Diploma thesis

on

Computer simulations of macromolecular systems with external constraints

submitted by

Franziska Müller

April 16th, 2014

Faculty of physics
Johannes Gutenberg University Mainz

Affidavit

Mainz, April 16th, 2014

I, Franziska Müller, student of physics at Johannes Gutenberg University Mainz, hereby confirm that this thesis is the result of my own work. I did not use any sources other than the ones specified. Furthermore, I confirm that this thesis has not yet been submitted as part of another examination process neither in identical nor in similar form.

Franziska Müller

Contents

1 Introduction

Polymers are widely used in the materials, chemical and food industry. They often serve as functional ingredients, which emphasis or even evoke desired features of the material. Therefore a detailed understanding of polymeric behavior is of interest. To examine specific properties of polymers, their molten state is most suitable. Although a broad knowledge about influencing and processing polymeric materials with desired behavior already exists, the connection between microscopic and macroscopic properties is still lacking in many cases. Computer simulations provide a great tool in bridging this gap. Especially, because the very same polymeric system, in computer experiments, is available to as many investigations as desired. However, in real experiments the configuration of a polymeric system can be destroyed during a measurement or at least changes in time. The properties of polymers are particularly interesting in *entangled* systems. Entanglements are topological constraints that occur in melts of long polymer chains due to the property that two chains cannot pass through each other. Entanglements were first discussed in 1940 ([Bus31], [Tre40]) and since then different theories predicting their effects have been developed. One category of these models are *tube models*. They subsume the effects of surrounding chains on a certain chain and describe them as a tube which confines the motion of that chain. With equilibrated systems of long polymer chains one could directly verify these tube models and draw the connection between microscopic theories and macroscopic behavior for entangled polymers. Present simulations mostly reproduce experiments only qualitatively, mainly because they are conducted with very small systems. Although reaching the macroscopic range of 10^{23} particles is a distant objective, advancing to bigger systems can provide quantitatively worthy results in the future. The major obstacle in computer

simulations with polymers is their equilibration. This is the procedure that transfers the system to a thermodynamic stable state, the *equilibrium*, which real systems adopt automatically after a long enough time. The initial setup of a simulated polymeric system often is so far away from equilibrium, that starting a simulation would fail. This will be explained in chapter 4. In addition computational time ranges are very small compared to real time and a straight equilibration would take too long, even if it was possible. Moreover, time scales for polymer motion are directly related to the lengths scale under ivestigation. The movement of a monomer influences its bonded neighbours and the motion of larger segments of a polymer prevails longer than the one of shorter segments. The time corresponding to the motion of the whole polymer is called *relaxation time*. For entangled polymer chains, this relaxation time rises rapidly with chain length. For example: the relaxation time increases by a factor of ten, when the chain length doubles.

For all the above reasons a powerful simulation software is needed along with an equilibration technique and criteria that indicate that the equilibrium is reached. This thesis exploits configurations obtained by simulations with ESPResSo++, a parallel software that uses molecular dynamics as a simulation method. Two quantities that provide criteria for the equilibrium are the static structure factor and the mean squared displacement of monomers. They were implemented within ESPResSo++ as a part of this thesis. Theoretical background on these two quantities can be found in chapter 2. Chapter 3 explains the parts of ESPResSo++ important to this work including the code for the two new analysis tools. The equilibration technique that was applied to relax even long chains is described in chapter 4 along with information on the model, its potentials and details about the chains. Performance and computation results of the structure factor and the monomer displacement are provided in chapter 5 and chapter 6 contains suggestions for further improvements of the computation.

2 Theory

This chapter explains the theoretical background to the two quantities examined in this thesis. First, theory on the static structure factor is provided, both for the collective and the single-chain structure factor. Their relation to other quantities, namely the form factor and the compressibility is given. Furthermore a peculiar feature of the structure factor for simulated systems is explained. Second, the definition and meaning of monomer displacement in polymer physics is explained. The three forms of appearance are given along with their meaning in systems of entangled linear polymer melts.

2.1. Static structure factor

The static structure factor $S(q)$ describes how a material scatters incident radiation. Experimentally, it is determined by elastic scattering. In the case of polymers X-Rays or neutrons are used as projectiles. The intensity of scattered radiation measured in elastic scattering experiments is determined essentially by three factors as equation 2.1 shows. $C(q)$ combines factors due to the detector, such as the detector efficiency and its solid angle, $f(\sigma)$ contains information about the interaction between projectiles and target and $S(q)$ about the positions of the scattering centers. ([Hig94], 9)

$$I(q) = C(q)f(\sigma)S(q). \tag{2.1}$$

The variable q is related to the scattering angle and the difference in wavelength between the incident and the scattered wave and will be defined in the next section. $S(q)$ is called the *static structure factor* or *scattering function*.

2.1.1. Collective structure factor

This section uses the following notations for incident (subscript i) and scattered(subscript f for final) wave vectors (as in [Hig94], 12):

$$\mathbf{k_i} = \frac{2\pi}{\lambda} \cdot \hat{\mathbf{k}}_\mathbf{i} \tag{2.2}$$

$$\mathbf{k_f} = \frac{2\pi}{\lambda} \cdot \hat{\mathbf{k}}_\mathbf{f}, \tag{2.3}$$

where $\hat{\mathbf{k}}_\mathbf{i}$ and $\hat{\mathbf{k}}_\mathbf{f}$ are the directions of travel and λ is the wavelength, which, in elastic scattering, is the same for the incident and the scattered wave. The *scattering vector* $\mathbf{q}$ is defined as their difference.

$$\mathbf{q} = \mathbf{k_f} - \mathbf{k_i} = \frac{1}{\hbar} m(\mathbf{v_f} - \mathbf{v_i}) \tag{2.4}$$

It is also referred to as *momentum transfer* as can be seen from the right equality in 2.4, where the de Broglie wavelength enters and m and $\mathbf{v}$ denote the mass and velocities of the projectile.

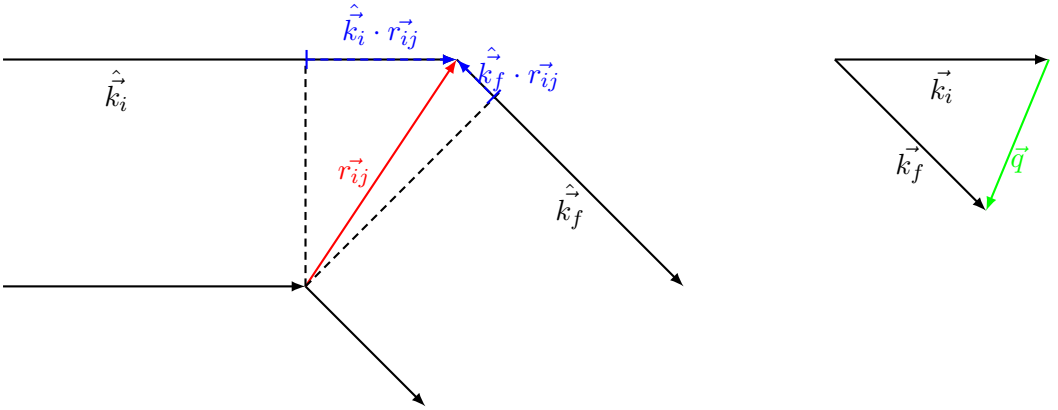

Figure 2.1.: Left side: elastic scattering from two point scatterers with relative vector $\mathbf{r_{ij}}$. The travel distance difference of the scattered waves (blue vectors) can be calculated as vertical projections of $\mathbf{r_{ij}}$ to the wave vectors $\hat{\mathbf{k}}_\mathbf{i}$ and $\hat{\mathbf{k}}_\mathbf{f}$. Right side: geometrical representation of the scattering vector $\mathbf{q}$.

The phase difference caused by scattering on different scattering centers is obtained by $\mathbf{q} \cdot \mathbf{r}_{ij}$ as figure 2.1 depicts (cf. [Hig94], 12).

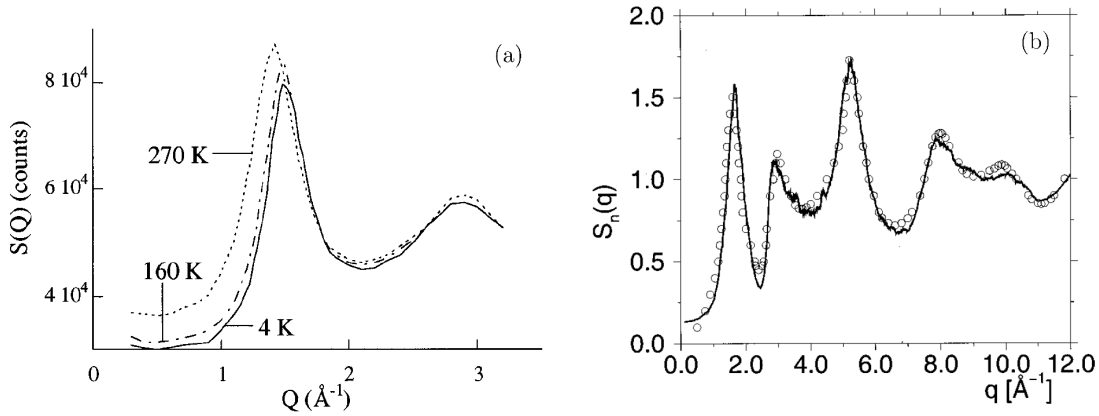

Figure 2.2.: (a) Static structure factor of polybutadiene melts (T = 270 K) and glasses (T = 4 K, 160 K) measured by neutron scattering due to Arbe *et al.*([Arb96]). The scattering background is not subtracted here, therefore the zero of the Y axis is not known precisely. (b) Static structure factor of silicon dioxide at $T = 300$ K. Circles denote data points obtained by neutron scattering from Price and Carpenter ([Pri87]). Data for the solid line was obtained by molecular dynamics simulations of Horbach and Kob ([Hor99]). The graph is taken from [Hor99].

The static structure factor is defined as (cf. [Rub03], 123)

$$S(\mathbf{q}) = \frac{1}{N \cdot M} \left[\sum_{i=1}^{N \cdot M} \sum_{j=1}^{N \cdot M} \left\langle \mathrm{e}^{-i\mathbf{q}\cdot(\mathbf{r}_i - \mathbf{r}_j)} \right\rangle \right], \tag{2.5}$$

where $\mathbf{r}_i$ and $\mathbf{r}_j$ are the positions of scatterers and $N \cdot M$ their total number. The angled brackets denote the average over different configurations. For ergodic systems it does not matter wether they are time or ensemble averages. In this thesis the structure factor is computed for polymeric systems. Hence, $N \cdot M$ is the total number of monomers in the system. The degree of polymerization N of the polymer describes the number of monomers that belong to one molecule. The number of molecules is represented by M. Since the computation is performed for a computer simulated system in this thesis "monomers" refers to the monomers of the particular model system. In experiments the total number of scatterers denotes the nuclei in the scattering volume only ([Rub03], 123) and the scattering length of the atoms has to be taken into account as well ([Cat00], 177).

Figure 2.2(a) shows the typical form of the static structure factor for a dense fluid[1].

[1]Polymer melts are dense fluids. Since the *coherent* collective static structure factor does not distinguish between monomers of different chains, it exhibits the same form as for "any other dense fluid"([Bin05], 94). Scattering is called "coherent", when scattering centers are identical. Experimentally this is conducted by deuteration of all chains.([Hig94])

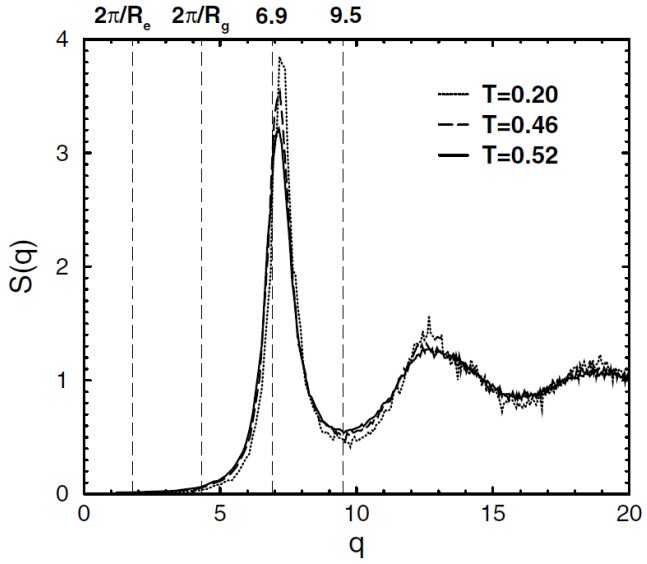

Figure 2.3.: Static structure factor for a bead-spring model of polymers obtained from simulations of Baschnagel *et al.* ([Bas00]). Beads interact via a Lennard Jones-potential. Temperature and lengths are given in units of the Lennard-Jones parameters ε and σ , respectively (cf. equation 4.1).

It was obtained experimentally by neutron scattering. The first and highest peak at $q_1 \approx 1.5\text{Å}^{-1}$ corresponds to the distance r_{nn} of nearest neighbouring atoms, which is given by $r_{nn} = \frac{2\pi}{q_1}$. The second, smaller peak at about $q_2 \approx 2.8\text{Å}^{-1}$ corresponds to intramolecular correlations along the chain. Figure 2.2(b) shows the structure factor for a silicon dioxide (SiO), obtained from both experiment and molecular dynamics simulation. It is shown as an example for a substance, where the first peak of the structure factor does not correspond to the distance of nearest neighbouring atoms. In the silicon dioxide network the nearest neighbour of a silicon atom is always an oxygen atom (see figure 2.4). This Si-O distance occurs in the further peaks at higher values

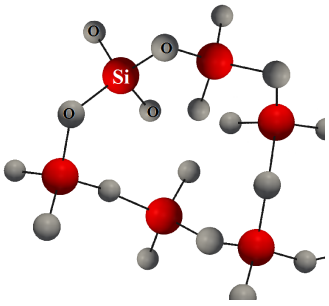

Figure 2.4.: Silicon dioxide network. From [Fil12]

of q. In this case the first peak at $q_1 \approx 1.6\text{Å}^{-1}$ corresponds to the distance between neighbouring silicon atoms. ([Bin05], 42 *ff.*)

The structure factor of a bead-sping model of a polymer is shown in figure 2.3. It also has the typical form of a dense fluid. A combination of Lennard-Jones and FENE potential were used to model the system. The minimum of the total potential is located at 0.96σ. As a result the favoured bondlength is 0.96σ, which does not match the first peak of the structure factor plot. The position of the peak is around $q_1 \approx 7$, which can be interpreted as $r_{nn} = \frac{2\pi}{7} \approx 0.90$. This mismatch conflicts crystalline ordering and indicates that the system is amorphous ([Bas00], 6366).

2.1.2. Single-chain and intramolecular structure factor

The structure factor can be decomposed into inter- and intramolecular parts (as in [Hig94], 123 *f.*).

$$S(\mathbf{q}) = \frac{1}{NM} \left[\sum_{i=1}^{N\cdot M} \sum_{j=1}^{N\cdot M} \left\langle \mathrm{e}^{-i\mathbf{q}\cdot(\mathbf{r}_i - \mathbf{r}_j)} \right\rangle \right] \tag{2.6}$$

$$= \frac{1}{N} \left[\sum_{p=1}^{M} \sum_{q=1}^{M} \sum_{k=1}^{N} \sum_{l=1}^{N} \left\langle \mathrm{e}^{-i\mathbf{q}\cdot(\mathbf{r}_{p,k} - \mathbf{r}_{q,l})} \right\rangle \right] \tag{2.7}$$

In the above equation M is the number of chains, or generally the number of molecules and N is the number of particles per molecule or for the case of this thesis the number of beads per chain. Indices p and q denote chains, k and l beads of a chain. Separating terms belonging to one chain from those belonging to different chains yields

$$S(\mathbf{q}) = \frac{1}{N} \left[\sum_{p=1}^{M} \sum_{k=1}^{N} \sum_{l=1}^{N} \left\langle \mathrm{e}^{-i\mathbf{q}\cdot(\mathbf{r}_{p,k} - \mathbf{r}_{p,l})} \right\rangle + \sum_{p=1}^{M} \sum_{q\neq p}^{M} \sum_{k=1}^{N} \sum_{l=1}^{N} \left\langle \mathrm{e}^{-i\mathbf{q}\cdot(\mathbf{r}_{p,k} - \mathbf{r}_{q,l})} \right\rangle \right]. \tag{2.8}$$

The first term is called the single chain structure factor or the form factor $P(\mathbf{q})$ of the molecule and corresponds to intermolecular interferences. The second term corresponds to interferences between radiation emitted from different molecules and is named $Q(\mathbf{q})$ in [Hig94](124).

$$S(\mathbf{q}) = \frac{1}{N} \left[MN^2 P(\mathbf{q}) + M^2 N^2 Q(\mathbf{q}) \right] = NP(\mathbf{q}) + MNQ(\mathbf{q}) \tag{2.9}$$

$$P(\mathbf{q}) = \frac{1}{M} \frac{1}{N^2} \sum_{p=1}^{M} \sum_{k=1}^{N} \sum_{l=1}^{N} \left\langle \mathrm{e}^{-i\mathbf{q}\cdot(\mathbf{r}_{p,k} - \mathbf{r}_{p,l})} \right\rangle = S_{\text{single-chain}}(\mathbf{q}) \tag{2.10}$$

$$Q(\mathbf{q}) = \frac{1}{M^2}\frac{1}{N^2}\sum_{p=1}^{M}\sum_{q\neq p}^{M}\sum_{k=1}^{N}\sum_{l=1}^{N}\left\langle \mathrm{e}^{-i\mathbf{q}\cdot(\mathbf{r}_{p,k}-\mathbf{r}_{q,l})}\right\rangle \tag{2.11}$$

The double sum over chains in 2.11 produces $M \cdot (M-1)$ terms. Since the number of chains is usually very high, it is approximated by M^2 (both in equation 2.9 and 2.11). In computer simulations one might also investigate systems with a rather small number of chains. One system used for testing purposes within this thesis only contained ten chains, so it would be appropriate to use the exact formula. However, equation 2.11 is not used in any implementation, since only the collective and the single-chain structure factor were implemented.

2.1.3. Relation to other quantities

Form Factor

The form factor usually describes the shape of the target particles. It can be the shape of the nucleus or a molecule. In the case of polymers, a molecule's shape is composed of the positions (and orientations) of its monomers. Even more general, in polymer models it is composed of the positions of the units the polymer is divided into. This can be, depending on the model, beads representing monomers or blobs representing bigger segments of a polymer.

The form factor is defined as

$$P(\mathbf{q}) \equiv \frac{I_s(\mathbf{q})}{I_s(0)}, \tag{2.12}$$

where $I_s(0) := \lim_{|\mathbf{q}|\to 0} I_s(\mathbf{q})$. The form factor is measured from a polymer in dilute solution, because molecules are seperated here. Since only the form and no motion is of interest, only elastic scattering is taken into account. The scattered intensity for such an experiment calculates from the incident intensity I_i as

$$I_s(\mathbf{q}) = I_i A^2 \sum_{k=1}^{N}\sum_{l=1}^{N}\cos[\mathbf{q}\cdot(\mathbf{r}_k - \mathbf{r}_l)], \tag{2.13}$$

where A contains factors such as the polarizability of the target particle. Thus, the form factor for polymers in dilute solution computes as

$$P(\mathbf{q}) = \frac{1}{N^2}\sum_{k=1}^{N}\sum_{l=1}^{N}\cos[\mathbf{q}\cdot(\mathbf{r}_k - \mathbf{r}_l)]. \tag{2.14}$$

In fact this relates directly to the single-chain structure factor from equation 2.10. ([Rub03], 82)

Pair distribution function

The pair distribution function is defined as

$$g(\mathbf{r}) = \frac{1}{\rho} \sum_{j \neq i} \langle \delta(\mathbf{r} - \mathbf{r}_i + \mathbf{r}_j) \rangle. \tag{2.15}$$

For amorphous substances $g(\mathbf{r}) \equiv g(r)$ is called the *radial distribution function*. The relation of the static structure factor to the pair distribution function is given in equation 2.16.

$$S(\mathbf{q}) = 1 + \rho \int \mathrm{e}^{-i\mathbf{q}\cdot\mathbf{r}} g(\mathbf{r}) d\mathbf{r} \tag{2.16}$$

Conversely, the fourier transform of $[S(\mathbf{q}) - 1]$

$$g(\mathbf{r}) = \frac{1}{\rho} \frac{1}{(2\pi)^3} \int \mathrm{e}^{-i\mathbf{q}\cdot\mathbf{r}} \left[S(\mathbf{q}) - 1\right] d\mathbf{q} \tag{2.17}$$

provides the pair distribution function. ([Bin05], 37 *f.*)

Compressibility

In the limit of low wavenumbers, i.e. $\mathbf{q}$ approaches zero, the structure factor can be written as

$$\lim_{q \to 0} S(q) \equiv S(q \to 0) = 1 + \rho \int \left[g(\mathbf{r}) - 1\right] d\mathbf{r}. \tag{2.18}$$

Here the limit $g(\mathbf{r} \to \infty) = 1$ is substracted in the integrant. The integration would transform it to a delta-distribution at $q = 0$, which does not contribute to the limit $q \to 0$. Equation 2.18 relates to density fluctuations (equation 2.19, [Han86], 29 *f.*), which also relate to the compressibility (equation 2.20).

$$\frac{\langle N^2 \rangle - \langle N \rangle^2}{\langle N \rangle} = 1 + \rho \int \left[g(\mathbf{r}) - 1\right] d\mathbf{r} \tag{2.19}$$

$$\frac{\langle N^2 \rangle - \langle N \rangle^2}{\langle N \rangle} = \rho k_B T \kappa_T, \tag{2.20}$$

where ρ is the mass density, k_B is the Boltzmann constant, T the temperature and κ_T the isothermal compressibility. Thus, the static structure factor is related to the isothermal compressibility by eqation 2.21.

$$\lim_{q \to 0} S(q) = \rho k_B T \kappa_T, \tag{2.21}$$

This paragraph follows [Bin05](45).

2.1.4. Restrictive choice of scattering vectors q

All simulation configurations used in this project have been run with periodic boundary conditions. The periodicity in coordinates yields a restriction in the choice of scattering vectors **q** for the calculation of the structure factor. This can be understood from the following equations 2.29. Considering only one dimension for a start, the periodic boundary conditions require the same result for $(r_x + L_x)$ as for r_x, where L_x is the box length in x-direction:

$$\mathrm{e}^{-iq_x r_x} \stackrel{!}{=} \mathrm{e}^{-iq_x(r_x+L_x)} \tag{2.22}$$

$$1 \stackrel{!}{=} \mathrm{e}^{-iq_x L_x} \tag{2.23}$$

$$1 \stackrel{!}{=} \cos(q_x L_x) - i\sin(q_x L_x) \tag{2.24}$$

$$1 \stackrel{!}{=} \cos(q_x L_x) \quad \wedge \quad \sin(q_x L_x) \stackrel{!}{=} 0 \tag{2.25}$$

$$\Rightarrow q_x \stackrel{!}{=} \frac{2\pi}{L_x} \cdot n \qquad \text{for } n \in \mathbb{N}_0 \tag{2.26}$$

Since the periodicity condition must always be true, the same restriction applies to y- and z-direction, in particular if two components of the scattering vector are zero.

$$\mathrm{e}^{-i\mathbf{q}\cdot\mathbf{r}} = \mathrm{e}^{-iq_x r_x} \cdot \mathrm{e}^{-iq_y r_y} \cdot \mathrm{e}^{-iq_z r_z} \tag{2.27}$$

$$\text{for } q_y = q_z = 0 \tag{2.28}$$

$$\Rightarrow \mathrm{e}^{-iq_x r_x} \stackrel{!}{=} \mathrm{e}^{-iq_x(r_x+L_x)} \tag{2.29}$$

So it is only possible to choose scattering vectors, whose components q_i are multiples of $\frac{2\pi}{L_i}$. In other words, scattering vectors are lying on a grid with spacings of $\frac{2\pi}{L_i}$. This grid will be discussed again in chapter 6.

Furthermore a restriction for the largest sensible scattering vector can be obtained by the bond length. Since q space is proportional to reciprocal coordinate space, q vectors larger than $\frac{2\pi}{b}$ correspond to distances smaller than the bond lenght, a scale on which the probability of finding more than one particle vanishes.

2.2. Mean-square displacement

For melts of polymer chains commonly three types of time displacements are calculated: The displacments of monomers, the monomer displacement in the chain's center-of-mass frame and the displacement of chains. This classification was introduced by K. Kremer in [Kre83](1635).

2.2.1. Displacements g_1, g_2 and g_3

The mean-square monomer displacement is referred to as g_1 and calculates as

$$g_1(t) = \langle [\mathbf{r_i}(t) - \mathbf{r_i}(0)]^2 \rangle, \tag{2.30}$$

where $\mathbf{r}$ are the coordinates of a monomer relative to the total system's center of mass and $\langle\rangle$ is the average over all NM monomers. The monomer displacement with respect to its chain's center of mass $\mathbf{r_{CM}}$ is

$$g_2(t) = \langle [\mathbf{r_i}(t) - \mathbf{r_i}(0) - \mathbf{r_{CM}}(t) + \mathbf{r_{CM}}(0)]^2 \rangle \tag{2.31}$$

and the mean-square displacement of the chain's center of mass, g_3, is defined as

$$g_3(t) = \langle [\mathbf{r_{CM}}(t) - \mathbf{r_{CM}}(0)]^2 \rangle, \tag{2.32}$$

where $\mathbf{r_{CM}}$ is the center of mass of the chain relative to the system's center of mass. Consequently, the angled brackets in g_3 indicate an average only over the M chains of the system.(cf. [Bul08], 17)

Subtracting the center of mass of the system excludes drift from the above displacements. For g_2 this is not necessary explicitly because it is included in the subtraction of the chain center of mass (if the system drifts, whole chains drift).

2.2.2. Monomer displacement in entangled linear melts

The motion of monomers in entangled systems is restricted by both the bonds to the other monomers of the molecule and by neighbouring molecules. In entangled polymer chains the restrictions arising from neighbour molecules are described by tube models. A single chain is confined by a tube of diameter a, i. e. its monomers' motion perpendicular to the tube's axis is restricted by the tube diameter and motion parallel to the tubes axis is not limited by surrounding chains. The different effects on monomer motion become notable only on the corresponding length- or timescales, respectively. Figure 2.5 represents the mean square monomer displacement for entangled chains over different time regimes. Between the relaxation time of a Kuhn monomer τ_0 and the entanglement time τ_e the motion of a monomer is mainly restricted by the bonded neighbours, because this interval is too short for movements of the order of the tube diameter. It was long assumed that hydrodynamic interactions are screened beyond the monomor length in melts[2]. Then the motion can be described by the subdiffusive part of the Rouse model, which is given by equation 2.33.

$$g_1(t) = \langle [\mathbf{r}(t) - \mathbf{r}(0)]^2 \rangle \propto b^2 \left(\frac{t}{\tau_0}\right)^{\frac{1}{2}} \qquad \text{for } t < \tau_e \tag{2.33}$$

[2]Farago et al. have shown in [Far11] that this is not valid. Viscoelastic hydrodynamic interactions contribute to the dynamics for short time scales. In this range the motion has to be corrected. The formulas in this section and figure 2.5 still follow the old assumtion.

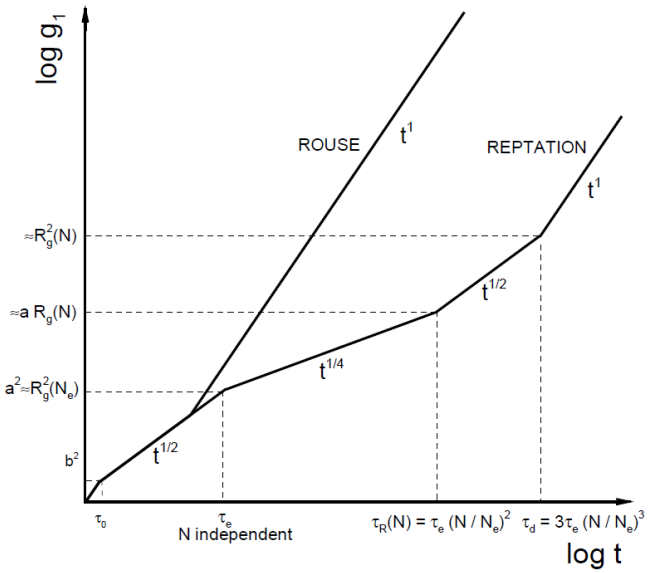

Figure 2.5.: Monomer displacement of entangled linear melts ("reptation") compared to unentangled displacement ("Rouse")([Bul08], 19)

At times greater than the entanglement time effects of the tube become notable. The motion described above only occurs along the tube's axis. Hence, equation 2.33 now applies to the curvilinear coordinates $s(t)$ as long as the relaxation time of the chain is not exceeded.

$$\langle [s(t) - s(0)]^2 \rangle \propto b^2 \left(\frac{t}{\tau_0} \right)^{\frac{1}{2}} \propto a^2 \left(\frac{t}{\tau_e} \right)^{\frac{1}{2}} \qquad \text{for } \tau_e < t < \tau_R$$

The second proportionality is obtained by switching from the description of Kuhn monomers to the one of entanglement segments. As well as the chain can be described as a random walk of step length b, the tube can be described by a random walk with step length a [3]. Whereas each monomer is in coherent motion with $\sqrt{t/t_0}$ neighbouring monomers, each segment is in coherent motion with $\sqrt{t/t_e}$ neighbouring segments. The transformation back to canonical coordinates follows [4]

$$\langle \Delta \mathbf{r}^2 \rangle \propto a \sqrt{\langle \Delta s^2 \rangle}$$

[3] or rather of the order of the tube diameter a. This is a reasonable assumption, since we are considering scaling laws (rather than absolute dependencies)

[4] This transformation can be understood from the relation between the contour length and the end-to-end distance of a freely-jointed chain with fixed bond length (which describes a random walk). The two quantities are connected by $\langle R^2 \rangle \propto bR_{max}$, because $Nb^2 = b(Nb)$. Curvilinear coordinates are defined along the contour and the end-to-end distance refers to a distance in canonical space. So $\Delta s(t)$ scales with $\Delta \mathbf{r}^2$ in the same way that R_{max} scales with $\langle R^2 \rangle$. The prefactor for the former is the tube's random walk step length a, since it is the chain's random walk step length b for the latter.

yielding a monomer displacement in space of

$$g_1(t) = \langle [\mathbf{r}(t) - \mathbf{r}(0)]^2 \rangle \propto a\sqrt{\langle [s(t) - s(0)]^2 \rangle} \propto a^2 \left(\frac{t}{\tau_e}\right)^{\frac{1}{4}} \qquad \text{for } \tau_e < t < \tau_R. \quad (2.34)$$

At times larger than the chain's relaxation time τ_R the effects of the bonded neighbours can be disregarded, leaving the restrictions of motion to the tube. So displacements of monomers is mainly determined by the center of mass of the chain, which moves in diffusive Rouse motion along the tube. In curvilinear coordinates this purely diffusive motion can be described as

$$\langle [s(t) - s(0)]^2 \rangle \propto D_c t \propto b^2 N \frac{t}{\tau_R} \propto a^2 \frac{N}{N_e} \frac{t}{\tau_R} \qquad \text{for } \tau_R < t < \tau_r ep$$

leading to a displacement in space of

$$g_1(t) = \langle [\mathbf{r}(t) - \mathbf{r}(0)]^2 \rangle \propto a\sqrt{\langle [s(t) - s(0)]^2 \rangle} \propto a^2 \left(\frac{N}{N_e}\right)^{\frac{1}{2}} \left(\frac{t}{\tau_R}\right)^{\frac{1}{2}} \qquad \text{for } \tau_R < t < \tau_{rep}. \quad (2.35)$$

In the above relation τ_{rep} is the reptation time which describes the time the chain needs to diffuse out of the tube. Or, more precisely, the time that corresponds to a motion of order of the tube length $\frac{aN}{N_e}$. For times larger than the reptation time the tube restrictions can be neglected and the monomers follow the chain's diffusive motion in space. The Rouse model provides a diffusion coefficient of

$$D_{rep} \propto \frac{R^2}{\tau_{rep}} \propto \frac{kT}{\rho} \frac{N_e}{N^2},$$

so the mean square monomer displacement is proportional to t on this large timescale:

$$g_1(t) = \langle [\mathbf{r}(t) - \mathbf{r}(0)]^2 \rangle \propto D_{rep} t \propto \frac{kT}{\rho} \frac{N_e}{N^2} t \qquad \text{for } t > \tau_{rep} \quad (2.36)$$

3

Creation of new analysis tools

In this project computer simulated polymer melts are analyzed. These are linear melts up to a chainlength of 2000. Since long chains have large relaxation times, their equilibration takes both time and computational power. Therefore a parallel simulation software is used for the equilibration, namely ESPResSo++ ([Hal13]). Wether the systems reached equilibrium is indicated by various quantities, such as the mean-square internal distance, the mean-square monomer displacement and the static structure factor. The last two are implemented as analysis tools of ESPResSo++. In this chapter the software is described briefly (section 3.1). Subsequently the way it computes the structure factor and the monomer displacement is described (sections 3.2 and 3.3).

3.1. Simulation software ESPResSo++

The Extensible Simulation Package for Research on Soft matter systems (ESPResSo++, [Hal13]) is a free and open-source software. It is parallelized and object oriented and targeted for a broad range of computer architectures. ESPResSo++ is designed for many-particle systems of condensed soft matter and uses Molecular dynamics and Monte Carlo algorithms. Its high modular kernel is written in C++, whereas it has a Python user interface. This makes the software very flexible and enables it to deal with a wide range of systems. Since the main design objective is extensibility, it is easy to add new features to ESPResSo++ and therefore the software package is still growing. Figure 3.1 shows the basic work flow of ESPResSo++, including the connection between the

[1] Sources: homepage of ESPResSo++ ([Stü]) and Openclipart.org ([OCAc], [OCAb], [OCAa])

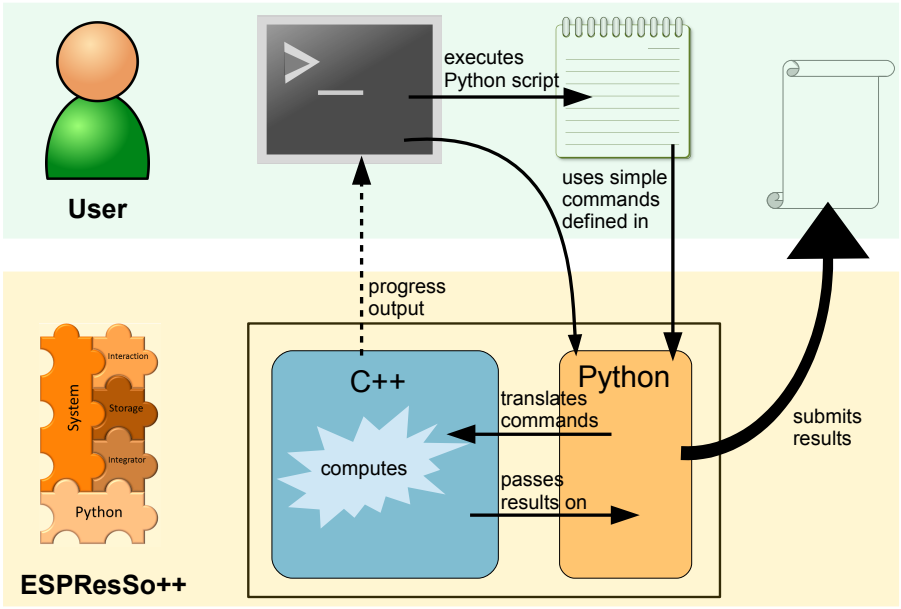

Figure 3.1.: Work flow of ESPResSO++ [1]

two programming languages. The user writes and runs a Python script, which contains the commands for building the desired system, starting a simulation and analyzing the results. These commands are defined on the Python level of ESPResSo++. The definition mainly consists in connecting the commands to the appropriate C++ code, which performs all computations. The simulation results are then passed to the Python level before they are submitted to the user.

Usage

A minimal example of a user's Python script is given below. It uses an analysis routine of ESPResSo++ added during this thesis: the static structure factor. Commented lines start with a hash tag (`#`) in Python. All ESPResSo++ scripts have to start with the `import` of `espresso` (line 2). The analyzed system can either be created with an ESPResSo++ simulation (within the same script)or read in from a file (if it was created and saved before). An object of the desired analysis has to be created (here `StatS`, line 8) before calling the corresponding function (here `StatS.compute()`). This is typical for all analysis with ESPResSo++.

```
1 #script for calculation of static structure factor
2 import espresso
```

```
4  #read in your configuration from a file(the system) ...
5  #or get it from a running simulation

7  #creating the StatS object
8  StatS = espresso.analysis.StaticStructF(system)

10 #compute the collective static structure factor
11 result_collective = StatS.compute(10,10,10,1, conf1)

13 print "collective static structure factor:", result_collective
```

Connection between Python and C++

They way ESPResSo++ connects the Python to the C++ level is now shown for analysis. The static structure factor computation of ESPResSo++ serves as a typical example.

For analysis, usually three internal files correspond to the computation of a quantity:

- a C++ header file: `StaticStructF.cpp`
- a C++ source file: `StaticStructF.hpp`
- a Python module: `StaticStructF.py`

The latter contains the connections between the user's commands (in the Python script) to the functions and classes implemented in C++. For example, in the script above in line 11 `compute()` is called. In the following, the connection to the C++ function that performs the main computation in this script is explained. The Python function `compute()` is defined in the corresponding Python module `StaticStructF.py` in lines 39 till 43:

```
34 class StaticStructFLocal(ObservableLocal, analysis_StaticStructF):
35   'The (local) compute the static structure function.'
36   #Python constructor...

39     def compute(self, nqx, nqy, nqz, bin_factor, ofile = None):
40     if ofile is None:
41       return self.cxxclass.compute(self, nqx, nqy, nqz, bin_factor)
42     else:
43     #same call PLUS creation of an output file...
```

In line 41 the first part of the connection to the C++ function is made. The second part can be found at the end of the source file `StaticStructF.cpp` in the lines listed below. In line 392 the function `compute()` is connected to the C++ function `computeArray()`. `computeArray()` contains the computation of the structure factor and is defined in the same file.

```
388         void StaticStructF::registerPython() {
389             using namespace espresso::python;
390             class_<StaticStructF, bases< Observable > >
```

```
391                     #registration of constructor...
392                     .def("compute", &StaticStructF::computeArray)
393                     #analog for single chain function...
394                     ;
395         }
```

The definition and connection of constructors is implemented analogously to the one of functions. A description of C++ functions computing analysis results is given in the next section.

3.2. Static structure factor

This section describes the implementation of the static structure as an analysis routine of ESPResSo++. The choice of the formula, or rather its analytic transformation is explained at first. Since ESPResSo++ is a parallel software, this routine is also parallelized and the way it is conducted is also stated in this section. Information about binning used in the implementation is provided along with the parameters of the main function of the routine (which are important to the user). The main loop of the code is given below, for full source codes see appendix A.1.1.

$$S(\mathbf{q}) = \frac{1}{N} \left[\sum_{i=1}^{N} \sum_{j=1}^{N} \mathrm{e}^{-i\mathbf{q}\cdot(\mathbf{r}_i - \mathbf{r}_j)} \right] \tag{3.1}$$

$$S(\mathbf{q}) = \frac{1}{N} \left[\left(\sum_{i=1}^{N} \cos\left(\mathbf{q}\cdot\mathbf{r}_i\right) \right)^2 + \left(\sum_{i=1}^{N} \sin\left(\mathbf{q}\cdot\mathbf{r}_i\right) \right)^2 \right] \tag{3.2}$$

The calculation of the static structure factor was implemented in ESPResSo++ within a class named `StaticStructF`. It inherits from `Observable`, making the system of particles available on C++ level. For the calculation two methods `computeArray()` and `computeArraySingleChain()` of `StaticStructF` were written, which compute the collective and the single chain structure factor, respectively. The definition of the static structure factor can be transformed algebraically as in equations 3.1 - 3.2. For the implementation equation 3.2 was chosen, because it neither contains a double sum nor an exponential function resulting in fastest performance.

The time consuming step of the computation is the access of memory. Therefore **q**-vectors are created as a part of the main computation loop rather than prior to it. This leaves the access of memory to the part of the summation loop, where the particle's position is requested. Accessing these positions N times, rather than N^2 causes a major acceleration scaling with the number of particles.

Furthermore equation 3.2 is favored due to the substitution of the exponential by sine and cosine and the absence of imaginary numbers. Both causing a small speedup in the calculation.

After deciding on the formula, the decision on parallelization was taken. The algorithm for computing the structure factor has to contain, in principle, two loops: one over different scattering vectors $\mathbf{q}$ and one over particle positions $\mathbf{r_i}$. Therefore parallelization can be performed clearly by distributing either the scattering vectors or the particle's to different tasks. Since ESPResSo++ is designed to calculate particularly big systems, consisting of 10^5 or 10^6 monomers, and one is sometimes interested in only a small range of scattering vectors (e.g. for calculation of the compressibility, cf. section 2.1.3), parallelization over the particles was chosen, so a great number of cores can be used even for a small number of scattering vectors[2].

For a two dimensional plot of the structure factor $S(\mathbf{q})$ one needs to reduce its dependency to a one-dimensional quantity. Since the systems under investigation are isotropic, changing from the scattering vector to its length is appropriate. Hence, an averaging over the values for scattering vectors of the same length has to be performed in addition to the calculation of $S(\mathbf{q})$,

$$S(q) = \langle S(\mathbf{q}) \rangle_{|\mathbf{q}|=q} = \frac{1}{n_q} \sum_{|\mathbf{q}|=q} S(\mathbf{q}) \tag{3.3}$$

where $q = |\mathbf{q}|$ and n_q is the number of $\mathbf{q}$-vectors with modulus q. According to [Bas94] it is advisable to not only average over scattering vectors of the same length, but average over vectors of similar length. The range of lengths averaged over then becomes a *bin size* and the averaging is replaced by *binning*. The computation for this binning can easily be used for the regular averaging of equation 3.3, by decreasing the bin size in such a way, that only vectors of the same length belong to one bin. Conversely, using the averaging calculation as binning is not as obvious. Therefore binning was implemented in the computation of the static structure factor (see lines 124 - 130 of the code below). Since users might want to go back to regular averaging, the bin size is kept adjustable via the parameter `bin_factor`. This factor is multiplied with the minimum grid distance (i.e. $\frac{2\pi}{L_{\max}}$) providing the size of the bins (line 130). A factor was chosen instead of an absolute value since the distance between $\mathbf{q}$-vectors is directly related to the size of the box. Thus, the number of $\mathbf{q}$-vectors sorted into a fixed bin scales with the box size as well. Scattering vectors that lie exactly on the boarder of two adjacent bins are sorted into the upper bin. This basically arbitrary choice matches the break condition for scattering vectors at the corners of the grid (see figure 6.1 and explanation below).This way to pigeonhole the $\mathbf{q}$-vectors also causes bin 0 to be empty, or more precisely, to only contain the structure factor for the zero scattering vector, which is the number of monomers. The numbering of bins is not shifted to keep the

[2]This decision is not crucial and will be different where appropriate, as in chapter 6

code more readable. Instead, bin 0 is taken out of the result by skipping it in the python list (appendix A.1.1 line 209).

Besides the `bin_factor`, `StaticStructF::compute` has three more parameters, nqx, nqy and nqz. These determine how far the creation of scattering vectors moves away from the middle of the grid on which they must lie (see section 5.2.1). Limiting the size of the scattering vector's components in this way causes the grid to have corners. Taking these corner vectors into account in the calculation gives rise to bad statistics at large moduli of the scattering vectors. Therefore scattering vectors of the corners were taken out by a modulus request inside the if statement of line 173.

```
116  //step size for qx, qy, qz
117  real dqs[3];
118  dqs[0] = 2. * M_PIl / Li[0];
119  dqs[1] = 2. * M_PIl / Li[1];
120  dqs[2] = 2. * M_PIl / Li[2];

122  Real3D q;

124  //calculations for binning
125  real maxX = nqx * dqs[0]; //maximum x value of a q vector
126  real maxY = nqy * dqs[1]; //maximum y value of a q vector
127  real maxZ = nqz * dqs[2]; //maximum z value of a q vector

129  real shortestDir = min(maxX, min(maxY,maxZ));  //#include<algorithm>??
130  real bin_size = bin_factor * min(dqs[0], (dqs[1], dqs[2]));


133  //            real q_sqr_max = nqx * nqx * dqs[0] * dqs[0]
134  //                    + nqy * nqy * dqs[1] * dqs[1]
135  //                    + nqz * nqz * dqs[2] * dqs[2];
136  //            real q_max = sqrt(q_sqr_max);
137  int num_bins = (int) ceil(shortestDir / bin_size);
138  vector<real> sq_bin;
139  vector<real> q_bin;
140  vector<int> count_bin;
141  sq_bin.resize(num_bins);
142  q_bin.resize(num_bins);
143  count_bin.resize(num_bins);

145  if (myrank == 0) {
146      cout << nprocs << " CPUs, new routine\n\n"
147              << "bin size \t" << bin_size << "\n"
148              << "q_max    \t" << shortestDir << "\n";
149  }

151  real n_reci = 1. / num_part;
152  real scos_local = 0; //will store cos-sum on each CPU
153  real ssin_local = 0; //will store sin-sum on each CPU
154  int ppp = (int) ceil((double) num_part / nprocs); //particles per proc

156  Real3D coordP;

158  python::list pyli;

160  //loop over different q values
```

```
161 //starting from zero because combinations with negative components
162 //will give the same result in S(q). so S(q) is the same for
163 //the 8 vectors q=(x,y,z),(-x,y,z), (x,-y,z),(x,y,-z),(-x,-y,z),...
164 for (int hx = -nqx; hx <= nqx; hx++) {
165     for (int hy = -nqy; hy <= nqy; hy++) {
166         for (int hz = 0; hz <= nqz; hz++) {

168             //values of q-vector
169             q[0] = hx * dqs[0];
170             q[1] = hy * dqs[1];
171             q[2] = hz * dqs[2];
172             real q_abs = q.abs();
173             if (q_abs > shortestDir){break;}

175             //determining the bin number
176             int bin_i = (int) floor(q_abs / bin_size);
177             q_bin[bin_i] += q_abs;
178             count_bin[bin_i] += 1;

180             //resetting the variables that store the local sum on each proc
181             scos_local = 0;
182             ssin_local = 0;

184             //loop over particles
185             for (int k = myrank * ppp; k < (1 + myrank) * ppp && k < num_part;
186                     k++) {
187                 coordP = config->getCoordinates(k);
188                 scos_local += cos(q * coordP);
189                 ssin_local += sin(q * coordP);
190             }
191             if (myrank != 0) {
192                 boost::mpi::reduce(*system.comm, scos_local, plus<real > (),
                        0);
193                 boost::mpi::reduce(*system.comm, ssin_local, plus<real > (),
                        0);
194             }

196             if (myrank == 0) {
197                 real scos = 0;
198                 real ssin = 0;
199                 boost::mpi::reduce(*system.comm, scos_local, scos, plus<real >
                        (), 0);
200                 boost::mpi::reduce(*system.comm, ssin_local, ssin, plus<real >
                        (), 0);
201                 sq_bin[bin_i] += scos * scos + ssin * ssin;
202             }
203         }
204     }
205 }
```

Furthermore, the limits of the scattering vector loop deserve explanation. The Z component `hz` ranges from zero to `nqz` whereas `hx` and `hy` include negative values. With this, double calculations are avoided. More precisely: Two vectors only differing in sign give the same contribution to the static structure factor. These vectors can be covered by leaving out the ones with negative Z component as table 3.1 shows. The vectors are grouped in pairs of $\mathbf{q}$ and $-\mathbf{q}$ with $\mathbf{q_1} = -\mathbf{q_8}$, $\mathbf{q_2} = -\mathbf{q_7}$ and so on.

Each vector of a pair has the same contribution to $S(q)$ as can be seen easiest from

equation 3.4.

$$S(\mathbf{q}) = \frac{1}{N} \left[\sum_{i=1}^{N} \sum_{j=1}^{N} \mathrm{e}^{-i\mathbf{q}\cdot(\mathbf{r}_i - \mathbf{r}_j)} \right] = \frac{1}{N} \left[\sum_{i=1}^{N} \sum_{j=1}^{N} \mathrm{e}^{i\mathbf{q}\cdot(\mathbf{r}_i - \mathbf{r}_j)} \right] \tag{3.4}$$

Double summation over all particles makes it possible to switch summation indices and since $\mathbf{r}_i - \mathbf{r}_j = -(\mathbf{r}_j - \mathbf{r}_i)$ the averaging over scattering vectors can be reduced to the set Q_+, which is the left side of table 3.1:

$$S(q) = \frac{1}{8} \sum_{|\mathbf{q}|=q} S(\mathbf{q}) = \frac{1}{8} \left[\sum_{\mathbf{q}\in Q_+} S(\mathbf{q}) + \sum_{\mathbf{q}\in Q_-} S(\mathbf{q}) \right] = \frac{1}{4} \sum_{\mathbf{q}\in Q_+} S(\mathbf{q}) \tag{3.5}$$

Since an averaging over scattering vectors of the same (or similar) length is performed, vectors with the same contribution can be left out of the calculation without additional correction. In other words, the factor $\frac{1}{4}$ in 3.5 is covered by the binning calculations, i.e. including it explicitly is not necessary. As figure 3.2 shows, the result differs, when a second component's negative values are kept out of the computation. From a mathematical point of view the above reasoning is enough to not consider the matter any further.

Thinking in terms of physics, the isotropy of the system suggests that using one octant (see figure 3.3) of $\mathbf{q}$-vectors[3] should be sufficient. However, isotropy is only assumed for the averaged system. In an averaged system the computation for one octant of $\mathbf{q}$-vectors comes very close to that with the full range. For one particular (not averaged) configuration, each octant yields slightly different results. In this sense the usage of four instead of one octant for the calculation is an averaging over octants. Besides this reasoning points out, that averaging over $\mathbf{q}$-vectors with the same modulus is only valid for isotropic systems. For the computation of the single chain static structure factor, the same method was used for a start, although a better distribution could be achieved here by parallelizing over the scattering vectors. In the computation for the single chain

[3]The isotropy of the system can easily be transferred to scattering vectors. The vectors of octant I give same result as the ones from octant II if the system is rotated by 90°.

①	②	③	④	⑤	⑥	⑦	⑧
q_x	$-q_x$	q_x	$-q_x$	q_x	$-q_x$	q_x	$-q_x$
q_y	q_y	$-q_y$	$-q_y$	q_y	q_y	$-q_y$	$-q_y$
q_z	q_z	q_z	q_z	$-q_z$	$-q_z$	$-q_z$	$-q_z$
	Q_+				Q_-		

Table 3.1.: The eight scattering vectors of the same length grouped in pairs only differing in sign, where $\mathbf{q_1} = -\mathbf{q_8}$, $\mathbf{q_2} = -\mathbf{q_7}$ and so on.

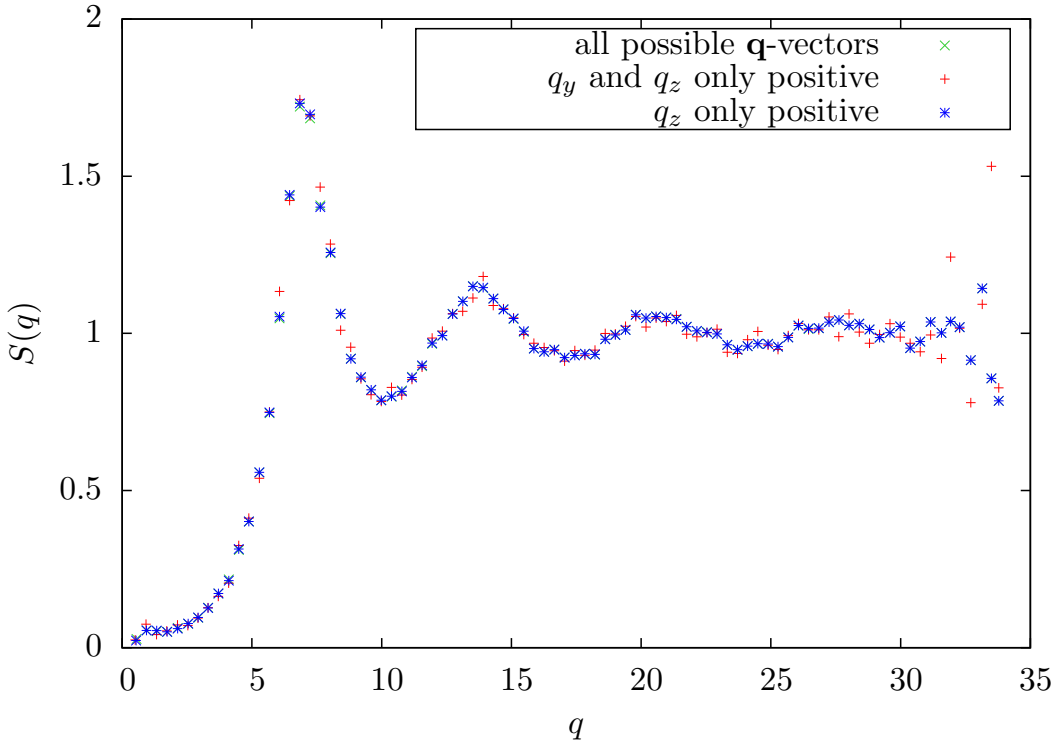

Figure 3.2.: Static structure factor calculated for all possible q vectors (green x), for all with non-negative Z component (blue asterisks) and for all with non-negative Z and non-negative Y component (red crosses).

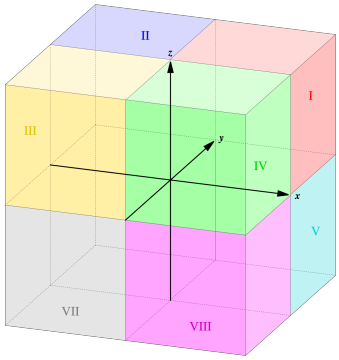

Figure 3.3.: Octants. Author: Lars H. Rohwedder (User: RokerHRO), URL: http://en.wikipedia.org/wiki/File:Octant_numbers.svg

structure factor, parallelizing over particles is subject to the condition that particles of one molecule have to be assigned to the same task. There is no such condition for scattering vectors. More importantly, the single chain computation is not affected by the box size of the whole system. This allows for a continuous choice of scattering vectors, making scattering vectors the preferable variable for the parallelization (cf. chapter 6).

3.3. Mean-square displacement

One of the three common mean-square displacement calculations was already implemented in ESPResSo++ previous to this work. The displacement of monomers with respect to the whole systems center of mass, g_1, is implemented as the member function `compute()` in the class `MeanSquareDispl`. The displacement of monomers in the chain's center-of-mass frame, g_2, and the chain's displacement in the system's center-of-mass frame, g_3, were implemented as `computeG2()` and `computeG3`, respectively as a part of this thesis.

They are implemented similar to g_1, but since they contain the chain's center of mass, it is necessary here to take into account the chain length. `MeanSquareDispl` inherits from the class `ConfigsParticleDecomp` which prepares the parallelization over particles. It decomposes the system by assigning a preferably equal number of particles to each task. The assignment is stored in a protected member variable `idToCpu`. This is a map, which means it contains 'keys' and 'values' and maps each key to a corresponding value. Here the keys are particle ID numbers and they are mapped to a CPU number[4]. With this map parallelization can be performed easily in all classes inheriting from `ConfigsParticleDecomp`.

3.3.1. Particle and chain decomposition

For a parallel computation of g_1 assigning the same number of monomers to each MPI task is desirable. The same holds true for every collective quantity which takes individual monomers into account, without considering to which molecule they belong. For these quantities the map `idToCpu` contains the information for a uniform distribution of monomers. It is filled within the constructor `ConfigsParticleDecomp(shared_ptr<System> system)` of `ConfigsParticleDecomp`:

```
116           int nodeNum = 0;
117           int count = 0;
```

[4] 'Task number' is the more accurate term, since the parallelization with boost::mpi uses MPI tasks. The tasks are distributed depending on the hardware. So a 'task' can stand for a CPU, a core of a CPU or some other computing unit. In the following code variable names contain 'CPU' (or some 'node'), such as `idToCpu`. This is why I used the term above.

```
118         for (vector<int>::iterator it = tot_idList.begin(); it!=tot_idList.end
                (); ++it) {
119           idToCpu[*it] = nodeNum;
120           count ++;
121           if(count>=local_num_of_part){
122             count = 0;
123             nodeNum++;
124           }
125         }
```

In the above code `tot_idList` is a vector storing the ID numbers of all particles. The map filling is performed using an iterator. Iterators guarantee to loop over every entry of the corresponding object (in this case a vector), but they do not ensure the order of accessing the elements. In MD simulations of polymers the monomers are commonly numbered consistent with the molecule they belong to, e.g. particles belonging to molecule 0 are numbered from 0 till 99, particles belonging to molecule 1 from 100 to 199, and so on. Therefore the order of monomers is essential for the computation of a molecules center of mass. Therefore, in the implementation of g_2 and g_3 the iterator-loop is replaced by a regular integer-loop. This way the particles are distributed in way, such that whole molecules are assigned to one task. This 'chain decomposition', instead of (single) particle decomposition, is implemented inside an overloaded constructor of `ConfigsParticleDecomp`. It contains the chain length as an additional parameter:

```
150   ConfigsParticleDecomp(shared_ptr<System> system, int _chainlength):
          SystemAccess (system){
151   //...

164         //for monodisperse chains
165         int num_chains = num_of_part / chainlength;
166         int local_num_chains = (int) ceil( (double)num_chains / n_nodes );
167         int local_num_part = local_num_chains * chainlength;

169         //in case the chainlength does not match the total number of particles
170         if(num_of_part % chainlength != 0){
171             cout << "chainlength does not match total number of particles\n"
172                     << "chainlength: " << chainlength
173                     << "\n num_of_part " << num_of_part << "\n\n";
174         }

176         //CPU0 will use particles 0, 1, 2, ... local_num_particles-1.
177         //CPU1 will use particles local_num_particles, local_num_particles
                +1,...
178         int nodeNum = -1;
179         for(long unsigned int id = 0; id < num_of_part ;id++){
180             if(id % local_num_part == 0) ++nodeNum;
181             idToCpu[id] = nodeNum;
182         }
```

The chain decomposition is prepared in line 115 by calculating the number of chains per MPI task. The local number of particles is then a multiple of the number of the chain length, ensuring that whole chains are computed by one MPI task.

3.3.2. Statistics for different time intervals

Besides chain decomposition, the implementation mostly consists of translating the definitions of g_2 and g_3 (see equation 3.6) into C++ code. The only difference being an averaging over different time intervals.

$$\mathrm{MSD} \equiv \langle(\mathrm{x(t)} - \mathrm{x(0)})^2\rangle = \begin{cases} g_1(t) & \text{where} \quad \mathrm{x_i} = \mathrm{x_{i,abs}} - \mathrm{x_{CMS}} \\ g_2(t) & \text{where} \quad \mathrm{x_i} = \mathrm{x_{i,abs}} - \mathrm{x_{CMC}} \\ g_3(t) & \text{where} \quad \mathrm{x_i} = \mathrm{x_{CMC}} - \mathrm{x_{CMS}} \end{cases} \tag{3.6}$$

Omitting this averaging for a start, the principal computation composes as follows: First, the number of gathered snapshots is obtained. Here 'snapshot' denotes a text file (usually with extension '.xyz' or '.pdb') which stores data for the configuration at one point in (simulation) time. The data contains inter alia particle ID numbers, positions and velocities. Snapshots are usually stored regularly during simulation, i.e. after a fixed number of MD steps corresponding to a fixed (simulation) time interval. For the calculation of the mean-square displacement only ID numbers and positions are needed, amongst their respective time. So here the snapshot number corresponds to the time. The respective centers of mass are calculated for each snapshot (i.e. for each available point in time). Subsequently the mean-square displacement is calculated within a loop over all snapshots (see line ...). Eventually the results from all MPI tasks are summed, divided by the number of particles and returned within a python list.

The formula above suggests to use a certain (and then every desired) snapshot at a time t together with the snapshot at time zero for the computation, e.g. snapshot $t = 5$ together with snapshot $t = 0$ for an interval of 5. Since we are interested in the displacement per time interval (rather than at absolute times), we can also use snapshot $t = 6$ together with snapshot $t = 1$ and snapshot $t = 7$ with snapshot $t = 2$ and so on (for an interval of 5). The averaging over different time intervals is introduced as another loop inside the (first) snapshot loop. With this, all intervals of the same length are taken into account in the calculation (see equation 3.7, $\Delta\tau$ denotes the time difference between subsequent snapshots).

$$g(n \cdot \Delta\tau) = \langle(\mathrm{x}(\tau_\mathrm{j}) - \mathrm{x}(\tau_\mathrm{i}))^2\rangle \qquad \text{with} \quad \mathrm{n} \cdot \Delta\tau = \tau_\mathrm{j} - \tau_\mathrm{i} \tag{3.7}$$

This yields better statistics for each interval, except for the largest one. The precision of the calculated value increases with decreasing time difference. At the same time this procedure averages out differences that might occur in the course of the simulation. So if, for example, all particles would have a greater change in position during a certain interval at the beginning of the simulation than at the same interval in the middle and end, this would not be visible in the result, only the results would be slightly greater. For a displacement per simulation time, only the averaging over particles can be used. With equilibrated systems, one is interested in the mean-square displacement per time interval, so this second averaging is valid and improves the accuracy of the results.

4 Systems

All polymeric model systems investigated in this thesis are linear melts, i.e. they purely consist of polymer chains in a liquid state without solvent. The chains are modeled by beads and springs, where the beads represent the monomers and the springs represent the bonds connecting two monomers of a chain. The interaction between monomers belonging to different chains is also integrated in the model by a so called non-bonded potential, which is different from the bonded spring potential. Analysis is performed on equilibrated systems. Producing such consists of mainly three steps. First, the chains are generated. Secondly, they are equilibrated and thirdly the data required for further investigation is saved. Systems of different chain stiffness were examined, which means their distributions of angles between bonds differed. The stiffness is accounted for at both the first step of chain generation and the second step of equilibration, whereas the potentials are only applied in the equilibration stage (and in the simulation itself; see section 4.3). This chapter explains the three steps to obtain equilibrated configurations. The model and the equlibration are based on the work of Moreira et al. [Mor14].

4.1. Model and chain generation

The chains are modeled as bead-spring systems. Each bead represents a monomer as a sphere of fixed diameter σ and mass m. Chains are generated as *non-reversal random walks*. A random walk is a procedure of setting up a chain with an equal and fixed bond length b. It starts with one bead and places the second bead a distance b apart, but in a random direction. The next step uses the second bead as a starting point and,

again, the third bead is placed in a random direction in a distance of b from the second bead. These steps are repeated up to the last bead. A freely-jointed chain is described by a (completely) random walk. In a non-reversal random walk the steps are executed mainly in the same way, but with a restriction on subsequent beads. Assume the beads are numbered consecutively, then bead $(i-1)$ and bead $(i+1)$ are required to have a minimal distance l_{min}. This distance corresponds to the stiffness of the chain. Stiff chains (stiffness $k_\theta > 0$) are created with a bigger distance l_{min}, whereas freely-jointed chains ($k_\theta = 0$) are created without a minimal distance between beads $(i-1)$ and $(i+1)$, tantamount to a random walk. This also means that a random walk can fold back, meaning two beads $(i-1)$ and $(i+1)$ are assigned the same position. Therefore the freely-jointed chains are also referred to as *fully flexible* chains. In a non-reversal random walk folding back is prohibited by the demanded distance l_{min}. A chain with stiffness is described by a non-reversal random walk and it is also called *semi-flexible* chain.

After the creation via a (non-reversal) random-walk, the chains are randomly placed in a cubic simulation box. The size of the box is chosen such that the number density of beads reaches a value of $0.85\sigma^{-3}$ for each system.

Interactions apply when a simulation is started. The unit of energy for the simulation is ϵ and the suitable unit of time is τ, with $\tau = \sqrt{(\sigma^2 m/\varepsilon)}$ (σ and m are the diameter and the mass of a bead). Equations 4.1 show the potentials which model the interactions of the system.

$$U_{\mathrm{WCA}}(r) = \begin{cases} 4\varepsilon\{(\sigma/r)^{12} - (\sigma/r)^6 + \frac{1}{4}\} & \text{for } r \le r_c \\ 0 & \text{for } r > r_c \end{cases} \tag{4.1a}$$

$$U_{\mathrm{FENE}}(r) = \begin{cases} -0\ 5kR_o^2 \ln[1-(r/R_o)^2] & \text{for } r \le R_o \\ \infty & \text{for } r > R_o \end{cases} \tag{4.1b}$$

$$U_{\mathrm{bend}}(\theta) = k_\theta(1-\cos\theta) \tag{4.1c}$$

The bonded potential, for the neighboring beads of a chain, is described by a finite extensible non-linear elastic potential (FENE). The spring's constant k in the FENE potential is set to $k = 30\,\varepsilon/\sigma^2$ and its maximum extension R_o is set to $R_o = 1.5\sigma$. Interactions between non-bonded beads are described by a truncated Lennard-Jones potential. If one cuts the Lennard-Jones potential at its minimum $r_c = 2^{1/6}\sigma$ shifts the left part ($r \le r_c$) by the minimum value, such that the repulsive wall eases to zero at r_c and sets the right part to zero entirely, one attains a purely repulsive short range potential. It is called WCA potential after Weeks, Chandler and Anderson ([Wee71], 5238).The point where it reaches zero is labeled r_c because it is the *c*utoff radius. In the case of chains with stiffness constant $k_\theta \neq 0$ another potential is applied. This bending potential is described by U_{bend}, where θ_i is the angle between beads $i-1$, i and $i+1$ (see figure 4.1). The steps where the potentials are used are described in the next section.

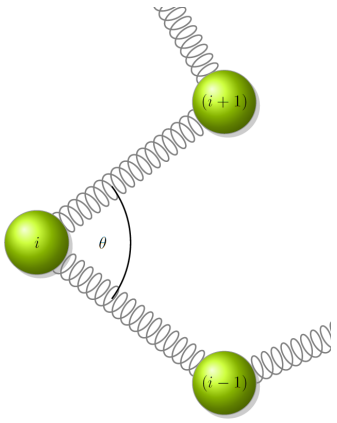

Figure 4.1.: Illustration of the bond angle θ

4.2. Equilibration

Thermodynamic systems reach, after a long enough time, an equilibrium at which the quantities examined in this work are measured. In order to achieve this state also for the simulated system, one has to apply suitable interactions to the particles for a long enough time. Since the chains are generated and placed in the simulation box randomly, a great overlap of beads is probable. Because the potentials (4.1) rise rapidly for short distances, this will produce huge repulsive forces. These lead to distorsions in the system. Moreover, the high forces evoke numerical errors and huge particle velocities, which spread cascaded and do not ease. This is called an *explosion*. Hence, one cannot apply the forces at once, but needs an equilibration technique, which prevents from the explosion. As mentioned earlier, a goal of computer simulations is to further investigate how microscopic behavior relates to macroscopic quantities. In order to do so, microscopic characteristics must be the same as in experiments. Here this means, the single-chain statistics must be preserved. The equilibration procedure is mainly the one suggested by Auhl et al. ([Auh03]), slightly modified in the warm-up.

This equilibration consists of three phases: pre-packing, warm-up and relaxation. Pre-packing uses the Monte Carlo method to reduce local density fluctuations[1]. During the pre-packing the chains are moved as rigid bodies. As a consequence, the correct single-chain characteristics, which apply to the intial chains, are kept during pre-packing. Possible movements are translation, rotation, reflexion, inversion and the swap of two chains. A move is accepted and conducted, if it reduces local density fluctuations.

The next stage of equilibration is the warm-up phase. It consists in a molecular dynamics simulation (MD simulation). So this stage makes use of the potentials above

[1] An even density distribution is one characteristic of equilibrium. Smoothing dense regions is also a first step in preventing from a numerical explosion, since particle overlaps are more likely in denser areas.

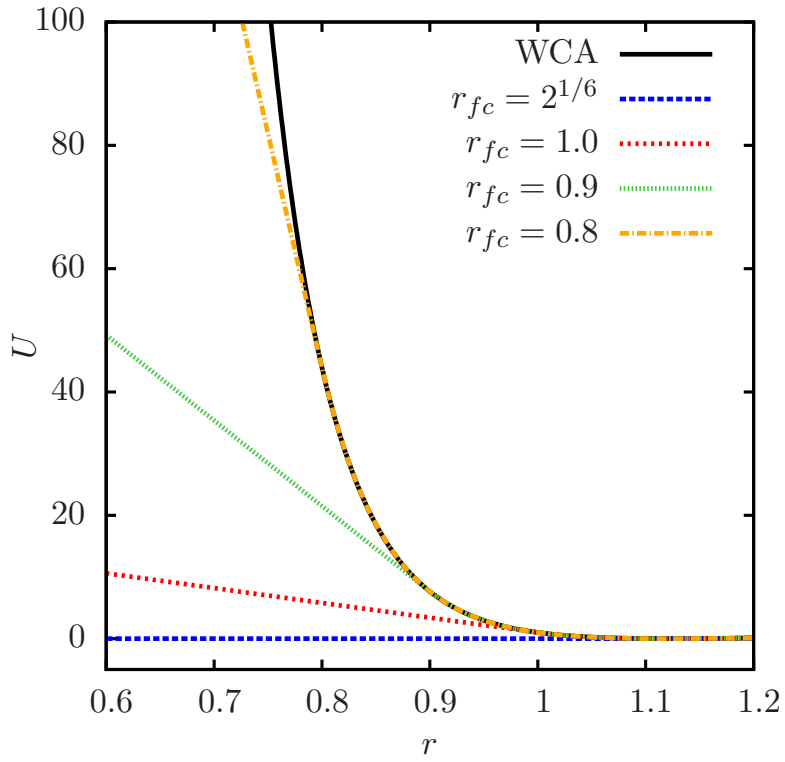

Figure 4.2.: The WCA potential is truncated via a force-capped radius r_{fc} during the warmup.

(4.1). Again, the system must be prevented from explosion. Therefore the friction of the system is set to high value (of $\Gamma = 1.0$) and the basic time step of the simulations is chosen very small ($\Delta t = 0.0001\,\tau$). During small times particles can only move short distances and when they departed a little the forces will be less in the next time step. Most notably, at the beginning of the warm-up potentials are truncated in a way that keeps the chain characteristics, but counteracts numerical errors. They are slowly morphed into full potentials given in section 4.1. The morphing is realized via a *force-capped radius* r_{fc} as shown in figure 4.2. For distances greater than r_{fc} the potential equals the regular WCA potential. To the left of r_{fc} the potential extends linearly with the same slope, resulting in a constant force for small distances. During the first part of the warm-up the force-capped radius is adjusted such that mean squared internal distances between the beads of chain remain the same. In the last part of the warm-up force-capped radius is reduced linearly and the full WCA potential is reached.

After pre-packing and warm-up the last phase of equilibration starts: the relaxation. This is an MD simulation, too, where the friction coefficient is reduced compared to the warm-up ($\Gamma = 0\ 5$) and the time step is enlarged (first $\Delta t = 0\ 001\,\tau$, later $\Delta t = 0\ 005\,\tau$). The relaxation uses full potentials (eq. 4.1).

4.3. Configurations from simulation

Once an equilibrated system is produced, further simulations are conducted. They produce the data from which the desired quantities are calculated. There are pricipally two ways of analyzing the data: *online*, which means while simulating or from files after the simulation. In order to perform online analysis with ESPResSo++, the user writes the analysis commands in the same Python script as the simulation commands so they are executed together. This is especially advisable for small systems, where a new generation of the system takes less time and computational power than reading in a stored configuration from a textfile[2]. The analysis from files is performed with two separate scripts. The simulation script contains commands to save data necessary for analysis in files. The analysis script starts with reading the files and calls the analysis functions subsequently.

In this thesis files are used to analyze the structure and the monomer displacement of the systems. The mean squared displacement computes from particle positions together with their time. The particular centers of mass for the computation are also calculated from those values. The data is collected during an MD simulation as in [Mor14]. All particle positions are stored at the desired number of time steps. This is realized by taking *snapshots*. Here, a snapshot is a file, which contains particle positions (together with id numbers and velocities. File extension '.xyz'.). The static structure factor computes from different configurations. Those are obtained basically in the same manner. Only the interval between the snapshots used for the structure factor is large. This is why I refer to them as different *configurations* rather than different snapshots (in time). Of course, different configurations can also be taken from different initial setups.

All MD simulations mentioned in this section are executed with ESPResSo++. The software uses a velocity verlet algorithm ([All89], 78 *ff.*) in the integrator. Furthermore, all simulations named in warm-up, relaxation and simulation are performed at a constant volume (NVT simulations) and use periodic boundary conditions ([All89], 24 *ff.*).

[2]This process is not parallelized up to now.

5 Results

This chapter presents the results of the implementations of the static structure factor and the mean-square displacement described in chapter 3. In the first section, performance of the usage of multiple tasks is shown. The second section lists some results of computations executed with the newly implemented functions.

5.1. Implementation results

This section shows how the speed of the computations scales with the number of tasks used.

5.1.1. Static structure factor

Figure 5.1 shows the speedup of computation time using a higher number of cores. The speedup is the quotient of the computation time with one core and the time with n cores:

$$\text{speedup} \equiv \frac{\text{time (1 core)}}{\text{time (n cores)}} \tag{5.1}$$

The dashed line is the identity. The non-linear increase in speedup for the computation with eight cores could be due to the "CPU caches". A cache is a hidden temporary storage each CPU possesses, designed to increase computational efficiency (see figure

5.2). In the computation of the static structure factor the important temporarily stored data consists of the particle positions. Therefore the amount of data decreases with increasing number of cores. From eight cores on, it might be that all local particle positions fit in the cache, speeding up the computation at a higher rate than before. For a further increase of the number of cores this effect still occurs. So I would expect the data points to have a constant offset of the dashed line (the identity) from the point where the cache effect occurs on. The slope will still be close to one at first, but decrease at the number of cores where inter-core communication takes more time than the usage of that number of cores saves. For larger systems (more particles) both effects should occur at a higher number of cores.

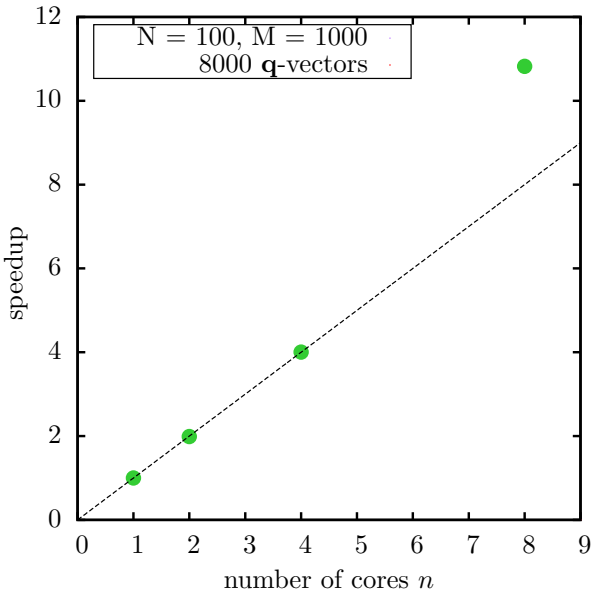

Figure 5.1.: Speedup for static structure factor computation

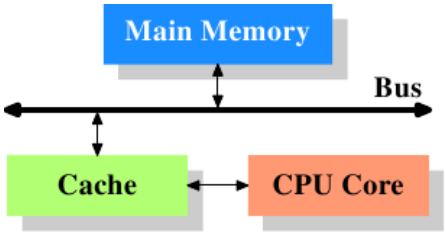

Figure 5.2.: Minimum cache configuration

5.1.2. Mean square displacement

An additional speedup as for the computation of the static structure is also visible (figure 5.3) for the mean square displacement. It is due to the same effect, since the implementation uses particle decomposition.

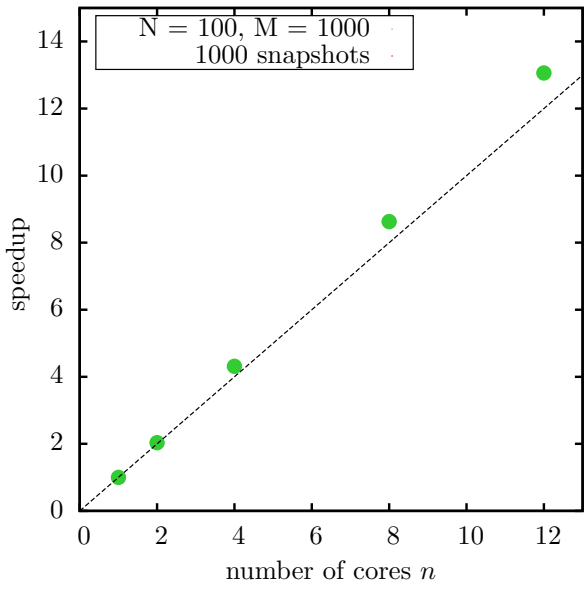

Figure 5.3.: Speedup for the computation of g_1

5.2. Computation results

This section shows computation results performed with the static structure factor analysis routine of ESPResSo++. Their physical meaning is discussed in relation to the equilibration procedure (described in chapter 4). Also polymer chains of different length and stiffness are considered. The first subsection contains results of the computation of the single-chain structure factor. The second subsection portrays the collective structure factor computed for different systems. All results are obtained as an average over five different configurations, if not stated differently.

5.2.1. Single chain structure factor

The single-chain structure factor, or the form factor, is a measure for the structure of individual chains. In figure 5.4 the single-chain structure factor is plotted along with

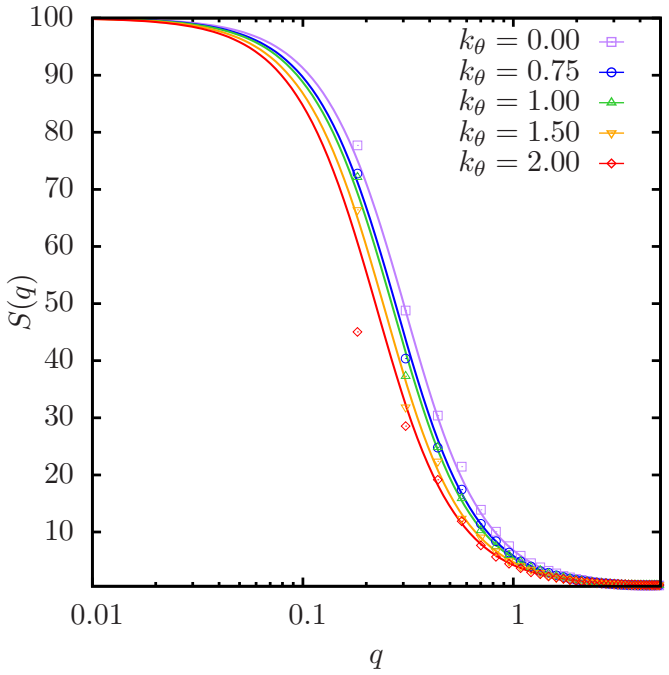

Figure 5.4.: The static structure factor for initial non-reversal random walk systems. The solid lines stand for the single chain structure factor, the markers stand for the collective structure factor. All systems contained 1000 chains with 100 beads per chain. The data is obtained by an average over three configurations. For the single chain structure factor the prefactor is adjusted matching the collective one ($\frac{1}{M \cdot N}$ instead of $\frac{1}{M \cdot N^2}$ in eq. 2.10). Markers were computed with ESPResSo++ (prefactor $\frac{1}{M \cdot N}$).

the collective structure factor. Solid lines correspond to the single chain structure factor (multiplied by the chain length N). The markers stand for the collective structure factor. The markers resemble the lines, especially for values from $q = 0.4$ on. This conformity implies, the structure of the whole system is similar to the one of a single chain. Since the investigated system is the initial non-reversal random walk configuration, structure can only result from single chains. The position of chains cannot contribute, for the chains are placed randomly in the simulation box.

The graph also exhibits some general charcteristics of a structure factor from a simulated system as well as some consequences of the particular implementation. A lower limit for data points is visible between $q = 0.1$ and $q = 0.2$. This is an artefact of periodic boundary conditions, which limit scattering vectors to multiples of $q_{\min} = \frac{2\pi}{L}$, where L is the box length (see section). So in a system of 100 000 beads with a number density

of $0.85\sigma^{-3}$ the shortest scattering vector has a length of

$$q_{\min} = \frac{2\pi}{L} = \frac{2\pi}{(V)^{1/3}} = 2\pi \cdot \left(\frac{100000}{0.85}\right)^{-\frac{1}{3}} \approx \frac{2\pi}{49.00} \approx 0.13. \tag{5.2}$$

This matches the position of data points for the smallest scattering vector in the graph. Conversely, there is no lower limit to the single chain structure factor values, because the restriction of the periodic boundary does not apply to single chains. Thus, scattering vectors can be chosen densely here, even for small q-values giving rise to the continuous lines.

Another consequence of the restriction on scattering vectors is inferior statistics for smaller scattering vectors. More precisely, for those values of $S(q)$, which are computed from a small number of scattering vectors. In the computation conducted in this work, with increasing length of $\mathbf{q}$, a growing number of scattering vectors is used to calculate one value of the structure factor (see figure 6.1). For example, only three scattering vectors of the minimal length $q_{\min}$ can be created for a cubic simulation box: ($q_{\min}$, 0, 0), (0, $q_{\min}$, 0) and (0, 0, $q_{\min}$). Hence, the value of $S(q)$ for $q = q_{\min}$ is only averaged over three values of $S(\mathbf{q})$.

As a result, the fluctuation of the static stucture factor between different configurations is higher for small q. In figure 5.4 this characteristic becomes visible at the difference between markers and lines. Collective values for the smallest q-moduli deviate majorly from the single chain structure factor data, whereas the collective markers are in perfect agreement with the lines for higher q-moduli.

Albeit hidden within the logarithmic scale of the X axis, the even bin spacing of the the implementation becomes visible: One can see that vertical spaces between data points follow the logarithmic scale. More easily, the constant bin size can be seen from the figures in the following subsection (5.2.2, especially figures 5.5 and 5.8).

5.2.2. Collective structure factor

The collective static structure factor is a measure for structure and density fluctuations on all length scales. An even density distribution is characteristic for the desired equilibrium configurations. A flat structure factor at low wavenumbers indicates an even density distribution. The initial systems of non-reversal random walk chains used in this project do not exhibit this flattening, as the green markers of figure 5.5 depict. Contrawise, they increase steeply as $\mathbf{q}$-vectors approach zero. Pre-packing, the first stage of the equlibration procedure, already improves the density distribution closer towards equilibrium, as markers go down for small q-values. A better resolution of the benefits of pre-packing is given in figure 5.6. The plot is displayed on non-linear scales to make the difference at small q-values visible in more detail. At small q pre-packing

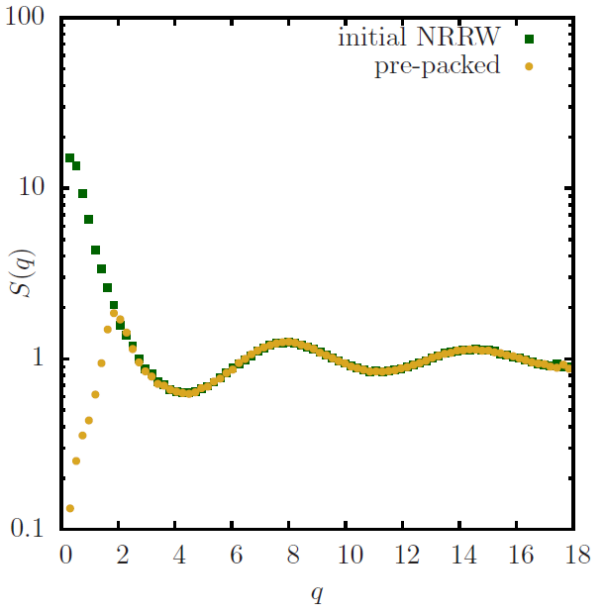

Figure 5.5.: Collective structure factor for the initial and the pre-packed configuration of 1000 fully flexible chains of chain length 20

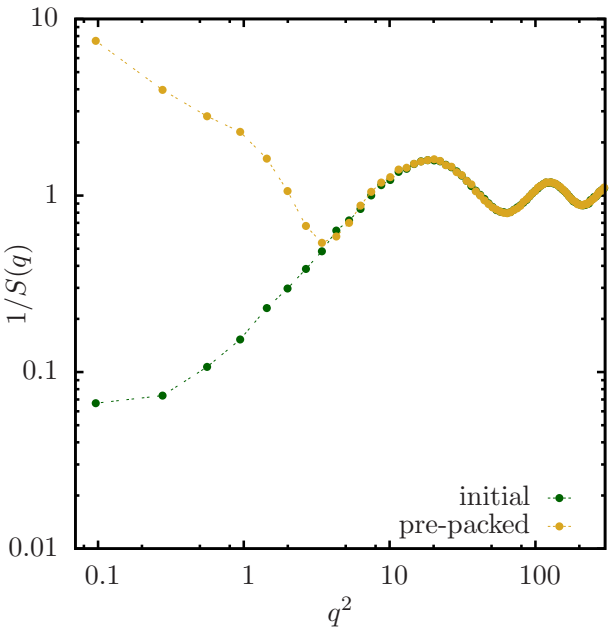

Figure 5.6.: Reciprocal structure factor for the initial and the pre-packed configuration of 1000 fully flexible chains of chain length 20

reduced the structure factor by two orders of magnitude. Since small values of q correspond to large distances in space, density fluctuations are reduced globally and in mid-size areas of the system. At large wave numbers, i.e. large values of q, the results are the same for the initial and the pre-packed configuration. Pre-packing moves chains as rigid bodies, which means that, on length scales smaller than the chain size, it does not alter the structure of the system. Hence, the equality for large values of q agrees with the expectation. Therefore, the kink in the plot for the pre-packed configuration probably corresponds to the chain size. As further evidence, the kink disappears both

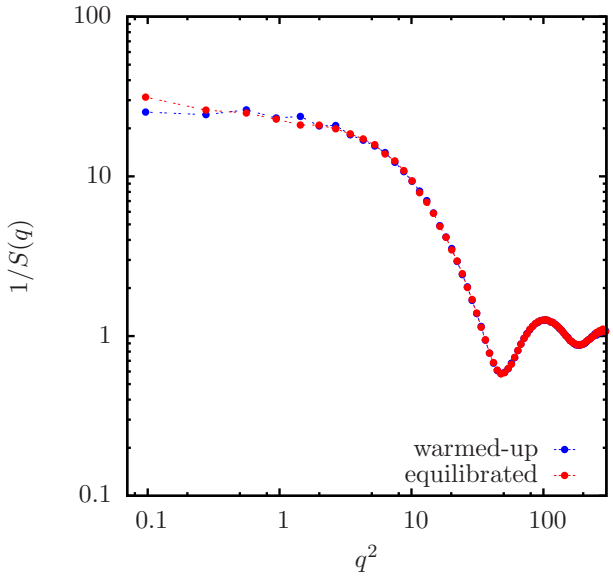

Figure 5.7.: Reciprocal structure factor for the warmed-up and the equilibrated configuration of 1000 fully flexible chains of chain length 20

in the warmed-up and in the equilibrated stage, as figure 5.7 shows.

The results of the static structure factor computations for a system of 1000 fully flexible chains with 20 beads each are plotted in figure 5.8. The typical form of a structure factor for equilibrated polymer chains is now visible (cf. figure 2.2(a)). The flat beginning of the plot at low values of q indicates an even distribution of beads, which is characteristic for equilibrated melts and at the same time indicates low compressibility (see equation 2.21). The first peak corresponds to the most probable distance between particles in the system ([Bin05], 43). In this case its position agrees with the bondlength. One reason is, that the neighbouring beads of a chain are connected by FENE springs which give, together with the WCA potential (for all pairs of beads), a narrow minimum. So the distances between neighbouring beads are similar. Also the point at which the first peak occurs, at about $q \approx 7$ compares to the average value of the bond length $\langle b^2 \rangle^{\frac{1}{2}} = 0.97\sigma$, since $\frac{2\pi}{q_{\min}} \approx 0.90$. One might argue that the difference is significant. For a crystal lattice, in which the distance 0.97 between neighbouring atoms is the most frequent,

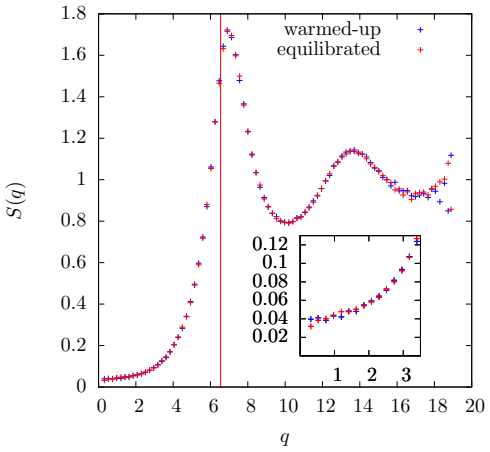

Figure 5.8.: Static structure factor for the warmed-up (blue squares) and equilibrated (red circles) configurations. They consisted of 1000 fully flexible chains of chain length 20. The vertical line corresponds to the favoured bondlength $b = 0.96\sigma$ with its position at $\frac{2\pi}{b}$.

the peak in the static structure factor, would be at $\frac{2\pi}{0.97} \approx 6.48$. In figure 5.8 the peak is at a greater q value. This shift must be a consequence of the non-crystlline structure ([Bas00], 6366). Non-bonded nearest neighbours can only have a minor contribution.

type of distance	**distance** d	**corresponding** $q = \frac{2\pi}{d}$
favoured bondlength	0.96	6.54
initial bondelength	0.97	6.47
average distance between nearest neighbours $(0.85^{1/3})$	1.06	5.95

Table 5.1.: Characteristic distances in the system

Table 5.1 lists the typical distances in the system along with their value in reciprocal space. The average distance between two beads is calculated from the average density of beads. If it was structurally important, it would produce a peak further to the left of the one observed in figure 5.8 or rather shift that peak further. This indicates, that the variation of the non-bonded shortest distance is significantly larger than that of the bondlength. Since this is true for covalently bonded monomers, the model potentials are chosen appropriately in this regard. In table 5.1 both the favoured bondlength (obtained from the minimum of the model potential) and the initial bondlength (used for the creation of chains) are mentioned, in order to show, that the favoured bondlength provides a slightly better estimate of the peak. However, the two are too close together to draw a conclusion on the structure from this fact. Since the factor is always an

average over different configurations it still contains deviations. The smaller peaks at greater q-values indicate a loss of spacial correlation.

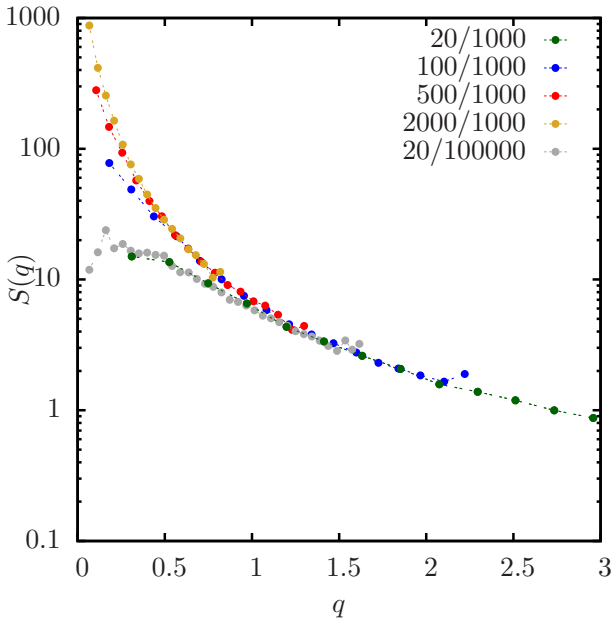

Figure 5.9.: Structure factor for the initial configurations of 1000 fully flexible chains with different chainlenght (colored markers). Gray markers correspond to a system of 100 000 chains with chainlenght 20. The numbers in the key refer to chainlength N and number of chains M as "N/M". The data is obtained from an average over three configurations for systems with 1000 chains. The data for the case of 100000 chains is unaveraged.

As figure 5.9 shows, chainlength effects the structure factor of the initial configurations only in the range of small q. The longer the chain, the higher the value of the $S(q)$ if q appoaches zero. A high value can be interpreted as high density fluctuations and means that the system is highly compressible. For the shortest examined chainlegth, a second sample with 100 times as many beads was investigated. Its structure factor resembles the one of the first system with chainlegth 20. This is a first indication that the number of chains does not effect the structure here[1].

Figure 5.10 explores the dependence on the number of chains further. It shows the structure factor of intial systems with different numbers of chains and two different stiffness constants. Just as in the case for short chains (chainlength 20, in figure 5.9), the structure factor is independent of the number of chains for medium sized chains (chainlength 350). The data points for the semi-flexible chains are lower than the ones for fully flexible chains. Since the plot shows low wavenumbers, this means stiffer

[1] Density is kept constant by adjusting the box size.

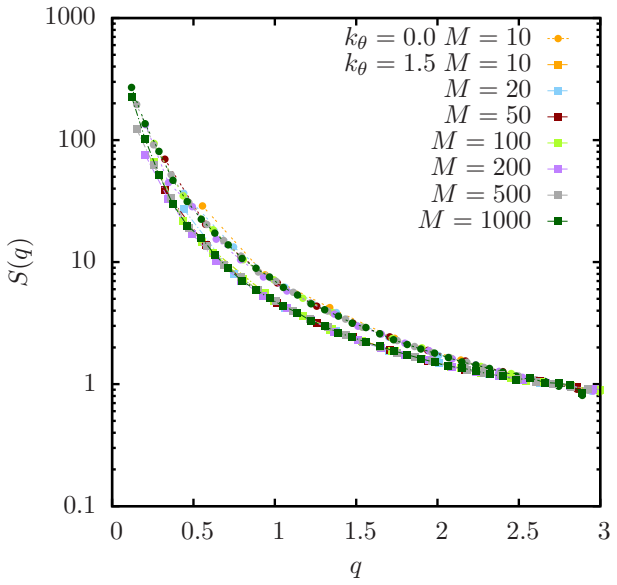

Figure 5.10.: Structure factor for initial configurations of a different number of chains with 350 beads each. Circles correspond to fully flexible chains ($k_\theta = 0$), squares to semi-flexible chains with a stiffness constant of $k_\theta = 1.5$. The data is obtained by an average over three configurations.

chains show, in their initial setup, less density fluctuations on large length scales. On the largest scale they assimilate, since the graphs meet for the smallest scattering vectors. In the limit of low scattering vectors, the structure factor relates to the isothermal compressibility (see equation 2.21). This relation fits descriptively to the interpretation in terms of density fluctuations. If there are global or medium scale density fluctuations, which can be depicted as big holes, the system is more compressible. A plot of the limit $\lim_{q\to 0} S(q)$ is given in figure 5.11. It shows the limit values depending on the chain length for all investigated chain flexibilities. The first feature that attracts attention is the insignificance of chain stiffness. It does barely contribute to $S(q \to 0)$ and hence neither to the compressibility. Only at the pre-packing stage (empty markers) a very slight trend is visible: Stiff chains (marked by red diamonds) are slightly less compressible than fully flexible chains (purple squares) of the same length. Or in other words, after pre-packing the density fluctuations in fully flexible systems are slightly bigger than in the stiffest investigated systems. However, this minor difference can be neglected compared to big jumps that occur between the three stages of equilibration. The general trend is a reduction of the compressibility in agreement to the desired reduction of density fluctuations. Pre-packing reduces $S(q \to 0)$ by two orders of magnitude and warm-up and equilibration reduces it further. For the initial and the pre-packed configurations the low wavenumber limit of the structure factor increases with chain length. Descriptively, longer chains produce bigger holes when randomly

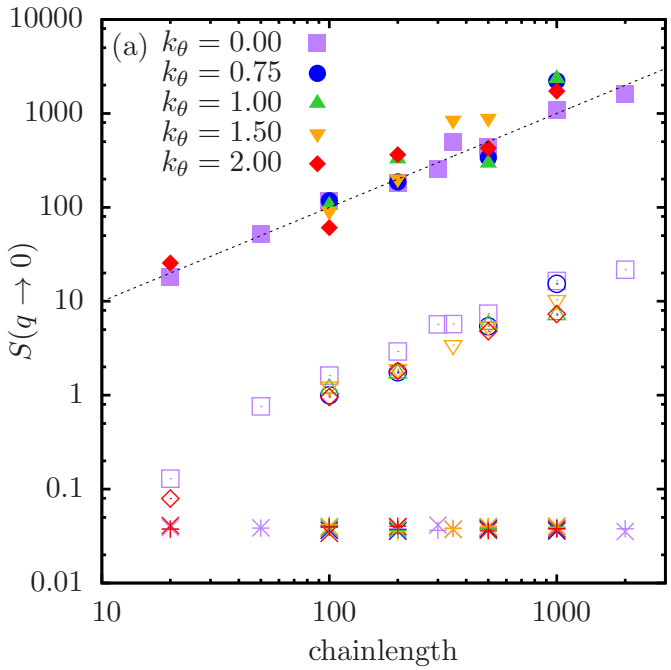

Figure 5.11.: The low-q-limit of the collective structure factor for different chain lengths, stiffnesses and stages in the equilibration procedure. Filled markers belong to the initial configuration, empty markers to the pre-packed stage and crosses to equilibrated configurations. The dashed line shows the identity. All systems contained 1000 chains. The data is obtained from an average over three configurations each.

placed in a simulation box. The equlibrated configurations show no dependence on chain length, which means, in the picture of holes, that holes vanish during equilibration for every investigated chain length. For the shortest chains ($N = 20$) the data points in the pre-packed configuration almost meet the ones from the equilibrated one. This means that, for short chains, already pre-packing nearly evens out density fluctuations. It should also be noted, that the data points for the initial configurations are directly proportional to the chainlength. They follow the dashed line, which is the identity. For all systems used for the plot contained the same number of chains (1000). Since they also have the same density, the box volume grows together with the chainlength. Therefore the observed dependency might be due to the box size rather than the chain length. However, figure 5.10 shows that the structure factor is not sensible to the number of chains for short q-vectors and therefore neither to the box size. In fact, the compressibility for initial configurations grows with the chain length. This trend can be understood from the relation to the single chain structure factor (see figure 5.4). As mentioned earlier, the structure of initial configurations only results from the structure of individual chains. The intermolecular sum becomes zero. Thus, the second term of equation 5.3 vanishes (cf. theory, p. 7).

$$S(\mathbf{q}) = NP(\mathbf{q}) + MNQ(\mathbf{q}) \tag{5.3}$$

The intramolecular or single-chain part becomes one as q approaches zero:

$$\lim_{q\to 0} P(\mathbf{q}) = \frac{1}{N^2}\sum_{k=1}^{N}\sum_{l=1}^{N}\lim_{q\to 0}\cos[\mathbf{q}\cdot(\mathbf{r}_k - \mathbf{r}_l)] = \frac{1}{N^2}\sum_{k=1}^{N}\sum_{l=1}^{N} 1 = 1, \tag{5.4}$$

leaving the limit of the collective structure factor $S(q)$ as q approaches zero proportional to the chainlenght N.

6

Further improvements

This chapter suggests and describes ideas to enhance the computation of the static structure factor, as described in this work. The current version can be improved mainly by a reduction of computation time. A minor improvement consists in a more intuitive way of user input for the parameters. Further speedup can be obtained by reducing the number of scattering vectors of a given length that are used for the calculation, especially in the case of long $\mathbf{q}$-vectors. At first the way of reduction is described. Subsequently some tests of this methods are displayed. At last an impementation is provided.

Since, for systems with periodic boundary conditions, scattering vectors have to lie on a usually cuboid shaped grid, and binning is executed according to the vector's modulus, bins of higher number contain the results for more scattering vectors. Figure 6.1 shows these circumstances for two dimensions.

A first and easy approach to reduce the number of scattering vectors was performed by skipping certain grid points by using a modified box size for the computation of the static structure factor. More precisely, a cubic system with an actual box length L was assigned a fake box length of $\frac{1}{2}L$ for the computation resulting in a skip of every other grid point. See figure 6.2 for a geometrical representation. Figure 6.3 shows the results of analysis with modified box lengths. The results were obtained from a version of the program which still contained the scattering vectors at the corners of the grid producing a tail with bad statistics. Along with the modification of the box length comes a change in the bin size, since this is directly related to the box length. So for a coarser grid of scattering vectors, a coarser binning is applied automatically. Figure 6.4 shows the

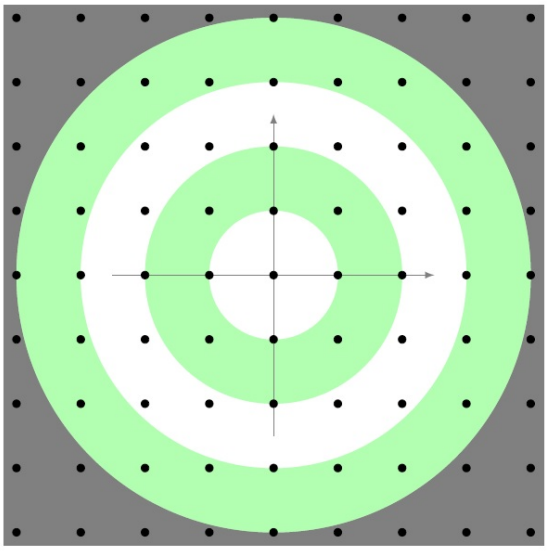

Figure 6.1.: Showing a squared grid with evenly spaced circles. The grid points correspond to scattering vectors, the shells (here in green and white) represent the bins in the static structure factor computation. Vectors on the boarder count to their inner shell, vectors in the corners (gray background) are left out of the computation.

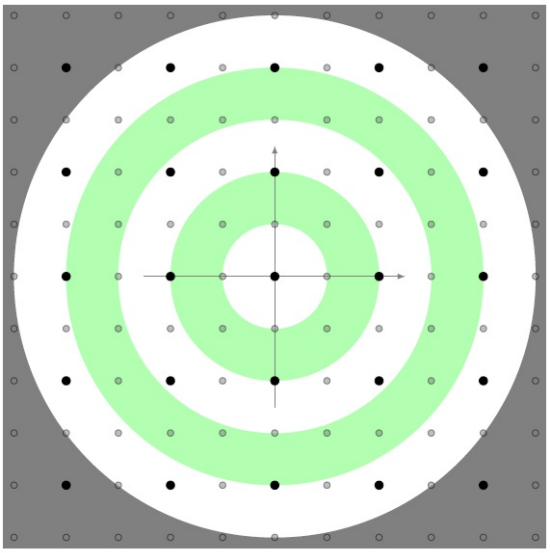

Figure 6.2.: Skipping every other grid point by providing a fake box length of half the size as the actual one

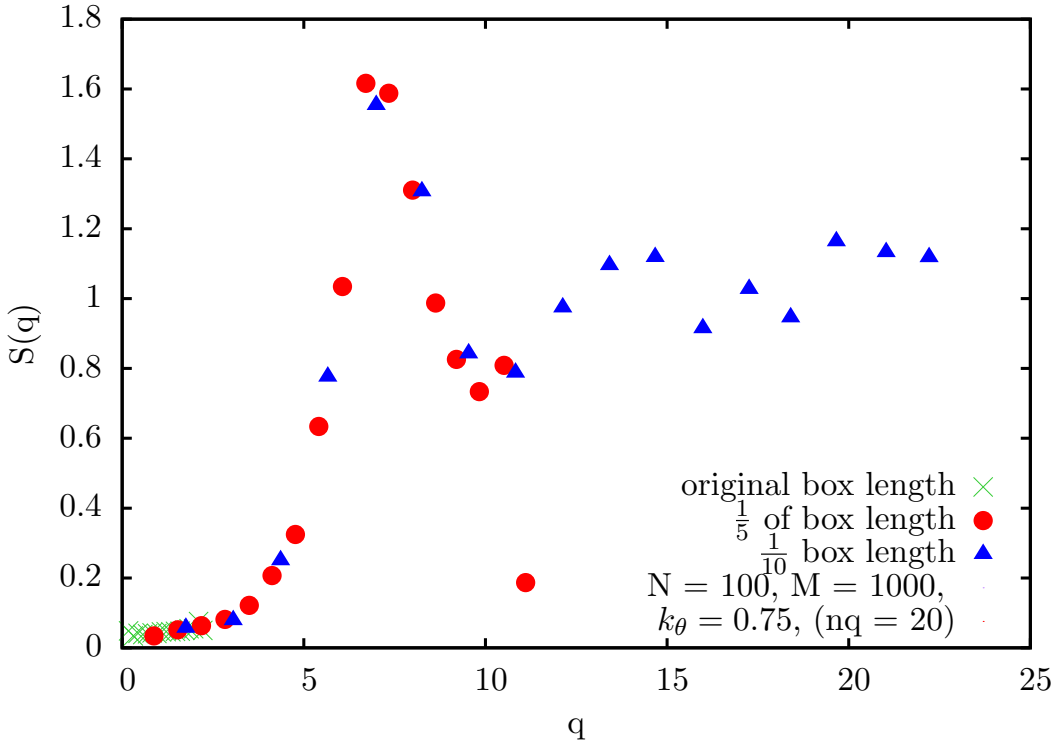

Figure 6.3.: Results of the static structure factor computation with artificial box lengths. Green crosses: actual box length of the system. Red circles: $\frac{1}{5}$ of the actual box length, i.e. every fifth **q**-vector is taken into account. Blue triangles: $\frac{1}{10}$ of the actual box length

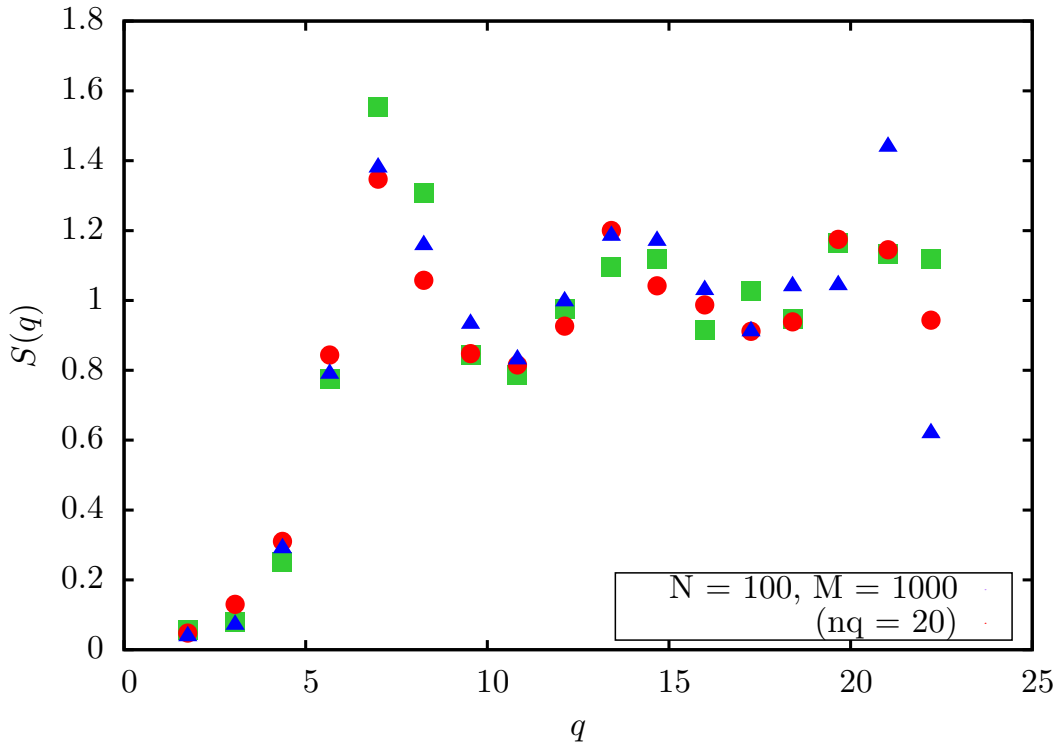

Figure 6.4.: Structure factor for three different configurations. Each type of marker corresponds to a single configuration, i.e. an independent snapshot of the equilibrated system.

structure factor of three different configurations. The box length in the computation was modified by the factor $\frac{1}{10}$. The appendix contains the graph for the actual box length (A.1) and the one with box length modification by $\frac{1}{5}$(A.2) for the same system. In comparison of 6.3 and 6.4 one can see, that the graphs for different coarse grained grids of scattering vectors show deviations of the same order as the variance of different configurations. Combining different grid spacings is possible within the deviation of the results. Therefore this method provides a valid speedup for the static structure factor computation, at least as far as the accuracy of results is concerned.

Regarding usability, an internal modification of the box length is preferable, such that the user only chooses his or her desired range and resolution of scattering vectors and keeps the box length at its correct value. I suggest to introduce a new grid of scattering vectors. It consists of *layers* with fixed lattice constants. The innermost layer contains the smallest grid spacings. In the outward following layers the spacings become bigger and bigger in order to reduce the number of scattering vectors corresponding to one modulus. How many layers the new grid contains is left to the user's choice. Also the

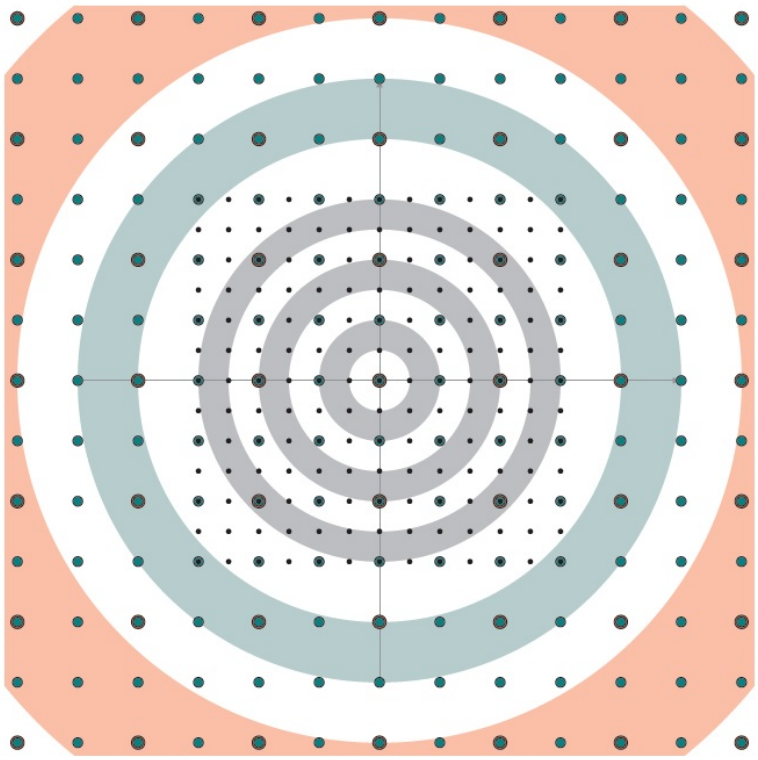

Figure 6.5.: Showing reducded number scattering vectors for greater q-moduli. One layer consists of three steps each. The picture shows two full layers and on the outside the beginning of the third (red, biggest dots).

layer size is user-adjustable. Such a grid, with layers of three steps each, is portrayed in figure 6.5 .

In the following, my ideas for an implementation are sketched. A grid structured in such a way is easy to built via a loop over layers with an internal loop over steps. Within these loops only the integer multipliers (`hx`, `hy`, `hz`) of the components of the scattering vectors are created making the creation fast and clear. Already at this stage the respective corner vectors can be skipped to prevent an overhead of scattering vectors which, again, would give rise for bad statistics (see figure 6.5 for corner vectors at the switchover to the next layer). Skipping the corners early, i.e. after the creation of the multipliers, but before the calculation of the actual modulus of the scattering vector, speeds up the computation. However, this is only possible in a cubic box, since only in that case the grid is isotropic.

As foreshadowed in chapter 3 (on the implementation of the static structure factor) a different choice for the parallelization improves the computation for certain cases. If the parallelizing over monomers is replaced by one over scattering vectors, this would

affect the computation of the single-chain structure factor.

The previous implementation restricts the distribution of monomers to tasks by the chain length if the single-chain structure factor is computed. As a result, monomers are not distributed evenly. Especially for polymers with a high polymerization index, this probably contributes to a major slow down. Scattering vectors can be distributed evenly for both the single-chain and the collective computation. The parallelization over **q**-vectors should be combined with the new grid. It can be realized inside of the grid creating double loop by using a counter for the even distribution of **q**-vectors to the available number of MPI tasks. All in all, the suggested new version of the static structure factor computation results in the code of figure 6.6. Where `computeS` is a function containing the scaling of the scattering vector with the lattice constants and the calculation of the static structure factor for the given scattering vector, including the loop over monomers. A slightly different version can be found in the appendix (A.2). It creates all scattering vectors within the loops avoiding the quad call of `computeS`. This makes the construction of an additional array, prior to the actual loop over scattering vectors, necessary. Furthermore a calculation to reobtain the layer number is needed making the code less readable than the version of figure 6.6.

```
0  //this is designed with a cubic symmetry (as far as interger multipliers for
       the scattering vectors are concerned)
1  int qcount = -1; //counts the q vectors that are used for the computation. is
       also used for parallelization

3  int num_layers = 10;  //number of layers. example value (specified by user in
       final code)
4  int num_steps = 3;    //number of steps per layer. example value (specified by
        user in final code)

6  int layer;  //the layer in which a gridpoint is positioned

8  int hx = 0;
9  int hy = 0;
10 int hz = 0;

12 int stepsize_x = 1; //starting value, grows exponentially with jump to next
       layer
13 int stepsize_y = 1; //starting value, grows exponentially with jump to next
       layer
14 int stepsize_z = 1; //starting value, grows exponentially with jump to next
       layer

16 // x - loop
17 for(int layer_x = 0; layer_x < num_layers; layer_x++){
18   for(int step_x = 0; step_x < num_steps; step_x++){
19     hx += stepsize_x;
20     layer = layer_x;
21     // y - loop
22     for(int layer_y = 0; layer_y < num_layers; layer_y++){
23       for(int step_y = 0; step_y < num_steps; step_y++){
24         hy += stepsize_y;
25         layer = max(layer_x, layer_y);
26         // z - loop
27         for(int layer_z = 0; layer_z < num_layers; layer_z++){
28           for(int step_z = 0; step_z < num_steps; step_z++){
29             hz += stepsize_z;
30             layer = max(layer, layer_z);
31             //skip overhead of q vectors on edges
32             int longestQ = num_steps * 2 ^ (layer + 1);
33             if(hx*hx + hy*hy + hz*hz > longestQ*longestQ)
34               break;
35             else{
36               qcount++;
37               //assign proc to current q-vector and call computeS
38               if(qcount%nprocs == myrank){
39                 computeS(hx,hy,hz);
40                 computeS(-hx,hy,hz);
41                 computeS(hx,-hy,hz);
42                 computeS(-hx,-hy,hz);
43               }
44             }
45           }//end of step_z loop
46           stepsize_z *= 2;
47         }//end of layer_z loop
48       }//end of step_z loop
49       stepsize_y *= 2;
50     }//end of layer_y loop
51   } //end of step_x loop
52   stepsize_x *= 2;
53 } //end of layer_x loop
```

Figure 6.6.: Static structure factor computation, which is parallelized over scattering vectors and uses less scattering vectors for larger q

7 Conclusion

A parallel implementation of the static structure factor $S(q)$ as well as the single-chain structure factor as an analysis routine for the simulation software ESPResSo++ is given. It was used as one criterion to determine equilibrium for entangled linear melts. Furthermore it served as a measure to show that *pre-packing* reduces local density fluctuations and is therefore suitable as a first stage of an equilibration procedure. Preparations for further improvements of the computation were made. At the same time, they can serve as the basis of an implementation for the dynamic structure factor $S(q,t)$, which takes time into account:

$$S(\mathbf{q}) = \frac{1}{N} \left[\sum_{i=1}^{N} \sum_{j=1}^{N} \mathrm{e}^{-i\mathbf{q}\cdot(\mathbf{r}_i(t)-\mathbf{r}_j(0))} \right] \tag{7.1}$$

Besides, the computation for the mean-square displacement was extended. The monomer displacement g_1 was already implemented in ESPResSo++ previous to this thesis. The displacement of monomers with respect to the center of mass of the chain was added as well as the displacement of the chain's center of mass g_3. Furthermore two new structures were added to the software, namely to maps within the base class that handles the distribution of particles to different tasks. They will make further implementation of those quantities clear and easy that involve the belonging of individual monomers to a certain chain. All in all, one basic class of ESPResSo++ was extended and the parallel computations of four quantities were added. Two of which contributed to a major challenge in computer simulations of long chains, their equilibration. The results of the computations indicated that equilibrium was achieved. Hence they served as evidence

for the success of the applied equilibration technique. The software, including the new routines, is able to simulate big systems. This gives reason for hope that computer simulations can provide quantitative forecasts for real experiments in the future. This would help towards a more specific processing or modification of polymeric substances, which is great, for polymers are widely and specifically used in different industries.

8

Acknowledgement

Last but not least I want to thank everyone who helped me accomplish this thesis. Starting with my Professor who provided me the opportunity to write a thesis such as I desired: one in which I could improve and practice my programming skills (without learning FORTRAN). Great thanks I owe to the whole group, above all to my academic supervisor, who had many good suggestions of what to try next and often encouraged me to ask questions. What is more, he always took his time to explain or discuss my matter and even answered questions beyond the scope of making my code work. My office mate helped me with many small things, such as Linux commands as well as some bigger issues, such as explanations and discussions of my ideas. She also provided her equilibrated systems, so I could test and she collaborated with me in the use of my code. The same applies to my other office mate, who also helped me with his scientific remarks. Miscellaneous questions were answered by various group members. Thanks for explaining the use of the boost library of C++ and discussing your code for g_1, for help with Mercurial, for general C++ help, for answering a cluster question, for advice on the structure factor and for finding a flawed kde-setting. Everyone else helped with pleasant coffee breaks, cakes, encouraging, inspiring or just fun chats in the K-bar. Moreover, I'd like to thank my friends, especially Matthias, Katharina, Christina, Miriam, Adeline and Vera for being there for me even during exhausting times of the thesis and my parents for ensuring their safe backup no matter what would happen. Needless to say that I would not have been able to write my thesis without all these people. Thank all of you very much!

A Appendix

A.1. Complete source codes

This section shows the source code for whole classes `StaticStructF` and `MeanSquareDispl`, precisely the respective files with extensions '.cpp' and '.hpp'. Also included are the corresponding Python files, which define the constructors and functions on Python level. These constructors and functions are called from the user's python script and connect them to the corresponding C++ implementation. All files printed in this section can be found in the ESPResSo++ directory `src/analysis`.

A.1.1. Static structure factor

Static structure factor - source file

```
1  /*
2    Copyright (C) 2012,2013
3        Max Planck Institute for Polymer Research
4    Copyright (C) 2008,2009,2010,2011
5        Max-Planck-Institute for Polymer Research & Fraunhofer SCAI

7    This file is part of ESPResSo++.

9    ESPResSo++ is free software: you can redistribute it and/or modify
10   it under the terms of the GNU General Public License as published by
11   the Free Software Foundation, either version 3 of the License, or
12   (at your option) any later version.
```

```
14    ESPResSo++ is distributed in the hope that it will be useful,
15    but WITHOUT ANY WARRANTY; without even the implied warranty of
16    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17    GNU General Public License for more details.

19    You should have received a copy of the GNU General Public License
20    along with this program.  If not, see <http://www.gnu.org/licenses/>.
21  */

23  #include "python.hpp"
24  #include "storage/DomainDecomposition.hpp"
25  #include "iterator/CellListIterator.hpp"
26  #include "Configuration.hpp"
27  #include "StaticStructF.hpp"
28  #include "esutil/Error.hpp"
29  #include "bc/BC.hpp"

31  #include <boost/serialization/map.hpp>

33  #include <math.h>       // cos and ceil and sqrt
34  #include <algorithm>    // std::min
35  #include <functional>   // std::plus
36  #include <time.h>       // time_t, for particle-distribution-to-cpu time

38  #ifndef M_PIl
39  #define M_PIl 3.14159265358979323846264338327950291
40  #endif

42  using namespace espresso;
43  using namespace espresso::iterator;
44  using namespace std;

46  namespace espresso {
47      namespace analysis {
48          // currently only works for particles numbered like 0, 1, 2,...

50          // nqx is a number which corresponds to the different x-values of the
51          // diffraction vector q. greater nqx produces more different x-values
52          // bin_factor determines the size for the binning of q-vectors in
                using
53          // dq = 2*PI/boxlength as a reference value such that
54          // bin_size = bin_factor * dq
55          // more in detail:
56          // dq is the shortest step of dqx, dqy, dqz - corresponding to the
57          // longest side of the box. dq = min(dqx, dqy, dqz)
58          // dqx, dqy, dqz are the cell length of the grid of possible q-vectors
59          // dqx = 2*PI/Lx, dqy = 2*PI/Ly, dqz = 2*PI/Lz

61          python::list StaticStructF::computeArray(int nqx, int nqy, int nqz,
62                  real bin_factor) {
63              time_t start;
64              time(&start);
65              cout << "collective calc starts " << ctime(&start) << "\n";
66              //fist the system coords are saved at each CPU
67              System& system = getSystemRef();
68              esutil::Error err(system.comm);
69              Real3D Li = system.bc->getBoxL(); //Box size (Lx, Ly, Lz)

71              int nprocs = system.comm->size(); // number of CPUs
72              int myrank = system.comm->rank(); // current CPU's number
```

```
74              if (myrank == 0) {
75                  cout << "collective calc starts " << ctime(&start) << "\n";
76              }

78              int num_part = 0;
79              ConfigurationPtr config = make_shared<Configuration > ();
80              // loop over all CPU-numbers - to give all CPUs all particle
                    coords
81              for (int rank_i = 0; rank_i < nprocs; rank_i++) {
82                  map< size_t, Real3D > conf;
83                  if (rank_i == myrank) {
84                      CellList realCells = system.storage->getRealCells();
85                      for (CellListIterator cit(realCells); !cit.isDone(); ++cit
                            ) {
86                          int id = cit->id();
87                          conf[id] = cit->position();
88                      }
89                  }
90                  boost::mpi::broadcast(*system.comm, conf, rank_i);

92                  // for simplicity we will number the particles from 0
93                  for (map<size_t, Real3D>::iterator itr = conf.begin(); itr !=
                        conf.end(); ++itr) {
94                      size_t id = itr->first;
95                      Real3D p = itr->second;
96                      config->set(id, p[0], p[1], p[2]);
97                      //config->set(num_part, p[0], p[1], p[2]);
98                      num_part++;
99                  }
100             }
101             if (myrank == 0) {
102                 time_t distributed;
103                 time(&distributed);
104                 cout << "particles on all CPUs " << ctime(&distributed) << "\n
                        ";
105                 cout << "distribution to CPUs took "
106                         << difftime(distributed, start) << " seconds \n";
107             }
108             // now all CPUs have all particle coords and num_part is the total
                     number
109             // of particles

111             // use all CPUs
112             // TODO it could be a problem if   n_nodes > num_part

114             // here starts calculation of the static structure factor

116             //step size for qx, qy, qz
117             real dqs[3];
118             dqs[0] = 2. * M_PIl / Li[0];
119             dqs[1] = 2. * M_PIl / Li[1];
120             dqs[2] = 2. * M_PIl / Li[2];

122             Real3D q;

124             //calculations for binning
125             real maxX = nqx * dqs[0]; //maximum x value of a q vector
126             real maxY = nqy * dqs[1]; //maximum y value of a q vector
127             real maxZ = nqz * dqs[2]; //maximum z value of a q vector
```

```
129             real shortestDir = min(maxX, min(maxY,maxZ));  //#include<
                    algorithm>??
130             real bin_size = bin_factor * min(dqs[0], (dqs[1], dqs[2]));


133 //            real q_sqr_max = nqx * nqx * dqs[0] * dqs[0]
134 //                    + nqy * nqy * dqs[1] * dqs[1]
135 //                    + nqz * nqz * dqs[2] * dqs[2];
136 //            real q_max = sqrt(q_sqr_max);
137             int num_bins = (int) ceil(shortestDir / bin_size);
138             vector<real> sq_bin;
139             vector<real> q_bin;
140             vector<int> count_bin;
141             sq_bin.resize(num_bins);
142             q_bin.resize(num_bins);
143             count_bin.resize(num_bins);

145             if (myrank == 0) {
146                 cout << nprocs << " CPUs, new routine\n\n"
147                         << "bin size \t" << bin_size << "\n"
148                         << "q_max    \t" << shortestDir << "\n";
149             }

151             real n_reci = 1. / num_part;
152             real scos_local = 0; //will store cos-sum on each CPU
153             real ssin_local = 0; //will store sin-sum on each CPU
154             int ppp = (int) ceil((double) num_part / nprocs); //particles per
                    proc

156             Real3D coordP;

158             python::list pyli;

160             //loop over different q values
161             //starting from zero because combinations with negative components
162             //will give the same result in S(q). so S(q) is the same for
163             //the 8 vectors q=(x,y,z),(-x,y,z), (x,-y,z),(x,y,-z),(-x,-y,z)
                    ,...
164             for (int hx = -nqx; hx <= nqx; hx++) {
165                 for (int hy = -nqy; hy <= nqy; hy++) {
166                     for (int hz = 0; hz <= nqz; hz++) {

168                         //values of q-vector
169                         q[0] = hx * dqs[0];
170                         q[1] = hy * dqs[1];
171                         q[2] = hz * dqs[2];
172                         real q_abs = q.abs();
173                         if (q_abs > shortestDir){break;}

175                         //determining the bin number
176                         int bin_i = (int) floor(q_abs / bin_size);
177                         q_bin[bin_i] += q_abs;
178                         count_bin[bin_i] += 1;

180                         //resetting the variables that store the local sum on
                                each proc
181                         scos_local = 0;
182                         ssin_local = 0;

184                         //loop over particles
```

```
185                         for (int k = myrank * ppp; k < (1 + myrank) * ppp && k
                                 < num_part;
186                                 k++) {
187                             coordP = config->getCoordinates(k);
188                             scos_local += cos(q * coordP);
189                             ssin_local += sin(q * coordP);
190                         }
191                         if (myrank != 0) {
192                             boost::mpi::reduce(*system.comm, scos_local, plus<
                                    real > (), 0);
193                             boost::mpi::reduce(*system.comm, ssin_local, plus<
                                    real > (), 0);
194                         }

196                         if (myrank == 0) {
197                             real scos = 0;
198                             real ssin = 0;
199                             boost::mpi::reduce(*system.comm, scos_local, scos,
                                     plus<real > (), 0);
200                             boost::mpi::reduce(*system.comm, ssin_local, ssin,
                                     plus<real > (), 0);
201                             sq_bin[bin_i] += scos * scos + ssin * ssin;
202                         }
203                     }
204                 }
205             }
206             //creates the python list with the results
207             if (myrank == 0) {
208                 //starting with bin_i = 1 will leave out the value for q=0,
                        otherwise start with bin_i=0
209                 for (int bin_i = 1; bin_i < num_bins; bin_i++) {
210                     real c = (count_bin[bin_i]) ? 1 / (real) count_bin[bin_i]
                            : 0;
211                     sq_bin[bin_i] = n_reci * sq_bin[bin_i] * c;
212                     q_bin[bin_i] = q_bin[bin_i] * c;

214                     python::tuple q_Sq_pair;
215                     q_Sq_pair = python::make_tuple(q_bin[bin_i], sq_bin[bin_i
                            ]);
216                     pyli.append(q_Sq_pair);
217                 }
218             }
219             return pyli;
220         }

222         // this routine is for ordered configurations, e.g. particle 0 to 9
223         // belong to chain 1, particle 10 to 19 to chain 2 etc.

225         python::list StaticStructF::computeArraySingleChain(int nqx, int nqy,
                int nqz,
226                     real bin_factor, int chainlength) {
227             //fist the system coords are saved at each CPU
228             System& system = getSystemRef();
229             esutil::Error err(system.comm);
230             Real3D Li = system.bc->getBoxL(); //Box size (Lx, Ly, Lz)

232             int nprocs = system.comm->size(); // number of CPUs
233             int myrank = system.comm->rank(); // current CPU's number

235             int num_part = 0;
236             ConfigurationPtr config = make_shared<Configuration > ();
```

```
237             // loop over all CPU-numbers - to give all CPUs all particle
                    coords
238             for (int rank_i = 0; rank_i < nprocs; rank_i++) {
239                 map< size_t, Real3D > conf;
240                 if (rank_i == myrank) {
241                     CellList realCells = system.storage->getRealCells();
242                     for (CellListIterator cit(realCells); !cit.isDone(); ++cit
                            ) {
243                         int id = cit->id();
244                         conf[id] = cit->position();
245                     }
246                 }
247                 boost::mpi::broadcast(*system.comm, conf, rank_i);

249                 // for simplicity we will number the particles from 0
250                 for (map<size_t, Real3D>::iterator itr = conf.begin(); itr !=
                        conf.end(); ++itr) {
251                     size_t id = itr->first;
252                     Real3D p = itr->second;
253                     config->set(id, p[0], p[1], p[2]);
254                     //config->set(num_part, p[0], p[1], p[2]);
255                     num_part++;
256                 }
257             }
258             cout << "particles are given to each CPU!\n";
259             // now all CPUs have all particle coords and num_part is the total
                     number
260             // of particles

262             // use all CPUs
263             // TODO it could be a problem if   n_nodes > num_part

265             // here starts calculation of the static structure factor

267             //step size for qx, qy, qz
268             real dqs[3];
269             dqs[0] = 2. * M_PI1 / Li[0];
270             dqs[1] = 2. * M_PI1 / Li[1];
271             dqs[2] = 2. * M_PI1 / Li[2];

273             Real3D q;

275             //calculations for binning
276             real bin_size = bin_factor * min(dqs[0], (dqs[1], dqs[2]));
277             real q_sqr_max = nqx * nqx * dqs[0] * dqs[0]
278                     + nqy * nqy * dqs[1] * dqs[1]
279                     + nqz * nqz * dqs[2] * dqs[2];
280             real q_max = sqrt(q_sqr_max);
281             int num_bins = (int) ceil(q_max / bin_size);
282             vector<real> sq_bin;
283             vector<real> q_bin;
284             vector<int> count_bin;
285             sq_bin.resize(num_bins);
286             q_bin.resize(num_bins);
287             count_bin.resize(num_bins);

289             if (myrank == 0) {
290                 cout << nprocs << " CPUs\n\n"
291                         << "bin size \t" << bin_size << "\n"
292                         << "q_max    \t" << q_max << "\n";
293             }
```

```
295             real n_reci = 1. / num_part;
296             real chainlength_reci = 1. / chainlength;
297             real scos_local = 0; //will store cos-sum on each CPU
298             real ssin_local = 0; //will store sin-sum on each CPU
299             //will store the summation of the the single chain structure
                    factor
300             real singleChain_localSum = 0;
301             Real3D coordP;
302             python::list pyli;

304             //calculations for parallelizing (over chains)
305             int num_chains;
306             if (num_part % chainlength == 0)
307                 num_chains = num_part / chainlength;
308             else {
309                 cout << "ERROR: chainlenght does not match total number of "
310                         << "particles. num_part % chainlenght is unequal 0. \n
                            "
311                         << "Calculation of SingleChain_StaticStructF aborted\n
                            ";
312                 return pyli;
313             }
314             int cpp = (int) ceil((double) num_chains / nprocs); //chains per
                    proc
315             cout << "chains per proc\t" << cpp << "\n";


318             //loop over different q values
319             //starting from zero because combinations with negative components
320             //will give the same result in S(q). so S(q) is the same for
321             //the 8 vectors q=(x,y,z),(-x,y,z), (x,-y,z),(x,y,-z),(-x,-y,z)
                    ,...
322             for (int hx = -nqx; hx <= nqx; hx++) {
323                 for (int hy = -nqy; hy <= nqy; hy++) {
324                     for (int hz = 0; hz <= nqz; hz++) {

326                         //values of q-vector
327                         q[0] = hx * dqs[0];
328                         q[1] = hy * dqs[1];
329                         q[2] = hz * dqs[2];
330                         real q_abs = q.abs();

332                         //determining the bin number
333                         int bin_i = (int) floor(q_abs / bin_size);
334                         q_bin[bin_i] += q_abs;
335                         count_bin[bin_i] += 1;

337                         //resetting the variable that stores the sum for each
                                q-vector
338                         singleChain_localSum = 0;

340                         //loop over chains (cid is chain_id)
341                         for (int cid = myrank * cpp; cid < (1 + myrank) * cpp
342                                 && cid < num_chains; cid++) {
343                             scos_local = 0; //resetting the cos sum for the
                                    each chain
344                             ssin_local = 0; //resetting the sin sum for the
                                    each chain
345                             //loop over particles
```

```
346                             for (int k = cid * chainlength; k < (1 + cid) *
                                    chainlength && k < num_part;
347                                     k++) {
348                                 coordP = config->getCoordinates(k);
349                                 scos_local += cos(q * coordP);
350                                 ssin_local += sin(q * coordP);
351                             }
352                             //the (summation part of the) single chain
                                    structure
353                             // factors are summed up for the averaging at the
354                             // end (over the chains)
355                             singleChain_localSum += scos_local * scos_local
356                                     + ssin_local * ssin_local;
357                         }


360                         if (myrank != 0) {
361                             boost::mpi::reduce(*system.comm,
                                    singleChain_localSum, plus<real > (), 0);
362                         }

364                         if (myrank == 0) {
365                             real singleChainSum = 0;
366                             boost::mpi::reduce(*system.comm,
                                    singleChain_localSum, singleChainSum, plus<
                                    real > (), 0);
367                             sq_bin[bin_i] += singleChainSum;
368                         }
369                     }
370                 }
371             }
372             //creates the python list with the results
373             if (myrank == 0) {
374                 //starting with bin_i = 1 will leave out the value for q=0,
                        otherwise start with bin_i=0
375                 for (int bin_i = 1; bin_i < num_bins; bin_i++) {
376                     real c = (count_bin[bin_i]) ? 1 / (real) count_bin[bin_i]
                            : 0;
377                     sq_bin[bin_i] = n_reci * chainlength_reci * sq_bin[bin_i]
                            * c;
378                     q_bin[bin_i] = q_bin[bin_i] * c;

380                     python::tuple q_Sq_pair;
381                     q_Sq_pair = python::make_tuple(q_bin[bin_i], sq_bin[bin_i
                            ]);
382                     pyli.append(q_Sq_pair);
383                 }
384             }
385             return pyli;
386         }

388         // TODO: this dummy routine is still needed as we have not yet
                ObservableVector
389         // there has to be a function 'compute' because of the used template
390         // otherwise a compiling error will occur

392         real StaticStructF::compute() const {
393             return -1.0;
394         }

396         void StaticStructF::registerPython() {
```

```
397             using namespace espresso::python;
398             class_<StaticStructF, bases< Observable > >
399                     ("analysis_StaticStructF", init< shared_ptr< System > >())
400                     .def("compute", &StaticStructF::computeArray)
401                     .def("computeSingleChain", &StaticStructF::
                            computeArraySingleChain)
402                     ;
403         }
404     }
405 }
```

Static structure factor - header file

```
1  /*
2    Copyright (C) 2012,2013
3        Max Planck Institute for Polymer Research
4    Copyright (C) 2008,2009,2010,2011
5        Max-Planck-Institute for Polymer Research & Fraunhofer SCAI

7    This file is part of ESPResSo++.

9    ESPResSo++ is free software: you can redistribute it and/or modify
10   it under the terms of the GNU General Public License as published by
11   the Free Software Foundation, either version 3 of the License, or
12   (at your option) any later version.

14   ESPResSo++ is distributed in the hope that it will be useful,
15   but WITHOUT ANY WARRANTY; without even the implied warranty of
16   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17   GNU General Public License for more details.

19   You should have received a copy of the GNU General Public License
20   along with this program.  If not, see <http://www.gnu.org/licenses/>.
21 */

23 // ESPP_CLASS
24 #ifndef _ANALYSIS_STATICSTRUCTF_HPP
25 #define _ANALYSIS_STATICSTRUCTF_HPP

27 #include "types.hpp"
28 #include "Observable.hpp"
29 #include "python.hpp"

31 namespace espresso {
32     namespace analysis {

34         /** Class to compute the static structure function of the system. */
35         class StaticStructF : public Observable {
36         public:

38             StaticStructF(shared_ptr< System > system) : Observable(system) {
39             }

41             ~StaticStructF() {
42             }
43             virtual real compute() const;
44             virtual python::list computeArray(int nqx, int nqy, int nqz,
45                     real bin_factor) const;
46             virtual python::list computeArraySingleChain(int nqx, int nqy, int
                    nqz,
```

```
                      real bin_factor, int chainlength) const;
            static void registerPython();

        };
    }
}


#endif
```

Static structure factor - python file

```
#  Copyright (C) 2012,2013
#      Max Planck Institute for Polymer Research
#  Copyright (C) 2008,2009,2010,2011
#      Max-Planck-Institute for Polymer Research & Fraunhofer SCAI
#
#  This file is part of ESPResSo++.
#
#  ESPResSo++ is free software: you can redistribute it and/or modify
#  it under the terms of the GNU General Public License as published by
#  the Free Software Foundation, either version 3 of the License, or
#  (at your option) any later version.
#
#  ESPResSo++ is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#  GNU General Public License for more details.
#
#  You should have received a copy of the GNU General Public License
#  along with this program.  If not, see <http://www.gnu.org/licenses/>.


"""
**********************************
**espresso.analysis.StaticStructF**
**********************************

"""
from espresso.esutil import cxxinit
from espresso import pmi

from espresso.analysis.Observable import *
from _espresso import analysis_StaticStructF

class StaticStructFLocal(ObservableLocal, analysis_StaticStructF):
  'The (local) compute the static structure function.'
  def __init__(self, system):
    cxxinit(self, analysis_StaticStructF, system)

  def compute(self, nqx, nqy, nqz, bin_factor, ofile = None):
    if ofile is None:
      return self.cxxclass.compute(self, nqx, nqy, nqz, bin_factor)
    else:
      #run compute on each CPU
      result = self.cxxclass.compute(self, nqx, nqy, nqz, bin_factor)
      #create the outfile only on CPU 0
      if pmi.isController:
        myofile = 'qsq_' + str(ofile) + '.txt'
        outfile = open (myofile, 'w')
```

```
49          for i in range (len(result)):
50            line = str(result[i][0]) + "\t" + str(result[i][1]) + "\n"
51            outfile.write(line)
52          outfile.close()
53        return result

55    def computeSingleChain(self, nqx, nqy, nqz, bin_factor, chainlength, ofile =
          None):
56      if ofile is None:
57        return self.cxxclass.computeSingleChain(self, nqx, nqy, nqz, bin_factor,
              chainlength)
58      else:
59        #run computeSingleChain on each CPU
60        result = self.cxxclass.computeSingleChain(self, nqx, nqy, nqz,
              bin_factor, chainlength)
61        print result #this line is in case the outfile causes problems
62        #create the outfile only on CPU 0
63        if pmi.isController:
64          myofile = 'qsq_singleChain' + str(ofile) + '.txt'
65          outfile = open (myofile, 'w')
66          for i in range (len(result)):
67            line = str(result[i][0]) + "\t" + str(result[i][1]) + "\n"
68            outfile.write(line)
69          outfile.close()
70        return result

72  if pmi.isController:
73    class StaticStructF(Observable):
74      __metaclass__ = pmi.Proxy
75      pmiproxydefs = dict(
76        pmicall = [ "compute", "computeSingleChain" ],
77        cls = 'espresso.analysis.StaticStructFLocal'
78      )
```

A.1.2. Particle decomposition

The member functions `gather()` and `gatherFromFile()`, which are necessary to collect the information from different snapshots or at different simulation times, are implemented in the source file. The constructor is directly implemented in the header file.

Particle decomposition - source file

```
1   /*
2     Copyright (C) 2012,2013
3         Max Planck Institute for Polymer Research
4     Copyright (C) 2008,2009,2010,2011
5         Max-Planck-Institute for Polymer Research & Fraunhofer SCAI

7     This file is part of ESPResSo++.

9     ESPResSo++ is free software: you can redistribute it and/or modify
10    it under the terms of the GNU General Public License as published by
11    the Free Software Foundation, either version 3 of the License, or
12    (at your option) any later version.
```

```
14    ESPResSo++ is distributed in the hope that it will be useful,
15    but WITHOUT ANY WARRANTY; without even the implied warranty of
16    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17    GNU General Public License for more details.

19    You should have received a copy of the GNU General Public License
20    along with this program.  If not, see <http://www.gnu.org/licenses/>.
21  */

23  #include <iostream>
24  #include <fstream>
25  #include <sstream>
26  #include <string>
27  #include <vector>

29  #include "ConfigsParticleDecomp.hpp"
30  #include "bc/BC.hpp"
31  #include <boost/serialization/map.hpp>

33  using namespace std;
34  using namespace espresso;

36  namespace espresso {
37    namespace analysis {

39      using namespace iterator;

41      int ConfigsParticleDecomp::getListSize() const{
42        return configurations.size();
43      }

45      ConfigurationList ConfigsParticleDecomp::all() const{
46        return configurations;
47      }

49      ConfigurationPtr ConfigsParticleDecomp::getConf(int position) const{
50        int nconfigs = configurations.size();
51        if (0 <= position and position < nconfigs) {
52          return configurations[position];
53        }
54        else{
55          System& system = getSystemRef();
56          esutil::Error err(system.comm);
57          stringstream msg;
58          msg << "Error. Velocities::get <out-of-range>" << endl;
59          err.setException( msg.str() );
60          return shared_ptr<Configuration>();
61        }
62      }

64      void ConfigsParticleDecomp::pushConfig(ConfigurationPtr config){
65        configurations.push_back(config);
66      }

68      void ConfigsParticleDecomp::gather() {
69        System& system = getSystemRef();
70        esutil::Error err(system.comm);

72        int nprocs = system.comm->size();
73        int myrank = system.comm->rank();
```

```
75        int localN = system.storage->getNRealParticles();

77        int curNumP = 0;
78        boost::mpi::all_reduce(*system.comm, localN, curNumP, std::plus<int>());
79        if(myrank==0){
80          // check whether the number of particles is the same during the
              gathering
81          if( curNumP != num_of_part ){
82            stringstream msg;
83            msg<<"   ConfigsParticleDecomp gathers the configurations of the
                  same system\n"
84                 " with the same number of particles. If you need to store the
                      systems\n"
85                 " with different number of particles you should use something
                      else."
86                 " E.g `Configurations`";
87            err.setException( msg.str() );
88          }
89        }

91        ConfigurationPtr config = make_shared<Configuration> ();
92        for (int rank_i=0; rank_i<nprocs; rank_i++) {
93          map< size_t, Real3D > conf;
94          if (rank_i == myrank) {
95            CellList realCells = system.storage->getRealCells();
96            for(CellListIterator cit(realCells); !cit.isDone(); ++cit) {
97              int id = cit->id();
98              Real3D property = Real3D(0,0,0);
99              if(key=="position")
100               property = cit->position();
101             else if(key=="velocity")
102               property = cit->velocity();
103             else if(key=="unfolded"){
104               Real3D& pos = cit->position();
105               Int3D& img = cit->image();
106               Real3D Li = system.bc->getBoxL();
107               for (int i = 0; i < 3; ++i) property[i] = pos[i] + img[i] * Li[i
                      ];
108             }
109             else{
110               stringstream msg;
111               msg<<"Error. Key "<<key<<" is unknown. Use position, unfolded or
                      "
112                    " velocity.";
113               err.setException( msg.str() );
114             }

116             conf[id] = property;
117           }
118       }

120         boost::mpi::broadcast(*system.comm, conf, rank_i);

122         for (map<size_t,Real3D>::iterator itr=conf.begin(); itr != conf.end();
                ++itr) {
123           size_t id = itr->first;
124           Real3D p = itr->second;
125           if(idToCpu[id]==myrank) config->set(id, p[0], p[1], p[2]);
126         }
127       }
```

```
129         pushConfig(config);
130       }

132       void ConfigsParticleDecomp::gatherFromFile(string filename) {
133         System& system = getSystemRef();
134         esutil::Error err(system.comm);

136         int nprocs = system.comm->size();
137         int myrank = system.comm->rank();

139         int localN = system.storage->getNRealParticles();

141         ConfigurationPtr config = make_shared<Configuration> ();
142         map< size_t, Real3D > conf;

144         if (myrank==0) {
145             int id, type;
146             real xpos, ypos, zpos;
147             string line;
148             ifstream file(filename.c_str());
149             if (file.is_open()) {
150             // skip first 2 lines
151               getline(file, line);
152             getline(file, line);
153             int count = 0;
154               while (getline(file, line)) {
155                 stringstream sl(line);
156                 sl >> id;
157                 sl >> type;
158                 sl >> xpos;
159                 sl >> ypos;
160                 sl >> zpos;
161                 // cout << id << ":" << x << "," << y << "," << z << endl;
162                 conf[id] = Real3D(xpos, ypos, zpos);
163                 count++;
164               }
165               file.close();
166               cout << "read " << count << " particles from file " << filename <<
                       endl;
167               if (count != num_of_part) {
168                   stringstream msg;
169                   msg << "Number of read particles does not match the number of
                          particles of the system (which is " << num_of_part << ")";
170                   err.setException( msg.str() );
171               }
172             } else {
173                 stringstream msg;
174                 msg << "Unable to open file " << filename;
175                 err.setException( msg.str() );
176             }
177         }

179         boost::mpi::broadcast(*system.comm, conf, 0);

181         for (map<size_t,Real3D>::iterator itr=conf.begin(); itr != conf.end();
              ++itr) {
182           size_t id = itr->first;
183           Real3D p = itr->second;
184           if(idToCpu[id]==myrank) config->set(id, p[0], p[1], p[2]);
185         }
186         pushConfig(config);
```

```
187       }

189       // Python wrapping
190       void ConfigsParticleDecomp::registerPython() {
191         using namespace espresso::python;

193         class_<ConfigsParticleDecomp, boost::noncopyable >(
194           "analysis_ConfigsParticleDecomp", no_init
195           //init< shared_ptr< System > >()
196         )
197         .def_readonly("size", &ConfigsParticleDecomp::getListSize)

199         .def("gather", &ConfigsParticleDecomp::gather)
200         .def("gatherFromFile", &ConfigsParticleDecomp::gatherFromFile)
201         .def("__getitem__", &ConfigsParticleDecomp::getConf)
202         .def("all", &ConfigsParticleDecomp::all)
203         .def("clear", &ConfigsParticleDecomp::clear)
204         .def("compute", &ConfigsParticleDecomp::compute)
205         ;
206       }
207     }
208   }
```

Particle decomposition - header file

Here the constructors are implemented, which contain the filling of the map `idToCpu` using particle or chain decomposition.

```
1   /*
2     Copyright (C) 2012,2013
3         Max Planck Institute for Polymer Research
4     Copyright (C) 2008,2009,2010,2011
5         Max-Planck-Institute for Polymer Research & Fraunhofer SCAI

7     This file is part of ESPResSo++.

9     ESPResSo++ is free software: you can redistribute it and/or modify
10    it under the terms of the GNU General Public License as published by
11    the Free Software Foundation, either version 3 of the License, or
12    (at your option) any later version.

14    ESPResSo++ is distributed in the hope that it will be useful,
15    but WITHOUT ANY WARRANTY; without even the implied warranty of
16    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17    GNU General Public License for more details.

19    You should have received a copy of the GNU General Public License
20    along with this program.  If not, see <http://www.gnu.org/licenses/>.
21  */

23  // ESPP_CLASS
24  #ifndef _ANALYSIS_CONFIGSPARTICLEDECOMP_HPP
25  #define _ANALYSIS_CONFIGSPARTICLEDECOMP_HPP

27  #include "python.hpp"
28  #include "mpi.h"
29  #include "types.hpp"
30  #include "SystemAccess.hpp"
31  #include "Configuration.hpp"
```

```
33 #include "storage/Storage.hpp"
34 #include "iterator/CellListIterator.hpp"
35 #include "esutil/Error.hpp"

37 #include <string>

39 using namespace std;

41 namespace espresso {
42   namespace analysis {

44     using namespace iterator;
45     /*
46      * Class that stores particle !!properties (velocities at the moment)!!
          for later
47      * analysis. It uses object Configuration to store data.
48      *
49      * Here the concept of particle decomposition is used, i.e. each processor
          stores
50      * relevant number of particles. It's useless to get the data on python
          level from
51      * here. Therefore it is abstract class. A derived class should realize
          the function
52      * 'compute'.
53      *
54      * Important: Mainly it was created in order to observe the system in time
          .
55      * !!At the moment the number of particles should be the same for
          different snapshots.!!
56      * Otherwise it will throw a runtime error exception
57     */

59     typedef vector<ConfigurationPtr> ConfigurationList;

61     class ConfigsParticleDecomp : public SystemAccess {

63     public:
64       /*
65        * Constructor, allow for unlimited snapshots. It defines how many
            particles
66        * correspond to different cpu.
67        */
68       ConfigsParticleDecomp(shared_ptr<System> system): SystemAccess (system){
69         // by default key = "position", it will store the particle positions
70         // (option: "velocity" or "unfolded")
71         esutil::Error err(system->comm);

73         key = "position";

75         int localN = system -> storage -> getNRealParticles();
76         boost::mpi::all_reduce(*system->comm, localN, num_of_part, std::plus<
               int>());

78         int n_nodes = system -> comm -> size();
79         int this_node = system -> comm -> rank();

81         int local_num_of_part = num_of_part / n_nodes + 1;

83         vector<int> tot_idList;
84         for(int rank_i = 0; rank_i<n_nodes;rank_i++){
```

```
86            int numLocPart = 0;
87            if(rank_i==this_node){
88              numLocPart = system -> storage -> getNRealParticles();
89            }
90            boost::mpi::broadcast(*system->comm, numLocPart, rank_i);

92            int* idList = new int[numLocPart];

94            if(rank_i==this_node){

96              int count = 0;
97              CellList realCells = system -> storage -> getRealCells();
98              for(CellListIterator cit(realCells); !cit.isDone(); ++cit) {
99                int id = cit->id();

101               idList[count] = id;
102               count++;
103             }
104           }

106           boost::mpi::broadcast(*system->comm, idList, numLocPart, rank_i);

108           for(int i=0; i<numLocPart;i++){
109             tot_idList.push_back( idList[i] );
110           }

112           delete [] idList;
113           idList = NULL;
114         }

116         int nodeNum = 0;
117         int count = 0;
118         for (vector<int>::iterator it = tot_idList.begin(); it!=tot_idList.end
              (); ++it) {
119           idToCpu[*it] = nodeNum;
120           count ++;
121           if(count>=local_num_of_part){
122             count = 0;
123             nodeNum++;
124           }
125         }

127         try{
128           if(num_of_part <= n_nodes){
129             stringstream msg;
130             msg<<"Warning. Number of particles less then the number of nodes.\
                  n";
131             msg<<"It might be a problem. NPart="<<num_of_part<<" NNodes="<<
                  n_nodes;
132             err.setException( msg.str() );
133             err.checkException();
134           }
135         }
136         catch(std::exception const& e){
137           if(this_node==0)
138             cout << "Exception: " << e.what() << "\n";
139         }
140       }

142        /*
```

```
143         * Constructor, allow for unlimited snapshots. It defines how many
                particles
144         * correspond to different cpu without breaking chains. So the monomers
                of
145         * one chain correspond to one CPU only.
146          *
147          * !! currently only works for particles numbered like 0, 1, 2,... !!
148          * !! with each chain consisting particles with subsequent ids     !!
149         */
150        ConfigsParticleDecomp(shared_ptr<System> system, int _chainlength):
               SystemAccess (system){
151          // by default key = "position", it will store the particle positions
152          // (option: "velocity" or "unfolded")
153          esutil::Error err(system->comm);

155          key = "position";
156          chainlength = _chainlength;

158          int localN = system -> storage -> getNRealParticles();
159          boost::mpi::all_reduce(*system->comm, localN, num_of_part, std::plus<
                 int>());

161          int n_nodes = system -> comm -> size();
162          int this_node = system -> comm -> rank();

164          //for monodisperse chains
165          int num_chains = num_of_part / chainlength;
166          int local_num_chains = (int) ceil( (double)num_chains / n_nodes );
167          int local_num_part = local_num_chains * chainlength;

169          //in case the chainlength does not match the total number of particles
170          if(num_of_part % chainlength != 0){
171              cout << "chainlength does not match total number of particles\n"
172                      << "chainlength: " << chainlength
173                      << "\n num_of_part " << num_of_part << "\n\n";
174          }

176          //CPU0 will use particles 0, 1, 2, ... local_num_particles-1.
177          //CPU1 will use particles local_num_particles, local_num_particles
                 +1,...
178          int nodeNum = -1;
179          for(long unsigned int id = 0; id < num_of_part ;id++){
180              if(id % local_num_part == 0) ++nodeNum;
181              idToCpu[id] = nodeNum;
182          }
183          //output if the assignment failed
184          if (nodeNum >= n_nodes) {
185              if(this_node == 0){
186                  cout << "assignment went wrong. Particles were assigned to
                         proc "
187                          << nodeNum << "\n";
188                  cout << "highest process number should be " << n_nodes - 1 <<"
                         \n";
189                  cout << "check if total number of particles matches with
                         chainlength\n";
190              }
191          }
192           //output for testing
193          if(this_node == 0){
194              for(map<size_t, int>::iterator itr=idToCpu.begin(); itr != idToCpu
                     .end(); itr++){
```

```
195             size_t key = itr -> first;
196             int mapped = itr -> second;
197             //cout << key << "\t" << mapped << "\n";
198         }
199     }

201         try{
202           if(num_chains < n_nodes){
203             stringstream msg;
204             msg<<"Warning. Number of chains less then the number of nodes.\n";
205             msg<<"It might be a problem. NChains="<<num_chains<<" NNodes="<<
                    n_nodes;
206             err.setException( msg.str() );
207             err.checkException();
208           }
209         }
210         catch(std::exception const& e){
211           if(this_node==0)
212             cout << "Exception: " << e.what() << "\n";
213         }
214       }
215       ~ConfigsParticleDecomp() {}


218       // get number of available snapshots. Returns the size of
              Configurationlist
219       int getListSize() const;

221       // Take a snapshot of property (all current particle velocities at the
              moment)
222       void gather();

224       // Read in a snapshot from a xyz-file
225       void gatherFromFile(string filename);

227       // Get a configuration from ConfigurationList
228       ConfigurationPtr getConf(int position) const;

230       // it returns all the configurations
231       ConfigurationList all() const;

233       // it erases all the configurations from ConfigurationList
234       void clear(){
235         configurations.clear();
236       }

238       virtual python::list compute() const = 0;

240       static void registerPython();

242     protected:

244       static LOG4ESPP_DECL_LOGGER(logger);

246       // all cpus handle defined number of particles
247       int num_of_part;
248       int chainlength; //for calculations with chains (instead of monomers)
249       map< size_t, int > idToCpu; // binds cpu and particle id

251       string key; // it can be "position", "velocity" or "unfolded"
```

```
253       private:

255         void pushConfig(ConfigurationPtr config);

257         // the list of snapshots
258         ConfigurationList configurations;
259       };
260     }
261   }

263   #endif
```

Particle decomposition - python file

```
1   #  Copyright (C) 2012,2013
2   #      Max Planck Institute for Polymer Research
3   #  Copyright (C) 2008,2009,2010,2011
4   #      Max-Planck-Institute for Polymer Research & Fraunhofer SCAI
5   #
6   #  This file is part of ESPResSo++.
7   #
8   #  ESPResSo++ is free software: you can redistribute it and/or modify
9   #  it under the terms of the GNU General Public License as published by
10  #  the Free Software Foundation, either version 3 of the License, or
11  #  (at your option) any later version.
12  #
13  #  ESPResSo++ is distributed in the hope that it will be useful,
14  #  but WITHOUT ANY WARRANTY; without even the implied warranty of
15  #  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16  #  GNU General Public License for more details.
17  #
18  #  You should have received a copy of the GNU General Public License
19  #  along with this program.  If not, see <http://www.gnu.org/licenses/>.


22  """
23  ********************************************
24  **espresso.analysis.ConfigsParticleDecomp**
25  ********************************************

27  """
28  #from espresso.esutil import cxxinit
29  from espresso import pmi

31  from _espresso import analysis_ConfigsParticleDecomp

33  class ConfigsParticleDecompLocal(analysis_ConfigsParticleDecomp):
34      'The (local) storage of configurations.'
35      def __init__(self, system):
36        cxxinit(self, analysis_ConfigsParticleDecomp, system)
37      def gather(self):
38        return self.cxxclass.gather(self)
39      def gatherFromFile(self, filename):
40        return self.cxxclass.gatherFromFile(self, filename)
41      def clear(self):
42        return self.cxxclass.clear(self)
43      def __iter__(self):
44        return self.cxxclass.all(self).__iter__()
```

```
46      def compute(self):
47        return self.cxxclass.compute(self)

49  if pmi.isController:
50    class ConfigsParticleDecomp(object):
51      """Abstract base class for parallel analysis based on particle
            decomposition."""
52      __metaclass__ = pmi.Proxy
53      pmiproxydefs = dict(
54        #cls =  'espresso.analysis.ConfigsParticleDecompLocal',
55        pmicall = [ "gather", "gatherFromFile", "clear", "compute" ],
56        localcall = ["__getitem__", "all"],
57        pmiproperty = ["size"]
58      )
```

A.1.3. Mean squared displacement

Mean squared displacement - source file

```
1   /*
2     Copyright (C) 2012,2013
3         Max Planck Institute for Polymer Research
4     Copyright (C) 2008,2009,2010,2011
5         Max-Planck-Institute for Polymer Research & Fraunhofer SCAI

7     This file is part of ESPResSo++.

9     ESPResSo++ is free software: you can redistribute it and/or modify
10    it under the terms of the GNU General Public License as published by
11    the Free Software Foundation, either version 3 of the License, or
12    (at your option) any later version.

14    ESPResSo++ is distributed in the hope that it will be useful,
15    but WITHOUT ANY WARRANTY; without even the implied warranty of
16    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17    GNU General Public License for more details.

19    You should have received a copy of the GNU General Public License
20    along with this program.  If not, see <http://www.gnu.org/licenses/>.
21  */

23  #include "MeanSquareDispl.hpp"
24  //#include <algorithm> //for std::sort
25  using namespace std;
26  //using namespace espresso;

28  namespace espresso {
29    namespace analysis {

31      //using namespace iterator;

33      /*
34       * calculates the mean square displacement of the particles/monomers in
             the COM of the whole system
35       *
36       * calc <r^2> the output is the average mean sq. displacement over 3
             directions.
```

```
37        * !! Important!! For D calculation factor 1/6 is already taken into
              account.
38        * !! all confs should contain the same number of particles
39       */

41       python::list MeanSquareDispl::compute() const{

43         int M = getListSize(); //number of snapshots/configurations
44         real* totZ; //will store the mean squared displacement
45         totZ = new real[M];
46         real* Z;
47         Z = new real[M];

49         python::list pyli;

51         System& system = getSystemRef();

53         //creating vector which stores particleIDs for each CPU
54         vector<longint> localIDs;
55         for (map<size_t,int>::const_iterator itr=idToCpu.begin(); itr!=idToCpu.
               end(); ++itr) {
56           size_t i = itr->first;
57           int whichCPU = itr->second;
58           if(system.comm->rank()==whichCPU){
59             localIDs.push_back(i);
60           }
61         }

63         // COM calculation
64         vector<Real3D> centerOfMassList;
65         for(int m=0; m<M; m++){
66           Real3D posCOM = Real3D(0.0,0.0,0.0);
67           real mass = 0.0;
68           Real3D posCOM_sum = Real3D(0.0,0.0,0.0);
69           real mass_sum = 0.0;

71           for (vector<longint>::iterator itr=localIDs.begin(); itr!=localIDs.end
                 (); ++itr) {
72               size_t i = *itr;
73               Real3D pos = getConf(m)->getCoordinates(i);
74               posCOM += pos;
75               mass += 1;
76           }

78           boost::mpi::all_reduce(*mpiWorld, posCOM, posCOM_sum, std::plus<Real3D
                 >());
79           boost::mpi::all_reduce(*mpiWorld, mass, mass_sum, std::plus<real>());

81           centerOfMassList.push_back( posCOM_sum / mass_sum );
82         }

84         // MSD calculation
85         int perc=0, perc1=0;
86         real denom = 100.0 / (real)M;
87         for(int m=0; m<M; m++){

89           totZ[m] = 0.0;
90           Z[m] = 0.0;
91           for(int n=0; n<M-m; n++){
92             for (vector<longint>::iterator itr=localIDs.begin(); itr!=localIDs.
                   end(); ++itr) {
```

```
93              size_t i = *itr;

95              Real3D pos1 = getConf(n + m)->getCoordinates(i) - centerOfMassList
                    [n+m];
96              Real3D pos2 = getConf(n)->getCoordinates(i)     - centerOfMassList
                    [n];
97              Real3D delta = pos2 - pos1;
98              Z[m] += delta.sqr();
99            }
100         }
101         if(print_progress && system.comm->rank()==0){
102           perc = (int)(m*denom);
103           if(perc%5==0){
104             cout<<"calculation progress (mean square displacement): "<< perc
                    << " %\r"<<flush;
105           }
106         }
107       }
108       if(system.comm->rank()==0)
109         cout<<"calculation progress (mean square displacement): 100%"<<endl;
110       //summation of results from different CPUs
111       boost::mpi::all_reduce( *system.comm, Z, M, totZ, plus<real>() );

113       for(int m=0; m<M; m++){
114         totZ[m] /= (real)(M - m);
115       }

117       real inv_coef = 1.0 / (6.0 * num_of_part);

119       for(int m=0; m<M; m++){
120         totZ[m] *= inv_coef;
121         pyli.append( totZ[m] );
122       }

124       delete [] Z;
125       Z = NULL;
126       delete [] totZ;
127       totZ = NULL;

129       return pyli;
130     }

132     /*
133      * calculates mean square displacement of monomers in COM of their chains
134      *
135      * !! currently only works for particles numbered like 0, 1, 2,... !!
136      * !! with each chain consisting particles with subsequent ids     !!
137      *
138      * calc <r^2> the output is the average mean sq. displacement over 3
           directions.
139      * !! Important!! For D calculation factor 1/6 is already taken into
           account.
140      * !! all confs should contain the same number of particles
141     */
142     python::list MeanSquareDispl::computeG2() const{
143       cout << "0 got here!\n";
144       int M = getListSize(); //number of snapshots/configurations
145       real* totZ; //will store the mean squared displacement
146       totZ = new real[M];
147       real* Z;
148       Z = new real[M];
```

```
150         python::list pyli;

152         System& system = getSystemRef();

154         //creating vector which stores particleIDs for each CPU
155         vector<longint> localIDs; //for each CPU this will store particle IDs of
                particles calculated by CPU
156         for (map<size_t,int>::const_iterator itr=idToCpu.begin(); itr!=idToCpu.
                end(); ++itr) {
157           size_t i = itr->first; //particle ID
158           int whichCPU = itr->second; //CPU number
159           printf("id %u  CPU %i \n", i, whichCPU);
160           if(system.comm->rank()==whichCPU){
161             localIDs.push_back(i);
162           }
163         }
164         sort(localIDs.begin(), localIDs.end()); //sorts entries from low to high
165         //should not be necessary as long as the above iterator
166         //iterates in ascending order according to the keys

168         // COM calculation
169         Real3D posCOM = Real3D(0.0,0.0,0.0);
170         real mass = 0.0;
171         int count = 0; //counts number of particles of one chain
172         vector< vector<Real3D> > local_chainCOMlist; //will store COM of conf n
                and chain cid as chainCOMlist[n][cid]
173         for(int m=0; m<M; m++){
174           vector<Real3D> innerList; //will store the local chains' COM of one
                  conf/snapshot

176           //loop over local particles
177           for(int entry = 0; entry < localIDs.size(); entry++){
178               longint i = localIDs[entry]; //pid
179               Real3D pos = getConf(m)->getCoordinates(i);
180               posCOM += pos;
181               mass += 1;
182               count += 1;
183               // this is the right if request. remember that particle 0 also has
                      a mass of 1
184               if (count == chainlength){
185                   innerList.push_back( posCOM / mass);
186                   posCOM = Real3D(0.0,0.0,0.0);
187                   mass = 0;
188                   count = 0;
189               }
190           } //now innerList contains the local chains' COMs of snapshot m
191           local_chainCOMlist.push_back(innerList);
192         }
193         //now local_chainCOMlist contains the local chains' COMs of each
                snapshot

195         // MSD calculation
196         int perc=0, perc1=0;
197         real denom = 100.0 / (real)M;
198         for(int m=0; m<M; m++){
199           totZ[m] = 0.0;
200           Z[m] = 0.0;
201           for(int n=0; n<M-m; n++){
202             int part_count = 0;
```

```
203             int local_cid = 0; //local chainID. each CPU starts with chain
                    local_cid = 0, so it is not a global id
204             //loop over local particles
205             for(int entry = 0; entry < localIDs.size(); entry++){
206               longint i = localIDs[entry]; //pid
207               if(part_count == chainlength){
208                   ++local_cid;
209                   part_count = 0;
210               }
211               //cout << "n, m, i, local_cid, part_count   "
212               //        << n << "\t" << m << "\t"<< i << "\t" << local_cid << "\
                      t" << part_count << "\n";
213               Real3D pos1 = getConf(n + m)->getCoordinates(i) -
                      local_chainCOMlist[n+m][local_cid];
214               Real3D pos2 = getConf(n)->getCoordinates(i)     -
                      local_chainCOMlist[n][local_cid];
215               Real3D delta = pos2 - pos1;
216               Z[m] += delta.sqr();
217               part_count++;
218             }
219           }
220           if(print_progress && system.comm->rank()==0){
221             perc = (int)(m*denom);
222             if(perc%5==0){
223               cout<<"calculation progress (mean square displacement): "<< perc
                      << " %\r"<<flush;
224             }
225           }
226         }

228         if(system.comm->rank()==0)
229           cout<<"calculation progress (mean square displacement): 100%"<<endl;
230         //summation of results from different CPUs
231         boost::mpi::all_reduce( *system.comm, Z, M, totZ, plus<real>() );

233         for(int m=0; m<M; m++){
234           totZ[m] /= (real)(M - m);
235         }

237         real inv_coef = 1.0 / (6.0 * num_of_part);

239         for(int m=0; m<M; m++){
240           totZ[m] *= inv_coef;
241           pyli.append( totZ[m] );
242         }

244         delete [] Z;
245         Z = NULL;
246         delete [] totZ;
247         totZ = NULL;

249         return pyli;
250       }



254       // Python wrapping
255       void MeanSquareDispl::registerPython() {
256         using namespace espresso::python;

258         class_<MeanSquareDispl, bases<ConfigsParticleDecomp> >
```

```
259         ("analysis_MeanSquareDispl", init< shared_ptr< System > >() )
260         .def(init< shared_ptr< System >, int>() )
261         .def("computeG2", &MeanSquareDispl::computeG2)
262         .add_property("print_progress", &MeanSquareDispl::getPrint_progress, &
                MeanSquareDispl::setPrint_progress)
263         ;
264       }
265     }
266 }
```

Mean squared displacement - header file

```
1  /*
2    Copyright (C) 2012,2013
3        Max Planck Institute for Polymer Research
4    Copyright (C) 2008,2009,2010,2011
5        Max-Planck-Institute for Polymer Research & Fraunhofer SCAI

7    This file is part of ESPResSo++.

9    ESPResSo++ is free software: you can redistribute it and/or modify
10   it under the terms of the GNU General Public License as published by
11   the Free Software Foundation, either version 3 of the License, or
12   (at your option) any later version.

14   ESPResSo++ is distributed in the hope that it will be useful,
15   but WITHOUT ANY WARRANTY; without even the implied warranty of
16   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17   GNU General Public License for more details.

19   You should have received a copy of the GNU General Public License
20   along with this program.  If not, see <http://www.gnu.org/licenses/>.
21  */

23 // ESPP_CLASS
24 #ifndef _ANALYSIS_MEANSQUAREDISPL_HPP
25 #define _ANALYSIS_MEANSQUAREDISPL_HPP

27 #include "ConfigsParticleDecomp.hpp"

29 namespace espresso {
30     namespace analysis {

32         /*
33          * Class derived from ConfigsParticleDecomp.
34          *
35          * This implementation of mean square displacement calculation does
                not take into
36          * account particle masses. It is correct if all the particles have
                equal masses only.
37          * Otherwise it should be modified.
38          */

40         class MeanSquareDispl : public ConfigsParticleDecomp {
41         public:

43             MeanSquareDispl(shared_ptr<System> system) : ConfigsParticleDecomp
                    (system) {
44                 // by default
45                 setPrint_progress(true);
```

```
46                  key = "unfolded";
47              }

49              MeanSquareDispl(shared_ptr<System> system, int chainlength) :
50                                  ConfigsParticleDecomp(system, chainlength) {
51                  // by default
52                  setPrint_progress(true);
53                  key = "unfolded";
54              }

56              ~MeanSquareDispl() {
57              }

59              virtual python::list compute() const;
60              python::list computeG2() const;

62              void setPrint_progress(bool _print_progress) {
63                  print_progress = _print_progress;
64              }

66              bool getPrint_progress() {
67                  return print_progress;
68              }

70              static void registerPython();
71          private:
72              bool print_progress;
73          };
74      }
75  }

77  #endif
```

Mean squared displacement - python file

```
1   #  Copyright (C) 2012,2013
2   #      Max Planck Institute for Polymer Research
3   #  Copyright (C) 2008,2009,2010,2011
4   #      Max-Planck-Institute for Polymer Research & Fraunhofer SCAI
5   #
6   #  This file is part of ESPResSo++.
7   #
8   #  ESPResSo++ is free software: you can redistribute it and/or modify
9   #  it under the terms of the GNU General Public License as published by
10  #  the Free Software Foundation, either version 3 of the License, or
11  #  (at your option) any later version.
12  #
13  #  ESPResSo++ is distributed in the hope that it will be useful,
14  #  but WITHOUT ANY WARRANTY; without even the implied warranty of
15  #  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16  #  GNU General Public License for more details.
17  #
18  #  You should have received a copy of the GNU General Public License
19  #  along with this program.  If not, see <http://www.gnu.org/licenses/>.


22  """
23  **************************************
24  **espresso.analysis.MeanSquareDispl**
25  **************************************
```

```
27 """
28 from espresso.esutil import cxxinit
29 from espresso import pmi

31 from espresso.analysis.ConfigsParticleDecomp import *
32 from _espresso import analysis_MeanSquareDispl

34 class MeanSquareDisplLocal(ConfigsParticleDecompLocal,
       analysis_MeanSquareDispl):
35     'The (local) compute autocorrelation f.'
36     def __init__(self, system, chainlength = None):
37       if chainlength is None:
38         cxxinit(self, analysis_MeanSquareDispl, system)
39       else:
40         cxxinit(self, analysis_MeanSquareDispl, system, chainlength)

42     def computeG2(self):
43       return self.cxxclass.computeG2(self)

45     def strange(self):
46       print 1
47       return 1

49 if pmi.isController:
50   class MeanSquareDispl(ConfigsParticleDecomp):
51     __metaclass__ = pmi.Proxy
52     pmiproxydefs = dict(
53       cls =  'espresso.analysis.MeanSquareDisplLocal',
54       pmiproperty = [ 'print_progress' ],
55       pmicall = ["computeG2", 'strange']
56     )
```

A.2. Alternative codes

A.2.1. Static structure factor

Alternative code for the computation of the static structure factor, which uses less scattering vectors for higher moduli and which is parallelized over scattering vectors.

```
0  //this is designed with a cubic symmetry (as far as interger multipliers for
       the scattering vectors are concerned)


3  int num_layers = 10;  //number of layers. example value (specified by user in
       final code)
4  int num_steps = 3;    //number of steps per layer. example value (specified by
        user in final code)

6  vector<int> posAxisPoints; //positive values that hx, hy and hz the
       multipliers for the scattering vector can take

8  int gridpoint = 1; //one value on the axis. initialized with "1" since
       positive values are created first
9  while(gridpoint <= num_steps){
10   posAxisPoints.pushback(i);
11   gridpoint++;
```

```
12  }


15  //now: gridpoint = num_steps;
16  int stepsize = 1;
17  for(int layer = 0; layer < num_layers; layer++){
18    printf("-%i-  \t", stepsize);
19    for(int step=0; step<num_steps; step++){
20      gridpoint += stepsize;
21      posAxisPoints.pushback(gridpoint);
22      //cout << gridpoint << "\t";
23      printf("%i   ",gridpoint);
24    }
25    printf("\n");
26    stepsize *= 2; //grows exponentially with jump to next layer
27  }
28  //check size of posAxisPoints
29  int num_posAxisPoints = num_layers * num_steps + num_steps; // '+num_steps'
        because of the innermost layer
30  if(posAxisPoints.size() != num_posAxisPoints){
31    printf("ERROR: wrong number of axis points.");
32    printf(" Desired number: %i", num_posAxisPoints);
33    printf(" Currently filled: %i \n", posAxisPoints.size());
34  }

36  vector<int> axisPoints;
37  //filling the axis. first negative side, then 0, then positive side
38  for(int i = posAxisPoints.size() - 1; i >= 0; i--){
39    int negPoint = - posAxisPoints[i];
40    axisPoints.pushback(negPoint);
41  }
42  axisPoints.pushback(0);
43  for(int i = 0; i < posAxisPoints.size(); i++){
44    int posPoint = posAxisPoints[i];
45    axisPoints.pushback(posPoint);
46  }

48  //int layer;  //the layer in which a gridpoint is positioned
49  int qcount = -1; //counts the q vectors that are used for the computation. is
        also used for parallelization

51  int num_axisPoints = 2*num_posAxisPoints + 1; //'2*' because of negative
        values. '+1' because of zero.
52  for(int ix = 0; ix < num_axisPoints; ix++){
53    int hx = axisPoints[ix];
54    int axisPos_x = abs(ix - num_posAxisPoints); //abs is the absolute value
55    int layer = (int) ceil(axisPos_x / (double) num_steps) - 2; //the layer in
          which a gridpoint is positioned
56                                  //correction for layer 0 in if statement below
57    if(layer < 0) layer = 0;
58    for(int iy = 0; iy < num_axisPoints; iy++){
59      int hy = axisPoints[iy];
60      int axisPos_y = abs(iy - num_posAxisPoints); //abs is the absolute value
61      int layer_y = (int) ceil(axisPos_y / (double) num_steps) - 2; //the layer
            in which a gridpoint is positioned
62                                    //correction for layer 0 in if statement
                  below
63      if(layer_y < 0) layer_y = 0;
64      layer = max(layer, layer_y);
65        for(int iz = 0; iz < num_posAxisPoints + 1; iz++){
66        int hz = axisPoints[iz];
```

```
67        int axisPos_z = abs(iz - num_posAxisPoints); //abs is the absolute value
68        int layer_z = (int) ceil(axisPos_z / (double) num_steps) - 2; //the
              layer in which a gridpoint is positioned
69                                        //correction for layer 0 in if statement
                                              below
70        if(layer_z < 0) layer_z = 0;
71        layer = max(layer, layer_z);
72        int longestQ = num_steps * 2 ^ (layer + 1);
73        if(hx*hx + hy*hy + hz*hz > longestQ*longestQ)
74          break;
75        else{
76          qcount++;
77          //assign proc to current q-vector and call computeS
78          if(qcount%nprocs == myrank)
79            computeS(hx,hy,hz);
80        }
81      }
82    }
83  }
```

A.3. Additional graphs

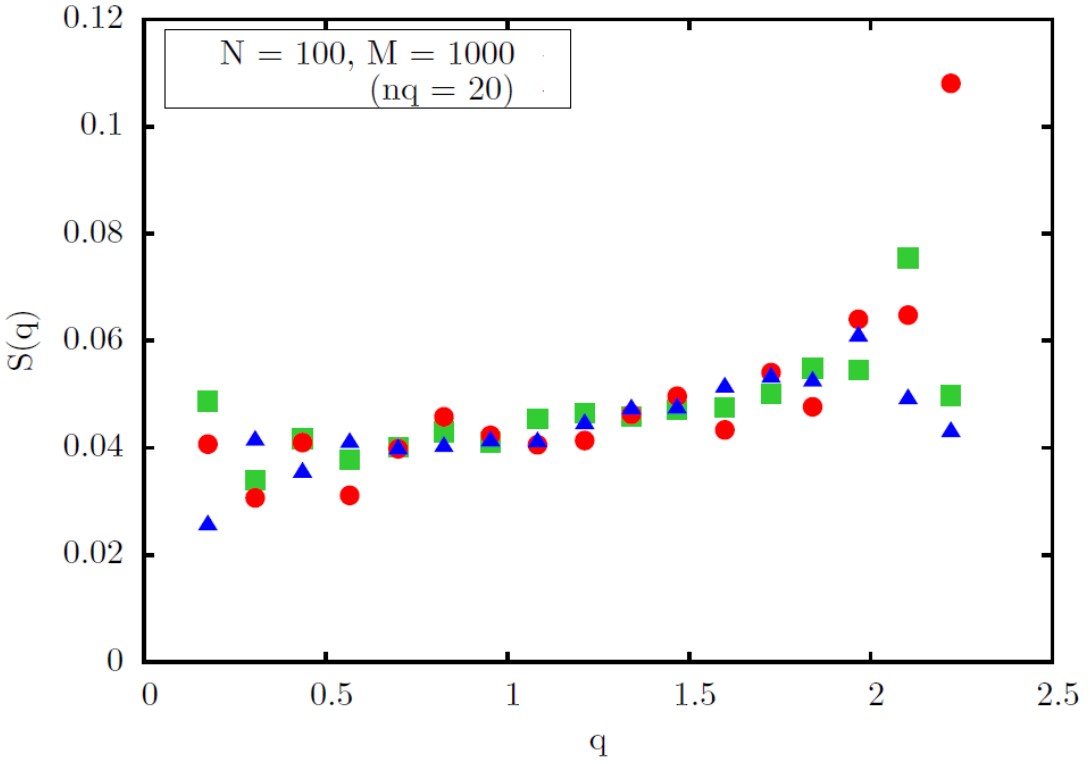

Figure A.1.: The static structure factor for three configurations of stiffness $k_\theta = 0.75$ with original box length. Each marker type corresponds to one configuration.

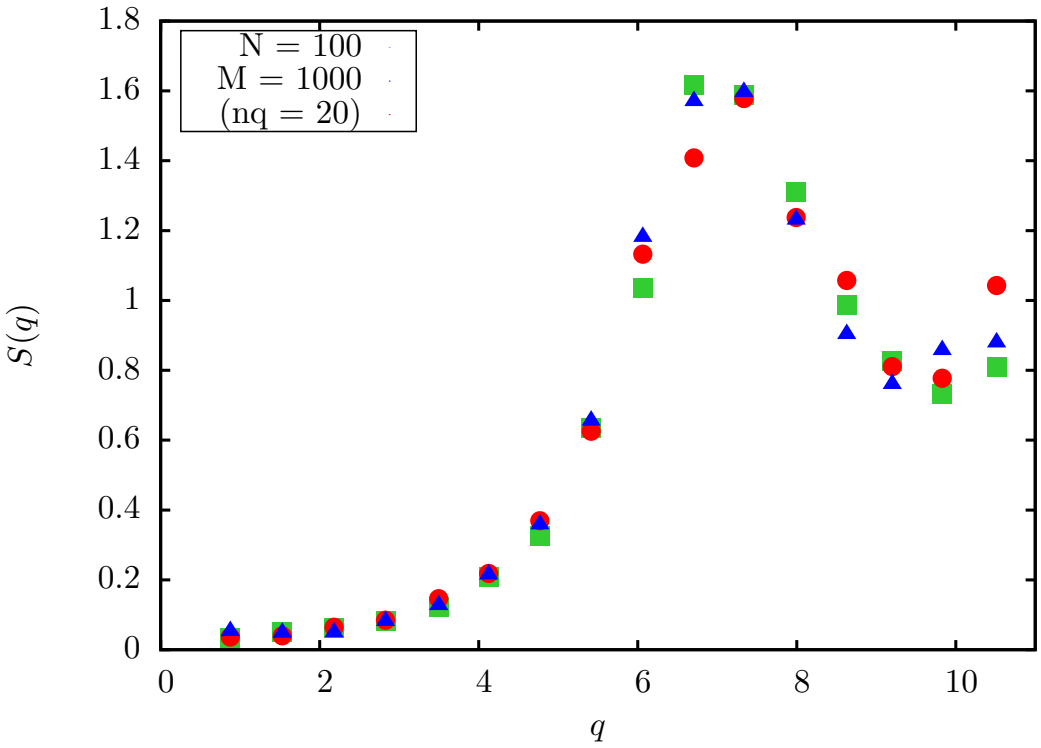

Figure A.2.: The static structure factor for three configurations of stiffness $k_\theta = 0.75$ with modified box length for the calculation. The box length was divided by five. Each marker type corresponds to one configuration.

Bibliography

[All89] Allen M.P. and Tildesley D.J., *Computer Simulation of Liquids*, Clarendon Press, New York, NY, USA (1989).

[Arb96] Arbe A., Richter D., Colmenero J. and Farago B., *Merging of the α and β relaxations in polybutadiene: A neutron spin echo and dielectric study*, Phys. Rev. E, **54**, 3853–3869 (1996), doi:10.1103/PhysRevE.54.3853, URL `http://link.aps.org/doi/10.1103/PhysRevE.54.3853`.

[Auh03] Auhl R., Everaers R., Grest G.S., Kremer K. and Plimpton S.J., *Equilibration of long chain polymer melts in computer simulations*, The Journal of Chemical Physics, **119**, 12 718–12 728 (2003), doi:10.1063/1.1628670.

[Bas94] Baschnagel J. and Binder K., *Structural aspects of a three-dimensional lattice model for the glass transition of polymer melts: a Monte Carlo simulation*, Physica A: Statistical Mechanics and its Applications, **204**(1), 47–75 (1994), URL `http://EconPapers.repec.org/RePEc:eee:phsmap:v:204:y:1994:i:1:p:47-75`.

[Bas00] Baschnagel J., Bennemann C., Paul W. and Binder K., *Dynamics of a supercooled polymer melt above the mode-coupling critical temperature: cage versus polymer-specific effects*, Journal of Physics: Condensed Matter, **12**(29), 6365 (2000), URL `http://stacks.iop.org/0953-8984/12/i=29/a=308`.

[Bin05] Binder K. and Kob W., *Glassy Materials and Disordered Solids*, World Scientific Publishing, 5 Toh Tuck Link, Singapore 596224 (2005).

[Bul08] Bulacu M., *Molecular Dynamics Studies of Entangled Polymer Chains*, dissertation, Rijksuniversiteit Groningen (2008), URL `http://dissertations.ub.rug.nl/faculties/science/2008/m.i.bulacu/thesis.pdf`.

[Bus31] Busse W.F., *The Physical Structure of Elastic Colloids*, The Journal of Physical Chemistry, **36**(12), 2862–2879 (1931), doi:10.1021/j150342a002, URL `http://pubs.acs.org/doi/abs/10.1021/j150342a002`.

[Cat00] Cates M. and Evans R., *Soft and Fragile Matter: Nonequilibrium Dynamics, Metastability and Flow (PBK)*, Scottish Graduate Series, Taylor & Francis (2000), URL `http://books.google.de/books?id=WX5uxkEpDYAC`.

[Far11] Farago J., Meyer H. and Semenov A.N., *Anomalous Diffusion of a Polymer Chain in an Unentangled Melt*, Phys. Rev. Lett., **107**, 178 301 (2011), doi:10.1103/PhysRevLett.107.178301, URL `http://link.aps.org/doi/10.1103/PhysRevLett.107.178301`.

[Fil12] Filipovic L., *Topography Simulation of Novel Processing Techniques*, dissertation, Technische Universität Wien (2012), URL `http://www.iue.tuwien.ac.at/phd/filipovic/node26.html`.

[Hal13] Halverson J.D., Brandes T., Lenz O., Arnold A., Bevc S., Starchenko V., Kremer K., Stuehn T. and Reith D., *ESPResSo++: A modern multiscale simulation package for soft matter systems*, Computer Physics Communications, **184**(4), 1129 – 1149 (2013), doi:http://dx.doi.org/10.1016/j.cpc.2012.12.004, URL `http://www.sciencedirect.com/science/article/pii/S0010465512004006`.

[Han86] Hansen J.P. and McDonald I.R., *Polymers and Neutron Scattering*, Academic Press Inc., Orlando, Florida 32887, second edition Auflage (1986).

[Hig94] Higgins J.S. and Benoît H.C., *Polymers and Neutron Scattering*, Oxford University Press Inc., Walton Street, Oxford OX2 6DP (1994).

[Hor99] Horbach J. and Kob W., *Static and Dynamic Properties of a Viscous Silica Melt Molecular Dynamics Computer Simulations* (1999), doi:10.1103/PhysRevB.60.3169.

[Kre83] Kremer K., *Statics and dynamics of polymeric melts: a numerical analysis*, Macromolecules, **16**(10), 1632–1638 (1983), doi:10.1021/ma00244a015, URL `http://pubs.acs.org/doi/abs/10.1021/ma00244a015`.

[Mor14] Moreira L.A., Müller F., Stühn T. and Kremer K., *Direct Equilibration and Characterization of Polymer Melts* (2014), in preparation.

[OCAa] OCA, *notepad*, `https://openclipart.org/image/2400px/svg_to_png/169875/Rmx_Notes.png`, accessed: 2014-04-13.

[OCAb] OCA, *terminal*, `https://openclipart.org/image/2400px/svg_to_png/171762/1345126450.png`, accessed: 2014-04-13.

[OCAc] OCA u.d., *Simple icon representing a user*, `https://openclipart.org/image/800px/svg_to_png/167201/1326861328.png`, accessed: 2014-04-13.

[Pri87] Price D. and Carpenter J., *Scattering function of vitreous silica*, Journal of Non-Crystalline Solids, **92**(1), 153 – 174 (1987), doi:http://dx.doi.org/10.1016/S0022-3093(87)80366-6, URL `http://www.sciencedirect.com/science/article/pii/S0022309387803666`.

[Rub03] Rubinstein M. and Colby R.H., *Polymer Physics*, Oxford University Press Inc., Great Claredon Street, Oxford OX2 6DP (2003).

[Stü] Stühn T.

[Tre40] Treloar L.R.G., *Elastic recovery and plastic flow in raw rubber*, Trans. Faraday Soc., **35**, 538–549 (1940), doi:10.1039/TF9403500538, URL `http://dx.doi.org/10.1039/TF9403500538`.

[Wee71] Weeks J.D., Chandler D. and Andersen H.C., *Role of Repulsive Forces in Determining the Equilibrium Structure of Simple Liquids*, The Journal of Chemical Physics, **54**(12) (1971).