Dynamic rates for spatial reactions were not possible in ReaDDy, until I added the feature via a pull request[5]. Before the contribution, only static rates were possible. Dynamic rates allowed to introduce capping to the number of motors, as well as decreasing the binding rate with the number of motors already present. This is optional, and can be controlled with the parameter `n_max_motors`. Setting it to `None` will disable capping and disable adjusting the rate with the number of motors. Setting it to a positive integer will decrease the rate linearly until it reaches 0 at the defined number,

$$\lambda_{\text{effective}} = (1 - n_{\text{motors}}/n_{\text{motors, max}})\lambda.$$

4.1.3 Network Assembly

The Python package `bead_state_model` contains tools to generate polymers and polymer networks. This section describes the algorithm used by the class `CreateNetworkSimple` to generate networks with N polymers with identical number of beads M. The suffix "simple" in the class' name is to discriminate it from class `CreateNetworkNatural`, which generates networks with polymers of different lengths, rather than polymers of all identical lengths.

Nucleation of Polymers

These steps are performed to place the first two beads of N polymers in the simulation box, referred to as nucleation of the polymers:

[5] Pull request #192: `https://github.com/readdy/readdy/pull/192`

Nucleation Algorithm

For N polymers, do the following:

1. Place the first bead of the polymer at a random position r in the simulation box.
2. If the position overlaps with a previoulsy placed bead, go back to step 1.
3. For the second bead, draw distance l from a normal distribution with mean $\mu_l = 1x_0$ and standard deviation $\sigma_l = 1/\sqrt{k_{\text{stretch}}}$.
4. Draw a random direction $(\phi, \theta) \in [0, 2\pi) \times [0, \pi)$.
5. Check if placing second bead on designated position (ϕ, θ) on sphere with radius l around the first bead would lead to overlap.
6. If it leads to collision, go back to step 3.

After nucleation, there are $2N$ beads in the simulation box.

Elongation

After nucleation, polymers are being elongated by adding more beads. Beads are placed following the direction of last segment plus some noise. The algorithm in detail is as follows:

Elongation Algorithm (Part 1)

Repeat the following $M - 2$ times:

For all N filaments, do:

1. Determine the unit direction vector u pointing from second to last to last bead.
2. Draw random distance l from a normal distribution with mean $\mu_l = 1x_0$ and standard deviation $\sigma_l = 1/\sqrt{k_{\text{stretch}}}$.
3. Draw random angle θ from normal distribution with mean $\mu_\theta = 0$ and standard deviation $\sigma_\theta = 1/\sqrt{k_{\text{bend}}}$.
4. Draw random rotation from uniform distribution $[0, \pi)$.
5. Construct vector $v = (l, 0, 0)$.
6. Rotate vector v around anchor $(0, 0, 1)$ by angle θ, yielding $v_\theta = (l \cos\theta, l \sin\theta, 0)$.
7. Rotate vector around anchor $(1, 0, 0)$ by angle γ, yielding $v_{\theta,\gamma} = (l \cos\theta, l \sin\theta \cos\gamma, l \sin\theta \sin\gamma)$.

(continued on next page)

Elongation Algorithm (Part 2)

8. Construct rotation matrix R that would rotate vector $(1, 0, 0)$ onto u, use it to rotate $v_{\theta,\gamma}$.
9. Add the final vector to the position of the last bead. This is the designated position of the bead to append.
10. Check for overlap at designated position. If it would lead to overlap, repeat from step 1.
11. Place bead at designated position.

The nucleation as well as the elongation do not take repulsion between beads into account. This was implemented, because networks were supposed to work also with simulation software that has no repulsion between direct neighbors. The plot of segment length distributions in figure 4.7 sheds some light how polymers behave during network assembly in contrast to simulations.

Include Potentials during Network Generation

The `bead_state_model` framework is tailored towards simulations of systems that include "external" particles like microspheres in microrheology or optical tweezer experiments and limiting geometries like cortex-like thin layers of actin. Overlap between external particles / energy walls can lead to very large forces during simulations, which can cause the collapse of whole systems. To avoid this, overlaps can be carefully removed by simulations with very small time steps, which are referred to as equilibration simulations in this thesis. In order to reduce the risk of very high energies in initial states, arbitrary external potentials can be included by providing a custom `acceptance_rate_handler` to a `CreateNetworkSimple` instance. The potentials are considered during filament placement by assigning a batch of possible positions probabilities that decrease with higher energy. From these batches, only one position is randomly chosen based on its assigned probability, making it more likely that a lower energy position is chosen.
While this approach does not yield equilibrated systems, it allows the user to cut down the time that is required for equilibration simulations, since very high energies are avoided. This allows for a larger time step during equilibration simulations without the risk of a collapse due to very high forces.
For the systems used in this thesis, predefined `acceptance_rate_handler` exist. For example, how to incorporate a spherical repulsive potential at the center of the simulation box is shown in the following code snippet:

```
from bead_state_model.network_assembly.create_network import\
  AcceptanceRateHandlerBeadInNetwork
from bead_state_model.network_assembly.create_network_simple import\
  CreateNetworkSimple

system = CreateNetworkSimple(
```

```
    box=box,  # specifies your system size
    k_bend=k_bend,  # stiffness of created polymers
    k_stretch=k_stretch,  # spring constant defining segment
                          # length of polymers
    n_filaments=700,
    n_beads_per_filament=25
)

ar_handler = AcceptanceRateHandlerBeadInNetwork(
    n_batch=30,  # Number of possible positions of which
                 # only one is chosen.
    radius=radius_bead,  # Cut-off for repulsive potential.
    k_repulsion=k_repulsion,  # Strength of repulsive potential.
    position=box/2  # Place bead at center.
)

system.acceptance_rate_handler = ar_handler

system.run()  # generate network
```

4.1.4 Including External Particles and Potentials

When simulations are set up, `bead_state_model` handles the definition of polymers, cross-links/motors, polymerization and depolymerization. Any other particles that are to be added to the system have to be defined manually. In order to do so, the user can utilize the full set of methods that ReaDDy (the simulation engine that `bead_state_model` is based on) provides to

- define new particle species,
- define new topology types and topology particle species (a topology in ReaDDy is a set of particles connected with bonds, and defines what energies arise due to the bonds),
- define potentials between pairs of particle species,
- define external potentials and on which particle species they act.

To access these ReaDDy methods in `bead_state_model` simulations, one has to create an implementation of the `BaseSetupHandler` class and override the `__call__` method, which receives the ReaDDy "system" as a parameter. The example below shows how this would be done to create a spherical particle that interacts repulsively with polymer particles:

```
from readdy import ReactionDiffusionSystem
from bead_state_model import BaseSetupHandler, Simulation

class RepulsiveSphere(BaseSetupHandler):
```

```
    def __init__(self, radius: float, force_constant: float):
        self.r = radius
        self.k = force_constant

    def __call__(self, system: ReactionDiffusionSystem):
        system.add_species('sphere', diffusion_constant=1/self.r)

        for polymer_particle in ['head', 'tail', 'core', 'motor']:
            system.potentials.add_harmonic_repulsion(
                'sphere', polymer_particle,
                force_constant=self.k,
                # 0.5 is the radius of the polymer particles:
                interaction_distance=0.5 + self.r
            )

r = 5.0
k = 100.0

# create bead_state_model simulation and pass it an
# instance of RepulsiveSphere:
s = Simulation(
    ...,
    interaction_setup_handler=RepulsiveSphere(r, k)
)
```

A Full examples how the above is done exactly can be found in section A2 and in the online documentation[6]. Furthermore, some of the methods above were used to define microspheres in the simulations in chapters 6 and 7, and to restrict actin to layers in the simulations in chapter 7. The publicly available data and simulation scripts of these simulations (see also section A1 in the appendix) serve as additional examples.

4.1.5 Example: Cross-linked Actin Network

This example[7] demonstrates how to generate and then simulate a homogeneous 3D actin network with cross-links. The user need only interact through 4 classes with the `bead_state_model` package: `CreateNetworkSimple`, `Parameters`, `Simulation`, and `DataReader`. Their usage is demonstrated in the following sections. For a more advanced example including non-polymer particles, see section A2 in the appendix.

[6] Documentation: `http://akbg.uni-goettingen.de/docs/bead_state_model/`

[7] This example serves the purpose of a "How To" as you would find it in a manual of a software, and includes technical language and plots that might seem odd for scientific writing. Bear with me!

Create Network

Executing the script in listing L4 in section A7 will generate 300 filaments consisting of 15 beads each in a simulation box of size `[20.0, 20.0, 20.0]`. It creates the folder `initial_state` and populates it with some output files, the most important of which is the file `beads.txt`, which holds information on filament beads' coordinates and bonds between them (i.e. which beads belong to which filament and in which order). The network assembly algorithm is described in detail in section 4.1.3.

Note that the network's initial state does not contain cross-links, there are no bonds between filaments yet. These bonds will be formed during the simulation. To start a simulation with initial cross-links, one would have to run two separate simulation: the first one starting from the initial state without cross-links, cross-links would then form during that simulation; the second one would then have to be started from the final state of the first simulation.[8]

Run Simulation

The script in listing L5 loads the initial state that was generated in the last section and conducts a simulation. The simulation is rather short in order to not take up too much time. The script creates the folder `simulation_output` and writes parameters and simulation output into that folder.

Visualization

The file `simulation_output/data.h5` is a regular readdy trajectory file, and we can use readdy's export method to export it to `xyz` format:

```
import readdy

traj = readdy.Trajectory('simulation_output/data.h5')
traj.convert_to_xyz(generate_tcl=False)
```

This creates the file `simulation_output/data.h5.xyz`. For the quick visualization in figure 4.4, the software ovito was used [78]. The displayed box is rather crowded. To visualize filaments with thin tubes, you can use the `actomyosin_analyser` package:

```
from bead_state_model.data_reader import DataReader
from actomyosin_analyser.analysis.analyser import Analyser
from actomyosin_analyser.file_io.xyz_exporter import XYZExporter

dr = DataReader('simulation_output/data.h5')
a = Analyser(dr, 'simulation_output/analysis.h5')
```

[8] In general, it is often a good idea to split your simulations in two parts, where the first part serves to equilibrate the network, and the second part then starts from that equilibrated state. Examples of that can be found in section A2 and in the online manual

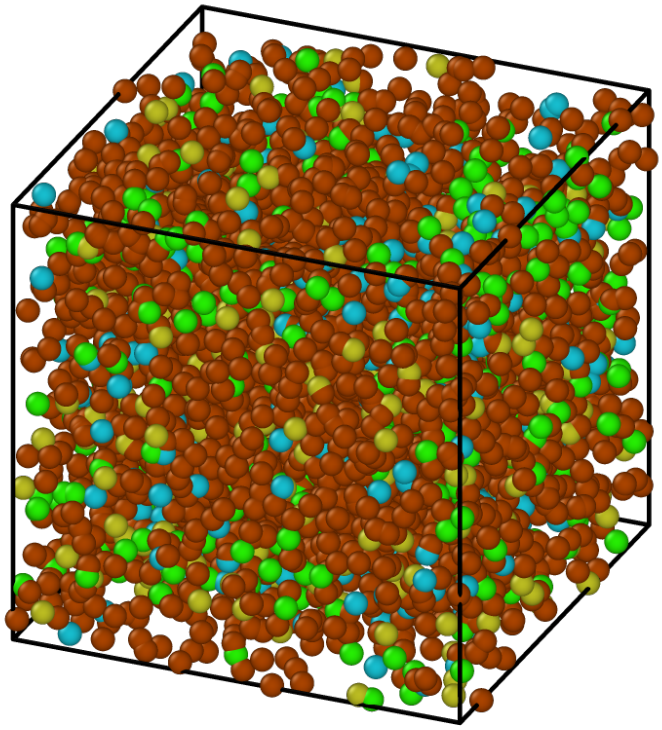

Figure 4.4: 3D visualization of beads of filaments. Colors indicate different bead types: (brown) core, (green) motor, (gold/yellow) head, (teal) tail.

```
expo = XYZExporter()

expo.export_all(a, folder='simulation_output/xyz')
```

When using ovito for visualizations similar to that in figure 4.5, see section 3.5.4 how to map the columns of the `xyz` files.

Reading the Output Data

The trajectories of particles can be read via the `DataReader` class:

```
from bead_state_model.data_reader import DataReader

dr = DataReader('simulation_output/data.h5')
coordinates = dr.read_particle_positions()
filaments = dr.get_filaments_all()
```

To get a feeling for the data loaded by the code above, print out the following:

```
print(coordinates.shape)
# prints (21, 4500, 3), if you did not change the default
# number of simulation steps / write frequency or number of
# polymers / beads per polymer.
```

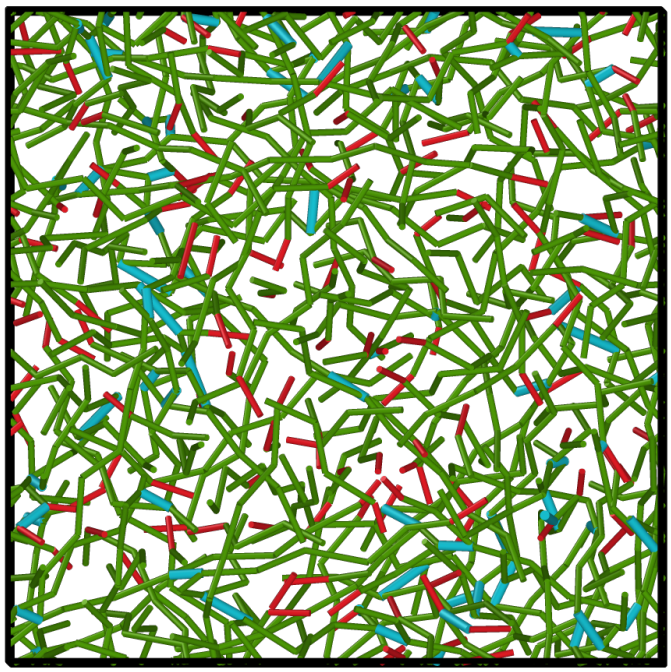

Figure 4.5: 3D visualization of filaments as tubes. Data was generated using the `actomyosin_analyser` Python package. A slice of width 5 bead diameters is shown. Green are inner segments, red are heads/tails of filaments, teal are links.

```
# First axis is number of frames,
# second axis is number of beads,
# third is number of dimensions.

print(len(filaments))
# prints 21, one item per frame.
print(len(filaments[0]))
# prints 300, one item per polymer
print(filaments[0][0])
# Information which beads belong to this filament,
# we will use them later to select coordinates
# from our (21, 4500, 3) array.
```

The following code snippet demonstrates how to access spatial data of individual filaments, by plotting a few randomly selected filaments at frame 0:

```
import matplotlib.pyplot as plt
import numpy as np

# select 10 indices
selected_indices = np.random.choice(
```

```
    np.arange(len(filaments[0])), 10
)

for idx in selected_indices:
    c_idx = coordinates[0, filaments[0][idx].items]
    x = c_idx[:, 0]
    y = c_idx[:, 1]
    plt.plot(x, y, '-o')

plt.xlabel('$x/x_0$')
plt.ylabel('$y/x_0$')

plt.show()
```

The result is a 2D projection of the filament coordinates as shown in figure 4.6. For more recorded data, using the `Analyser` class of the `actomyosin_analyser` package is recommended. The plot was actually created using the `Analyser` class, the code is shown in listing L6 in section A7 in the appendix. Due to

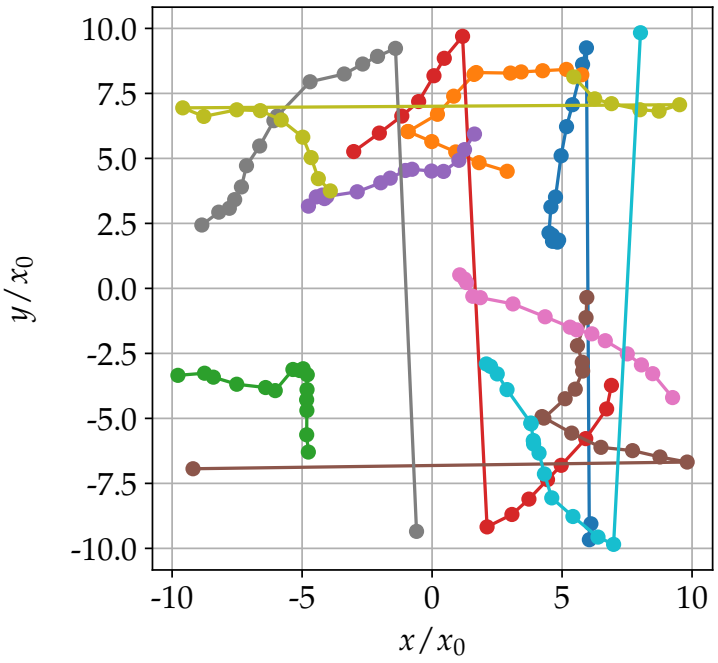

Figure 4.6: 2D projection of 10 randomly selected filaments.

drawing the filaments with lines, artifacts in the form of box-spanning straight lines appear at periodic boundary crossings. How to handle lines at periodic boundaries is discussed in section 3.5.4.

Segment Length Distributions

This section further demonstrates how to handle the simulation data by computing and plotting the segment length distributions, i.e. the distance between direct neighbor beads in all filaments, at the beginning and at the end of the simulation. In addition, it visualizes the different properties of filaments in network generation versus filaments in simulations. The code for the plot can be found in listing L7 in the appendix. The resulting plot is shown in figure 4.7. A difference between the distributions at the initial state (frame 0) versus the

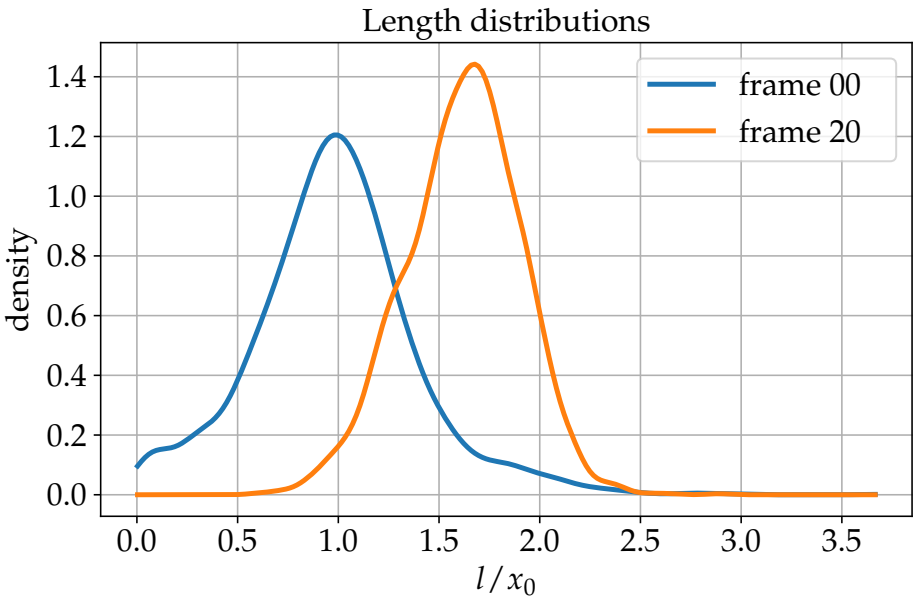

Figure 4.7: Gaussian kernel density estimates of the segment length distributions (distance between neighboring beads of a filament) at frame 0 (equivalent to the initial state) and at the last recorded frame 20.

final state (frame 20) can clearly be seen. The initial angles and distances between beads are drawn from normal distributions that do not take repulsion between neighbours into account. In the simulation, however, repulsion plays an important role in shaping the length distributions. In the plot of the length distributions, it is visible that the lengths are initialized (frame 0) with mean $\mu = 1x_0$, where x_0 is the diameter of a bead. However, due to repulsion between these beads during simulation, overlap is penalized and the distribution quickly becomes skewed and the mean shifts to $\mu > 1$.

This concludes the example. During the example, the user should have gotten an idea of how to create a network, start simulations of the created network, and access data of simulations using the `bead_state_model` Python package. The following chapters demonstrate the application of `bead_state_model` on a larger scale with many repitions of simulations of isolated filaments as well as simulations of filament networks and their interaction with external particles.

Chapter 5

Dynamics of Isolated Filaments

Actin and other biopolymers are commonly described by the worm-like chain (WLC) model [8, 15, 43, 79]. In order to validate that filaments show properties characteristic of worm-like chains (see section 2.2.2), simulations of isolated filaments were analyzed in this chapter. The bending potential governing the stiffness of polymers in the bead-state model (see equation 4.1), has successfully been shown to capture WLC characteristics [43, 64]. However, a key difference to previous studies using this potential, is the repulsion between polymer beads. This repulsion was introduced to enable steric repulsion between filaments in networks, such that entanglement dynamics are included in `bead_state_model` simulations. Due to technical constraints in ReaDDy (the general particle simulation framework that `bead_state_model` is based on), these repulsions also exist between direct neighbor beads within a filament. Hence, hallmark characteristics of WLCs were re-evaluated in this chapter.
Furthermore, this chapter demonstrates polymer dynamicsand what type of data can be analyzed in conjunction with the `actomyosin_analyser` python package.

Simulations were set up to contain one single filament each (availability of simulation data is described in detail in section A1). In this and the following chapters, rather than using the greek letter variables introduced in section 2.1.1, spatial values were specified in units of x_0, temporal values in units of t_0 and energies in units of kT. Where no factors for conversion to physical units are specified, x_0, t_0 and kT denote dimensionless distance, time and energy quantities.
For all simulations in this chapter, the number of beads per filament was set to $N = 25$. The edge length of the cubical simulation box was chosen to be $60x_0$, where x_0 is the diameter of the beads. The large edge length asserts that the filament can't interact with itself across the periodic boundaries.
The full list of default parameters used for the isolated filament simulations is given in section A7 (listing L8) in the appendix. These parameters were used for all simulations in this chapter where not stated otherwise.

5.1 Persistence Length

Stiffnesses of polymers are controlled via the bending force constant k_{bend}. For the bending potential in equation 4.1, the relation between persistence length l_p and k_{bend} is governed by the following identity[43]:

$$k_{\text{bend}} = l_p \frac{kT}{\langle l \rangle}, \tag{5.1}$$

where $\langle l \rangle$ is the average distance between consecutive beads of the filaments. Due to the aforementioned repulsion between beads, $\langle l \rangle$ is greater than the resting length x_0 of the stretching potential. To determine values for $\langle l \rangle$, the distributions of l were evaluated from simulation data for simulations with different k_{bend}. Plots of the distributions are shown in figure 5.1. As shown in figure 5.1,

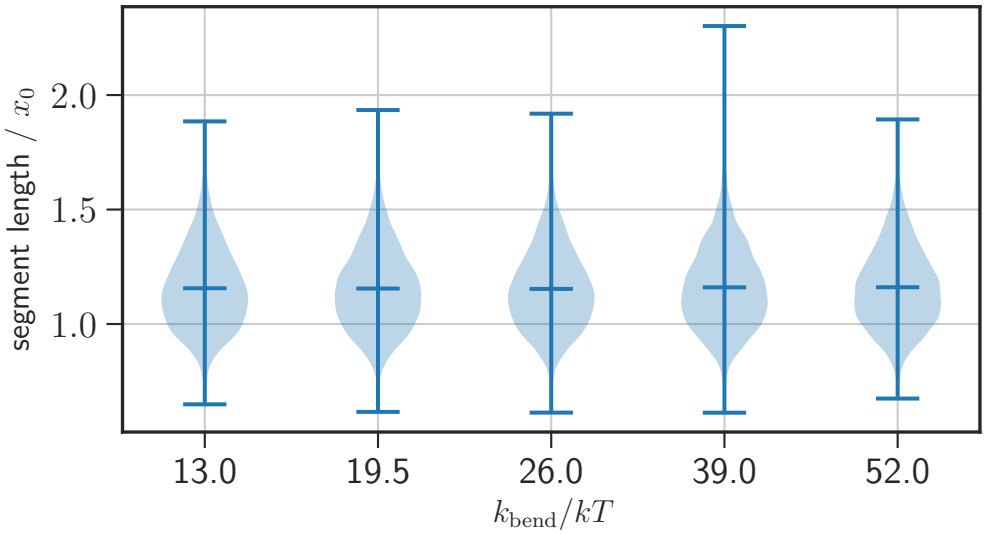

Figure 5.1: Distributions (KDEs with Gaussian kernel) of segment lengths evaluated from 480 filaments for each k_{bend}.

$\langle l \rangle$ is not affected by k_{bend}, hence leading to a linear relationship $k_{\text{bend}} \sim l_p$. Values for l_p predicted via this linear relationship are shown in figure 5.2, where a value of $\langle l \rangle = 1.16$ (and dimensionless $kT = 1$) was used.
Another way to determine l_p from simulation data is by fitting an exponential decay function to the tangent-tangent correlations (see equation 2.7). Figure 5.3 shows the tangent-tangent correlations and the decaying exponential fits for various bending moduli k_{bend}. Results of the fits are plotted into figure 5.2 and agree well with l_p predicted via equation 5.1.
Data in figure 5.3 and figure 5.1 have been evaluated for 120 simulations at 4 independent frames, resulting in 480 contours total, for each value of k_{bend}. For independence across frames, a large interval between those frames was chosen, motivated by the plot of autocorrelations of end-to-end vectors $\mathbf{R}$ (see equation 2.4) in figure 5.4. The autocorrelations drop to 0 at a difference Δ_{frames} of about 4000 to 6000 frames.

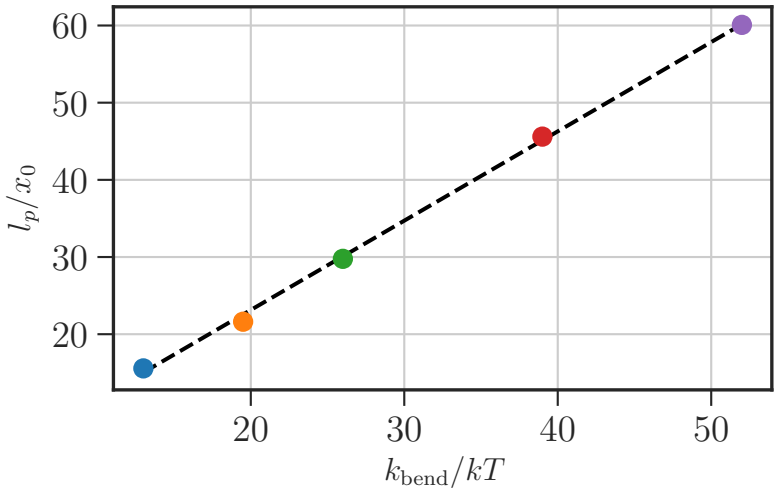

Figure 5.2: Persistence lengths l_p is linearly related to bending force constant k_{bend}. **Black dashed line:** l_p computed using equation 5.1. **Colored markers:** l_p retrieved from exponential decay fits in figure 5.3.

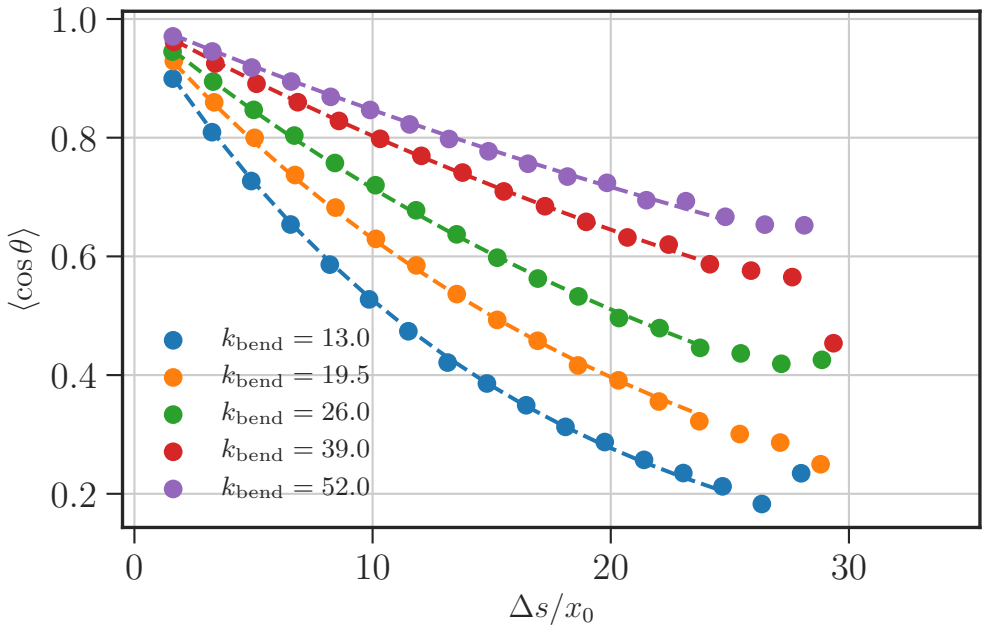

Figure 5.3: Tangent-tangent correlations along the contour of 480 filaments for each k_{bend}. Data were averaged in 17 bins (circles). A decaying exponential was fitted to data with $\Delta s < 25x_0$.

For analyses of the end-to-end distances in the next paragraph and conversion to physical units in the next section, the average contour lengths $\langle l_c \rangle$ are required. The contour lengths of the filaments represented by 25 beads is com-

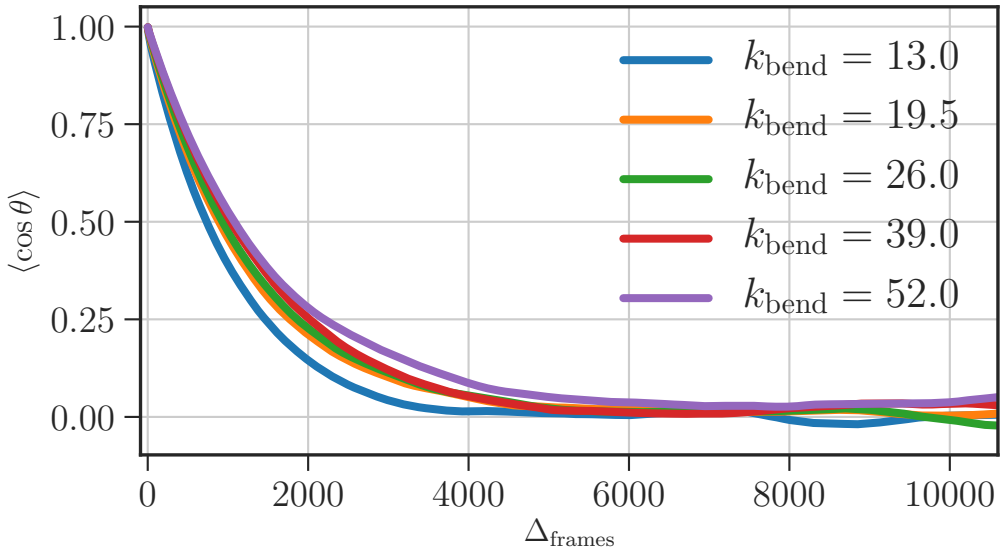

Figure 5.4: Autocorrelations of the orientation of end-to-end vectors **R**. Shown curves are averages over $N = 120$ simulations of isolated filaments for each k_{bend}.

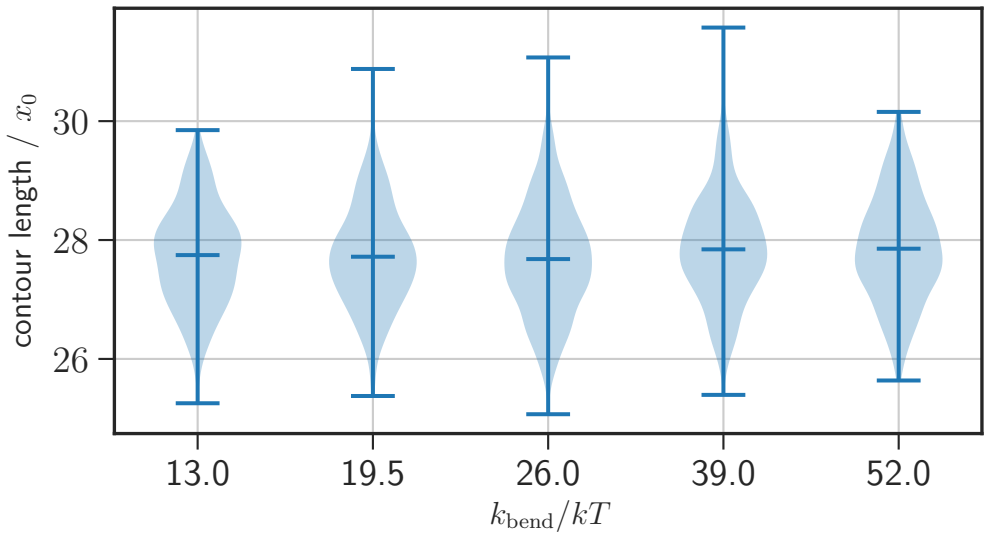

Figure 5.5: Violin plots (with Gaussian KDEs) of contour lengths at different k_{bend}. Mean contour lengths $\langle l_c \rangle$ are marked by central horizontal lines.

puted as the length of the segments r_i (vectors from bead i to bead $i+1$),

$$l_c = \sum_{i=0}^{N=23} ||r_i||.$$

Figure 5.5 shows the contour length distributions for the 480 contours for each k_{bend}. The bending modulus does not have an influence on contour lengths, with averages $\langle l_c \rangle$ ranging from $27.7x_0$ to $27.9x_0$.

Using the persistence lengths from the exponential fits and the average contour lengths computed above, we can use equation 2.8 to estimate the expected average end-to-end distances $\overline{R}$. Figure 5.6 shows the expected $\overline{R}$ and compares

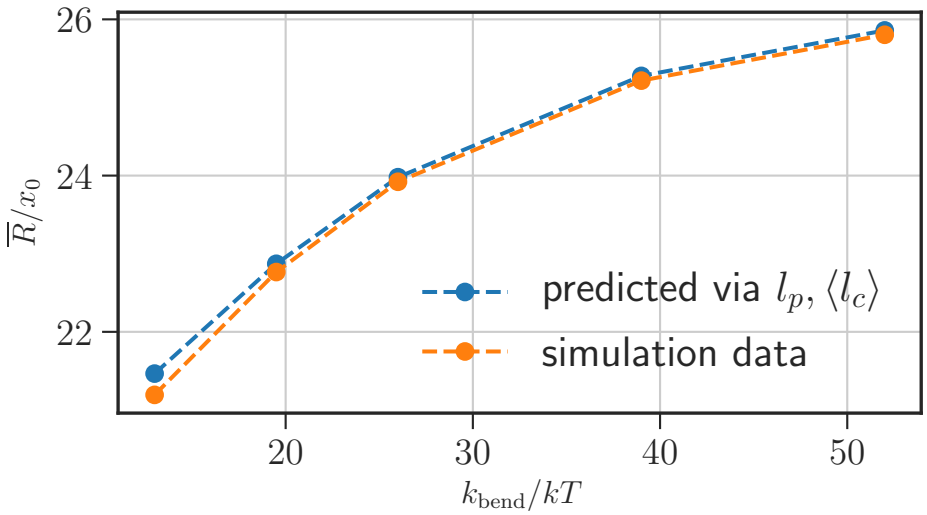

Figure 5.6: Average end-to-end distances predicted using equation 2.8 for WLCs and computed directly from simulation data ($N = 480$ filaments for each k_{bend}).

them to the actual $\overline{R}$ computed from simulation data. Good agreement between both $\overline{R}$ results as well as good agreement between l_p calculated in two different ways shown in figure 5.2, demonstrate that WLC characteristics are well captured by simulated polymers.

5.2 Conversion to Physical Units

`bead_state_model` simulations are not suitable for accurate predictions in physical units. The main reason for that is the coarse-graining, where single polymers consist of bead chains with tens to hundreds of beads. The diameter of biopolymers is usually largely overestimated, since the diameter of polymer beads is rather large. Nonetheless, a rough estimation of conversion factors for length and time is described in this section.
Let us define the polymers represent actin filaments of length 15 µm. With the mean contour length $\langle l_c \rangle$ in internal units (with bead diameter set to unity), we can calculate the conversion factor

$$x_0 = \frac{15\ \mu\text{m}}{\langle l_c \rangle}. \tag{5.2}$$

$\langle l_c \rangle$ are the mean values shown in figure 5.5. Computing x_0 with these values yields $x_0 \approx 0.54$ µm.
This value for x_0 can be used to pick a persistence length matching that of

actin. The persistence length of actin lies between 10 µm and 17 µm according to recent reviews [1, 8]. In the following sections and chapters, a value of $k_{\text{bend}} = 26\ kT$ was used, which leads to a persistence length of $l_p = 29.77$, or $l_p \approx 16.1$ µm in physical units.

From nondimensionalization of the overdamped Langevin equation (equation 2.3), the time conversion factor is defined via the diffusion coefficient, $t_0 = \frac{x_0^2}{D}$. For a rough estimation of the time in physical units, one can use the diffusion of a bead with diameter x_0. With the viscosity of water at room temperature, $\eta(T = 293.15\text{K}) = 0.001002\ \text{kg}/(\text{m} \cdot \text{s})$ [80], matching conditions of actin model systems in experiments, this yields

$$D = \frac{kT}{6\pi\eta r} = 0.794 \cdot 10^{-12}\ \text{m}^2/\text{s}$$

via the Stokes-Einstein equation, with r = $x_0/2$. The resulting time conversion factor is

$$t_0 \approx 0.37\ \text{s}.$$

5.3 Filament Diffusion

Key assumptions of the rouse model are not met by our simulated polymers. The MSD predicted by the Rouse model (see figure 2.3) applies to ideal chains, or very long chains that can be described in their ideal chain limit, with lengths many times that of the Kuhn length (see section 2.2.1). Even though the simulated filaments are far from that criterion (for WLCs the Kuhn length is approximately 2 times the persistence length, $b \approx 2l_p$ [45]), it is a fair expectation that we should see similar regimes in the MSD of a bead of polymers simulated with `bead_state_model`. At low lag times, we will not see the restrictions neighbouring beads impose on the observed bead (regime I), at intermediate lag times the bead has to display subdiffusive behaviour (regime II), at long lag times the bead displays the free diffusion of the whole chain (regime III).
To cover the large lag time range necessary for such a MSD, I conducted simulations of 120 filaments at a time step of `dt=0.001` for 3 different `(n_steps, observation_interval)` parameter pairs:

1. Small observation interval (*SOI*): `(10000, 1)`,
2. intermediate obs. interval (*IOI*): `(300000, 60)`,
3. large obs. interval (*LOI*): `(60000000, 12000)`.

Figure 5.7 shows the MSD of central beads of polymers combined for those three experiments in the upper plot, and the slope p of the MSD on double logarithmic scales (see equation 2.11) in the lower plot. The MSD curve shows qualitatively the expected shape. At low lag time, it follows the free diffusion line, it then visibly steers away from it at around $\tau \approx 10^{-2}t_0$, entering a subdiffusive phase that last up until above $\tau \approx 10^2 t_0$. At $\tau \gtrsim 10^3 t_0$, the MSD curve run nearly parallel to the free difussion line again.
The plot of the slope in the bottom chart of figure 5.7 reveals a fourth regime.

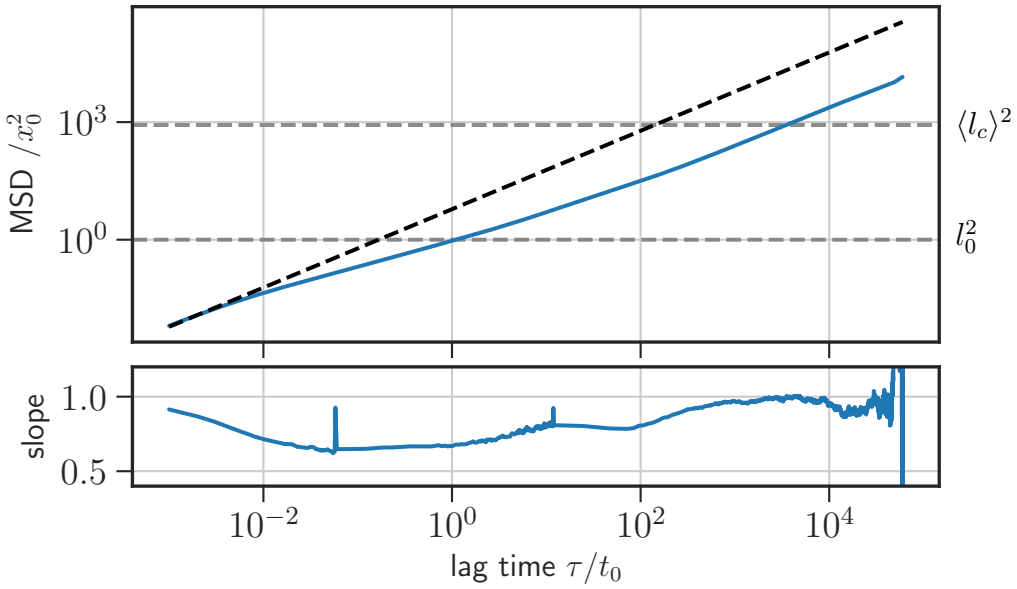

Figure 5.7: MSD of central beads (position 13 of 25) on double logarithmic scale (top) and corresponding slope (bottom) of experiments with 3 different observation intervals combined to cover a large range of lag times τ. Where data from 2 experiments were overlapping, only the data of the experiment with larger observation interval were used. The black dashed line is a hypotheical MSD for a freely diffusing bead.

The intermediate τ range displays two plateaus rather than one, one with a slope of $p \approx 0.65$ for $10^{-1}t_0 \lesssim \tau \lesssim 10^0 t_0$, and another with a slope of $p \approx 0.8$ for $10^1 t_0 \lesssim \tau \lesssim 10^2 t_0$.
The slope curve shows artifacts in form of two spikes at just below $\tau \approx 10^{-1}t_0$ and a smaller one at $\tau \approx 10^1 t_0$, which are artifacts from stitching together the simulations with different lag times. The data before the first peak stem from *SOI* simulations, the data between the two peaks stem from *IOI* simulations. The MSD from *SOI* simulations was cut at the lowest τ of the *IOI* simulations. As described in section 3.2, the larger the lag times in MSD computation, the lower the number of available data points, which usually lead to larger variance. Hence, at the point where the *SOI* MSD and the *IOI* MSD were stitched together, the transition from large fluctuations (low number of data points) to a smooth curve (high number of data points) leads to the spikes in the slope.

In order to further investigate the two plateaus of the slope of the MSD at intermediate lag times, the MSDs of central beads and their slopes of filaments with different bending force constants k_{bend} are shown in figure 5.8. The simulations with varied k_{bend} cover about two decades less in low and high lag times. Nonetheless, the intermediate regime including the plateaus in the slope, starting with slope $p \approx 0.65$ and increasing to slope $p \approx 0.8$, are both present in the covered time scale. The rise from the first to the second plateau in the slopes, however, increases in steepness with increasing k_{bend}. This gives rise to the hypothesis, that the two plateaus in the intermediate lag time regime are caused

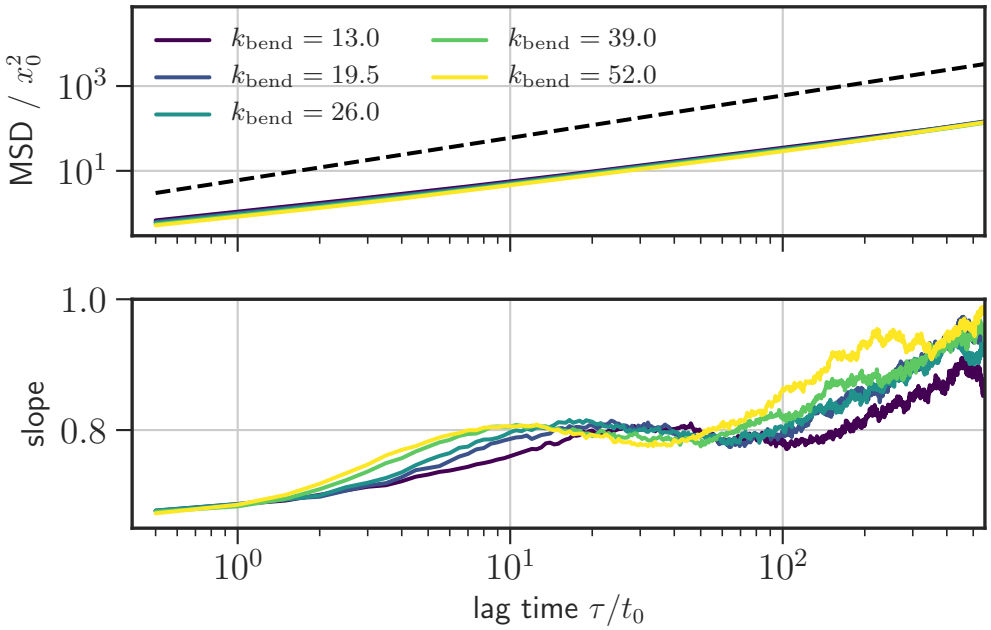

Figure 5.8: MSDs of central beads (position 13 of 25) of filaments with different stiffnesses (different k_{bend}) on double logarithmic scale (top) and corresponding slopes (bottom). Shown MSDs are averages over MSDs of 120 individual filaments. The black dashed line indicates hypothetical free diffusion (if the beads were not part of filaments).

by the two different potentials, U_{bend} and U_{stretch}, restricting the beads' trajectories (see model definition in chapter 4).
The onset of the climb of the slope towards $p \approx 1$ is also visible in figure 5.8. This onset occurs at lower lag times for higher stiffness (higher k_{bend}) as well. At the lag times where the slope is close to 1, the filament moves like one coherent particle. The data indicates that this free diffusion of whole filaments occurs faster the stiffer the filaments are.

5.4 Influence of Time Step on Filament Dynamics

In particle simulations, it is desirable to use time steps as large as possible, that still lead to physical behaviour, in order to minimize the number of simulation steps (one of the main factors for how long the simulations take to run on your computer) required to cover the same simulated time (see also section 3.1). But since the equation of motion is numerically integrated, too large time steps can lead to unphysical situations like two strongly repulsive particles overlapping. This, in turn, can lead to unphysically large forces on these overlapping particles, potentially triggering a chain reaction causing more particles to overlap.
Through trial and error, a time step of $\Delta t = 0.006 t_0$ has been found to be near the upper limit of time steps that reliably lead to physical behaviour in simulated systems with the default force constants ($k_{\text{bend}} = 26, k_{\text{stretch}} = 20, k_{\text{repulsion}} =$

80). Changing the force constants or introducing other interacting particles likely leads to different optimal time steps.

To rule out that polymer dynamics are impacted by the chosen time step, the MSDs and slopes of the outermost beads of polymers were plotted in figure 5.9 for 3 different values of Δt. Outermost beads were chosen, since they display the fastest fluctuations, which would make possible deviations in the dynamics likely best visible for them. Data were combined from the *IOI* and *LOI* experi-

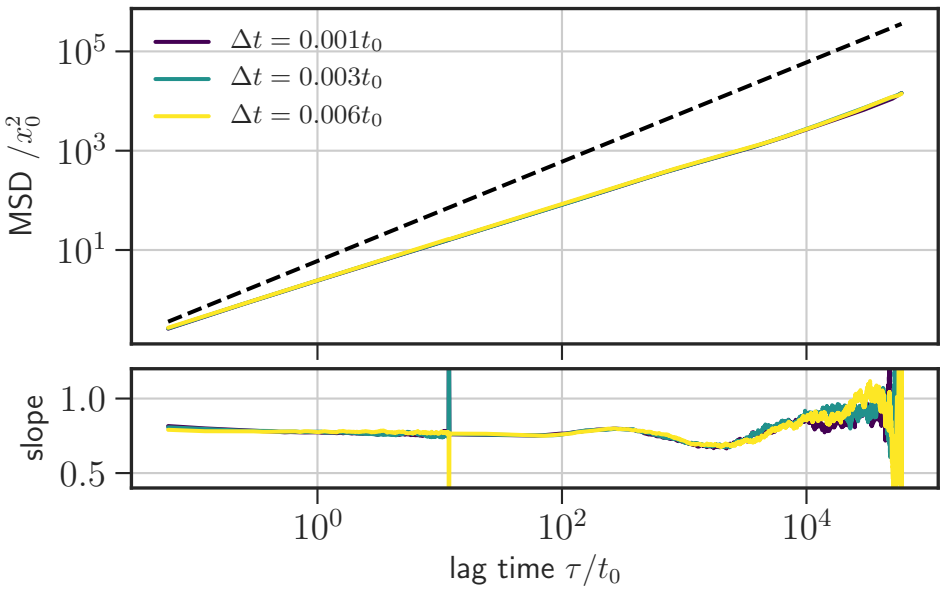

Figure 5.9: MSDs of outermost beads (positions 1 and 25) on double logarithmic scale (top) and their slope (bottom) of simulations with different time steps. Each curve is the average over $N = 120$ filaments. Data were combined from two sets of simulations (experiments), to cover a larger lag time range. The black dashed line is the hypothecical MSD of a freely diffusing bead.

ments of section 5.3. The peaks at $\tau \approx 10^1 t_0$ in the slopes are artifacts of the data combination as explained above. The MSD curves appear to be almost identical for all time steps. This is also reflected in the slopes, which are more sensitive to changes.

5.5 Discussion

Biopolymers in computer simulations are commonly modeled as discrete WLCs [10, 15, 43]. In this chapter, it was confirmed that polymers simulated with `bead_state_model` behave like WLCs, by comparing WLC model predictions on persistence length and end-to-end distances to simulation data (section 5.1).
In other coarse-grained biopolymer simulations, either no repulsion between polymers was defined [10, 11, 15, 43], or repulsion between polymers was realized as repulsion between cylindrical segments [12, 14, 37]. In contrast, in `bead_state_model` repulsion between polymers, which is needed for entangle-

ment dynamics in polymer networks, is realized as repulsion between beads of the bead chains. Due to technical constraints in the underlying ReaDDy framework, this repulsion also applies to direct neighbors in the bead chains. As a consequence the distance between consecutive beads (segment length) is significantly greater than the resting length of the streching potential which was illustrated in figure 5.1.
Another consequence of using beads of the bead chains for steric repulsions, is that diameters of filaments are largerly overestimated in `bead_state_model` simulations. According to the conversion factor estimated in section 5.2, the bead diameter and hence the polymer diameter is $x_0 = 0.54$ µm. I.e., the diameter of actin ($7 - 8$ nm [1, 8]) is overestimated by a factor of around 70. This overestimation might impact the polymer dynamics in `bead_state_model` simulations, and should carefully be considered when converting simulation results to physical units.
Filament diffusion was studied in section 5.3 via the displacement of central beads of the filaments. While, based on the Rouse model (section 2.2.3), three distinct regimes in the MSD were expected, the plot of the slope in figure 5.7 rather revealed four distinct regimes. Instead of one regime at intermediate lag times, two regimes are present. When the slope of filament MSDs was investigated for filaments of different stiffnesses (figure 5.8), an impact of filament stiffness on the onset of the third and fourth regime was found. This might indicate that the two regimes at intermediate lag times are caused by the two different potentials U_{bend} and U_{stretch} both of which constrain the motion of particles.

Chapter 6

Passive Microrheology

One of the reasons to create the bead-state model and its implementations was to establish a simulation framework for various actin model systems. To put the `bead_state_model` simulation framework's flexibility and performance to a test, passive microrheology (MR) simulations were conducted analogous to in-vitro experiments on actin model systems performed by our department [26, 35] and many others. In these experiments microbeads are placed in a homogeneous 3D actin network. Their mean-squared displacement (MSD) is then determined (e.g. using diffusing wave spectroscopy [56] or from trajectories recorded via particle tracking [58]). Similarly, in the simulations presented here (availability of simulation data is described in detail in section A1), a single MR probe bead was placed in the center of the 3D actin network. The MSDs of the MR beads were computed in order to perform a fit with the Paust model (see section 3.3.2), retrieve the complex viscoelastic modulus G^*.
In a first set of simulations, the radius of the MR bead was varied, and the effect of the radius on the resulting plateaus of G' compared to experiments.
In a second set of simulations, the actin concentration was varied in entangled and cross-linked networks, and the dependence of the plateau moduli compared to theory.

6.1 Network Assembly and Simulation Setup

Networks with space for an MR probe bead at the center of the cubical simulation box were generated using the `CreateNetworkSimple` class. The space was kept nearly filament free by placing a spherical repulsive potential at the center, which is taken into account by a Monte-Carlo-like filament bead placement algorithm. More details on how custom potentials can be set up for network generation are described in section 4.1.3.
Keeping the space for the MR bead free of filaments is helpful for the equilibration process, as direct overlaps between filaments and the MR bead can lead to unphysically large forces, potentially leading to artifacts. To start the equilibration process, the MR bead was placed at the center of the networks, and the networks with beads inside were simulated for 5000 steps with very small time steps of $\Delta t = 10^{-6} t_0$ in order to carefully remove any remaining overlap

between filaments and beads.
Example code similar to that used to create the networks, run the equilibration simulations, and after that run the actual simulations, can be found in section A2.

6.2 Varying the MR Bead Radius

Not all sizes are suitable for MR probe beads [31]. When probe beads are chosen too small, the storage moduli G' (see section 2.4) computed from the probe beads' MSDs are underestimated and become bead-size dependent [31]. In order to establish which probe bead sizes are appropiate for MR simulations, simulations with varying probe bead radii were conducted and analyzed. The fol-

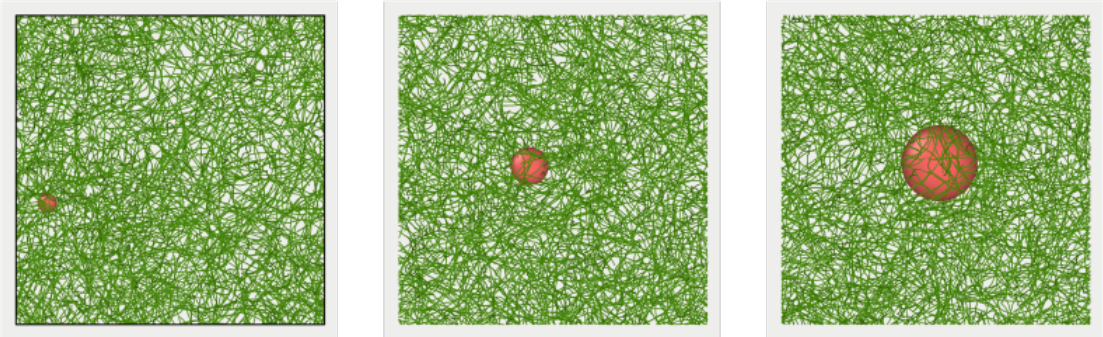

Figure 6.1: Side view of the simulation boxes at $t = 100t_0$ for 3 simulations with different bead radii. The radii are $a = 1.86x_0$ (left), $a = 3.7x_0$ (center), and $a = 7.4x_0$ (right).

lowing simulations were conducted with 800 filaments consisting of 25 beads each, a simulation box with edge length $60x_0$, a time step of $\Delta t = 0.001t_0$ and 4 repetitions for each bead radius a. We varied the bead radius a logarithmically in 4 steps from $1.85x_0$ up to $14.8x_0$. The values had initially been chosen with a preliminary estimation of $x_0 \approx 0.54$ to match radii of ≈ 1 µm up to ≈ 8 µm, respectively. A visualization of simulations with 3 different MR bead sizes is shown in figure 6.1. The systems are shown at $t = 100t_0$ (corresponding to the 100,000th simulated step with step size $\Delta t = 0.001t_0$). The visualization illustrates, that the smaller beads showed faster diffusion, since they moved further away from the center of the box, where they were initially placed, while the largest bead remains nearly at its initial location.
The MR beads in these simulations were "slippery" in the sense that they interacted with the filaments through a purely repulsive potential,

$$V_{\text{slippery}} = \begin{cases} \frac{k}{2}\left(d - a - \frac{x_0}{2}\right) & \text{, if } d < a + \frac{x_0}{2}, \\ 0 & \text{, otherwise,} \end{cases}$$

where d is the distance from a MR bead's center to a filamentous bead's center. To quantify the diffusion of slippery MR beads, the MSDs were computed. Figure 6.2 shows the ensemble MSDs of slippery MR beads. The MSDs are averages over the individual MSDs (see section 3.2) of the 4 repetitions for each bead

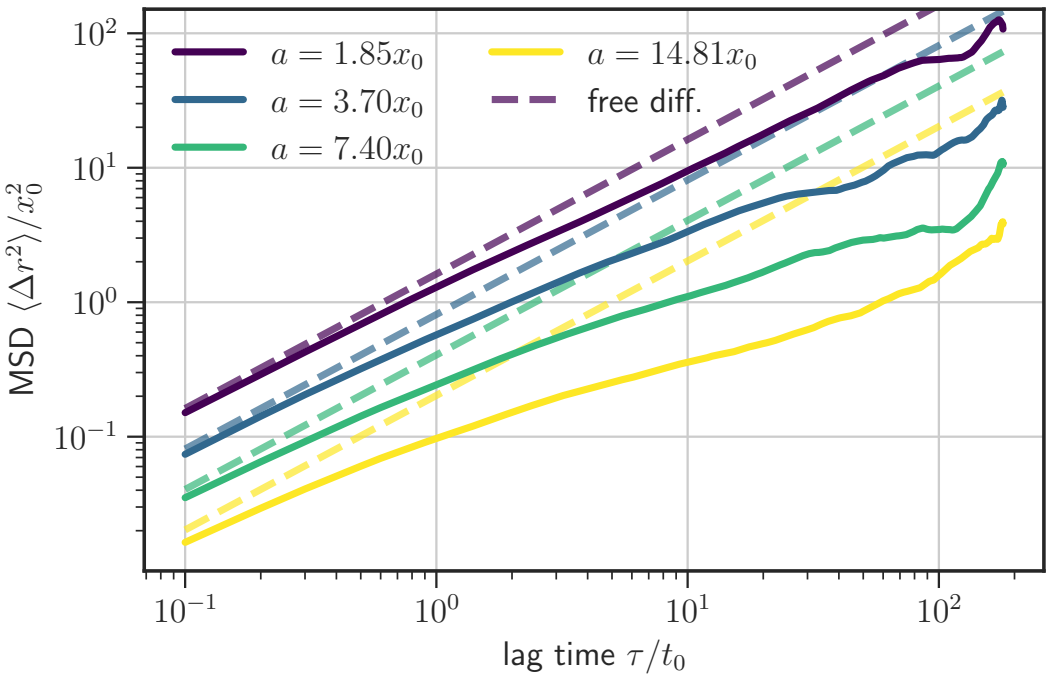

Figure 6.2: Ensemble MSDs of slippery MR beads of different sizes. Each ensemble MSD is the average over 4 individual MSDs. The dashed lines are hypothetical free diffusion MSDs of beads of matching sizes.

radius a. The MSDs display a nearly unhindered diffusion (MSD curve close to free diffusion line) for the smallest bead of radius $a = 1.85x_0$. This indicates that those small beads slip through the pores of the networks. Starting at the next smallest bead of radius $a = 3.7x_0$, the MSDs reveal subdiffusive bead trajectories (curves diverging from free diffusion line), which might indicate that beads of sizes $a \geq 3.7x_0$ are suitable for probing viscoelastic properties of the actin networks.
Figure 6.3 shows the complex shear moduli G^*, estimated by applying the Paust method (section 3.3.2) to the ensemble MSDs in figure 6.2. Plots of the fits applied to the MSDs are shown in the appendix in section A4, and some noteworthy details are described there. The plateau moduli G_0 were evaluated by numerically determining the saddle point ω_{saddle} (see section 3.4) and using $G_0 = G'(\omega_{\text{saddle}})$. $G_0(a)$ is plotted in figure 6.4. The plot shows that the plateau G_0 increases with increasing a.

According to a study by He and Tang [31], a correlation between G_0 and a indicates that the bead size is too small to be suitable for MR analyses (viscoelastic properties of the network should be independent of the probe bead). For slippery beads, the radius a would have to be in orders multiple times that of the mesh size ξ. The mesh size can be estimated using equation 2.12. In the bead-state model, the radius of polymers is $x_0/2$, and the volume of the polymers is the sum of the volume of their beads. This leads to $V_{\text{polymer}} = 800 \cdot 25 \cdot \frac{4}{3}\pi(x_0/2)^3$. The resulting volume fractions are between $\phi(a = 1.85x_0) \approx 0.0485$ and $\phi(a = 14.8x_0) \approx 0.052$ and correspond to mesh sizes between $\xi(a =$

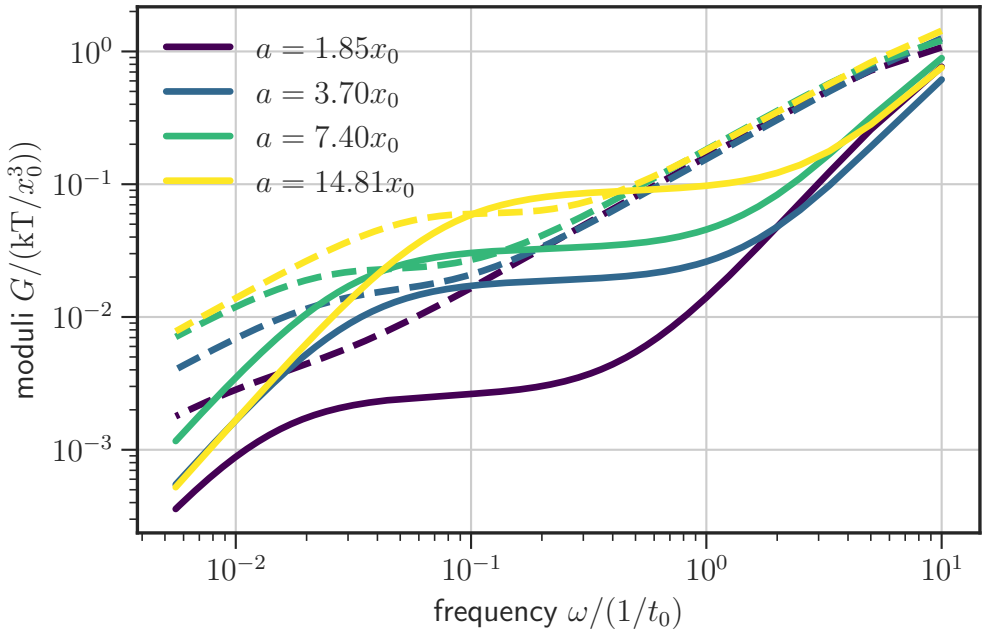

Figure 6.3: G' (solid lines) and G'' (dashed lines) computed by applying the Paust method to the ensemble MSDs in figure 6.2.

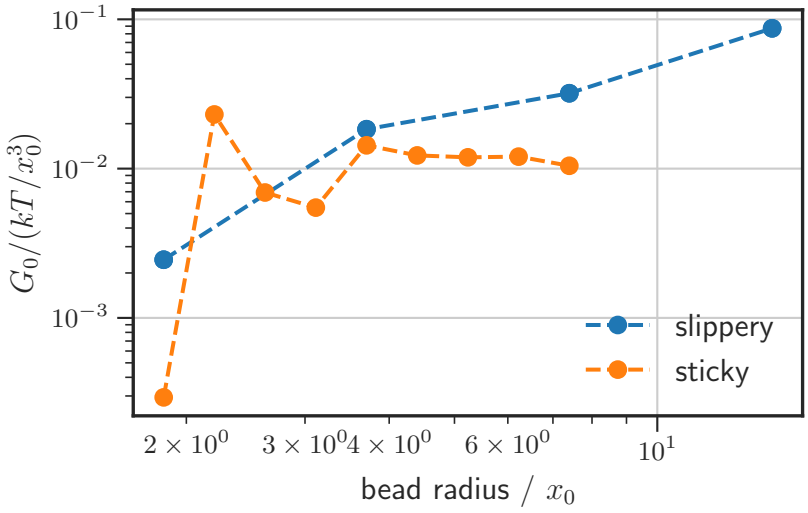

Figure 6.4: Plateau modulus G_0 plotted against the MR bead radius a. G_0 was determined from G' in figure 6.3 (slippery) and figure 6.7 (sticky) by determining the saddle points.

$1.85x_0) \approx 2.27x_0$ and $\xi(a = 14.8x_0) \approx 2.2x_0$. Therefore, the largest bead that was used is only $\approx 6-7$ times the mesh size. To increase the ratio of MR bead size to mesh size, either the MR bead size or the actin density (which would decrease the mesh size) have to be increased. However, increasing the bead size or the actin density much further is not feasible, since that would massively increase the run time of the simulations (see section A3 on performance of MR

simulations).

In contrast to slippery probe beads, sticky beads, i.e. beads that show attractive interactions to the actin network, should be more suitable for MR measurements at bead sizes of the order of the mesh size [31]. Therefore, another set of simulations were conducted, where the interaction between MR beads and filament beads was governed by a potential that repels filaments if $d < a + 0.5x_0$, and attracts them if $a + 0.5x_0 \leq d < a + 0.58x_0$ (see figure 6.5). The potential was defined using the piece-wise harmonic potential provided by the ReaDDy simulation framework [64] (which `bead_state_model` is based on).

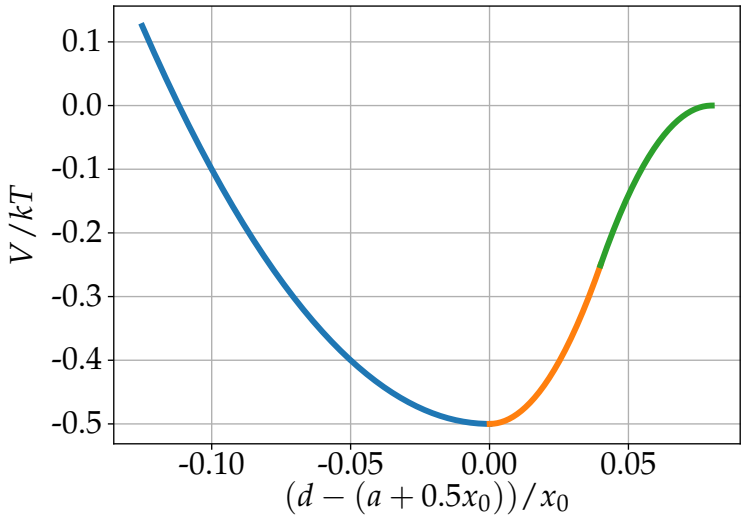

Figure 6.5: Interaction potential of the sticky bead with filament beads, which is repulsive at distances $d < a + 0.5x_0$, and attractive at slightly larger distances.

Due to exponential increase in run time with the MR bead radius (see section A3.1), the radius a was now restricted to a range of $[1.85x_0, 7.4x_0]$, with a finer logarithmic step size. This lead to shorter run times per simulation, and allowed for 16-20 repetitions for each a. The MSDs of the sticky beads are shown in figure 6.6, the complex moduli computed via the Paust fit are shown in figure 6.7.

The MSDs of beads with the two smallest radii displayed behaviour rather close to free diffusion, while those with larger radii entered a more pronounced subdiffusive regime. The shape of the MSDs of the two smallest beads leads to problems in fitting the Paust model, which is to be expected, since it comes to overfitting when only the free diffusion regime is present in the MSD. The fitting of individual MSDs is shown in section A4.1. The plateau modulus for all but the smallest two bead sizes (see G' in figure 6.7) lie within a corridor of one decade and G_0 does not show a correlation to bead radius a for those sizes (see sticky bead graph in figure 6.4). This shows that sticky MR beads with a radius of $a \geq 2.62x_0$ are suitable for probing viscoelastic properties of actin networks with densities similar to or larger than those used here. Therefore, in the following MR simulations with different actin densities, only sticky beads were used.

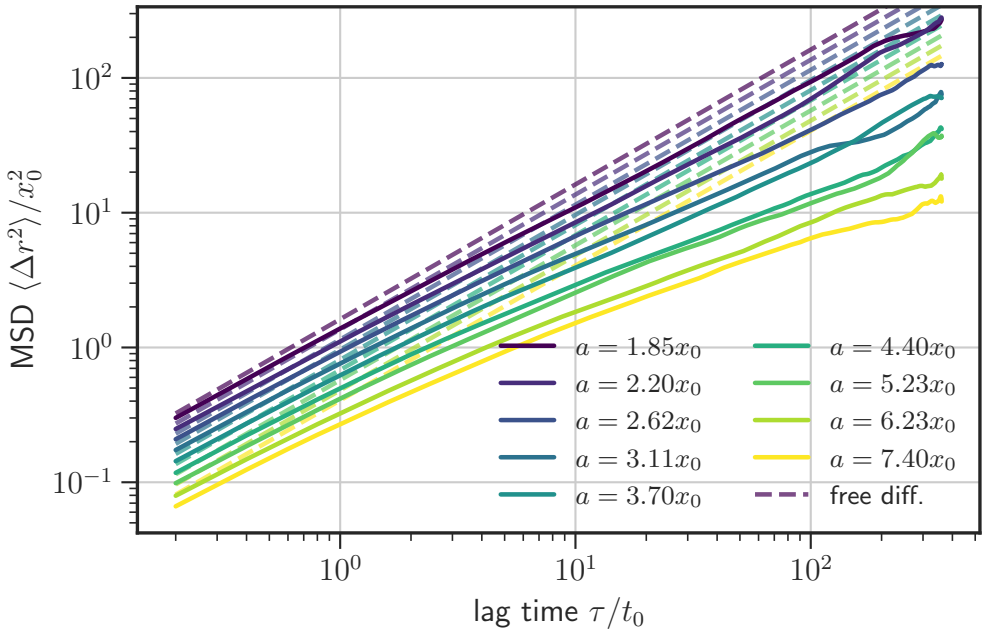

Figure 6.6: Ensemble MSDs of sticky MR beads of different sizes. The dashed lines are hypothetical free diffusion MSDs of beads of matching sizes. Ensemble MSDs with $a = 1.85, 3.7, 7.4$ are averages over 20 individual MSDs, other ensemble MSDs are averages over 16 individual MSDs.

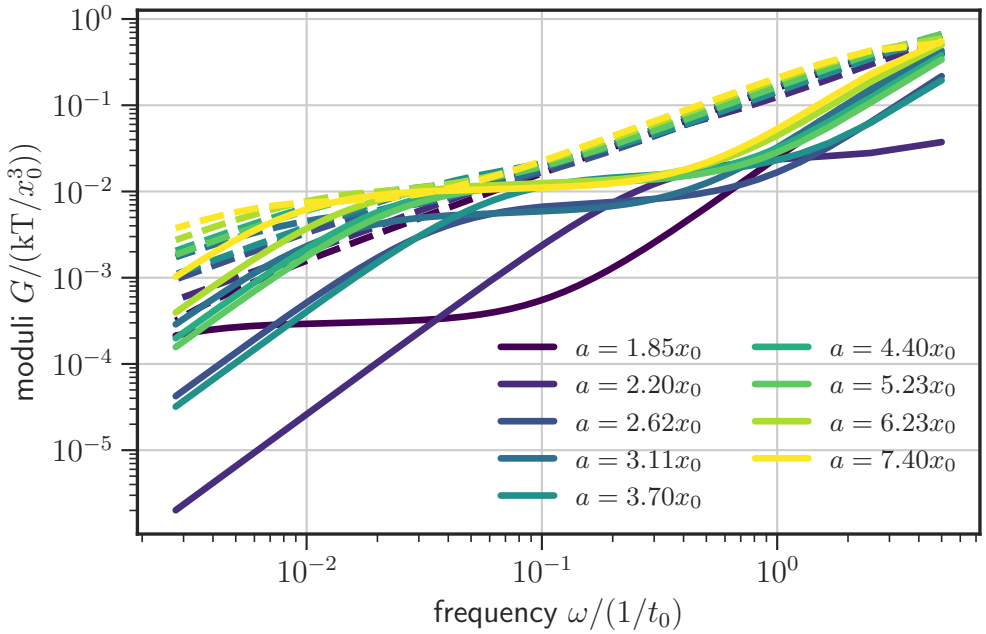

Figure 6.7: G' (solid lines) and G'' (dashed lines) computed by applying the Paust method to the ensemble MSDs in figure 6.6.

6.2.1 Comparison of Paust and Mason Methods

The Paust method and the Mason method are two different ways to compute the complex shear modulus G^* from a MR bead's MSD. More details on the methods are given in section 3.3. In this chapter, the Paust method was used exclusively. However, since the Mason method is more commonly used (for example in [26, 55, 56, 73, 74]), this section compares G^* computed via both methods. My colleague Peter Nietmann kindly evaluated the ensemble MSDs of the slippery beads using the Mason method and provided the final results. For this comparison, spatial and temporal values were converted with $x_0 = 0.54\,\mu\text{m}$ and $t_0 = 0.37$ s, respectively. This puts the MR beads at radii of $a = 1\,\mu\text{m}, 2\,\mu\text{m}, 4\,\mu\text{m}, 8\,\mu\text{m}$. The temperature was set to $T = 293$ K.
The resulting complex moduli are shown in figure 6.8. In figure 6.9, the same curves are shown individually for direct comparisons to their counterparts retrieved via the Paust method. While at low frequency (high lag time in the

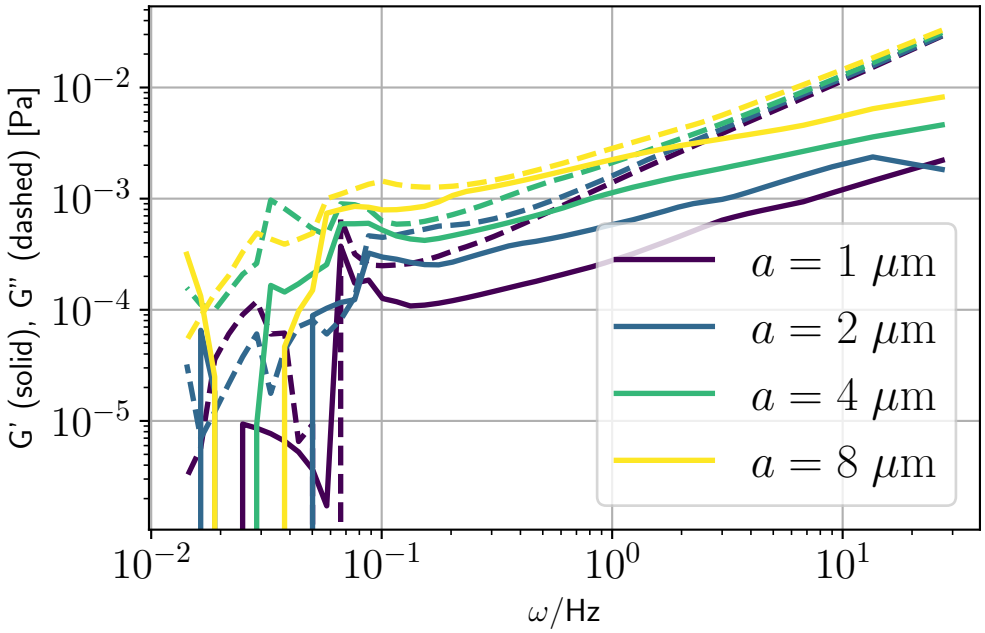

Figure 6.8: G' (solid lines) and G'' (dashed lines) computed by applying the Mason method to the ensemble MSDs in figure 6.2. Data were kindly computed by Peter Nietmann.

MSDs) the fits lead to fluctuating G' and G'' due to insufficient data (number of data points decreases for larger lag times), at values above $\omega = 10^{-1}$ the Mason fits produced consistently smooth curves. Looking at this part, one can see the same order dependent on the bead radius as in the plot produced with the Paust method (figure 6.3). The plateaus can not be determined by fully automatically finding the saddle point, as with the Paust method. Whichever method one were to apply to determine the height of the plateaus, though, one can clearly see an increase in G' with the bead radius a, which would be reflected in the plateau modulus G_0 as well. In other words, evaluating G_0 via

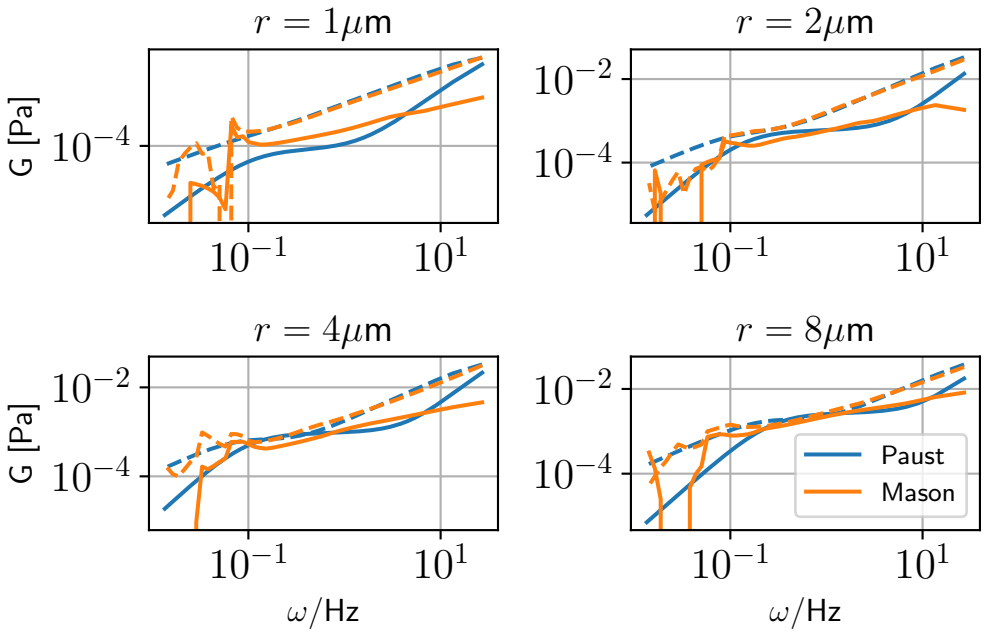

Figure 6.9: Direct comparison of moduli computed via Paust method with moduli computed via Mason method. For each MR bead radius r, the ensemble MSDs in figure 6.2 were used. Data computed with Mason method were kindly provided by Peter Nietmann.

the Mason method leads to the same conclusion as using the Paust method. For its ease of use and robustness, I used the Paust method for all computations of G from ensemble MSDs. To not include complex moduli that are the result of failed fits, all individual fits were inspected (see section A4 in the appendix), and the relative errors of the fit parameters monitored. The plots in figure 6.9 show a direct comparison of G^* produced with Mason and with Paust method for individual bead sizes. At values above $\omega = 10^{-1}$, G'' curves are very similar for both methods. The plateaus in G' do not match for the smallest bead. The plateau is estimated higher in the intermediate frequency regime with the Mason method. The smallest bead did, however, show almost free diffusive behaviour, and might not be suitable for either method. For all other bead radii, the height of the plateaus of G' in the intermediate frequency regimes are almost identical. G' produced with the Mason method displays steeper slopes, starting a bit lower, but crossing G' produced with the Paust method at around $\omega \approx 10^0$. The tails of G' at high frequencies show an increasing slope for the Paust method, but stay rather at the same slope for those produced with the Mason method.
In conclusion, while there are differences between the results produced by both methods, they show good agreement in the plateau moduli, which are mainly analysed in this thesis. Therefore, the more robust Paust method will be used going forward.

6.3 Varying the Actin Density

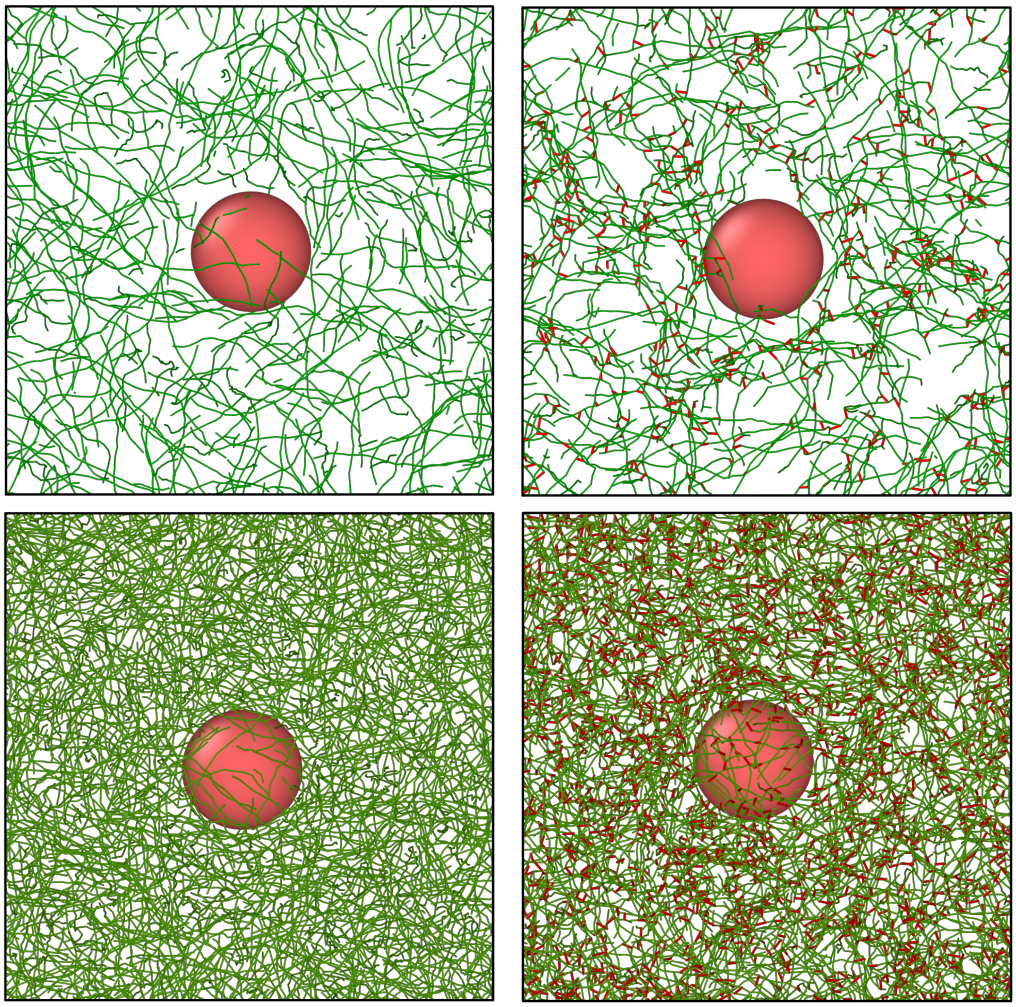

Figure 6.10: 3D visualization of networks with varying actin density and embedded MR probe bead of radius $a = 7.4x_0$, showing actin filaments in green, MR bead in light red, cross-links in dark red. The left-hand side column shows entangled actin networks, the right-hand side column shows cross-linked networks. Rather than all particles/filaments, only those within a slice of width $18x_0$ are shown. Simulations contain $N = 696$ filaments (top) and $N = 2160$ filaments (bottom).

In the simulations in this section, the numbers of actin filaments were varied, i.e. the actin concentration (see figure 6.10). The MR probe bead radius was fixed at $a = 7.4x_0$, the time step was set to $\Delta t = 0.003t_0$ for a part of simulations with lower concentrations and to $\Delta t = 0.006t_0$ for those with larger concentrations. Other parameters were identical to those in the previous section.

The number of filaments N was varied logarithmically ranging from $N = 606$ to $N = 2599$. The ensemble MSDs of the MR beads were calculated (figure 6.11), where each one is an average over $n = 16$ individual MSDs. The MSDs of beads in lower density networks initially display a section, where the

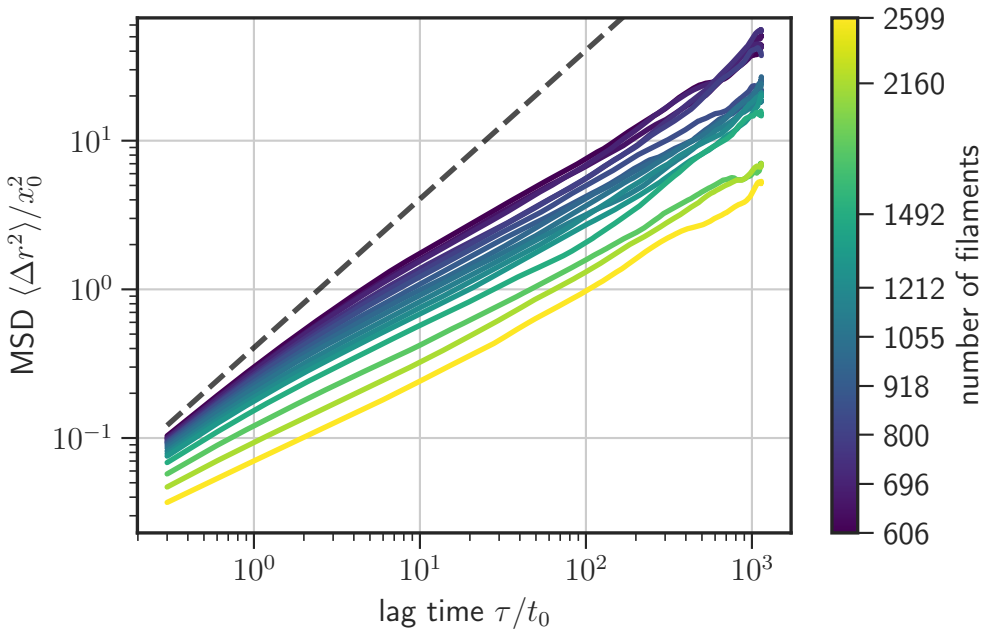

Figure 6.11: Ensemble MSDs of MR beads in entangled networks with varying number of filaments. Each ensemble MSD is the average over 16 individual MSDs. The dashed lines are hypothetical free diffusion MSDs of beads of matching sizes.

curve is near the free diffusion line with slope 1 (dashed line), and then enter into a subdiffusive section at larger lag times. This transition is not visibly covered for at least the highest two densities ($N = 2160, 2599$). Since the Paust fit model is a linear combination of 3 different curve shapes, the quality of the fit might suffer significantly when the MSD shows only one phase, as was the case with the two smallest beads in the previous section. However, inspecting the fit plots and uncertainties of fit parameters in figure A4.4 did not indicate any problems.
Via the Paust fit, the complex moduli were computed (figure 6.12). Due to the number of different N values (16 different values), only half of them are shown in this plot.

A trend of increasing storage modulus G', and with that an increase of G_0, with the number of filaments can be seen. This matches expectations from theoretical work by Morse. The binary collision approximation by Morse [81] based on the tube model [44, 82, 83] predicts the relation

$$G_0 \sim c_A^{7/5}$$

for entangled networks (without cross-links/motors). c_A is the actin concentration, which is equivalent to the number of filaments in the simulations.
As can be seen in figure 6.13, G_0 values computed from the simulations do agree with the predicted exponent of $7/5$.

With identical number of filaments, we repeated the simulations with cross-links that can dynamically form bonds and break bonds between filaments. The

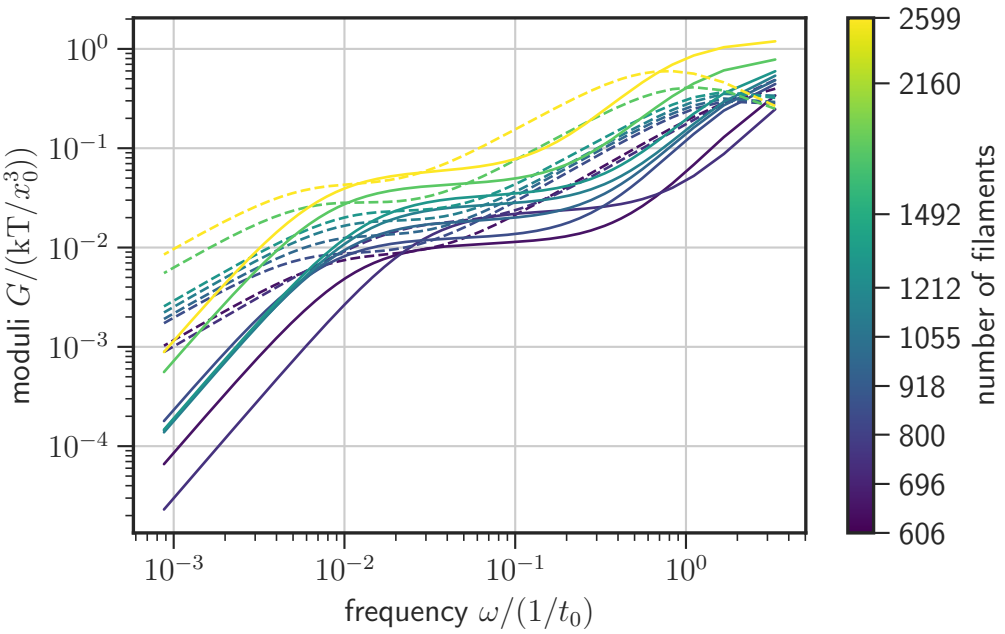

Figure 6.12: G' (solid lines) and G'' (dashed lines) computed by applying the Paust method to the ensemble MSDs of MR beads in entangled networks (figure 6.11).

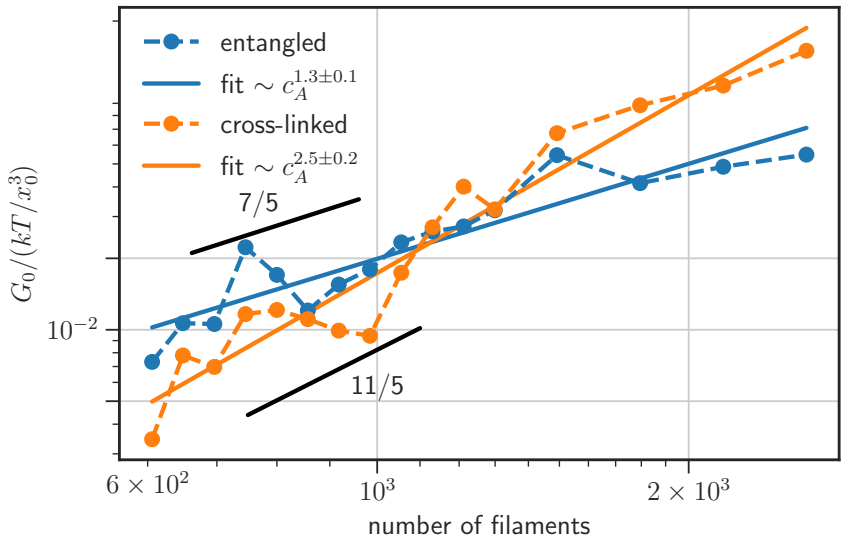

Figure 6.13: Plateau modulus G_0 plotted against the number of filaments, which is equivalent to the actin concentration c_A. Power law fits were performed, to determine the optimum exponent p in the relation $G_0 \sim c_A^p$. Values of p predicted by theory are drawn in as straight lines with slopes $7/5$ (entangled) and $11/5$ (cross-linked).

rate $r_{\mathrm{CL,bind}}$ was set to $2/t_0$, and rate $r_{\mathrm{CL,unbind}}$ to $1/t_0$ (see section 4.1.2). As initial states, the same networks (after removing overlap) as for the entangled

network simulations were used. The maximum number of links was uncapped. On average, it settled at around 3 to 3.5 times the number of filaments, as can be seen in figure 6.14. Note, that the number of links goes slightly super linear with number of filaments, since higher density increases the number of possible binding sites (black dashed line illustrates hypothetical linear increase).

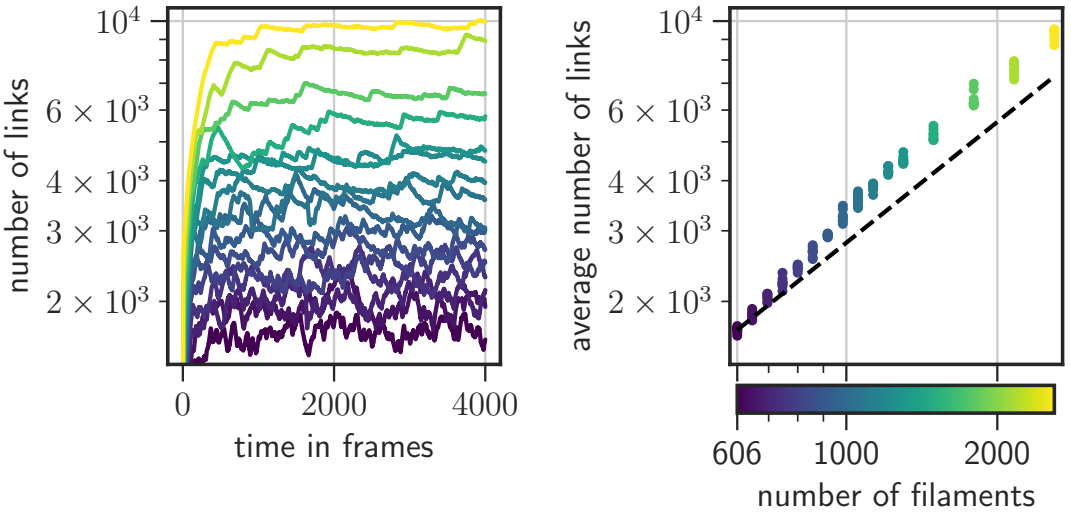

Figure 6.14: (Left) Number of cross-links (number of filament beads with state `motor` divided by 2) plotted at each recorded frame. Lines are from individual simulations, color codes for number of filaments, as per the color bar of the plot on the right. (Right) Time averaged number of cross-links plotted against number of filaments. Averages were computed for individual simulations (multiple data points of same color). The black dashed line indicates linear increase.

Due to simulations running slower with the added interactions of filament linking, the number of simulated steps was reduced to $n_{\text{steps}} = 10^5$. The observation interval was reduced as well to every 25 steps, to cover a similar number of decades in lag times of MSDs, albeit shifted to lower lag times. The time step for the cross-linked network simulations was set to the slightly different value of $\Delta t = 0.005 t_0$ (in entangled simulations it was $\Delta t = 0.006 t_0$).

The ensemble MSDs ($n = 8$) of the MR beads are plotted in figure 6.15. The shift to lower lag times might in part explain the presence of the transition point of the near-free diffusion regime to a subdiffusive regime even for the two highest densities, which was not visible in the entangled networks (figure 6.11). However, taking a closer look at the lag time axis, these transitions are visible roughly around $\tau = 10^0 t_0$ for both $N = 2599$ and $N = 2160$, a lag time that is also covered for the MR bead MSDs in entangled networks. This indicates that MR beads embedded in cross-linked networks are a little less hindered in their diffusion, and that pockets in the network are larger on average compared to entangled networks. This could already be suspected from the overview visualization in figure 6.10. An intuitive explanation for this is that links reduce the distance between filaments, leading to clusters of high filament density, which in turn leads to more space (larger pockets) between these clusters.

The ensemble MSDs have been fitted with the Paust model (individual fit re-

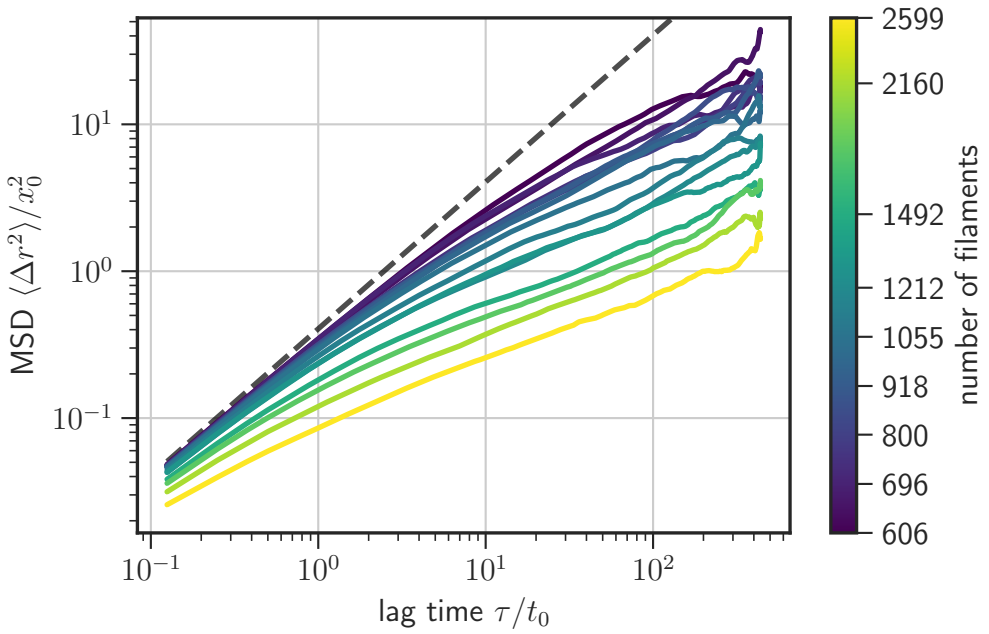

Figure 6.15: Ensemble MSDs of MR beads in cross-linked networks with varying number of filaments. Each ensemble MSD is the average over 16 individual MSDs. The dashed lines are hypothetical free diffusion MSDs of beads of matching sizes.

sults in figures A4.5 and A4.6) and half of the resulting viscoelastic moduli are shown in figure 6.16. In the storage moduli G', we again see an increase in height with number of filaments. The plot of G_0 and power law fit in figure 6.13 show a steeper increase compared to the entangled networks. Plateau moduli G_0 are lower for cross-linked networks with low density, but exceed those of entangled networks at around $N \approx 1100$. The steeper increase is in agreement with theory by MacKintosh et al. [84], which predicts the relation

$$G_0 \sim c_A^{11/5}.$$

While their theory initially predicted this relation for dense entangled networks that are affine under deformation, it was since then rather used in the context of cross-linked networks (e.g. in [22, 85–87]). The power law fit in figure 6.13 leads to a slightly higher exponent of 2.5 ± 0.2.

6.4 Discussion

With the MR simulations in this chapter, it was demonstrated that `bead_state_model` is capable of simulating large actin networks and that its flexibility allows for construction of model systems with actin, links and external components.
As shown in section 6.2, the dynamics of MR beads of different sizes and different surface properties embedded in polymer networks match the dynamics

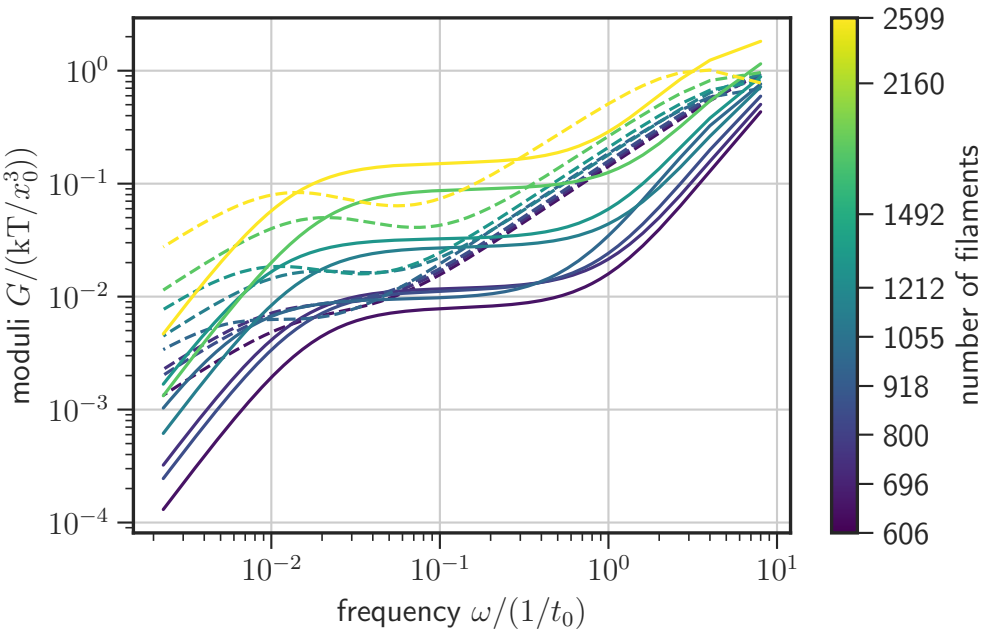

Figure 6.16: G' (solid lines) and G'' (dashed lines) computed by applying the Paust method to the ensemble MSDs of MR beads in cross-linked networks (figure 6.15).

observed in experiments [31]. This implies that MR beads with purely repulsive interactions with polymers (slippery surface) are not suitable for probing networks with mesh sizes of the same order of the bead size. Whether much larger (many times the mesh size) slippery beads are more suitable has not been studied, since the performance of the simulations is highly impacted by the size of the largest particle interaction radius (see run times in figure A3.1). Since the MR set up adds only one external (non-filament) particle species (the MR bead) to the simulation box, we deal with a bidisperse system. The performance of these systems could be largely improved by the implementation of hierarchical grids (see section 3.1.5) in the ReaDDy2 framework.
Sticky MR beads (with a short range of attractive interactions right outside of the repulsive interaction range) have proven more suitable for probing viscoelastic network properties through MR in simulations.
In section 6.3, sticky MR beads were used to study the plateau modulus G_0 in dependence of the network density. These simulations revealed exponents in the G_0-power-law that are quite close to theoretical predictions. This is true for entangled (computed: 1.0 ± 0.2, predicted: 1.4), as well as cross-linked networks (computed: 2.0 ± 0.1, predicted: 2.2). How well these theoretical values predict exponents determined through in vitro networks is an ongoing matter of research [22, 35, 88, 89]. The exponents determined through simulations are at the lower end of ranges found through in vitro experiments for entangled networks (2.1 ± 0.8 [35], 1.3 ± 0.3 [89], 1.8 ± 0.4 [88], 1.4 [24], 1.2 ± 0.2 [60]), and close to the value of 2.2 found for cross-linked networks [22].
The effect of cross-links was studied with one set of parameters only, even

though several cross-link parameters were found to have different impacts on viscoelastic properties of networks [1, 90]. One parameter directly affecting the plateau modulus is the cross-link concentration which has been studied in networks isotropically cross-linked with heavy meromyosin [22]. Simulations in this chapter were conducted with uncapped numbers of links and static binding rate. Using static binding rates was the only possibility for spatial reactions in ReaDDy [64] (see section 4.1.2). In the meantime, means to allow finer control of the number of links have been implemented, made possible by a feature contribution to the open source ReaDDy framework[1]. The user can now choose to set a cap to the number of links. Defining this cap automatically causes the rate $r_{\text{CL, bind}}$ to dynamically decrease with the number of cross-links present. Using this cap parameter, one could study the effect of varying cross-link numbers on G_0 in `bead_state_model` simulations.

To my knowledge, no simulations studying passive MR in polymer networks have been published prior to this thesis (active MR simulations have been studied in [91]). However, multiple studies investigating rheology of actin networks through global shear deformation exist, for example by Teayoon Kim and coworkers [12, 18, 92]. One of those studies introduces a method called segment-tracking, which is inspired by microrheology. In segment-tracking, trajectories of segments of filaments rather than trajectories of MR beads [12] are used to determine G^*. Employing segment-tracking in `bead_state_model` simulations could be an option to study viscoelastic properties of networks without the need for large MR beads that severly increase the simulation run times. Performing passive MR simulations should be possible with Taeyoon Kim's simulation code as well. In another study, they have demonstrated its capabilities to include large spheres that interact with the actin network [93]. However, their simulation code is not publicly available, and thus did not pose an option for MR simulations in this thesis.
A popular and publicly available[2] framework for simulations of actin and other components of the cytoskeleton is cytosim [11, 17, 38, 94]. Utilizing built-in features, one can configure spherical particles [11] that can serve as passive MR probe beads. An attempt to replicate the entangled network simulations of this chapter is discussed in section A5 in the appendix. Section A5 concludes that although MR studies via cytosim are possible, simulations perform better in `bead_state_model`, and cytosim would require larger beads or decreased mesh size (both of which would likely decrease performance further) in order to probe viscoelastic properties with MR beads (how MR bead size and mesh size can affect measurements was described in section 6.2)

[1] Pull request #192: `https://github.com/readdy/readdy/pull/192`
[2] Project page: `https://gitlab.com/f-nedelec/cytosim`

Chapter 7

Indentation and Relaxation

A common way to experimentally probe the mechanics of actin networks or cells, is indentation with an atomic force microscope [25, 95–97]. Inspired by atomic force microscopy experiments, this chapter presents simulations (availability of simulation data is described in detail in section A1) where a single large spherical indenter was pressed into an actin network layer (see overview in figure 7.1). Microspheres directly exerting strain to actin networks have been studied experimentally before, e.g. by Gurmessa et al. [24]. In contrast to their

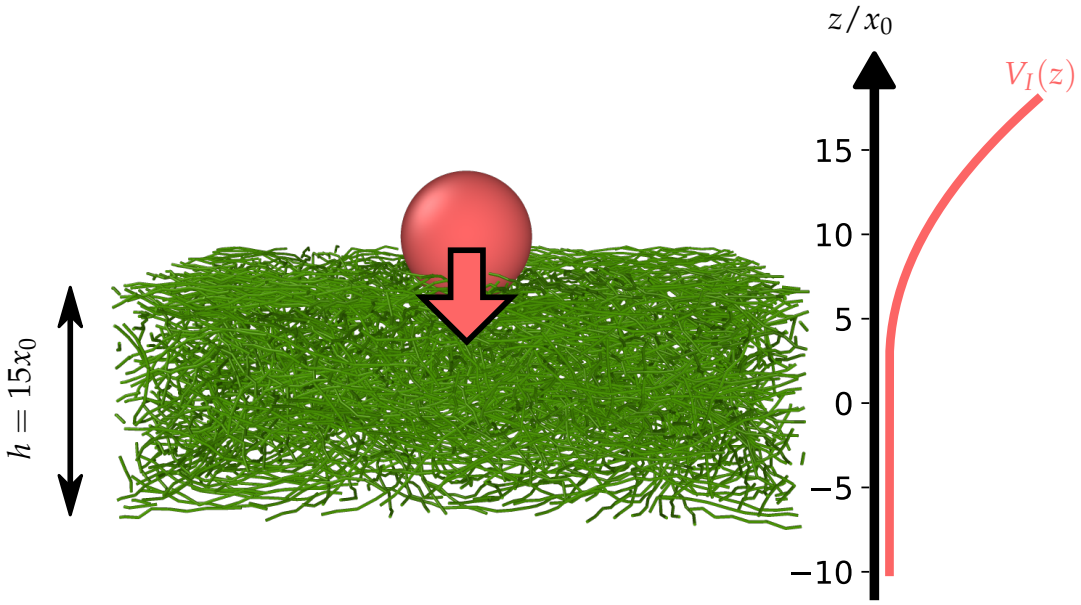

Figure 7.1: 3D visualization of the indentation simulations. A spherical indenter of radius $a = 5x_0$ (where x_0 is the diameter of the beads in the bead chains that represent actin polymers), under the influence of a truncated harmonic potential $V_I(z)$, presses downwards into an actin network confined to a layer. The layer has a height of $h = 15x_0$. In x and y directions, periodic boundaries are used.

setup, where the bead was embedded in the actin network, in the simulations the sphere was placed outside the network initially and then pushed into the network. In order to resemble the shape of actin cortices, the actin network was confined to a layer. Straining of actin layers via the indenter/microsphere and subsequent relaxation of the networks were studied for networks with different conditions, which were chosen to resemble biochemical conditions that enable or disable processes like treadmilling, cross-linking or myosin motor steps.

7.1 Network Assembly

Without the indenter present, 8 networks were generated within the boundaries of the layer of height $15x_0$ (figure 7.2 A), using the algorithm described in section 4.1.3. Simulations of the networks were run for 30000 steps at $\Delta t = 0.005t_0$ to allow the networks to relax. For each network, these relaxation simulations were run without cross-links ($r_{\text{bind}} = 0/t_0$) as well as with cross-links ($r_{\text{bind}} = 2/t_0$) (figure 7.2 B). The number of links/motors was capped at 1000 (i.e. 2000 filament beads with state `motor`). A potential V_{actin} was applied to the network, which effectively keeps the filaments inside the layer in z direction:

$$V_{\text{actin}} = \begin{cases} k_{\text{box, actin}}(z - 7.5x_0)^2, & \text{if } z \geq 7.5x_0, \\ 0, & \text{if } -7.5x_0 < z < 7.5x_0, \\ k_{\text{box, actin}}(z + 7.5x_0)^2, & \text{if } z \leq -7.5x_0 \end{cases}$$

Periodic boundaries were set in x and y directions. These networks were used as initial states for five different types of systems: entangled networks (*E*), entangled networks with treadmilling (*T*), statically linked networks (*SCL*), dynamically linked networks (*DCL*), and active networks with motors (*A*) (see figure 7.2 C).

The binding rate for the links was set very high at $r_{\text{bind}} = 2.0/t_0$, and the maximum number of links was capped at $n_{L,\text{max}} = 1000$ (i.e. 2000 link particles). With this setting, the number of links remains at the niveau of $n_L = 1000$ at almost all times. In the case of dynamic unbinding and binding (conditions *DCL*, *A*), this means that when a link bond is broken, a new link bond will be formed right within the same or the next time step, and the number of links will stay at (or just below) $n_L = 1000$. However, the new link bond will likely be created at a different location and between different polymers than the one that had been broken, as there are many more possible binding sites than the 1000 links can cover.

7.2 Forces on the Indenter

The indenter is driven by a potential

$$V_I(z) = \begin{cases} \frac{k_I}{2}(2.5 - z)^2 & \text{, if } z > 2.5 \\ 0 & \text{, else,} \end{cases}$$

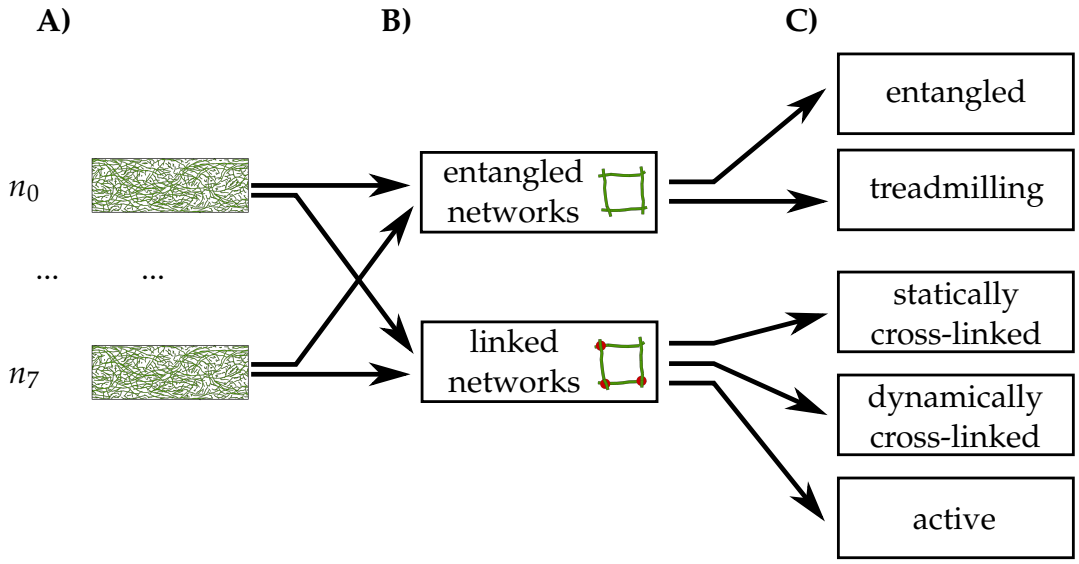

Figure 7.2: Generation of initial conditions. **A)** Layers of actin networks generated, repeated 8 times. **B)** 8 networks were simulated for a short period under 2 different conditions (16 simulations total): without any links (entangled), and with link formation activated. The 16 final states served as initial states for 5 different conditions. **C)** 8 entangled networks from B) served as initial states for conditions entangled and treadmilling. 8 linked networks from B) served as initial states for conditions passively cross-linked (no unbinding), dynamically cross-linked (unbinding and binding), and active (links are motors, i.e. they can bind, unbind and perform steps).

which is illustrated in the overview in figure 7.1. The indenter is placed in contact right above the actin layer. Moving into the layer, the indenter starts feeling a repulsion by the filaments, with which it interacts through

$$V_{\text{repulsion}} = \frac{k_r}{2}(d - ||r||)^2, \tag{7.1}$$

if the distance $||r|| = ||\mathbf{r}_b - \mathbf{r}_I||$ between filament bead (r_b) and indenter (r_I) is smaller than $d = a + x_0/2$ (indenter radius a plus the filament radius $x_0/2$).
To estimate the average repulsion force F_r exerted on the indenter by all filaments, the equation of motion (EoM) of the indenter for the repulsion force was solved. The EoM is the overdamped langevin equation (equation 2.2). The z-component of its discrete form can be written as

$$\gamma \frac{\Delta z}{\Delta t} = F_{\text{tot}} + F_{\text{thermal}}.$$

When we are looking at Δz from one frame to the next, F_{thermal} will be the sum over 50 values from a Gaussian distribution with zero mean, since frames were recorded every 50 simulations steps. We will neglect this thermal contribution, as $\langle F_{\text{thermal}} \rangle = 0$. The remaining total force F_{tot} is comprised of two terms, $F_I = -\frac{\partial}{\partial_z} V_I$ and F_r, the contributions on the indenter due to the repulsion between

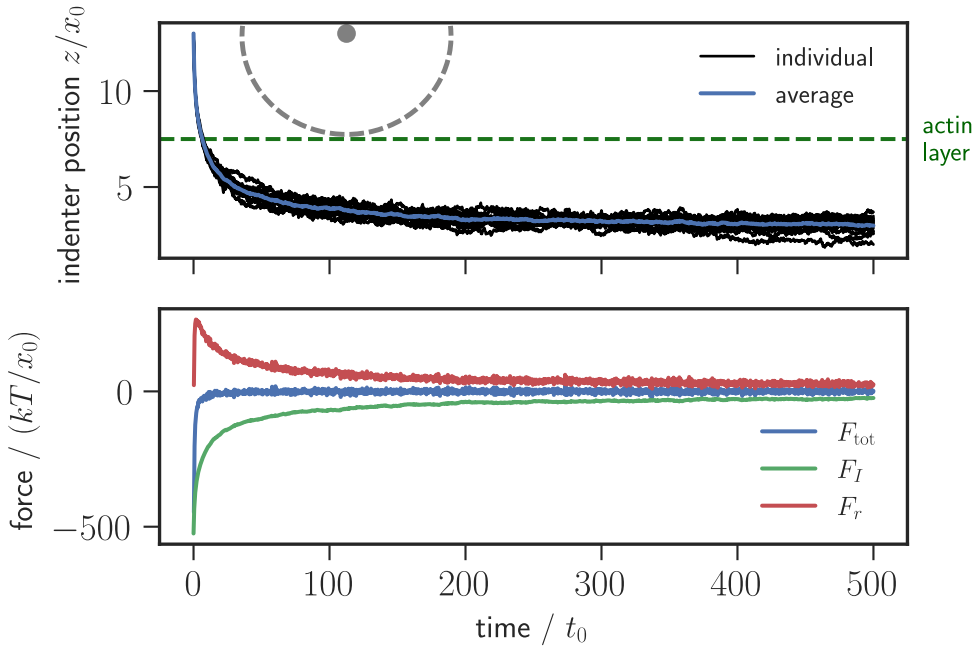

Figure 7.3: **(Top)** z-position of the indenter plotted against time. Thin black lines are individual positions of indenters moving into entangled actin layers, blue line is the average over the $n = 16$ individuals. Indenter and layer are within touching distance initially, which is indicated by the gray (indenter position plus radius) and green (upper boundary of the actin layer) dashed lines. **(Bottom)** Average forces acting on the indenter. The total force F_{tot} is estimated from the position change Δz from one frame to the next using equation 7.2. Subsequently the repulsion force F_r is computed as $F_{\text{tot}} - F_I$.

indenter and network. Solving for F_r yields

$$F_r = F_{\text{tot}} - F_I = \frac{kT}{D}\frac{\Delta z}{\Delta t} - F_I. \tag{7.2}$$

To further smoothen the computed F_r and justify the neglection of F_{thermal}, the average z position of $n = 16$ simulations is used to compute the ensemble averaged Δz increments. An example of the introduced variables is shown in figure 7.3 for the 16 repetitions with the entangled network.

In later iterations of indentation simulations, the force on the indenter was recorded directly. A comparison between average total force estimated via equation 7.2 and recorded force on the indenter shows good agreement between the two methods (see figure 7.4).

7.3 Indenter Positions and Repulsion Forces for Different Networks

Figure 7.5 shows a plot of average indenter heights for the different conditions at the top, and repulsion forces computed from the positions via equation 7.2 at

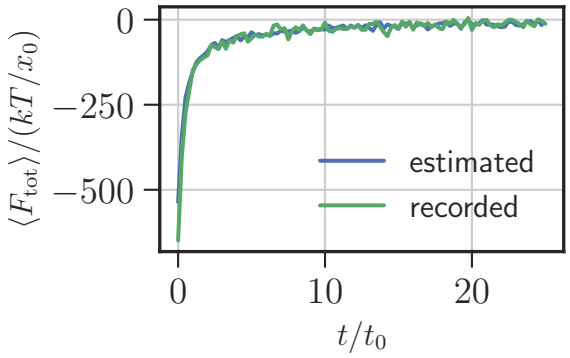

Figure 7.4: Comparison between recorded average total force and average total force estimated via equation 7.2 acting on indenters pressing into an entangled actin layer.

the bottom. Due to repulsion between filaments and indenter, the equilibrium

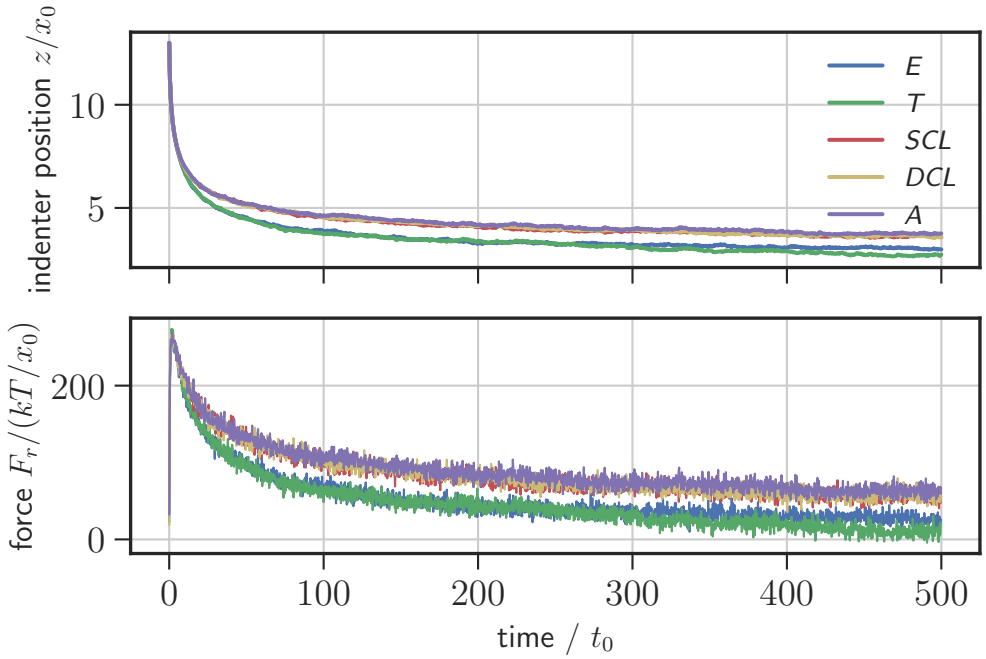

Figure 7.5: Average z-position (top) and repulsion force (bottom) of indenters in simulations of different unlinked (*E*, *T*) and linked (*SCL*, *DCL*, *A*) network conditions. Averages are composed of $n = 16$ individual simulations for each condition.

point of the indentation potential (at $z = 2.5x_0$) is never reached by the indenter. The indenter entered deeper into the network layer for entangled networks (*E*, *T*), than for the linked networks (*DCL*, *SCL*, *A*). Within the group of the linked networks, the layers showed similar resistance against indentation. All linked conditions *SCL*, *DCL*, *A* seem to resist the indentation for identical amounts, leading to almost identical curves for indenter position and repulsion force.

The same can be observed for the two unlinked polymer conditions E and T.

7.4 Polymer Density Under the Indenter

In simulations we have full information on all polymers. This allows us to visualize the impact the indenter has on the network layer with fine-grained local densities. Network densities were estimated via kernel density estimation (KDE) with Gaussian kernels using SciPy's [98] `gaussian_kde` function. A color coded plot of densities of the entangled layer estimated at different points in time on a 81x33x33 grid is shown in figure 7.6 (left).

Densities were averaged over the $n = 16$ entangled layers. The grid spans an x-range of $x \in [-20x_0, 20x_0]$, which is not the full extent of the box ($x \in [-22.5x_0, 22.5x_0]$), to exclude artifacts in the density estimation at the border. The y and z-ranges are $[-8x_0, +8x_0]$. In addition, the voxels for which the kernel (bandwidth ≈ 0.243, resulting in an extent of the kernel of $\pm 1.2x_0$, beyond which the kernel is approximately 0) lies completely within the indenter were removed from the density estimate. The voxels for which the kernel overlapped with the indenter were corrected, s.th. the numerical integral over the kernel over the part of the volume that does not overlap with the indenter yields unity. I.e., the overlapping volume was treated as unphysical volume and was excluded from the density estimate.
For the 2D color coded representation of the densities, the densities were averaged over all 33 voxels in y-direction, and the z-limits were set to $[-5.5x_0, 5.5x_0]$ for the plots. Visualized via color coding, one can identify where the network is displaced and compressed by the indenter resulting in an increased density under the indenter (bright yellow/white). Just next to the indenter, the density is decreased (dark purple). This is likely due to the entanglement of the polymers: as the indenter pushes the polymers right below it, these polymers will pull down other polymers they are entangled with as well.
Since the spatially different density can be hard to see with color alone, density profiles are shown in the right column of figure 7.6. The profiles show mean densities over x ranges covered by the indenter (using grid points in $x \in [-5x_0, 5x_0]$) at different heights. The density profile at $t = 0t_0$ (top right) is used as a reference *normal* line, shown as dashed lines in the density profile plots at later points in time.
The density profiles again show the decrease in mean densities where the indenter pushes into the network. The displaced polymer is pushed downwards into the layer leading to an increased density right below the indenter. This can be seen well for the profile at $t = 2.5t_0$ (where the blue solid line is right of the dashed blue line for most of the z-range), and at the lower z values for $t = 25t_0$. At $t = 250t_0$ the density below the indenter is almost back to normal (blue solid line close to blue dashed line at low z).

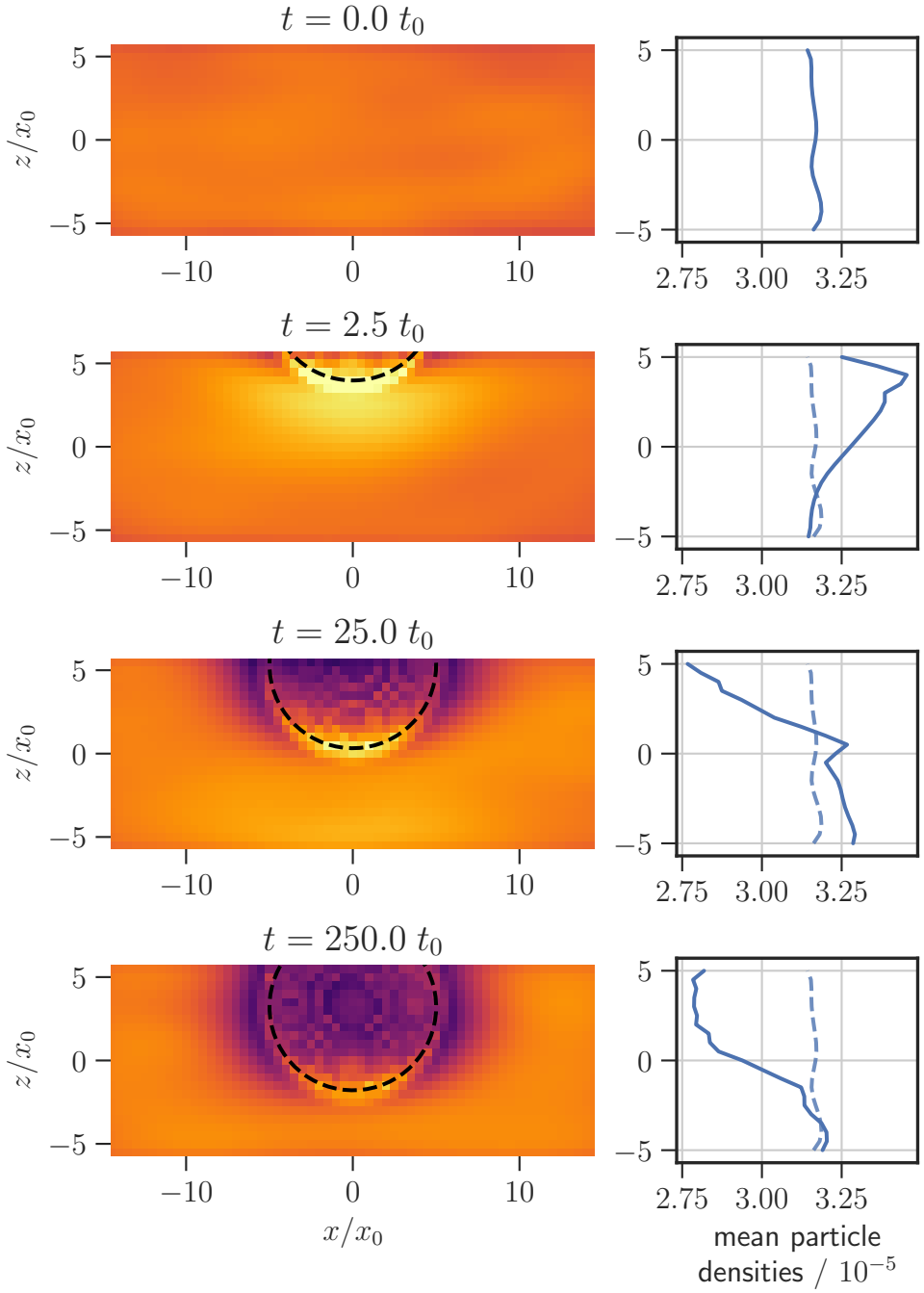

Figure 7.6: Average local particle densities in entangled (E) actin layers at different points in time. **(Left)** Average local densities evaluated on a grid with 81x33x33 voxels. For the 2D visualization, averages over the 33 voxels in y-direction were computed. Colors represent different densities, ranging from low density (dark purple) to high density (bright yellow). Area occupied by the indenter is shown as black dashed line. **(Right)** Mean density profiles were computed at the x-positions of the indenter ($x \in [-5x_0, x_0]$) and plotted at different z-positions. The profile at $t = 0t_0$ was plotted as dashed line in the plots at later points in time.

7.5 Relaxation Times

To study relaxation times, the indentation simulations above were continued from their state at $t = 25t_0$, with the indenter fixed to its position at that frame. Technically, the indenter particle was replaced with with a spherical harmonic exclusion potential, which is identical to the repulsion in equation 7.1, but the indenter is fixed to the position $r_I(t = 25t_0)$. The repulsion force F_r exterted by the network on the fixed indenter is computed as the sum of contributions in z-direction f_z by individual beads, where

$$\begin{aligned} f_z = -\frac{\partial V_s}{\partial_z} &= -k \cdot (d - ||\mathbf{r}||) \frac{\partial}{\partial_z} \sqrt{x^2 + y^2 + z^2} \\ &= -k \cdot (d - ||\mathbf{r}||) \frac{z}{||\mathbf{r}||} \\ &= k \cdot z \cdot \left(1 - \frac{d}{||\mathbf{r}||}\right). \end{aligned} \tag{7.3}$$

For the entangled (E) layers, figure 7.7 shows the z-position and computed force on the indenter before and while it was fixed to one position. The repulsion force during downwards movement of the indenter (left of black dashed line) was computed from the average indenter position of $n = 16$ individual simulations as before (see equation 7.2 and figure 7.3). The repulsion force on the

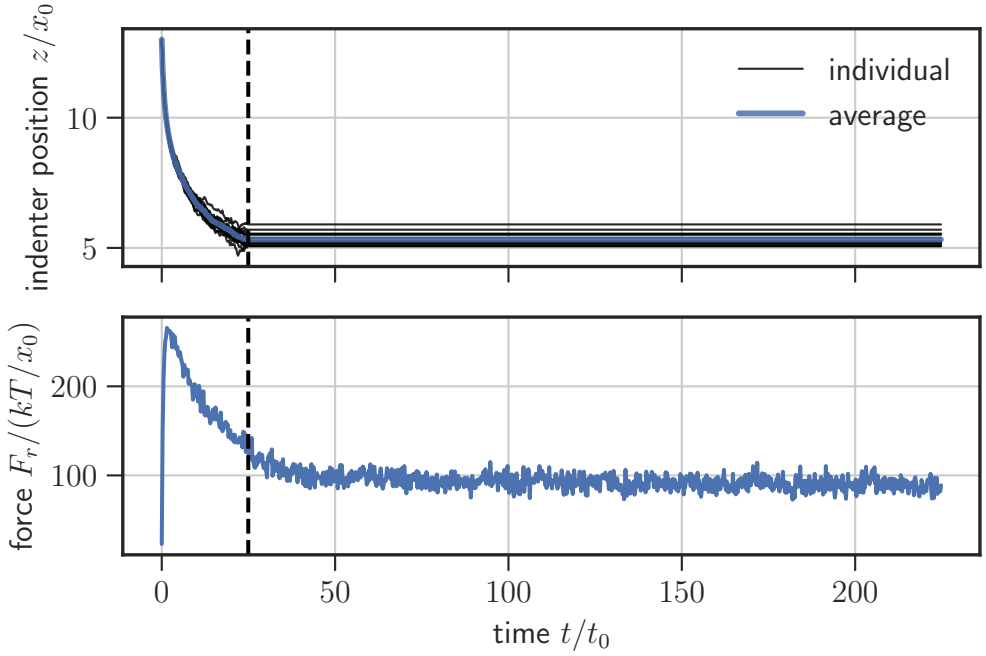

Figure 7.7: Indenter position and repulsion force acting on the indenter during downwards motion ($t < 25t_0$, left of black dashed line) as well as during fixation ($t > 25t_0$) in simulations with entangled actin networks. Positions (top) are plotted for individual simulations ($n = 16$) as well as the average position over these individuals. Force during downwards motion is estimated via equation 7.2, during fixation via equation 7.3.

fixed indenter (right of black dashed line) was computed via equation 7.3 for the $n = 16$ individual indenters and subsequently averaged. As can be seen in the plot of the repulsion force F_r, the layer exerts a positive repulsion force on the indenter when the indenter stops moving and becomes fixed. The exerted repulsion force then decays, where the decay consists of two different regimes. Directly after fixation, the force decays rapidly until around $t = 50t_0$. This rapid exponential decay has presumably to be attributed to the active pushing of the indenter before it was fixed. This pushing causes polymers to be displaced radpidly, which in turn have to pervade the space of neighboring polymers, causing the displacement of the neighbors and so on.

At $t > 50t_0$, the decay continues in a drastically slower fashion, which appears to be linear on the time scales covered in these simulations. The likely cause for this slow mode of decay is the slow diffusion of the polymers in the entangled networks. A relaxation process where polymers on average diffuse from the higher density volumes (caused by compression from the indenter) to lower density volumes, slowly maximizing the entropy in the system.

Figure 7.8 shows the average positions and repulsion forces over the $n = 16$ simulations for each of the 5 different conditions *E*, *T*, *SCL*, *DCL*, *A*. Note that the *T* simulations with fixed indenter were started from the states at $t = 25t_0$ of the *E* simulations, as starting from a simulation with varying particle numbers

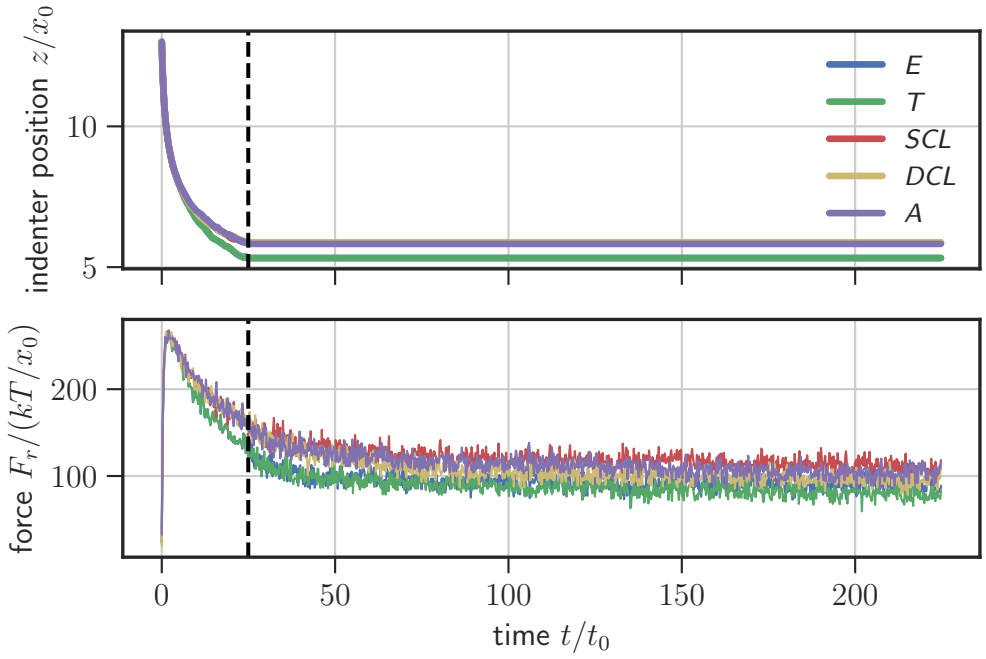

Figure 7.8: Average indenter positions and repulsion forces acting on indenters in simulations with different unlinked (*E*, *T*) and linked (*SCL*, *DCL*, *A*) networks. Averages were computed from $n = 16$ individual simulations for each condition. At $t < 25t_0$ (left of black dashed line), indenters were in a downward motion, after which the were fixated at their location at $t = 25t_0$. Forces during downward motion were computed via equation 7.2, whereas during fixation via equation 7.3.

(like in T) is not supported in `bead_state_model`. The same decay of the repulsion force from the network on the indenter that was seen for E in figure 7.7 can be seen for all conditions here. However, the initial exponential decay seems to be significantly slower for the linked networks of conditions *SCL*, *DCL* and *A*. To further dissect the repulsion force curves, the forces after the indenter was fixed were smoothed[1] and plotted again in figure 7.9. The smoothed force curves show more clearly the above described combination of initial fast decay followed by the slow decay.
The described shape of the curves gives rise to a fit model that is a combination of two exponentials, where one represents the fast and the other the slow decay,

$$F(t; f_{\text{fast}}, t_{\text{fast}}, f_{\text{slow}}, t_{\text{slow}},) = f_{\text{fast}} \cdot \exp\left(-t/t_{\text{fast}}\right) + f_{\text{slow}} \cdot \exp\left(-t/t_{\text{slow}}\right) \quad (7.4)$$

The model was fitted to the force curves of the different network conditions, and forces predicted by the fits as well as the parameters t_{fast} and t_{slow} are shown in figure 7.9.
Looking at the force curves of the two entangled conditions, E and T, the initial rapid decay appears to be almost identical, which is also reflected in the fit parameter t_{fast}. The following slow decay shows a clearly steeper decay for networks with treadmilling enabled (T), which leads to a lower t_{slow} in the fit. This

[1] Smoothing was done using a Savitzky-Golay filter via the `savgol_filter` function of the scipy python package [98].

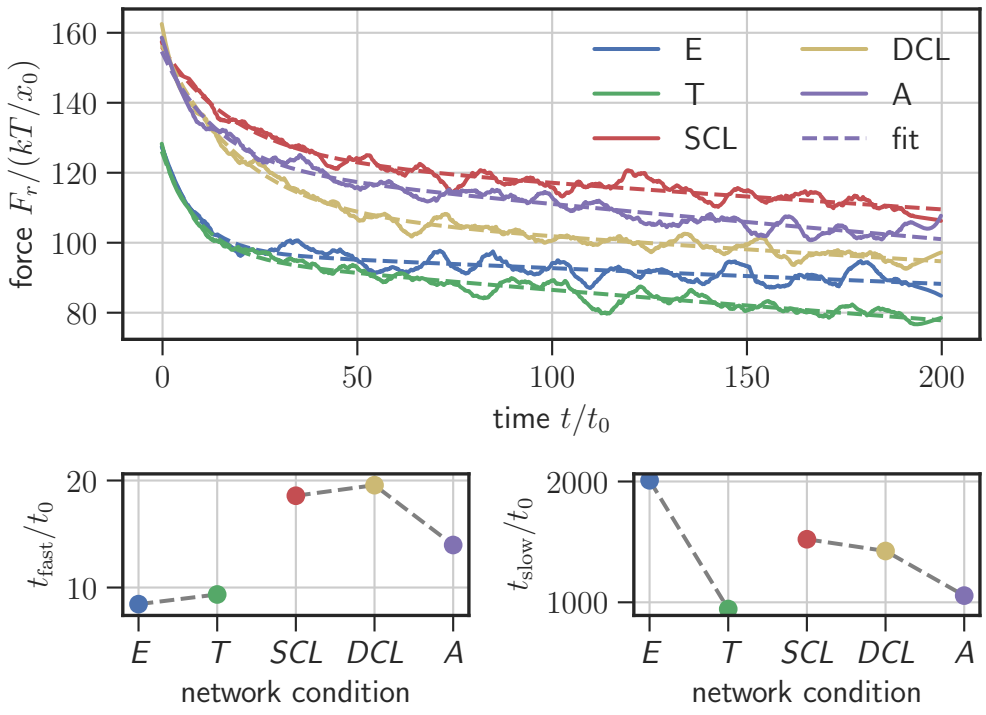

Figure 7.9: Forces on fixed indenter and fit with a combined fast and slow exponential decay (equation 7.4). **(Top)** Smoothed force curves (solid lines) and fits (dashed lines). Fits were performed with the original rather than the smoothed data. **(Bottom)** Resulting values for fit parameters t_{fast} (left) and t_{slow} (right).

matches the expectations, since the polymers without treadmilling enabled (*E*) can only passively diffuse through the network, whereas the treadmilling (*T*) adds an active component of motion to the passive diffusion (see also section 7.6 on filament motion).
For the cross-linked conditions (*SCL*, *DCL*) a slower decay due to polymer diffusion than for the entangled network (*E*) is to be expected since the diffusion should become slower for cross-linked polymers (see also section 7.6 on filament motion). However, this is not reflected in the t_{slow} fit parameter, where the values are lower than for entangled networks (*E*). This steeper descent is also directly visible at large t in the force curves (comparing for example *DCL* and *E*) in figure 7.9. It has to be assumed that the double exponential fit model fails for the comparison across the entangled (*E*, *T*) and the linked networks (*SCL*, *DCL*, *A*) for the data available here, due to the large difference in the initial forces at $t = 0$ (see figure 7.9) and due to the short time scale covered where the fast decay is still dominant.
Within the group of linked networks, however, one can see that fits to the force curves of active networks (*A*) yield a lower t_{slow} as compared to the cross-linked networks (*SCL*, *DCL*). This matches the assumption that actively moving filaments lead to faster network relaxation. Comparing only the cross-linked networks, the fit yields a slightly longer relaxation time for statically cross-linked (*SCL*) than dynamically cross-linked networks (*DCL*). With $t_{\text{slow,SCL}} = 1522t_0 \pm 161t_0$ and $t_{\text{slow,DCL}} = 1425t_0 \pm 155t_0$ the difference is still within error margins, though.

7.6 Filament Motion for Different Conditions

The filament motion was used above as an argument for the expected relaxation times of the different network conditions. However, in systems as complex as actin networks, the impact of components like cross-links and motors on mechanical network properties depend on many factors like the motor unbinding rate [99] or the ratio of actin and cross-link concentrations [100]. To garner insights on the motion of filaments for different network conditions that were simulated here, the MSDs of the centers of mass (CoMs) of filaments were computed. A plot of the MSDs is shown in figure 7.10. The MSDs were computed as averages over the 800 filaments of an individual simulation, and then averaged over the 16 simulations for each condition. The errors were computed as the standard deviation over the 16 simulation MSDs. Except for the treadmilling condition (*T*), filaments did on average not travel the distance comparable to the size of the indenter. With treadmilling, filament CoMs cover distances comparable to that a freely diffusing filament bead would travel. Filaments of the unlinked conditions (*E*, *T*) moved larger distances than those of linked conditions (*SCL*, *DCL*, *A*).
Across the linked conditions, surprisingly, dynamic link binding and unbinding (*DCL*, *A*) results in shorter travelled distances than static linking (*SCL*). It is only towards the end of the covered lag time range, that the MSDs of filaments in networks with dynamic links overtake MSDs of filaments in statically

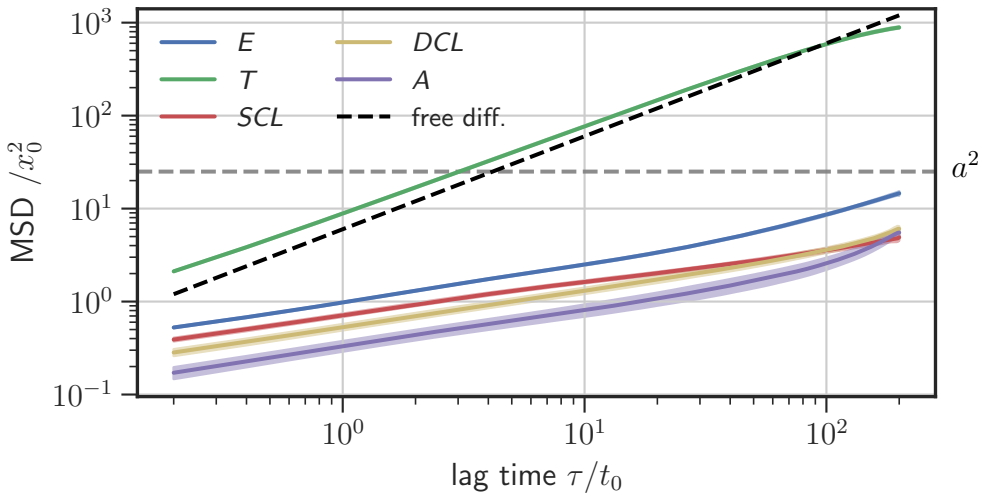

Figure 7.10: MSDs of the centers of mass (CoMs) of filaments for different network conditions. Errors (standard deviation over 16 MSDs of individual simulations) are plotted as background shade in matching color, but is only visible for *A*. The black dashed line (slope 1) is a hypothetical MSD of a freely diffusing filament bead. They gray dashed horizontal line marks the radius a of the indenter.

linked networks. Another noteworthy aspect that shows towards the end of the covered time scale is that the slope of the MSDs of condition *A* increased faster than those of other conditions. This might indicate larger travel distances for *A* at time scales just beyond those in these simulations.

7.7 Discussion

Simulations in this chapter demonstrate the flexibility of the `bead_state_model` python package. One important aspect of that flexibility concerns the configuration of systems with external particles and various potentials affecting polymers, external particles, and their interactions. With these tools, many different systems can be constructed, and various actin model systems can be mimicked. Furthermore, the full set of available standard reactions (attachment and detachment of monomers, link binding/unbinding, motor steps) were demonstrated in this chapter. While `bead_state_model` does not offer anything like the countless options to configure binding proteins as cytosim does [101–103], by providing fundamental processes like cross-linking and motor steps, `bead_state_model` becomes a useful tool to study the impact of these processes in various different scenarios.
Here, a hypothetical model system of an actin layer not covered by a membrane has been constructed and its mechanical properties probed with a spherical indenter. While (albeit much thinner) actin layers in experiments are usually studied in conjunction with a membrane (in vivo [95–97] or in vitro [25]), `bead_state_model` allows us to study polymer networks free of other compo-

nents. However, making use of the freedom the underlying ReaDDy engine provides in building topologies and interactions between topologies, one could add a dense sheet of connected particles (representing a highly coarse-grained membrane) and anchor polymers to this sheet. This would allow us to study systems similar to actin layers in conjunction with membranes.

To study the relaxation times of networks with different conditions, the indenter was fixed and the repulsion force F_r asserted by the polymers onto the indenter was measured. The repulsion force decayed rapidly right after the indenter was fixed, and exhibited a slower linear decay later on. It was argued that two different relaxation processes were at play, leading to two distinct regimes in the decay of F_r: (1) a fast exponential decay due to filaments being pressed into the volume below the indenter, and (2) a slow decay for filaments migrating/diffusing from the high density volume below the indenter (see also figure 7.6) into lower density volumes.
Based on this argument, the force curves were fitted with a combined fast and slow exponential decay model. Results of these fits matched the expectations (faster relaxation in networks with faster moving filaments) well for the entangled networks with and without treadmilling. It was argued that in the present analysis it is not possible to compare the relaxation times across entangled and linked networks, due to the large different in initial forces. Within the group of linked networks, however, the relaxation times for the slow relaxation process are as one would expect – longest relaxation time for statically cross-linked networks and shortest relaxation time for active networks. However, the covered time span is quite short. More insight might be gained by conducting simulations covering longer time spans.
In a similar analysis by Gurmessa et al. [24] performed on entangled networks that were strained with a microsphere, they observed exponential force decay dropping down to zero force eventually, which is the expected response of a viscoelastic liquid to strain [45]. It is possible that the forces in the systems simulated in this chapter would also decay towards zero given long enough simulated time spans. However, one clear difference between experiments by Gurmessa et al. [24] and the simulations presented here, was that filaments were only below the indenter in the simulations, whereas the indenter was fully surrounded by filaments in their experiments. With filaments only below, one would expect a positive residue force (exerting a force in positive z direction) due to thermal fluctuations always driving some filaments back into overlap with the indenter, even for a fully relaxed network.

One of the main drivers of run time of simulations is the particle number. Thus, on one hand, keeping the particle number low enough, that simulations do not take months or longer to finish, is crucial. On the other hand, one has to carefully consider how many particles need to be in the simulation to accurately capture the physics of the systems that are to be studied. In the simulations in this chapter, due to the experience from the MR simulations (chapter 6), simulations in this chapter were conducted with $N = 800$ filaments ($25 \cdot N = 20000$ filament particles). These particles were placed in a volume only 14% of the vol-

ume used for the homogeneous 3D networks for the MR simulations, leading to about 7 times higher densities. Still, the indenter moved quickly down into this rather dense layer, as can be seen in figure 7.5. The networks seems to offer only little resistance to the movement of the indenter.
One possible strategy to increase the network's density further, would be to make the layer thinner. It was set to $h = 15x_0$. Applying the conversion factors obtained in section 5.2, be it to only get a rough estimate, would put the layer height at $h \approx 7.5$ µm, which is at the order of the size of a cell. Hence, making the layer much thinner would also make sense in order to make the layer more cortex-like. However, one might have to carefully tune the combination of time step size and how thin the layer can become, before the forces on indenter and polymers become too large and lead to unphysical behaviour.

Chapter 8

Conclusion

In this thesis it was demonstrated that the `bead_state_model` framework can be used to replicate actin systems in computer simulations. These replications include the actin networks' interactions with external particles introduced for specific measurements techniques.
In order to verify that mechanical properties of single polymers as well as polymer networks are accurately captured by `bead_state_model`, the dynamics of isolated polymers were compared with assumptions from the worm-like chain model in chapter 5, and microrheology (MR) results from simulations were compared against theory and experiments in chapter 6.
To demonstrate the flexibility of the framework, two different systems served as proof of principle: homogeneous three-dimensional actin networks with inserted passive microspheres for MR in chapter 6, and actin layers that were compressed with spherical indenters in chapter 7. For both systems, the flexible and extendible network generation tools provided by `bead_state_model` were used to generate initial states that avoided overlap between networks and external particles, which enabled equilibration simulations with larger time steps (see section 4.1.3).
As described in section 4.1.4, `bead_state_model` provides full access to functionality of the underlying ReaDDy simulation framework, to define arbitrary particles and bonds between particles. Here, these functionalities were used to add passive microspheres (MR) and active microspheres that were subject to a potential driving them down into actin network layers (indentation).

One concern in simulations of relevant length scales and time scales was the performance that determines how long the simulations take. In `bead_state_model` as well as in cytosim, the performance was further reduced significantly when particles that were large in comparison to the polymers were introduced (see section A3). Nonetheless, the run time of `bead_state_model` simulations presented in this thesis remained within reasonable bounds. The longest ones, MR simulations with microspheres of around 30 times the size of a bead in the polymer bead chains, took 30-40 days. Halving the size of the microspheres drastically reduced the run times. The size of the largest particle heavily impacts the run time, hence this size should be carefully considered when using the `bead_state_model` simulation framework.

For systems without external components, the frameworks cytosim [11] and MEDYAN [14] might well excel in performance and flexibility. For example, both these frameworks give their users better control over cross-links and motors. Thus, to study for example effects of different cross-links with different angles, or the interplay when different cross-links/motors are present at the same time, cytosim or MEDYAN would likely be the better choice.
The `bead_state_model` simulation framework was built as a tool to help shed light on systems where external components interact with actin networks. Its here demonstrated ability to replicate these kind of systems is a crucial difference and distinguishes `bead_state_model` from the aforementioned simulation frameworks. Hence, `bead_state_model` expands the ecosystem of open source actin simulation tools.

Bibliography

[1] Laurent Blanchoin et al. "Actin Dynamics, Architecture, and Mechanics in Cell Motility". In: *Physiological Reviews* 94.1 (Jan. 2014), pp. 235–263. DOI: 10.1152/physrev.00018.2013.

[2] Yashar Bashirzadeh and Allen P. Liu. "Encapsulation of the cytoskeleton: towards mimicking the mechanics of a cell". In: *Soft Matter* 15 (42 2019), pp. 8425–8436. DOI: 10.1039/C9SM01669D.

[3] J.Victor Small. "Lamellipodia architecture: actin filament turnover and the lateral flow of actin filaments during motility". In: *Seminars in Cell Biology* 5.3 (1994), pp. 157–163. ISSN: 1043-4682. DOI: https://doi.org/10.1006/scel.1994.1020.

[4] A. Mogilner and G. Oster. "Cell motility driven by actin polymerization". In: *Biophysical Journal* 71.6 (1996), pp. 3030–3045. ISSN: 0006-3495. DOI: https://doi.org/10.1016/S0006-3495(96)79496-1.

[5] Gary G Borisy and Tatyana M Svitkina. "Actin machinery: pushing the envelope". In: *Current Opinion in Cell Biology* 12.1 (2000), pp. 104–112. ISSN: 0955-0674. DOI: https://doi.org/10.1016/S0955-0674(99)00063-0.

[6] Antje Schirenbeck et al. "The Diaphanous-related formin dDia2 is required for the formation and maintenance of filopodia". In: 7.6 (May 2005), pp. 619–625. DOI: 10.1038/ncb1266.

[7] Thomas H. Cheffings, Nigel J. Burroughs, and Mohan K. Balasubramanian. "Actomyosin Ring Formation and Tension Generation in Eukaryotic Cytokinesis". In: *Current Biology* 26.15 (2016), R719–R737. ISSN: 0960-9822. DOI: https://doi.org/10.1016/j.cub.2016.06.071.

[8] C. P. Broedersz and F. C. MacKintosh. "Modeling semiflexible polymer networks". In: *Reviews of Modern Physics* 86.3 (July 2014), pp. 995–1036. DOI: 10.1103/revmodphys.86.995.

[9] D.H. Boal. "Computer simulation of a model network for the erythrocyte cytoskeleton". In: *Biophysical Journal* 67.2 (1994), pp. 521–529. ISSN: 0006-3495. DOI: https://doi.org/10.1016/S0006-3495(94)80511-9.

[10] R. Everaers et al. "Dynamic Fluctuations of Semiflexible Filaments". In: *Phys. Rev. Lett.* 82 (18 May 1999), pp. 3717–3720. DOI: 10.1103/PhysRevLett.82.3717.

[11] Francois Nedelec and Dietrich Foethke. "Collective Langevin dynamics of flexible cytoskeletal fibers". In: *New Journal of Physics* 9.11 (2007), p. 427. DOI: 10.1088/1367-2630/9/11/427.

[12] Taeyoon Kim et al. "Computational analysis of viscoelastic properties of crosslinked actin networks". In: *PLoS computational biology* 5.7 (2009), e1000439.

[13] Claus Heussinger. "Stress relaxation through crosslink unbinding in cytoskeletal networks". In: *New Journal of Physics* 14.9 (2012), p. 095029.

[14] Konstantin Popov, James Komianos, and Garegin A. Papoian. "MEDYAN: Mechanochemical Simulations of Contraction and Polarity Alignment in Actomyosin Networks". In: *PLOS Computational Biology* 12.4 (Apr. 2016), pp. 1–35. DOI: 10.1371/journal.pcbi.1004877.

[15] Simon L. Freedman et al. "A Versatile Framework for Simulating the Dynamic Mechanical Structure of Cytoskeletal Networks". In: *Biophysical Journal* 113.2 (2017), pp. 448–460. ISSN: 0006-3495. DOI: https://doi.org/10.1016/j.bpj.2017.06.003.

[16] Henry E. Amuasi et al. "Linear rheology of reversibly cross-linked biopolymer networks". In: *The Journal of Chemical Physics* 149.8 (2018), p. 084902. DOI: 10.1063/1.5030169.

[17] Daniel S. Seara et al. "Entropy production rate is maximized in non-contractile actomyosin". In: *Nature Communications* 9.1 (Nov. 2018). DOI: 10.1038/s41467-018-07413-5.

[18] Taeyoon Kim, Wonmuk Hwang, and Roger D. Kamm. "Dynamic Role of Cross-Linking Proteins in Actin Rheology". In: *Biophysical Journal* 101.7 (2011), pp. 1597–1603. ISSN: 0006-3495. DOI: https://doi.org/10.1016/j.bpj.2011.08.033.

[19] Taeyoon Kim. "Determinants of contractile forces generated in disorganized actomyosin bundles". In: *Biomechanics and Modeling in Mechanobiology* 14.2 (Aug. 2014), pp. 345–355. DOI: 10.1007/s10237-014-0608-2.

[20] S J Kron and J A Spudich. "Fluorescent actin filaments move on myosin fixed to a glass surface". In: *Proceedings of the National Academy of Sciences* 83.17 (1986), pp. 6272–6276. ISSN: 0027-8424. DOI: 10.1073/pnas.83.17.6272.

[21] Akiyoshi Kishino and Toshio Yanagida. "Force measurements by micromanipulation of a single actin filament by glass needles". In: *Nature* 334.6177 (July 1988), pp. 74–76. DOI: 10.1038/334074a0.

[22] R. Tharmann, M. M. A. E. Claessens, and A. R. Bausch. "Viscoelasticity of Isotropically Cross-Linked Actin Networks". In: *Phys. Rev. Lett.* 98 (8 Feb. 2007), p. 088103. DOI: 10.1103/PhysRevLett.98.088103.

[23] Gijsje H. Koenderink et al. "An active biopolymer network controlled by molecular motors". In: *Proceedings of the National Academy of Sciences* 106.36 (2009), pp. 15192–15197. ISSN: 0027-8424. DOI: 10.1073/pnas.0903974106.

[24] Bekele Gurmessa et al. “Entanglement Density Tunes Microscale Nonlinear Response of Entangled Actin”. In: *Macromolecules* 49.10 (2016), pp. 3948–3955. DOI: 10.1021/acs.macromol.5b02802.

[25] Stefan Nehls and Andreas Janshoff. “Elastic Properties of Pore-Spanning Apical Cell Membranes Derived from MDCK II Cells”. In: *Biophysical Journal* 113.8 (2017), pp. 1822–1830. ISSN: 0006-3495. DOI: https://doi.org/10.1016/j.bpj.2017.08.038.

[26] Helen Nöding et al. “Rheology of Membrane-Attached Minimal Actin Cortices”. In: *The Journal of Physical Chemistry B* 122.16 (2018). PMID: 29589937, pp. 4537–4545. DOI: 10.1021/acs.jpcb.7b11491.

[27] Laurent Limozin and Erich Sackmann. “Polymorphism of Cross-Linked Actin Networks in Giant Vesicles”. In: *Phys. Rev. Lett.* 89 (16 Sept. 2002), p. 168103. DOI: 10.1103/PhysRevLett.89.168103.

[28] Mireille M. A. E. Claessens et al. “Actin-binding proteins sensitively mediate F-actin bundle stiffness”. In: *Nature Materials* 5.9 (Aug. 2006), pp. 748–753. DOI: 10.1038/nmat1718.

[29] F. Amblard et al. “Subdiffusion and Anomalous Local Viscoelasticity in Actin Networks”. In: *Phys. Rev. Lett.* 77 (21 Nov. 1996), pp. 4470–4473. DOI: 10.1103/PhysRevLett.77.4470.

[30] Manlio Tassieri et al. “Dynamics of Semiflexible Polymer Solutions in the Highly Entangled Regime”. In: *Phys. Rev. Lett.* 101 (19 Nov. 2008), p. 198301. DOI: 10.1103/PhysRevLett.101.198301.

[31] Jun He and Jay X. Tang. “Surface adsorption and hopping cause probe-size-dependent microrheology of actin networks”. In: *Phys. Rev. E* 83 (4 Apr. 2011), p. 041902. DOI: 10.1103/PhysRevE.83.041902.

[32] S. Kaufmann et al. “Talin anchors and nucleates actin filaments at lipid membranes A direct demonstration”. In: *FEBS Letters* 314.2 (1992), pp. 203–205. ISSN: 0014-5793. DOI: https://doi.org/10.1016/0014-5793(92)80975-M.

[33] H. Isambert and A. C. Maggs. “Dynamics and Rheology of Actin Solutions”. In: *Macromolecules* 29.3 (Jan. 1996), pp. 1036–1040. DOI: 10.1021/ma946418x.

[34] L. Limozin, M. Bärmann, and E. Sackmann. “On the organization of self-assembled actin networks in giant vesicles”. In: 10.4 (Apr. 2003), pp. 319–330. DOI: 10.1140/epje/i2002-10118-9.

[35] Helen Nöding. “Active and Passive Microrheology of F-Actin Membrane Composites”. PhD thesis. Georg-August-Universität Göttingen, 2018.

[36] Qin Ni and Garegin A. Papoian. “Turnover versus treadmilling in actin network assembly and remodeling”. In: *Cytoskeleton* 76.11-12 (2019), pp. 562–570. DOI: 10.1002/cm.21564.

[37] Gaelle Letort et al. “Geometrical and Mechanical Properties Control Actin Filament Organization”. In: *PLOS Computational Biology* 11.5 (May 2015), pp. 1–21. DOI: 10.1371/journal.pcbi.1004245.

[38] Hajer Ennomani et al. "Architecture and Connectivity Govern Actin Network Contractility". In: *Current Biology* 26.5 (2016), pp. 616–626. ISSN: 0960-9822. DOI: https://doi.org/10.1016/j.cub.2015.12.069.

[39] Wei Yung Ding et al. "Plastin increases cortical connectivity to facilitate robust polarization and timely cytokinesis". In: *Journal of Cell Biology* 216.5 (Apr. 2017), pp. 1371–1386. ISSN: 0021-9525. DOI: 10.1083/jcb.201603070.

[40] Mariia Burdyniuk et al. "F-Actin nucleated on chromosomes coordinates their capture by microtubules in oocyte meiosis". In: *Journal of Cell Biology* 217.8 (June 2018), pp. 2661–2674. ISSN: 0021-9525. DOI: 10.1083/jcb.201802080.

[41] Donald L. Ermak and J. A. McCammon. "Brownian dynamics with hydrodynamic interactions". In: *The Journal of Chemical Physics* 69.4 (1978), pp. 1352–1360. DOI: 10.1063/1.436761.

[42] Alexei Podtelezhnikov and Alexander Vologodskii. "Simulations of Polymer Cyclization by Brownian Dynamics". In: *Macromolecules* 30.21 (1997), pp. 6668–6673. DOI: 10.1021/ma970391a.

[43] Stuart A Allison. "Brownian Dynamics Simulation of Wormlike Chains. Fluorescence Depolarization and Depolarized Light Scattering". In: *Macromolecules* 19.1 (1986), pp. 118–124.

[44] Masao Doi and Samuel Frederick Edwards. *The Theory of Polymer Dynamics*. Vol. 73. International Series of Monographs on Physics. Oxford University Press, 1988. ISBN: 978-0198520337.

[45] Michael Rubinstein and Ralph H. Colby. *Polymer Physics*. Oxford University Press, 2004. ISBN: 978 0 19 852059 7.

[46] Prince E. Rouse. "A Theory of the Linear Viscoelastic Properties of Dilute Solutions of Coiling Polymers". In: *The Journal of Chemical Physics* 21.7 (1953), pp. 1272–1280. DOI: 10.1063/1.1699180.

[47] P. G. de Gennes. "Reptation of a Polymer Chain in the Presence of Fixed Obstacles". In: *The Journal of Chemical Physics* 55.2 (1971), pp. 572–579. DOI: 10.1063/1.1675789.

[48] J. Käs et al. "F-actin, a model polymer for semiflexible chains in dilute, semidilute, and liquid crystalline solutions". In: *Biophysical Journal* 70.2 (Feb. 1996), pp. 609–625. ISSN: 0006-3495. DOI: 10.1016/s0006-3495(96)79630-3.

[49] Marie-France Carlier and Shashank Shekhar. "Global treadmilling coordinates actin turnover and controls the size of actin networks". In: *Nature Reviews Molecular Cell Biology* 18.6 (Mar. 2017), pp. 389–401. DOI: 10.1038/nrm.2016.172.

[50] Thomas D. Pollard, Laurent Blanchoin, and R. Dyche Mullins. "Molecular Mechanisms Controlling Actin Filament Dynamics in Nonmuscle Cells". In: *Annual Review of Biophysics and Biomolecular Structure* 29.1 (2000). PMID: 10940259, pp. 545–576. DOI: 10.1146/annurev.biophys.29.1.545.

[51] Romain Levayer and Thomas Lecuit. "Biomechanical regulation of contractility: spatial control and dynamics". In: *Trends in Cell Biology* 22.2 (2012), pp. 61–81. ISSN: 0962-8924. DOI: https://doi.org/10.1016/j.tcb.2011.10.001.

[52] Henry J. Karam. "Viscometers and their use". In: *Industrial & Engineering Chemistry* 55.2 (1963), pp. 38–43.

[53] D Weipert. "The Benefits of Basic Rheometry in Studying Dough Rheology". In: *Cereal chem* 67.4 (1990), pp. 311–317.

[54] M. Mours and H. H. Winter. "Time-resolved rheometry". In: 33.5 (1994), pp. 385–397. DOI: 10.1007/bf00366581.

[55] Bivash R. Dasgupta et al. "Microrheology of polyethylene oxide using diffusing wave spectroscopy and single scattering". In: *Phys. Rev. E* 65 (5 May 2002), p. 051505. DOI: 10.1103/PhysRevE.65.051505.

[56] T. G. Mason and D. A. Weitz. "Optical Measurements of Frequency-Dependent Linear Viscoelastic Moduli of Complex Fluids". In: *Phys. Rev. Lett.* 74 (7 Feb. 1995), pp. 1250–1253. DOI: 10.1103/PhysRevLett.74.1250.

[57] F.C. MacKintosh and C.F. Schmidt. "Microrheology". In: *Current Opinion in Colloid & Interface Science* 4.4 (1999), pp. 300–307. ISSN: 1359-0294. DOI: https://doi.org/10.1016/S1359-0294(99)90010-9.

[58] T. G. Mason et al. "Particle Tracking Microrheology of Complex Fluids". In: *Phys. Rev. Lett.* 79 (17 Oct. 1997), pp. 3282–3285. DOI: 10.1103/PhysRevLett.79.3282.

[59] T. Gisler and D. A. Weitz. "Scaling of the Microrheology of Semidilute F-Actin Solutions". In: *Phys. Rev. Lett.* 82 (7 Feb. 1999), pp. 1606–1609. DOI: 10.1103/PhysRevLett.82.1606.

[60] Andre Palmer et al. "Diffusing Wave Spectroscopy Microrheology of Actin Filament Networks". In: *Biophysical Journal* 76.2 (1999), pp. 1063–1071. ISSN: 0006-3495. DOI: https://doi.org/10.1016/S0006-3495(99)77271-1.

[61] Thomas G. Mason. "Estimating the viscoelastic moduli of complex fluids using the generalized Stokes-Einstein equation". In: *Rheologica Acta* 39.4 (Aug. 2000), pp. 371–378. DOI: 10.1007/s003970000094.

[62] Thomas Soddemann, Burkhard Dünweg, and Kurt Kremer. "Dissipative particle dynamics: A useful thermostat for equilibrium and nonequilibrium molecular dynamics simulations". In: *Phys. Rev. E* 68 (4 Oct. 2003), p. 046702. DOI: 10.1103/PhysRevE.68.046702.

[63] Amit Mor, Guy Ziv, and Yaakov Levy. "Simulations of proteins with inhomogeneous degrees of freedom: The effect of thermostats". In: *Journal of Computational Chemistry* 29.12 (2008), pp. 1992–1998. DOI: https://doi.org/10.1002/jcc.20951.

[64] Moritz Hoffmann, Christoph Fröhner, and Frank Noé. "ReaDDy 2: Fast and flexible software framework for interacting-particle reaction dynamics". In: *PLOS Computational Biology* 15.2 (Feb. 2019), pp. 1–26. DOI: 10.1371/journal.pcbi.1006830.

[65] David Fincham. "Choice of timestep in molecular dynamics simulation". In: *Computer Physics Communications* 40.2 (1986), pp. 263–269. ISSN: 0010-4655. DOI: https://doi.org/10.1016/0010-4655(86)90113-X.

[66] Sangrak Kim. "Issues on the Choice of a Proper Time Step in Molecular Dynamics". In: *Physics Procedia* 53 (2014). 26th Annual CSP Workshop on "Recent Developments in Computer Simulation Studies in Condensed Matter Physics", CSP 2013, pp. 60–62. ISSN: 1875-3892. DOI: https://doi.org/10.1016/j.phpro.2014.06.027.

[67] Daniel A. Beard and Tamar Schlick. "Inertial stochastic dynamics. I. Long-time-step methods for Langevin dynamics". In: *The Journal of Chemical Physics* 112.17 (2000), pp. 7313–7322. DOI: 10.1063/1.481331.

[68] Loup Verlet. "Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules". In: *Phys. Rev.* 159 (1 July 1967), pp. 98–103. DOI: 10.1103/PhysRev.159.98.

[69] A. Iniesta and J. García de la Torre. "A second-order algorithm for the simulation of the Brownian dynamics of macromolecular models". In: *The Journal of Chemical Physics* 92.3 (1990), pp. 2015–2018. DOI: 10.1063/1.458034.

[70] T Iwai, C-W Hong, and P Greil. "FAST PARTICLE PAIR DETECTION ALGORITHMS FOR PARTICLE SIMULATIONS". In: *International Journal of Modern Physics C* 10.05 (1999), pp. 823–837. DOI: 10.1142/S0129183199000644.

[71] V. Ogarko and S. Luding. "A fast multilevel algorithm for contact detection of arbitrarily polydisperse objects". In: *Computer Physics Communications* 183.4 (2012), pp. 931–936. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2011.12.019.

[72] Yuliang Wang et al. "Quantitative Characterization of Cell Behaviors through Cell Cycle Progression via Automated Cell Tracking". In: *PLOS ONE* 9.6 (June 2014), pp. 1–17. DOI: 10.1371/journal.pone.0098762.

[73] John C. Crocker and Brenton D. Hoffman. "Multiple-Particle Tracking and Two-Point Microrheology in Cells". In: *Cell Mechanics*. Vol. 83. Methods in Cell Biology. Academic Press, 2007, pp. 141–178. DOI: https://doi.org/10.1016/S0091-679X(07)83007-X. URL: http://www.sciencedirect.com/science/article/pii/S0091679X0783007X.

[74] Anna V. Schepers et al. "Multiscale mechanics and temporal evolution of vimentin intermediate filament networks". In: *Proceedings of the National Academy of Sciences* 118.27 (2021). ISSN: 0027-8424. DOI: 10.1073/pnas.2102026118.

[75] Timo Maier and Tamás Haraszti. "Python algorithms in particle tracking microrheology". In: *Chemistry Central Journal* 6.1 (2012), pp. 1–9.

[76] Tobias Paust. “New Methods for Passive and Active Microrheology”. PhD thesis. Universität Ulm, 2013. DOI: 10.18725/OPARU-2611.

[77] R. G. Larson et al. “Definitions of entanglement spacing and time constants in the tube model”. In: *Journal of Rheology* 47.3 (2003), pp. 809–818. DOI: 10.1122/1.1567750.

[78] Alexander Stukowski. “Visualization and analysis of atomistic simulation data with OVITO–the Open Visualization Tool”. In: *Modelling and Simulation in Materials Science and Engineering* 18.1 (Dec. 2009), p. 015012. DOI: 10.1088/0965-0393/18/1/015012.

[79] M. Reza Shaebani et al. “Computational models for active matter”. In: *Nature Reviews Physics* 2.4 (Mar. 2020), pp. 181–199. DOI: 10.1038/s42254-020-0152-1.

[80] Lawrence Korson, Walter Drost-Hansen, and Frank J. Millero. “Viscosity of Water at Various Temperatures”. In: *The Journal of Physical Chemistry* 73.1 (Jan. 1969), pp. 34–39. DOI: 10.1021/j100721a006.

[81] David C. Morse. “Tube diameter in tightly entangled solutions of semiflexible polymers”. In: *Phys. Rev. E* 63 (3 Feb. 2001), p. 031502. DOI: 10.1103/PhysRevE.63.031502.

[82] David C Morse. “Viscoelasticity of Concentrated Isotropic Solutions of Semiflexible Polymers. 1. Model and Stress Tensor”. In: *Macromolecules* 31.20 (1998), pp. 7030–7043.

[83] David C. Morse. “Viscoelasticity of Concentrated Isotropic Solutions of Semiflexible Polymers. 2. Linear Response”. In: *Macromolecules* 31.20 (1998), pp. 7044–7067. DOI: 10.1021/ma980304u.

[84] F. C. MacKintosh, J. Käs, and P. A. Janmey. “Elasticity of Semiflexible Biopolymer Networks”. In: *Phys. Rev. Lett.* 75 (24 Dec. 1995), pp. 4425–4428. DOI: 10.1103/PhysRevLett.75.4425.

[85] Margaret Lise Gardel. “Elasticity of F-actin Networks”. PhD thesis. Harvard University, 2004.

[86] R. Tharmann, M. M. A. E. Claessens, and A. R. Bausch. “Micro- and Macrorheological Properties of Actin Networks Effectively Cross-Linked by Depletion Forces”. In: *Biophysical Journal* 90.7 (2006), pp. 2622–2627. ISSN: 0006-3495. DOI: https://doi.org/10.1529/biophysj.105.070458.

[87] Rainer Tharmann. “Mechanical Properties of Complex Cytoskeleton Networks”. Dissertation. München: Technische Universität München, 2006.

[88] M. L. Gardel et al. “Microrheology of Entangled F-Actin Solutions”. In: *Phys. Rev. Lett.* 91 (15 Oct. 2003), p. 158302. DOI: 10.1103/PhysRevLett.91.158302.

[89] Hanna Hubrich. “Active Matter in Confined Geometries - Biophysics of Artificial Minimal Cortices”. PhD thesis. Georg-August-Universität Göttingen, 2020.

[90] O. Lieleg et al. "Cytoskeletal Polymer Networks: Viscoelastic Properties are Determined by the Microscopic Interaction Potential of Cross-links". In: *Biophysical Journal* 96.11 (June 2009), pp. 4725–4732. DOI: 10.1016/j.bpj.2009.03.038.

[91] Nikita Ter-Oganessian et al. "Active microrheology of networks composed of semiflexible polymers: Computer simulation of magnetic tweezers". In: *Phys. Rev. E* 72 (4 Oct. 2005), p. 041510. DOI: 10.1103/PhysRevE.72.041510.

[92] Tamara Carla Bidone et al. "Multiscale impact of nucleotides and cations on the conformational equilibrium, elasticity and rheology of actin filaments and crosslinked networks". In: *Biomechanics and modeling in mechanobiology* 14.5 (Feb. 2015), pp. 1143–1155. DOI: https://doi.org/10.1007/s10237-015-0660-6.

[93] Hyungsuk Lee et al. "Cytoskeletal Deformation at High Strains and the Role of Cross-link Unfolding or Unbinding". In: *Cellular and Molecular Bioengineering* 2.1 (Feb. 2009), pp. 28–38. DOI: 10.1007/s12195-009-0048-8.

[94] Johanna Roostalu et al. "Determinants of Polar versus Nematic Organization in Networks of Dynamic Microtubules and Mitotic Motors". In: *Cell* 175.3 (Oct. 2018), 796–808.e14. DOI: 10.1016/j.cell.2018.09.029.

[95] Stefan Nehls et al. "Stiffness of MDCK II Cells Depends on Confluency and Cell Size". In: *Biophysical Journal* 116.11 (2019), pp. 2204–2211. ISSN: 0006-3495. DOI: https://doi.org/10.1016/j.bpj.2019.04.028.

[96] Emad Moeendarbary et al. "The cytoplasm of living cells behaves as a poroelastic material". In: 12.3 (Jan. 2013), pp. 253–261. DOI: 10.1038/nmat3517.

[97] Marco Fritzsche et al. "Actin kinetics shapes cortical network structure and mechanics". In: *Science advances* 2.4 (Apr. 2016), e1501337. DOI: 10.1126/sciadv.1501337.

[98] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[99] Atsushi Matsuda et al. "Mobility of Molecular Motors Regulates Contractile Behaviors of Actin Networks". In: *Biophysical Journal* 116.11 (2019), pp. 2161–2171. ISSN: 0006-3495. DOI: https://doi.org/10.1016/j.bpj.2019.04.018.

[100] Qilin Yu et al. "Balance between Force Generation and Relaxation Leads to Pulsed Contraction of Actomyosin Networks". In: *Biophysical Journal* 115.10 (2018), pp. 2003–2013. ISSN: 0006-3495. DOI: https://doi.org/10.1016/j.bpj.2018.10.008.

[101] Thomas Surrey et al. "Physical Properties Determining Self-Organization of Motors and Microtubules". In: *Science* 292.5519 (2001), pp. 1167–1171. DOI: 10.1126/science.1059758.

[102] Beat Rupp and François Nédélec. "Patterns of molecular motors that guide and sort filaments". In: *Lab Chip* 12 (22 2012), pp. 4903–4910. DOI: 10.1039/C2LC40250E.

[103] Julio M Belmonte, Maria Leptin, and François Nédélec. "A theory that predicts behaviors of disordered cytoskeletal networks". In: *Molecular Systems Biology* 13.9 (2017), p. 941. DOI: 10.15252/msb.20177796.

Chapter A

Appendix

A1 Data and how to Reproduce it

Data were grouped into "experiment" folders. An experiment, here, is a set of similar simulations, where up to a few parameters were varied. Experiment folders were uploaded to a dataverse in the the Göttingen Research Online repository[1]. A list of experiments with direct links to their respective dataset and some additional information is given in section A1.3 below.
The raw data (as well as analysis files containing raw particle trajectories, see section 3.5.2) were excluded in most uploads, since their sizes made them unsuitable for the online repository. Every experiment folder, however, contains a `README.rst` including some instructions how equivalent raw data can be reproduced through simulations. The exact same data (down to every last digit by using the same random numbers), can not be reproduced, since ReaDDy does not provide means to get or set the random seed. For a copy of the original raw data, please contact me directly[2].
Note that most plots can be reproduced with the included intermediate, processed data. I.e., there is no need to have the raw data to reproduce most plots.

A1.1 Concepts for Reproducibility

Each of the experiment folders uploaded to the online dataverse contains a foler named `reproduce`, containing scripts to reproduce equivalent raw data, plots, and intermediate data used for plots. Each experiment folder contains the file `README.html` (generated from `README.rst`) that might contain further details specific for each experiment.

For redundancy, two independent concepts for reproducing data and plots were implemented. The first one works only with conda[3]. Conda is a package / version management tool, that allows you to have multiple python environments, where each environment can have its own specific versions of packages installed, independent of any other environment. To create a conda en-

[1] Full URL: `https://data.goettingen-research-online.de/dataverse/thesis-kuhlemann`
[2] Contact: ikuhlem@gwdg.de
[3] Conda documentation: `https://docs.conda.io/en/latest/`

vironment with packages in compatible versions, the `reproduce` folders include `environment.yml` files that contain information on exact version numbers. More details on how to use these files are given in the `README`s in the specific experiment folders.
The second option to set up an environment for reproduction is Docker[4]. Docker is a tool to build, share and manage containers of bundled applications. Similar to a virtual machine, a container includes all the dependencies (down to the kernel of the operating system), that are required by the bundled application to run. Docker has become the de facto standard in the software industry to make application building and installing reproducible across any kind of computer running (or emulating) a GNU/Linux operating system.

Both concepts were implemented such that the conda environment or docker container can be activated with a few commands, after which scripts in the `reproduce` folder can be executed with all required libraries in place.

A1.2 Experiment Folder Structure

Experiment folder names start with consecutive, zero-padded, two digit number, followed by a very short description of a few words. For example `25_entangled_MR_vary_bead_radius`.

How Simulations were Run

Experiment folders initially usually contained 3 files:

```
Experiment folder/
├── create_networks.py
├── default_params.json
└── run_simulations.py
```

The 3 initial files were used to initialize and run the simulations. Running the file `create_networks.py` created folder `initial_states` and populated it with `N` network folders `network00`, `network01`, etc. Execution of the file `run_simulations.py` with flag `--equlibrate` used these initial states, and ran rather short simulations with a carefully small time step. The goal was to equilibrate the networks to a level where it was safe to simulate them with a larger time step for the actual, data-generating simulations.
Final states of equilibration were saved in folder `initial_states_equilibrated`. In most experiments, folders in there already used the same numbering as the simulations will. Namely, the folders were named `out0000`, `out0001`, and so on. The numbers did not necessarily correspond to the numbers of the initial state folders. This depended on the instructions in the `run_simulations.py` file. Usually the same network / initial state was used for simulations under different conditions. E.g. if we wanted to compare entangled versus cross-linked conditions, the network might have been run once with links disabled, once

[4]Docker website: `https://www.docker.com/`

with links enabled. The mapping of initial state index to simulation index was written to a comma-separated values (CSV) table, when the simulations were started with `run_simulations.py` (without the `--equilibrate` flag). The table was stored in file `simulations/index.csv`. Simulation data was saved in folder `simulations` as well, in subfolders `out0000`, `out0001`, and so on.

Files after Running Simulations

After the procedure described in the last section, the experiment folder looked like this:

```
Experiment folder/
├── create_networks.py
├── default_params.json
├── initial_states/
│   ├── index.csv
│   ├── network000
│   ├── network001
│   ...
├── initial_states_equilibrated/
│   ├── out0000
│   ├── out0001
│   ...
│   └── simulation_index.csv
├── run_simulations.py
└── simulations/
    ├── out0000
    ├── out0001
    ...
    └── simulation_index.csv
```

Analysis and Results Files

In addition to the simulation scripts and output data, I put plot scripts and other analysis scripts at the top level of the experiment folders. Results produced with these scripts were usually put into a folder named `results` and into several subfolders therein.

Simulation Folder

The individual simulation folders were named `out0000`, `out0001`, and so on, and were placed in folder `simulations`. A simulation folder was created at the start of the simulation and a copy of the parameters used was placed in file

parameters.json (or in later versions in config.toml). During simulation, trajectories of particles were saved to file data.h5, and information on which particles belong to which filament and information on the bonds between filament particles was placed in links.h5. This file might have been ommited in simulation of single filaments. During analysis of simulation data, the Analyser class of the actomyosin_analyser package saved some results and interim results in file analysis.h5.

A1.3 List of Experiments

This section contains a list of experiments whose data went into this thesis at any point. The entry of an experiment contains a brief description, a link to the dataset on data.gro.uni-goettingen.de, and a list of figures based on their data.

E19

Description: Simulations of isolated single filaments with different stiffness, which is controlled via the bending force constant k_bend. These simulations were used to verify simulated polymers behaved like worm-like chains.
Link: https://doi.org/10.25625/TCTEZN
Figures: 5.1, 5.2, 5.3, 5.4, 5.5.

E25

Description: Microrheology simulations. Microspheres of varying radius were placed at the center of homogeneous 3D entangled networks. In contrast to E34, microspheres in E25 were "slippery", i.e. interactions between microsphere and polymers were purely repulsive.
Link: https://doi.org/10.25625/V7KVRJ
Figures: 6.1, 6.2, 6.3, 6.4, 6.8, 6.9, A3.1, A4.1.

E34

Description: Microrheology simulations. Microspheres of varying radius were placed at the center of homogeneous 3D entangled networks. In contrast to E25, microspheres in E34 were "sticky", i.e. interactions between microsphere and polymers were repulsive on overlap, and attractive at short ranges just outside overlap.
Link: https://doi.org/10.25625/OHVUR7
Figures: 6.4, 6.6, 6.7, A4.2.

E36

Description: Microrheology simulations, where microspheres of identical size were placed at the center of homogeneous 3D entangled networks with varying actin density. Microspheres were "sticky", i.e. interactions between microsphere and polymers were repulsive on overlap, and attractive at short ranges

just outside overlap. Counterpart with cross-linked rather than entangled networks can be found in E37.
Link: `https://doi.org/10.25625/K09BOV`
Figures: 6.10, 6.11, 6.12, 6.13, A3.2, A4.3, A4.4.

E37

Description: Microrheology simulations, where microspheres of identical size were placed at the center of homogeneous 3D cross-linked networks with varying actin density. Microspheres were "sticky", i.e. interactions between microsphere and polymers were repulsive on overlap, and attractive at short ranges just outside overlap. Counterpart with entangled rather than cross-linked networks can be found in E36.
Link: `https://doi.org/10.25625/CMHWAK`
Figures: 6.13, 6.14, 6.15, 6.16, A4.5, A4.6.

E38

Description: Short simulations of actin in a layer (periodic boundaries in x and y direction, potential in z keeping polymers inside layer). Final states of these simulations served as initial states for experiments E40, E41, E42, E43 and E45.
Link: `https://doi.org/10.25625/MVORG2`
Figures: 7.2.

E40

Description: Indentation simulations, where microspheres are pressed into layers of entangled actin networks. Final states of entangled networks of E38 served as initial states.
Link: `https://doi.org/10.25625/SP3BTR`
Figures: 7.3, 7.5, 7.7, 7.8, 7.9, 7.10, 7.6.

E41

Description: Indentation simulations, where microspheres are pressed into layers of dynamically cross-linked actin networks. In contrast to static cross-links in E42, the dynamic cross-links used here can bind and unbind during simulation. Final states of cross-linked networks of E38 served as initial states.
Link: `https://doi.org/10.25625/P9JJVA`
Figures: 7.5, 7.8, 7.9, 7.10.

E42

Description: Indentation simulations, where microspheres are pressed into layers of statically cross-linked actin networks. In contrast to dynamic cross-links in E41, the static cross-links used here are fixed, they can not bind or unbind. Final states of cross-linked networks of E38 served as initial states.

Link: https://doi.org/10.25625/OMXQOE
Figures: 7.5, 7.8, 7.9, 7.10.

E43

Description: Indentation simulations, where microspheres are pressed into layers of actomyosin networks. The myosin motors can dynamically bind, unbind and perform steps. Final states of cross-linked networks of E38 served as initial states, where cross-links were replaced by motors.
Link: https://doi.org/10.25625/TKYBEK
Figures: 7.5, 7.8, 7.9, 7.10.

E45

Description: Indentation simulations, where microspheres are pressed into layers of actin networks with treadmilling enabled. Final states of entangled networks of E38 served as initial states.
Link: https://doi.org/10.25625/NEHNA6
Figures: 7.5, 7.8, 7.9, 7.10.

E50

Description: Simulations of isolated single filaments. Data of E50 cover intermediate frame rate regimes, compared to the other similar isolated filament experiments (E51, E52).
Link: https://doi.org/10.25625/AOGTAS
Figures: 5.7, 5.9.

E51

Description: Simulations of isolated single filaments. Simulations in experiment E51 use a low frame rate / large observation interval, and a large number of simulated steps compared to similar single filament experiments (E50, E52).
Link: https://doi.org/10.25625/JT25UF
Figures: 5.7, 5.9.

E52

Description: Simulations of isolated single filaments. Simulations in experiment E52 use the highest possible frame rate (observation interval 1, i.e. record every simulation step), and are simulated for a smaller number of simulation steps compared to similar isolated filament experiments (E50, E51).
Link: https://doi.org/10.25625/G0MXSU
Figures: 5.7.

A2 Example: Set Up Simulations with External Particles

The example in section 4.1.5 defines `bead_state_model` simulations with actin components only. One goal when I developed the `bead_state_model` python package was flexibility that would allow us to set up simulations of actin combined with other components, which I refer to as *external* particles. Simulations including external particles were used in chapters 6 and 7. Those simulations, however, were configured through a now deprecated user interface. All functions used there for simulation configuration are still available as a low level interface, but usage of the new high level interface with classes `Simulation` and `Parameters` is recommended. In this section, I describe how to set up simulations with a passive microrheology (MR) bead embedded in a network.

A2.1 Scripts and Configuration Files

The 4 source files used in this example are in full length in listings L9 , L10 , L11 and L12 (section A7). A digital version of very similar files can be found in the online documentation (note, however, that files there might get updated, changed, or removed, and that you might have to fall back to the code in the referenced listings).
Create a folder `example_microrheo`, into which you place the 4 source files:

```
example_microrheo
├── create_networks.py
├── default_params.json
├── run_equilibrations.py
└── run_simulations.py
```

A2.2 Create Networks

The first script you have to run is `create_networks.py`. Run it e.g. with:

```
python create_networks.py 3
```

The number in the end is the number of processes to run in parallel. Adjust it to match ne number of processes that you want to use (more than 6 won't have any benefit, as only 6 networks will be generated). The script will generate 6 networks with 17500 particles in total (forming 700 filaments with 25 beads each). The script saves the generated networks in folders `initial_states/network000` to `initial_states/network005`. An index table is generated and saved as `initial_states/index.csv`.

A2.3 Equilibrate

The network generation algorithm creates isolated filaments nearly equilibrated, but that's not necessarily the case for crowded networks. And when it comes to placing large beads like the one we want to use for microrheology, the algorithm is not too successful either. To remove potential overlaps of filaments and the larger bead, we need to carefully equilibrate the system.
We do that here by running Brownian dynamics simulations of the 6 networks with very small time steps for several thousand steps with the script `run_equilibrations.py`. These simulations might take 30 minutes up to several hours each, depending on the computing power of your CPU. I recommend you run as many of the processes as you can in parallel. It's 6 networks that we want to equilibrate, so 6 processes would be ideal. Pass the number of processes to be run in parallel to the script when you execute it in the command line.
The essential difference when setting up simulations with external particles lies in the arguments passed to the `Simulation` class:

```
s = Simulation(
    output_folder=output_folder,
    parameters=params,
    non_filament_particles={'bead': bead_diffusion_const},
    interaction_setup_handler=mr_bead_setup_handler
)
```

The `non_filament_particles` argument expects a dictionary mapping particle names (strings) to diffusion constants (floats). The `interaction_setup_handler` needs to be an instance of a class that inherited from `BaseSetupHandler`, and in its `__call__` method handles defining all rules for the external particles through the `system` parameter. In our example that is:

```
class MRBeadSetupHandler(BaseSetupHandler):

    ...

    def __call__(self, system: readdy.ReactionDiffusionSystem):
        for other in ['core', 'head', 'tail', 'motor']:
            system.potentials.add_harmonic_repulsion(
                "bead", other,
                force_constant=self._k_repulsion,
                interaction_distance=.5 + self._radius
             )

    ...
```

The function iterates over all filament particles and defines harmonic repulsion between them and the MR bead (`"bead"`).
The results of the equilibration simulations are stored in folder `initial_states_equilibrated`.

A2.4 Simulations

Now to the actual simulations. The simulations are run for all combinations of parameters defined in the variable `varied_parameters` in the script:

```
varied_parameters = {
    'rate_motor_bind': [0.0, 0.1, 0.25, 0.5],
    'initial_state_index': np.arange(6)
}
```

The function call to `_create_parameter_table` creates these combinations twice (due to multiplier 2) and stores them in a pandas DataFrame:

```
parameter_table = _create_parameter_table(
    varied_parameters, 2, offset
)
```

In the DataFrame, each of these $4 \cdot 6 \cdot 2 = 48$ parameter sets is assigned an unique index (sequential integer), that we then use to defer the output folder name.

The number of steps to be simulated (and related to that: the number of frames / data points to capture of the simulation) is defined in the file `default_params.json`. For this example, the number is set very low, to have the simulations finish within reasonable time. You can increase the number of steps, if you want to run proper scientific simulations (simulations of 100.000 steps take around 5-20 days on our compute cluster, ideally I would run all 48 simulations in parallel).

If you don't adjust them the default parameters regarding the number of simulated steps are

```
"n_steps": 100,
"observation_interval": 5
```

`n_steps` is the total number of simulated steps. Every 5 steps, one frame will be added to the resulting output file.

Once you are happy with the current parameters, execute the script. This will create the folder `simulations` with 48 subfolders `out0000` to `out0047`. In addition, an index table will be written to `simulations/simulation_index.csv`. This script as well expects you to pass the number of parallel processes as an argument.

A2.5 Reading the Data

Compared to the previous network example (section 4.1.5), we now have one more trajectory to deal with: that of the MR bead. This will be automatically included, when you use ReaDDy2's data reading facilities, for example:

```
import readdy
```

```
sim_index = 0

file_data = f"simulations/out{sim_index:04}/data.h5"
traj = readdy.Trajectory(file_data)
# this places a data.h5.xyz file next to the source data.h5 file:
traj.convert_to_xyz(generate_tcl=False)
```

If you want to read the trajectories, I recommend using the `actomyosin_analyser` python package (see section 3.5). The `Analyser` class provides methods to read external particle (`non_filament`) trajectories separately from filament coordinates. This way you don't have to deal with where external particle data is stored. The following code snippet reads the trajectories of the MR beads for the first three simulations, and uses a data to make a few scatter plots of the positions:

```
import os
import numpy as np
import matplotlib.pyplot as plt
from bead_state_model.data_reader import DataReader
from actomyosin_analyser.analysis.analyser import Analyser

simulation_indices = [0, 1, 2]

fig, ax = plt.subplots(1, len(simulation_indices))

for i, sim_index in enumerate(simulation_indices):

    sim_dir = os.path.join('simulations', f'out{sim_index:04}')
    dr = DataReader(os.path.join(sim_dir, 'data.h5'))
    # To Analyser class, provide a data_reader and the
    # path to the output file.
    a = Analyser(dr, os.path.join(sim_dir, 'analysis.h5'))

    # There is only one non-filament particle in these simulations,
    # the large bead for microrheo measurements.
    # select it with index 0:
    bead_traj = a.get_trajectories_non_filament()[:, 0]

    ax_i = ax[i]
    ax_i.scatter(
        bead_traj[:, 0], bead_traj[:, 1], s=0.5,
        cmap='viridis', label='sim {:02}'.format(sim_index),
        c=np.arange(len(bead_traj))
    )
    ax_i.legend()
    ax_i.set_aspect('equal')

plt.show()
```

A3 Performance of MR Simulations

A3.1 Varied Bead Size

As explained in section 3.1.5, polydisperse particle populations can have a large impact on performance of the simulations, since the number of interaction computations increases with the edge length of linked cells. In MR simulations, we have only two different particle sizes (the MR bead and the filament beads), and the choice of the (larger) MR bead size determines the edge length of the linked cells. The average run time of simulations increases exponentially with the MR bead radius, as shown in figure A3.1.

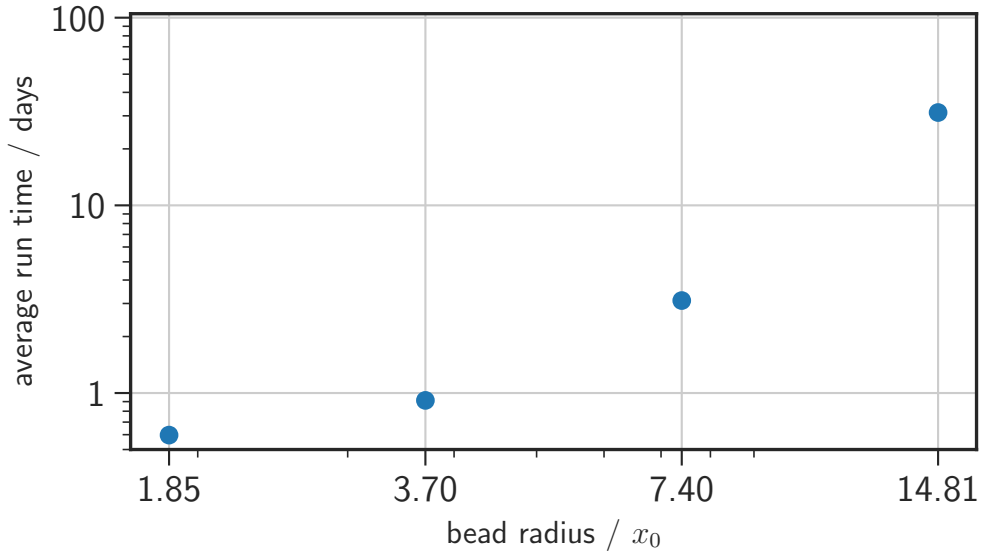

Figure A3.1: Run times of MR simulations with microspheres of different radii.

A3.2 Networks of Different Densities

In the context of MR simulations, I conducted experiments with entangled and cross-linked networks that were otherwise identical (section 6.3). However, they were conducted on different computers and can not directly be compared. Rather, I plotted only the run times against the number of filaments of the entangled networks in figure A3.2. The entangled network simulations were split onto two different machines as well, the batches are shown in two different colors in the plot. The majority of simulations was run on a slower processor (upper curve, green), that was able to run more simulations in parallel. The other processor (lower curve, blue) has faster single core speed, but had less cores available in total. Fits with

$$f(N; a, b, c) = a \cdot N^b + c$$

were performed (dashed lines), and revealed an exponent of $b \approx 1.6$. Without linked cells (or similar methods) we expect a relation $t \sim N^2$, while for

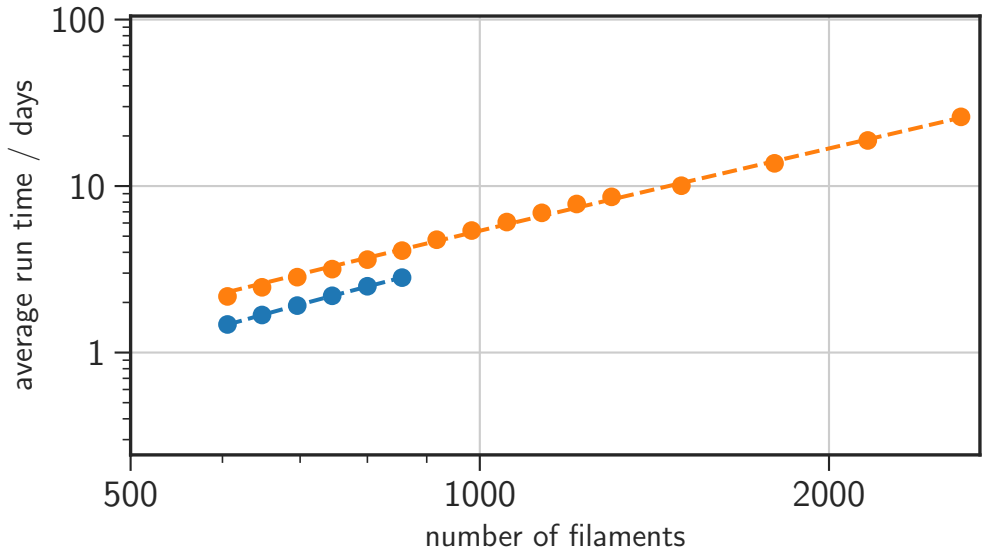

Figure A3.2: Run times of MR simulations with different actin densities (different number of actin filaments of uniform length).

monodisperse particles and linked cells, we expect a relation $t \sim N^b$ with $b_{\text{mono}} \gtrsim 1$. However, due to the MR bead increasing the edge length of the linked cells, the exponent increases to $b \approx 1.6$.

A4 Paust Fits

This section shows individual fits of the Paust fit model (equation 3.9) to ensemble MSDs of the microrheology probe beads simulated in chapter 6. Note that due to the weights of $1/\tau$, deviations from the fit at larger lag times are not penalized as much as deviations at lower lag times.
Plots use linear scales, which is not customary for MSD plots. But the intricacies of the fits for the lag time ranges used in this thesis lie especially in the details of the transition from nearly free diffusion to subdiffusion, which appears at the very low end of the lag time range when it appears at all. This can better be spotted on linear scales. For example, compare the plots for $a = 1.85x_0$ and $a = 2.20x_0$ with the rest of the plots in figure A4.2 . The large errors for $a = 1.85x_0$ for B, C, D show, that they are not taken into account for the fit. This is inverted for $a = 2.20x_0$, where A does not find any consideration. The fit model is not suited for these MSDs. The fits to the rest of the MSDs optimize all parameters. The difference between the shapes of the MSD curves is at low lag times, where all curves but those for $a = 1.85x_0, 2.20x_0$ show a transition from steep slopes to less steep slopes.

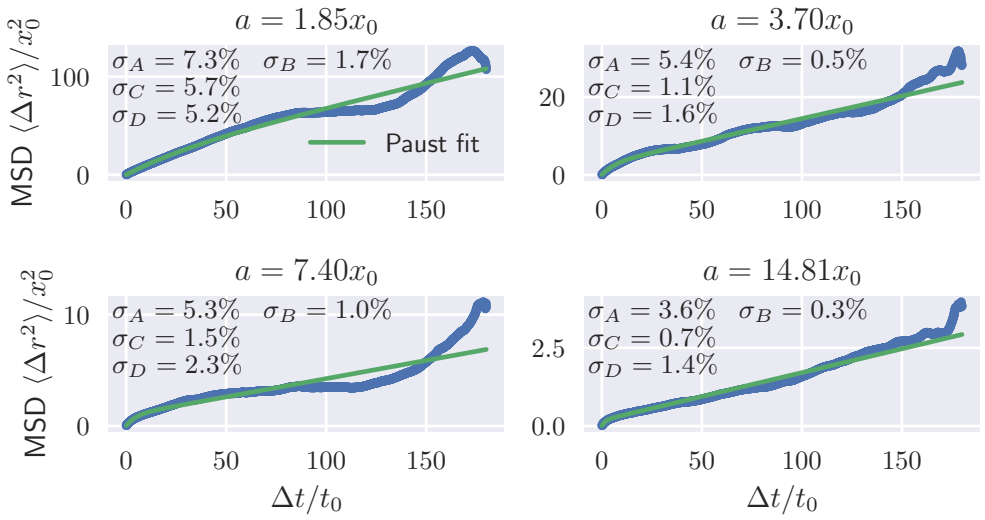

Figure A4.1: Paust model fits to ensemble MSDs of slippery MR beads (figure 6.2). Relative errors of fit parameters are shown in the plots.

A4.1 Vary Radius

A4.2 Vary Actin Concentration

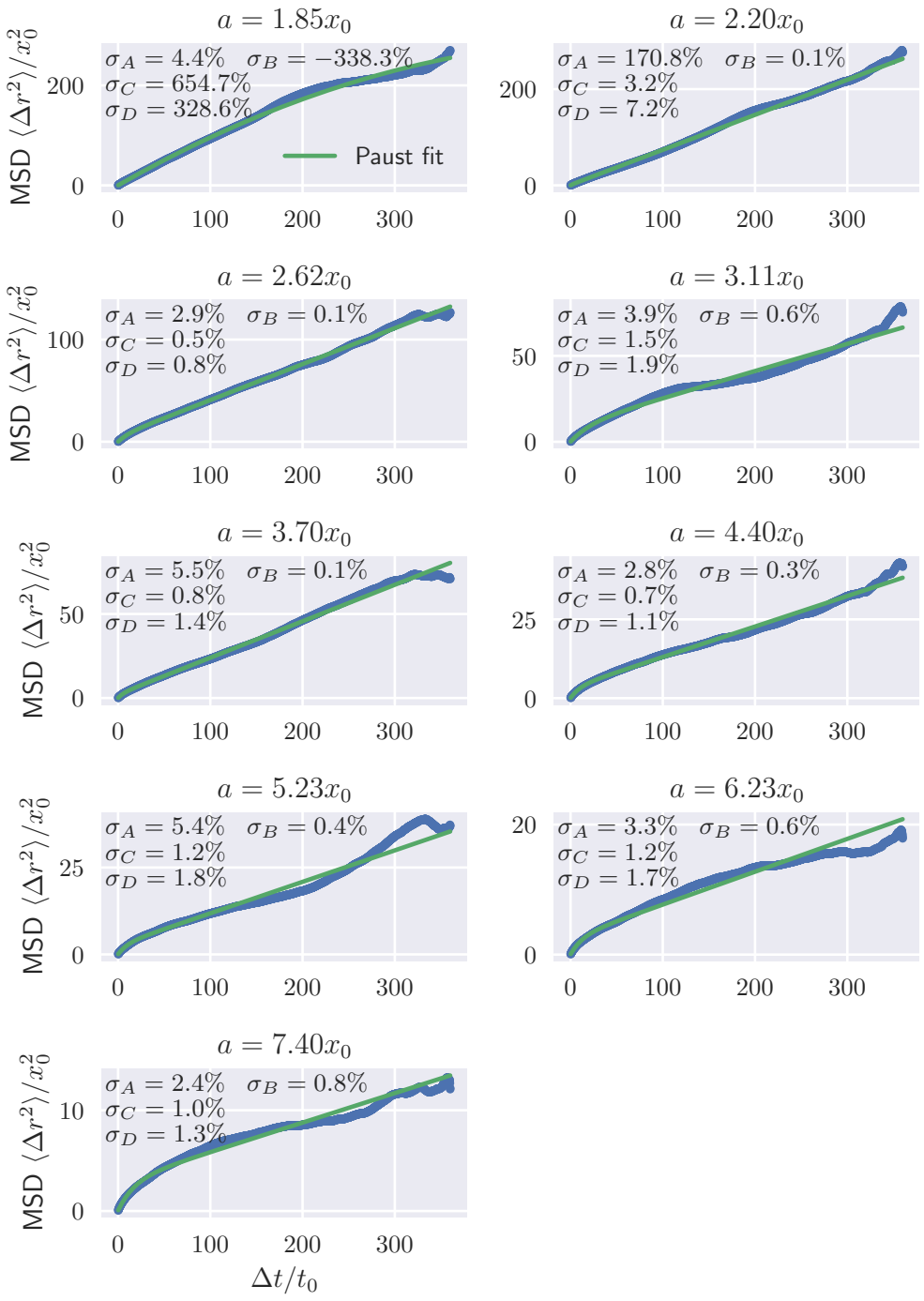

Figure A4.2: Paust model fits to ensemble MSDs of sticky MR beads (section 6.6). Relative errors of fit parameters are shown in the plots.

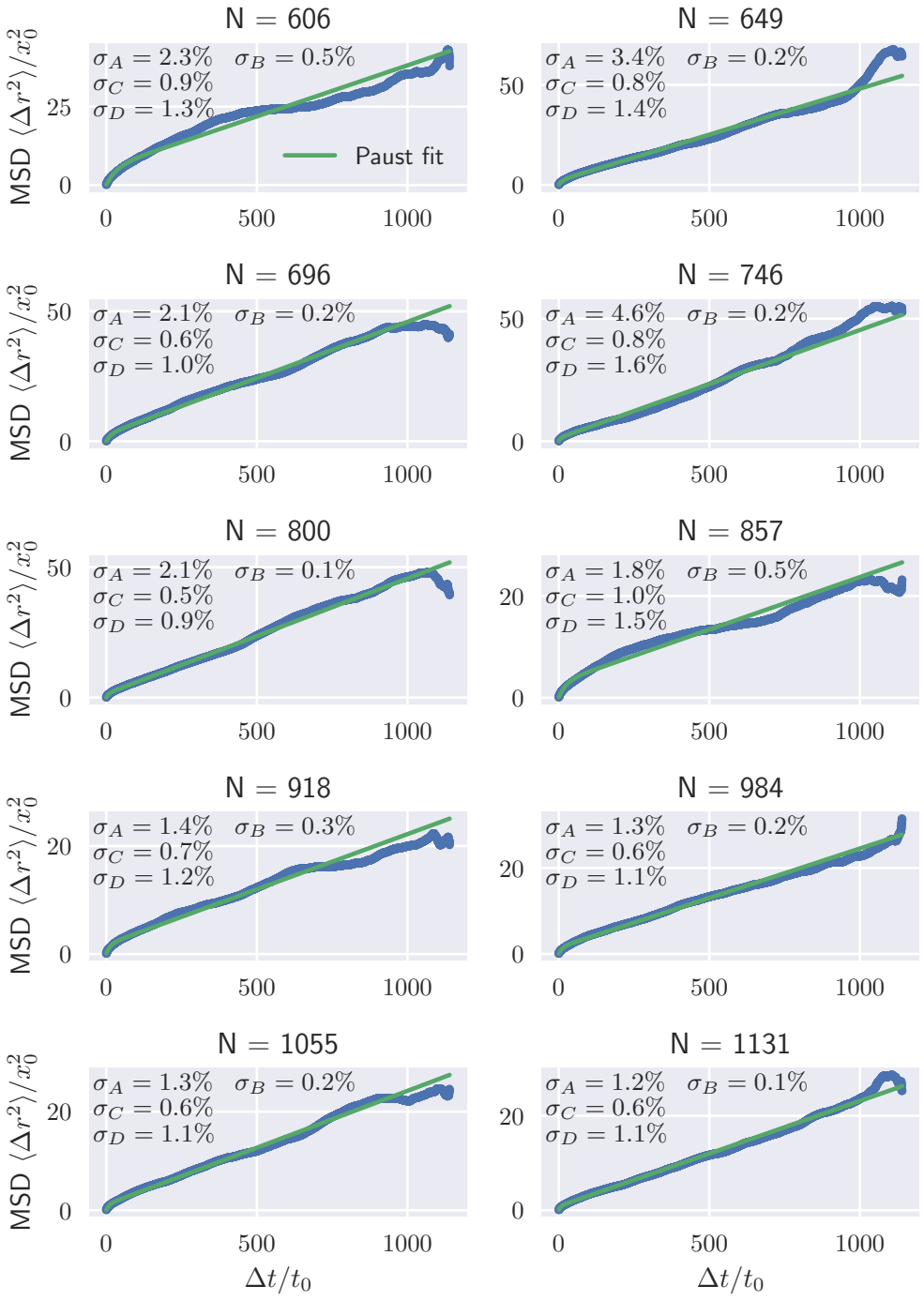

Figure A4.3: Paust model fits to ensemble MSDs of MR beads in entangled networks with varied number of filaments (figure 6.11). Relative errors of fit parameters are shown in the plots. This figure covers number of filaments from $N = 606$ to $N = 1131$, the rest are shown in the next figure.

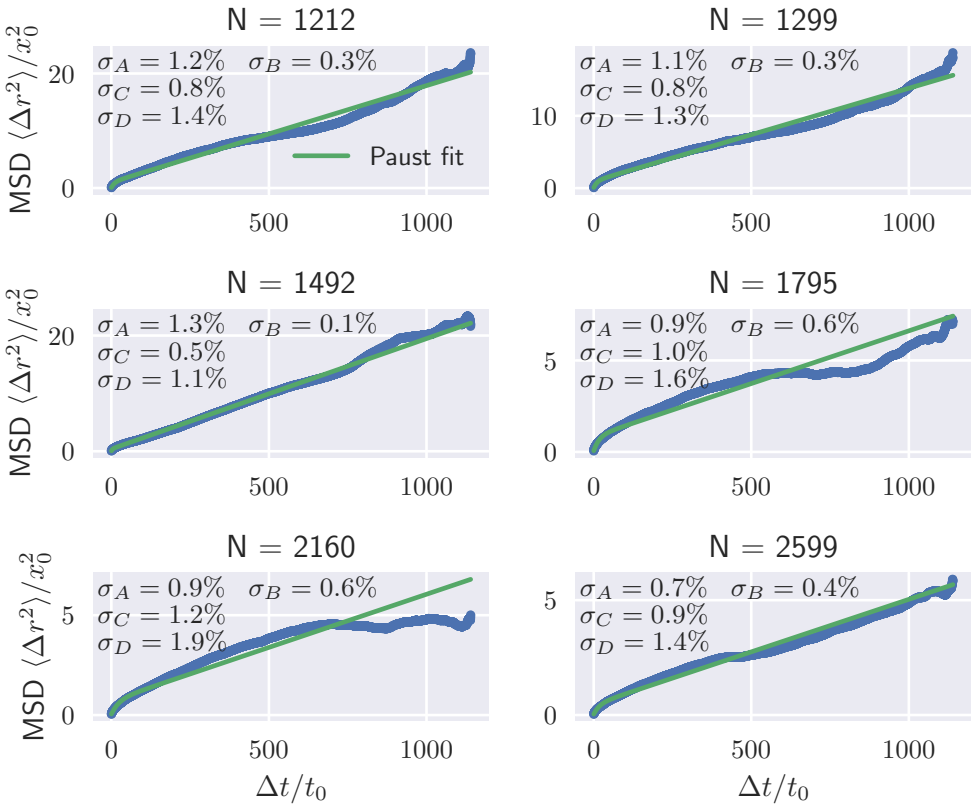

Figure A4.4: Paust model fits to ensemble MSDs of MR beads in entangled networks with varied number of filaments (figure 6.11). Relative errors of fit parameters are shown in the plots. This figure covers number of filaments from $N = 1212$ to $N = 2599$, the rest are shown in the previous figure.

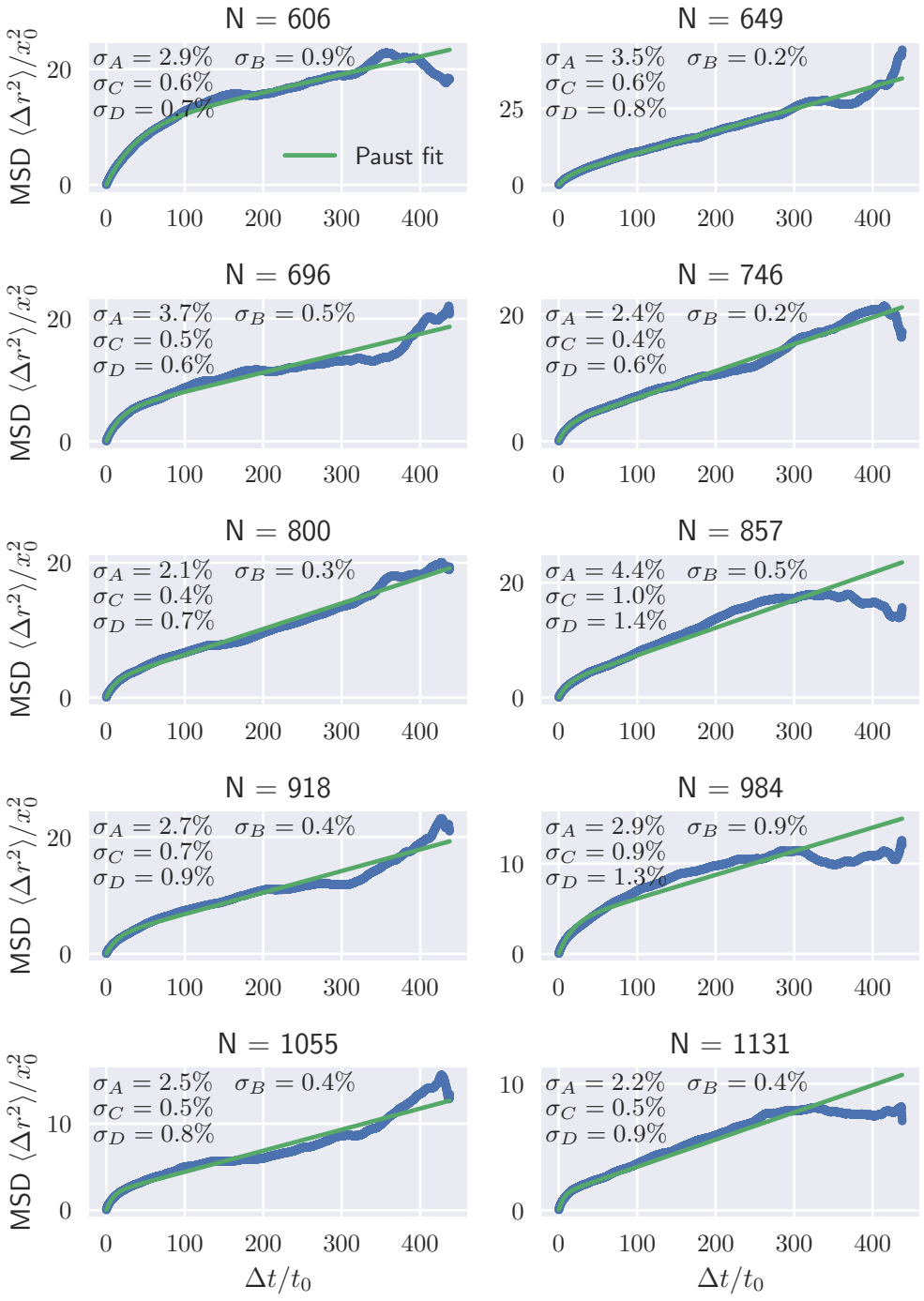

Figure A4.5: Paust model fits to ensemble MSDs of MR beads in cross-linked networks with varied number of filaments (figure 6.15). Relative errors of fit parameters are shown in the plots. This figure covers number of filaments from $N = 606$ to $N = 1131$, the rest are shown in the next figure.

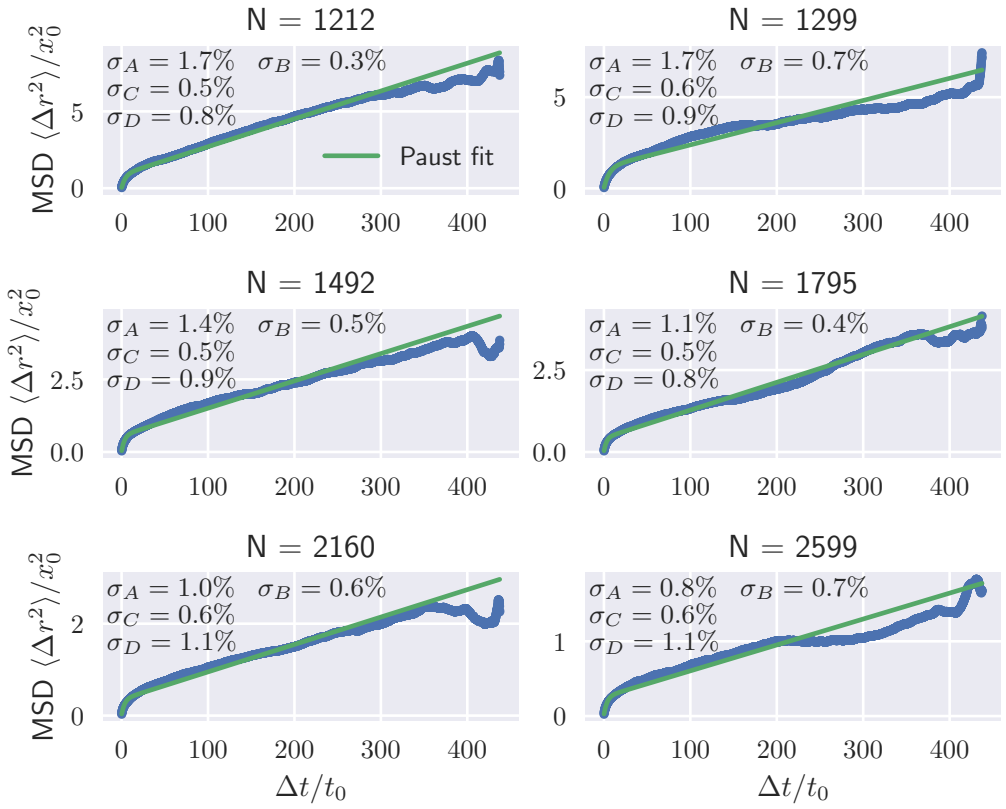

Figure A4.6: Paust model fits to ensemble MSDs of MR beads in cross-linked networks with varied number of filaments (figure 6.15). Relative errors of fit parameters are shown in the plots. This figure covers number of filaments from $N = 1212$ to $N = 2599$, the rest are shown in the previous figure.

A5 Microrheology Simulations with Cytosim

Cytosim[5] is a popular framework for simulations of actin and other components of the cytoskeleton [11, 17, 38, 94]. Utilizing built-in features, one can define spherical particles that can serve as passive microrheology (MR) probe beads. This theoretically allows for simulations similar to those performed with the `bead_state_model` simulation framework that were described in chapter 6. In collaboration with Tharaphat Ruamsukkunlasak, to assess whether MR simulations (with properties similar to those in section 6.3) are feasible with cytosim[6], the run time of cytosim MR simulations was analyzed and the MSD of MR beads was computed. During this assessment, simulations were limited to 10^4 steps, in contrast to the $2 \cdot 10^5$ and $4 \cdot 10^5$ steps used in section 6.3.

Simulations were conducted with MR probe beads of radius $a = 2$ µm and

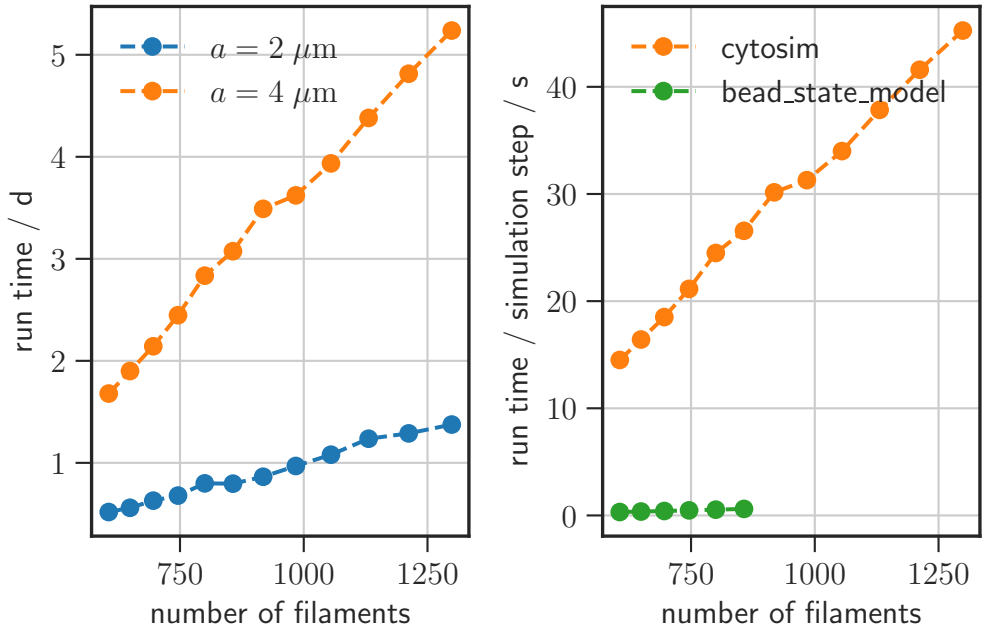

Figure A5.1: **(Left)** Run time of MR simulations with microspheres of different radii using cytosim. Run times are shown for simulations with varying number of filaments. **(Right)** The time it takes for one simulated time step with cytosim compared with `bead_state_model`. Data is from simulations with microsphere radius $a = 4$ µm.

$a = 4$ µm. The MR beads were made *sticky*, by defining attractive forces between MR beads and filaments, when the distance d between their centers is in range $a < d < a + 0.01$ µm. The radius of the MR beads had a large impact on the performance of the simulations, likely due to similar reasons as discussed in section A3 (the large MR bead increases the size of the linked cells). The run times are shown in figure A5.1 and compared to `bead_state_model` simulations.

[5] Project page: `https://gitlab.com/f-nedelec/cytosim`

[6] Processed data and configuration files to reproduce raw data are available here: `https://doi.org/10.25625/EPTSUK`.

Increasing the bead by a factor of 2 from $a = 2$ µm to $a = 4$ µm leads to an increase in run time by a factor of 3-4.
For comparison to `bead_state_model` simulations, only the MR beads of radius $a = 4$ µm were used, as they approximately resemble those used in section 6.3. Run times per simulation step are shown in figure A5.1 (right-hand side). Of the `bead_state_model` simulations, only data of those conducted on an identical CPU as the cytosim simulations were used. Comparing the run times, one can see that cytosim takes much more time to compute one simulation step (approximately 45 times longer). From the performance comparison, it can be concluded that `bead_state_model` is more suitable for MR simulations of entangled networks.
For networks including cross-links or motors, this would have to be evaluated separately (has not been done yet). Cross-links and motors severely increase the rum time with `bead_state_model` (roughly by factor 8, due to slow pure python implementation), but only slighly increase the run time with cytosim.

Using the `actomyosin_analyser` package (see section 3.5), the MSDs of MR beads were computed from their trajectories and are shown in figure A5.2. The

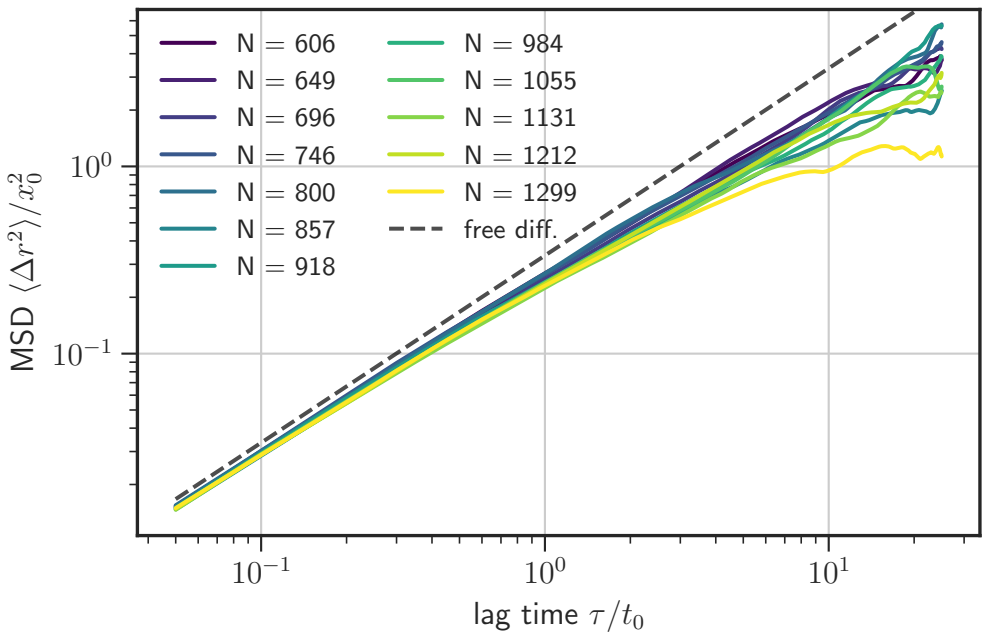

Figure A5.2: cytosim MSDs

MSDs show nearly free diffusion (close to dashed line with slope 1), except for the highest number of filaments, where the slope of the MSD decreases at large lag times. This behaviour is different from MR beads in `bead_state_model` simulations (see figure 6.11), where MSDs exhibit subdiffusive slopes for lower filament numbers as well. This is likely caused by a difference in the models of actin filaments. Cytosim uses very thin filaments, with diameters more closely resembling real cytoskeletal filaments. In contrast, in `bead_state_model` filaments consist of large beads, which leads to rather large filament diame-

ters. Large diameters lead to comparably large volumes occupied by filaments, which in turn lead to a decreased mesh size. To achieve similar ratios of MR bead radius to mesh size as in `bead_state_model`, in cytosim we would have to either increase the number of actin filaments, increase the MR bead radius, or increase the diameter of the filaments. These options have not yet been further explored.

In conclusion, while cytosim can be used to conduct MR simulations, the simulations perform worse (which has been demonstrated for entangled networks), and require further adjustment to either decrease the mesh size or increase the MR bead diameter.

A6 Contributions to ReaDDy

The actomyosin simulation framework presented in this thesis, `bead_state_model`, is based on ReaDDy [64]. ReaDDy is a general particle simulation framework and was not suitable for `bead_state_model` without some modifications. These modifications were contributed to the ReaDDy project in the form of two feature pull requests on ReaDDy's github page, and are now part of ReaDDy's code.

In order to depict my contributions in more detail than in chapter 4.1, the changes made in the two aforementioned pull requests are described in the following two subsections.

A6.1 Self-fusion with Distance Threshold

Problem Definition

In the bead-state model, two polymers that are touching (or almost touching, the reaction range can be specified and is set to 1.05 bead diameters per default) can form a link, either via cross-links or motors. More precisely, the link is formed between two beads of the state `core` of the bead chains, and the new state `link` or `motor` is assigned to the beads between which the bond is formed. Using the tools provided by ReaDDy, one can define this type of reaction as a spatial topology reaction:

```
system.topologies.add_spatial_reaction(
  "link: polymer(core) + polymer(core)"
  "-> polymer(motor--motor)", ...
)
```

In ReaDDy, a topology is a structure consisting of two or more particles with bonds between them. In the example code snippet above, the reaction is named `link`, the topology type is named `polymer`. The particle type of the particles between which the reaction is defined is `core`. After the reaction, the two topologies merge into one topology of type `polymer`, and the type of the particles changes to `motor`.
For two polymers, the above code would suffice. However, in a network of

polymers, a problem occurs: the two polymers that would be elligible for the bond formation, might be part of the same topology, which would prohibit the reaction. This would lead to situations as depicted in figure 4.3 (left).
ReaDDy has the option to allow reactions within topologies via the self flag:

```
system.topologies.add_spatial_reaction(
  "link: polymer(core) + polymer(core)"
  "-> polymer(motor--motor) [self=true]", ...
)
```

However, this would allow two adjacent `core` beads of the same polymer to turn into `links/motors`, which would quickly prohibit polymers to form links with other polymers at all (since almost no beads of type `core` would remain).

Contributed Solution

To solve the problem stated above, I implemented the optional addition of a `distance` parameter to the self flag:

```
system.topologies.add_spatial_reaction(
  "link: polymer(core) + polymer(core)"
  "-> polymer(motor--motor) [self=true, distance>6]",
  ...
)
```

In the example above, which is part of the `bead_state_model` code base, the parameter `self=true` allows the reaction to take place within the same topology, while the parameter `distance>6` limits this reaction to beads that have at least 6 bonds (edges in the topology graph) between them. This prevents keeps beads within the same polymer from turning into a link pair. The resulting desired behaviour is depicted in figure 4.3 (right).

To implement the solution above, I first implemented the feature of the distance parameter with a slightly different syntax. Then I created a pull request [7] on ReaDDy's project repository, and made some slight changes to the interface as requested by the ReaDDy developers. The page of the pull request states that my contributions in their final form consisted of 385 new lines of code and 32 removed lines of code. These changes were made at 11 C++ files and two config files. Included in these changes were automated tests that were passed in ReaDDy's automated continuous integration pipeline. The full conversations with the ReaDDy developers and all changes I made to the ReaDDy code are fully disclosed at the publically available pull request page. In addition, some excerpts of the contributed code are shown below.

Excerpts of Contributed Code

In order to determine whether two particles (referred to as vertices of the graph here) are connected via n or less bonds (referred to as edges of the graph), a

[7] Pull request #171: `https://github.com/readdy/readdy/pull/171`

method was added to the `Graph` class. The following excerpt is from the file `readdy/main/model/topologies/graph/Graph.cpp`:

```
bool Graph::areConnectedWithNOrLessEdges(
  std::size_t n,
  const Vertex &v1,
  const Vertex &v2
) const {
  if (n==0) return false;
  bool found = false;
  for (const auto neigh: v1.neighbors()) {
    if (neigh->particleIndex == v1.particleIndex) continue;
    if (neigh->particleIndex == v2.particleIndex) {
      return true;
    }
    found = areConnectedWithNOrLessEdges(n-1, *neigh, v2);
    if (found) return true;
  }
  return false;
}
```

The method above comes to bare when reaction candidates are evaluated. There can be multiple compute kernel implementations of these evaluations [64]. At the time of my contribution, ReaDDy had two compute kernels, a sequential one named `SCPU` and a multiprocessing one named `CPU`. The following excerpt from file `kernels/cpu/src/actions/CPUEvaluateTopologyReactions.cpp` is part of `CPU` implementation. When the two particles under consideration for the reaction are connected with n or less edges, the reaction is skipped (`reaction_index` is incremented and `continue` is invoked to jump to the next reaction in queue) since it prohibted by the `distance` parameter:

```
if (reaction.allow_self_connection() &&
  entry.topology_index == neighbor.topology_index) {
  const auto& topol = model.topologies().at(
    static_cast<std::size_t>(neighbor.topology_index)
  );
  const readdy::model::top::graph::Graph& gr = topol->graph();
  const readdy::model::top::graph::Graph::vertex_ref& v1 =
   ↪ topol->vertexForParticle(event.idx1);
  const readdy::model::top::graph::Graph::vertex_ref& v2 =
   ↪ topol->vertexForParticle(event.idx2);
  if (gr.areConnectedWithNOrLessEdges(
        reaction.min_graph_distance(), *v1, *v2)
  ) {
    ++reaction_index;
    continue;
  }
}
```

Another important part of the feature contribution is in file `readdy/main/model/topologies/reactions/SpatialTopologyReaction.cpp` as part of the class `STRParser`. It deals with the parsing of the reaction descriptor language, i.e. the string containing the options `[self=true, distance>6]`. The following snippet looks for options in square brackets and determines whether the self and distance parameters are present. Since at the time of implementation the self parameter always had to be present, a runtime error is thrown if it is not:

```
191 if(std::regex_search(rhs, option_match, option_rx)) {
192   auto option_str = option_match.str();
193   std::vector<std::string> options = parse_options(
194     option_str.substr(1, option_str.length()-2)
195   );
196   if (!has_option_allow_self(options)) {
197     throw std::runtime_error(ERROR_MESSAGE_SELF_OPTION_MISSING);
198   }
199   reaction._mode = STRMode::TT_FUSION_ALLOW_SELF;
200   int d = get_distance(options);
201   if (d == -1) reaction._min_graph_distance = 0;
202   else reaction._min_graph_distance = (unsigned) d;
203 } else {
204   reaction._mode = STRMode::TT_FUSION;
205 }
```

For readability, several steps of the excerpt above were implemented in separate methods:

```
218 std::vector<std::string> STRParser::parse_options(const
 ↪  std::string &option_str) const {
219   std::vector<std::string> options;
220   size_t pos = 0;
221   std::string s = option_str.substr();
222   std::string o;
223   while ((pos = s.find(",")) != std::string::npos) {
224     o = s.substr(0, pos);
225     readdy::util::str::trim(o);
226     options.push_back(o);
227     s.erase(0, pos + 1);
228   }
229   readdy::util::str::trim(s);
230   options.push_back(s);
231   return options;
232 }
233
234 bool STRParser::has_option_allow_self(const
 ↪  std::vector<std::string> &options) const {
```

```
  auto it = std::find(options.begin(),
                      options.end(),
                      "self=true");
  if (it != options.end()) return true;
  return false;
}

int STRParser::get_distance(const std::vector<std::string>
 ↪  &options) const {
  for (auto &o: options) {
    if(o.find("distance>") == std::string::npos) continue;
    auto dist = std::stoi(o.substr(9));
    return dist;
  }
  return -1;
}
```

A6.2 Dynamic Rates for Link Binding

Problem Definition

The reaction that allows core beads to form a bond and turn into a link/motor pair is implemented as a spatial topology reaction in ReaDDy. The rate for spatial topology reactions could only be specified as a constant value. This would lead to the following issues for implementing bead-state model with ReaDDy:

1. The link binding rate would never decrease, no matter how many links are already formed. This would correspond to a system with an infinite reservoir of cross-links or motors. This also means that the number of links and motors cannot be capped.

2. The total number of links in the system would be very hard to estimate. As filaments form links, they are likely to stay in close contact (unless the link bond breaks right away, but with relevant binding and unbinding rates, eventually some polymers would form a bond for non-negligible amounts of time). This close proximity over a long period of time would create more possibilities for more bonds to form. With uncapped number of links, eventually all beads of the filaments would be linked to neighboring filaments.

Contributed Solution

I implemented the option to use a function that computed a current rate instead of a fixed rate. This feature was implemented in ReaDDy in C++, but I also adapted the Python user interface to make the feature available through Python. This allowed me to define an upper limit for the number of links/motors and a linear decrease of the rate with the number of presently formed links/motors, as

described in section 4.1.2 in the "Reaction Rates" subsection. This usage of the contributed feature to ReaDDy in `bead_state_model` is only one example. The feature was implemented in a generalized way that allows any kind of function that returns a floating point number to be used as rate for spatial topology reactions.

The feature contribution started with a discussion with the ReaDDy developers after I created an issue[8] on their project repository. After this discussion I started working on the feature implementation and created a pull request[9].
The page of the pull request states that my contributions in their final form consisted of 506 new lines of code and 208 removed lines of code. These changes were made at 8 files of the C++ code base, 5 Python and C++ files of the C++/Python interface, and several config files and submodules. Included in these changes were automated tests that were passed in ReaDDy's automated continuous integration pipeline for the C++ code as well as the C++/Python interface. The full conversations with the ReaDDy developers and all changes I made to the ReaDDy code are fully disclosed at the publically available pull request page. In addition, some excerpts of the contributed code are shown below.

Excerpts of Contributed Code

Most of the changes I made on the C++ side of ReaDDy are scattered over multiple files in many different places, and cannot be displayed here nicely. For those changes I want to refer again to the pull request page (see above). Instead, I want to highlight the changes to the C++/Python interface and the Python user interface here.
At the interface of C++ and Python, for which ReaDDy uses the pybind11 library[10], I introduced a new method `add_spatial_reaction_with_rate_function` to the Python class `TopologyRegistry`, which is defined in file `wrappers/python/src/cxx/api/ExportKernelContext.cpp`:

```
187 py::class_<TopologyRegistry>(module, "TopologyRegistry")
188   .def("add_spatial_reaction_with_rate_function",
189        [](TopologyRegistry &self, const std::string &descriptor,
190           spatial_rate_function_sink rate_function,
191           scalar radius) {
192            self.addSpatialReaction(descriptor, rate_function,
                ↪ radius);
193        }
194   )
```

The `spatial_rate_function_sink` is defined in `wrappers/python/src/cxx/api/PyTopology.h` to be a callable that takes two graph topologies as inputs and returns a `rate_function::result_type`, which is a double precision float in most cases:

[8] Issue #191: `https://github.com/readdy/readdy/issues/191`
[9] Pull request #192: `https://github.com/readdy/readdy/pull/192`
[10] Project repository: `https://github.com/pybind/pybind11`

```
struct spatial_rate_function_sink {
  std::shared_ptr<pybind11::function> f;
  explicit spatial_rate_function_sink(const pybind11::function& f)
   ↪  : f(std::make_shared<pybind11::function>(f)) {};

  inline readdy::model::top::reactions::SpatialTopologyReaction::\
    rate_function::result_type operator()(
          const readdy::model::top::GraphTopology& top1,
          const readdy::model::top::GraphTopology& top2
  ) {
      pybind11::gil_scoped_acquire gil;
      PyTopology pyTop1
       ↪  (&const_cast<readdy::model::top::GraphTopology&>(top1));
      PyTopology pyTop2
       ↪  (&const_cast<readdy::model::top::GraphTopology&>(top2));
      auto t1 = pybind11::cast(&pyTop1,
       ↪  pybind11::return_value_policy::automatic_reference);
      auto t2 = pybind11::cast(&pyTop2,
       ↪  pybind11::return_value_policy::automatic_reference);
      auto rv = (*f)(*(t1.cast<PyTopology*>()),
       ↪  *(t2.cast<PyTopology*>()));
      return rv.cast<readdy::model::top::reactions::\
        SpatialTopologyReaction::rate_function::result_type>();
  }
};
```

To make sure that the implemented feature works as intended through the Python user interface, `wrappers/python/src/python/readdy/tests/test_topology_reactions.py` contains an acceptance test that gets executed automatically as part of ReaDDy's continuous integration pipeline. The methods that get executed are first two that start with the `test_` prefix. The other defined methods get called by these first two:

```
def test_spatial_reaction_rate_function_SingleCPU(self):
    self.spatial_reaction_rate_function("SingleCPU")

def test_spatial_reaction_rate_function_CPU(self):
    self.spatial_reaction_rate_function("CPU")

@staticmethod
def spatial_reaction_rate_function(kernel):
    """
    Create a small simulation of 4 polymers that
    form arranged as a square. Heads of the polymers
    are close, such that they could bind.
    The rate function should allow only two
```

```
    of the polymers to bind, so that we end up with
    2 pairs of two connected polymers.
    """
    def rate_function(top1, top2):
        vert1 = top1.get_graph().get_vertices()
        vert2 = top2.get_graph().get_vertices()
        if len(vert1) + len(vert2) > 12:
            return 0.0
        return 1e10

    system = readdy.ReactionDiffusionSystem(box_size=[30., 30.,
    ↪  30.])
    system.topologies.add_type("Polymer")
    system.add_topology_species("Head", 0.002)
    system.add_topology_species("Core", 0.002)

    system.topologies.configure_harmonic_bond("Head", "Core",
    ↪  force_constant=50, length=1.)
    system.topologies.configure_harmonic_bond("Core", "Core",
    ↪  force_constant=50, length=1.)

    system.topologies.add_spatial_reaction(
        "Association: Polymer(Head) + Polymer(Head) ->
        ↪  Polymer(Core--Core)",
        rate=rate_function, radius=2.0
    )

    simulation = system.simulation(kernel=kernel)
    types_and_positions =
    ↪  TestTopologyReactions._get_polymer_types_and_positions()
    for t, p in types_and_positions:
        top = simulation.add_topology("Polymer", t, p)
        for i in range(5):
            top.get_graph().add_edge(i, i+1)

    simulation.run(10, 1.)

    np.testing.assert_equal(2, len(simulation.current_topologies))

@staticmethod
def _get_polymer_types_and_positions():
    """
    Construct a square of 4 polymers.
    """
    types_and_positions = [
      TestTopologyReactions._get_types_and_positions_polymer_1(),
```

```
      TestTopologyReactions._get_types_and_positions_polymer_2(),
      TestTopologyReactions._get_types_and_positions_polymer_3(),
      TestTopologyReactions._get_types_and_positions_polymer_4()
    ]

    return types_and_positions

@staticmethod
def _get_types_and_positions_polymer_1():
    types = ["Head"] + ["Core"]*4 + ["Head"]
    positions = np.zeros((6, 3))
    positions[:, 0] = np.arange(6)
    return types, positions

@staticmethod
def _get_types_and_positions_polymer_2():
    types = ["Head"] + ["Core"]*4 + ["Head"]
    positions = np.zeros((6, 3))
    positions[:, 0] = 5
    positions[:, 1] = np.arange(0, -6, -1)
    positions[:, 2] = 1
    return types,positions

@staticmethod
def _get_types_and_positions_polymer_3():
    types = ["Head"] + ["Core"]*4 + ["Head"]
    positions = np.zeros((6, 3))
    positions[:, 0] = np.arange(5, -1, -1)
    positions[:, 1] = -5
    return types, positions

@staticmethod
def _get_types_and_positions_polymer_4():
    types = ["Head"] + ["Core"]*4 + ["Head"]
    positions = np.zeros((6, 3))
    positions[:, 1] = np.arange(6) - 5
    positions[:, 2] = 1
    return types, positions
```

A7 Code Listings

L1 Source Code Excerpt: Cut-off distance in ReaDDy's Linked Cells

The following code was copied from file `readdy/main/model/Context.cpp` (ReaDDy version 2.0.12). It implements the `calculcateMaxCutoff` method of the `Context` class. The `max_cutoff` is determined as the maximum cut-off across all interactions / potentials and reactions.

```
scalar Context::calculateMaxCutoff() const {
  scalar max_cutoff{0};
  for (const auto &entry : potentials().potentialsOrder2()) {
      for (const auto &potential : entry.second) {
          max_cutoff = std::max(max_cutoff,
                                potential->getCutoffRadius());
      }
  }
  for (const auto &entry : reactions().order2()) {
      for (const auto &reaction : entry.second) {
          max_cutoff = std::max(max_cutoff,
                                reaction->eductDistance());
      }
  }
  for (
    const auto &entry : _topologyRegistry.spatialReactionRegistry()
  ) {
      for (const auto &reaction : entry.second) {
          max_cutoff = std::max(max_cutoff, reaction.radius());
      }
  }
  return max_cutoff;
}
```

L2 Source Code Excerpt: Definition of Link Binding

The following excerpt was taken from the `bead_state_model` source code. It defines the reaction between two polymer/filament particles of type `core`, creating a bond between them, and turning them into particles of type `motor`.

```
if rate_motor_bind > 0:
    if n_max_motors is None:
        rate = rate_motor_bind
    else:
        # If n_max_motors is not None, a dynamic function
        # to determine the rate has to be defined.
        # Function takes into account the topologies
```

```
        # top1 and top2, between which the motor binding
        # is about to occur.
        def _rate_func(top1, top2):
            # fh is an instance of the FilamentHandler class
            return fh.rate_function_max_motor_count(
                top1, top2, rate_motor_bind,
                n_max_motors
            )
        rate = _rate_func
    react_str = ("link: filament(core) + filament(core) "
                 "-> filament(motor--motor)"
                 f"[self=true, distance>{min_network_distance-1}]")

    system.topologies.add_spatial_reaction(
        react_str, rate=rate, radius=reaction_radius_motor_binding
    )
```

L3 Source Code Excerpt: Definition of Motor Step Recipe

The following excerpt was taken from the `bead_state_model` source code. It contains the function/method of the `FilamentHandler` class, that creates the recipe for the motor step reaction. Recipes allow users of the ReaDDy simulation framework [64] to define arbitrary reactions within one topology. A topology is a set of particles connected via bonds between those particles. A recipe can be used to add bonds (`add_edge`), remove bonds (`remove_edge`), and change particles types.

```
def reaction_function_motor_step(
        self,
        topology
) -> readdy.StructuralReactionRecipe:
    recipe = readdy.StructuralReactionRecipe(topology)
    top_id = self._get_topology_id(topology)
    motors = self._get_motors(top_id)
    if len(motors) == 0:
        return recipe
    self._update_required = True

    map_pid_to_vid =\
        self._build_map_particle_id_to_vertex_index(topology)
    motor_id = motors[
        np.random.choice(len(motors))
    ]
    fwd_neighbor_id = self.links[motor_id, 1]
    if self.links[fwd_neighbor_id, 1] == -1:
        # forward neighbor is head of filament: do nothing
```

```
        return recipe
    if self.links[fwd_neighbor_id, 2] != -1:
        # forward neighbor is cross-link or motor: do nothing
        return recipe

    linked_motor_id = self.links[motor_id, 2]
    recipe.remove_edge(map_pid_to_vid[motor_id],
                       map_pid_to_vid[linked_motor_id])
    self.links[motor_id, 2] = -1
    recipe.change_particle_type(map_pid_to_vid[motor_id], "core")
    recipe.change_particle_type(map_pid_to_vid[fwd_neighbor_id],
                                "motor")
    recipe.add_edge(map_pid_to_vid[fwd_neighbor_id],
                    map_pid_to_vid[linked_motor_id])
    self.links[fwd_neighbor_id, 2] = linked_motor_id
    self.links[linked_motor_id, 2] = fwd_neighbor_id
    self.map_topology_id_to_motors[top_id].remove(motor_id)
    self.map_topology_id_to_motors[top_id].append(fwd_neighbor_id)

    return recipe
```

L4 Example: Create Network

Script `create_network.py` of the example in section 4.1.5 It creates a network of filaments.

```
import os
import time
import numpy as np
from bead_state_model.network_assembly.create_network_simple \
    import CreateNetworkSimple


os.makedirs('initial_state', exist_ok=True)
np.random.seed(np.uint32(hash(str(time.time()))))

"""
CreateNetworkSimple generates random entangled networks,
i.e. filaments without any cross-links or motors.
Reduce the number of filaments and/or beads if you want to
speed up the network assembly and the simulation.
"""

n_filaments = 300
n_beads_per_filament = 15
```

```
# size of simulation BOX
box = np.array([20., 20., 20.])

# parameters for bending (determines persistence length) and
# stretching between beads
k_bend = 26.0
k_stretch = 20.0

system = CreateNetworkSimple(
    box=box,
    k_bend=k_bend,
    k_stretch=k_stretch,
    n_filaments=n_filaments,
    n_beads_per_filament=n_beads_per_filament
)

system.run(verbose=True)
system.write(folder='initial_state')
print("done")
```

L5 Example: Run Simulation

Script `run_simulation.py` of the example in section 4.1.5. It runs a simulation of the network created with the script in the previous listing.

```
import os
from bead_state_model import Simulation, Parameters

def main():
    if not os.path.exists('initial_state'):
        print("No folder initial_state found. Run the script "
              "create_network.py before run_simulation.py")
        return

    box, k_stretch, k_bend =\
        Parameters.load_parameters_from_generated_network(
            'initial_state/config.json'
        )

    parameters = Parameters(
        box_size=box,
        k_bend=k_bend,
        k_stretch=k_stretch,
        n_beads_max=5000,
        rate_motor_bind=0.1,  # set this to 0 if you want to
                              # disable links
```

```
        rate_motor_step=0.0,  # assign a positive value here for
                              # links to behave like motors
                              # instead of passive cross-links
        n_max_motors=500  # this limits the maximum number of
                          # motors/links in a simulation
    )

    s = Simulation(output_folder='simulation_output',
                   parameters=parameters)

    # Add filaments/beads from file initial_state/beads.txt .
    s.add_filaments('initial_state/beads.txt')

    # record one frame every 100 time steps
    observation_interval = 100
    n_steps = 2000
    dt = 0.006
    s.run(n_steps, dt, observation_interval)


if __name__ == "__main__":
    main()
```

L6 Plot 10 randomly selected filaments

The following code was used to generate the plot in figure 4.6.

```
import matplotlib.pyplot as plt
import numpy as np

from bead_state_model.data_reader import DataReader
from actomyosin_analyser.analysis.analyser import Analyser

dr = DataReader('simulation_output/data.h5')
analyser = Analyser(dr, 'simulation_output/analysis.h5')
coordinates = analyser.get_trajectories_filaments()
filaments = analyser.get_filaments()

# select 10 indices
selected_indices = np.random.choice(
    np.arange(len(filaments[0])),
    10
)

for idx in selected_indices:
    c_idx = coordinates[0, filaments[0][idx].items]
```

```
    x = c_idx[:, 0]
    y = c_idx[:, 1]
    plt.plot(x, y, '-o')

plt.xlabel('$x/x_0$')
plt.ylabel('$y/x_0$')

plt.gca().set(
    aspect='equal'
)

plt.show()
```

L7 Plot length distributions

The following code was used to generate the plot in figure 4.7.

```
from typing import Union, List, Tuple
import argparse

import matplotlib.pyplot as plt
import numpy as np

from bead_state_model.data_reader import DataReader
from actomyosin_analyser.analysis.analyser import Analyser
from actomyosin_analyser.analysis import geometry
from scipy.stats import gaussian_kde


def main(frames: List[int],
         style: Union[str, None]):
    if style is not None:
        plt.style.use(style)

    fig, ax = plt.subplots(1, 1)

    dr = DataReader('simulation_output/data.h5')
    analyser = Analyser(dr, 'analysis.h5')

    _plot_length_distributions(ax, analyser, frames)

    fig.tight_layout()
    fig.savefig('distributions.svg')


def _plot_length_distributions(
```

```
        ax: plt.Axes,
        analyser: Analyser,
        frames: List[int]
):
    kdes = []
    global_min = 1e5
    global_max = 0.0
    for f in frames:
        kde, min_, max_ = _get_lengths_kde_single_frame(
            analyser, f)
        global_min = min(global_min, min_)
        global_max = max(global_max, max_)
        kdes.append(kde)

    x = np.linspace(global_min, global_max, 200)

    for i, f in enumerate(frames):
        kde = kdes[i]
        ax.plot(x, kde(x), label=f'frame {f:02}')

    ax.legend()
    ax.set(
        xlabel='$l / x_0$',
        ylabel='density',
        title='Length distributions'
    )


def _get_lengths_kde_single_frame(
        analyser: Analyser,
        frame: int
) -> Tuple[gaussian_kde, float, float]:
    coords = analyser.get_trajectories_filaments()[frame]
    filaments = analyser.get_filaments()[frame]

    box = analyser.data_reader.read_box_size()
    box = np.array([box[1, 0] - box[0, 0],
                    box[1, 1] - box[0, 1],
                    box[1, 2] - box[0, 2]])

    lengths = []

    for f in filaments:
        fcoords = coords[f.items]
        v_segments = geometry.get_minimum_image_vector(
            fcoords[:-1], fcoords[1:], box)
        d = np.sqrt(np.sum(v_segments**2, 1))
```

```
        lengths.append(d)

    lengths = np.concatenate(lengths)

    kde = gaussian_kde(lengths)

    return kde, lengths.min(), lengths.max()


if __name__ == "__main__":

    parser = argparse.ArgumentParser()
    parser.add_argument('frames', type=int, nargs='+')
    parser.add_argument('--style', '-s')
    args = parser.parse_args()

    main(args.frames, args.style)
```

L8 Default parameters used in chapter 5

Default parameters listed in `toml` syntax. Note that most simulations have their parameters stored in `json` files (`"parameters.json"`), but I changed the default to `toml` files (`"parameters.toml"`) due to better human-readability.

```
[parameters]
box_size = [ 60.0, 60.0, 60.0,]
k_bend = 26.0
k_stretch = 20.0
n_beads_max = 25
diffusion_const = 1.0
min_network_distance = 6
rate_motor_step = 0.0
rate_motor_bind = 0.0
rate_motor_unbind = 0.0
reaction_radius_motor_binding = 1.05
rate_attach = 0.0
rate_detach = 0.0
k_repulsion = 80.0

[run-parameters]
n_steps = 10000000
dt = 0.001
observation_interval = 500
```

L9 MR Example: Parameters

```
{
    "box_size": [60.0, 60.0, 60.0],
    "rate_motor_step": 3.0,
    "rate_motor_bind": 2.0,
    "rate_motor_unbind": 1.0,
    "min_network_distance": 6,
    "reaction_radius_motor_binding": 1.05,
    "k_bend": 26.0,
    "k_stretch": 20.0,
    "microrheology_bead": {
        "radius": 7.41,
        "k_repulsion": 800.0
    },
    "dt": 0.001,
    "n_steps": 100,
    "observation_interval": 5,
    "n_max_motors": 1000,
    "n_beads_max": 20000
}
```

L10 MR Example: Create Networks

```
"""
Creates a few networks and writes all coordinates and links (forming
filaments) into folder `initial_states`.
After this, you should run the script `run_equilibration.py`, and
finally `run_simulations.py`.

2020-06-17 -- Ilyas Kuhlemann
"""

from typing import Dict, Any
import argparse
import os
import time
import json
from multiprocessing.pool import Pool

import numpy as np
import pandas as pd

from bead_state_model.network_assembly.create_network import\
    AcceptanceRateHandlerBeadInNetwork
from bead_state_model.network_assembly.create_network_simple import\
    CreateNetworkSimple
```

```
# read some parameter from default config
with open('default_params.json', 'rt') as fp:
    config = json.load(fp)

# size of simulation BOX
# (with periodic boundary conditions)
box = np.array(config['box_size'])
# Parameters for filament stiffness
k_bend = config['k_bend']
k_stretch = config['k_stretch']

# default = 7.41 = 4 microns / 0.54 microns
# (see documentation on tuning the length scale):
radius_bead = config['microrheology_bead']['radius']
k_repulsion = config['microrheology_bead']['k_repulsion']


def create_network(folder: str) -> Dict[str, Any]:
    """
    Create a network with global parameters defined in this file.
    Write final state to specified folder.

    :return: Dictionary with information on created network,
             e.g. number of filaments, beads per filament, etc.
    """
    np.random.seed(np.uint32(hash(folder + str(time.time()))))

    system = CreateNetworkSimple(
        box=box,
        k_bend=k_bend,
        k_stretch=k_stretch,
        n_filaments=700,
        n_beads_per_filament=25
    )

    arh = AcceptanceRateHandlerBeadInNetwork(
        n_batch=30,
        radius=radius_bead,
        k_repulsion=k_repulsion,
        position=box/2
    )
    system.acceptance_rate_handler = arh

    system.run(verbose=False)
    system.write(folder)
    print('created \'{}\''.format(folder))
    return system.get_info()
```

```
def load_index() -> pd.DataFrame:
    """
    Load index if it exists, prepare an empty data frame if
    it does not.
    """
    fname = 'initial_states/index.csv'
    if os.path.exists(fname):
        return pd.read_csv(fname)
    return pd.DataFrame(columns=['index',
                                 'n_filaments',
                                 'n_beads_max',
                                 'n_beads_mean',
                                 'n_beads_std',
                                 'n_beads_total'])


if __name__ == "__main__":

    parser = argparse.ArgumentParser()
    parser.add_argument(
        'number_of_processes', type=int,
        help=("Number of processes to run in parallel, "
              "adjust this to match the number of threads "
              "you want to use.")
    )
    args = parser.parse_args()

    pool = Pool(args.number_of_processes)

    indices = list(range(6))
    folders = ['initial_states/network{:03}'.format(i)
               for i in indices]
    os.makedirs('initial_states', exist_ok=True)

    # run create network algorithm with n_proc processes in parallel
    results = pool.map(create_network, folders)

    network_index = load_index()
    # add created networks to index
    for idx, res in zip(indices, results):
        network_index.loc[len(network_index)] = [
            idx,
            res['n_filaments'],
            res['n_beads_max'],
            res['n_beads_mean'],
            res['n_beads_std'],
            res['n_beads_total']
        ]
    network_index.to_csv('initial_states/index.csv', index=None)
```

L11 MR Example: Run Equilibrations

```
import argparse
import json
import os
import time
from multiprocessing import Pool
from typing import Dict, Any, Sequence, List

import numpy as np
import pandas as pd
from readdy import ReactionDiffusionSystem

from bead_state_model import Simulation, BaseSetupHandler

_ParamSet = Dict[str, Any]


def main(n_processes: int):

    network_index = pd.read_csv('initial_states/index.csv')
    network_index = np.array(network_index['index']).astype(int)

    parameter_sets = _generate_parameter_sets(network_index)

    pool = Pool(n_processes)
    results = pool.map(run_equilibration, parameter_sets)

    r = np.array(results)
    # store some information on run time of each simulation
    np.savetxt('duration_of_equilibrations.txt', r, header='# run_time')


def run_equilibration(params: _ParamSet) -> float:

    params = params.copy()

    idx = params.pop('index')
    idx_network = idx

    print('running equilibration {:03}'.format(idx))

    t_start = time.time()

    output_folder = params.pop('out_dir')
    obs_interval = params.pop('observation_interval')
    n_steps = params.pop('n_steps')
    dt = params.pop('dt')
    box = np.array(params['box_size'])
    mr_bead_parameters = params.pop('microrheology_bead')
```

```
    folder_network = 'initial_states/network{:03}'.format(idx_network)
    file_beads = os.path.join(folder_network, 'beads.txt')
    file_config = os.path.join(folder_network, 'config.json')

    with open(file_config, 'rt') as fp:
        config = json.load(fp)

    bead_position = np.array(config['acceptance_rate_handler']
                             ['bead_position']) - box / 2
    bead_diffusion_const = 0.5 / mr_bead_parameters['radius']

    mr_bead_setup_handler = MRBeadSetupHandler(
        mr_bead_parameters['radius'],
        mr_bead_parameters['k_repulsion']
    )

    s = Simulation(
        output_folder=output_folder,
        parameters=params,
        non_filament_particles={'bead': bead_diffusion_const},
        interaction_setup_handler=mr_bead_setup_handler
    )

    s.add_non_filament_particles(bead=bead_position[np.newaxis, :])
    s.add_filaments(file_beads)

    s.run(n_steps, dt, obs_interval, mute=True)

    t_end = time.time()
    t = t_end - t_start
    print('done with equilibration {}'.format(idx))

    return t


class MRBeadSetupHandler(BaseSetupHandler):

    def __init__(self, radius: float, k_repulsion: float):
        self._radius = radius
        self._k_repulsion = k_repulsion

    def __call__(self, system: ReactionDiffusionSystem):
        for other in ['core', 'head', 'tail', 'motor']:
            system.potentials.add_harmonic_repulsion(
                "bead", other,
                force_constant=self._k_repulsion,
                interaction_distance=.5 + self._radius
            )
```

```
    @staticmethod
    def from_config_dict(d: Dict[str, Any]) -> 'MRBeadSetupHandler':
        return MRBeadSetupHandler(**d)

    def to_config_dict(self) -> Dict[str, Any]:
        return {'radius': self._radius, 'k_repulsion': self._k_repulsion}


def _generate_parameter_sets(
        network_index: Sequence[int]
) -> List[_ParamSet]:

    out_dir = 'initial_states_equilibrated'
    os.makedirs(out_dir, exist_ok=True)
    with open('default_params.json', 'rt') as fh:
        default_config = json.load(fh)

    parameter_sets = []
    for i in network_index:
        # default parameter are read from config file
        params = default_config.copy()

        # add parameters for run_equilibration function
        params['index'] = int(i)
        params['out_dir'] = os.path.join(out_dir, 'network{:03}'.format(i))

        # most important parameters to overwrite for equilibration:
        # number of steps (to have comparably short simulations),
        # time step (to safely remove overlaps of particles)
        params['n_steps'] = int(6e3)
        params['observation_interval'] = int(2e3)
        params['dt'] = 1e-6

        # don't allow motor binding in equilibration
        params['rate_motor_bind'] = 0.0

        parameter_sets.append(params)

    return parameter_sets




if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('number_of_processes', type=int)
    args = parser.parse_args()

    main(args.number_of_processes)
```

L12 MR Example: Run Simulations

```
import argparse
import json
import os
import time
from multiprocessing import Pool
from typing import Dict, Sequence, List, Any

import numpy as np
import pandas as pd

from bead_state_model import Simulation

# Parameters to be varied in the simulations. The default
# has 4 values for rate_motor_bind, and 6 different initial_states,
# leading to 4*6 = 24 simulations.
from bead_state_model.data_reader import DataReader

from run_equilibrations import MRBeadSetupHandler

varied_parameters = {
    'rate_motor_bind': [0.0, 0.1, 0.25, 0.5],
    'initial_state_index': np.arange(6)
}

# The index offset. Adjust this, if you already ran this script, and
# later want to run some more (e.g. a repetition with the same
# parameters, because you need to aggregate more data, or a set of
# different variations of rate_motor_bind/initial_state_index.
offset = 0

n_repetitions = 2

ParamSet = Dict[str, Any]


def main(n_processes: int):
    """
    This function gets executed when you run the script
    with `python run_simulation.py`. The call
    of the function happens at the very end of this script.
    """

    parameter_table = _create_parameter_table(
        varied_parameters, 2, offset
    )

    if not os.path.exists('simulations'):
        os.mkdir('simulations')
```

```
    fname_sim_index = 'simulations/simulation_index.csv'
    _write_parameter_table(parameter_table, fname_sim_index)
    parameter_sets = _generate_parameter_sets(parameter_table)

    pool = Pool(n_processes)
    results = pool.map(run_simulation, parameter_sets)

    r = np.array(results)
    # store some information on run time of each simulation
    np.savetxt('duration_of_simulations.txt', r, header='# run_time')


def run_simulation(parameters: ParamSet) -> float:
    # copy and retrieve via pop to trim the parameters
    # down to valid parameters for the Simulation class
    parameters = parameters.copy()
    idx = parameters.pop('index')
    print('running simulation {:03}'.format(idx))
    t_start = time.time()

    output_folder = parameters.pop('out_dir')
    dt = parameters.pop('dt')
    n_steps = parameters.pop('n_steps')
    obs_interval = parameters.pop('observation_interval')
    network_index = parameters.pop('initial_state_index')
    mr_bead_parameters = parameters.pop('microrheology_bead')
    # get diffusion constant by calculating the radius relative
    # to polymeric bead radius (set to 0.5)
    mr_bead_diffusion_const = 0.5/mr_bead_parameters['radius']

    mr_bead_setup_handler = MRBeadSetupHandler(
        radius=mr_bead_parameters['radius'],
        k_repulsion=mr_bead_parameters['k_repulsion']
    )

    s = Simulation(
        output_folder=output_folder,
        parameters=parameters,
        non_filament_particles={'bead': mr_bead_diffusion_const},
        interaction_setup_handler=mr_bead_setup_handler
    )

    init_state_file = os.path.join(
        'initial_states_equilibrated',
        'network{:03}'.format(network_index),
        'data.h5'
    )

    mr_bead_coordinates = _get_mr_bead_coordinates(init_state_file)
    s.add_non_filament_particles(bead=mr_bead_coordinates)
```

```
    s.add_filaments(init_state_file)

    s.run(n_steps, dt, obs_interval, mute=True)

    t_end = time.time()
    t = t_end - t_start

    print('done with simulation {:03}'.format(idx))

    return t


def _get_mr_bead_coordinates(init_state_file) -> np.ndarray:
    dr = DataReader(init_state_file)
    positions_final_frame = dr.read_particle_positions(
        minimum_image=True
    )[-1]
    # there is only a single mr bead, it's the first particle in the array:
    coords = positions_final_frame[0]
    return coords[np.newaxis, :]


def _create_parameter_table(
        params_dict: Dict[str, Sequence[float]],
        multiplier: int, idx_offset: int = 0
) -> pd.DataFrame:
    """
    Create parameter sets. Iterate over all combinations defined in
    ``param_dict``, and store combinations in a DataFrame.

    :param params_dict: Defines which values should be varied for the
                        simulations, all combinations will be created
                        and stored in a DataFrame.
    :param multiplier: Defines number of repetitions for each unique
                       parameter combination.
    :param idx_offset: Offset of indices for DataFrame (use this if
                       your index already contains parameters from
                       previous runs).

    :return: DataFrame with sequential integer index (starting at
             idx_offset, ending at
             idx_offset + n_combinations * multiplier), and one
             column per varied parameter.
    """
    labels = []
    params = []
    for lbl in sorted(params_dict.keys()):
        labels.append(lbl)
        params.append(params_dict[lbl])
```

```
    n_param_sets = multiplier
    for p in params:
        n_param_sets = n_param_sets * len(p)

    table = pd.DataFrame(columns=labels)

    p_set = [[p[0] for p in params]]
    filled_sets = pd.DataFrame(p_set * multiplier, columns=labels)
    table = table.append(filled_sets)
    for j, p in enumerate(params):
        lbl = labels[j]
        for k, pp in enumerate(p[1:]):
            param_sets_k = filled_sets.copy()
            param_sets_k[lbl] = pp
            table = table.append(param_sets_k)
        filled_sets = table.copy()

    table.index = np.arange(len(table)) + idx_offset
    return table


def _write_parameter_table(table: pd.DataFrame, fname: str):
    if os.path.exists(fname):
        # Add parameter sets to existing index e.g. from previous run
        # of this script. Adjust offset accordingly! Index needs to have
        # the exact same columns defined, i.e. the varied parameters have
        # to identical as in any previous runs.
        old_param_sets = pd.read_csv(fname, index_col=0)
        combined_param_sets = pd.concat([old_param_sets, table])
        if len(combined_param_sets.index.unique()) \
           != len(combined_param_sets.index):
            msg = "tried to concatenate old and new index tables, but "
            msg += "some indices appear in both tables"
            raise RuntimeError(msg)
        combined_param_sets.to_csv(fname)
        return
    # Save index.
    table.to_csv(fname)


def _generate_parameter_sets(table: pd.DataFrame) -> List[ParamSet]:
    with open('default_params.json', 'rt') as fh:
        default_config = json.load(fh)
    param_list = []
    # for i in range(offset, offset+len(param_sets)):
    for i in table.index:
        # load default parameters
        params = default_config.copy()
        # get current parameter set as dictionary
```

```
        d = table.loc[i].to_dict()
        # add entry required for ``run_sim`` function
        d['index'] = i
        params['out_dir'] = os.path.join('simulations',
                                         'out{:04}'.format(i))
        # update copy of default params with varied params
        params.update(d)
        param_list.append(params)
    return param_list


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('number_of_processes', type=int)
    args = parser.parse_args()

    main(args.number_of_processes)
```

Acknowledgements

Ich möchte an dieser Stelle noch einigen Unterstützerinnen und Unterstützern danken.

Zunächst ein mal Burkhard. Vielen Dank, dass du jemandem, der ohne Ahnung von Biologie oder statistischer Physik bei dir aufgetaucht ist, die Gelegenheit gegeben hast, in diesem spannenden Gebiet zu arbeiten.
Ich möchte mich bei meinem Thesis Advisory Committee, Burkhard, Bert und Sarah, herzlich für die Unterstützung bedanken. In unseren Treffen gab' es dank euch viele spannende Diskussionen, aus denen ich viel Hilfreiches ziehen konnte.

Danke an Moritz Hoffmann und Christoph Fröhner für eure Unterstützung bei meinen Beiträgen zu ReaDDy. Ihr habt da etwas echt schönes geschaffen.

Viel Motivation für meine Arbeit habe ich aus den Bewegungen für freie und für open-source Software gezogen. Vielen Dank an alle, die so viel Ihrer Zeit und Energie in diese Arbeit stecken, und sie dann allen zur Verfügung stellen. Ohne eure Arbeit wäre ich keinen Zentimeter weit gekommen.

Vielen Dank an den ganzen AK. Mit euch lieben Menschen und unseren Serienabenden, Filmabenden, Zockabenden, Drohnenpausen, Abholungen, Mariokartturnieren, Dartturnieren, MTG Boosterdrafts, Lauftrainings und Laufwettkämpfen (für Senioren) ließ es sich doch ganz gut aushalten hier.

Besonders möchte ich Hannes und Jana danken. Hannes, für dein Interesse an dem Projekt und dem ganzen hilfreichen Feedback. Aber deine Unterstützung ging weit über die Arbeit hinaus. Dadurch, dass du mich zum AK dazugeholt hast, hatten wir noch mal ein paar Jahre mehr, um gemeinsam in Göttingen rumzuhängen. Danke an euch beide für die schöne Zeit, ohne euch wär's voll doof gewesen.

Der größte Danke gilt meiner Familie. Danke für die Geduld und eure Unsterstützung bei jedem Schritt des Weges. Ich freue mich schon so, so sehr darauf, wieder mehr Zeit mit euch zu verbringen.