Simulation Studies of Correlation Functions and Relaxation in Polymeric Systems

Dissertation

zur Erlangung des Grades

„Doktor der Naturwissenschaften“

am Fachbereich Physik
der Johannes Gutenberg – Universität Mainz

vorgelegt von

Martin Aichele

geboren am 16.06.1974 in Marburg

Mainz, April 2003

Datum der mündlichen Prüfung: 25. Juli 2003

Abstract

We investigate the statics and dynamics of a glassy, non-entangled, short bead-spring polymer melt with molecular dynamics simulations. Temperature ranges from slightly above the mode-coupling critical temperature to the liquid regime where features of a glassy liquid are absent. Our aim is to work out the polymer specific effects on the relaxation and particle correlation.
We find the intra-chain static structure unaffected by temperature, it depends only on the distance of monomers along the backbone. In contrast, the distinct inter-chain structure shows pronounced site-dependence effects at the length-scales of the chain and the nearest neighbor distance. There, we also find the strongest temperature dependence which drives the glass transition. Both the site averaged coupling of the monomer and center of mass (CM) and the CM-CM coupling are weak and presumably not responsible for a peak in the coherent relaxation time at the chain's length scale. Chains rather emerge as soft, easily interpenetrating objects. Three particle correlations are well reproduced by the convolution approximation with the exception of model dependent deviations.
In the spatially heterogeneous dynamics of our system we identify highly mobile monomers which tend to follow each other in one-dimensional paths forming "strings". These strings have an exponential length distribution and are generally short compared to the chain length. Thus, a relaxation mechanism in which neighboring mobile monomers move along the backbone of the chain seems unlikely. However, the correlation of bonded neighbors is enhanced.
When liquids are confined between two surfaces in relative sliding motion kinetic friction is observed. We study a generic model setup by molecular dynamics simulations for a wide range of sliding speeds, temperatures, loads, and lubricant coverings for simple and molecular fluids. Instabilities in the particle trajectories are identified as the origin of kinetic friction. They lead to high particle velocities of fluid atoms which are gradually dissipated resulting in a friction force. In commensurate systems fluid atoms follow continuous trajectories for sub-monolayer coverings and consequently, friction vanishes at low sliding speeds. For incommensurate systems the velocity probability distribution exhibits approximately exponential tails. We connect this velocity distribution to the kinetic friction force which reaches a constant value at low sliding speeds. This approach agrees well with the friction obtained directly from simulations and explains Amontons' law on the microscopic level. Molecular bonds in commensurate systems lead to incommensurate behavior, but do not change the qualitative behavior of incommensurate systems. However, crossed chains form stable load bearing asperities which strongly increase friction.

Zusammenfassung

Statik und Dynamik einer glasartigen Schmelze aus kurzkettigen, nicht-verschlauften Kugel-Feder Polymeren werden mittels Molekulardynamik-Simulationen untersucht. Die Temperatur reicht von kurz oberhalb der kritischen Temperatur der Modenkopplungstheorie bis in die Flüssigphase, in der keine glastypischen Eigenschaften mehr auftreten. Unser Ziel ist es, die für Polymere spezifischen Effekte der Relaxation und Teilchenkorrelation heraus zu arbeiten.
Die statische Einzelkettenstruktur ändert sich kaum mit der Temperatur, sie hängt nur vom Abstand der Monomere entlang der Molekülkette ab. Im Gegensatz dazu zeigt die Fremdkettenstruktur deutliche Abhängigkeiten von der Monomerposition auf den Längenskalen der Kette und des nächsten Nachbarn. Dort stellen wir auch die stärkste Temperaturabhängigkeit fest, die den Glasübergang treibt. Sowohl die über die Kette gemittelte Kopplung zwischen Monomeren und Schwerpunkt (SP) als auch die SP-SP Kopplung sind schwach und vermutlich nicht für ein Maximum der kollektiven Relaxationszeit auf der Längenskala der Kette verantwortlich. Die Polymere stellen sich vielmehr als weiche, sich leicht durchdringende Objekte dar. Dreiteilchenkorrelationen werden durch die Faltungsnäherung bis auf modellabhängige Abweichungen gut wieder gegeben.
In der räumlich heterogenen Dynamik unseres Systems identifizieren wir hoch mobile Teilchen, die sich in eindimensionalen „Schnüren“ folgen. Die Schnüre haben eine exponentielle Längenverteilung und sind im Allgemeinen kurz im Vergleich zur Kettenlänge. Ein Relaxationsmechanismus, in dem benachbarte mobile Teilchen einander entlang des Moleküls folgen, erscheint daher unwahrscheinlich. Die Korrelation von gebundenen Nachbarn ist allerdings erhöht.
Flüssigkeitsschichten, die sich zwischen zwei gegeneinander gescherten Wänden befinden, zeigen kinetische Reibung. Wir untersuchen einen generischen Aufbau mit Molekulardynamik-Simulationen über einen weiten Bereich der Schergeschwindigkeit, Temperatur, Last und Flüssigkeitsbedeckung für einfache und molekulare Flüssigkeiten. Wir identifizieren Instabilitäten in den Teilchentrajektorien als den Ursprung der kinetischen Reibung. Diese führen zu hohen Teilchengeschwindigkeiten, die allmählich dissipiert werden, woraus eine Reibungskraft resultiert. In kommensurablen Systemen folgen die Flüssigkeitsatome stetigen Bahnen für Bedeckungen unterhalb einer Monolage, so dass die Reibung für niedrige Geschwindigkeiten verschwindet. Für inkommensurable Systeme zeigt die Wahrscheinlichkeitsverteilung der Geschwindigkeit näherungsweise exponentielle Schwänze. Diese Geschwindigkeitsverteilung verknüpfen wir mit der kinetischen Reibung, die einen konstanten Wert für niedrige Schergeschwindigkeiten annimmt. Dieser Zugang stimmt gut mit der direkt in der Simulation gemessenen Reibung überein und erklärt das Gesetz von Amontons auf mikroskopischer Ebene. Molekülbindungen in kommensurablen Systemen führen zu inkommensurablem Verhalten, ändern aber nicht das qualitative Verhalten von inkommensurablen Systemen. Überkreuzte Ketten bilden jedoch stabile Unebenheiten, die eine hohe Last tragen und die Reibung stark erhöhen.

Résumé

Nous avons étudié à l'aide de simulations de Dynamique Moléculaire les propriétés statiques et dynamiques d'un fondu vitreux de polymères courts et non-enchevêtrés en utilisant un modèle bille-ressort. L'échelle de température considérée s'étend d'une valeur légèrement supérieure à la température critique de la théorie de couplage de modes jusqu'au régime liquide pour lequel les caractéristiques vitreuses sont absentes. Notre objectif est de déterminer les effets typiques des polymères pour la relaxation et les corrélations entre particules.
Nous avons montré que la structure statique à l'intérieur d'une même chaîne n'est pas modifiée par la température mais dépend uniquement de la distance entre monomères le long de la chaîne. Par contre, la structure inter-chaîne dépend fortement du site considéré aux échelles de longueur correspondant à la taille de la chaîne et à la distance entre plus proches voisins. A cette distance, nous avons trouvé la plus forte dépendance en température qui induit la transition vitreuse. Le couplage monomère-centre de masse (CM) moyenné par site et le couplage CM-CM sont tous les deux faibles et ne sont vraisemblablement pas à l'origine d'un pic dans le temps de relaxation de la fonction de structure cohérente à l'échelle de longueur de la chaîne. Les chaînes ressortent plutôt comme des objets mous, s'interpénétrant facilement. Les corrélations à trois particules sont bien reproduites par l'approximation de convolution, à l'exception de certaines déviations qui sont fonctions du modèle.
Pour la dynamique spatiallement hétérogène de notre système, nous avons identifié des monomères ayant une grande mobilité qui ont tendance à se suivre les uns les autres sur des chemins unidimensionnels, formant des «cordes». Ces cordes ont une distribution des longeurs exponentielle et sont en général courtes par rapport à la longueur de la chaîne. Un mécanisme de relaxation dans lequel les monomères mobiles voisins se déplacent le long de la chaîne semble donc peu probable. Cependant, les corrélations entre monomères voisins liés sont renforcées.
Quand des liquides sont confinés entre deux surfaces en mouvement de glissement relatif, une friction cinétique est observée. Nous avons étudié un dispositif modèle par des simulations de Dynamique Moléculaire pour une vaste gamme de vitesses de glissement, températures, charges et taux de couverture d'adsorbants pour des fluides simples et moléculaires. Des instabilités dans les trajectoires des particules ont pu être identifiées comme origine de la friction cinétique. Elles induisent des vitesses élevées des atomes du fluide qui sont progressivement dissipées, ce qui conduit à l'apparition d'une force de friction. Dans des systèmes commensurables, les atomes suivent des trajectoires continues pour des couvertures moindres qu'une monocouche et par suite, la friction disparaît à basse vitesse de glissement. Pour des systèmes incommensurables, la distribution de probabilités des vitesses comprend des queues de forme approximativement exponentielle. Nous avons relié cette distribution à la force de friction qui atteint une valeur constante pour des faibles vitesses de glissement.

Cette approche s'accorde bien avec la friction tirée directement des simulations et permet d'expliquer la loi d'Amontons à l'échelle microscopique. Les liaisons moléculaires dans des systèmes commensurables conduisent à un comportement incommensurable, mais ne modifient pas le comportement qualitatif des systèmes incommensurables. Cependant, des croisements de chaînes forment des aspérités qui augmentent fortement la friction.

Contents

List of Figures

Chapter 1

Introduction

Most non-metallic materials in the solid state that surround us are made up of molecules. If these molecules consist of smaller repeating units (monomers) which are chemically bonded to each other, one speaks of polymers [1]. Many of these chain molecules are very familiar to us in the form of plastics. The understanding of the properties of the static and dynamic structure of polymers is a long-standing problem in the physics of condensed matter with a strong impact on applications and has often been tackled by computer simulations [2]. While some polymers crystallize, others do not, but form a "glassy solid" when they are cooled below the melting temperature [3]. The ordinary glass we know from windows and wine glasses is more precisely silica glass and behaves in a similar fashion. It does not have a crystalline structure as opposed to its crystallized form, quartz. In fact, many substances show a dramatic increase of the relaxation time in a narrow temperature interval below the melting temperature. If the relaxation time exceeds days we perceive these substance as solids. Their behavior has universal features, which justifies to call all these materials "glasses" in a physical sense [4, 5]. In the glassy state, the structure is characterized by the absence of long-range periodic order just like in a liquid, hence one can say that glasses are "supercooled liquids".

Although the static and dynamic properties of supercooled fluids, in particular polymers, have already been studied by simulation methods very extensively, there are still many open questions, as we shall discuss succinctly below. The present study builds on these previous efforts, and extends them in several directions of high current interest. These directions are briefly summarized in the following.

1.1 Concepts Pertaining to Glassy Dynamics

A very successful description of the glassy dynamics of liquids was proposed by the "mode-coupling theory" (MCT) [6–8]. It describes the density fluctuations by coupled integro-differential equations and predicts a novel dynamic transition phenomenon, causing a dramatic slowing down of the dynamics leading to structural arrest of the system at the critical temperature T_c of MCT. The driving force of the dramatic slowing down of the relaxation is in the MCT-picture the neighbor shell of a particle which becomes so tight at T_c that the dynamics comes to a standstill. MCT needs as input only the static structure data of the system under consideration and predicts the long-time dynamics.

Originally, MCT has been developed for simple fluids, in particular hard spheres. Yet, both qualitative and quantitative predictions of MCT have been reported to describe very well the dynamics of a variety of glass-forming systems including polymers, for reviews see Refs. [5, 7, 9–11].

Computer simulations offer the possibility to analyze the statics and dynamics in a very detailed fashion for well defined model systems. Simplified model systems allow for the investigation of generic features and have the advantage that the computational cost decreases. To this end, a computer simulation model system of a short, non-entangled glass-forming polymer has been developed by Bennemann et al. as a simplified "bead-spring" model [12, 13]. In this model, the full atomistic description is replaced by point particles with empirical pair potentials. For this system it was shown in a series of studies that the basic form of MCT for simple liquids describes the dynamics near the glass transition fairly well [14–17]. Some deviations remained unexplained, however. In particular, the role of the molecular bonds in the relaxation dynamics was not investigated and a comparison with theoretical predictions in the framework of MCT concerning the influence of bonds was not possible, because an extension of MCT to chain molecules did not exist.

This situation has changed very recently, as MCT has been extended to molecules [18, 19]. In particular for short chain molecules this extension contains both the well-known Rouse model [20] and the so-called 'idealized' MCT for simple fluids as special cases. Hence, it is very promising to extend the analysis of the simulation data to test to what extent this new theory correctly accounts for the relevant chain specific effects. We will study the precise connection of center of mass and monomer motion which arises naturally for molecules and is important not only for this new theoretical approach. Furthermore, we will address the site dependence of the structure, e.g. if chain ends behave differently than monomers in the middle of the chain. In addition, the necessary static input data for MCT can be provided so that a quantitative comparison with simulation data without any adjustable parameters can be achieved. Finally, the validity of assumptions entering the derivation of the MCT for polymers can be tested.

Another interesting feature of the glassy dynamics is that the particle dynamics becomes increasingly spatially non-uniform [21–24]. This so-called "dynamic heterogeneity" has been proposed to be related to the long, stretched form of the decay of dynamic correlators of glass formers. For our polymer melt it has also been observed [25, 26]. These investigations focused on analyses which stressed the similarity with simple fluids. However, for polymers it is tempting to ask whether bonds provide a preferred direction for the 'transmission' of dynamical heterogeneity.

Our studies thus aim at specifically analyzing the effect of molecular bonds and of the center of mass motion on the relaxation and correlated motion in a simulated glassy polymer melt.

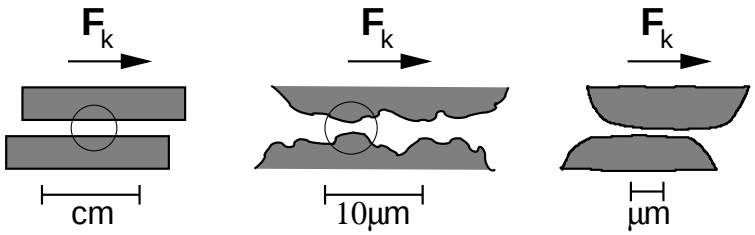

Figure 1.1: A macroscopic contact of nominally flat walls (left) has microscopic contact zones which are corrugated on a smaller length scale (middle). The most fundamental unit in tribology is the single micro-asperity (right). We will investigate its response to an external shear force F_k by means of molecular dynamics simulations.

1.2 From Macroscopic to Microscopic Friction

Another mechanism strongly affecting the particle dynamics of a liquid is the coupling to surfaces. When two surfaces form a contact, a thin film of fluid can be confined between them which will strongly influence the frictional behavior when the surfaces are sheared against each other [27, 28]. This is something we use in everyday life when we oil a squealing door. Although friction is such an ubiquitous phenomenon with classical macroscopic descriptions [29] dating back to the days of DA VINCI (1452–1519) which are appropriate even for modern engineering tasks, we still do not have a satisfying microscopic understanding of the kinetic friction of sheared contacts.

However, the interest in understanding the physics of nanoscopic contacts has grown, too, because of the miniaturization of technical devices down to very small dimensions, like in computer hard disks. Furthermore, macroscopic contacts interact only via small contact zones, called "asperities" [27, 28], as depicted in Fig. 1.1. The understanding of the microscopic origins of friction in single asperities is therefore needed as a base on which a full macroscopic description of tribological phenomena can be built upon.

One distinguishes the static friction force F_s as the force needed to initialize sliding and the kinetic friction force F_k which maintains sliding. For static friction, the determining role of absorbed surfactants has been recognized over the last years [30–32]. For many experimental situations and applications, the kinetic friction is more important, and a microscopic description of it is still missing. Computer simulations provide an excellent tool for the investigation of microscopic friction, as the 'experimental' conditions can be completely controlled. On the contrary, controlling the surfaces and ambient conditions is notoriously difficult for friction experiments, like with the surface forces apparatus [33]. In order to identify the microscopic mechanisms of friction for a generic case, we use a setup where two crystalline walls confine a liquid under shear and concentrate on the particle dynamics and its link to the friction force. We begin with single particle adiabatic trajectories and successively enhance the complexity of our model. In this way, we arrive at thin films with varying thickness confined between atomically smooth surfaces with LENNARD-JONES type interaction which are studied by molecular dynamics simulations. While a large

part of our study is done for simple fluids, we extend it also to polymers under shear.

When preparing ultrathin films comprised of bead-spring polymers for our tribological studies we found that it was difficult to construct long chains. This lead us to implement the recently proposed "recoil growth"-algorithm [34, 35] for the construction of long polymers in thin films and strictly two-dimensional systems. As this is the first study of two-dimensional polymers with the recoil growth algorithm we investigated the dependence of its efficiency on the parameters of the algorithm.

1.3 Organization of the Present Work

This work is layed out as follows. First we will describe the molecular dynamics simulation methods used for the study of the tribology of interfaces in chapter 2. They are similar to the methods used to simulate the glassy polymer melt which we discuss in chapters 3 and 4.

Chapter 3 contains our analysis of the static properties of a polymer melt under special consideration of chain-specific effects. They will be connected to the dynamics in the framework of MCT. The dynamical heterogeneities and correlated motion found in this glassy polymer will be investigated in chapter 4.

We show how discontinuous particle trajectories arise in boundary lubricated systems and connect them to kinetic friction in chapter 5. Each of the aforementioned chapters will be closed by a summary and conclusion.

Topics which arose during the course of this work which have not been fully worked out can be found in appendix A and appendix B. The remainder of the appendix contains the simulation and analysis code used for our study.

Chapter 2

Simulation Methods and Models

2.1 Introduction

Computer simulation methods have become an established third pillar of the physical sciences next to analytic theory and experiment for the investigation of molecular scale models of matter [2, 36, 37]. Their success began in the 1950s with the first computers and took a second upswing in the 1970s when computers with ever increasing computational power become affordable.

When quantitative measurements by means of computer simulations are performed, one computes an ensemble average, e.g. in the canonical NVT ensemble. More precisely, one averages over a representative subset of the hypersurface in phase space defined by the thermodynamic conditions (particle number N, volume V, and temperature T in our example). Molecular dynamics (MD) simulations compute the (continuous) trajectories of the particles when all positions and interactions are known at the start of the simulation. If the system is ergodic, the trajectories cover the hypersurface densely in time and the long time average equals the ensemble average. Hence, MD for classical systems basically consists of solving NEWTON's equations of motion numerically with certain boundary conditions, such as constant temperature. It is important to note that the computed trajectories need not be the 'true' physical ones, it suffices that they lie on the correct hypersurface. In fact, they do not even have to be continuous. In Monte Carlo (MC) simulations, the system is moved from one point on the hypersurface to another one discontinuously. Each MC move is determined by the probability of the old and new point in the chosen ensemble and randomly accepted according to it. The hypersurface is thereby sampled pointwise yielding the ensemble average for an infinite number of MC steps. Depending on the problem, 'unphysical' MC moves can (often) sample the phase space faster to obtain a static quantity, whereas the microscopic dynamics of the system is (often) better accessed by MD simulations.

After these general remarks, we will next give an overview of our simulation program and lay out the algorithms employed in more detail.

2.2 Program Overview

Our simulation code we will describe in this chapter can perform MD or MC simulations in one, two or three dimensions. The basic set-up consists of two flat, two-dimensional walls confining a liquid. The liquid can be a simple one or a polymer with a degree of polymerization N up to $N \approx 10^3$. We can also perform bulk simulations without walls in $d = 1$, 2, or 3 dimensions. All results reported in the present work are for 2-d fluid films, either purely 2-d films or thin 3-d films confined between two walls. The program code itself is printed in appendix C.

Walls are composed of one layer of Lennard-Jones (LJ) atoms on a hexagonal lattice. In the MD part of the program, in each direction three simulation modes can be chosen in the simulation: First, the "force mode" with constant force or with a linear force ramp. Second, the "velocity mode" moving one wall with constant velocity in the direction (including velocity zero, which fixed the walls in space). Third, the "spring mode" where one wall is pulled with a spring of variable stiffness which is in turn moved with constant velocity. Not all modes can sensibly be employed in every direction.

When polymers are simulated, we use a "bead-spring" model devised by Kremer and Grest [38]. The start-up configurations can be defined using a simple random walk, a configurational bias monte carlo method or the so called "recoil-growth" algorithm.

The results reported in chapters 3 and 4 for a polymer melt are based on simulations done with another MD-simulation program by Bennemann. We will not describe this code here, but refer the reader to section 3.2 and references therein for a description. The MD-methods used there are however similar to the ones which we are going to describe next.

2.3 Potentials

Because the length and time scales of real polymers diverge by many orders of magnitude [2] (electron motion 10^{-17}s, typical time scale for the relaxation of liquid polymers 10^{-3}s) not all details can be kept in computer simulations when performing simulations involving many particles or which span a long time. Hence, a "coarse-graining" of the model has to be done in order to obtain a computationally tractable model. To this end, one or several repeat units of the polymer are replaced by a single "effective monomer". In addition, the exact interaction potentials are replaced by empirical effective potentials.

For walls of a crystal surface, similar reasonings can be made. Instead of using realistic atomic interactions, simplified models can dramatically reduce the computational cost while yielding the same information.

A very generic and computationally efficient model for particle interaction is given by the Lennard-Jones (LJ) potential which we used in the simulation for all particle interactions (fluid-fluid, fluid-wall, wall-wall),

$$U_{\mathrm{LJ}}(r_{ij}) = \begin{cases} 4\varepsilon \left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6} \right] + C(r_{\mathrm{cut}}), & r_{ij} < r_{\mathrm{cut}} \\ 0, & r_{ij} \geq r_{\mathrm{cut}} \end{cases} , \tag{2.1}$$

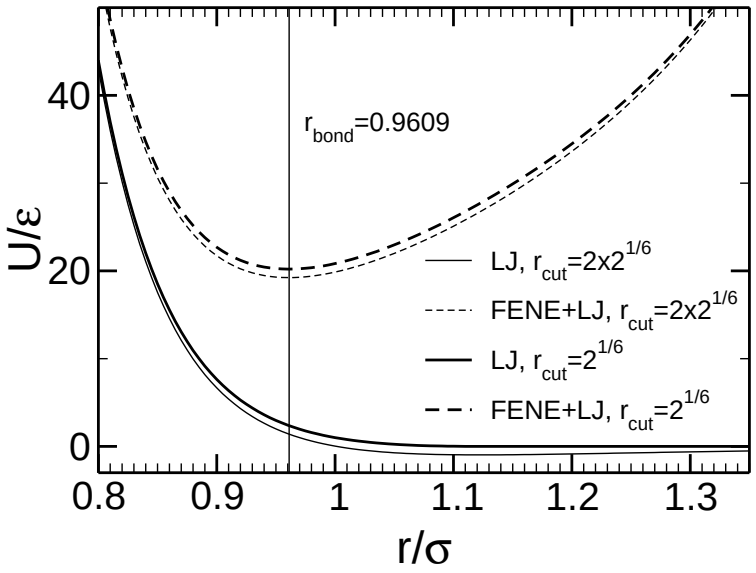

Figure 2.1: LENNARD-JONES (LJ) potentials, used in the simulation as interaction potentials. Bond connectivity is ensured by an additional FENE potential. The superposition of the LJ and FENE-potential defines the bond-length of the polymers, $r_{\rm bond} = 0.9609\sigma$. The minimum of the (full) LJ-potential is at $r_{\rm min} = 2^{1/6} = 1.1225$. Potentials are shifted to ensure continuity at the cut-off radius $r_{\rm cut}$ which is either equal to $r_{\rm min}$ (thick lines) or $r_{\rm cut} = 2r_{\rm min}$ (thin lines).

where r_{ij} denotes the distance of particles i, j and $C(r_{\rm cut})$ ensures continuity of the potential at the cut-off radius $r_{\rm cut}$. ε defines the energy scale and σ the length scale of the system. The LJ-parameters ε and σ can be chosen independently for fluid and wall atoms, but were set to unity in all presented studies. There are two commonly employed choices for $r_{\rm cut}$. First, setting $r_{\rm cut} = 2^{1/6}\sigma$, the minimum of the LJ-potential. It corresponds to a purely repulsive interaction, or physically to a substance in good solvent conditions. Second, $r_{\rm cut} = 2 \times 2^{1/6}\sigma$, including a great part of the attractive interaction.

For the bond-potential in polymers, the widely used FENE (finitely extensible nonlinear elastic) potential introduced by KREMER and GREST [38] was added to the LJ-interaction between neighboring monomers. It diverges for $r_{ij} \to R_0$ and restricts the separation of monomers,

$$U_{\rm FENE}(r_{ij}) = -\frac{k}{2}R_0^2 \ln\left[1 - \left(\frac{r_{ij}}{R_0}\right)^2\right] \ . \tag{2.2}$$

Throughout the present work $R_{\rm ch} = 1.5\sigma$ and $k_{\rm ch} = 30\varepsilon/\sigma^2$ [38]. Typical values for hydrocarbons are $\varepsilon \approx 30$meV, $\sigma \approx 0.5$nm, resulting in a typical time scale of $t_{\rm LJ} \approx 3$ps [38]. The different potentials are plotted in Fig. 2.1, where one can read off the bond-length $r_{\rm bond} = 0.96909$.

An advantage of these generic choices for the interactions is that the results will be of general validity and are less likely to reflect special properties of the potentials.

2.4 Walls Models

An important ingredient of the simulation of frictional behavior we will discuss in chapter 5 are the interfaces. The first simplification consists of assuming flat, crystalline walls. This is an important simplification, as most real, macroscopic contacts are more irregular and not perfectly flat in general, but rather curved. We will elaborate on this point in more detail in section 5.1. Here, we focus on presenting the features of the simulation program.

Ideally, one would simulate walls made up of thick slabs containing several layers of atoms. Because only the outermost layers of the walls interact with the lubricant, contributions of atoms deeper inside are much less important and only one layer of wall atoms was used in the simulation.

2.4.1 Wall Geometry

Walls were modeled as hexagonal closed packed lattices (the (1,1,1)-plane of an fcc crystal) with an LJ-atom sitting at each lattice point and lattice constant d_{nn} as shown in Fig. 5.2 (where $d_{\mathrm{nn}} = 1$). The two walls could be set up identically leading to a commensurate geometry or with the upper wall rotated by $\theta = 90°$ resulting in an incommensurate setup, as the ratios of the length scales in each direction are irrational, as seen most easily for the coordinate directions, where it is $\sqrt{3}$. If the ratio is rational but not unity, one speaks of quasi-incommensurate walls.

The nearest neighbor spacing d_{nn} in the walls is 1.20914σ unless noted otherwise. This choice for d_{nn} is incommensurate with other length scales in the system and thus more generic than the minimum distance of the LJ potential r_{min}. We will see in section 5.5.2 how important this point is. In addition, a larger value for d_{nn} enhances the effects of surface corrugation in which we are particularly interested.

The use of periodic boundary conditions in the wall plane required a slight distortion of the perfect hexagonal geometry in order to obtain two quadratic walls. Therefore, walls were not perfectly incommensurate anymore, but quasi-incommensurate (as every setup realized with finite number precision, strictly speaking).

A rectangular wall unit cell consists of two atoms, at positions $(0, 0)$ and $(d_{\mathrm{nn}}/2, \sqrt{3}\, d_{\mathrm{nn}}/2)$, cf. Fig. 5.2. By choosing the ratio of the wall unit cells in x and y close to the ideal value $\sqrt{3}$ this distortion was minimized. Neither in our present work, nor in previous simulations using the same wall setup any effect of this adjustment could be seen [31, 39, 40]. We did not use other relative wall rotations, as it was found in Refs. [31, 40] that the influence of the rotation angle is weak if it exceeds $\approx 5°$ and that all these walls geometries can be considered incommensurate. This is consistent with our analysis of adiabatic trajectories using similar interaction potentials in section 5.4.2.3. We note that when two solids in an experiment come in contact they will most likely be incommensurate, as it would take utmost care to have two identical defect-free crystals and orient them perfectly. As detailed calculations show, elastic deformations do not generally alter this argument provided the solids are treated as 3-dimensional objects [41, 42].

2.4.2 Wall Atom Coupling

For the coupling of the wall atoms to the lattice sites we have different choices. The obviously most simple one is to fix them at the lattice positions. This corresponds to very hard and inelastic materials and is consuming the least computing time. We used this model preferably for our studies in chapter 5 where we verified that results remained valid if wall atoms were elastically coupled.

The wall model implemented in the simulation is more flexible then fixed wall atoms, as it allows for two different types of coupling to the lattice sites. We discern two potentials. First, the harmonic (or TOMLINSON) coupling,

$$U_{\mathrm{T}} = \frac{1}{2} k_{\mathrm{T}} \left(\mathbf{r}_i - \mathbf{r}_i^{\mathrm{eq}}\right)^2 \ , \tag{2.3}$$

where the actual position of atom i is at $\mathbf{r}_i$ and the equilibrium (lattice) position at $\mathbf{r}_i^{\mathrm{eq}}$. This spring-like potential keeps the wall atoms attached to their equilibrium sites. Second, we have a potential that prefers to keep relative positions close to the equilibrium values. This is the so-called FRENKEL-KONTOROVA interaction,

$$U_{\mathrm{FK}} = \frac{1}{2} k_{\mathrm{FK}} \sum_{j \in \mathrm{nn}(i)} \left(\mathbf{r}_i - \mathbf{r}_i^{\mathrm{eq}} - (\mathbf{r}_j - \mathbf{r}_j^{\mathrm{eq}})\right)^2 \ . \tag{2.4}$$

The sum runs over all nearest neighbors, i.e. 6 for the hexagonal lattice. This potential introduces long range elasticity in the walls and is a diagonalization of the more general 3×3 tensorial coupling of actual and equilibrium site differences. The full tensorial coupling was used in the more sophisticated wall model discussed in [43, 44] for realistic modeling of metal surfaces. Yet, our wall model gave qualitatively similar results which we will present in section B.

For adjusting to the same wall hardness, one can use $k_{\mathrm{T}} = k_{\mathrm{FK}}/6$ as there are six nearest neighbors. To get a feeling for the spring constants we compare to a molten LJ-fluid. According to the LINDEMANN criterion a crystal melts, when the thermal energy causes the atoms to vibrate by 10% of the lattice spacing which is of order σ, $\frac{1}{2}k_{\mathrm{T}}(0.1\sigma)^2 \approx \frac{1}{2}k_{\mathrm{B}}T$. For a typical temperature $T = 0.5$ one obtains $k_{\mathrm{T}} = 50$. This value for k_{T} leads thus to very soft walls.

In the simulation a variant of the potential U_{FK} is also implemented, which is proportional to the square of the moduli of the distances $|\mathbf{r}_i - \mathbf{r}_j| - |\mathbf{r}_i^{\mathrm{eq}} - \mathbf{r}_j^{\mathrm{eq}}|$. The difference is $2|\mathbf{r}_i - \mathbf{r}_j||\mathbf{r}_i^{\mathrm{eq}} - \mathbf{r}_j^{\mathrm{eq}}|(1 - \cos\varphi)$, where φ is the angle between $\mathbf{r}_i - \mathbf{r}_j$ and $\mathbf{r}_i^{\mathrm{eq}} - \mathbf{r}_j^{\mathrm{eq}}$, which becomes appreciately only if i has moved far out of the plane defined by its neighbors. The combination of both variants could be used to adjust the elasticity perpendicular to the wall-plane. (We call the first one "vector" and the latter "linear" FK potential).

2.5 Molecular Dynamics Techniques

For the numerical integration of NEWTON's equations of motion in our molecular dynamics simulation we implemented the GEAR predictor-corrector algorithm up to fifth order. Using it in second order leads to the velocity-VERLET algorithm. We found that we could use a time-step $\Delta t = 0.005$ for systems with polymers using

the fifth order algorithm which is significantly higher than the time-step $\Delta t = 0.002$ which was used for the same polymer model simulated with the velocity-VERLET algorithm and NOSÉ-HOOVER thermostat [13, 45].

For maintaining constant temperature, a stochastic thermostat, the so-called LANGEVIN thermostat was used. The implementation of the algorithm and the thermostat in the simulation program will be given in a form which resembles the routines implemented in the simulation code itself. For a discussion of the GEAR predictor-corrector algorithm see e.g. [37]. For acceleration of the time consuming force calculation standard binning and VERLET neighborhood list techniques were used [36, 46].

2.5.1 Predictor Step

Let $\mathbf{r}_i(t)$ be the position of particle i at time t. First, in the predictor step the positions and derivatives of the particle positions are predicted,

$$\frac{(\Delta t)^n}{n!}\frac{\partial^n}{\partial t^n}\mathbf{r}_i(t+\Delta t) = \frac{(\Delta t)^n}{n!}\frac{\partial^n}{\partial t^n}\mathbf{r}_i(t) + \sum_{m=n+1}^{N_{\text{order}}} C^{\text{p}}_{n,m}\frac{(\Delta t)^m}{m!}\frac{\partial^m}{\partial t^m}\mathbf{r}_i(t)\ ,\ n = 0, \ldots, N_{\text{order}}\ , \tag{2.5}$$

where $C^{\text{p}}_{n,m}$ is the m-th predictor coefficient of n-th order. The predictor coefficients are just the TAYLOR expansion coefficients

$$C^{\text{p}}_{n,m} = \frac{m!}{n!(m-n)!}\ . \tag{2.6}$$

For efficiency, the powers of the time-step and faculties are saved on the arrays holding the derivatives of the positions. When configurations are written out, they are however divided out, to obtain the time-derivatives of the positions independently of the time-step. They can be used to propagate saved configurations with another time-step Δt.

2.5.2 Thermostat

In the thermostatting routine we first define the random force for time t. This random force is ideal white noise with no memory with a width σ^{r} chosen to satisfy the fluctuation-dissipation theorem [46].

We draw a uniform random force distribution $\mathbf{f}^{\text{r}}_i(t)$ as random numbers on $(-\sigma^{\text{r}}, \sigma^{\text{r}})$ in each dimension for each particle with width

$$\sigma^{\text{r}} = \sqrt{3}\sqrt{\frac{2Tmk_{\text{B}}\gamma}{\Delta t}} \tag{2.7}$$

The factor $\sqrt{3}$ is needed because we use a uniform, not a GAUSSian distribution. Uniform random numbers can be obtained more efficiently and because they do not contain rare, high values like the GAUSSian distribution, the algorithm is more stable. γ is the damping constant of the thermostat and set to 0.5 in almost all simulations unless noted otherwise. We explicitely write the argument of $\mathbf{f}^{\text{r}}_i(t)$ as time t, and not $t+\Delta t$ to stress that the random forces do not depend on the predicted positions and derivatives at time $t + \Delta t$ as the other forces, but are defined before.

Next, this randomly fluctuating force $\mathbf{f}_i^{\mathrm{r}}(t)$ is applied,

$$\mathbf{r}(t+\Delta t) = \mathbf{r}(t) + \frac{(\Delta t)^2}{2m_i}\mathbf{f}_i^{\mathrm{r}}(t)\ , \quad \frac{\partial}{\partial t}\mathbf{r}_i(t+\Delta t) = \frac{\partial}{\partial t}\mathbf{r}_i(t) + \frac{\Delta t}{m_i}\mathbf{f}_i^{\mathrm{r}}(t)\ , \tag{2.8}$$

where m_i is the mass of particle i. The random force acts on the position and the velocity of the particles only, not on the acceleration. The reason is that if we would do this, we would have a (direct, not via the *forces* computed in the present propagation step) contribution of the random forces in the corrector step, leading to a higher instability, so that the time-step Δt had to be chosen smaller.

Thermostatting can be done in different reference frames and sometimes it is not appropriate to thermostat in all spacial dimensions. We usually use the center of walls of the two walls (if present) as reference frame and thermostat in all dimensions. This choice is symmetric with respect to the wall motion. Other choices will be mentioned wherever there is fit to do so. When wall particles are at fixed positions only the fluid can be thermostatted. If wall atoms were elastically bound, we usually only thermostatted the walls.

The friction force on the particles is the first deterministic contribution to the force at the propagated time $\mathbf{f}_i^{\mathrm{d}}(t+\Delta t)$,

$$\mathbf{f}_i^{\mathrm{d}}(t+\Delta t) = -m\gamma\frac{\partial}{\partial t}\mathbf{r}_i(t+\Delta t)\ . \tag{2.9}$$

The random contribution in the velocities from the previous predictor step enter here. Friction force and random force counterbalance in a way that in equilibrium detailed balance is fulfilled.

We note in passing that the quality of the random numbers is not crucial in MD simulations as there are so many particles in the simulation that the system would be ergodic anyways. For generation of random numbers we use the "ranlux" (pseudo-) random number generator [47, 48] for which a Fortran90 module was publicly available. An advantage of this random number generator is that the "quality level" (or "luxury level") can be set. It has been tested extensively and is believed to pass any known test for luxury level 2 (in a range from 0 to 4) and higher. We use level 2 in our simulations. In particular a test using random walks has been performed in [49] and an investigation of the correlation of different sequences of random numbers with regard to MD-simulations was done in [50] where ranlux passed and the commonly used R250 failed.

2.5.3 Interactions

Interaction forces are calculated using the positions from the predictor step (including the random contribution) and added to the deterministic force. In our model these forces are two body potential forces $\mathbf{f}_i^{\mathrm{c}}$ which depend only on the position of the particles,

$$\mathbf{f}_i^{\mathrm{d}}(t+\Delta t) = \mathbf{f}_i^{\mathrm{d}}(t+\Delta t) + \mathbf{f}_i^{\mathrm{c}}\left(\{\mathbf{r}_i(t+\Delta t),\ i=1,\ldots,N_{\mathrm{particles}}\}\right) \tag{2.10}$$

	$N_{\text{order}} = 2$	$N_{\text{order}} = 3$	$N_{\text{order}} = 4$	$N_{\text{order}} = 5$
C_0^{c}	0	1/6	19/120	3/16
C_1^{c}	1	5/6	3/4	251/360
C_2^{c}	1	1	1	1
C_3^{c}	–	1/3	1/2	11/18
C_4^{c}	–	–	1/12	1/6
C_5^{c}	–	–	–	1/60

Table 2.1: Corrector coefficients for the GEAR predictor corrector algorithm.

2.5.4 Corrector Step

The most difficult part of the GEAR predictor corrector algorithm is the correction step after which a propagation step is completed. In this step, positions and derivatives are corrected proportionally to the difference between computed and predicted force (or equivalently acceleration),

$$\frac{(\Delta t)^n}{n!}\frac{\partial^n}{\partial t^n}\mathbf{r}_i(t+\Delta t) = \frac{(\Delta t)^n}{n!}\frac{\partial^n}{\partial t^n}\mathbf{r}_i(t+\Delta t) + C_n^{\text{c}}\frac{(\Delta t)^2}{2}\left(\frac{\mathbf{f}_i^{\text{d}}(t+\Delta t)}{m} - \frac{\partial^2}{\partial t^2}\mathbf{r}_i(t+\Delta t)\right). \tag{2.11}$$

Here, C_n^{c} is the n-th corrector coefficient. The correction does not depend directly on the random forces. Thus, mainly the physical trajectory is corrected and stability is improved. By correcting this way, the effect of the random forces is weakened as compared to treating them like deterministic forces. This resulted in a system temperature (measured by the kinetic energy of the fluid particles) which was about $10^{-4}\,T$ too low in equilibrium.

GEAR devised the correction coefficients to minimize errors such that the local truncation error is of $\mathcal{O}(\Delta t^{N_{\text{order}}+1})$ for linear differential equations [51]. For second-order differential equations the global error is then $\mathcal{O}(\Delta t^{N_{\text{order}}-1})$. We list the corrector coefficients in table 2.5.4 following Ref. [37].

For monitoring stability of the integration algorithm we found empirically that a difference in the deviation with magnitude

$$\left|\frac{(\Delta t)^2}{2}\left(\frac{\mathbf{f}_i^{\text{d}}(t+\Delta t)}{m} - \frac{\partial^2}{\partial t^2}\mathbf{r}_i(t+\Delta t)\right)\right| > 0.001 \tag{2.12}$$

was an indication for the onset of instability. For typical simulation runs this value was $\lesssim 0.0004$.

2.5.5 Pressure Tensor

For controlling pressure, the pressure tensor of the complete system [52, 53]

$$P^{ab}(t) \stackrel{\text{def}}{=} \frac{1}{V}\left[\sum_{\text{fluid atoms } i} m_i v_i^a(t) v_i^b(t) + \sum_{\text{interacting pairs } ij} F_{ij}^a(t) r_{ij}^b(t)\right] \tag{2.13}$$

could be recorded during a simulation. $v_i(t)$ are the peculiar velocities, i.e. measured in the center of mass of the fluid. For conservative forces, P^{ab} is symmetric. For

elastic walls the normalization to the volume is difficult, and the average z-positions of the wall were used for obtaining the total volume of the system containing lines connecting interacting particles ("force lines"). For a discussion of the forces lines see [54]. We note that an apparent problem in deriving the pressure tensor in systems with periodic boundary conditions can be avoided in a derivation using momentum fluxes and imaginary planes for pair-wise interactions [37, App. B]. The pressure tensor could be used for viscosity measurements like in [55].

2.6 Initializing Configurations

At the beginning of a simulation, a starting configuration has to be created. While in three dimensional space usually this does not pose any severe problem, the situation changes for restricted geometries. In 3-d, one could just start with a large simulation box, create random walk polymers and then compress the system. With this method it is straightforward to create a dense bulk melt or a dense thick film, however it is not possible to similarly create a sub-monolayer of a flat and dense polymer film, as upon compression chain crossings occur due to chains which lie over each other. This effect can happen even for low densities, yet for a single chain which crosses itself. Configurations where chains lie over each other are not equilibrium configurations and equilibrate very slowly. Hence this situation is to be avoided and the simulation program provides subroutines for detecting them (but this alone does not help to completely avoid the problem). In section 5.6 we will demonstrate the dramatic effect of crossed chains on the wall motion.

The touchstone for the creation of a thin polymer film is the single chain in two dimensions. If one chain can be created, several chains can be fit in the system up to very high densities by creating scaled down chains which are expanded in the course of the simulation (or the system size is continously decreased). The method we use consists in starting the simulation with monomers with reduced LJ-radius σ which is linearly increased during the simulation. When σ is increased, particles can feel a very strong repulsion leading to an instability of the integration algorithm. The LJ interaction is therefore cut off at a maximum value or, equivalently, the particle distance is cut off at a distance of closest allowed approach. This closest approach is gradually reduced to zero in an initial relaxation phase of the simulation. In addition, overstretched bonds (bond-length $\geq R_0$) are 'repaired' by placing monomers closer to each other instantaneously. These two mechanisms make the program more tolerant towards high energy configurations during the initialization phase.

We now comment on the construction of polymers themselves, where we will focus on the "recoil-growth" algorithm with which we could construct chains up to length $N = 2048$.

2.6.1 CBMC and Recoil Growth Algorithm

The fact that it becomes increasingly difficult to construct of a polymer of growing length N, especially in dense systems, is known as "attrition problem" [56]. It originates from the fact that a monomer which we want to add to a polymer can hit a position already occupied. This happens in two and three dimensions alike.

There is an important difference, however, which is related to the POLYA-problem in mathematics. It states that two dimensional random walks are "recurrent" , i.e. will surely return to the start point. On the contrary, three dimensional random walks are not recurrent (they are "transient") and will not return to the origin. This shows that the attrition problem is more severe in two dimensions than in three.

The most basic approach for constructing a polymer is a self avoiding random walk. This works not very well even for 3-d systems as the probability for the chain construction rate falls exponentially, $p = \exp[-\lambda(d)N]$ [34, 56]. The random walk is 'blind' for excluded volume, so to say. A considerable improvement can be achieved by the Configurational Bias Monte Carlo scheme (CBMC) [36, Chap. 13] based on a scheme proposed by ROSENBLUTH and ROSENBLUTH [57] with an additional bias introduced to recover the correct distribution in a Monte Carlo scheme. At each step during the construction of the polymer the interaction energies of N_{choice} trial directions for setting the next bond are calculated and one direction is chosen according to its BOLTZMANN weight, so that low energy configurations are strongly favored.

An implicit assumption of the CBMC algorithm is that it does not matter at which position we are for adding the next monomer. At some positions it is however not possible to grow the chain any further, namely when the chain has grown into a "dead-end" as depicted in figure 2.2 where after growing from position 3 to 3´ there is no way to go anymore for a bond, the particle at 3´ is completely blocked. In 3-d this situation is relatively rare and CBMC has become a standard simulation technique if the density is not too high in which case local propagation is to be preferred. As in 2-d it becomes more and more likely for a chain to see itself and consequently trap itself (the chain can just not not grow 'over' itself as in 3-d) CBMC will fail even for single chains in 2-d and we found that in our implementation we could not construct chains longer than $N \approx 100$ repeat units within an acceptable time. We used the CBMC for initializing short polymers which were further equilibrated by MD simulation runs for the simulations in chapter 5.

The reason for this behavior is the 'shortsightedness' of the CBMC method, it can not recoil back to position 3 and try another direction (4 in our example) after it becomes trapped in 3´. Curing this ill-behavior was the motivation for the recoil-growth (RG) algorithm [34, 35]. For presenting it we follow the more pedagogical description in [36, Sect. 13.7]. Because this method is a generalization of the CBMC method we abstain from describing CBMC separately. A good description of the CBMC method can be found in [36, Sect. 13.2]. Results for single chains can be found in chapter A.

The basic idea of the RG algorithm is antique and is similar to the problem of systematically finding a way out of a maze. At a crossing one takes the leftmost turn and makes a mark, if one has entered a dead-end one goes back to the last crossing where not all branches are already tried out and takes the leftmost branch still not tried. If there is a way out it is found surely, unless one dies from starvation first (or there are loops in the maze, in this case any recursive algorithm can end up in an infinite loop [58]). This is pretty much the recoil-growth algorithm. We may identify the way out of a maze with the growing of a chain, each step in a certain direction is a bond created, and if the algorithms has to go back too far it discards the chain.

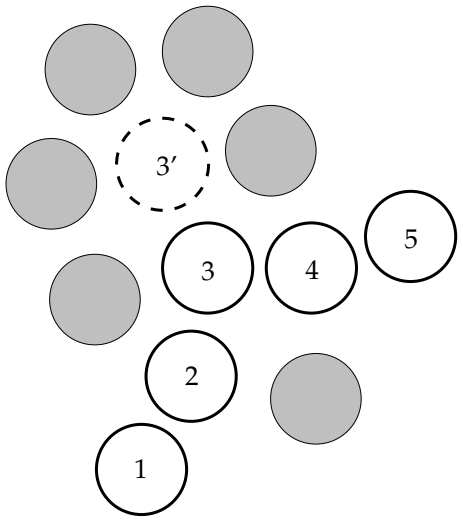

Figure 2.2: When conventional CBMC fails: When trying to grow the polymer past position 3´ all directions are blocked and CBMC fails. The RG scheme on the contrary retracts to position 3 and can find 4 and continue growing the full chain.

The difficulty lies in the computation of the correct weight of the conformation.

The following steps are carried out for the generation of a new chain:

1. Place the first monomer 1 at the random trial position $\mathbf{t}_1$. Compute the probability
$$p_1^{\text{open}}(\mathbf{t}_1) = \min\{1, \exp[-\beta u(\mathbf{t}_1)]\} \tag{2.14}$$
that this position with interaction energy $u(\mathbf{t}_1)$ is "open". If it is "open", i.e. does not have a very high energy, continue with step 2.

2. Generate a trial position $\mathbf{t}_{i+1}$ from monomer i, compute $p_i^{\text{open}}(\mathbf{t}_i)$ and decide if this is an open direction. If not make another try unless N_{choice} trial positions were discarded, because they were "closed". As soon as an open direction is found, proceed to step 3.

   If no open trial direction is found, make a recoil step and retract to monomer $i-1$ (if existing) and try to find an open position $\mathbf{t}_{i-1}$. If no open trial direction is found at monomer $i-1$ recoil back to $i-2$, and so on. A maximal number of N_{recoil} recoil steps is allowed. If this number is exceeded we restart chain building from step 1.

3. Add monomer $i - N_{\text{recoil}}$ 'permanently' to the chain, as recoiling can now only reach $i - N_{\text{recoil}} + 1$.

4. Repeat step 2 and 3 unless the complete chain of length N has been grown.

Now a polymer has been built and we have to decide on its acceptance. A chain is accepted if it has low energy and high weight, i.e. has a very likely conformation. Compared to the CBMC scheme the RG algorithm is very efficient, as is does the weight calculation only if a complete chain has been constructed. The weight for the new chain is obtained as follows:

1. At monomer i we have found in the construction part that at least one trial direction out of $k_{\rm checked} \leq N_{\rm choice}$ did not 'die' within the next $N_{\rm recoil}$ steps. We now test the remaining $k_{\rm rest} = N_{\rm choice} - k_{\rm checked}$ directions to find out how many "feelers" of length $\min(N_{\rm recoil}, N-i)$ can be grown (at the chain end we stop, thus the min() condition). Let the number of directions where such a feeler can be grown be $1 \leq m_i(new) \leq 1 + k_{\rm rest}$. It can be shown (and this is the most difficult part of the RG algorithm) that monomer i contributes the factor
$$w_i(new) = \frac{m_i(new)}{p_i^{\rm open}} \ , \tag{2.15}$$
to the total weight of the chain.

2. Repeat step 1 for the whole chain. At the chain end, no feelers can be grown and one just sets $m_N(new) = 1$ as this is the minimal value for each i.

3. The weight for the whole new chain is given by
$$W(new) = \prod_{i=1}^{N} w_i(new) = \prod_{i=1}^{N} \frac{m_i(new)}{p_i^{\rm open}(new)} \ . \tag{2.16}$$

For the weight calculation of the old chain we have to generate $N_{\rm choice} - 1$ trial directions at each monomer but the last and proceed exactly as for the new chain by counting the number of feelers to obtain
$$W(old) = \prod_{i=1}^{N} w_i(old) = \prod_{i=1}^{N} \frac{m_i(old)}{p_i^{\rm open}(old)} \ . \tag{2.17}$$
Note that the weight for the old chain depends on the trial positions chosen each time the weight is calculated, but the average value reaches a constant.

Finally, the new chain is accepted with probability
$$P_{\rm acc}(old \to new) = \min\left\{1, \frac{W(new)\exp[-\beta U(new)]}{W(old)\exp[-\beta U(old)]}\right\} \tag{2.18}$$
which fulfills detailed balance. U denotes to total internal energy of the chains.

Some remarks are in order. The criterion of "openness", $p_i^{\rm open}$ can be chosen differently, it affects efficiency. Taking it to be the BOLTZMANN weight works well for LJ-interactions. The algorithm creates a random tree on the fly and searches through it until it finds a complete chain. Thus, it works with the smallest number of trial directions possible leading to high efficiency. The number of choices for trial positions $N_{\rm choice}$ need not be an integer, it can be chosen differently at each i with an average value $N_{\rm choice}$. Setting $N_{\rm recoil} = 1$ leads to a scheme which resembles CBMC. While we found that for 2-d systems $N_{\rm recoil}$ is not affecting efficiency very strongly, it is very sensitive on $N_{\rm choice}$ which had to be adjusted to 4 significant digits for our longest chains with $N = 2048$, as we show in Fig. A.2.

To our best knowledge this is the first time the recoil-growth algorithm was used for a study of 2-d chains.

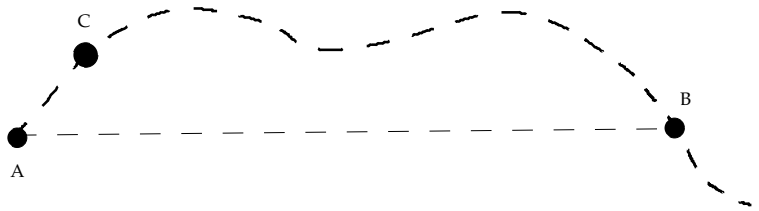

Figure 2.3: After measuring the distance $\overline{AB}$ one can calculate the number of monomers which can not interact with A by dividing $\overline{AB}$ by the maximal interaction range and skip this number of monomers, calculating again only for C.

2.6.2 Intra-Chain Energy Calculation

Every time a new trial position is defined, its potential energy must be computed, which includes the intra-chain energy. A straightforward approach would be to just loop over the $\frac{1}{2}N(N-1)$ (here N is the present length of the chain) possible pairs. A single chain is not very dense, so geometrical binning is not very efficient and has to be done dynamically when a chain is constructed, which is CPU-intensive. One can however use the topology of the chain to reduce the prefactor for the energy calculation. Let us turn to Fig. 2.3. For calculating the interaction energy between monomers at positions A and B we need to know the distance $\overline{AB}$. If $\overline{AB}$ is larger than the maximal interaction range r_{cut} then we know that the next $[\overline{AB}/r_{\text{cut}}]$ monomers along the chain backbone in direction A can not interact with A, as a straight line is the shortest connection between A and B. We so obtain the next possible interaction candidate, C. In our figure, we can only skip a few monomers at C, but still some. By employing this method for calculating the intra-chain energy, a factor 10 in computing time could be saved for single chains with $N = 1024$.

Chapter 3

Statics and Dynamics of Polymer Melts

3.1 Introduction

Polymers are macromolecules each consisting of a chain of repeat units linked together by chemical bonds [1]. They are of great industrial and biological importance – one may only think of the vast number of products made of plastics which surround us and of the DNA molecules in cells. Their properties are not only determined by their chemical characteristics, but also by their static structure and dynamic properties.

Many polymers are found in the amorphous or glassy state, i.e. they are solid but retain a liquid-like structure when they are cooled to solidification. As opposed to a crystal, a liquid is characterized by the absence of long range periodic ordering. When a glass forming substance is cooled, the relaxation time grows by many orders of magnitude in a small temperature range near the glass transition temperature [4, 5, 8]. This effect is used to define the experimental glass temperature T_{g} at which the viscosity reaches a value of 10^{14}Poise corresponding to relaxation times of days.

The microscopic understanding of the nature of the glass transition is a challenging problem in contemporary condensed matter physics. A very successful approach is the mode-coupling theory (MCT), see e.g. [6–8] and references therein. MCT predicts that due to non-linear coupling of the particles, there exists a critical temperature T_{c} above the experimental glass temperature T_{g} where the dynamics changes qualitatively. At T_{c} complete structural arrest is predicted in the most basic version of MCT, the "ideal MCT". Sufficiently close to T_{c} the dynamics is determined by the mutual blocking of the particle by the shell of its nearest neighbors, the so-called "cage-effect". In the 'extended' form of MCT, total structural arrest is circumvented by the inclusion of alternative relaxation channels [59–61]. These corrections become important only very close to T_{c}. Originally developed for simple liquids, the encouraging success of MCT in describing experimental and computer simulation data [7] stimulated the extension to the inclusion of orientational degrees of freedom [62–64].

In previous studies [14–16] the ideal MCT of simple liquids was found to describe well our simulation data for polymers of length 10 repeat units close to, but not too close to the critical MCT temperature T_{c}. The quality of the description provided by

the idealized MCT for simple liquids is insofar surprising, as that in a simple liquid each particle can move independently, whereas the presence of bonds in polymer melts gives rise to specific chain effects due to the steric constraints.

Recently, an extension of MCT for polymers was proposed by CHONG and FUCHS [19] which included full information on intra-chain conformations. This theoretical development stimulated the investigation of the static structure under full consideration of the polymeric character of the system, i.e. the dependence of the structure on the place a monomer has on the polymer backbone. This allowed for the test of the validity of the assumptions made in Ref. [19]. Furthermore, we calculated the static and dynamic structure of our polymer melt model including the centers of mass (CM) of the polymers. We are therefore able to discuss the role of the CM-motion in our system.

Very often in liquid theory and MCT, approximations are used which neglect three particle correlations. In light of recent MCT investigations which included the three particle direct correlation function c_3 [65, 66] and reported significant improvements of the descriptions with respect to molecular dynamics simulation results for molecular liquids, we also calculated the three particle structure factors, from which c_3 can be obtained.

3.2 Model Details

This section briefly describes the polymer model used in the simulations. More details may be found in [12, 17]. We used a well established "bead-spring" polymer model first proposed by KREMER and GREST where monomers interact via LENNARD-JONES (LJ) interaction and chain connectivity is assured by a FENE potential as defined in section 2.3. The number of monomers (repeat units) was $N = 10$ in all simulations. In the following all quantities are given in Lennard-Jones units. That is, distance, temperature and time are measured in units of σ, ϵ/k_{B} ($k_{\mathrm{B}} =$ BOLTZMANN's constant), and $(m\sigma^2/\epsilon)^{1/2}$, respectively. The monomers mass m is set to unity.

The LJ-potential included a part of the long range interaction, with a cut-off radius $r_{\mathrm{cut}} = 2r_{\mathrm{min}}$. Here, $r_{\mathrm{min}} = 2^{1/6}\sigma = 1.1225\sigma$ is the minimum of the (full) LJ- potential. The FENE-potential parameters were set to $R_0 = 1.5$ and $k = 30$. The superposition of the LJ- and FENE potentials leads to a steep effective bond potential with a sharp minimum at $r_{\mathrm{bond}} = 0.9606$, cf. Fig. 2.1. This prevents bonds from crossing each other in the course of the simulation.

The length scale r_{bond} has a further consequence. If there was no bond potential, the monomer fluid would crystallize at low temperature, the lattice spacing being close to r_{min}. The bond potential locally distorts the regular arrangement of the monomers. It favors the inter-monomer distance r_{bond} which is incompatible with r_{min}. Thus, the competition between r_{bond} and r_{min} hinders crystallization.

However, this competition alone is not sufficient to preclude crystallization [67–69]. The chains should also be flexible. This has recently been pointed out by simulations of a bead-spring model in which large bond angles are favored by a bending potential [68, 69]. This leads to an increase of chain stiffness at low temperatures. The interplay of stiffness and excluded volume interactions suffices to induce

crystallization from the melt. Contrary to that, the chains of our model are flexible. In the whole temperature range studied, the end-to-end distance ($R_e^2 \simeq 12.3$) and the radius of gyration ($R_g^2 \simeq 2.09$) are almost constant, and the collective static structure factor is typical of an amorphous material [15, 70].

The simulations were performed in two steps [12, 17, 67]: First, the volume of the simulation box was determined in a constant pressure simulation at $p = 1$. Then, this volume was fixed and the simulations were continued in the canonical ensemble (NOSÉ-HOOVER thermostat). Periodic boundary conditions were applied in all three spatial directions of the cubic simulation box. The number of polymers was in the range 100 – 120. We simulated at $T = 0.46$, 0.47, 0.48, 0.50, 0.52, 0.55, 0.6, 0.65, 0.7 and 1. These temperatures correspond to monomer densities ranging between $1.04 \geq \rho \geq 0.91$. The critical temperature of MCT was estimated to be $T_c = 0.450 \pm 0.005$ [16, 45]. The temperature range in which simulations were done thus covers temperatures very close to T_c up to temperatures were all features of a glassy liquid are absent.

3.3 Basic Notations

In the following, let $\mathbf{r}_i^a(t)$ denote the position of the ath monomer in the ith chain at time t. We may drop the explicit time dependence for static quantities. Here, $a = 1, \ldots, N$ is the monomer index and $i = 1, \ldots, n$ the chain index. Hence we have $M = nN$ monomers in the melt of volume V. Furthermore, let ρ_n and ρ_m denote the chain and monomer number densities, defined by

$$\rho_n \stackrel{\text{def}}{=} \frac{n}{V} \quad \text{and} \quad \rho_m \stackrel{\text{def}}{=} \frac{M}{V} \,. \tag{3.1}$$

The position of the center of mass of the ith chain at time t is written

$$\mathbf{R}_i(t) \stackrel{\text{def}}{=} \frac{1}{N} \sum_{a=1}^{N} \mathbf{r}_i^a(t) \,. \tag{3.2}$$

The (squared) radius of gyration R_g^2 and (squared) end-to-end distance R_e^2 are in this notation

$$R_g^2 \stackrel{\text{def}}{=} \frac{1}{M} \Big\langle \sum_{i=1}^{n} \sum_{a=1}^{N} \left(\mathbf{r}_i^a - \mathbf{R}_i\right)^2 \Big\rangle \,, \tag{3.3}$$

$$R_g^2 \stackrel{\text{def}}{=} \frac{1}{n} \Big\langle \sum_{i=1}^{n} \left(\mathbf{r}_i^1 - \mathbf{r}_i^N\right)^2 \Big\rangle \,. \tag{3.4}$$

We introduce the FOURIER-transformed monomer densities at wave vector $\mathbf{q}$,

$$\rho_i^a(\mathbf{q},t) \stackrel{\text{def}}{=} \exp\left[\mathrm{i}\mathbf{q}\cdot\mathbf{r}_i^a(t)\right] \ , \tag{3.5}$$

$$\rho^{\mathrm{p}}(\mathbf{q},t) \stackrel{\text{def}}{=} \sum_{a=1}^{N} \exp\left[\mathrm{i}\mathbf{q}\cdot\mathbf{r}_i^a(t)\right] \ , \tag{3.6}$$

$$\rho(\mathbf{q},t) \stackrel{\text{def}}{=} \sum_{i=1}^{n}\sum_{a=1}^{N} \exp\left[\mathrm{i}\mathbf{q}\cdot\mathbf{r}_i^a(t)\right] \ , \tag{3.7}$$

$$\rho^{\mathrm{C}}(\mathbf{q},t) \stackrel{\text{def}}{=} \sum_{i=1}^{N} \exp\left[\mathrm{i}\mathbf{q}\cdot\mathbf{R}_i(t)\right] \ . \tag{3.8}$$

In our notation we use a superscript "p" for indicating polymer quantities, the subscript "s" stands for self, and "C" indicates the center of mass. Upper indices a and b are monomer indices along the backbone of the polymer. A subscript "A" will be used to collect all other dependencies except time.

3.4 Density Correlators

The density correlators or structure factors are often used for the description of the structure and the dynamics of a liquid. We can now list the various density correlators which result from the coupling of the different densities. Here, $\langle\cdot\rangle$ denotes the canonical average, i.e. the mean of all independent configurations simulated. We note that in a homogeneous and isotropic system, all two-particle quantities like the density correlators depend on the distance or equivalently on the length of the reciprocal vector only.

We start with the monomer separated structure factor, which is the sum of the intra-polymer and the distinct contribution,

$$S^{ab}(q,t) \stackrel{\text{def}}{=} \left\langle \rho^a(\mathbf{q},t)\rho^b(\mathbf{q},0)^*\right\rangle = S_{\mathrm{s}}^{ab}(q,t) + S_{\mathrm{d}}^{ab}(q,t) \ . \tag{3.9}$$

where we defined the monomer resolved (or separated) structure factor of a polymer

$$w^{ab}(q,t) \equiv S_{\mathrm{s}}^{ab}(q,t) \stackrel{\text{def}}{=} \left\langle \rho_{\mathrm{s}}^a(\mathbf{q},t)\rho_{\mathrm{s}}^b(\mathbf{q},0)^*\right\rangle = \frac{1}{n}\Big\langle \sum_{i=1}^{n} \exp\big\{\mathrm{i}\mathbf{q}\cdot[\mathbf{r}_i^a(t)-\mathbf{r}_i^b(0)]\big\}\Big\rangle \ , \tag{3.10}$$

and the monomer separated structure factor of the melt from distinct chains

$$S_{\mathrm{d}}^{ab}(q,t) \stackrel{\text{def}}{=} \frac{1}{n}\Big\langle \sum_{i\neq j}^{n} \exp\big\{\mathrm{i}\mathbf{q}\cdot[\mathbf{r}_i^a(t)-\mathbf{r}_j^b(0)]\big\}\Big\rangle \ . \tag{3.11}$$

Averaging over the monomer pairs (a,b) leads to the intermediate scattering function of the melt,

$$S(q,t) \stackrel{\text{def}}{=} \langle\rho(\mathbf{q},t)\rho(\mathbf{q},0)^*\rangle = \frac{1}{N}\sum_{a,b=1}^{N} S^{ab}(q,t) = w(q,t) + S_{\mathrm{d}}(q,t) \ , \tag{3.12}$$

where $S(q)$ is referred to as the static structure factor of the melt.

The structure factor of the chain is given by

$$w(q,t) \stackrel{\text{def}}{=} \langle \rho^{\mathrm{p}}(\mathbf{q},t)^* \rho^{\mathrm{p}}(\mathbf{q},t) \rangle = \frac{1}{N} \sum_{a,b=1}^{N} w^{ab}(q,t) \; . \tag{3.13}$$

Similarly, one obtains the distinct part of $S(q,t)$,

$$S_{\mathrm{d}}(q,t) \stackrel{\text{def}}{=} \frac{1}{N} \sum_{a,b=1}^{N} S_{\mathrm{d}}^{ab}(q,t) \; . \tag{3.14}$$

Furthermore, we can analogously define the structure factor of the centers of mass of the polymers,

$$S^{\mathrm{C}}(q,t) \stackrel{\text{def}}{=} \langle \rho^{\mathrm{C}}(q,t) \rho^{\mathrm{C}}(q,0)^* \rangle = \frac{1}{n} \Big\langle \sum_{i,j=1}^{n} \exp\big\{ \mathrm{i}\mathbf{q} \cdot [\mathbf{R}_i(t) - \mathbf{R}_j(0)] \big\} \Big\rangle \; . \tag{3.15}$$

Coupling the monomer and center of mass densities leads to the structure factor of the ath monomer relative to the center of mass. We separate again in self (intra-chain) and distinct contributions,

$$S_{\mathrm{s}}^{a,\mathrm{C}}(q,t) \stackrel{\text{def}}{=} \langle \rho_i^a(q,t) \rho^{\mathrm{C}}(q,0)^* \rangle = \frac{1}{n} \Big\langle \sum_{i=1}^{n} \exp\big\{ \mathrm{i}\mathbf{q} \cdot [\mathbf{r}_i^a(t) - \mathbf{R}_i(0)] \big\} \Big\rangle \; . \tag{3.16}$$

When summing over all monomers we obtain (note that there is no prefactor $1/N$ here)

$$S_{\mathrm{s}}^{\mathrm{m,C}}(q) \stackrel{\text{def}}{=} \sum_{a=1}^{N} S_{\mathrm{s}}^{a,\mathrm{C}}(q,t) \; . \tag{3.17}$$

Likewise,

$$S_{\mathrm{d}}^{a,\mathrm{C}}(q,t) \stackrel{\text{def}}{=} \frac{1}{n} \Big\langle \sum_{i \neq j}^{n} \exp\big\{ \mathrm{i}\mathbf{q} \cdot [\mathbf{r}_i^a(t) - \mathbf{R}_j(0)] \big\} \Big\rangle \; , \tag{3.18}$$

and the corresponding

$$S_{\mathrm{d}}^{\mathrm{m,C}}(q) \stackrel{\text{def}}{=} \sum_{a=1}^{N} S_{\mathrm{d}}^{a,\mathrm{C}}(q,t) \; . \tag{3.19}$$

The sum of both contributions is

$$\begin{aligned} S^{a,\mathrm{C}}(q,t) &\stackrel{\text{def}}{=} S_{\mathrm{s}}^{a,\mathrm{C}}(q,t) + S_{\mathrm{d}}^{a,\mathrm{C}}(q,t) \; , \\ S^{\mathrm{m,C}}(q) &\stackrel{\text{def}}{=} S_{\mathrm{s}}^{\mathrm{m,C}}(q) + S_{\mathrm{d}}^{\mathrm{m,C}}(q) \; . \end{aligned} \tag{3.20}$$

By normalizing the time dependent quantities defined above by their static value at $t = 0$ we obtain their corresponding dynamic correlation functions ("correlators")

$$\phi^{\mathrm{x}}(q,t) \stackrel{\text{def}}{=} S^{\mathrm{x}}(q,t) / S^{\mathrm{x}}(q,0) \; , \tag{3.21}$$

where the superscript "x" stands for any of the aforementioned quantities.

3.5 Consideration of Periodic Boundary Conditions

Since all density correlators defined above are real, $\mathbf{q}$ can be chosen from say, the positive half reciprocal space. For all reciprocal vectors $\mathbf{q}$ and all dimensions α in which periodic boundary conditions are used, the periodic boundary conditions employed in the simulations require

$$\exp\{\mathrm{i}q_\alpha r_\alpha\} = \exp\{\mathrm{i}q_\alpha(r_\alpha \ + L_\alpha)\} \ , \tag{3.22}$$

where L_α is the linear dimension of the simulation volume the αth direction of the d-dimensional system. It follows that every reciprocal vector which is compliant with the periodic boundary conditions can be written in the following form

$$\mathbf{q} = \sum_{\alpha=1}^{d} \frac{2\pi}{L_\alpha} \hat{e}_\alpha n_\alpha \ , \quad n_\alpha \in \mathbb{Z} \ , \tag{3.23}$$

with $\hat{e}_\alpha$ denoting the unit vector in direction α.

When one is interested in static quantities, it is crucial for computational efficiency to calculate first $S^{ab}(q,0)$ and $S^{ab}_{\mathrm{s}}(q,0)$ and to obtain the difference $S^{ab}_{\mathrm{d}}(q,0)$ only afterwards, because the double sums in the former two quantities can be rewritten as a product of two simple sums. Additionally, the monomer resolved static quantities are symmetric in a and b, so that they need to be calculated for $a \leq b$ only (with $S^{aa}_{\mathrm{s}}(q,0) \equiv 1 \quad \forall a$).

To improve statistics it is necessary to average over a range of $q = |\mathbf{q}|$, i.e. to put all $\mathbf{q}$ with $q - \Delta q/2 \leq |\mathbf{q}| \leq q + \Delta q/2$ in a bin labeled q, provided that the average of the moduli $|\mathbf{q}|$ in this bin is close to the nominal value of the bin. When we quote a value $S^{\mathrm{x}}(q,t)$ we therefore actually mean the average

$$S^{\mathrm{x}}(q,t) = \frac{\sum_{\{\mathbf{q}|q-\Delta q/2 \leq |\mathbf{q}| \leq q+\Delta q/2\}} S^{\mathrm{x}}(\mathbf{q},t)}{\left|\{\mathbf{q}|q - \Delta q/2 \leq |\mathbf{q}| \leq q + \Delta q/2\}\right|} \ . \tag{3.24}$$

The bin width Δq was typically 0.2. It had to be chosen in general bigger for small q where it is more difficult to collect sufficiently many q-vectors. The number of q-vectors per bin was restricted to 1000 in order to save CPU-time. Dynamic scattering functions were computed for the following set of q-values: 0.6, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 6.5, 6.9, 7.15, 7.5, 8.0, 8.5, 9.5, 10.25, 11.0 12.8, 14.0, 15.0, 16.0, 17.5, 19.0, 20.5, and 22.0.

3.6 Direct Correlation Functions

Another important quantity for the description of liquids is the direct correlation function which we will introduce in real space for a simple liquid (see e.g. [53]).

The pair correlation function $g(r)$,

$$g(r) \stackrel{\mathrm{def}}{=} \frac{1}{\rho_{\mathrm{m}} N} \left\langle \sum_{i \neq j} \delta(r - |\mathbf{r}_i - \mathbf{r}_j|) \right\rangle \ , \tag{3.25}$$

is proportional to the probability to find another particle at distance r from a particle at the origin. For very long distances $g(r)$ approaches unity,

$$\lim_{r\to\infty} g(r) = \frac{N-1}{N} \overset{N\to\infty}{\to} 1 \; . \tag{3.26}$$

A directly related quantity is the total correlation function $h(r) \overset{\text{def}}{=} g(r) - 1$ which fluctuates around 0.

While $h(r)$ measures the integrated contribution of all particles at once, the direct correlation function $c(r)$ measures only the two-particle contribution. The Ornstein-Zernike equation connects $h(r)$ and $c(r)$ in the following way:

$$h(r) = c(r) + \rho_{\mathrm{m}} \int \mathrm{d}r' c(r') h(r - r') \; , \tag{3.27}$$

which expresses $h(r)$ as the two-particle contribution $c(r)$ and the integral over $c(r')$ weighed with h at the distance $r - r'$.

The Ornstein-Zernike equation in reciprocal space reads

$$h(q) = c(q) + \rho_{\mathrm{m}} c(q) h(q) = c(q)[1 + \rho_{\mathrm{m}} h(q)] \; , \tag{3.28}$$

by virtue of the folding theorem of Fourier theory. The structure factor can also be written in term of the total correlation function as

$$S(q) = 1 + \rho_{\mathrm{m}} h(q) \; . \tag{3.29}$$

In this equation the one comes from the self contribution of the particle and $\rho_{\mathrm{m}} h(q)$ from the other particles.

Combining the last two equations yields

$$\rho_{\mathrm{m}} c(q) = \frac{\rho_{\mathrm{m}} h(q)}{S(q)} = 1 - S(q)^{-1} \; . \tag{3.30}$$

For a polymeric system the generalized equations analogous to Eq. (3.28) and (3.29) including the site-dependence of molecular systems read [71],

$$h^{ab}(q) = \sum_{c,d} dw^{ac}(q) c^{cd}(q) \left[w^{db}(q) + \rho_{\mathrm{n}} h^{db}(q) \right] = \sum_{c,d} w^{ac}(q) c^{cd}(q) S^{db}(q) \; , \tag{3.31}$$

$$S^{ab}(q) = w^{ac}(q) + \rho_{\mathrm{n}} h^{ab}(q) \; . \tag{3.32}$$

Note that we use the chain number density ρ_{n}. The generalized site-site Ornstein-Zernike equation (3.31) is also known as RISM (Reference Site Interaction Model) equation; $c^{ab}(q)$ is the site-site direct correlation function.

We then obtain for the direct correlation function matrix

$$\rho_{\mathrm{n}} c^{ab}(q) = \left[w^{ab}(q) \right]^{-1} - \left[S^{ab}(q) \right]^{-1} \tag{3.33}$$

For later reference we note that the monomer pair averaged quantities satisfy

$$\begin{aligned} h(q) &= w(q) c(q) [w(q) + \rho_{\mathrm{n}} h(q)] \; , \\ S(q) &= w(q) + \rho_{\mathrm{n}} h(q) \; , \\ \rho_{\mathrm{n}} c(q) &= \frac{1}{w(q)} - \frac{1}{S(q)} \; . \end{aligned} \tag{3.34}$$

Given the structure factors $w^{ab}(q)$ and $S^{ab}(q)$, the calculation of the direct correlation function matrix $c^{ab}(q)$ can be done by matrix inversion for which we used the LAPACK library [72, 73].

3.7 Fast Calculation of Static Structure Factors

When calculating the static structure factors one needs $S^{\mathrm{x}}(\mathbf{q})$ for a significant subset of all allowed reciprocal lattice vectors $\mathbf{q}$ up to a maximal length q_{cut}. Excellent statistics was needed in order to have sufficiently good data even for the monomer-pair separated structure factors whose statistics is $1/N^2$ worse than the collective structure factor.

To get a better understanding of the number of operations involved in the computation of the structure factors we first note that by expansion of the real part in trigonometric terms we can write

$$\begin{aligned} MS(\mathbf{q}) &= \Big\langle \sum_{i,j=1}^{n} \sum_{a,b=1}^{N} \exp\big\{\mathrm{i}\mathbf{q}\cdot[\mathbf{r}_i^a - \mathbf{r}_j^b]\big\} \Big\rangle \\ &= \Big\langle \Big(\sum_{i=1}^{n} \sum_{a=1}^{N} \cos\{\mathbf{q}\cdot\mathbf{r}_i^a\} \Big)^2 + \Big(\sum_{i=1}^{n} \sum_{a=1}^{N} \sin\{\mathbf{q}\cdot\mathbf{r}_i^a\} \Big)^2 \Big\rangle \,. \end{aligned} \tag{3.35}$$

Here, we used S(q) as an example, but for the other structure factors we can write similar expressions. With this rewriting one immediately sees that the number of operations for the computation of S^{x} scales with the number of particles $nN = M$. As the structure factors are real, one can restrict the calculation of the say, positive half q-space.

A naive approach would be to loop over all vectors $\mathbf{q}$ in the half q-space and hence compute $\mathcal{O}(Mq_{\mathrm{cut}}^d)$ sines and cosines in a d-dimensional system.

One strategy for reducing the number of operations is to use only a subset of all $\mathbf{q}$ at a certain length q, provided that statistics is sufficient. For achieving low noise in the data this subset must be rather large, and consequently the savings are not very significant.

There is a faster way, however, and we are grateful to H. Meyer for pointing this out to us [74]. In this method we loop over the total half reciprocal space up to q_{cut} and employ a strategy for replacing CPU-intensive trigonometric calculations by simple additions and multiplications. Unfortunately, one has to loop over the lattice systematically so that this method is not compatible with the selection of subsets for each q.

The basic idea is quite simple and is based on the observation that all sines and cosines can be decomposed using the addition theorems in the following way (here written for a cosine):

$$\begin{aligned} \cos\{\mathbf{q}\cdot\mathbf{r}_i^a\} &= \cos\left\{ 2\pi \sum_{\alpha=1}^{d} \frac{n_\alpha r_{i,\alpha}^a}{L_\alpha} \right\} \\ &\overset{d=2}{=} \cos\left(\frac{2\pi n_1 r_{i,1}^a}{L_1}\right) \cos\left(\frac{2\pi n_2 r_{i,2}^a}{L_2}\right) - \sin\left(\frac{2\pi n_1 r_{i,1}^a}{L_1}\right) \sin\left(\frac{2\pi n_2 r_{i,2}^a}{L_2}\right) . \end{aligned} \tag{3.36}$$

For three dimensions a similar, lengthier expression is obtained.

We thus need only to compute the sines and cosines for each component and each particle and can obtain the trigonometric functions with arguments $\mathbf{q}\cdot\mathbf{r}_i^a$ by multiplication. If intermediate results for lower dimension product terms are saved

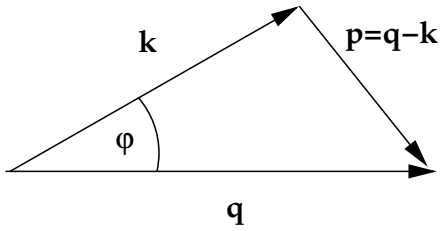

Figure 3.1: Geometry of the vectors used for defining the three particle structure factor $S_3(\mathbf{q}, \mathbf{k})$.

for $d > 2$ the number of operations can be further decreased and the calculation of the structure factor now needs only $\mathcal{O}(Mq_{\text{cut}})$ trigonometric terms and $\mathcal{O}(Mq_{\text{cut}}^d)$ additions and multiplications, which are much faster. For efficiency it is also advantageous to calculate all static structure factors at once by collecting the appropriate terms successively.

We note that we considered to use DE MOIVRE's formula

$$\cos(kx) + \mathrm{i}\sin(kx) = [\cos(x) + \mathrm{i}\sin(x)]^k \tag{3.37}$$

for replacing even more trigonometric calculations. However, there are two objections against doing so. First, the number of trigonometric terms to evaluate is $\mathcal{O}(q_{\text{cut}}^{d-1})$ smaller than the number of other operations. Hence there is not much potential for saving CPU time. Second, for large integers k round-off errors could accumulate and introduce systematic deviations.

In appendix D.1 the program code for the calculation of the static structure factors is given.

3.8 Three Particle Structure Factors

The collective structure factor is given by $S(\mathbf{q}) = \langle \rho(\mathbf{q})\rho(-\mathbf{q}) \rangle$. It measures the distribution of two particles and consequently depends on one reciprocal vector $\mathbf{q}$. The dependence reduces for homogeneous, isotropic systems to the length of $|\mathbf{q}| = q$. A generalization to a three particle distribution is the three monomer structure factor which depends on two vectors $\mathbf{q}$ and $\mathbf{k}$ connecting three particles. The vector $\mathbf{p} = \mathbf{q} - \mathbf{k}$ is the third side of the triangle formed by the three particles, see Fig. 3.1. $\mathbf{q}$ and $\mathbf{k}$ span an angle φ given by the law of cosines,

$$\cos\varphi = \frac{q^2 + k^2 - p^2}{2qk} \; . \tag{3.38}$$

We now define the three particle structure factor for the melt, $S_3(\mathbf{q},\mathbf{k})$ and for the polymer, $S_3^{\mathrm{p}}(\mathbf{q},\mathbf{k})$. For the melt the defining equation is

$$\begin{aligned}
S_3(\mathbf{q},\mathbf{k}) &\stackrel{\text{def}}{=} \frac{1}{nN}\langle\rho(-\mathbf{q})\rho(\mathbf{k})\rho(\mathbf{q}-\mathbf{k})\rangle \\
&= \frac{1}{nN}\Big\langle \sum_{i,j,l=1}^{n}\sum_{a,b,c=1}^{N} \exp\big\{\mathrm{i}[-\mathbf{q}\cdot\mathbf{r}_i^a + \mathbf{k}\cdot\mathbf{r}_j^b + (\mathbf{q}-\mathbf{k})\cdot\mathbf{r}_l^c]\big\}\Big\rangle \\
&= \frac{1}{nN}\Big\langle \sum_{i,j,l=1}^{n}\sum_{a,b,c=1}^{N} \cos(\mathbf{k}\cdot\mathbf{r}_j^b)\cos(\mathbf{k}\cdot\mathbf{r}_l^c)\cos(\mathbf{q}\cdot\mathbf{r}_i^a)\cos(\mathbf{q}\cdot\mathbf{r}_l^c) \\
&\qquad + \cos(\mathbf{q}\cdot\mathbf{r}_i^a)\cos(\mathbf{q}\cdot\mathbf{r}_l^c)\sin(\mathbf{k}\cdot\mathbf{r}_j^b)\sin(\mathbf{k}\cdot\mathbf{r}_l^c) \\
&\qquad + \cos(\mathbf{k}\cdot\mathbf{r}_l^c)\cos(\mathbf{q}\cdot\mathbf{r}_l^c)\sin(\mathbf{k}\cdot\mathbf{r}_j^b)\sin(\mathbf{q}\cdot\mathbf{r}_i^a) \\
&\qquad - \cos(\mathbf{k}\cdot\mathbf{r}_j^b)\cos(\mathbf{q}\cdot\mathbf{r}_l^c)\sin(\mathbf{k}\cdot\mathbf{r}_l^c)\sin(\mathbf{q}\cdot\mathbf{r}_i^a) \\
&\qquad - \cos(\mathbf{k}\cdot\mathbf{r}_l^c)\cos(\mathbf{q}\cdot\mathbf{r}_i^a)\sin(\mathbf{k}\cdot\mathbf{r}_j^b)\sin(\mathbf{q}\cdot\mathbf{r}_l^c) \\
&\qquad + \cos(\mathbf{k}\cdot\mathbf{r}_j^b)\cos(\mathbf{q}\cdot\mathbf{r}_i^a)\sin(\mathbf{k}\cdot\mathbf{r}_l^c)\sin(\mathbf{q}\cdot\mathbf{r}_l^c) \\
&\qquad + \cos(\mathbf{k}\cdot\mathbf{r}_j^b)\cos(\mathbf{k}\cdot\mathbf{r}_l^c)\sin(\mathbf{q}\cdot\mathbf{r}_i^a)\sin(\mathbf{q}\cdot\mathbf{r}_l^c) \\
&\qquad + \sin(\mathbf{k}\cdot\mathbf{r}_j^b)\sin(\mathbf{k}\cdot\mathbf{r}_l^c)\sin(\mathbf{q}\cdot\mathbf{r}_i^a)\sin(\mathbf{q}\cdot\mathbf{r}_l^c)\Big\rangle\,.
\end{aligned} \tag{3.39}$$

The intra-polymer static three particle correlator $S_3^{\mathrm{p}}(\mathbf{q},\mathbf{k})$ is defined analogously with ρ replaced by ρ^{p}.

For homogeneous, isotropic systems S_3 depends on the moduli of the three vectors only, $S_3(\mathbf{q},\mathbf{k}) \equiv S_3(q,k,p)$. We can furthermore exploit symmetry relations: First, $S_3(\mathbf{k},\mathbf{p}) = S_3(\mathbf{p},\mathbf{k})$, and we can choose without loss of generality $|p| \le |k|$. We used this relation for improving statistics by a factor two. Second, from the reality of S_3 we have $S_3(\mathbf{q},\mathbf{k}) = S_3^*(\mathbf{k},\mathbf{q}) = S_3(-\mathbf{k},-\mathbf{q})$, i.e. symmetry under simultaneous reflection of $(\mathbf{q},\mathbf{k})$. Thus, we can choose one vector from $\mathbb{R}^3$ and the other one the semi-positive half-space $\mathbb{R}_0^+ \times \mathbb{R}^2$ without loosing information. Third, the triangle inequality $|q-k| \le p \le |q+k|$ further restricts the possible combinations of vectors.

The triplet direct correlation function is related to the three particle structure factors by the triplet Ornstein-Zernike equation [75]

$$S_3(q,k,p) = S(q)S(k)S(p)\left[1 + \rho_{\mathrm{m}}^2 c_3(q,k,p)\right] \tag{3.40}$$

The often used convolution approximation [76] states $c_3(q,k,p) \equiv 0$, in other words, that there are no three body correlations which are not contained in the product of two particle correlations.

3.8.1 Annotations on the Implementation of S_3

Even when using the symmetry relations and restrictions on the vectors above, looping over a six-dimensional space is not feasible even when using a sophisticated scheme for the calculation of trigonometric terms. In fact, what was needed were the values of $S_3(k,p,q)$ at certain lattice points in $\mathbb{R}^3$. Each lattice point corresponds to a bin of width $\Delta q = 0.2$ in our calculation. If N_{bin} is the number of bins in one dimension, the problem is of order $\mathcal{O}(N_{\mathrm{bin}}^3)$. At each of the lattice points given by the values for (q,k,p) a set of 100 vector tuples $\{(\mathbf{q},\mathbf{k}) \;\big|\; |\mathbf{q}| = q, |\mathbf{k}| = k, |\mathbf{p}| = p)\}$

was chosen and $S(q,k,p)$ was taken to be the average value for this set, like it was described for the structure factors S^{x} in section 3.5.

We obtained S_3 up to $q = 40$, that is $N_{\mathrm{bin}} = 200$. Because of the numerical complexity of S_3 it was extremely important to find a fast implementation.

We proceeded in several steps:

1. Create a mapping between all q and all $\mathbf{q}$ of this length in the full reciprocal lattice and a mapping between all k and all $\mathbf{k}$ of this length in the half reciprocal lattice.

2. Loop over the MCT-lattice values for q and k. Find all $\mathbf{q}$ in this q-bin and all $\mathbf{k}$ in this k-bin. Draw vector tuples $(\mathbf{q},\mathbf{k})$ randomly and put them in bins according to $|\mathbf{p}|$ until all p-bins are filled with (approximately) 100 vectors. The outermost bins for p have to be dropped, because only special entries in the lists for q and k can produce entries there. Remove duplicate entries in the lists of vectors $\mathbf{q}$ and $\mathbf{k}$ to avoid double computations.

3. From Eq. (3.39) one sees that $S_3(\mathbf{q},\mathbf{k})$ can be decomposed in sines and cosines which depend either on $\mathbf{q}$ or on $\mathbf{k}$. Save all sines and cosines for all monomers for every $\mathbf{q}$ which is in the set of tuples we are considering and loop over the vectors $\mathbf{k}$ and put the results in the proper (q,k,p) bin. Thus, every trigonometric function involving $\mathbf{q}$ is evaluated only once for the given values of (q,k) at the cost of using lots of memory.

For the details it is best to refer to the program code for the analysis routines which is given in appendix D.2.

For validation of the code we checked the following relations. First, one has from the definition (3.39) $S_3(q=0,k,p=k) = S(k)$, so for $q = 0$ (or $k = 0$) we could test the results of the S_3 calculation against the data for the static structure factor which were obtained independently. Second, if all vectors to the lengths q and k were used for the computation, the symmetry relation $S(q,k,p) = S(q,p,k)$ must hold. If only a subset of vectors is used, this symmetry is increasingly well borne out for increased statistics and provides a measure for the fluctuations of the data. For further computations, the average $\frac{1}{2}[S(q,k,p) + S(q,p,k)]$, $p \leq k$ was used. Another method for obtaining the magnitude of the fluctuations would have been to compute the imaginary part of S_3, too. However, the already high computational effort would have been increased even more, so we did not pursue this option.

The 100 $(\mathbf{q},\mathbf{k})$ tuples in each (q,k,p) bin were drawn anew for each configuration. Hence, for bins where only a small subset was needed, the tuples chosen have only a small overlap. This is especially true at high values of the arguments. As a consequence, data-sets obtained for configurations which are not independent themselves could nevertheless be considered independent at high moduli of the wave vectors, increasing the statistical quality of the data. This is an improvement over the method used in [65] where one fixed set of vector tuples was used for all configurations. Data was accumulated for 1155 configurations. For one configuration 16 hours of CPU time were necessary to compute the S_3 data up to $q,k = 40$ on a Pentium III processor with 1GHz clock speed.

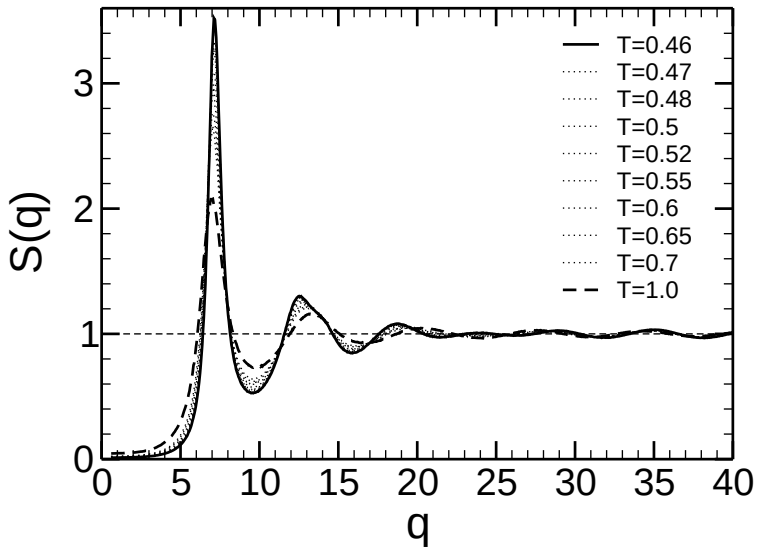

Figure 3.2: Collective monomer static structure factor $S(q)$ at all simulated temperatures. The peak at $q_{\max} \simeq 7.15$ decreases with increasing temperature.

3.9 Results for the Static Structure Factors

In this section we will present our results on the static structure of the polymer melt. We begin by the collective static structure factors of the chain and the melt. For all temperatures below $T = 1.0$, $S(q)$ has already been presented in previous works [16, 77] for a smaller q-range, but we include it in our presentation for completeness. Due to the improved algorithm we could calculate the static structure factors up to $q = 50$ averaged over more than 1000 configurations, whereas in Ref. [77] only data for $q \leq 20$ was obtained with less statistics.

3.9.1 Monomer Quantities

In Fig. 3.2 we show $S(q)$ at all temperatures which we simulated. The typical features of a liquid are seen: The nearest neighbor shell causes an "amorphous halo" around the maximum $q_{\max} \simeq 7.15$ of $S(q)$. A crystal structure would show up as a very sharp peak in contrast. Oscillations around one are present for growing q. Upon cooling, the main peak at $q_{\max}$ grows and shifts to slightly higher q, as the melt becomes denser. The temperature dependence is most pronounced around $q_{\max}$, but otherwise weak. For $q \gtrsim 30$, $S(q)$ oscillates very regularly around unity. This is due to the intra-polymer contribution $w(q)$ to $S(q)$ as we will see in Fig. 3.3.

This figure displays the intra-chain contribution separately for all temperatures up to $q = 20$. Regular oscillations around unity continue for $q > 20$. We can rationalize this high-q behavior by the following argument:

$$\begin{aligned} w(q) &= \frac{1}{N}\sum_{a,b=1}^{N} w^{ab}(q) = 1 + \frac{2}{N}\sum_{a<b}^{N} w^{ab}(q) \\ &= 1 + \frac{2}{N}(N-1)w^{a\,a+1}(q) + \frac{2}{N}(N-2)w^{a\,a+2}(q) + \dots \ . \end{aligned} \tag{3.41}$$

The next neighbor contribution $w^{a\,a+1}(q)$ can be further evaluated when we use the fact that the bond-potential fixes the distance $|\mathbf{r}^a - \mathbf{r}^{a+1}|$ of monomers a and $a+1$ at positions $\mathbf{r}^a$ and $\mathbf{r}^{a+1}$ to the bond-length $r_{\mathrm{bond}} = 0.9609$ in our model [15, 77], if we neglect thermal fluctuations:

$$\begin{aligned} w^{a\,a+1}(q) &= \left\langle \frac{1}{4\pi q^2} \int_{S^2} \exp[\mathrm{i}\mathbf{q}\cdot(\mathbf{r}^a - \mathbf{r}^{a+1})] q^2 \sin(\theta)\,\mathrm{d}\phi\,\mathrm{d}\theta \right\rangle \\ &= \left\langle \frac{\sin(q|\mathbf{r}^a - \mathbf{r}^{a+1}|)}{q|\mathbf{r}^a - \mathbf{r}^{a+1}|} \right\rangle \approx \frac{\sin(qr_{\mathrm{bond}})}{qr_{\mathrm{bond}}} \end{aligned} , \tag{3.42}$$

where $\mathbf{q}$ defined the spherical coordinate system. Since monomers with $|a-b| > 1$ are farther apart on the average than r_{bond}, these contributions to $w(q)$ should be less important as the scale with $1/|\mathbf{r}^a - \mathbf{r}^b|$. To lowest order, we can therefore expect that

$$w(q) \approx 1 + \frac{2}{N}(N-1)\frac{\sin(qr_{\mathrm{bond}})}{qr_{\mathrm{bond}}} \tag{3.43}$$

describes $w(q)$ well, if the contribution from not next neighbors has decayed for high q. Figures 3.3 and 3.4 corroborate this conjecture.

For large distances in real space, respectively small q, the DEBYE approximation [20] provides a good description of $w(q)$,

$$w(q) \approx N f_{\mathrm{D}}(q^2 R_g^2)\ , \quad f_{\mathrm{D}}(x) \stackrel{\mathrm{def}}{=} \frac{2}{x^2}\left[\mathrm{e}^{-x} + x - 1\right]\ . \tag{3.44}$$

It assumes a GAUSSian distribution for the neighbor distance, an assumption which will fail at sufficiently high q where the precise form of the potential is determining.

We have included these two approximations in Fig. 3.3. In the small q limit, the DEBYE approximation works quite well. The behavior crosses over to the oscillating next neighbor contribution $w^{a\,a+1}(q) \approx \sin(qr_{\mathrm{bond}})/qr_{\mathrm{bond}}$ which provides a very good description of $w(q)$ for $q \gtrsim 8$.

Because we saw that $w(q)$ does not depend on temperature, we may expect that also all components $w^{ab}(q)$ are independent of temperature and that only the distance along the polymer backbone is relevant for their behavior. Otherwise, temperature dependent effects had to cancel mutually. We have indeed confirmed that all $w^{ab}(q)$ are temperature independent but do not present the plot here. In other words, the polymer stays flexible and does not change its (static) structure.

Figure 3.4 presents the dependence of $w^{ab}(q)$ on the site indices a, b at temperature $T = 0.47$. $w^{ab}(q)$ depends on $|a-b|$ only, i.e. for the single polymer structure, there are no chain end effects. It is also verified that the main contribution to $w(q)$ comes from the nearest neighbors. For $|a-b| > 1$, $w^{ab}(q)$ decays very fast with q and is very close to zero at $q \gtrsim 5$. The approximation given in Eq. (3.42) is fulfilled very precisely for $w^{a\,a+1}$ and hence, $w(q)$ is well described by Eq. (3.43). We have also included the GAUSSian approximation,

$$w^{ab}(q) \approx \exp(-q^2|a-b|r_{\mathrm{bond}}^2/2d)\ , \tag{3.45}$$

with $d = 3$, which holds in the small q limit. We see that it is generally not a very good approximation. The $w^{ab}(q)$ decay faster and become negative, for next neighbors there are even long range oscillations. The GAUSSian approximation only works for $q \ll 2\pi/r_{\mathrm{bond}}$, where local effects (like the bond potential) do not play a role, so it cannot account for this.

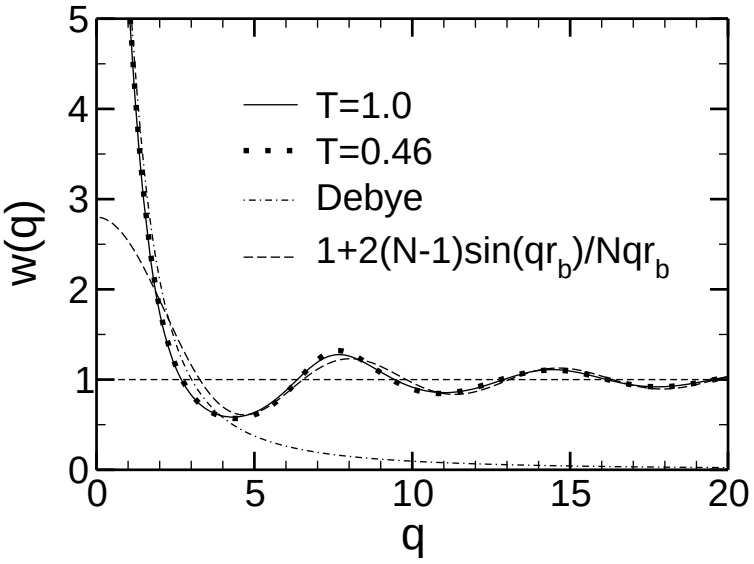

Figure 3.3: $w(q)$ at the lowest and the highest temperature $T = 0.46$ (dots) and $T = 1.0$ (lines), respectively. Both temperatures yield almost identical results so that the data for intermediate temperatures are not included. At small q, the DEBYE approximation provides a good description, whereas at high q the nearest neighbor contributions along the backbone, $1 + \frac{2(N-1)}{N} \sin(qr_{\rm bond})/qr_{\rm bond}$, dominate (we abbreviate $r_{\rm bond}$ by $r_{\rm b}$ in the legend).

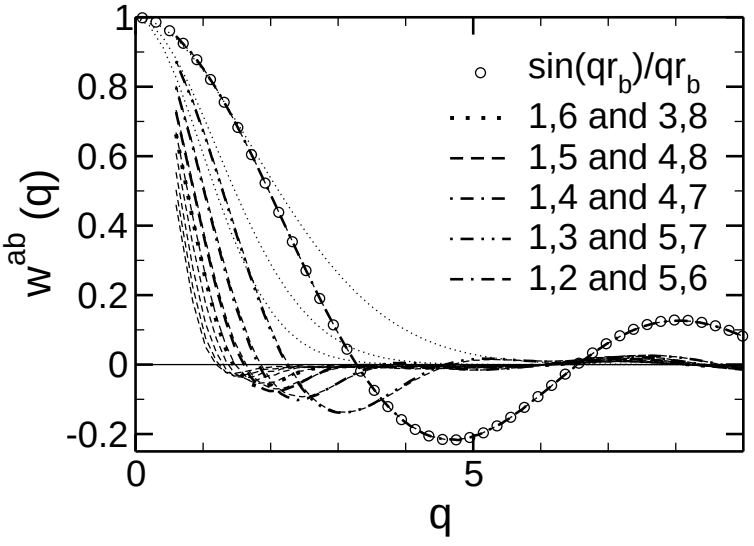

Figure 3.4: $w^{ab}(q)$ at $T = 0.47$. $w^{ab}(q)$ depends only on $|a - b|$. For $|a - b| = 1$, we confirm $w^{a\,a+1}(q) \simeq \sin(qr_{\rm bond})/(qr_{\rm bond})$ with high accuracy. All other w^{ab} decay very fast. Thin dotted lines are used for the GAUSSian approximation, $w^{ab}(q) = \exp(-q^2|a - b|r^2_{\rm bond}/6)$, shown for indices with a separation of 1, 2, and 3.

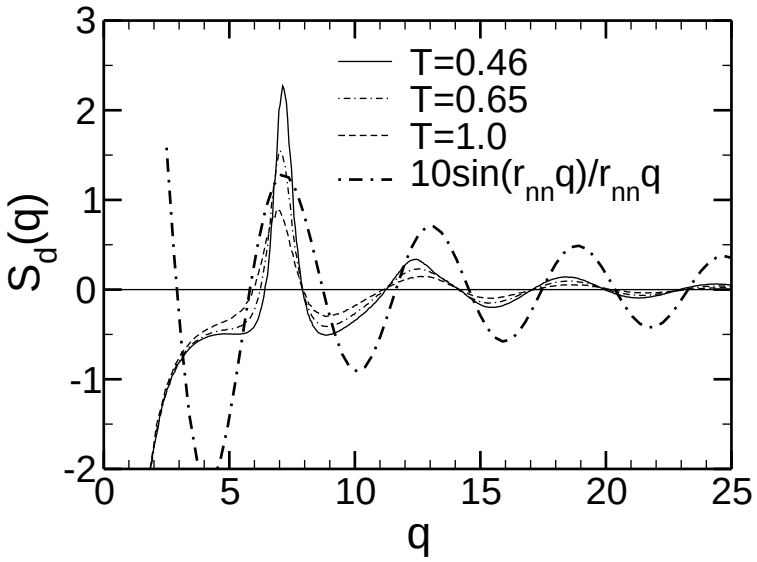

Figure 3.5: Distinct contribution to the static structure factor, $S_{\rm d}(q)$ at temperatures $T = 0.46$, 0.65 and 1.0. While the self contribution $w(q)$ is temperature independent, $S_{\rm d}(q)$ depends on temperature, especially around the maximum $q_{\rm max}$. The circles mark an approximation for $T = 0.46$ which assumes all particles to be at the preferred non-bonded nearest neighbor distance $r_{\rm nn} \approx 1.08$.

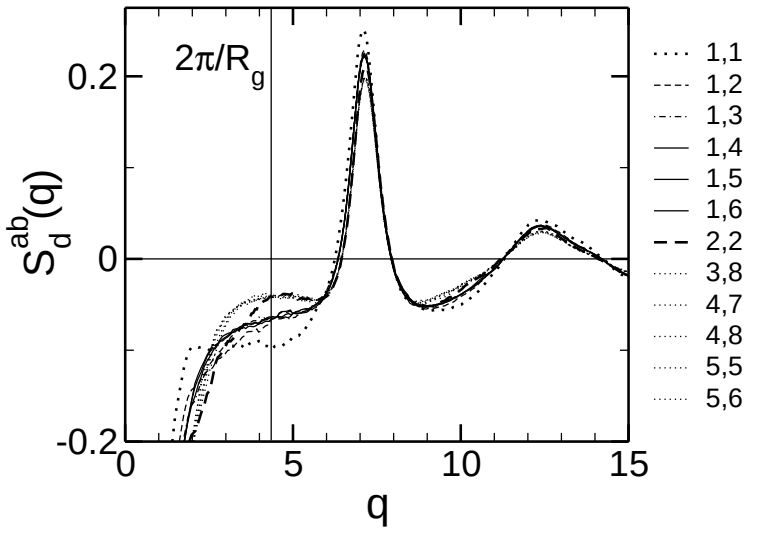

Figure 3.6: $S_{\rm d}^{ab}(q)$ at $T = 0.47$ for different index-pairs a, b. $S_{\rm d}^{ab}(q)$ depends on a, b at the maximum and for $q \lesssim 5.5$. The correlation of two chain ends ($a = b = 1$) behaves differently in comparison to all other curves.

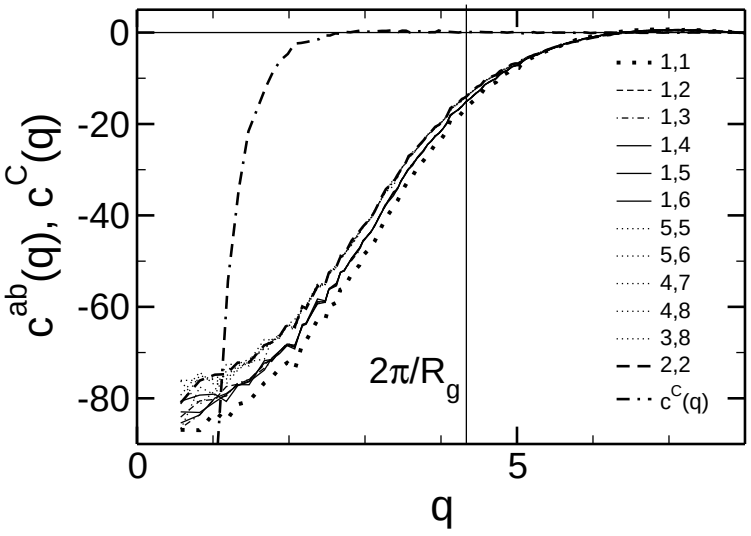

Figure 3.7: Direct correlation function $c^{ab}(q)$ at $T = 0.47$ and some selected monomer-pairs. Chain ends, $a = b = 1$ are correlated differently than other monomer pairs. The curves overlap completely for $q \gtrsim 5$ where $c^{ab}(q)$ is almost zero. The chain center of mass direct correlation function $c^{\mathrm{C}}(q)$ (dash-dash-dotted line) reaches zero very fast.

If the self (intra polymer) contribution $w^{ab}(q)$ to the collective monomer structure factor $S(q)$ is temperature independent, then the temperature dependence of $S(q)$ we stated must be due to temperature dependent distinct (inter polymer) contributions $S_{\mathrm{d}}^{ab}(q)$. This is indeed true, as Fig. 3.5 reveals. While the overall qualitative structure is liquid-like, the extrema are much more clearly borne out upon cooling. This trend is especially visible around the maximum of the structure factor, $q_{\max} \simeq 7.15$. That means that the glass transition is driven only by the distinct contribution to $S(q)$ in our model, i.e. by nearest neighbors which are not bonded. This is reasonable, as the bonds keep the bonded neighbors at an almost fixed distance and only non-bonded neighbors can close the cage tighter when T_{c} is approached.

If all non-bonded neighbors were arranged in neighbor shells at their preferred distance $r_{\mathrm{nn}} \simeq 1.08$ [15] we had $S_{\mathrm{d}} \propto \sin(qr_{\mathrm{nn}})/(qr_{\mathrm{nn}})$, with a prefactor depending on the number of neighbor shell members being approximately 10. (≈ 12 nearest neighbors for 3-d closest packing minus the number of bonded neighbors, $2 - 2/N = 1.8$). r_{nn} is slightly smaller then the minimum $r_{\min} = 1.1225$ of the LJ-potential due to finite pressure. This assumption provides a reasonable approximation for the nearest neighbor peak, but fails for the next extrema, an indication for a short ranged shell structure.

We now take a closer look at $S_{\mathrm{d}}^{ab}(q)$ at $T = 0.47$ in Fig. 3.6. While for $q \gtrsim 15$ there is no significant dependence on the sites a, b, we find a significant dependence around $q_{\max}$ and even more for $q \lesssim 5.5$. All pairs with inner sites ($1 < a, b < N$) fall on top of each other and exhibit a step-like increase around $2\pi/R_{\mathrm{g}}$. S_{d}^{22} shows slight deviations, whereas all S_{d}^{aa} with $2 < a < N - 1$ are falling on this bundle (not included in the figure). If one end-monomer ($a = 1$) is present, the step becomes weaker and is even reversed in the correlation S_{d}^{11} of two chain ends.

The direct correlation functions $c^{ab}(q)$ which we show in Fig. 3.7 for $T = 0.47$

is not very sensitive on the monomer indices. In this figure we use the same line styles as in Fig. 3.6 and observe that the curves form the same groups. The curves are very close and indistinguishable for $q \gtrsim 5$. This is a general feature of the direct correlation function which is constructed to be short ranged, because the indirect correlations via other particles are separated. At a value of $q = 2\pi/R_{\mathrm{g}} \approx 4.35$ we do not observe any particular splaying out of the curves, as we did for $S^{ab}(q)$. From this figure we can draw the conclusion that chain ends are stronger correlated than inner monomers, as $|c^{ab}(q)|$ is a measure for the correlation strength. Either chain end contributes equally to this effect. This explains why the spacing between the curves belonging to inner monomer pairs, $1 < a, b < N$ to the $a = 1, 1 < b < N$ pairs is the same as the difference of these to $c^{1\,1}(q)$. In this figure we also include the direct correlation function of the chain center of mass correlation, $\rho_{\mathrm{n}} c^{\mathrm{C}}(q) = 1 - 1/S^{\mathrm{C}}(q)$ (not considering monomer–CM couplings, see section 3.9.2). It is relatively featureless and has only for $q \lesssim 2$ a value appreciately different from zero. All direct correlation functions reach a finite value for $q \to 0$ determined by the isothermal compressibility [53],

$$\lim_{q\to 0} S(q) = \frac{\langle M^2\rangle - \langle M\rangle^2}{\langle M\rangle} = k_{\mathrm{B}} T \rho_{\mathrm{m}} \kappa_T \,, \tag{3.46}$$

$$\lim_{q\to 0} S^{\mathrm{C}}(q) = \frac{\langle n^2\rangle - \langle n\rangle^2}{\langle n\rangle} = \frac{S(q\to 0)}{N} \,, \tag{3.47}$$

because $nN = M$. The low values of the correlation function are thus caused by a low compressibility of our system.

3.9.2 Chain Center of Mass Correlators

Recently, the center of mass structure was connected to the monomer structure of the polymer chains [78]. This work is based on the PRISM formalism (polymer reference interaction site model formalism [71, 79], for a review see e.g. [80, 81]). In Ref. [78], the center of mass is included as an additional non-interacting site which is treated separately from the monomers. The monomer contribution itself is site-averaged. We sketch the main ideas before comparing with our simulation results.

The starting point in [78] is to use a 2×2 Ornstein-Zernike equation which is formally identical to Eq. (3.31), with indices taking on the values "m" for (averaged) monomer and "CM" for polymer center of mass, the pairs of indices indicate which densities were coupled. We first define the matrices $\underline{\underline{h}}$, $\underline{\underline{c}}$, $\underline{\underline{w}}$, $\underline{\underline{S}}$,

$$\underline{\underline{h}} \stackrel{\mathrm{def}}{=} \begin{pmatrix} N h_{\mathrm{m,m}} & \sqrt{N} h_{\mathrm{m,CM}} \\ \sqrt{N} h_{\mathrm{m,CM}} & h_{\mathrm{CM,CM}} \end{pmatrix} \,, \tag{3.48}$$

$$\underline{\underline{c}} \stackrel{\mathrm{def}}{=} \begin{pmatrix} N c_{\mathrm{m,m}} & \sqrt{N} c_{\mathrm{m,CM}} \\ \sqrt{N} c_{\mathrm{m,CM}} & c_{\mathrm{CM,CM}} \end{pmatrix} \,, \tag{3.49}$$

$$\underline{\underline{w}} \stackrel{\mathrm{def}}{=} \begin{pmatrix} w_{\mathrm{m,m}} & \frac{1}{\sqrt{N}} w_{\mathrm{m,CM}} \\ \frac{1}{\sqrt{N}} w_{\mathrm{m,CM}} & w_{\mathrm{CM,CM}} \end{pmatrix} \,, \tag{3.50}$$

$$\underline{\underline{S}} \stackrel{\mathrm{def}}{=} \begin{pmatrix} S_{\mathrm{m,m}} & \frac{1}{\sqrt{N}} S_{\mathrm{m,CM}} \\ \frac{1}{\sqrt{N}} S_{\mathrm{m,CM}} & S_{\mathrm{CM,CM}} \end{pmatrix} \,. \tag{3.51}$$

The quantities involved are defined analogously as in sections 3.4 and 3.6 using all possible couplings of the CM density, density of the monomers inside a polymer, and the monomer density in the melt. The matrices are symmetric by definition. The factors N and $\sqrt{N}$ arise from the averaging over N^2 monomer-monomer couplings, resp. N monomer-CM couplings.

The Ornstein-Zernike equations then read

$$\underline{\underline{h}}(q) = \underline{\underline{w}}(q)\underline{\underline{c}}(q)\left[\underline{\underline{w}}(q) + \rho_{\mathrm{n}}\underline{\underline{h}}(q)\right] , \tag{3.52}$$

$$\rho_{\mathrm{n}}\underline{\underline{c}}(q) = \left[\underline{\underline{w}}(q)\right]^{-1} - \left[\underline{\underline{S}}(q)\right]^{-1} . \tag{3.53}$$

Using Cramer's rule one quickly arrives at

$$\begin{aligned}
\rho_{\mathrm{m}}c_{\mathrm{m,m}} &= \frac{1}{w_{\mathrm{m,m}} - \frac{w_{\mathrm{m,CM}}^2}{N w_{\mathrm{CM,CM}}}} - \frac{1}{S_{\mathrm{m,m}} - \frac{S_{\mathrm{m,CM}}^2}{N S_{\mathrm{CM,CM}}}} , \\
\rho_{\mathrm{m}}c_{\mathrm{m,CM}} &= -\frac{w_{\mathrm{m,CM}}}{w_{\mathrm{m,m}}w_{\mathrm{CM,CM}} - \frac{1}{N}w_{\mathrm{m,CM}}^2} + \frac{S_{\mathrm{m,CM}}}{S_{\mathrm{m,m}}S_{\mathrm{CM,CM}} - \frac{1}{N}S_{\mathrm{m,CM}}^2} , \\
\rho_{\mathrm{n}}c_{\mathrm{CM,CM}} &= \frac{1}{w_{\mathrm{CM,CM}} - \frac{w_{\mathrm{m,CM}}^2}{N w_{\mathrm{m,m}}}} - \frac{1}{S_{\mathrm{CM,CM}} - \frac{S_{\mathrm{m,CM}}^2}{N S_{\mathrm{m,m}}}} .
\end{aligned} \tag{3.54}$$

Note that $\rho_{\mathrm{n}} = N\rho_{\mathrm{m}}$. In the notation of sections 3.4 and 3.6 we have $w_{\mathrm{m,m}} = w$, $w_{\mathrm{m,CM}} = S_{\mathrm{s}}^{\mathrm{m,C}}$, $w_{\mathrm{CM,CM}} = 1$, $S_{\mathrm{m,m}} = S$, $S_{\mathrm{CM,CM}} = S^{\mathrm{C}}$, and $S_{\mathrm{m,CM}} = S^{\mathrm{m,C}}$.

In Ref. [78] it is assumed that

$$c_{\mathrm{CM,CM}} \equiv 0 \quad c_{\mathrm{CM,m}} = c_{\mathrm{m,CM}} \equiv 0 , \tag{3.55}$$

and only $c_{\mathrm{m,m}} \neq 0$ is retained. These assumptions state that the centers of mass do not couple to each other and that the centers of mass do not couple to the individual sites in the polymer chain. We will show in the following that the validity of these simplifications is rather good.

Note that if the monomer-CM couplings are set to zero, we recover the familiar relations $\rho_{\mathrm{m}}c_{\mathrm{m,m}}(q) = 1/w(q) - 1/S(q)$ and $\rho_{\mathrm{n}}c_{\mathrm{CM,CM}}(q) = 1 - 1/S^{\mathrm{C}}(q)$ and we can identify $c_{\mathrm{m,m}}(q) \equiv c(q)$ and $c_{\mathrm{CM,CM}}(q) \equiv c^{\mathrm{C}}(q)$.

Because the matrix $\underline{\underline{c}}(q)$ has only one non-zero entry, the components of $\underline{\underline{h}}(q)$ consist only of one term,

$$h_{\mathrm{m,m}}(q) = w_{\mathrm{m,m}}(q)c_{\mathrm{m,m}}(q)[w_{\mathrm{m,m}}(q) + N\rho_{\mathrm{n}}c_{\mathrm{m,m}}(q)] , \tag{3.56}$$

$$h_{\mathrm{m,CM}}(q) = w_{\mathrm{m,m}}(q)c_{\mathrm{m,m}}(q)\left[w_{\mathrm{m,CM}}(q) + N\rho_{\mathrm{n}}h_{\mathrm{m,CM}}(q)\right] , \tag{3.57}$$

$$h_{\mathrm{CM,m}}(q) = h_{\mathrm{m,CM}} , \tag{3.58}$$

$$h_{\mathrm{CM,CM}}(q) = w_{\mathrm{m,CM}}(q)c_{\mathrm{m,m}}(q)\left[w_{\mathrm{m,CM}}(q) + N\rho_{\mathrm{n}}h_{\mathrm{m,CM}}(q)\right] . \tag{3.59}$$

It then follows by solving this system of equations that

$$h_{\mathrm{CM,CM}}(q) = \frac{w_{\mathrm{m,CM}}(q)^2}{w_{\mathrm{m,m}}(q)^2}h_{\mathrm{m,m}}(q) . \tag{3.60}$$

Using our notation from sections 3.4 and 3.6, $h_{\mathrm{CM,CM}}(q) = [S^{\mathrm{C}}(q) - 1]/\rho_{\mathrm{n}}$ and $h_{\mathrm{m,m}}(q) \equiv h(q)$, the result reads

$$S^{\mathrm{C}}(q) = 1 + \frac{1}{N}\frac{S_{\mathrm{s}}^{\mathrm{m,C}}(q)^2}{w(q)^2}[S(q) - w(q)] . \tag{3.61}$$

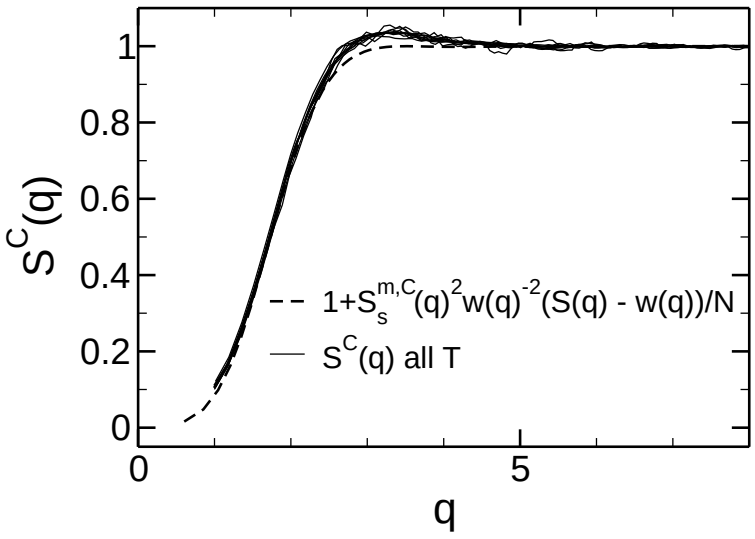

Figure 3.8: Static chain center of mass structure factor $S^{\mathrm{C}}(q)$ at all temperatures and approximation according to Eq. (3.61) calculated for $T = 1.0$.

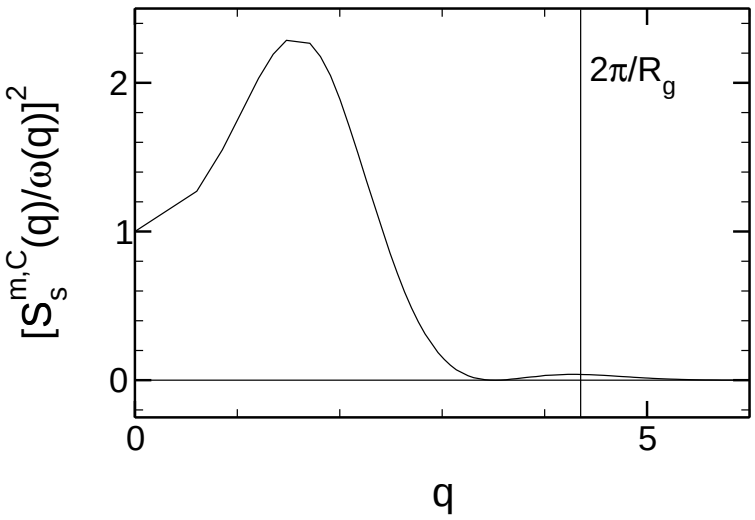

Figure 3.9: $S_{\mathrm{s}}^{\mathrm{m,C}}(q)^2/w(q)^2$ at $T = 1.0$. The maximum is around $q \approx 1.5$; for $q \gtrsim 5$ the function is close to zero, meaning that there is no coupling between the monomers and the chain's center of mass (cf. Eq. (3.61)). At $q \approx 4.35$ a small maximum can be seen.

In Ref. [78] this approximation was compared with MC simulation results of polymers with $N = 500$ repeat units on a cubic lattice at various densities from the dilute limit to semi-dilute regimes. It was found that Eq. (3.61) provided a very good description of the data.

Figure 3.8 plots the chain center of mass structure factor $S^{\mathrm{C}}(q)$ for all investigated temperatures, together with the approximation Eq. (3.61) calculated from input data at $T = 1.0$. There is no visible temperature dependence of $S^{\mathrm{C}}(q)$, as we found for the collective monomer quantities $w(q)$ and $S(q)$. Small, but systematic deviations with respect to Eq. (3.61) are detected at $q \approx 3$ similar to Ref. [78].

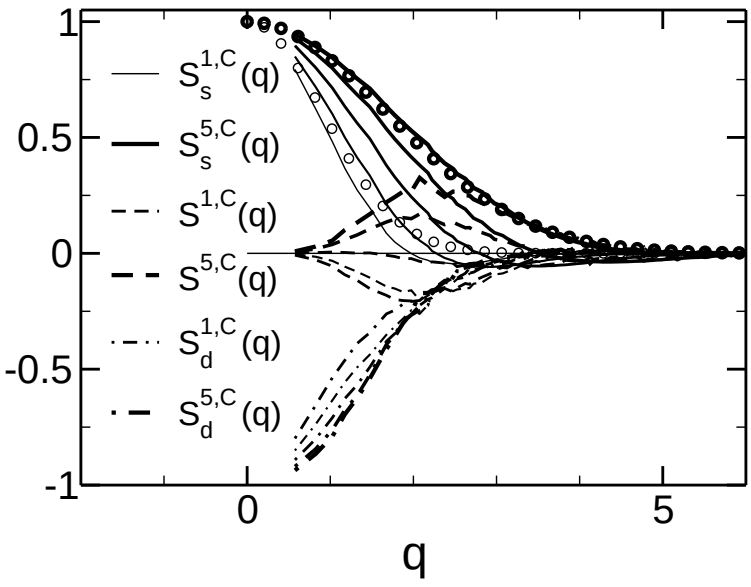

Figure 3.10: Structure factors between monomer positions and centers of mass, $S_{\rm s}^{a,{\rm C}}(q)$ (intra-polymer), $S_{\rm d}^{a,{\rm C}}(q)$ (distinct), and $S^{a,{\rm C}}(q)$ (all monomers) at $T = 0.47$. The thin lines correspond to monomers which are near the chain ends and line thickness increases towards the middle of the chains. Circles are used for plotting the GAUSSian approximation (Eq. (3.67)) of $S_{\rm s}^{a,{\rm C}}(q)$ for $a = 1$ and $a = 5$.

The coupling factor between the total correlation functions in Eq. (3.61),

$$\frac{S_{\rm s}^{\rm m,C}(q)^2}{w(q)^2} \underset{q\to 0}{\to} 1 \,, \tag{3.62}$$

provides information about on which length scale the coupling of $S^{\rm C}$ and $S - w$ takes place. The results for $T = 1.0$ are plotted in Fig. 3.9, where the small-q limit has been included by hand. The coupling is maximum at $q \approx 1.5$ and decays for larger q until it vanishes for $q \gtrsim 5$. At the length scale of the chains, $q \approx 4.35$, a small maximum might be an indication that the coupling is slightly enhanced.

Equation (3.55) stated that there was no coupling between monomers and CM. We now take a closer look at this correlation in Fig. 3.10 where we plotted $S_{\rm s}^{a,{\rm C}}(q)$, $S_{\rm d}^{a,{\rm C}}(q)$, and $S^{a,{\rm C}}(q)$ at $T = 0.47$. As for most other structure factors, there is no visible temperature dependence. Since the inner monomers are closer to the chain CM, the intra-polymer $S_{\rm s}^{5,{\rm C}}(q)$ extends to higher q, whereas for the chain ends, $S_{\rm s}^{5,{\rm C}}(q)$ decays very fast. This observation can be rationalized with a GAUSSian approximation.

We introduce to this end the bond-vector connecting monomers a and $a+1$ on a chain i,

$$\mathbf{b}_i^a \stackrel{\text{def}}{=} \mathbf{r}_{i+1}^a - \mathbf{r}_i^a \ . \tag{3.63}$$

With this notation we can rewrite the difference between the chain's center of mass and a monomer

$$\mathbf{r}_i^a - \mathbf{R}_i = \sum_{b=1}^{N-1} \left(\frac{b}{N}\right) \mathbf{b}_i^b - \sum_{b=a}^{N-1} \mathbf{b}_i^b \ . \tag{3.64}$$

This allows us to write $S_{\mathrm{s}}^{a,\mathrm{C}}$ (Eq. (3.16)) as

$$\begin{aligned} S_{\mathrm{s}}^{a,\mathrm{C}}(q) &= \frac{1}{n}\Big\langle \sum_{i=1}^{n} \exp\big\{\mathrm{i}\mathbf{q}\cdot[\mathbf{r}_i^a - \mathbf{R}_i]\big\}\Big\rangle \\ &= \frac{1}{n}\left\langle \sum_{i=1}^{n} \int \prod_{b=1}^{N-1} \left[\mathrm{d}\mathbf{b}_i^b\, p(\mathbf{b}_i^b)\right] \exp\left\{\mathrm{i}\mathbf{q}\cdot\left[\sum_{b=1}^{N-1}\left(\frac{b}{N}\right)\mathbf{b}_i^b - \sum_{b=a}^{N-1}\mathbf{b}_i^b\right]\right\}\right\rangle . \end{aligned} \tag{3.65}$$

When assuming a GAUSSian probability distribution for the bond vectors in 3-d,

$$p(\mathbf{b}_i^b) \approx \left(\frac{3}{2\pi B^2}\right)^{3/2} \exp\left[-\frac{3\mathbf{b}_i^{b\,2}}{2B^2}\right] \ , \tag{3.66}$$

we can evaluate the integrals (which do not depend on the chain index i in our homogeneous system) and relate the statistical bond length B to the radius of gyration, $R_{\mathrm{g}}^2 = NB^2/6$, and arrive at

$$S_{\mathrm{s}}^{a,\mathrm{C}}(q) \approx \exp\left\{-q^2 R_{\mathrm{g}}^2 \left[\frac{2N^2 + 3N + 1 - 6Na + 6a(a-1)}{6N^2}\right]\right\} \ . \tag{3.67}$$

Note that the fraction in brackets in the argument of the exponential is symmetric under $a \leftrightarrow N - a + 1$, i.e. counting the monomers from the other end of the chain. This fraction is a parabola with a minimum at $N/2$ which means that the function $S_{\mathrm{s}}^{a,\mathrm{C}}(q)$ decays faster for chain ends then for inner monomers. As we can learn from Fig. 3.10 this description is rather good, the negative values can however not be described with a GAUSSian ansatz.

A sort of anti-GAUSSian correlation is observed for the contribution from other chains: $S_{\mathrm{s}}^{a,\mathrm{C}}(q)$ is qualitatively mirrored to obtain $S_{\mathrm{d}}^{a,\mathrm{C}}(q)$. This suggests that chain-ends are farther away from the CM of other chains, just like chain ends are farther away from the CM of the chain they belong to. The sum of both aforementioned contributions, $S^{a,\mathrm{C}}(q)$, is connected to the probability to find a monomer around a CM (of the same chain or another one). This is more likely for inner monomers. As this is a local ordering effect, the positive overshoot for inner monomers must be compensated with negative values for chain ends. This compensation leads to a very small (monomer averaged) $S^{\mathrm{m,C}}(q)$ and we therefore expect also $c_{\mathrm{m,CM}}(q)$ to be small.

A small $c_{\mathrm{CM,CM}}(q)$ together with $S^{\mathrm{C}}(q)$ without a characteristic length scale at $2\pi/R_{\mathrm{g}}$ suggests that chains interact only weakly and interpenetrate easily.

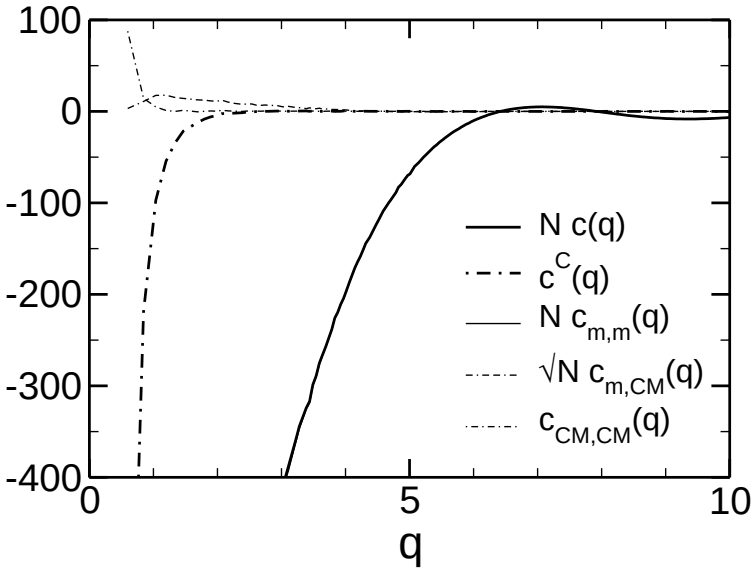

Figure 3.11: Direct correlation functions with (two lower indices) and without (no lower indices) inclusion of monomer-CM coupling at $T = 0.47$ multiplied by the weights appearing in Eq. (3.52). The monomer-monomer direct correlation functions $c(q)$ and $c_{\mathrm{m,m}}(q)$ fall together and one sees that they largely dominate over the other contributions.

We assess now the validity of the assumptions in Eqs. (3.55), i.e. we check if the direct correlation functions in Eqs. (3.54) involving the CM are small compared to $c_{\mathrm{m,m}}$. We learn from Fig. 3.11 that the collective monomer direct correlation function $c_{\mathrm{m,m}}$ indeed largely dominates the other components of $\underline{\underline{c}}$. From this point of view, Eqs. (3.55) seem completely justified. However, this does not rule out that the coupling between the monomers and the CM could be important for site resolved monomer-CM couplings, because the contributions from different sites could cancel like they did in $S^{\mathrm{m,C}}(q)$. This question could be answered by a site-wise inclusion of the monomer-CM coupling in the PRISM formalism.

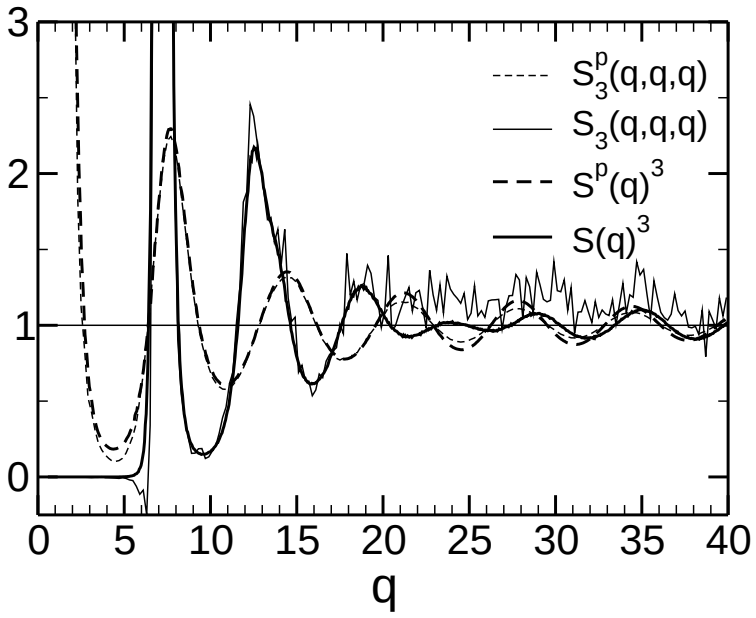

Figure 3.12: $S_3(q,q,q)$ and $S_3^{\rm p}(q,q,q)$ at $T = 0.47$ (thin lines) and their convolution approximations $S(q)^3$, resp. $S^{\rm p}(q)^3$ plotted with thick lines. A high noise level in $S_3(q,q,q)$ for $q \gtrsim 20$ hinders interpretation.

3.10 Three Particle Correlation Results

In this section we will investigate to what extent we can expect the three particle correlations to be important for the fluid dynamics. To this end, we compare $S_3(q,k,p)$ with the convolution approximation, $S_3(q,k,p) = S(q)S(k)S(p)$, for selected subsets of (q,k,p).

Figure 3.12 presents $S_3(q,q,q)$ and $S_3^{\rm p}(q,q,q)$, that is, all three vectors $\mathbf{q}$, $\mathbf{k}$, and $\mathbf{p}$ make up an equilateral triangle characterized by the length of its side q. The S_3 data (thin lines) are not smoothed to highlight the higher noise level compared to the (two particle) structure factor data. In the case of $S_3^{\rm p}(q,q,q)$ the convolution approximation provides a very good description of the curve, the amplitude of the oscillations being underestimated, however. For $q \lesssim 20$, $S_3(q,q,q)$ is equally well represented in the convolution approximation, expect for a sharp dip at $q \approx 6.3$. This dip is an indication for an anti-correlation. For $q \gtrsim 20$ the interpretation of the data is made difficult due to the high noise level. In this region, $S_3(q,q,q)$ is constantly higher as $S(q)^3$ and stays above unity, the large q-limit of both quantities. This could be due to insufficient statistics.

In order to investigate whether there is an angular dependence of the goodness of the convolution approximation for the three particle structure factors we show a comparison with arguments $(q, k = q, p = q\sqrt{2(1-\cos\varphi)})$, i.e. of an isosceles triangle with two sides of length q enclosing an angle φ. These data-sets are plotted as functions of q and $\cos\varphi$ in Fig. 3.13.

For the polymer contribution $S^{\rm p}$ the agreement with the convolution approximation is generally good for most q, with the exception of $q = 7.7$ for $\cos\varphi \approx -0.75$ which is probably linked to the similar deviation in S_3 at $q = 7.1$. For both $S_3^{\rm p}$ and S_3 the convolution approximation produces oscillations at $q = 24.9$ which are absent

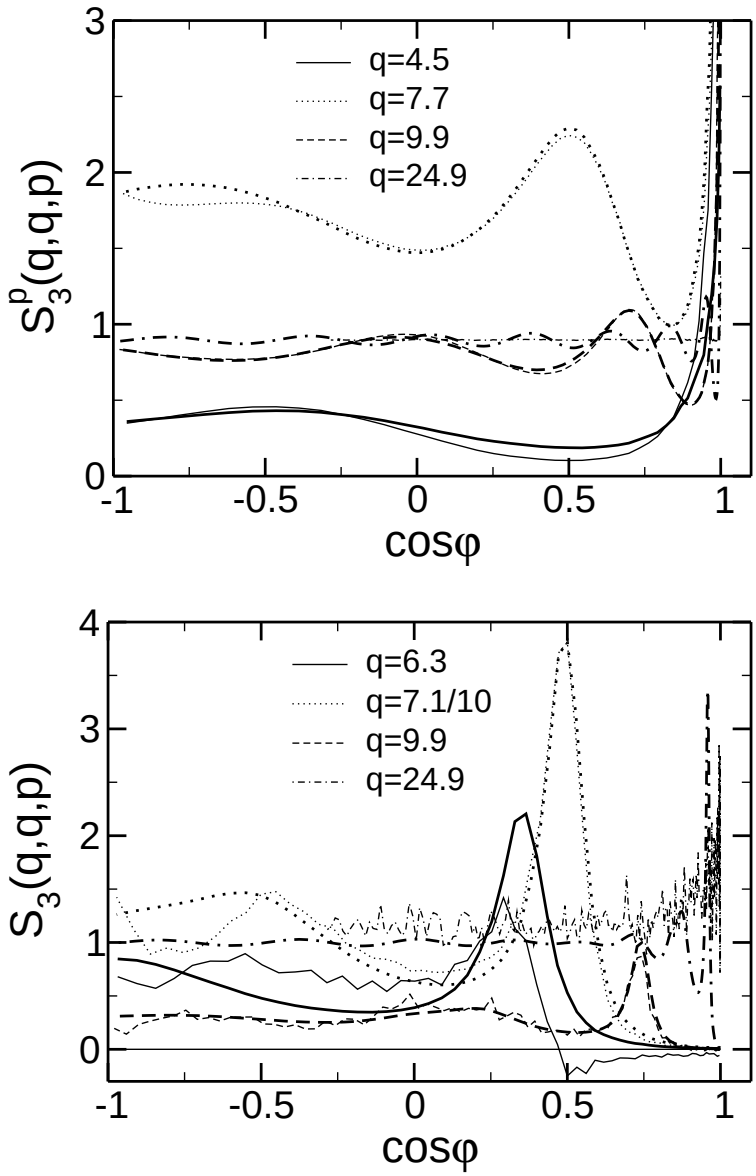

Figure 3.13: $S_3^{\mathrm{p}}(q,q,p=q\sqrt{2(1-\cos\varphi)})$ (top) and $S_3(q,q,p=q\sqrt{2(1-\cos\varphi)})$ (bottom) at $T=0.47$ (thin lines) and the convolution approximation (thick lines) for some selected q. Note that the inter-chain S_3 data at $q=7.1$ are rescaled by a factor of 10.

in the triple correlation data. Given the general quality of the convolution approximation the strong deviations of S_3 at $q=6.3$ for all $\cos\varphi$ is quite remarkable. For $\varphi \lesssim 60°$ there is an anti-correlation not observed for any other q. Consequently, this leads to a higher value for bigger angles. One might speculate, that this effect is due to the polymer backbone which favors a stretched configuration with a mean of $\cos\theta_{\mathrm{bond}}=0.205$ which coincides approximately with the maximum of the S_3 curve at $q=6.3$.

We can summarize our findings concerning the triple correlation functions by saying that the convolution approximation works well in general. The triple direct correlation function $c_3(q,k,p)$ will therefore be very small. Exceptions are special geometries of (q,k,p) were specific effects of the model come into play.

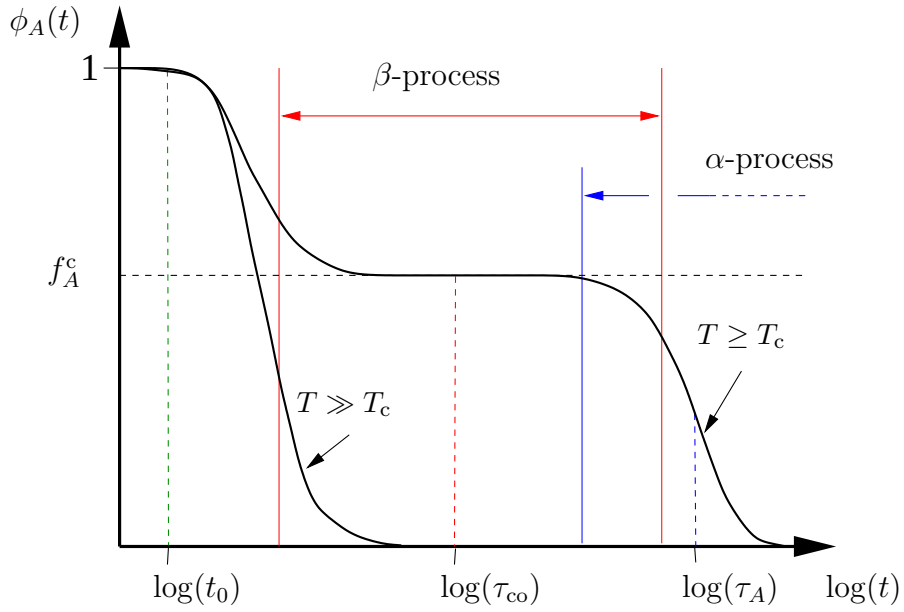

Figure 3.14: Schematic representation of a density correlator $\phi_A(t)$ on a logarithmic time scale. One distinguishes the microscopic time scale, and the β and α-regime. For temperatures in the glassy regime, a typical two-step decay is found.

3.11 Aspects of Mode Coupling Theory

In this section we give a brief overview of some aspects of mode-coupling theory (MCT) [6–8, 64, 82] which will be useful for the subsequent discussion. It should serve as a framework for more detailed statements in the following chapters.

To introduce the relevant time scales we show in Fig. 3.14 the schematic form of a time dependent correlator $\phi_A(t)$ (cf. section 3.3) in the glassy regime. Upon cooling towards the critical temperature $T_{\rm c}$ of mode-coupling theory a typical two step decay develops. The correlator decays from unity (by definition) at very short times until reaching the so-called β-plateau. At the center of the plateau the value f_A is reached at the (β-) cross-over time $\tau_{\rm co}$. In reality, the plateau is usually less well borne out, as a comparison with Fig. 3.15 shows. In the idealized version of MCT the correlator stays at this value for $T \leq T_{\rm c}$, $\lim_{t\to\infty} \phi_A(t) = f_A$, and does not decay anymore. For this reason, f_A is called "non-ergodicity parameter", as the system is not ergodic anymore for finite f_A, and can thus be used to discriminate the liquid and the glassy phases. The final relaxation of the correlator takes place in the α-regime. For measuring the time-scale of the structural relaxation, one introduces the α-relaxation time τ_A by a decay to a suitably chosen value, $\phi_A(\tau_\alpha) = C_A$.

Using the Mori-Zwanzig projector formalism [53] the dynamics of a density correlator can be exactly rewritten as a set of coupled integro-differential equations. One obtains

$$\partial_t^2 \phi(q,t) + \Omega_q^2 \phi(q,t) + \Omega_q^2 \int_0^t \mathrm{d}t'\, m_q(t-t') \partial_{t'} \phi(q,t') = 0 \,, \tag{3.68}$$

where we wrote the equations for $\phi(q,t)$, but they are analogous for other correlators

which couple to the particle density. At this stage there is no difference for simple or polymeric liquids. Equation (3.68) describes an oscillator with frequency $\Omega_q = q^2v^2/S(q)$ with v denoting the monomer thermal velocity. The density 'modes' $\phi(q,t)$ at each wavelength q are coupled by the memory kernel $m_q(t)$, which includes the temporal history of the system.

There is an exact expression for the memory kernel, which is generally not used, because it is too complicated for further evaluation. Instead, one uses simplifying assumptions: Short time effects are separated and neglected, as only the long-time dynamics is determining the glass transition. More importantly and not strictly controlled is the assumption that certain density couplings factorize. This yields the following memory kernel:

$$m_q(t) = \frac{1}{2}\int\limits_{\mathbf{k}+\mathbf{p}=\mathbf{q}} \mathrm{d}\mathbf{k}\mathrm{d}\mathbf{p}\, V(\mathbf{q};\mathbf{k},\mathbf{p}\,)\phi(k,t)\,\phi(p,t)\;, \tag{3.69}$$

where the integral respects momentum conservation with $\mathbf{p} = \mathbf{q} - \mathbf{k}$. The so-called coupling vertices in the memory kernel are

$$V(\mathbf{q};\mathbf{k},\mathbf{p}) = \rho_\mathrm{m} S(q)S(k)S(p)\frac{\{\mathbf{q}\cdot[\mathbf{k}c(k)+\mathbf{p}c(p)]\}^2}{(2\pi)^3 q^4}\;. \tag{3.70}$$

Given the number density and the static structure factors, the equations are closed. In other words, the static structure completely determines the dynamics of the glassy phase.

In the vertex above, the three particle couplings are set to zero. For silica [65] and the molecular glass former OTP [66] it was shown that the inclusion of three particle correlations significantly improved the MCT-description of simulation data, while for a simple hard-sphere liquid it is not important [83]. As the triple correlation function could be computed for our system, the influence of the three particle coupling could be investigated. In this case the vertex function reads [83]

$$V(\mathbf{q};\mathbf{k},\mathbf{p}) = \rho_\mathrm{m} S(q)S(k)S(p)\frac{\{\mathbf{q}\cdot[\mathbf{k}c(k)+\mathbf{p}c(p)] - \rho_\mathrm{m} q^2 c_3(q,k,p)\}^2}{(2\pi)^3 q^4}\;, \tag{3.71}$$

with $c_3(q,k,p)$ defined in Eq. (3.40).

The extension to molecules is done by including the single-chain conformational dynamics by the set of $N \times N$ matrix equations [18, 19]

$$\begin{aligned}
\partial_t^2 \underline{w}(q,t) + \underline{\underline{\Omega^2_{q,\mathrm{s}}}}\,\underline{w}(q,t) + \underline{\underline{\Omega^2_{q,\mathrm{s}}}}\int_0^t \mathrm{d}t'\, \underline{\underline{m_{q,\mathrm{s}}}}(t-t')\partial_{t'}\underline{w}(q,t') &= 0\;,\\
m^{ab}_{q,\mathrm{s}}(t) = \sum_c \frac{w^{ac}(q)}{q^2}\int \mathrm{d}\mathbf{k} V_\mathrm{s}(\mathbf{q};\mathbf{k},\mathbf{p}\,)\, w^{cb}(k,t)\,\phi(p,t)\;.
\end{aligned} \tag{3.72}$$

Here $\underline{\underline{\Omega^2_{q,\mathrm{s}}}} = q^2v^2\left[\underline{w}(q)\right]^{-1}$ and $V_\mathrm{s}(\mathbf{q};\mathbf{k},\mathbf{p}\,) = \rho_\mathrm{m}(\mathbf{q}\cdot\mathbf{p}/q)^2\, S(p)c(p)^2/(2\pi)^3$.

The indices a, b of the matrices denote monomer pairs with each index running from 1 to N, the chain-length.

In principle, the MCT-equations for the collective melt correlators, Eqs. (3.68) and (3.69), have to be formulated like the single chain correlators, i.e. monomer-pair separated [18]. Because of severe difficulties to solve such equation systems a

site-independent direct correlation function was assumed, $c^{ab}(q) = c(q)$. This means that each monomer is assumed to experience a site independent surrounding in the melt (including all chains). Lacking direct simulation input in Ref. [19], a GAUSSian chain conformation was assumed, $w^{ab}(q) = \exp(-q^2|a-b|r_{\text{bond}}/6)$ and collective quantities were obtained from hard-sphere calculations.

From our analysis of the MD-simulation data we were able to test the validity of aforementioned assumptions in the MCT-equations as well as to provide the necessary static input data for quantitative MCT calculations.

We first note that the consideration of $c_3(q,k,p)$ in the MCT vertex will not alter the results very much, because it is overall very small as stated in section 3.10. Explicit MCT-calculations including c_3 were performed by CHONG for our system and his results confirm this prediction [84]. However, the high noise level makes quantitative statements difficult.

As assumed, the direct correlation functions can be considered as being site-independent with high accuracy, cf. Fig. 3.7. Using a GAUSSian approximation for the intra-chain structure captures the small-q region well, the approximation will however not describe the q-range corresponding to distances smaller than the radius of gyration accurately as we learned in Fig. 3.4. Because the chain structure was seen to be independent of temperature, this does not introduce a conceptual problem, as the glass transition of our model is driven by the correlation between monomers belonging to different chains. To what extend the center of mass has to be considered, can not be answered conclusively. While neglecting the CM for the static structure seems to be in order, the next section will show that the dynamics might be clearly affected by the CM-motion.

3.12 Center of Mass and Monomer Dynamics

In this section we will present some findings concerning the α-relaxation times of the dynamical scattering functions. We will compare the results for our model with the studies for other molecular glass formers. In particular we want to address the question what influence on the dynamics in the α-regime the monomer-CM coupling may have.

In order to obtain information on the relaxation times we used so-called KOHLRAUSCH-WILLIAMS-WATTS (KWW) fits to the late part α-regime of the decay of the scattering functions. KWW functions provide a good description in this regime as we have layed out in previous work [16, 77] for the system under consideration and as was found also for OTP [66].

The KWW function describing the stretched exponential decay of a correlator $\phi^{\text{x}}(q,t)$ in the α-regime is defined by

$$\phi^{\text{x,K}}(q,t) \stackrel{\text{def}}{=} f^{\text{x,K}}(q) \exp\left[-\left(\frac{t}{\tau^{\text{x,K}}(q)}\right)^{\beta^{\text{x,K}}(q)}\right] . \tag{3.73}$$

The KWW-decay-time $\tau^{\text{x,K}}$ marks the time at which the correlator has decayed to $f^{\text{x,K}}(q)/\text{e}$. $f^{\text{x,K}}(q)$ is a measure for the plateau height of the scattering functions. Within (ideal) MCT it can be proved mathematically [85] that

$$\phi^{\text{x}}(q,t) = f^{\text{x}}(q) \exp\left[-\Gamma^{\text{x}}(q)(t/\tilde{\tau})^b\right] ,\ \Gamma^{\text{x}}(q) > 0,\ \Gamma^{\text{x}}(q) \propto q \text{ for } q \to \infty . \tag{3.74}$$

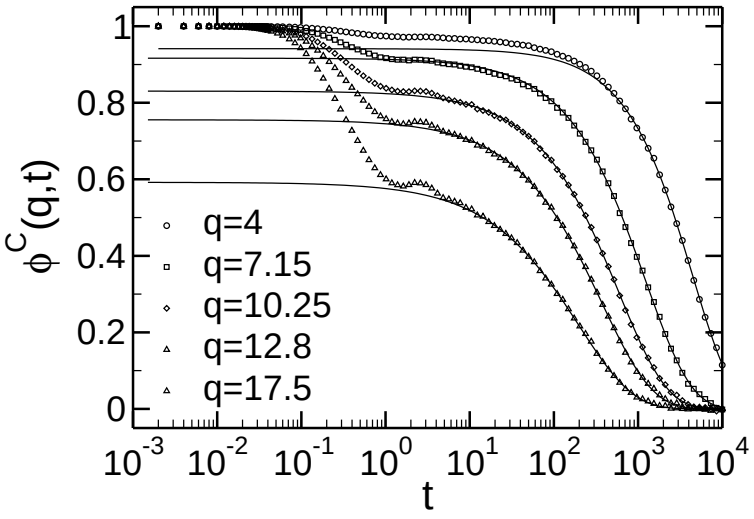

Figure 3.15: $\phi^{\mathrm{C}}(q,t)$ at $T = 0.47$ for $q = 4$, 7.15, 10.25, 12.8, 17.5 from simulation data (symbols) with KWW fits (lines). The KWW-fits provide a good description of the α-regime.

In particular, one has

$$\lim_{q\to\infty} \beta^{\mathrm{x,K}}(q) = b \,, \tag{3.75}$$

$$\lim_{q\to\infty} f^{\mathrm{x,K}}(q) = f^{\mathrm{x}}(q) \,, \tag{3.76}$$

where b is the so-called VON SCHWEIDLER exponent governing the late β-decay. It depends on the system, for our system $b = 0.75 \pm 0.04$ [14].

Indeed, we found that the $f^{\mathrm{x,K}}(q)$ behave qualitatively like the non-ergodicity parameters $f^{\mathrm{x}}(q)$ in the whole q-range studied [77], as often observed [66].

While KWW fits to the monomer scattering functions were discussed in [16, 77], we can now perform a comparison between the monomer and the CM dynamics. Unfortunately, the scattering functions for the coupling between CM and monomers are very noisy, rendering a more detailed analysis impossible.

The KWW-fits were performed in the temperature range from $T = 0.47$ to $T = 0.52$ in which glassy behavior is manifest and ideal MCT works well [15]. The time-temperature-superposition principle of MCT [7] states that in the α-regime the curves for different temperatures of a correlator fall on a single master curve for each correlator after rescaling with a T-dependent relaxation time. Hence, it is sufficient to investigate only one temperature, $T = 0.47$ in our case. We have however checked that the results are indeed independent of temperature in this T-range. This also assures the numerical stability of the algorithm used. The fits were performed in the following way: In a first step the KWW-function was fitted to all data points at time $t \geq t_{\mathrm{start}}$, with t_{start} at the beginning of the plateau, usually $t_{\mathrm{start}} = 1$. This gave us an initial estimate of the plateau height. Next, fits were done on all data with a function value less than $x_{\mathrm{cut}} f^{\mathrm{x,K}}$, until the fit parameters converged (usually in two–three iterations). The actual non-linear fitting was done using a LEVENBERG-MARQUARD method in gnuplot V3.7pl1. Not all data-sets could easily be fitted, especially for very high or low plateau values the iterative fitting did not

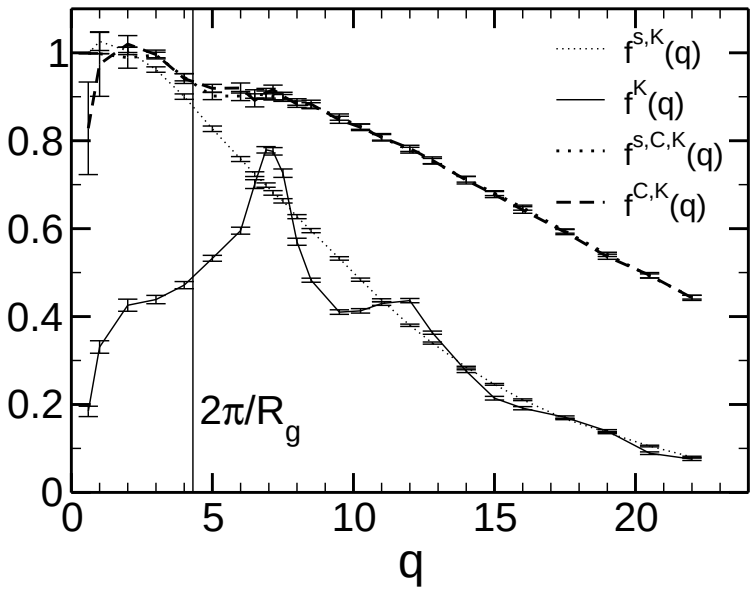

Figure 3.16: The plateau heights for the scattering functions of monomer and chain CM, $f^{\mathrm{s,K}}(q)$, $f^{\mathrm{K}}(q)$, $f^{\mathrm{s,C,K}}(q)$, $f^{\mathrm{C,K}}(q)$ at $T = 0.47$ as determined by KWW fits. The values for coherent and incoherent CM scattering functions are nearly identical, indicating that collective coupling of the CM is weak.

always converge and parameters were adjusted by hand. In particular, the results at very small q are not very reliable because the statistics is very poor, the plateaus are very high, and the simulation did not cover the complete decay so that the fit was performed only on the early α-regime. The choice for x_{cut} influences also the fit results, but $x_{\mathrm{cut}} = 0.8$ is a reasonable choice. For a more complete discussion see Refs. [15, 77].

To get an impression for the shape of the correlators and the KWW-functions, we present in Fig. 3.15 the coherent CM correlator $\phi^{\mathrm{C}}(q,t)$ at $T = 0.47$ at some q. The different time-regimes are clearly distinguished. The KWW fits provide a very good approximation for the simulation data in the α-regime.

The plateau heights $f^{\mathrm{x,K}}(q)$ of the incoherent (superscript s) and coherent monomer and CM (superscript C) correlation functions at $T = 0.47$ are plotted in Fig. 3.16. The non-ergodicity parameters provide information about the decay in the β-regime, where the cage-effect dominates the dynamics. A high value $f^{\mathrm{x,K}}(q)$ means that the scattering function $\phi^{\mathrm{x}}(q,t)$ is only slowly decaying ('freezing') in the β-regime. There is a striking difference between monomer dynamics and CM dynamics: While the plateau heights for coherent and incoherent CM scattering functions are nearly identical (we will see similar behaviors for the other KWW-parameters), they differ significantly for the corresponding monomer values. For the monomers this is due to the coupling of the monomers in the melt giving rise to the cage effect at $q_{\mathrm{max}} \simeq 7.15$, the length scale of the nearest neighbor distance. For the coherent CM coupling an analogous effect would be the hindrance of a whole chain in a (soft) cage formed by the polymer coils of surrounding chains. This would presumably happen at $q \approx 2\pi/R_{\mathrm{g}}$, the typical length scale of a polymer coil. The absence of a difference in $f^{\mathrm{s,C,K}}(q)$ and $f^{\mathrm{C,K}}(q)$ indicates that there is no such caging and no

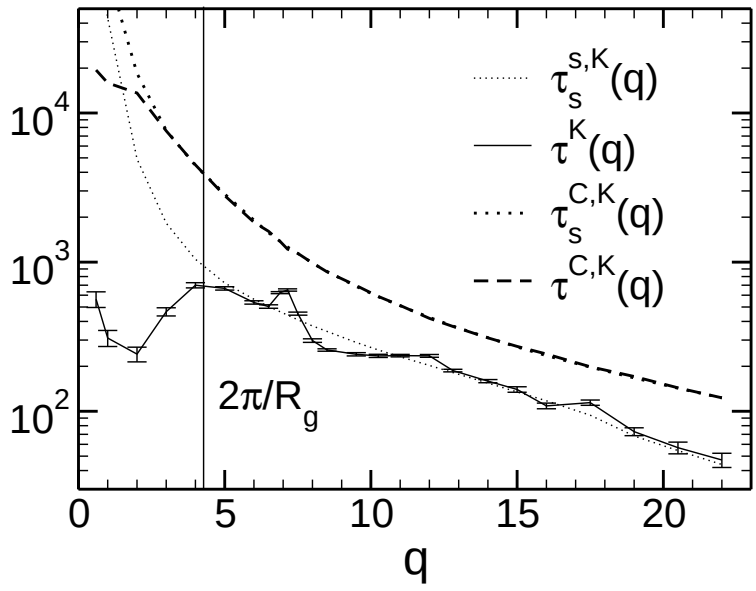

Figure 3.17: KWW-times for the coherent monomer-monomer and CM-CM correlation function, $\tau^{\mathrm{K}}(q)$, respectively $\tau^{\mathrm{C,K}}(q)$ and the corresponding incoherent functions, $\tau_{\mathrm{s}}^{\mathrm{K}}(q)$ and $\tau^{\mathrm{K}}(q)$ at $T = 0.47$. While there is a pronounced peak around $q = 2\pi/R_{\mathrm{g}}$ in the relaxation times of the coherent monomer scattering function $\tau^{\mathrm{K}}(q)$, this feature is absent in the coherent center of mass scattering function. Error bars are one standard deviation. Where not drawn, error bars are comparable to the line width.

glassy dynamics leading to a slow decay of collective CM-modes. In contrast, a peak in $f^{\mathrm{C,K}}(q)$ at $2\pi/$(size of OTP) was reported for MD-data of an OTP model [66], which is a rigid three-body molecule. This suggests that the polymers can easily interpenetrate so that no caging of polymer coils can happen.

In Fig. 3.17 we present a comparison between the KWW decay times for the different scattering functions. The broad peak in $\tau^{\mathrm{K}}(q)$ might suggest that the CM-motion of the polymers gives rise to it, because it is at the length scale corresponding to the radius of gyration, R_{g}, namely $2\pi/R_{\mathrm{g}} \approx 4.35$. For the monomer density coupling, the preferred neighbor spacing leads to the much sharper peak in $\tau^{\mathrm{K}}(q)$ at q_{max}. However, a peak at $2\pi/R_{\mathrm{g}}$ in $\tau^{\mathrm{C,K}}(q)$ is absent. From this finding we draw the conclusion that the CM–CM coupling is not at the origin of the first, broad peak at $q \approx 2\pi/R_{\mathrm{g}}$ in the relaxation times of the monomer scattering function. In addition, we find that there is very little difference between the self and collective CM-CM correlation times, as it was found for the plateau heights. This supports our statement that this coupling must be weak. In contrast, in Ref. [66] it is shown that $\tau^{\mathrm{C,K}}(q)$ has a peak just at a length corresponding to the size of the molecule and consequently it has been suggested that the collective CM-dynamics dominates also the monomer dynamics of OTP at this length scale.

Finally, we plot the stretching exponents $\beta^{\mathrm{x,K}}(q)$ for the monomer and CM scattering functions in Fig. 3.18. As for the other KWW parameters, the curves for the incoherent and coherent CM correlators coincide. Interestingly, there is a broad peak at $q \approx 2\pi/R_{\mathrm{g}}$, which means that the decay is less stretched as for other q. It

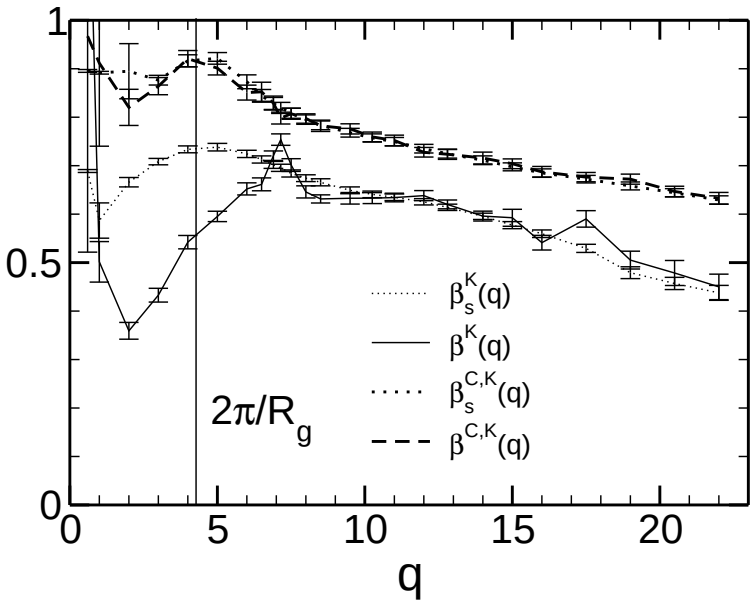

Figure 3.18: KWW stretching exponents $\beta^{\mathrm{x,K}}(q)$ for the monomer and CM scattering functions at $T = 0.47$.

is at the same positions as a weak shoulder in $\beta^{\mathrm{K}}(q)$ and the maximum of $\beta_{\mathrm{s}}^{\mathrm{K}}(q)$.

Our observations for the KWW parameters are consistent with the fact that the direct correlation function of the CM–CM coupling decays to zero very fast at $q \approx 2$, which is by a factor of two lower than the peak we are interested in (see Fig. 3.7). They also corroborate that chains are only weakly interacting, as suggested by S^{C}.

3.13 Summary and Conclusions

We have performed an extensive analysis of the statics and dynamics of a non-entangled polymer melt consisting of short chains with $N = 10$ repeat units in the liquid and glassy state. By using a new, improved method for calculating the static structure factors we were able to investigate the site-dependence and the influence of the center of mass (CM) on the static structure with unprecedented precision. In addition, the importance of three particle correlations could be assessed by the three particle static structure factors.

The chain structure $w(q)$ was seen to be unaffected by temperature, and only the distinct part of the collective structure factor, $S_{\mathrm{d}}(q)$, changed with temperature. We found that the coherent intra chain structure is dominated by the next neighbor contribution for $q \gtrsim 2\pi/R_{\mathrm{g}} \simeq 4.35$. This is because $w^{a\,a+1}(q)$ has a much higher magnitude than other site-site resolved $w^{ab}(q)$, due to molecular bonds which keep the distance of bonded neighbors almost fixed, independent of temperature. This constraint is absent in the distinct structure factors, explaining why the change in the static structure upon cooling towards the glass transition can only come from monomers on other chains. As a GAUSSian ansatz for $w^{ab}(q)$ can not include a fixed bond-length, deviations from $w^{ab}(q)$ calculated from simulation data were seen.

While $w^{ab}(q)$ depends only on the distance of sites $|a - b|$ along the backbone,

the distinct, site resolved inter-chain structure $S_{\mathrm{d}}^{ab}(q)$ depends on $|a-b|$ together with the number of end-monomers in a, b. Hence, for the investigation of chain-end effects, this quantity is to be considered. The site-dependence of $S_{\mathrm{d}}^{ab}(q)$ is most pronounced around $q \approx 2\pi/R_{\mathrm{g}}$, indicating an influence of the coupling to the chains' center of mass (CM). In the direct correlation functions, $c^{ab}(q)$, the site dependence was found to be only weak, justifying a replacement by the monomer-pair averaged $c(q)$ in MCT-calculations.

The static structure of the centers of mass, $S^{\mathrm{C}}(q)$, is relatively featureless. We found that both the site averaged monomer-CM and the CM-CM direct correlation function are small compared to the monomer-monomer direct correlation function. This suggests that polymers are soft, only weakly interacting objects which can easily interpenetrate. However, we found a pronounced qualitative site dependence of the monomer-CM structure factor $S^{a,\mathrm{C}}(q)$ with different signs for chain ends ($a = 1, N$) and inner monomers ($a = N/2$). Averaging over all sites might therefore lead to an underestimation of this coupling.

Our results concerning the three particle static structure factors $S_3(q,k,p)$ and $S_3^{\mathrm{p}}(q,k,p)$ indicate that for our system the convolution approximation which factorizes the three particle static structure factors and sets the triple correlation function to zero works overall quite well. This is consistent with the fact that the structure is close to a simple liquid and the polymer remains flexible at all temperatures. There are deviations at intermediate q and at certain angles which result presumably from a preferred bond-angle in our model.

By analyzing the dynamics and performing KWW-fits we found that the CM-dynamics shows no indication that it is particularly slowed down at the length scale R_{g} of a polymer. This would be the chain analog to the cage-effect for monomers. In contrast to the monomer correlators, the incoherent and coherent CM correlators behave very similar. These findings are consistent with a weak CM-CM coupling inferred from the small magnitude of the CM-CM direct correlation function. Thus, the pronounced peak in the relaxation time of the monomer dynamics at $q \approx 2\pi/R_{\mathrm{g}}$ is unlikely to be due to the CM-CM coupling which leaves therefore only a site dependent monomer-CM coupling as cause for it because the site averaged monomer-CM direct correlation function is also small. The exact origin of the peak in $\tau^{\mathrm{K}}(q)$ at lengths corresponding to the chain size for polymeric systems still remains to be elucidated.

3.14 Outlook

A numerical evaluation of the MCT-equations (3.68) – (3.70) and (3.72) using the static input data from the simulation has been done by Chong [86]. From the comparison of results for the non-ergodicity parameter $f(q)$ in Fig. 3.19 we can learn that while the MCT calculation using the Gaussian approximation for $w^{ab}(q)$ together with hard-sphere collective structure data yields satisfactory results, the utilization of the MD data leads to a considerable improvement. In this case $f(q)$ obtained from a detailed MCT-analysis [15] is described very well for $q \gtrsim q_{\max}$. The dip in the MCT data at $q \approx 10.25$ is a numerical artifact. The amplitude of the KWW fits, $f^{\mathrm{K}}(q)$ is included to show that it matches the non-ergodicity parameter

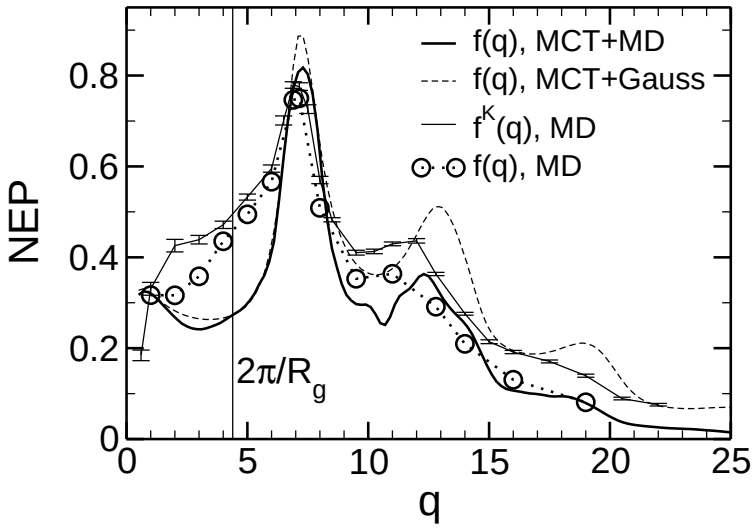

Figure 3.19: Comparison of the non-ergodicity parameters for the coherent scattering function $f(q)$ from simulation data (circles, from Ref. [15]), $f^{\mathrm{K}}(q)$ at $T = 0.47$, (thin line, as in Fig. 3.16), MCT calculations using a GAUSSian approximation for $w^{ab}(q)$ and hard-sphere collective structure data (dashed line, from Ref. [19]), and MCT calculations using the static input from the simulation data (thick solid line, from Ref. [86]).

$f(q)$ semi-quantitatively. Since the non-ergodicity parameter is a very fundamental quantity for the description of the glassy dynamics, we do not show a comparison of other quantities obtained in the MCT calculation where a similar matching with simulation data was observed.

The good agreement of the MCT calculations using simulation data at $q \gtrsim q_{\max}$ contrasts with the clear deviations around $q \approx 2\pi/R_{\mathrm{g}}$. This finding can be interpreted as indication that the CM dynamics has not been appropriately included in the MCT equations for polymers. As pointed out before, this feature rests unexplained and needs further study.

To the end of investigating the influence of the CM in more detail, an extended PRISM analysis would be desirable, where the monomer-CM couplings are treated site-wise and not site averaged as done so far.

In addition, it would be interesting to perform simulations with varying chain length. This would shift R_{g} and consequently the features caused by the CM-motion would shift accordingly, allowing to identify them more clearly. Longer chains would also allow for the investigation of entanglement effects in our polymer melt, which are absent for $N = 10$.

One could also think of simulations of two-dimensional films in the glassy state as natural extension of bulk and film simulations in the glassy state [17]. Again, simulations of different chain lengths would be very interesting as connectivity effects are more pronounced in 2-d as in 3-d.

Chapter 4

Polymer Specific Effects and String-Like Correlated Motion in the Dynamics of Supercooled Polymer Melts

4.1 Introduction

Many liquids can be supercooled below the melting temperature T_{m} and transformed into a glass, a solid phase without long-range positional order, at the glass transition temperature T_{g}. Structurally, the liquid well above T_{g} and the amorphous solid below T_{g} are almost identical. Yet, the dynamics is very different. In the temperature interval $T_{\mathrm{g}} \lesssim T \lesssim T_{\mathrm{m}}$ the relaxation time slows dramatically, typically by more than 10 orders of magnitude [4, 5, 8].

A possible explanation could be that a glass former develops spatially heterogeneous dynamics upon cooling toward T_{g} [21–24]. The term "dynamic heterogeneity" means that the amorphous packing in the supercooled state engenders aggregates ("subensembles") of particles with enhanced or reduced mobility relative to the average. These aggregates are not envisaged as static entities. Rather they are supposed to be fluctuating objects with some finite life time. To test this idea experimentally several techniques, such as multi-dimensional NMR [87, 88], optical bleaching [89], non-resonant spectral hole burning [90] or solvation dynamics [91], have been applied to a variety of glass formers (for reviews see [21, 22, 24]). These experiments show that it is possible to select subensembles of slow or fast particles close to T_{g}. Although the possibility of detecting such subsets suggests that the fast or slow particles are close in space, a direct evidence of this spatial correlation is often hard to extract by these experimental approaches [21]. In multidimensional NMR experiments, it is typically of the order of 3 nm close to T_{g} [88].

Information on this correlation has, however, become accessible in recent experiments on colloidal suspensions [92, 93]. Colloidal suspensions undergo a glass transition driven by density [94], which, for hard-sphere colloids, is well described by the idealized mode-coupling theory (MCT) [95–98], cf. section 3.11. Individual particle trajectories can be monitored by confocal spectroscopy [92, 93]. This new technique thus provides the same information as computer simulations on different

length and time scales. In the supercooled phase, the experiments [92] revealed that the fastest particles form clusters, the size of which grows as the glass transition is approached. These findings closely agree with the results obtained from computer simulations of various model glass formers [23], such as binary soft-spheres [99] and Lennard-Jones (LJ) mixtures [100–102], polymer melts [26], and water [103] (see also [104–106] for related work on two- and three-dimensinal hard spheres as well as [107] for dynamical heterogeneities in a LJ-mixture below $T_{\rm g}$).

In this chapter we analyze molecular-dynamics simulations of a bead-spring model for a non-entangled glassy polymer melt [12], which was already discussed in chapter 3. While chapter 3 was mainly concerned about the static structure and the long-time dynamics under special consideration of the role of the center of mass motion, we present here a detailed study of the polymer aspects of the intermediate time dynamics.

This model has been extensively investigated in the supercooled regime above the critical temperature $T_{\rm c}$ of MCT[15, 16, 70, 108, 109]. In this temperature regime, it exhibits dynamical heterogeneities. This was evidenced by two approaches: The first consisted in measuring spatial correlations between the displacements of different monomers. Reference [25] showed that the strength of these correlations depends on time, that it is largest for times where a monomer is likely to break out of its nearest-neighbor "cage", and that this maximum correlation grows on cooling toward $T_{\rm c}$. These findings were supported by a second approach [26] which identifies clusters of highly mobile particles and analyzes their size distribution as a function of time and temperature close to $T_{\rm c}$.

The analyses of Refs. [25, 26] averaged over all monomers in the melt without discriminating between monomers bonded in a chain and non-bonded ones. The goal of the present work is to study the influence of chain connectivity on the dynamical heterogeneities in our model. To this end, we have considerably extended the simulations to longer times at the lowest and the highest temperature compared to Refs. [25, 26], and we investigated spatial correlations in the motion of the highly mobile monomers, similarly to Refs. [100–102].

This chapter is organized as follows: Section 4.2 reviews some properties of typical mean-square displacements. These quantities are important reference points for the following analysis. Section 4.3 discusses two non-Gaussian parameters and explains how we define mobile monomers. In Sec. 4.4 we show that the most mobile monomers follow each other in a stringlike fashion and we discuss the influence of chain connectivity on this motion. A summary of the basic concepts used in the analysis and of the main results is given in Sec. 4.5.

4.2 Mean-square Displacements

While the temperature $T = 0.46$ is slightly larger than $T_{\rm c} \simeq 0.45$ (critical temperature of mode-coupling theory (MCT) [7, 8]), the highest temperature $T = 1$ lies well above the temperature range where the transient blocking of neighboring monomers, the "cage effect" [7, 8], can be observed. This blocking is responsible for the glass-like, slow relaxation of our model [15, 16, 70], cf. sections 3.11–3.13. We illustrate this point by the mean-square displacement (MSD) of a monomer $g_0(t)$

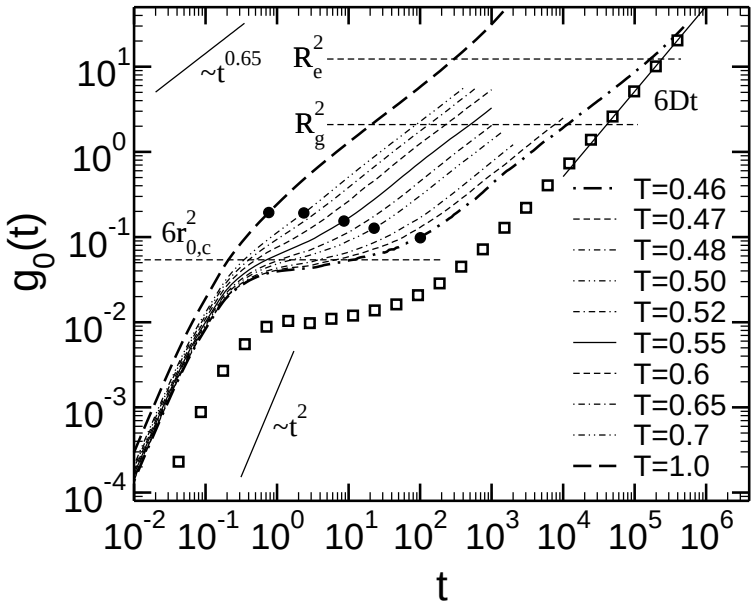

Figure 4.1: Time-dependence of the MSD of all monomers, $g_0(t)$, for all T studied and of the MSD of the center of mass, $g_3(t)$, at $T = 0.46$ (shown by $\square$) [see Eq. (4.1)]. Temperature decreases from the left ($T = 1$) to the right ($T = 0.46$) in the figure. The lowest temperature is slightly above $T_c \simeq 0.45$ [15, 16, 70]. The dashed horizontal lines indicate $6r_{0,c}^2$ (LINDEMANN localization length $r_{0,c} \simeq 0.095$) and the values of the radius of gyration R_g^2 ($= 2.09$) and of the end-to-end distance R_e^2 ($= 12.3$). The filled circles ($\bullet$) for $T = 0.46, 0.5, 0.55, 0.7, 1$ mark the values of g_0 which correpond to the time $t^*_{\alpha_2}$ where the non-GAUSSian parameter α_2 is maximum (see Figure 4.2). $t^*_{\alpha_2}$ is in the regime of the α-process. The solid line labeled $\sim t^{0.65}$ shows an effective power law describing the data in the regime $1 \lesssim g_0 \lesssim R_e^2$, where the connectivity between the monomers dominates the dynamics [109]. Two other solid lines indicate the behavior in the ballistic regime ($\sim t^2$) and the diffusion of the center of mass ($6Dt$).

and of the center of mass of a chain $g_3(t)$,

$$g_0(t) \stackrel{\text{def}}{=} \left\langle \left[\mathbf{r}_i(t) - \mathbf{r}_i(0)\right]^2 \right\rangle \quad \text{and} \quad g_3(t) \stackrel{\text{def}}{=} \left\langle \left[\mathbf{R}_c(t) - \mathbf{R}_c(0)\right]^2 \right\rangle . \tag{4.1}$$

Here, $\langle\cdot\rangle$ denotes the canonical ensemble average, $\mathbf{r}_i(t)$ is the position of monomer i (in any chain) at time t, and $\mathbf{R}_c(t)$ is the position of the center of mass of chain c at time t. Note that we use a modified notation with respect to the notation in section 3.3 by including both the chain index and the site index in i. This is appropriate, as the full site dependence of the monomers in a chain is not discussed here.

Figure 4.1 depicts $g_0(t)$ and $g_3(t)$. At short times, $g_0(t) \sim t^2$ for all temperatures (regime of ballistic motion). If $T = 1$, the displacement becomes subdiffusive for longer times, $g_0 \sim t^x$ ($x \simeq 0.65$). This subdiffusive behavior can be attributed to chain connectivity which determines the monomer dynamics for $g_0 \gtrsim 1$ (= monomer diameter) [70, 109]. If g_0 is comparable to the size of the chain, free diffusion sets in ($g_0 \sim t$).

At low temperature, this scenario is interrupted by an intermediate time window where g_0 increases very slowly ("plateau regime"). This protracted motion reflects a temporary localization of the monomers by their nearest neighbors ("cage") because g_0 is of the order of 10% of the monomer diameter (g_0 is close to $6r_{0,\mathrm{c}}^2$ with $r_{0,\mathrm{c}} \simeq 0.095$ [15, 70]). The intermediate time window is called "β-regime" by MCT [8]. It precedes the final structural relaxation, the "α-regime". In the α-regime a monomer leaves its initial cage ($g_0 \gg 6r_{0,\mathrm{c}}^2$), begins to move subdiffusively due to the bonding to its neighbors, and finally diffuses freely as soon as $g_0 > R_{\mathrm{e}}^2$.

Relative to g_0 the MSD of the center of mass is suppressed by about a factor of $1/N$ in the β-regime. For longer times g_3 directly crosses over to free diffusion. There is no intervening subdiffusive regime because the center of mass is not subject to chain connectivity. These findings are in good agreement with the theoretical predictions in the framework of molecular MCT [19] which we sketched in section 3.11.

4.3 How to Select Mobile Particles

4.3.1 Non-Gaussian Parameters

In the limit $t \to 0$, the monomers behave as if there were no interactions between them. The probability for a displacement r, the self-part of the VAN HOVE correlation function $G_{\mathrm{s}}(\mathbf{r}, t)$, is proportional to the MAXWELL-BOLTZMANN distribution [53]. In the opposite limit $t \to \infty$, the polymers behave as if they were isolated Brownian particles embedded in a heat bath. They diffuse freely. Due to chain connectivity the monomers have to follow the diffusive motion of the center of mass, and $G_{\mathrm{s}}(\mathbf{r}, t)$ is again GAUSSian.

At intermediate times, however, there may be deviations from GAUSSian behavior. A possible means to measure them is the non-GAUSSian parameter $\alpha_2(t)$ [53]

$$\alpha_2(t) \stackrel{\mathrm{def}}{=} \frac{3\left\langle [\mathbf{r}_i(t) - \mathbf{r}_i(0)]^4 \right\rangle}{5\left\langle [\mathbf{r}_i(t) - \mathbf{r}_i(0)]^2 \right\rangle^2} - 1 \; . \tag{4.2}$$

Similar to $\alpha_2(t)$ which quantifies deviations of the monomer dynamics from GAUSSian behavior, one can also measure these deviations for the chain motion by calculating a non-GAUSSian parameter $\alpha_2^{\mathrm{p}}(t)$ for the polymers as

$$\alpha_2^{\mathrm{p}}(t) \stackrel{\mathrm{def}}{=} \frac{3\left\langle [\mathbf{R}_c(t) - \mathbf{R}_c(0)]^4 \right\rangle}{5\left\langle [\mathbf{R}_c(t) - \mathbf{R}_c(0)]^2 \right\rangle^2} - 1 \; . \tag{4.3}$$

By definition the non-GAUSSian parameters vanish in the limits $t \to 0$ and $t \to \infty$. Otherwise, they are bound from below because the mean-quartic displacement is always larger than or equal to the square of the mean-square displacement ($\alpha_2(t), \alpha_2^{\mathrm{p}}(t) \geq -0.4$). A negative value of the parameter means that the particles move on average less far than expected for a random walk, whereas a positive value implies that they move farther. The latter case is often observed in computer simulations [110–115] and in experiments [92, 93, 116].

Figure 4.2 shows $\alpha_2(t)$ and $\alpha_2^{\mathrm{p}}(t)$. The non-GAUSSian parameters vanish in the ballistic regime. With increasing time the amplitude of $\alpha_2(t)$ and $\alpha_2^{\mathrm{p}}(t)$ goes up to a maximum which occurs at $t^*_{\alpha_2}$ for $\alpha_2(t)$ and at $t^*_{\alpha_2^{\mathrm{p}}}$ for $\alpha_2^{\mathrm{p}}(t)$. Notice that the peak

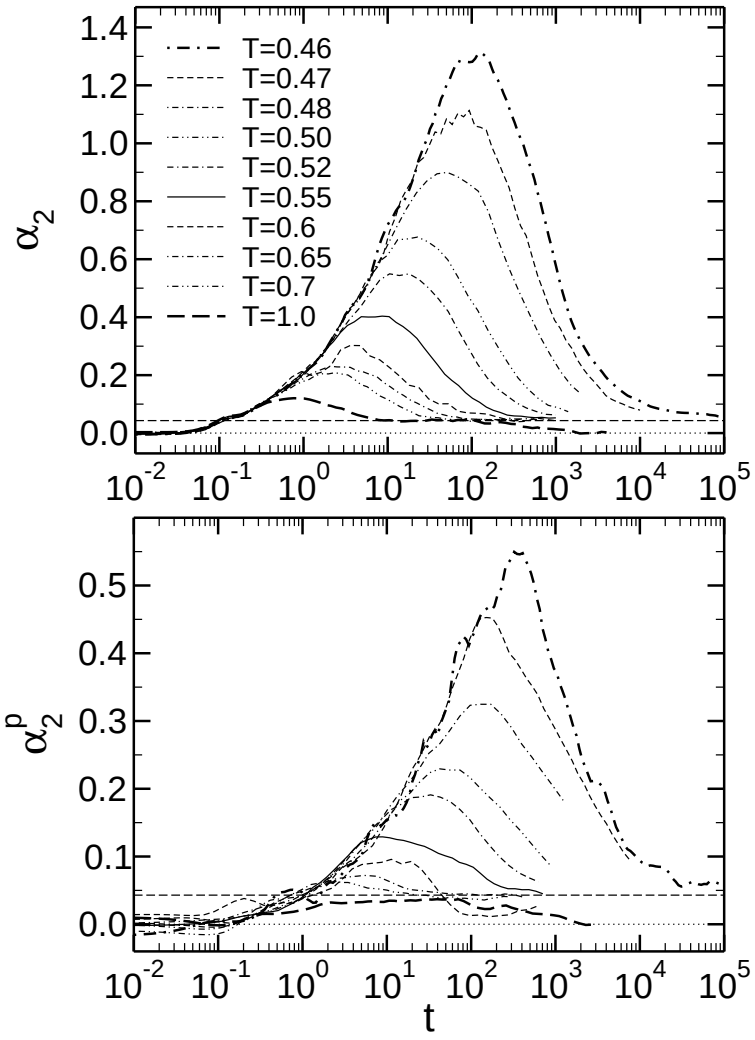

Figure 4.2: Non-GAUSSian parameter of the monomers $\alpha_2(t)$ (upper panel) and of the center of mass of the polymers $\alpha_2^{\rm p}(t)$ (lower panel) versus time for different temperatures. $\alpha_2(t)$ and $\alpha_2^{\rm p}(t)$ are defined in Eqs. (4.2,4.3). The temperatures range from the high-T, normal liquid state ($T = 1$; thick dashed lines in both panels) to the supercooled state of the melt slightly above $T_{\rm c} \simeq 0.45$ ($T = 0.46$; thick dash-dotted lines). Temperature decreases from the bottom curve to the top curve in both panels. The dashed horizontal line (= 0.043) in the upper panel indicates a possible intermediate plateau toward which all $\alpha_2(t)$-curves could converge. This line is also included in the lower panel.

height is larger for $\alpha_2(t)$ than for $\alpha_2^{\rm p}(t)$ and that $t^*_{\alpha_2}$ and $t^*_{\alpha_2^{\rm p}}$ are different. $t^*_{\alpha_2^{\rm p}}$ is shifted by about half a decade to longer times if $T \leq 0.52$.

The small amplitude of $\alpha_2^{\rm p}(t)$ may be attributed to the difference in packing of the monomers and of the chains. The monomers of our model exhibit an ondulating pair-distribution function $g(r)$ [15] whose shape and range are very similar to those found in simple liquids. In contrast to that, the pair-distribution function $g_{\rm cm}(r)$ for the centers of mass is fairly structureless, according our findings for the static structure for the chains' centers of mass in section 3.9.2, cf. Fig. 3.8 ($g_{\rm cm}$ is the FOURIER-transform of $S^{\rm C}$). This reflects the fact that polymers are soft, strongly

interpenetrating objects. The effective interaction between the centers of mass is weak. If this interaction were zero, there would be no resulting force on the center of mass. Then, the chain would diffuse freely (outside the ballistic regime). The small, non-zero $\alpha_2^{\mathrm{p}}(t)$ at $T = 1$ may thus be related to a weak force arising from the presence of other chains in the volume occupied by a polymer [117].

On cooling the melt toward T_{c} a pronounced maximum occurs for both non-GAUSSian parameters. Since $g_{\mathrm{cm}}(r)$ is (essentially) temperature independent, the maximum of α_2^{p} cannot be attributed to enhanced interchain interactions at low T. The similarity between α_2 and α_2^{p} rather suggests that the coupling between monomer and chain dynamics [19] drives the behavior of $\alpha_2^{\mathrm{p}}(t)$. If the monomers of chain are blocked in their cages, the center of mass cannot move either. On the other hand, if a sufficient number of monomers move far during the time t, a large displacement of the center of mass results. As many monomers of the same chain are involved in this motion, a large displacement of the center of mass should take a longer time than for a single monomer. This might explain why $t^*_{\alpha_2^{\mathrm{p}}}$ is larger than $t^*_{\alpha_2}$.

In addition to the maximum α_2 shows two conspicuous features. First, there is a small, temperature independent step at $t \simeq 0.1$. This time corresponds to the crossover from the ballistic to the plateau regime (Fig. 4.1). Thus, the step can be more or less pronounced, depending on the microscopic properties of the system studied [111, 114, 115, 118]. Second, $\alpha_2(t)$ does not vanish continuously for large times. Rather it seems to relax toward a plateau before going to zero. This behavior is clearly visible for $T = 1.0$, while the other temperatures are only indicative of a similar trend. Figure 4.2 suggests that the plateau value is the same for all T, but that the time when it is reached increases on cooling. For $T = 1$ the plateau is attained if the MSD of all monomers is $g_0 \approx 1$, whereas, for $T = 0.46$, it is only reached if $t \gtrsim 10^5$. This time corresponds to displacements of the order of the chain size ($R_{\mathrm{g}}^2 < g_0 < R_{\mathrm{e}}^2$) for $T = 0.46$ (Fig. 4.1). Because the motion of the monomers becomes diffusive for $g_0 \gg R_{\mathrm{e}}^2$, one can speculate that the length of the plateau decreases with T.

For all temperatures the plateau occurs if $g_0 \gtrsim 1$. This corresponds to times where the ROUSE model [20] is believed to describe the dynamics of non-entangled chains in the melt [119]. In this model the displacements of the monomers and of the chains are GAUSSian distributed. Thus, $\alpha_2(t)$ and $\alpha_2^{\mathrm{p}}(t)$ should vanish at all times. The finite value of the plateau points to small, but systematic deviations from ROUSE behavior. This value is approximately the same for both α_2 and α_2^{p}, and roughly agrees with the maximum of α_2^{p} found at $T = 1$. The latter observation could imply that the occurrence of the plateau is related to the weak interactions between the centers of mass alluded to above.

In summary, the interpretation of Fig. 4.2 suggests that deviations from GAUSSian behavior in our model might have two origins: The weak, temperature independent interaction between the centers of mass leads to small deviations in the sub-diffusive regime. Preceding the sub-diffusive regime strong, T-dependent deviations occur due to the motion of the monomers in their nearest-neighbor cages and due the escape from them. This drives the sluggish glass-like relaxation of the monomer and, as a consequence, also that of the center of mass. Therefore, the subsequent analysis will focus on the monomer dynamics.

4.3.2 Definition of Mobility

The most prominent feature of α_2 is the pronounced maximum. At low temperatures $t^*_{\alpha_2}$ lies in the crossover regime from the late-β to the early-α process [6, 8] (see Fig. 4.1). The large positive amplitude of $\alpha_2(t^*_{\alpha_2})$ indicates that the onset of the α-relaxation involves displacements which are much larger than expected for a random walk. Thus, $t^*_{\alpha_2}$ is a good time to select mobile particles.

In the following we characterize the "mobility" of a monomer by the magnitude of its displacement at time t_μ,

$$\mu_i(t_\mu) \stackrel{\text{def}}{=} |\mathbf{r}_i(t_\mu) - \mathbf{r}_i(0)| \; . \tag{4.4}$$

A monomer is said to be "mobile" if it is among the 6.5% with the largest $\mu_i(t_\mu)$ at time t_μ. We denote the resulting set of mobile monomers by $\mathcal{M}_{\rm m}(t_\mu)$. This definition does not necessarily imply large displacements in absolute terms. It just selects those monomers which move farthest in some time interval at a given temperature. At short times or low temperatures the actual distance covered may be small compared to long times or high temperatures.

A practical advantage is the easy implementation: One just sorts all monomers according to the modulus of their displacement and finds the 6.5% with the largest $\mu_i(t_\mu)$. Nevertheless, this choice seems fairly arbitrary and the question arises to what extent the results depend on it. A thorough study of this problem has been performed in Ref. [26]. This work suggests that 6.5% is a reasonable choice due to the following observation: The first approach to define mobile particles [100, 102] employed the criterion that a particle has to move further than some distance r^* in the time $t_\mu = t^*_{\alpha_2}$. Since this choice identifies t_μ with the time where deviations from GAUSSian behavior are maximum, it is natural to determine r^* in a similar way, for instance, by comparing the simulated $G_{\rm s}(\mathbf{r}, t)$ with its GAUSSian approximation. The comparison shows that the GAUSSian approximation underestimates the probability of large displacements. The distance r^* can therefore be defined by the intersection point of both distributions in the tail region [100, 102]. Applying this approach to the present polymer model one finds that the fraction of monomers in the tail beyond r^* ranges between 6.2% and 6.8%. Thus, 6.5% is a good compromise.

Nevertheless, this definition of mobility seems to have a drawback: If a monomer first moves far away from its origin and then returns at the end of the time window $[0, t_\mu]$, it will not be classified as mobile. The selection is made at t_μ and ignores the history of motion between the start and end points. To circumvent this problem a different definition was utilized in Ref. [102]: A particle is called "mobile" provided its maximum displacement,

$$\widetilde{\mu}_i(t_\mu) \stackrel{\text{def}}{=} \max_{0 \le t \le t_\mu} \left[|\mathbf{r}_i(t) - \mathbf{r}_i(0)| \right] \; , \tag{4.5}$$

occurring at some time $0 \le t \le t_\mu$, belongs to the 5% of largest displacements found in the interval $[0, t_\mu]$.

In order to decide which definition to use some of the quantities to be discussed below were calculated for both choices (4.4) and (4.5). The comparison revealed no qualitative differences between the two definitions, a result also found for a binary LJ-mixture [102]. Therefore, we only use the computationally simpler criterion (4.4) in the following.

4.4 Properties of Mobile Monomers

4.4.1 Mean-Square Displacement of Mobile Monomers

We first characterize the motion of the mobile monomer subset, and investigate how that motion is affected by the choice of t_μ. To this end, we calculate a specially defined MSD $g_{0,\mathrm{m}}(t_\mu, t)$, which calculates the MSD, as a function of time, of a subset of monomers that are identified to be highly mobile in a specific time window t_μ,

$$g_{0,\mathrm{m}}(t_\mu, t) \stackrel{\mathrm{def}}{=} \frac{1}{|\mathcal{M}_\mathrm{m}(t_\mu)|}\left\langle \sum_{i\in\mathcal{M}_\mathrm{m}(t_\mu)} \left[\mathbf{r}_i(t) - \mathbf{r}_i(0)\right]^2 \right\rangle . \tag{4.6}$$

Here, $|\mathcal{M}_\mathrm{m}(t_\mu)|$ $(= 0.065 \times M)$ denotes the total number of particles in the subset of mobile monomers at time t_μ, $\mathcal{M}_\mathrm{m}(t_\mu)$. For all times the number $|\mathcal{M}_\mathrm{m}(t_\mu)|$ remains fixed, although the composition of $\mathcal{M}_\mathrm{m}(t_\mu)$ changes with t_μ – at different times different monomers will be among the 6.5% with the largest displacement. By fixing t_μ and varying t we fix this composition for all t and follow the motion of only those monomers that have been found to be mobile at t_μ. However, we can also calculate the MSD of the mobile monomers $g_{0,\mathrm{m}}(t)$, as done in Ref. [26]; this corresponds to the choice $t_\mu = t$ in Eq. (4.6). In that case, the composition of $\mathcal{M}_\mathrm{m}(t)$ may vary for different times.

Figures 4.3a and 4.3b compare $g_{0,\mathrm{m}}(t)$ with the average MSD, $g_0(t)$, of the melt at $T = 1$ and $T = 0.46$. Qualitatively, $g_0(t)$ and $g_{0,\mathrm{m}}(t)$ exhibit the same behavior at both temperatures. However, there are quantitative differences. As expected, the mobile particles always move significantly farther than the average. At early and very late times, the ratio $g_{0,\mathrm{m}}(t)/g_0(t)$ is approximately 3.14 (Fig. 4.3c). This value can be understood from the GAUSSian behavior of the displacements in the ballistic and diffusive regimes. If $G_\mathrm{s}(\mathbf{r}, t)$ is a GAUSSian, we have

$$G_\mathrm{s}(\mathbf{r}, t) = \left(\frac{3}{2\pi g_0(t)}\right)^{3/2} \exp\left[-\frac{3\mathbf{r}^2}{2g_0(t)}\right] . \tag{4.7}$$

The fraction of the 6.5% of the most mobile monomers can then be expressed as:

$$0.065 = 4\pi \int_{r^*}^{\infty} \mathrm{d}r\; r^2 G_\mathrm{s}(\mathbf{r}, t) = (2/\sqrt{\pi})\Gamma(3/2, \sqrt{u^*}) , \tag{4.8}$$

where $u = 3r^2/2g_0$ and $\Gamma(\alpha, x)$ is the incomplete gamma function. This condition yields $\sqrt{u^*} \simeq 3.614$. Similarly, we can write for the MSD of the mobile monomers

$$g_{0,\mathrm{m}}(t) = (4\pi/0.065) \int_{r^*}^{\infty} \mathrm{d}r\; r^4 G_\mathrm{s}(\mathbf{r}, t) = (4/0.065\sqrt{9\pi}) g_0(t) \Gamma(5/2, \sqrt{u^*}) . \tag{4.9}$$

Using the identity $\Gamma(\alpha + 1, x) = \alpha\Gamma(\alpha, x) + x^\alpha \mathrm{e}^{-x}$ we arrive at

$$\frac{g_{0,\mathrm{m}}(t)}{g_0(t)} = 1 + \frac{4}{0.065\sqrt{9\pi}} (x^*)^{3/2}\, \mathrm{e}^{-x^*} , \tag{4.10}$$

where x^* is determined by the normalization condition $(4/\sqrt{\pi}) \int_{x^*}^{\infty} \mathrm{d}x\, x^2 \exp(-x^2) = 0.065$. This yields $x^* \simeq 3.6136$ so that $g_{0,\mathrm{m}}(t)/g_0(t) \simeq 3.143$.

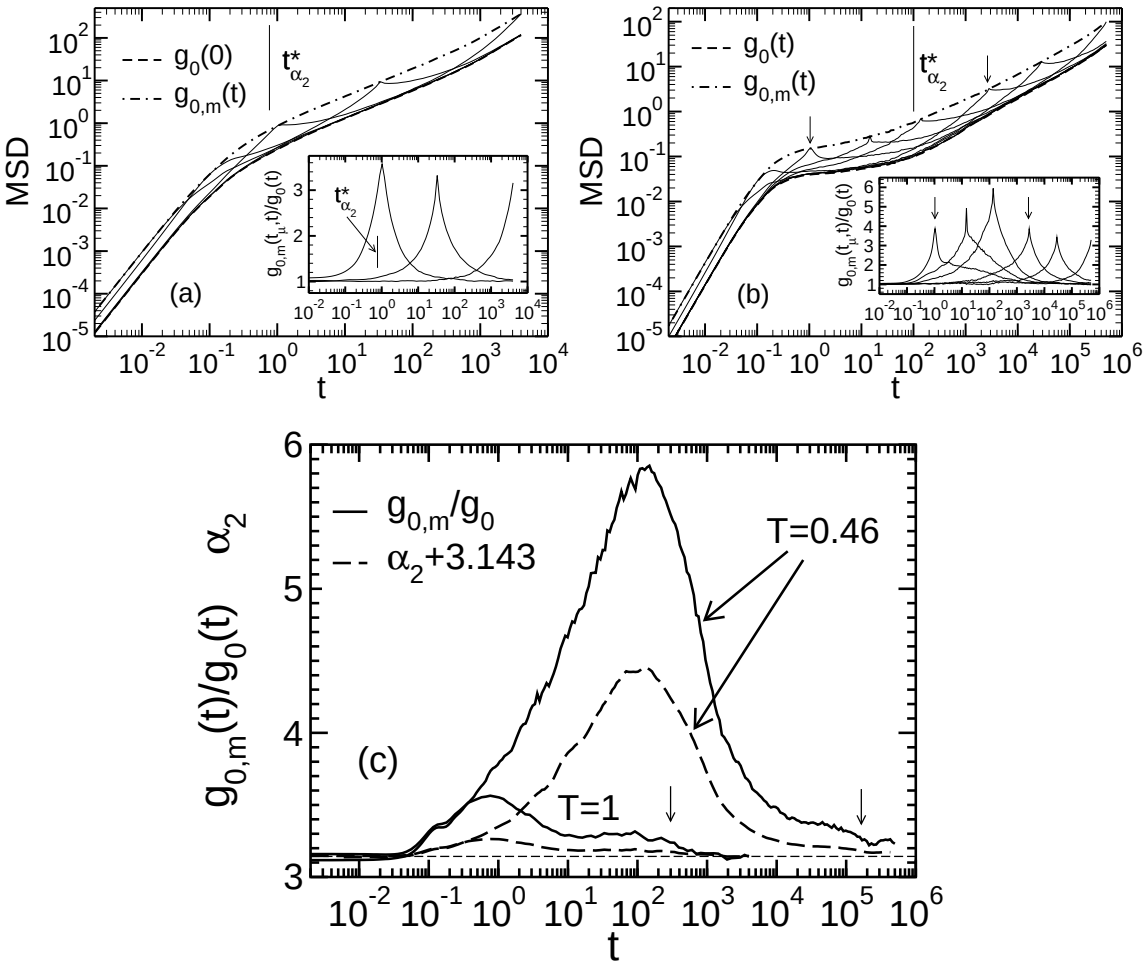

Figure 4.3: Comparison of various MSD's at $T = 1$ [panel (a), top left] and $T = 0.46$ [panel (b), top right]. The thick dashed curve is the average MSD, $g_0(t)$, of all monomers in the melt, whereas the dash-dotted curve represents the MSD of all mobile monomers, $g_{0,\mathrm{m}}(t)$. $g_{0,\mathrm{m}}(t)$ results from the concatenation of the tips of the thin solid curves. Each of these solid curves shows the time evolution of the MSD $g_{0,\mathrm{m}}(t_\mu, t)$ which is obtained by averaging over all those monomers that are mobile at $t = t_\mu$ (= time when the tip occurs). The vertical solid lines in both panels show the time $t^*_{\alpha_2}$ where the non-GAUSSian parameter α_2 is maximum. In panel (b) the vertical arrows indicate two times t_μ, one at the beginning of the plateau ($t_\mu = 1.038$) and in the α-regime ($t_\mu = 2634$). The insets in panels (a) and (b) illustrate the difference between the average MSD and that of the mobile particles by showing $g_{0,\mathrm{m}}(t_\mu, t)/g_0(t)$ for t_μ well beyond the ballistic regime. Panel (c) compares $g_{0,\mathrm{m}}(t)/g_0(t)$ to $\alpha_2(t)$. The non-GAUSSian parameter was shifted by 3.143 [dashed horizontal line, see Eq. (4.10)]. The vertical arrows indicate the time when $g_0 = R_\mathrm{e}$ for $T = 1$ and $T = 0.46$, respectively.

At intermediate times, $g_{0,\mathrm{m}}(t)/g_0(t)$ is larger than 3.143 and behaves qualitatively in the same way as α_2 (Fig. 4.3c). This finding is not unreasonable. The mean-quartic displacement is more sensitive to large displacements than g_0. Thus, it particularly samples the large-r wing of $G_\mathrm{s}(\mathbf{r},t)$ similar to $g_{0,\mathrm{m}}$. In particular, $g_{0,\mathrm{m}}(t)/g_0(t)$ exhibits a maximum at $t = t^*_{\alpha_2}$, the amplitude of which increases with decreasing T. This illustrates again that the most mobile monomers move farther than expected for a random walk. The difference in mobility between all and the fastest monomers grows on cooling and is most pronounced at the beginning of the α-process.

Figure 4.3 also shows the MSD $g_{0,\mathrm{m}}(t_\mu,t)$ which illustrates how the monomers that are mobile at t_μ move at other times. Apparently, the mobile monomers do not belong to a special class of particles which are always faster than the average. Rather they behave as the average for $t \ll t_\mu$, accelerate as t approaches t_μ, and finally relax back to the average for $t \gg t_\mu$. This cycle is almost symmetric on a log-scale at $T = 1$ if $t_\mu \gtrsim t^*_{\alpha_2}$, i.e., in the subdiffusive regime [inset of panel (a)]. Similar behavior is found at $T = 0.46$ for times beyond the plateau in the subdiffusive regime [inset of panel (b)]. However, the cycle is fairly asymmetric in the β-regime. If t_μ is at the beginning of the plateau (curve indicated by an arrow at $t_\mu \approx 1$ in Fig. 4.3b), the monomer accelerates fast, but takes a long time to relax back to $g_0(t)$, whereas the behavior is opposite if t_μ is in the α-relaxation regime (curve indicated by an arrow at $t_\mu \approx 2600$ in Fig. 4.3b).

4.4.2 Mobile Monomers and Chain Connectivity

The previous subsection discussed the motion of mobile monomers without distinguishing whether they are connected to each other or not. Of course, the interplay of connectivity and mobility is an interesting issue. To analyze this polymer-specific aspect the following quantities were introduced. Let $|\mathcal{C}_\mathrm{m}(t_\mu)|$ be the number of chains which contain at least one mobile monomer. Then, we define the fraction of end-monomers in the set $\mathcal{M}_\mathrm{m}(t_\mu)$, $f_{\mathrm{e,m}}(t_\mu)$, and the number of mobile monomers per chain averaged over all chains c_m in the set $\mathcal{C}_\mathrm{m}(t_\mu)$

$$N_\mathrm{m}(t_\mu) \stackrel{\mathrm{def}}{=} \left\langle \frac{1}{|\mathcal{C}_\mathrm{m}(t_\mu)|} \sum_{c_\mathrm{m}\in\mathcal{C}_\mathrm{m}(t_\mu)} \big|\{i \in \mathcal{M}_\mathrm{m}(t_\mu),\ i \in c_\mathrm{m}\}\big| \right\rangle . \tag{4.11}$$

Note that N_m does not discriminate between mobile monomers separated along the backbone of the chain and those which are directly connected to one another. The latter property, however, reflects the extent to which mobile monomer excite their bonded neighbors. Therefore, we also calculate the average number of contiguous segments of mobile monomers along the backbone of a chain, $N_{\mathrm{c,m}}(t_\mu)$. In the following we will mainly discuss this latter quantity. Figure 4.4 illustrates these definitions.

4.4.3 Mobile End Monomers

Figure 4.5a shows the time evolution of $Nf_{\mathrm{e,m}}/2$ for all temperatures. The factor $N/2$ takes into account that the *a-priori* probability of finding an end monomer

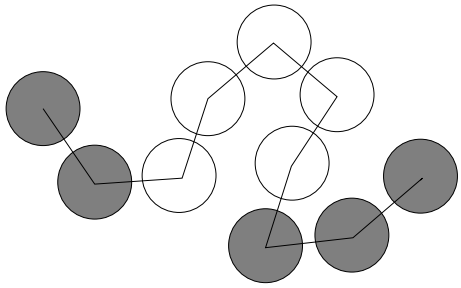

Figure 4.4: An example of how the number of mobile monomers per chain, $N_{\mathrm{m}}(t_\mu)$, and the average length of contiguous segments of mobile monomers, $N_{\mathrm{c,m}}(t_\mu)$, are defined. At time t_μ only the shaded monomers are supposed to be mobile. The first two monomers constitute a contiguous segment of length 2, then there are five non-mobile monomers, followed by another contiguous segment of length 3. So, $N_{\mathrm{m}}(t_\mu)$ is five and the average length of contiguous mobile segments is $N_{\mathrm{c,m}}(t_\mu) = 2.5$.

among the N monomers of a chain is $2/N$. If the mobility of the ends cannot be distinguished from the average, $Nf_{\mathrm{e,m}}/2$ should be 1. This is the case in the ballistic regime, where the monomers are independent of each other, and in the diffusive regime, where they follow the motion of the center of mass. At intermediate times, however, we find $Nf_{\mathrm{e,m}}/2 > 1$. Chain ends are more mobile than inner monomers.

We compare $Nf_{\mathrm{e,m}}/2$ with the ratio $g_4(t)/g_0(t)$, where $g_4(t)$ is the MSD of the end monomers (Fig. 4.5b). The time dependence of $Nf_{\mathrm{e,m}}/2$ closely agrees with that of $g_4(t)/g_0(t)$. This implies that the qualitative relationship between the motion of the mobile ends and that of all mobile monomers is not different from the average behavior of the melt. However, there are quantitative differences. For times outside the ballistic and diffusive regimes $Nf_{\mathrm{e,m}}(t)/2$ is larger than $g_4(t)/g_0(t)$, except in the window of the intermediate plateau (MCT β-process), where $Nf_{\mathrm{e,m}}(t)/2 \approx g_4(t)/g_0(t)$.

When leaving the ballistic regime g_4/g_0 first increases. This increase can be understood by the short-time expansion of the MSD of monomer i [53]

$$\left\langle [\mathbf{r}_i(t) - \mathbf{r}_i(0)]^2 \right\rangle = 2t \int_0^t \mathrm{d}t' \left(1 - \frac{t'}{t}\right) \left\langle \mathbf{v}_i(t) \cdot \mathbf{v}_i(0) \right\rangle \approx 3Tt^2 \left[1 - \frac{\langle |\mathbf{F}_i|^2 \rangle}{36T} t^2\right] \quad (t \text{ small}) \,,$$

where $\langle \mathbf{v}_i(t) \cdot \mathbf{v}_i(0) \rangle$ is the velocity auto-correlation function and $\mathbf{F}_i$ the total force on monomer i. Since an end is only bonded to one monomer, $\mathbf{F}_i$ is smaller than for inner monomers. Thus, one expects $g_4/g_0 > 1$ (and also $Nf_{\mathrm{e,m}}/2 > 1$) for times just outside the ballistic regime. In our model the ratio continues to increase up to a maximum that occurs around $t \approx 0.13$. This is close to the time where the velocity auto-correlation function becomes negative [12]. The inversion of the initial direction of the velocity is caused by rebounding collisions between a monomer and its neighbors. It is typical of dense liquids and must occur in the same way for end and inner monomers. Therefore, the difference in mobility should diminish and g_4/g_0 should decrease. In fact, the simulation shows that, for longer times, g_4/g_0 first decreases toward a minimum and then, at about $t \approx t^*_{\alpha_2}$, crosses over to a steep rise. The rise reaches a maximum close to the time where the MSD of the center of mass equals R_{g}^2. This roughly corresponds to the Rouse time τ_{R} [20] of our model. Thereafter, the transition to free diffusion takes place.

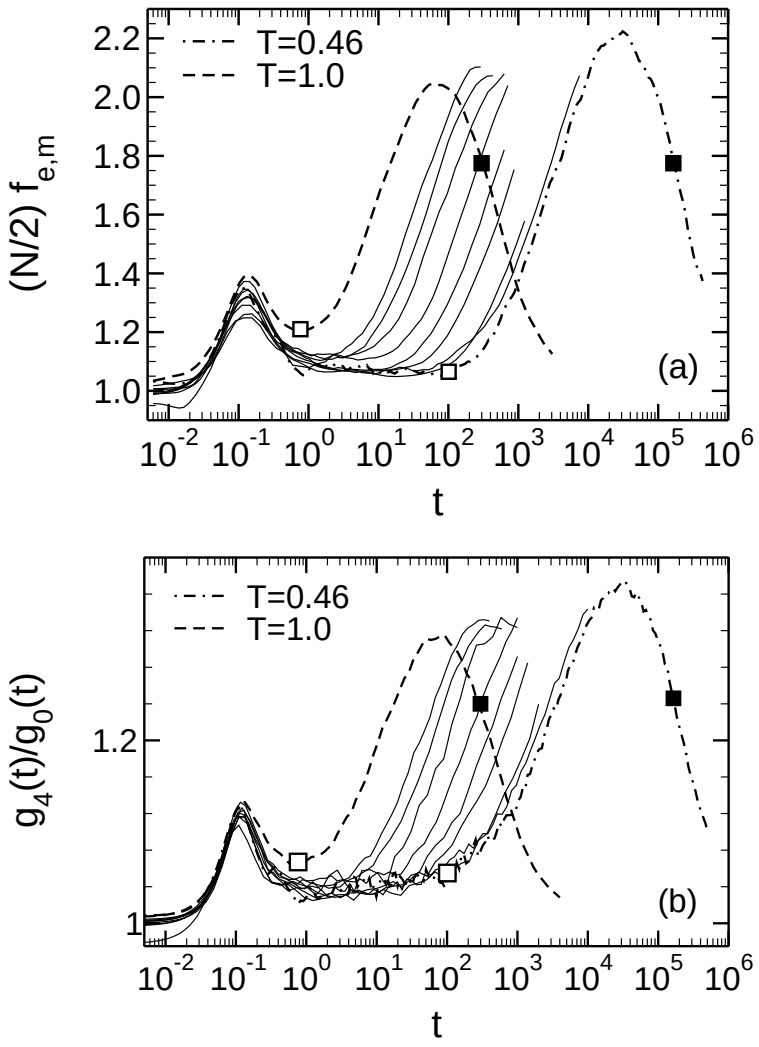

Figure 4.5: Panel (a): Fraction of mobile end monomers $f_{\rm e,m}$ versus t. $f_{\rm e,m}$ is multiplied by $N/2$ (= 5) to account for the fact that there are only two ends per chain. Besides $T = 1$ (dashed curve) and $T = 0.46$ (dash-dotted curve) the following temperatures are shown (solid curves from left to right): $T = 0.7$, 0.65, 0.6, 0.55, 0.52, 0.5, 0.48, 0.47. For $T = 1$ and $T = 0.46$, the open squares indicate $t^*_{\alpha_2}$, the filled squares the time when $g_0 = R^2_{\rm e}$. Panel (b): Same as in panel (a), but for $g_4(t)/g_0(t)$. $g_4(t)$ is the MSD of the end monomers.

The enhanced mobility of the end monomers for $t > t^*_{\alpha_2}$ is not unexpected. The ROUSE theory predicts $g_4/g_0 = 2$ in the time regime where the monomer displacement follows a $t^{1/2}$ behavior (i.e., for $t \leq \tau_{\rm R}$) [20]. In the present simulation, the maximum of g_4/g_0 is smaller than 2, partially due to finite-N effects. Longer chains should attain the ROUSE prediction more closely if entanglements can be neglected [19].

The double-peak structure of g_4/g_0 is present at all temperatures. When cooling the melt toward $T_{\rm c}$ two additional features can be observed: First, the second peak (α-relaxation) strongly shifts to longer times. This is the signature of the slowing down of structural relaxation. Second, the curves collapse in time window of the minimum which deepens and evolves into a protracted plateau with decreasing T.

Previous analysis [15, 70, 108, 109] showed that this time window corresponds to the MCT β-process where one expects temporary intermittence of particle motion due to the cage effect [7, 8]. In the β-regime, $g_n(t)$ is close to the localization length $6r_{n,\mathrm{c}}^2$ [7, 8, 19]

$$g_n(t) = 6r_{n,\mathrm{c}}^2 - 6h_n G(t) \quad (n = 0, 4) \; . \tag{4.12}$$

Here, only the β-correlator $G(t)$ depends on time and temperature, whereas the other parameters $r_{n,\mathrm{c}}$ and h_n are independent of T (close to T_c). Since $6h_nG(t)$ represents a small correction to $6r_{n,\mathrm{c}}^2$, Eq. (4.12) suggests $g_4/g_0 \approx (r_{4,\mathrm{c}}/r_{0,\mathrm{c}})^2$ ($\simeq 1.046$) in the time window of the β-relaxation. Figure 4.5 shows that g_4/g_0 and $Nf_{\mathrm{e,m}}/2$ are indeed close to 1 in this time window. This illustrates that the different bonding of end and inner monomers does not crucially alter the dynamics in the β-regime. Here, the relaxation is determined by the local packing of a monomer and its nearest neighbors, which is (almost) the same for end and inner monomers (in our model).

4.4.4 Correlations of Mobile Monomers in a Chain

Intuitively, one may expect that the bonds in a chain provide a preferred direction along which mobility can be transmitted. To investigate this point we calculated the mean contiguous segment length $N_{\mathrm{c,m}}(t)$, that is, the average number of mobile monomers which are directly connected to each other in a chain (Fig. 4.4).

Figure 4.6 shows $N_{\mathrm{c,m}}(t)$ for all temperatures studied. In the ballistic regime $N_{\mathrm{c,m}} \approx 1.06$. Thus, each of the chains in the set $\mathcal{C}_\mathrm{m}$ mostly contains one mobile monomer. The value $N_{\mathrm{c,m}} \approx 1.06$ could also be obtained by calculating $N_{\mathrm{c,m}}$ after selecting 6.5% of monomers at random and labeling them as "mobile". Therefore, no dynamic correlations between bonded nearest neighbors exist.

Beyond the ballistic regime $N_{\mathrm{c,m}}(t)$ increases, but it never exceeds ~ 1.5 as long as $t \lesssim t^*_{\alpha_2}$. In the studied temperature interval a relaxation mechanism is thus unlikely, in which many consecutive monomers of a chain are mobile and slide along the backbone. Such long-range dynamic correlations would require $N_{\mathrm{c,m}}$ to be of order N. The small value of $N_{\mathrm{c,m}}$ rather suggests that the relaxation in the β-regime is predominantly determined by the dense local packing of the melt and not by chain connectivity. However, this does not imply that chain connectivity is completely irrelevant. For $T \leq 0.7$, $N_{\mathrm{c,m}}(t)$ exhibits a maximum at $t_{\mathrm{seg}}^{\mathrm{max}}$ ($\gtrsim t^*_{\alpha_2}$) in the time window of the late-β/early-α process. The maximum becomes more pronounced on cooling toward T_c. Thus, the colder the melt, the larger the tendency of the mobile monomers to be nearest neighbors in the chain. This trend to cluster is not only observed for monomers bonded to each other, but for all mobile monomers [26] in the late-β/early-α regime.

For times larger than $t_{\mathrm{seg}}^{\mathrm{max}}$ the clustering is suppressed again. The length of the contiguous segments relaxes back to a minimum. The minimum occurs at $t_{\mathrm{seg}}^{\mathrm{min}}$ which roughly corresponds to the time where $g_0 = 1$ (subdiffusive regime). For $t > t_{\mathrm{seg}}^{\mathrm{min}}$, the crossover to free diffusion takes place and $N_{\mathrm{c,m}}$ continuously increases because the displacement of the center of mass is predicated upon a concomitant motion of many monomers in the chain. Presumably, $N_{\mathrm{c,m}}$ reaches a long time limit slightly below N, which is a bound by definition of $N_{\mathrm{c,m}}$.

The occurrence of the maximum and the minimum suggests that there are two

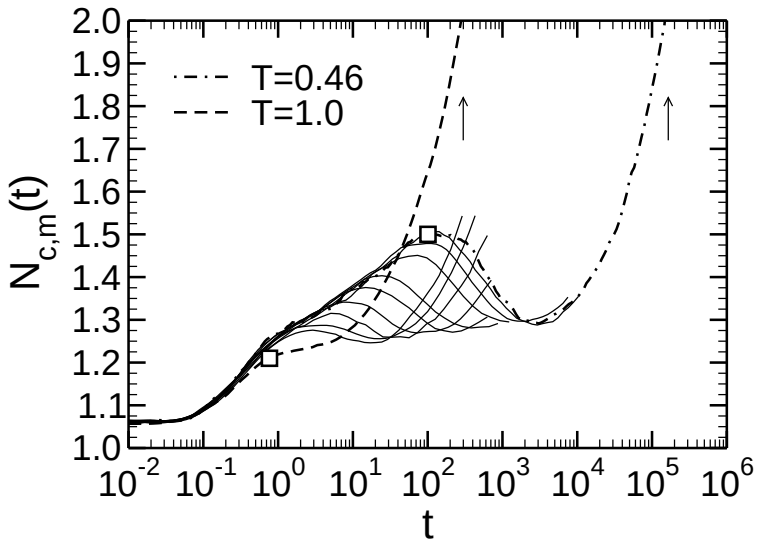

Figure 4.6: Mean contiguous segment length $N_{\mathrm{c,m}}(t)$, i.e., average number of mobile monomers directly bonded to each other, versus time for all temperatures. Besides $T = 1$ (dashed curve) and $T = 0.46$ (dash-dotted curve) the following temperatures are shown (solid curves from left to right): $T = 0.7$, 0.65, 0.6, 0.55, 0.52, 0.5, 0.48, 0.47. For $T = 1$ and $T = 0.46$, the open squares indicate $t^*_{\alpha_2}$ and the arrows the time when $g_0 = R^2_{\mathrm{e}}$.

relaxation mechanisms at low temperature: The maximum corresponds to the cage-breaking process. Here, clustering of highly mobile particles is most pronounced, irrespective of whether they are bonded to each other or not [26]. Cluster formation at the beginning of the α-process is also observed in a binary LJ-mixture close to T_{c} [102] and in experiments on colloidal suspensions close to the glass transition [92]. Thus, chain connectivity is not an imperative premise for clustering. The clustering is rather a consequence of the self-generated cooperativity between the local motion of the caged monomers in the cold melt. To a large extent, this cooperativity is lost as $N_{\mathrm{c,m}}$ crosses over to the minimum. The minimum and the subsequent steep rise are a signature of ROUSE-like, polymer-specific dynamics because they are, at least as precursors, already present at $T = 1$ where no caging occurs.

In addition to $N_{\mathrm{c,m}}$ we also analyzed the average number of mobile monomers in a polymer, $N_{\mathrm{m}}(t)$ [see Eq. (4.11)]. Since mobile monomers need not necessarily be connected to each other, $N_{\mathrm{m}}(t)$ is larger than $N_{\mathrm{c,m}}(t)$. Qualitatively, however, both quantities are alike. Within 10% accuracy we find $N_{\mathrm{c,m}}(t)/N_{\mathrm{m}}(t) = 0.8$ for all times and temperatures. Thus, about 80% of the mobile monomers of a chains are connected nearest neighbors.

4.4.5 Characteristic Times

Quantities, such as $\alpha_2(t)$ or $N_{\mathrm{c,m}}(t)$, exhibit a maximum at some time in the window of the α-process. Ideal mode-coupling theory predicts that these times, as well as any other time from this window, may be chosen as a characteristic relaxation time of the α-process (see, e.g., [7, 8]). The relaxation times τ_A (cf. section 3.11) of

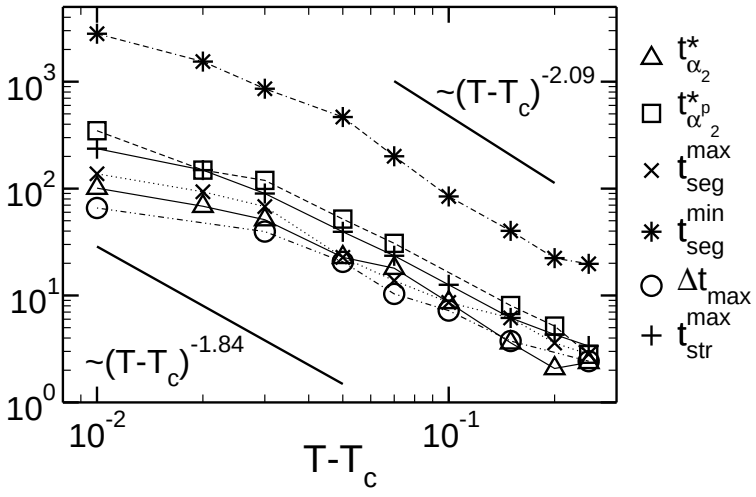

Figure 4.7: Log-log plot of characteristic time scales versus $T - T_c$ ($T_c = 0.45$). The times are: $t^*_{\alpha_2}$ (maximum of α_2), $t^*_{\alpha_2^p}$ (maximum of α_2^p), t^{max}_{seg} (maximum of $N_{c,m}$), t^{min}_{seg} (minimum of $N_{c,m}$), Δt_{max} (maximum of the cluster size of mobile monomers; taken from Ref. [26]), t^{max}_{str} (maximum of $\langle s \rangle$, see Fig. 4.8). The solid lines labeled by the power law (4.13) indicate the temperature dependence of the α-relaxation time resulting from a quantitative MCT analysis ($\gamma = 2.09$ [15, 70]) and of the inverse diffusion coefficient ($\gamma = 1.84$ [15, 109]).

different quantities "A" should only differ in prefactors, but scale with temperature in the same way

$$\tau_A = \tau_A^0 \left(\frac{T - T_c}{T_c} \right)^{-\gamma} \quad (T \to T_c^+) \, . \tag{4.13}$$

Provided that T is sufficiently close to T_c, the exponent γ is predicted to be independent of A. Contrary to that, the range of validity of Eq. (4.13), i.e., the upper bound up to which the power law is observable, depends on A [64, 118].

Fits to simulation data often show that Eq. (4.13) only holds approximately [110, 120]. Very close to T_c deviations occur in (almost) all systems (colloidal suspensions are an exception [94, 95, 98]). These deviations may be rationalized in the framework of MCT by relaxation mechanisms, other than the cage effect, which are not incorporated in the idealized theory [59, 121]. In the remaining temperature interval where Eq. (4.13) can be applied the fits often find the same result for T_c, whereas γ can depend on A.

For the present model, for instance, the relaxation times derived from intermediate scattering functions yield a γ that decreases with the modulus of the wave vector $\mathbf{q}$ and approaches γ_D, the exponent of the diffusion coefficient, in the low-q limit [16, 108]. Since quantities like the mean-square displacements or the non-Gaussian parameters are related to the small-q behavior of the intermediate scattering functions [19, 118], we expect that relaxation times derived from these quantities exhibit a temperature dependence which is compatible rather with γ_D than with the exponent found at the maximum of the static structure factor.

Figure 4.7 shows the temperature dependence of various time scales introduced in this study, including also the time Δt_{max} of Ref. [26] where the cluster size of mobile

monomers is maximum. As expected, deviations from the power law (4.13) are found for T close to T_c (i.e., $T - T_c \lesssim 0.02$). For larger temperatures the relaxation times roughly agree with Eq. (4.13). Within the (admittedly large) statistically uncertainties of the data and perhaps with the exception of $\Delta t_{\max}$ which seems to show a weaker increase [26] the exponents are closer to $\gamma_D = 1.84 \pm 0.02$ [16] than to the result obtained at the maximum of the static structure factor ($\gamma \approx 2.09$ [15, 70]), as suggested above.

4.4.6 Stringlike Motion

The cluster analysis of Ref. [26] showed that the spatial correlations between highly mobile monomers closely agree with those found in a binary Lennard-Jones mixture [102]. Reference [102] also reveals that the clusters consist of smaller objects called "strings". A string is a set of mobile particles moving in one direction. Due to the similarity of the results obtained from the cluster analyses of the polymer melt and the LJ-mixture we also expect to find strings here.

To define a string we first applied the prescription proposed in Ref. [101]: Two particles i and j, which are mobile at time t, are connected to a string if

$$\min\big[|\mathbf{r}_i(t) - \mathbf{r}_j(0)|, |\mathbf{r}_j(t) - \mathbf{r}_i(0)|\big] < \delta \; . \tag{4.14}$$

This equation means that one particle, say i, moved from $\mathbf{r}_i(0)$ to $\mathbf{r}_i(t)$ in time t, while the other particle simultaneously approached the initial position of i within a sphere of radius δ. δ must be sufficiently smaller than the Lennard-Jones diameter σ $(= 1)$ to guarantee that j unambiguously replaces i. For the binary LJ-mixture a good choice was $\delta = 0.6$ [101].

For the polymer model under consideration we found that Eq. (4.14) with $\delta = 0.6$ can lead to ambiguities. Sometimes several replacements are equally possible so that it is not clear which one to take. To circumvent this problem a different definition of a string was introduced. Particle i replaces particle j and j is in the same string as i if

$$j = \arg \min_{\{j|\,|\mathbf{r}_i(t) - \mathbf{r}_j(0)| < \delta\}} \big[|\mathbf{r}_i(t) - \mathbf{r}_j(0)|\big] \; , \tag{4.15}$$

and similarly, j replaces i and i is in the same string as j if

$$i = \arg \min_{\{i|\,|\mathbf{r}_j(t) - \mathbf{r}_i(0)| < \delta\}} \big[|\mathbf{r}_j(t) - \mathbf{r}_i(0)|\big] \; . \tag{4.16}$$

Out of several possible replacements Eqs. (4.15) and (4.16) add that particle to an already existing string, whose distance to the end of the string is shortest. The shortest possible stringlength is 1, meaning an isolated mobile monomer. We have considered the possibility of loops which would occur if a set of particles replaced each other in a circular fashion. In our analysis loops occurred very seldomly. Mostly, they were of length two (or four) and consisted of contiguous chain segments of length two. This means that bonded, neighboring monomers switch places. Because the loops were so rare, we decided not to analyze the results for open and closed strings separately.

For $\delta \leq 0.45$ Eq. (4.14) and the new definition give the same result. However, they become (slightly) different for larger δ because two or more particles fulfil the

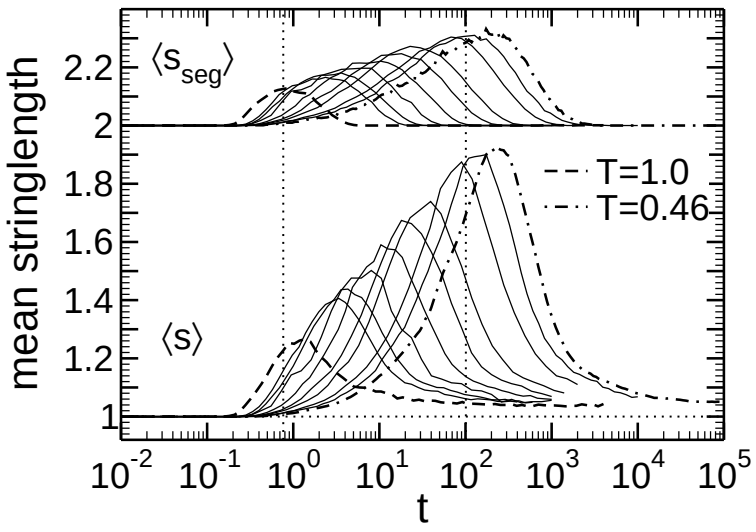

Figure 4.8: Average stringlength $\langle s(t)\rangle$ of all mobile monomers and average stringlength in contiguous segments of mobile monomers $\langle s_{\rm seg}(t)\rangle$ versus t with replacement parameter $\delta = 0.55$. For clarity the data for $\langle s_{\rm seg}(t)\rangle$ are shifted upwards by 1. The temperatures shown are (from left to right): $T = 1$, 0.7, 0.65, 0.6, 0.55, 0.52, 0.5, 0.48, 0.47, 0.46. The dotted vertical lines indicate the time when α_2 is maximum for $T = 1$ ($t^*_{\alpha_2} = 0.766$) and $T = 0.46$ ($t^*_{\alpha_2} = 100.894$). A stringlength of 1 means an isolated mobile monomer, i.e., no "bond" could be formed between two mobile monomers via the replacement criterion of Eqs. (4.15,4.16).

criterion (4.14). If this happens frequently, a "string" contains many Y-like portions, i.e., two particles j are equally likely to occupy the initial position of i. Then, the postulated one-dimensionality of the string is not preserved. For $\delta = 0.6$ we found such ambiguities in 1%, for $\delta = 0.55$ only in 0.2% of the replacements. Since we wanted to avoid these ambiguities as much as possible while still keeping a value close to that used in Ref. [101], $\delta = 0.55$ was chosen for the subsequent analysis.

Figure 4.8 shows the time evolution of the stringlengths averaged over all mobile monomers, $\langle s(t)\rangle$, and over the mobile monomers that are directly connected to each other in a chain, $\langle s_{\rm seg}(t)\rangle$. Qualitatively, both quantities behave in the same way. At short times, $\langle s(t)\rangle$ and $\langle s_{\rm seg}(t)\rangle$ are equal to 1. A stringlength of 1 means that all mobile monomers are separated from each other. For a replacement to occur the shortest distance a monomer must have covered is the nearest-neighbor distance (≈ 1) minus δ. So, roughly 0.45. Figure 4.3 shows that such a displacement ($g_{0,\rm m} \approx 0.2$) takes about $t \approx 0.2$ at $T = 1$ and about $t \approx 2$ at $T = 0.46$. These estimates are close to the times at which the stringlengths start increasing.

A stringlength larger than 1 implies that mobile monomers tend to replace each other. This trend is present at all temperatures, but becomes more pronounced on cooling. It is maximum at a time which is slightly shorter for $\langle s_{\rm seg}(t)\rangle$ than for $\langle s(t)\rangle$ (perhaps due to finite-N effects). We determined this time $t^{\rm max}_{\rm str}$ for $\langle s(t)\rangle$ (Fig. 4.7). A glance at Fig. 4.3 shows that $t^{\rm max}_{\rm str}$ represents a time where $g_{0,\rm m} \approx 1$. Thus, $\langle s(t)\rangle$ is maximum if a monomer displaces on average over its own size and another monomer moves into its position ($\langle s(t^{\rm max}_{\rm str})\rangle \approx 2$). An important contribution to this one-particle replacement comes from bonded neighbors, $\langle s_{\rm seg}(t^{\rm max}_{\rm str})\rangle / \langle s(t^{\rm max}_{\rm str})\rangle \approx 2/3$.

Does this mean that the connectivity is important or would randomly assigning bonds to pairs of mobile particles would lead to a similar value? String replacements happen preferably between next neighbors, and in a 3-d close packed system, there are $N_{\rm nn} \approx 12$ of them, leading to $MN_{\rm nn}/2$ possible nearest neighbor pairs. There are $(N-1)n$ bonds in the system. The probability of randomly finding a bonded pair (if bonds had no physical meaning) is then given by the ratio of bonds to neighbor pairs, $[(N-1)n]/(nNN_{\rm nn}/2) = [2(1-1/N)]/N_{\rm nn} \approx 0.15$. Assuming that this probability stays constant, a string of length $s \geq 2$ will thus lead with a probability of 0.15^{s-1} to a segment $s_{\rm seg} = s$ of the same length. As this value is small compared to the ratio $\langle s_{\rm seg}(t_{\rm str}^{\rm max})\rangle / \langle s(t_{\rm str}^{\rm max})\rangle$, we conclude that physical chain connectivity is important. However, as $\langle s\rangle > \langle s_{\rm seg}\rangle$ strings will consist of several segments, especially for long strings.

For times larger than $t_{\rm str}^{\rm max}$, $\langle s_{\rm seg}(t)\rangle$ relaxes back to its initial value 1, whereas $\langle s(t)\rangle$ asymptotically tends to a slightly larger value (~ 1.04). This disparity is related to the definition of both stringlengths which gives rise to a different large-t limit in the diffusive regime. We can explain this difference by the following argument: Let $P_\delta(|\mathbf{r}_i(t) - \mathbf{r}_j(0)| < \delta)$ denote the probability that a mobile monomer i at time t approaches the initial position of another mobile monomer j to within the range δ. Then, $(0.065M-1)P_\delta$ is the average number of mobile monomers that satisfy the criterion of string formation. Hence, the average stringlength is given by

$$\langle s(t)\rangle = 1 + (0.065M - 1)P_\delta \; . \tag{4.17}$$

For P_δ we can write

$$\begin{aligned} P_\delta &= \int_{\varepsilon<\delta} \mathrm{d}\varepsilon \, \Big\langle \delta\big(\big[\mathbf{r}_i(t) - \mathbf{r}_j(0)\big] - \varepsilon\big)\Big\rangle \\ &= \int_{\varepsilon<\delta} \mathrm{d}\varepsilon \, \Big\{ \int \mathrm{d}\mathbf{r}_i(t)\mathrm{d}\mathbf{r}_j(0) \; \delta\big(\big[\mathbf{r}_i(t) - \mathbf{r}_j(0)\big] - \varepsilon\big) \\ &\qquad\qquad G_{\rm s}\big(\mathbf{r}_i(t) - \mathbf{r}_i(0), t\big)\Big[\frac{\rho}{M}\, g\big(\mathbf{r}_i(0) - \mathbf{r}_j(0)\big)\Big]\Big\} \quad (t \text{ large}) \; , \end{aligned} \tag{4.18}$$

where $G_{\rm s}(\mathbf{r}, t)$ and $g(\mathbf{r})$ are the self-part of the VAN HOVE correlation function and the pair-distribution function, respectively. Since $\rho g(\mathbf{r}_{ij})/M$ is the probability density that the two monomers are initially at a distance $\mathbf{r}_{ij} = \mathbf{r}_i(0) - \mathbf{r}_j(0)$, and $G_{\rm s}(\mathbf{r}_i(t) - \mathbf{r}_i(0), t)$ is the probability density for the displacement $\mathbf{r}_i(t) - \mathbf{r}_i(0)$, the product $\rho g G_{\rm s}/M$ gives the probability density for the vector $\mathbf{r}_i(t) - \mathbf{r}_j(0)$, provided the displacement of monomer j is not correlated with $\mathbf{r}_i(0)$. This condition can only be valid for large times. If we perform the integral over the δ-function and use the homogeneity of space, we obtain

$$P_\delta = \frac{\rho}{M} \int_{\varepsilon<\delta} \mathrm{d}\varepsilon \int \mathrm{d}\mathbf{r}_{ij} \; G_{\rm s}\big(\mathbf{r}_{ij} + \varepsilon, t\big) g(\mathbf{r}_{ij}) \quad (t \text{ large}) \; . \tag{4.19}$$

In the diffusive regime the VAN HOVE function is vanishingly small for distances where $g(r)$ varies appreciably. Thus, we may replace $g(r)$ by its large-r limit. This limit is 0 for the polymer pair-distribution function as bonds restrict distances to finite values [15] so that $\langle s_{\rm seg}\rangle = 1$, whereas it is 1 in the case of the melt. Furthermore, since $G_{\rm s}$ is a normalized GAUSSian, the integral over $\mathbf{r}_{ij}$ just gives 1. Thus,

we find $P_\delta = (\rho/M)(4\pi\delta^3/3)$. Together with Eq. (4.17) this yields

$$\langle s \rangle = 1 + \frac{4\pi(0.065M - 1)\delta^3}{3M}\ \rho\ . \tag{4.20}$$

Using $\delta = 0.55$ and $M = 1020$, $\rho = 0.9058$ at $T = 1$ as well as $M = 1200$, $\rho = 1.0378$ at $T = 0.46$ we find $\langle s \rangle = 1.040$ and $\langle s \rangle = 1.046$ at $T = 1$ and 0.46, respectively. These estimates are in good agreement with the simulation data of Fig. 4.8 at large times.

The previous analysis was done with $\delta = 0.55$. When introducing the criterion for defining strings we argued that the precise choice of δ is not crucial, as long as its value is sufficiently small. To illustrate this point Fig. 4.9 shows the temperature dependence of the maximum average stringlength, $\langle s(t_{\rm str}^{\rm max}) \rangle$, for various δ. We find that the strings become longer if δ increases. This is expected, since more particles satisfy the $|\cdot| < \delta$ condition. However, the qualitative features are independent of δ. To support this point further we invoke an analogy, first proposed in Ref. [101], between the strings and equilibrium polymers (see for instance [122] and references therein).

Equilibrium polymers are systems in which the bonds between the monomers are not permanent. They can constantly break and recombine at random points along the backbone of a chain. In chemical equilibrium a melt of these self-assembling polymers is characterized by an exponential distribution of chain length s, $P(s) \sim \exp(-s/\langle s \rangle)$ (if s is large), and by a mean chain length that increases exponentially with the energy gain E obtained by bond formation, $\langle s \rangle \propto \exp(E/T)$.

In our context, the mobile monomers also self-assemble to (short) chains, driven by the sluggish dynamics of the cold melt. The dynamically created bonds can break and recombine at any instant. They are the more likely to form, and thus "the stronger", the larger δ. This suggests a correspondence between δ and E, the simplest assumption being $E \propto \delta$. Figure 4.9 shows that this assumption is not unrealistic. Despite the disparity between the theoretical premise of long chains and the shortness of our strings a reasonable superposition of stringlengths, found for various δ and T, is obtained. This implies that any of the shown values for δ could have been chosen for the present analysis.

From the analogy with equilibrium polymers one would expect that the strings are fairly polydisperse. Figure 4.10 shows the distribution of the stringlength s found at $t_{\rm str}^{\rm max}$. At the highest temperature, $T = 1$, $P(s)$ is a single exponential which decreases rapidly. The most frequent stringlengths are $s = 1, 2$. Their probability remains essentially unchanged on cooling, whereas longer strings occur much more frequently for $T < 1$. The tail of the distribution is roughly exponential, supporting the possible interpretation of strings as equilibrium polymers. Very similar observations were also made in the simulations of the binary LJ-mixture [101].

4.4.7 Directional Correlations Between Neighboring Mobile Monomers

The string analysis of the previous section showed that mobile monomers tend to move to the initial position of other mobile monomers. Since typical displacements of mobile monomers at $t_{\rm seg}^{\rm max}$ are still fairly local (see Fig. 4.3), a replacement is most

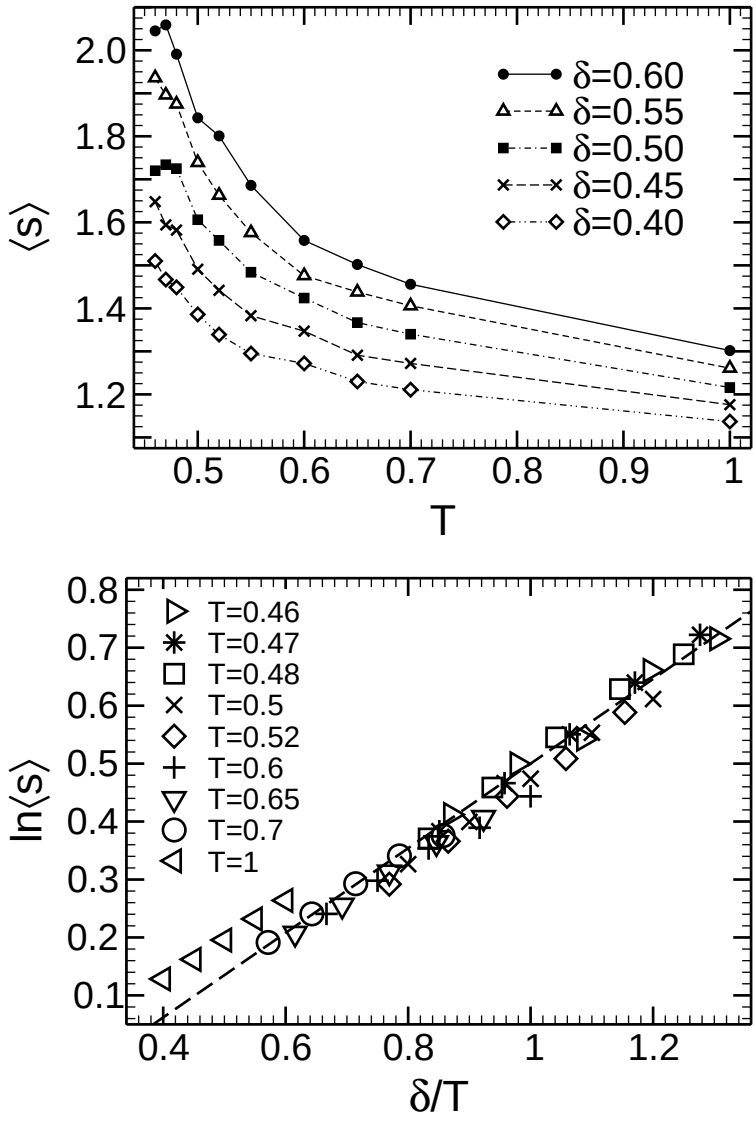

Figure 4.9: Top panel: Average stringlength $\langle s \rangle$ versus T for various δ [see Eqs. (4.15,4.16)]. The stringlength is calculated at $t_{\rm str}^{\rm max}$, where it is maximum. Bottom panel: Rescaling of $\langle s \rangle$ as suggested by the analogy with equilibrium polymers (see text for details). A satisfactory collapse of the data for all T and δ is obtained except for $T = 1$. At this temperature, strings larger than 1 occur very seldomly (see Fig. 4.10). The dashed straight line is a fit through the data for $T \leq 0.7$, yielding $\ln\langle s \rangle = -0.23 + 0.73\ \delta/T$.

likely to occur between nearest neighbors. In this section, we thus concentrate on neighboring mobile monomers and explore their correlated motion.

Let monomers i and j be nearest neighbors at $t = 0$ and mobile at some later time t. We define particle i to be a nearest neighbor of particle j if their initial distance is within the first neighbor shell of the pair-distribution function, i.e., $|\mathbf{r}_j(0) - \mathbf{r}_i(0)| < 1.5$ [15]. The same definition was also used in Ref. [26]. To simplify the notation we write for the distance between two neighbors at time $t = 0$

$$\mathbf{r}_{ij} \stackrel{\rm def}{=} \mathbf{r}_j(0) - \mathbf{r}_i(0)\ , \tag{4.21}$$

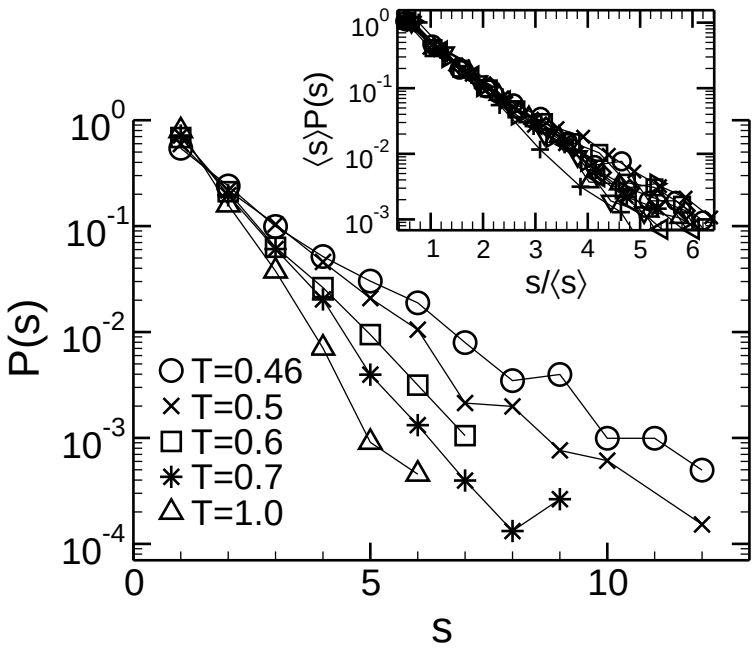

Figure 4.10: Semi-log plot of the probability distribution $P(s)$ of the stringlength s for various T. $P(s)$ is calculated at $t = t_{\rm str}^{\rm max}$ where $\langle s \rangle$ is maximum. It is roughly exponential. Inset: $P(s)$ rescaled by the mean value $\langle s \rangle$ versus $s/\langle s \rangle$. In addition to the temperatures $T = 0.46$, 0.47, 0.48, 0.50, 0.52, 0.55 for $\delta = 0.55$ the graph also includes $T = 0.46$ and 0.55 for both $\delta = 0.4$ and $\delta = 0.6$. The scaling would deteriorate if higher T were included.

and for the displacement of monomer i in time t,

$$\mathbf{d}_i(t) \stackrel{\rm def}{=} \mathbf{r}_i(t) - \mathbf{r}_i(0) \; . \tag{4.22}$$

Now we can define the following angles:

$$\theta\left[\mathbf{d}_i(t), \mathbf{r}_{ij}\right] \stackrel{\rm def}{=} \arccos\left[\frac{\mathbf{d}_i(t) \cdot \mathbf{r}_{ij}}{|\mathbf{d}_i(t)||\mathbf{r}_{ij}|}\right] \quad , \quad \theta\left[\mathbf{d}_i(t), \mathbf{r}_{ij}\right] \in [0, \pi] \tag{4.23}$$

and

$$\theta\left[\mathbf{d}_i(t), \mathbf{d}_j(t)\right] \stackrel{\rm def}{=} \arccos\left[\frac{\mathbf{d}_i(t) \cdot \mathbf{d}_j(t)}{|\mathbf{d}_i(t)||\mathbf{d}_j(t)|}\right] \quad , \quad \theta\left[\mathbf{d}_i(t), \mathbf{d}_j(t)\right] \in [0, \pi] \; . \tag{4.24}$$

These definitions are illustrated in Fig. 4.11. The first angle (4.23) was proposed in Ref. [101]. It indicates to what extent a mobile particle has occupied, at time t, the initial position of one of its neighbors which is also mobile at time t. The second angle (4.24) measures the correlations between the displacements $\mathbf{d}_i(t)$ and $\mathbf{d}_j(t)$ of the monomers at time t. Thus, it shows to what extent two mobile neighbors follow each other.

For both bonded and non-bonded nearest-neighbor pairs we computed the probability distributions $P_{\rm d,r}(\theta\left[\mathbf{d}_i(t), \mathbf{r}_{ij}\right])$ and $P_{\rm d,d}(\theta\left[\mathbf{d}_i(t), \mathbf{d}_j(t)\right])$. Since these probabilities are supposed to detect displacements under preferred angles, one has to take into account that the probability of θ is not uniform even for isotropic, uncorrelated

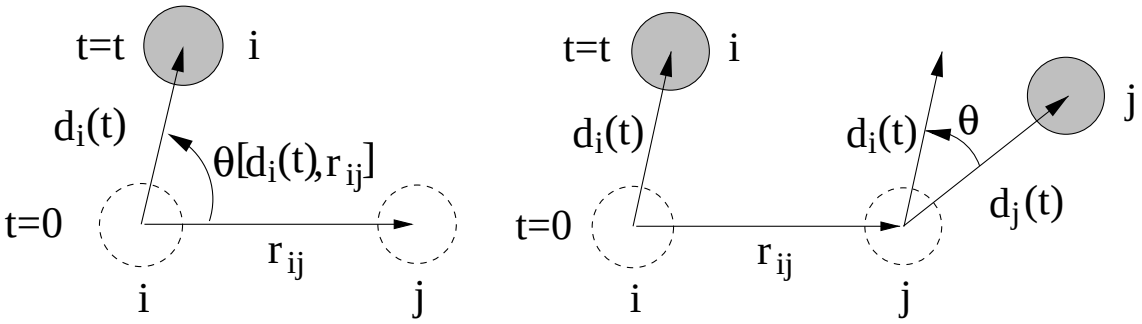

Figure 4.11: Left panel: definition of $\theta\left[\mathbf{d}_i(t), \mathbf{r}_{ij}\right]$, Eq. (4.23). Right panel: definition of $\theta\left[\mathbf{d}_i(t), \mathbf{d}_j(t)\right]$, Eq. (4.24). In both panels, the dashed circles depict the position of the mobile monomers i and j at $t = 0$, whereas the shaded circles represent their positions at time t. The vectors $\mathbf{r}_{ij}$ and $\mathbf{d}_i(t)$ are defined by Eqs. (4.21) and (4.22), respectively.

motion. Suppose we have a vector $\mathbf{d}$ on the unit sphere S^2 connecting the origin to a point on the surface of S^2 in direction $(\theta,\ \phi)$. $\theta \in [0, \pi[$ is the latitude and $\phi \in [0, 2\pi[$ the longitude. Furthermore, let $\hat{\mathbf{n}}$ be the vector from the origin to the north pole ($\theta = 0$, $\phi = 0$). Then, the angles defined in Eqs. (4.23,4.24) correspond to a measurement of the latitude θ, the angle between $\mathbf{d}$ and $\hat{\mathbf{n}}$, after integration over ϕ. Because the sector of the unit sphere for fixed θ is small if θ is close to the poles, but large if it is close to the equator, the ϕ-integrated probability of an isotropic distribution of vectors on S^2, $P_{\rm iso}(\theta)$, is maximum at $\theta = 90°$. More precisely,

$$P_{\rm iso}(\theta) = \frac{1}{2}\sin(\theta)\ . \tag{4.25}$$

In order to determine whether the angle, say, between two particle displacements is preferred relative to purely isotropic motion one has to divide its probability by the isotropic value. In the following analysis, we thus consider the ratios

$$\frac{P_{\rm d,d}(\theta\left[\mathbf{d}_i(t), \mathbf{d}_j(t)\right])}{P_{\rm iso}(\theta\left[\mathbf{d}_i(t), \mathbf{d}_j(t)\right])} \quad \text{and} \quad \frac{P_{\rm d,r}(\theta\left[\mathbf{d}_i(t), \mathbf{r}_{ij}\right])}{P_{\rm iso}(\theta\left[\mathbf{d}_i(t), \mathbf{r}_{ij}\right])}\ . \tag{4.26}$$

Figure 4.12 shows the ratio $P_{\rm d,r}(\theta)/P_{\rm iso}(\theta)$ at $T = 0.47$ for several selected times which are approximately uniformly distributed on a logarithmic scale. At the earliest and the latest time, we find $P_{\rm d,r}(\theta) \approx P_{\rm iso}(\theta)$. This lack of significant correlation between $\mathbf{d}_i(t)$ and $\mathbf{r}_{ij}(0)$ may be interpreted in the following way: For the longest time, $t = 10^4$, Fig. 4.1 shows that $g_0 > R_{\rm g}^2$. Since the mobile monomers move farther than the average (Fig. 4.3), d_i is certainly much larger than $R_{\rm g}$ and thus than r_{ij}. In this case, one would not expect significant correlations between $\mathbf{d}_i$ and $\mathbf{r}_{ij}$. The motion should be isotropic. On the other hand, for very short times the motion is ballistic. The monomers behave as if they were free particles, and any correlation between $\mathbf{d}_i$ and $\mathbf{r}_{ij}$ should vanish again.

Figure 4.12 shows that isotropic motion persists up to approximately $t \approx 0.1$. For longer times deviations occur. In the β/α-regime we find that $P_{\rm d,r}(\theta)$ is symmetric around $\theta = 90°$. The symmetry arises because we calculate $P_{d,r}(\theta\left[\mathbf{d}_i(t), \mathbf{r}_{ij}\right])$ by averaging over the displacements $\mathbf{d}_i(t)$ and $\mathbf{d}_j(t)$ for the same $\mathbf{r}_{ij}$. Thus, the rise of $P_{d,r}(\theta)/P_{\rm iso}(\theta)$ close to $\theta = 0°$ results from motions of particle i in direction of $\mathbf{r}_{ij}$,

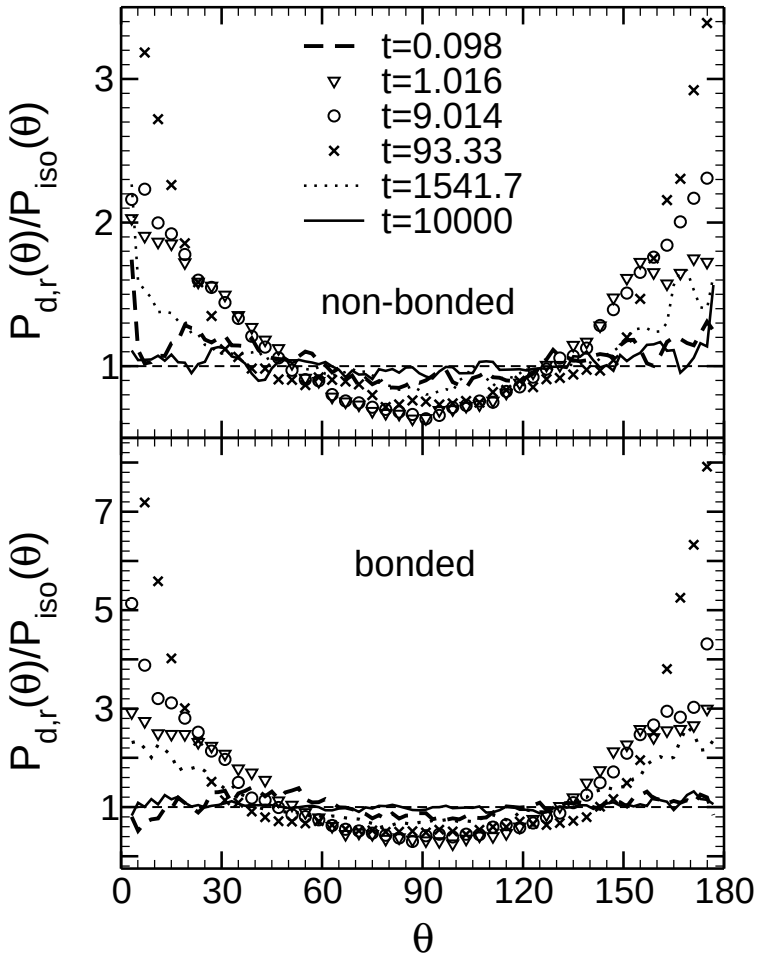

Figure 4.12: Probability distribution $P_{\mathrm{d,r}}(\theta\,[\mathbf{d}_i(t), \mathbf{r}_{ij}])$ of the angle between the displacement vector of a mobile monomer $\mathbf{d}_i(t)$ and of the vector $\mathbf{r}_{ij}$ between the initial positions of the monomer and of its mobile neighbor j. $P_{\mathrm{d,r}}$ is divided by the probability P_{iso} for isotropic motion. The upper figure presents the results for bonded nearest neighbors, the lower for non-bonded nearest neighbors. In both cases, $T = 0.47$. The times shown are separated from each other roughly by a factor of 10: $t = 0.098$ ($\approx$ first maximum of $f_{\mathrm{e,m}}$), $t = 1.016$ (beginning of the β-regime), $t = 9.014$ (center of the β-regime, where $g_0 \simeq 6r_{0,\mathrm{c}}^2$), $t = 93.33$ ($= t_{\mathrm{seg}}^{\mathrm{max}}$), $t = 1541.7$ ($= t_{\mathrm{seg}}^{\mathrm{min}}$), $t = 10^4$ ($\hat{=}\ g_0 > R_{\mathrm{g}}^2$).

whereas that close to $\theta = 180°$ comes from displacements of particle j against the orientation of $\mathbf{r}_{ij}$. Note that the analysis of Ref. [101] only consider the former case. Therefore, the probability distribution is asymmetric (see Fig. 3 of Ref. [101]).

Displacements perpendicular to the axis $\mathbf{r}_{ij}$ are suppressed relative to P_{iso}, whereas motion parallel to it is enhanced. The enhancement is particularly prominent in the late-β/early-α regime ($t \sim t_{\mathrm{seg}}^{\mathrm{max}}$). As the monomers escape from their cages, they tend to follow each other. This stringlike motion is more pronounced for nearest-neighbors along the chain backbone. In the β-regime ($t = 9.014, 93.33$), $P_{\mathrm{d,r}}(\theta)/P_{\mathrm{iso}}(\theta)$ is about a factor of 2 larger in the bonded than in the non-bonded case. This shows that chain connectivity promotes stringlike motion as we saw from the ratio $\langle s_{\mathrm{seg}}\rangle/\langle s\rangle$.

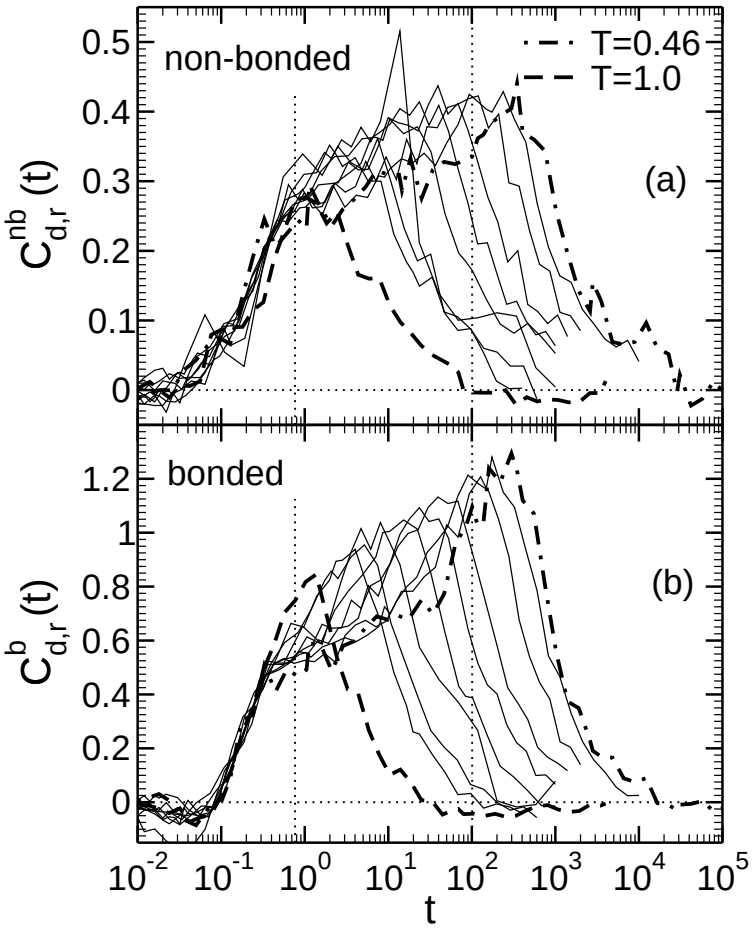

Figure 4.13: $C_{\mathrm{d,r}}(t)$ versus time for all T. $C_{\mathrm{d,r}}$ measures the tendency of a mobile particle to replace a mobile nearest neighbor at time t [Eq. (4.27)]. Panel (a) and panel (b) present the results for non-bonded and bonded nearest neighbors. In both panels, the vertical dotted lines indicate the times $t^*_{\alpha_2}$ for $T = 1$ ($t^*_{\alpha_2} = 0.766$) and for $T = 0.46$ ($t^*_{\alpha_2} = 100.894$). The following temperatures are shown (curves from left to right): $T = 1$, 0.7, 0.65, 0.6, 0.55, 0.52, 0.5, 0.48, 0.47, 0.46.

In order to study the influence of temperature on the time evolution of this anisotropic motion a direct comparison of the probabilities is not advisable. This resulting graphs are fairly crowded and often hide the main trends. It is better to look for an averaged quantity which is sensitive to preferential motion in direction parallel to $\mathbf{r}_{ij}$. A possible choice is

$$C_{\mathrm{d,r}}(t) \stackrel{\mathrm{def}}{=} \frac{1}{\pi}\int_0^\pi \left|\cos\left\{\theta\left[\mathbf{d}_i(t),\mathbf{r}_{ij}\right]\right\}\right| \frac{P_{\mathrm{d,r}}(\theta\left[\mathbf{d}_i(t),\mathbf{r}_{ij}\right])}{P_{\mathrm{iso}}(\theta\left[\mathbf{d}_i(t),\mathbf{r}_{ij}\right])}\,\mathrm{d}\theta\left[\mathbf{d}_i(t),\mathbf{r}_{ij}\right] - \frac{2}{\pi}\,. \qquad (4.27)$$

We use the absolute value of the cosine because the probability distribution is symmetric about 90°. Since $C_{\mathrm{d,r}}(t)$ should vanish for isotropic motion, we subtracted $(1/\pi)\int_0^\pi |\cos\theta|\mathrm{d}\theta = 2/\pi$. Note that Eq. (4.27) is not an expectation value because the ratio $P_{\mathrm{d,r}}/P_{\mathrm{iso}}$ is not a probability distribution. If $\mathbf{d}_i$ and $\mathbf{r}_{ij}$ are perfectly parallel to each other, $C_{\mathrm{d,r}}$ diverges.

Figure 4.13 depicts the time- and temperature dependence of $C_{\mathrm{d,r}}$ for both bonded and non-bonded nearest neighbors. Several observation can be made: First, for all

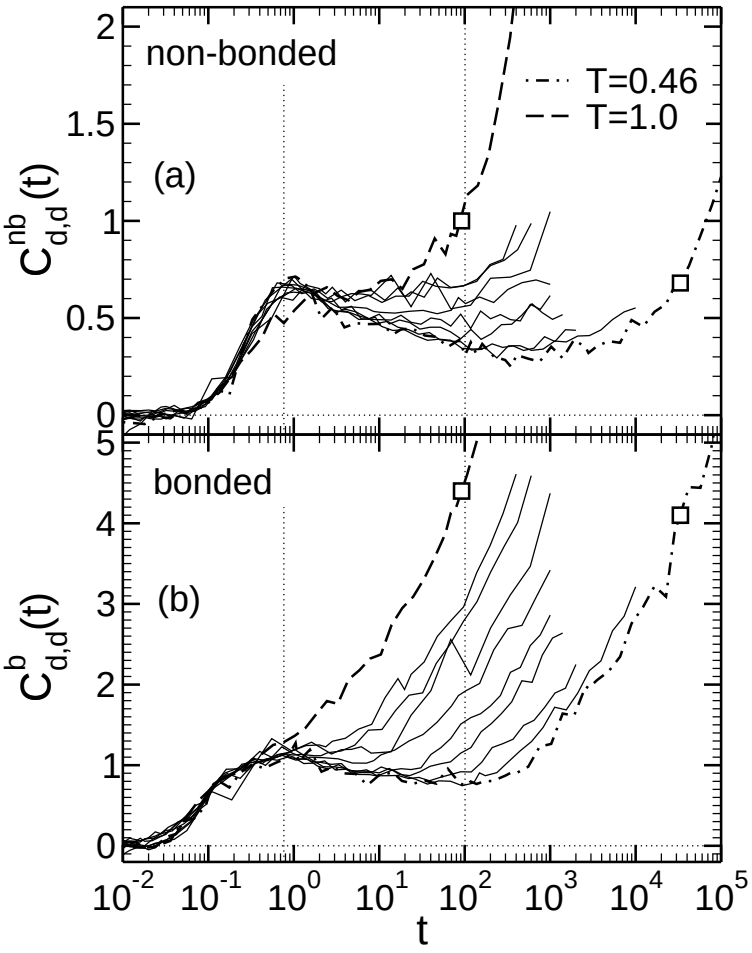

Figure 4.14: $C_{\rm d,d}(t)$ versus time for all T. $C_{\rm d,d}$ measures the tendency of two neighboring mobile particles to follow each other at time t [Eq. (4.28)]. Panel (a) and panel (b) present the results for non-bonded and bonded nearest neighbors. In both panels, the vertical dotted lines indicate the times $t^*_{\alpha_2}$ for $T = 1$ ($t^*_{\alpha_2} = 0.766$) and for $T = 0.46$ ($t^*_{\alpha_2} = 100.894$), whereas the open squares mark the times when g_4/g_0 reaches its second maximum ($t \simeq 92$ for $T = 1$, $t \simeq 33361$ for $T = 0.46$; see Fig. 4.5). The following temperatures are shown (curves from left to right): $T = 1$, 0.7, 0.65, 0.6, 0.55, 0.52, 0.5, 0.48, 0.47, 0.46.

temperatures the displacement of the mobile monomers is isotropic in the ballistic regime and for times in the α-process provided that $d_i \gg 1.5$ (for $T = 1$ and 0.46 this implies $t > 10$ and $t > 10^4$; see Fig. 4.3). Second, at intermediate times a mobile monomer tends to replace one of its inital mobile neighbors. This tendency is present at all T, but increases on cooling toward $T_{\rm c}$. It is maximum in the late-β/early-α process (at a time close to, but larger than $t^*_{\alpha_2}$). Third, the propensity of displacements along the nearest-neighbor axis is more pronounced if the neighbor is directly bonded to the mobile monomer.

For $\theta\left[\mathbf{d}_i(t), \mathbf{d}_j(t)\right]$ we define a similar quantity to measure the tendency of mobile neighbors to follow each other at some time t,

$$C_{\rm d,d}(t) \stackrel{\rm def}{=} \frac{1}{\pi} \int_0^\pi \cos\left\{\theta\left[\mathbf{d}_i(t), \mathbf{d}_j(t)\right]\right\} \frac{P_{\rm d,d}(\theta\left[\mathbf{d}_i(t), \mathbf{d}_j(t)\right])}{P_{\rm iso}(\theta\left[\mathbf{d}_i(t), \mathbf{d}_j(t)\right])}\, {\rm d}\theta\left[\mathbf{d}_i(t), \mathbf{d}_j(t)\right] \ , \quad (4.28)$$

which gives zero for uncorrelated motion and diverges to $+\infty$ if the mobile neighbors follow each other in perfect alignment (or to $-\infty$ for antiparallel motion).

Figure 4.14 depicts $C_{\mathrm{d,d}}(t)$ for all temperatures. Beyond the ballistic regime $C_{\mathrm{d,d}}(t)$ is positive for both bonded and non-bonded mobile monomers. The positive value implies that small angles between the displacements $\mathbf{d}_i(t)$ and $\mathbf{d}_j(t)$ are more likely than expected for isotropic motion. An analysis of $P_{\mathrm{d,d}}(\theta)/P_{\mathrm{iso}}(\theta)])$ in the time interval $1 \lesssim t \leq 10^4$ shows that $P_{\mathrm{d,d}}(\theta)]) > P_{\mathrm{iso}}(\theta)$ for $\theta \lesssim 60°$, whereas $P_{\mathrm{d,d}}(\theta[\mathbf{d}_i(t), \mathbf{d}_j(t)]) \to 0$, as $\theta \to 180°$. Thus, mobile monomers have an enhanced tendency to follow each other for all times outside the ballistic regime. This tendency is particularly pronounced at $t \approx 1$, which corresponds to the early β-relaxation at low temperatures, and for $t \gg t^*_{\alpha 2}$, where the monomer displacements are determined first by chain connectivity and later by the diffusion of the center of mass. In the diffusive regime one expects $C_{\mathrm{d,d}}(t)$ to be large. During the large time lapse t many monomers have moved in similar directions so that the center of mass advances substantially. Much before the diffusive regime, however, $C_{\mathrm{d,d}}(t)$ first "grows" and then "shrinks" as the observation time increases. A clue to interpret this behavior is obtained by calculating the ratio of the bonded (b) and the non-bonded (nb) neighbors. We find that $C^{\mathrm{b}}_{\mathrm{d,d}}(t)/C^{\mathrm{nb}}_{\mathrm{d,d}}(t) \sim g_4(t)/g_0(t)$ (see Fig. 4.5 for comparison). The difference between $C^{\mathrm{b}}_{\mathrm{d,d}}(t)$ and $C^{\mathrm{nb}}_{\mathrm{d,d}}(t)$ is large for $t \approx 0.1$ and for $t^*_{\alpha 2} \lesssim t \lesssim \tau_{\mathrm{R}}$, where the mean-square displacement of the ends is enhanced compared to the average. It is plausible that a highly mobile end monomer will trigger large displacements of the neighbor connected to it and bias this displacement in direction of its own motion. This effect is strongly suppressed in the β-regime, where $g_4(t)$ and $g_0(t)$ are alike. For $1 \lesssim t \lesssim t^*_{\alpha 2}$, $C^{\mathrm{b}}_{\mathrm{d,d}}(t)/C^{\mathrm{nb}}_{\mathrm{d,d}}(t) \approx$ constant, comparable to g_4/g_0.

4.5 Concepts, Summary, and Conclusions

The presented analysis of stringlike motion in the dynamics of a simulated polymer melt is based on the following prerequisites:

- We analyze the dynamics of a non-entangled melt in the supercooled state close to, but above the critical temperature of mode-coupling theory T_{c}.
- Our analysis utilizes the 6.5% of the monomers which undergo the largest displacements in some time t. These monomers are called "mobile". The fraction of 6.5% was derived in Ref. [26] following the procedure proposed in Refs. [100–102]: The non-GAUSSian parameter of the monomers has a pronounced maximum at a time $t^*_{\alpha 2}$ in the window of the late-β/early-α process (Fig. 4.2). This reflects that large displacements occur more frequently than for a random walk motion.
- The fraction of 6.5% is fixed for all times and temperatures. Thus, the number of particles in the set of mobile monomers is always the same, but the composition of the set will generally vary with t: Monomers that moved far in some time interval may be blocked at a later time, and *vice versa*.
- If a mobile monomer i approaches at time t the initial position of another mobile monomer j to within a radius δ, we say that i and j form a "string". δ is chosen such that the one-dimensionality of the replacement is guaranteed.

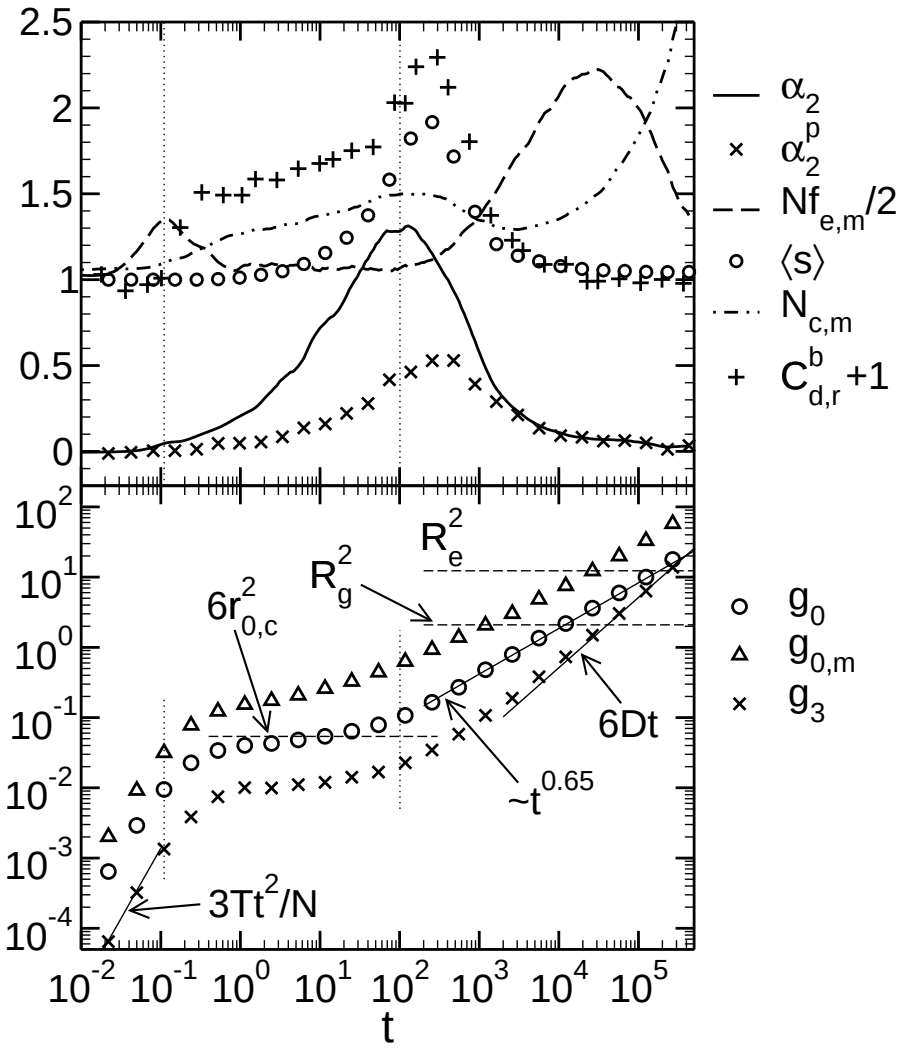

Figure 4.15: Recapitulating comparison of the time dependence of various quantities discussed before. In all cases, the temperature is $T = 0.46$ ($T_c \simeq 0.45$). The upper figure shows the non-GAUSSian parameters α_2 and α_2^p. These quantities are averages over all monomers and all chains in the melt. The other quantities shown are calculated from the 6.5% of highly mobile monomers. They are: the fraction of mobile end-monomers $Nf_{e,m}/2$, the average stringlength $\langle s \rangle$, the average length of mobile contiguous segments of a chain $N_{c,m}$, and the cosine of the angle between the displacement of a mobile monomer and the vector to one of its nearest mobile neighbors in the chain $C_{d,r}^b$. The vertical dotted lines indicate the time of the first peak of $Nf_{e,m}/2$ ($t = 0.1104$) and the time $t^*_{\alpha_2}$ ($= 100.894$) where α_2 is maximum. These times are also included in the lower figure (vertical dotted lines). This figure shows the MSD of all monomers g_0, of the mobile monomers $g_{0,m}$, and of the center of mass g_3. The behavior in the ballistic ($\sim t^2$), the subdiffusive ($\sim t^{0.65}$) and the diffusive regimes ($\sim t$) are indicated by solid lines. The horizontal dashed lines depict $6r_{0,c}^2$ (LINDEMANN localization length $r_{0,c} \simeq 0.095$ for g_0), the radius of gyration R_g^2 ($= 2.09$), and the end-to-end distance R_e^2 ($= 12.3$).

Using this method we can select a subensemble of highly mobile monomers at any time, ranging from the ballistic to the diffusive regime. For this subensemble we address the following questions: (1) Do these monomers follow each other in a stringlike fashion, as observed for a binary LJ-mixture [101, 102]? (2) If yes, what are the properties of these dynamic strings? (3) What is the influence of chain connectivity on this correlated motion?

The answers to these questions may be summarized as follows (see Fig. 4.15):

- There is an intermediate time interval in which string formation can be observed. In the cold melt this interval corresponds to the late-β/early-α process.

- The average stringlength $\langle s(t) \rangle$ is maximum at a time $t_{\rm str}^{\rm max}$. Close to $T_{\rm c}$, the temperature dependence of $t_{\rm str}^{\rm max}$ is approximately the same as that of the inverse diffusion coefficient $1/D$ (the product $Dt_{\rm str}^{\rm max} \approx$ const). This finding is not unexpected, as we select the monomers with the largest displacements. These monomers are most likely to determine the diffusion behavior.

- The maximum stringlength $\langle s(t_{\rm str}^{\rm max}) \rangle$ increases with decreasing T, approximately in an exponential fashion (Fig. 4.9). The distribution of stringlengths s, yielding this average, is close to an exponential (Fig. 4.10). These findings suggest an analogy between strings and equilibrium polymers, as proposed in Ref. [101].

- The average stringlength remains fairly small on cooling. Even at $T = 0.46$, $\langle s(t_{\rm str}^{\rm max}) \rangle \approx 2$. As the MSD of the mobile monomers is $g_{0,{\rm m}}(t_{\rm str}^{\rm max}) \approx 1$, this implies that a monomer substitutes one of its mobile neighbors.

- An important contribution to this process comes from chain connectivity. A monomer tends to replace one of its bonded neighbors (see $C_{\rm d,r}^{\rm b}$ in Fig. 4.15 and Fig. 4.8). However, mobility is not concentrated along the backbone of some chains. If this was the case, the average stringlength calculated for the monomers of a chain only $\langle s_{\rm seg} \rangle$ (Fig. 4.8) and the average length of mobile contiguous segments $N_{\rm c,m}$ should be of the order of N. Therefore, a relaxation mechanism, in which mobile monomers are connected to each other and slide along the backbone of the chain, seems to be unlikely.

Chapter 5

Kinetic Friction and Atomistic Instabilities in Boundary-Lubricated Systems

5.1 Introduction

The every-day phenomenon friction is of great practical and economical importance, which is one of the motivations to improve our understanding of tribological processes [27, 28]. Originally, the intention for this part of the study was to analyze the rheological and tribological behavior of thin polymeric films (two dimensional layers of varying density) that are confined between two solid surfaces sliding against each other by means of molecular dynamics simulations. The scientific challenge to be solved is to understand in detail the lubricating action of such polymeric films (which is well-known from everyday life: after all, short alkane chains are the main constituent in the motor oil we put into our car engines), and why polymers are a much better lubricant than small molecules.

However, it turned out that for this purpose still basic questions on the behavior of simple fluids (imitating small, spherical molecules) as boundary lubricants in the context of such simulations needed to be clarified as well and so actually the bulk of the following chapter necessarily addresses the simulation of kinetic friction and the related atomistic instabilities in such ultrathin fluid layers between sliding solids in general, and only the very last part contains a feasibility study including short polymer chains.

Friction between two solids differs from that between a solid and a fluid in that both static and kinetic friction appear finite, while the force between a solid and a fluid vanishes linearly with sliding velocity v_0 at small v_0. Static friction F_s is the externally applied force necessary to initiate relative sliding motion between two solids, whereas kinetic friction F_k is the force needed to maintain the sliding motion. Phenomenological friction laws, which date back to da Vinci, Amontons, and Coulomb [123], often provide a good description on the macroscopic scale. Amontons' laws state that F_k is proportional to the normal load and independent of the (apparent) area of contact. According to Coulomb's law kinetic friction is independent of sliding speed v_0.

The microscopic origins of kinetic friction are still a matter of debate, even

though it has long been recognized that kinetic friction must be due to dynamical instabilities [124, 125]. While there can be many different processes leading to instabilities, they all have in common that potential energy is converted abruptly into kinetic energy and ultimately lost as heat [41]. Although instabilities can occur on many different time and length scales, there has been an enhanced interest in identifying those that occur on atomic scales. This quest is not only motivated by the miniaturization of technical devices down to the nanometer scale, but also by the desire to better understand macroscopic friction. The understanding of single-asperity contacts is needed as basis for the full description of macroscopic friction, where the bulk-mediated coupling between contacts gives rise to additional effects.

Load-bearing, simple-asperity contacts are often in the order of microns. According to HERTZian contact mechanics [126] and generalizations thereof [127–132], the pressure is rather constant in the contact with the exception of the areas close to the circumference, where pressure gradients are large. In the center of the contact, most of the lubricant is squeezed out. One may assume that these boundary-lubricated areas often account for most of the energy dissipation when two solids are slid against each other, unless the solids are very compliant, in which case elastic instabilities may also contribute a significant amount to the net dissipation. If wear was the main source of friction, material would have to rub off from the surfaces much faster than observed experimentally [133]. Hydrodynamic lubrication would likewise result in values for friction orders of magnitude too small, if it were assumed to be the dominant dissipation process.

Two different avenues have been pursued in the recent past to study dynamics in boundary lubricants and its consequences for tribological properties. One is a minimalist approach, in which one single lubricant atom embedded between two shearing plates is considered [134, 135]. In the following, we will refer to this approach as the impurity limit. The other avenue incorporates a large ensemble of lubricant atoms [136, 137]. This approach can eventually include surface curvature and elastic deformation of the surfaces making it possible to study what effect the interplay of surface curvature and elastic deformations have on dry or boundary-lubricated friction [138, 139].

Since kinetic friction is intimately connected with instabilities, we focus on the analysis of instabilities. The central assumption of our analysis is the existence of instabilities or "pops" of certain degrees of freedom. A pop is a sudden, seemingly erratic motion of a particle (or a collective degree of freedom) characterized by a velocity much larger than the associated thermal velocity or the drift velocity of the atom. Pops heat the lubricant or alternatively they couple directly to the confining solid walls, e.g. by inducing phonons in the walls. They will eventually induce more dramatic effects such as generation of dislocations or abrade the surfaces. However, as argued above, these extreme processes are rare and hence presumably they are not responsible for the main part of the energy dissipation. This is the motivation to concentrate on the energy transfer to the phonon bath that is due to elementary processes in the lubricant. The underlying idea of the presented approach can be described as follows. Sliding-induced instabilities make the velocity probability distribution (PD) of the lubricant atoms deviate from the thermal equilibrium PD. This alters the balance of energy flow from and to the lubricant. The energy missing

in this balance is provided by the kinetic friction force shearing the plates.

We intend to analyze what features of simplistic models appear robust as the level of complexity in the description of the boundary lubricant is increased. Starting from the impurity limit we will include lubricant particle interaction and increase coverage for spherical fluids. Finally, polymers will be studied.

After discussing the PRANDTL-TOMLINSON model as a paradigmatic system for the occurrence of instabilities in section 5.2 we will connect the probability distribution of the lubricant atoms with the friction force in section 5.3. Then, our findings on different model systems with increasing complexity will be presented in sections 5.4 to 5.6. Section 5.7 contains our conclusions.

5.2 The Prandtl-Tomlinson Model

Most earlier work on instabilities and and its connection to friction is devoted to elastic processes, which are most simply described in the PRANDTL-TOMLINSON (PT) model [140, 141]. We will discuss it here to introduce the notions of instabilities and "pops" and how they lead to kinetic friction.

In this one-dimensional model, a wall atom of mass m is coupled harmonically with spring stiffness k_T to a slider. The slider is moved with a constant sliding velocity v_0 with respect to a completely rigid substrate. The slider's atom i being at position $x_i(t)$ at time t feels a potential V_b which is periodic in the substrate's lattice constant $b/2\pi$. For simplicity, a harmonic potential $V_\mathrm{b}(x) = f_0 \cos(x/b)$ is assumed. Finally, the atom i feels a viscous dissipative force $m\gamma v_i$ proportional to its velocity v_i. This damping mimics the coupling to a heat bath, which is not explicitly treated.

Thereby we arrive at the equation of motion for the individual slider atoms,

$$m\partial_t v_i(t) + m\gamma v_i(t) = -k_\mathrm{T}\left[x_i(t) - x_i^0(t)\right] + \frac{f_0}{b}\sin\left[x_i(t)/b\right] \ , \tag{5.1}$$

where we defined the atom's equilibrium position $x_i^0(t) = v_0 t$.

The system will show a different behavior for different values of the spring stiffness k_T due to the interplay of the elastic and the harmonic potential. Let

$$V_\mathrm{w}(x_i) = f_0 \cos(x_i/b) + \frac{k_\mathrm{T}}{2}(x_i - x_i^0)^2 \tag{5.2}$$

be the combined conservative potential. The second derivative is then

$$\partial_{x_i}^2 V(x_i) = k_\mathrm{T} - \frac{f_0}{b}\cos(x_i/b) \ , \tag{5.3}$$

which is greater than zero for all positions x_i if $k_\mathrm{T} > f_0/b$. Thus, there is only one well-defined stable equilibrium position for particle i in this case. At small sliding velocities v_0 where dissipation can be neglected, every atom will be close to its unique equilibrium position, moving at constant velocity v_0. The drag force is then of the order $m\gamma v_0$ and hence the time averaged friction force F_k tends linearly with v_0 to zero, $F_\mathrm{k} = \mathcal{O}(v_0)$.

This behavior changes qualitatively if $k_\mathrm{T} < f_0/b$, where there is more than one stable equilibrium position for each wall atom ("multistability"). The atom stays at

one side of the potential hill until the spring is sufficiently stretched by the external force to pull the particle over the barrier. The elastic energy contained in the spring will then be released rapidly, and the particle "pops" into the next stable position. The trajectory shows an "instability". This process will lead to particle velocities v_i much greater than v_0 which are only determined by the potential landscape and not by v_0. At sufficiently small sliding velocities there will therefore be a well defined limit of the frictional force $F_{\mathrm{k}}(v_0 \to 0^+)$, stemming from the energy dissipation of order $\mathcal{O}(m\gamma\langle v_i\rangle)$. This shows that the time evolution of mechanically stable positions is crucial for the understanding of sliding friction at low velocities, because friction arises from instabilities. In the PT-model they arise from the interplay of the elastic coupling of the slider atom and the energy landscape of the surface.

These instabilities result in non-vanishing F_{k} in the limit of zero v_0 as long as thermal fluctuations are absent, see e.g. the discussion in Ref. [41]. There is, however, a crucial difference between instabilities in boundary lubricants and instabilities occurring in elastic manifolds that are modeled in terms of the PT model and related approaches such as the FRENKEL-KONTOROVA model [142–144]. In boundary lubricants, atoms are only weakly connected to each other and to the confining walls. As a consequence, bond breaking can occur, whereas in elastic models, bonds are treated as unbreakable. This seemingly subtle difference leads to different tribological behavior.

5.3 Relation Between Velocity Distribution and Friction

The most fundamental assumption for the connection of particle motion and friction force is that the interaction between the lubricant atom i and the confining wall can be decomposed into one conservative part $V_{\mathrm{w}}(\mathbf{x}_i)$ and one non-conservative term consisting of a damping force plus thermal noise, like we use in our simulation (cf. section 2.5). $V_{\mathrm{w}}(\mathbf{x}_i)$ depends only on the difference between the position $\mathbf{x}_i$ and the positions of top wall $\mathbf{r}_{\mathrm{t}}$ and bottom wall $\mathbf{r}_{\mathrm{b}}$. It can be written as

$$V_{\mathrm{w}}(\mathbf{r}_i) = V_{\mathrm{b}}(\mathbf{x}_i - \mathbf{r}_{\mathrm{b}}) + V_{\mathrm{t}}(\mathbf{x}_i - \mathbf{r}_{\mathrm{t}}) \; . \tag{5.4}$$

Unless noted otherwise, the relative motion of the walls is imposed externally by constant separation (or constant load) and constant relative velocity $v_0\hat{\mathbf{e}}_x = (\dot{\mathbf{r}}_{\mathrm{t}} - \dot{\mathbf{r}}_{\mathrm{b}})$ of the walls parallel to the sliding direction indicated by the unit vector $\hat{\mathbf{e}}_x$. We assume the normal pressure variations to be small, which means that the coupling to each individual confining (crystalline) wall is periodic parallel to the interface, i.e. it is periodic in the xy-plane.

Consider a system in steady-state equilibrium with the following underlying equation of motion:

$$m\ddot{\mathbf{x}} + m\gamma\dot{\mathbf{x}} = \mathbf{F}_{\mathrm{b}}(\mathbf{x}) + \mathbf{F}_{\mathrm{t}}(\mathbf{x} - \mathbf{v}_0 t) + \mathbf{\Gamma}(t). \tag{5.5}$$

Here, $\mathbf{F}_{\mathrm{b}}(\mathbf{x})$ denotes the force of the bottom wall (likewise with subscript "t" for the top wall) on an impurity atom located at position $\mathbf{x}(\mathbf{t})$ and $\mathbf{v}_0$ is the velocity of the top wall with respect to the bottom wall. We multiply Eq. (5.5) with $\dot{\mathbf{x}}(t)$ and average over a long time interval τ. We then interpret the resulting individual terms.

They can be associated with the (average) power dissipated within the system or the (average) power put into the system.

First, the average change of kinetic energy with time equals zero, namely

$$\frac{1}{\tau}\int_0^\tau \mathrm{d}t\, m\ddot{\mathbf{x}}\dot{\mathbf{x}} = \frac{1}{\tau}\int_0^\tau \mathrm{d}t\, \frac{d}{dt}T_{\mathrm{kin}} = \frac{1}{\tau}[T_{\mathrm{kin}}(t=\tau) - T_{\mathrm{kin}}(t=0)] \underset{\tau\to\infty}{=} 0\,. \tag{5.6}$$

The second term is proportional to the time-averaged kinetic energy of the system with respect to the reference frame:

$$\frac{1}{\tau}\int_0^\tau \mathrm{d}t\, m\gamma\dot{\mathbf{x}}\dot{\mathbf{x}} = \gamma m\langle\dot{\mathbf{x}}^2\rangle = 2\gamma\langle T_{\mathrm{kin}}\rangle, \tag{5.7}$$

$\langle T_{\mathrm{kin}}\rangle$ being the time-averaged (steady-state) kinetic energy of an impurity. The next term is the average work of the bottom wall on the impurity

$$\frac{1}{\tau}\int_0^\tau \mathrm{d}t\, \dot{\mathbf{x}}\mathbf{F}_{\mathrm{b}}(\mathbf{x}) = \frac{1}{\tau}\int_0^\tau \mathrm{d}t\, \left(-\frac{d}{dt}V_{\mathrm{b}}(\mathbf{x})\right) = \frac{1}{\tau}\left\{V_{\mathrm{b}}[\mathbf{x}(\tau)] - V_{\mathrm{b}}[\mathbf{x}(0)]\right\} \underset{\tau\to\infty}{=} 0\,. \tag{5.8}$$

In any steady-state of the system the average force on the bottom wall (or the top wall) must be zero. If the time average was different from zero, the wall would be accelerated in contradiction to the steady-state assumption, as pointed out for instance, by Thompson and Robbins [137]. Of course, if the model was generalized such that (steady state) wear would occur, then the contribution discussed in Eq. (5.8) would indeed remain finite.

For the discussion of the next term in Eq. (5.5), it is necessary to keep in mind that

$$\frac{d}{dt}V_{\mathrm{t}}(\mathbf{x}-\mathbf{v}_0t) = -F_{\mathrm{t}}(\mathbf{x}-\mathbf{v}_0t)\,(\dot{\mathbf{x}}-\mathbf{v}_0)\ . \tag{5.9}$$

This and the same considerations invoked for Eq. (5.8) yield

$$\frac{1}{\tau}\int_0^\tau \mathrm{d}t\, \dot{\mathbf{x}}\mathbf{F}_{\mathrm{t}}(\mathbf{x}-\mathbf{v}_0t) = \langle\mathbf{F}_{\mathrm{t}}\rangle\mathbf{v}_0\ , \tag{5.10}$$

where $\langle\mathbf{F}_{\mathrm{t}}\rangle$ is the time-averaged force that the top wall exerts on an impurity. This force or depending on the definition its projection onto the sliding direction can be associated with the kinetic friction force F_{k}. One sees immediately that we would obtain the same result if we changed the reference frame where top and bottom wall would move by $\mathbf{v}_0/2$ in opposite directions and $\langle\mathbf{F}_{\mathrm{t}}\rangle = \langle\mathbf{F}_{\mathrm{b}}\rangle$.

The contribution due to the random force $\mathbf{\Gamma}(t)$ is the most difficult contribution to calculate. However, if the system is close to local equilibrium for most of the time, then the expectation value of $\mathbf{\Gamma}(t)\dot{\mathbf{x}}$ can be expected to be close to the value of this expression in thermal equilibrium. In equilibrium, it must compensate the expression discussed in Eq. (5.7), hence

$$\frac{1}{\tau}\int_0^\tau \mathrm{d}t\, \dot{\mathbf{x}}\mathbf{\Gamma}(t) \approx 2\gamma\langle T_{\mathrm{kin}}\rangle_{\mathrm{eq}}\ , \tag{5.11}$$

where $\langle T_{\mathrm{kin}}\rangle_{\mathrm{eq}}$ denotes the average kinetic energy in thermal equilibrium.

Assembling all above terms yields

$$P_{\mathrm{ext}} = \langle\mathbf{F}_{\mathrm{t}}\rangle\mathbf{v}_0 = 2\gamma[\langle T_{\mathrm{kin}}\rangle - \langle T_{\mathrm{kin}}\rangle_{\mathrm{eq}}]\ , \tag{5.12}$$

as the power provided by the *external* forces – the *internal* effect of the random forces was assumed to be the same in and out of thermal equilibrium in Eq. (5.11).

We can write the time averages for the kinetic energy equivalently as integrals over the velocity probability distributions in steady-state and in (thermal) equilibrium, for which the (MAXWELL-BOLTZMANN) distribution $P_{\mathrm{eq}}(v)$ applies by using Eqs. (5.12) and (5.7),

$$P_{\mathrm{ext}} = N_{\mathrm{fl}}\gamma m \left(\langle \dot{\mathbf{x}}^2 \rangle - \langle \dot{\mathbf{x}}^2 \rangle_{\mathrm{eq}}\right) = N_{\mathrm{fl}}\gamma m \int_0^\infty v^2 \left[P(v) - P_{\mathrm{eq}}(v)\right] \mathrm{d}v \ , \tag{5.13}$$

with P_{ext} being the total external power dissipated by the system and $v = |\dot{\mathbf{x}}|$ denoting the velocity of a particle in the reference frame of the thermostat. F_{k} can now be associated with

$$F_{\mathrm{k}} = P_{\mathrm{ext}}/v_0. \tag{5.14}$$

Typically, the time scales associated with the excitations leading to energy dissipation are short compared to the motion of a particle from one minimum to another, which justifies the assumption of δ correlated random forces for our purposes in the LANGEVIN thermostat. Of course, damping can and will be different normal and parallel to the interface. However, this detail does not have any significant consequences for the conclusions presented in this chapter. Similarly, the explicit treatment of internal elastic deformations does not alter the major conclusions either which we checked by simulations including elastic interactions in the walls.

We want to emphasize that Eqs. (5.13) and (5.14) allow one to calculate friction forces under more general conditions than those of our particular model, for instance, if the thermostat only acts on the atoms in the outermost layers of the walls as e.g. employed in Ref. [145]. The approach can also be extended in a straightforward manner if generalized forms of the thermostat are employed such as in dissipative particle dynamics [146, 147] or if the thermostat is based on a MORI ZWANZIG formalism [148, 149]. The main limitation of Eq. (5.13) in the present context is that effects due to heating of the walls are not included. Again, a minor modification would allow one to include heating of the walls into the presented framework as well. However, as we will mainly focus on small velocities, the mentioned effects will be small and shall be neglected in the following. In Fig. 5.10 we will assess the validity of Eq. (5.13) in our simulation more closely.

Note that an alternative way of determining the friction force in the steady state is to time average the conservative plus the non-conservative force that the upper wall exerts on the lubricant which can directly be recorded in a simulation. The observation that the work done by the conservative force is essentially zero does not imply that this time-average must be zero.

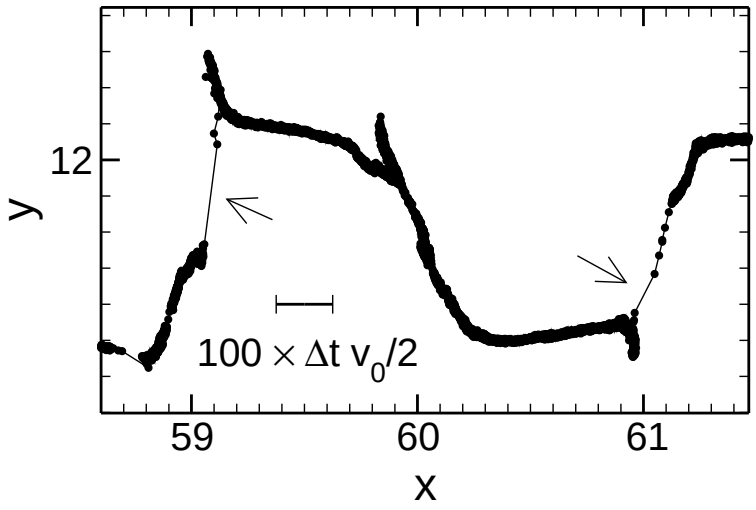

Figure 5.1: Trajectory of a lubricant impurity in the xy-plane tagged between two incommensurate surfaces (at large load and small temperature). The relative velocities of the walls is $v_0 = 10^{-3}$. The positions are plotted every $\Delta t = 0.5$. The bar denotes 100 times the average drift distance per time interval Δt. The arrows indicate dynamical instabilities.

5.3.1 Effect of Instabilities

As discussed in the introduction and for the PT-model, the externally imposed relative motion of the confining walls may induce sudden, dynamic instabilities or "pops" during which the particles' velocities greatly exceed both their thermal velocities and the relative sliding velocity v_0 of the walls. This means that at a time $t+\delta t$ the atom does not find a stable position in the $\mathcal{O}(v_0\delta t)$ vicinity of the old stable position at time t. The continuous trajectory ends at t and the particle has to move to the next mechanically stable position to resume its path. The particle will then pop into the next local potential minimum and for low sliding velocities, its peak velocity v_{peak} will be solely determined by the energy landscape and consequently $\lim_{v_0\to 0^+} v_{\text{peak}}/v_0$ diverges. This process will lead to a deviation of the velocity distribution $P(v)$ from the thermal equilibrium distribution $P_{\text{eq}}(v)$ valid for $v_0 = 0$. Fig. 5.1 shows such instabilities for a model system that is described in detail in section 5.4.3.1.

The velocity distribution $P(v)$ and hence the friction force F_{k} can be calculated in principle, once the precise form of the lubricant's interaction is known. Risken's book on the Fokker-Planck equation [150] gives an excellent overview of methods that allow one to treat models like ours, namely externally-driven systems that are mainly deterministic but also contain a certain degree of thermal noise. An analytical approach remains difficult in our case, due to the potential's complex time dependence. Therefore a different, phenomenological approach will be pursued.

An instability will invoke a trajectory during which potential energy is abruptly converted into kinetic energy. The kinetic energy will then be dissipated into the thermostat, i.e., the phonon bath of the confining walls. After some time, which depends on the coupling strength to the thermostat, the Maxwell-Boltzmann probability distribution (PD) will be resumed, provided no new instability has been invoked in the meantime. An instability will thus create a typical velocity PD that

will show up as a tail in the MAXWELL-BOLTZMANN PD. Unless the two confining walls are identical and perfectly aligned (thus commensurate), there is a class of instabilities in which the energy lost during the "pop" shows a broad distribution, see also Ref. [151]. Every pop, characterized for instance by the energy dissipated, will contribute to $P(v)$ in its own way. We assume that the net sum P_{tail} of all these individual tails shows exponential dependence on velocity, thus

$$P_{\text{tail}}(\mathbf{v}) \propto \exp\left[-B\left|\mathbf{v} - \langle\mathbf{v}\rangle\right|\right] , \tag{5.15}$$

where $\langle\mathbf{v}\rangle$ is the average drift velocity of the impurities under consideration, typically $\langle\mathbf{v}\rangle = \mathbf{v}_0/2$, and B is a constant. We can get rid of the drift term by changing the reference frame. The motivation for this particular choice of P_{tail} partly stems from JAYNES' principle of information theory [152]. It states that the most likely normalized PD with given mean about which we do not have more knowledge is the exponential distribution [153, p. 27]. More importantly, this choice of P_{tail} happens to be a quite accurate description for the velocity PDs of impurities between 2-d, incommensurate surfaces. This will be demonstrated later in the result section.

At small sliding velocity v_0, the statistical weight of the tails must increase linearly with velocity. Instead of using the full 2-d PD for the pops in the xy-plane, $P(v_x, v_y)$, we can use rotational symmetry and work with

$$P(v_\parallel) = 2\pi v_\parallel A v_0 e^{-Bv_\parallel} + \left(1 - \frac{2\pi A v_0}{B^2}\right) P_{\text{eq}}(v_\parallel), \tag{5.16}$$

instead. Here A, and B are phenomenological parameters that can (and will) depend on the externally applied load L that an impurity has to counterbalance, damping γ, and other parameters. However, they should depend only weakly on temperature T and sliding velocity v_0 at small T and small values of v_0. This is because Av_0 is a measure for the rate of the fast processes (which should be proportional to v_0 at small v_0), while B characterizes the instability related velocity PD determined by the potential landscape. The prefactor of $P_{\text{eq}}(v_\parallel)$ is unity minus the integrated probability of the tail.

The assumption of isotropy might seem counterintuitive as it implies that the PD in sliding direction and normal to it to be the same. However, from Fig. 5.7 we will learn that it is justified to use isotropy. To this end it is in order to comment on the form of the projection of $P(v_x, v_y)$ onto a coordinate axis. The projection can be computed from Eq. (5.15) using polar coordinates where we set $B = 1$ for simplicity,

$$\begin{aligned} P(v_x) &= \int P(v_x, v_y)\mathrm{d}v_y \\ &= \int \exp\left[-\sqrt{v_x^2 + v_y^2}\right] \mathrm{d}v_y = 4\int_0^{\pi/2} \exp\left[-v_x\sqrt{1 + \tan^2(\phi)}\right] \mathrm{d}\phi \\ &= 4\int_0^{\pi/2} \exp[-v_x \sec(\phi)]\mathrm{d}\phi \\ &= \exp[-C(v_x)v_x]\,,\ 1 < C(v_x) < 1.1\,,\ \forall v_x \gtrsim 1\,. \end{aligned} \tag{5.17}$$

The projection of the exponential distribution in the plane on an axis will therefore also be approximately exponential (with a different exponent prefactor) for sufficiently high v_x, the exact analytic result is however quite complicated.

Provided that the pops happen preferably in the xy-plane (such that the normal component $P(v_\perp)$ is small), inserting Eq. (5.16) into Eqs. (5.13) and (5.14) and integrating over v yields the following friction force per impurity atom $F_\mathrm{k}/N_\mathrm{fl}$:

$$\frac{1}{N_\mathrm{fl}}F_\mathrm{k} = 12\pi\gamma m\frac{A}{B^4} - 4\pi\gamma\frac{A}{B^2}k_\mathrm{B}T \ . \tag{5.18}$$

The friction force should of course be independent of the choice for the thermostat damping constant γ. In our simulations, we have verified the proportionality $A \propto 1/\gamma$ upon varying γ over 2 decades ($0.05 \leq \gamma \leq 5$) while B stayed constant with high accuracy, hence assuring this independence.

Of course, Eq. (5.18) can only be valid as long as Eqs. (5.15) and (5.16) give an accurate description of the non-equilibrium velocity PD and provided that the heat flow from the thermostat into the impurities is close to the thermal equilibrium heat flow. At extremely small v_0, two arguments show that the assumption of exponential tails cannot persist. First, the energy ΔE_diss that is dissipated during a pop has an upper bound, which in turn implies an upper bound for the peak velocity. Second, close to equilibrium, thermal noise is sufficient to invoke (multiple) barrier crossing and recrossing. The ratio of sliding and noise-induced instabilities becomes small, which in turn makes the non-equilibrium corrections be less significant.

5.4 Impurity Limit

5.4.1 1-d Model Systems

In a recent publication by Müser [154] a one-dimensional model system was investigated. Impurity-wall potentials were modeled by harmonic potentials with first higher order contribution and different periodicities for top "t" and bottom "b" substrate,

$$V_\mathrm{s} = V_{\mathrm{s},0}\cos(2\pi(x-x_\mathrm{s})/b_\mathrm{s}) + V_{\mathrm{s},1}\cos(4\pi(x-x_\mathrm{s})/b_\mathrm{s}) \ , \quad \mathrm{s} = \mathrm{t},\mathrm{b} \ . \tag{5.19}$$

As this model was a motivation for the present work and a simple model system for the analysis of instabilities we summarize the results of Ref. [154]. It was found that the existence of instabilities in the impurity limit and as a consequence the friction-velocity relationship $F_\mathrm{k}(v_0)$ depends on the "details" of the model. For instance, it was found that for 1-d, commensurate interfaces, the sign of the first higher harmonic in the lubricant-wall potential determines: (a) whether or not the athermal kinetic friction remains finite in the zero-velocity limit, and (b) the exponent β that describes the finite-velocity corrections by

$$F_\mathrm{k}(v) - F_\mathrm{k}(0) \ \propto \ v_0^\beta \ , \quad 0 < \beta < 1 \ . \tag{5.20}$$

Note that Eq. (5.20) changes its form when thermal noise is included into the treatment, i.e., it becomes linear at small velocities [154].

For (quasi-) incommensurate walls ($b_\mathrm{t} \neq b_\mathrm{b}$), the behavior is even richer. If the first higher harmonic is not included, one wall exerts a maximum force on the impurity and drags the impurity along. As a consequence, F_k is linear in v_0, which we call Stokes friction. For one certain value of the first higher harmonic $V^*_{t,1}$ (at a

fixed ratio $b_\mathrm{s} = b_\mathrm{b}/b_\mathrm{t}$), F_k can best be described as a power law in the limit of small v_0. For $V_{t,1} > V_{t,1}^*$, F_k remains finite in the limit $v \to 0$, again provided thermal fluctuations are absent.

5.4.2 2-d Model Systems

5.4.2.1 Model Details

We now allow the lubricant atoms to move within the xy-plane, but motion normal to the interface in z direction is still neglected. Eq. (5.19) must then be replaced with a new model potential. As in other studies, we consider the symmetry of the confining walls to be triangular, i.e. (111) surfaces of an fcc crystal, for which the potential V_s between a wall and an impurity can be written as:

$$V_\mathrm{s}(\mathbf{x}, z) = \sum_{\mathbf{g}} \tilde{V}_\mathrm{s}(\mathbf{g}, z) \exp\left[\mathrm{i}\mathbf{g}(\mathbf{x} - \mathbf{x}_\mathrm{s})\right]. \tag{5.21}$$

Here, $\mathbf{g}$ denotes the two-dimensional reciprocal lattice vectors of the triangular lattice, $\mathbf{x}$ is the position of the lubricant particle in the xy-plane and $\mathbf{x}_\mathrm{s}$ the in-plane position of the wall. z denotes the (fixed) distance between (top) wall and impurity. The FOURIER coefficients $\tilde{V}_\mathrm{s}(\mathbf{g}, z)$ between chemically non-bonding species often depend exponentially on $\mathbf{g}$ and z, thus $\tilde{V}_\mathrm{t}(\mathbf{g}, z)$ can be written as

$$\tilde{V}_\mathrm{t}(\mathbf{g}, z) = \tilde{V}_\mathrm{t}(\mathbf{g}, 0) \exp[-\alpha \left|\mathbf{g}\right| z], \tag{5.22}$$

where both $V_\mathrm{t}(\mathbf{g}, 0)$ and α depend on the chemical nature of impurity atom and confining wall. Potentials of this form are known as STEELE potentials [155]. They have proven to describe the potential energy landscape of atoms on crystalline surfaces reasonably well [156]. The fundamental harmonic in this potential is related to the smallest non-zero lattice vectors $\mathbf{g} = n_x\mathbf{b}_x + n_y\mathbf{b}_y$ with the lattice base vectors $\mathbf{b}_x = (4\pi/\sqrt{3}, 0)$, $\mathbf{b}_y = (2\pi/\sqrt{3}, 2\pi)$ and n_x, n_y being integers. This choice sets the wall lattice constants d_nn to unity. First higher harmonics are related to reciprocal lattice vectors that are the sum of a suitable pair of two different fundamental $\mathbf{g}$'s with length 4π and so on. The fundamental harmonic will be dominant at small loads. However, as the external pressure increases (which makes z decrease), the *relative* importance of higher harmonics will increase due to Eq. (5.22).

The top wall V_t was rotated relatively to the bottom wall V_b by an angle θ and shifted by a displacement $\Delta\mathbf{w}$ and the sliding direction was at an angle ϕ relative to the x-axis of the unrotated wall, defining a shear vector $\mathbf{v}_0$, with $\angle(\mathbf{v}_0, \hat{x}) = \phi$. Two walls are called commensurate if θ is an integer multiple of 60° (the potentials are in this case identical). A random choice of the angles θ and ϕ will in general lead to incommensurate ratios of the wall periods in sliding direction. Figure 5.2 shows this setup. An equivalent 2-d model without higher harmonics, was used recently by DALY et al. [151] for a study similar to that presented here.

In the following, we will be concerned with an analysis of mechanically stable positions for the impurity atoms and their motion as the walls slide against each other. The goal is to identify situations, where the trajectory of a mechanically stable position suddenly disappears, which would lead to a dynamical instability. The adiabatic trajectories were obtained as follows: in a given wall setup defined

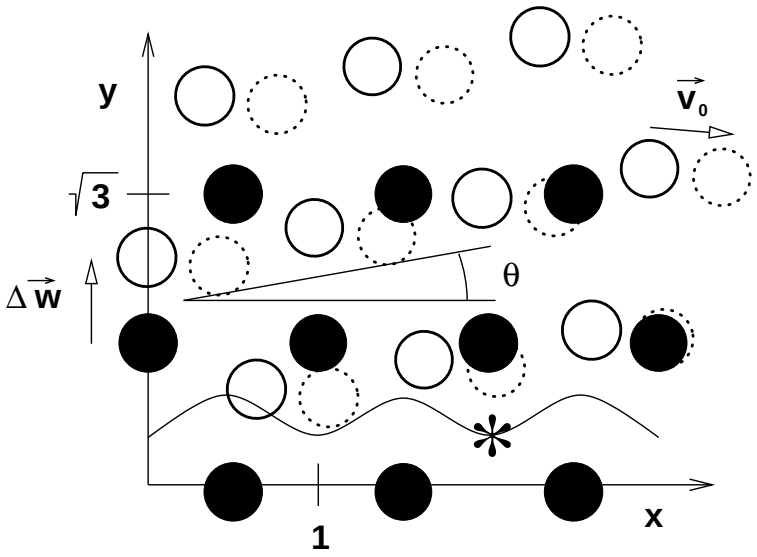

Figure 5.2: Top view of the superposition of surface potentials with hexagonal symmetry ("walls"). Circles are drawn at the positions of surface atoms. The bottom wall (full circles) defines the coordinate system relative to which the top wall (open circles) is shifted by $\Delta\mathbf{w}$ and rotated by an angle θ in the xy-plane. The top wall is sheared along $\mathbf{v}_0$ (open dotted circles). An asterisk is drawn at an (absolute) energy minimum of the bottom wall potential. The solid line depicts the collective adiabatic trajectory of particles of a sub-monolayer in a commensurate system ($\theta = 0°$) and $\mathbf{v_0}$ parallel to $\hat{x}$.

by $\Delta\mathbf{w}$, θ, and ϕ, an arbitrary initial position was chosen from which the closest local minimum of the combined wall potential was searched by a steepest descent method. This minimum defined the starting point of the adiabatic trajectory. The top wall was then moved by a step d$\mathbf{x}$ in the xy-plane in the direction of $\mathbf{v}_0$. the next local minimum in the vicinity of the old one was searched by the steepest descent procedure. By repeating these steps the adiabatic trajectory was constructed. A continuous trajectory will have only minimum displacements of $\mathcal{O}(\mathrm{d}x)$ at each step, while an instability will lead to large displacements outside the $\mathcal{O}(\mathrm{d}x)$ environment and will not change when varying the step width dx of the algorithm. A typical step width was $\mathrm{d}x = 10^{-3}$ and we defined an instability by a distance greater 0.1 between two subsequent points of the trajectory. This is somewhat arbitrary and might miss some pops as well as falsely detect some others. These cases are however rare. We checked that the instabilities were not significantly affected by a change of the step width $\mathrm{d}x = 10^{-2} \ldots 5 \times 10^{-5}$. This implies that instabilities will persist at arbitrarily small sliding velocities. Some instabilities disappeared when using a smaller step width which indicates a low barrier of height $\mathcal{O}(V^{(1,0)}\mathrm{d}x)$ which can be overcome by thermal activation of equal order. A systematic study with varied dx could thus be used to determine finite temperature effects.

5.4.2.2 Commensurate Walls

When absorbed molecules are present between commensurate surfaces the minimal energy is obtained for $|\Delta\mathbf{w}| \approx 0$ and $\theta = n\,60°$. All particles will then sit in the positions marked by an asterisk in figure 5.2, which are all equivalent. Various mechanically stable 'stacking' geometries can be envisioned for our walls of triangular symmetry, for instance hexagonal close packed (hcp) and face cubic centered (fcc) type configurations best characterized as respectively ABA and ABC layering structures. The boundary lubricant reflects the middle layer. While it does not correspond to an ideally crystalline layer, the probability for a lubricant atom to sit at a certain position would be indeed periodic, i.e. it would have a maximum in every single B position.

If the walls are slid parallel to a symmetry axis and the top wall is allowed to move freely in transverse direction there is an energetically favored collective motion $\Gamma_{\rm col}(t)$, which allows all particles to move along the valley of lowest energy between the ridges of the substrate molecules in $\phi = 0$ direction, as shown by the solid line in figure 5.2. Due to the symmetry between the walls the motion of the top wall is given by $\mathbf{v}_0 = 2\dot{\Gamma}_{\rm col}$. $\Gamma_{\rm col}$ is not exactly harmonic and its precise form depends on the coefficients of the Steele potential, but it can be approximately described by $\Gamma_{\rm col}(t) = |\mathbf{v}_0|(1/(4\sqrt{3})\cos(2\pi t)$. After every half lattice constant moved, the system will convert from an fcc type structure to an hcp type structure or vice versa. This behavior is also found in our 3-d default system (see Fig. 5.6), in which impurities interact with wall atoms through Lennard-Jones potentials rather than through Steele potentials.

Note that constraining the walls to zero velocity in the transverse direction leads to instabilities, because particles will in this case occupy positions with high potential which would not be occupied if transverse motion were allowed.

As the mechanically stable positions of the embedded impurities show no discontinuities, the kinetic friction force will tend to zero at small v_0 even if thermal fluctuations are absent. This will not happen linearly in general, but depend on the potential landscape. From the comparison to the 1-d model systems, one would expect a power-law behavior as in Eq. (5.20) with $F_{\rm k}(0) = 0$. This behavior does not depend on the sliding direction, but the form of $\Gamma_{\rm col}$ changes. For ratios of $V_{\rm t}(\mathbf{g}, z)/V_{\rm b}(\mathbf{g}, z)$ different from unity the qualitative behavior is similar.

A central issue of our study is the question how robust the property of the simple impurity model is as more complexity is added to the model. In the present case, one may argue that lubricant atoms would be able to move in a correlated fashion up to a coverage of one monolayer. Above this coverage, the impurity model breaks down, because not all particles can follow $\Gamma_{\rm col}$ anymore.

5.4.2.3 Incommensurate Walls

Impurity atoms between incommensurate walls have an infinite number of inequivalent minima in their potential energy landscape for a given relative wall displacement. This means that at a given moment in time, it is impossible to find two different positions where the value of the potential and all its derivatives are identical. Yet, the number of inequivalent trajectories of (meta)stable positions can be

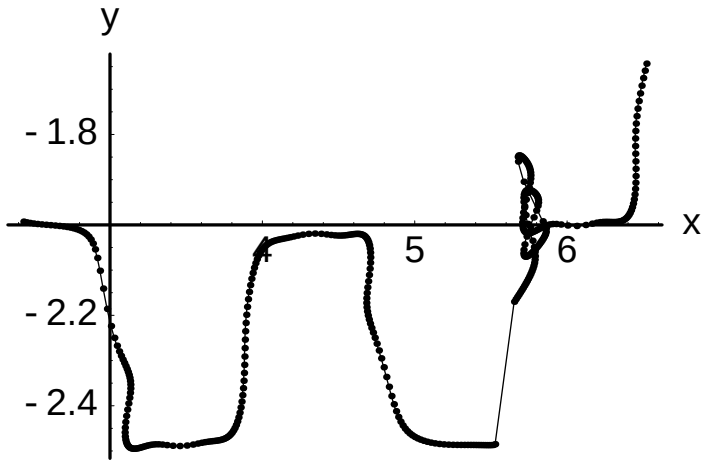

Figure 5.3: Example of instable trajectories in an incommensurate 2-d impurity model system. Walls are modeled by STEELE-potentials using only the fundamental harmonic contribution and moved with constant relative velocity. Each point corresponds to a lateral wall displacement of $\mathrm{d}x = 0.02$.

small, because in most cases, they will all be identical up to temporal shifts when the walls are in relative sliding motion. See also the discussion of the dynamics of the incommensurate PT-model by FISHER [157].

We analyze the instabilities by varying randomly the relative orientation θ between the two walls as well as the sliding direction ϕ. Unless θ is close to an integer multiple of 60°, we find that the number of instabilities depends only weakly on θ and ϕ. At this point, we are only concerned with the occurrence of instabilities, rather than with the (average) amount of energy dissipated during an instability. Instabilities between incommensurate walls are shown in Fig. 5.3. The similarity to the instable trajectory recorded during a simulation using LJ-potentials in Fig. 5.1 is well borne out.

If we chose the fundamental harmonics $\tilde{V}(\mathbf{g})$ of both walls to be identical and the higher harmonics to be absent, then we find on average one instability each time the upper wall has been moved laterally with respect to the lower wall by a distance of 200 d_{nn}. Increasing the interaction strength for just one wall does not change the behavior until the ratio $\tilde{V}_{\mathrm{t}}(\mathbf{g})/\tilde{V}_{\mathrm{b}}(\mathbf{g})$ or its inverse exceeds about 4.7. Above this threshold value, the metastable positions and hence the particles follow the motion of just one wall and no instability occurs anymore.

Like DALY et al. [151], we note that the instabilities are possible due to transverse motion of the impurities, see Figs. 5.1 and 5.3. One of the issues DALY et al. also discussed was the question above which value of $\tilde{V}_{\mathrm{t}}(\mathbf{g})/\tilde{V}_{\mathrm{b}}(\mathbf{g})$ the lubricant particle remains pinned to the (top) wall. They reported a value of 4.5, while we find a slightly higher value of 4.7, which essentially confirms their prediction. Furthermore, we also analyzed the effects of first higher harmonics, which were neglected in Ref. [151]. Including the first higher harmonics $\mathbf{g}_1$ in addition to the fundamental harmonics $\mathbf{g}_0$ increases the number of instabilities, in particular for higher harmonics with positive sign. For a ratio $V(\mathbf{g}_1)/V(\mathbf{g}_0) = 0.1$, the number of instabilities is increased by a factor of six. (At this ratio the absolute value of the

second harmonic would be in the order of 0.01 $V(\mathbf{g}_0)$, see Eq. (5.22), and is therefore rather small). One may argue that the observed increase in pops is related to an increase of incommensurability due to an additional length scale in the system.

We performed also studies with second higher harmonics and saw no qualitative change. Thus, the occurrence of instability remains a robust feature of incommensurate walls, independent of asymmetric interaction strengths and inclusion of higher order harmonics. This is a striking difference of 2-d and 1-d systems, due to the transverse motion of particles in 2-d.

5.4.3 3-d Model Systems

5.4.3.1 Model Details

We now turn to the analysis of a full three-dimensional model with crystalline walls, where wall atoms were fixed on their lattice sites. The general setup and simulation technique is described in detail in chapter 2. Simulations were done at constant pressure in z-direction, corresponding to constant load $N_{\text{wall}}L$ pressing down on the upper solid in $-\hat{\mathbf{e}}_z$ (thus L is positive). Note that L is the load per wall particle. Hence, the conversion factor to pressure is the wall area per particle, $\sqrt{3}d_{\text{nn}}^2/2$ and so depends on d_{nn}, the wall lattice constant.

The force $\mathbf{F}_{\text{t}}(t)$ acting on the top wall opposing the shear motion was recorded during the simulation runs. For commensurate systems $\mathbf{F}_{\text{t}}(t)$ was seen to be periodic with the wall corrugation with time constant d_{nn}/v_0. Thus, determination of the friction coefficient was done by averaging over complete periods after the system had reached the steady state. We checked that the configurations for sub-monolayers molecular lubricants did not contain crossed chains.

We concentrate on simulations with walls of size 19×11 unit cells. The distortion of the perfect hexagonal geometry is $1 - 19/(11\sqrt{3}) \approx 2.7 \times 10^{-3}$ and the number of wall atoms N_{walls} is given by $2 \times 2 \times 19 \times 11 = 836$. These walls confine a quarter layer (i.e. $N_{\text{fluid}} = N_{\text{walls}}/8 = 104$) of a simple fluid. All quantities are thereafter measured in reduced LJ-units. All interactions are purely repulsive and identical for fluid and wall particles. We set $r_{\text{cut}} = 2^{1/6}$, $\epsilon = 1$, $\sigma = 1$, and $d_{\text{nn}} = 1.20914$. This system with a quarter layer of spherical lubricant is what we call our "standard system".

The use of a purely repulsive interaction can be justified by the observation that at large pressures the essential behavior is caused by the repulsion of the particles. The main effect of including the attractive LJ contribution in the present context would be to add an adhesive pressure [40]. We have checked explicitely with simulations, that the tribological behavior is only slightly altered if the attractive part of the LJ potential is included. The changes can be understood by noting that higher order corrections in the Steele expansion are different in both cases.

Temperature was varied from $T = 10^{-3}\ldots1$, load from $L = 1\ldots100$ corresponding to pressures $P = L/(\sqrt{3}d_{\text{nn}}^2/2) = 0.032$ GPa$\ldots3.2$ GPa when setting in $\epsilon = 30$ meV and $\sigma = 0.5$ nm. The plastic yield stresses (pressure when plastic flow sets in) are in a range from 0.04 GPa (lead), 1 – 7 GPa (steel) to 80 GPa (diamond) [27, Table 5.1]. We are thus performing high pressure simulations. The lowest shear rate simulated was $3.3 \times 10^{-5} \approx 5.5$ mm/s, comparable to experiments.

Figure 5.4: Schematic geometric interpretation of commensurability. Two bare commensurate walls (left) can lock, whereas bare incommensurate surfaces (right) cannot lock if the lattice constants are (sufficiently) different.

Finite size (and finite commensurability) effects were checked by comparing simulations of systems of size 14×8, 19×11, and 28×16 wall unit cells, and no significant difference of the frictional behavior could be detected.

Wall planes lied in the xy-plane of the coordinate system, the constant shearing velocity v_0 was parallel to the x-direction $\hat{\mathbf{e}}_x$. The top wall was allowed to move freely in $\hat{\mathbf{e}}_y$, and we checked that even completely constraining the walls in $\hat{\mathbf{e}}_y$ did not alter the results significantly for incommensurate walls if the system was allowed to find an energetically favorable relative displacement first.

Checking the influence of the thermostat is crucial in our simulations. We compared two different methods: One method was thermostatting in the reference frame of the bottom wall in the directions perpendicular of the sliding direction (i.e. in $\hat{\mathbf{e}}_y$ and $\hat{\mathbf{e}}_z$), which had been extensively tested in [31, 40, 137, 158, 159] and found not to significantly influence the measured friction force unless very high shear rates were reached or the damping constant was set to an extremely high value. However, for our simulations we mainly used a second thermostat acting in all directions in the reference frame of the two centers of mass of the walls. Because this thermostat acts isotropically it was more adapted to the investigations of the distribution of the fluid particles velocities in the wall plane. Up to sliding velocities $v_0 \leq 0.03$ no quantitative differences in the friction force were found, at higher v_0 the thermostat in the fixed reference frame dissipated less energy than the thermostat in the center of mass of the walls, presumingly reflecting the missing direction for energy dissipation. We conclude that for the low sliding velocities we are primarily interested in, either method yields identical results.

5.4.3.2 Effect of Commensurability on Static Friction

While we are primarily focusing on kinetic friction, it is instructive to discuss the crucial role of surfactants for static friction which has been recognized in recent years [30–32]. It can be summarized by saying that static (and kinetic) friction would vanish for bare, incommensurate surfaces, because the bare interfaces cannot lock into each other as they could if they were commensurate. This difference is depicted in Fig. 5.4.

We use our standard system with 26×15 wall cells and allow the walls to diffuse freely in the wall plane so that the energy minimum can be found by the system. Thermostatting was done in the center of mass of the walls. Hence, the thermostat was not biasing the wall motion. The structure is analyzed by the static structure factor $S(\mathbf{q})$, Eq. (3.12), in the wall plane.

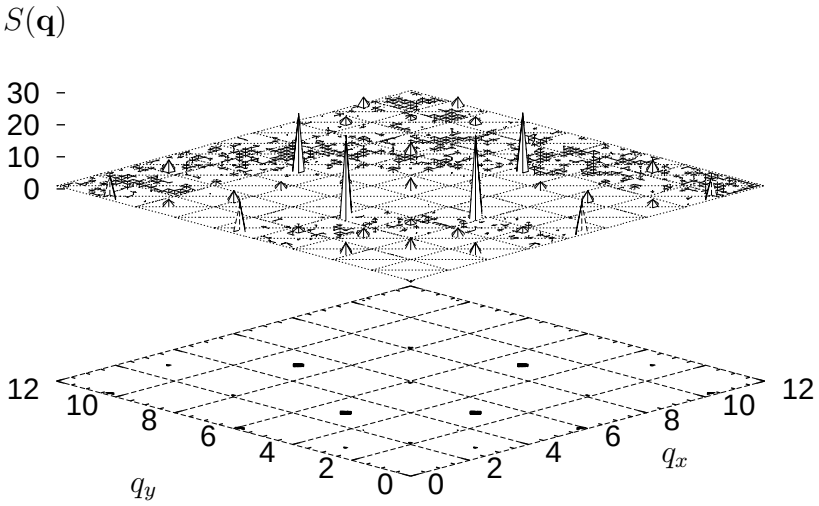

Figure 5.5: $S(q) > 1$ for a quarter layer of simple fluid confined between two incommensurate walls having hexagonal symmetry. 12 peaks for the fundamental (length $q_{\rm nn} = 4\pi/\sqrt{3}d_{\rm nn} \simeq 6$) and next higher BRAGG vectors are present in the full (q_x, q_y)-plane, indicating that the fluid simultaneously satisfies the corrugation of *both* walls. Note that peaks are cut in half at the coordinate axes.

Clearly, a corrugated surface will have an influence on the static structure of a boundary lubricant. The fluid atoms will prefer positions where they can minimize the fluid-wall potential. For commensurate (identical) walls one can predict the outcome. In this case the fluid atoms will sit in positions minimizing the energy with respect to both walls simultaneously, i.e. they all will occupy the minimum energy positions labeled with an asterisk in Fig. 5.2 for coverings below a monolayer. $S(q)$ has consequently the same symmetry where sharp peaks form a hexagonal lattice with lattice constant $q_{\rm nn} = 4\pi/\sqrt{3}d_{\rm nn}$.

In Fig. 5.5 we show $S(q)$ for *incommensurate* walls at $T = 0.6$ and $L = 3$. Due to the hexagonal symmetry of the wall atoms we expect to see six peaks arising from fluid atoms sitting in the potential minima of *one* wall at $q_{\rm nn} = 4\pi/\sqrt{3}d_{\rm nn} \simeq 6$, similarly for the next ring at $\sqrt{3}q_{\rm nn} \simeq 10.4$. The walls are rotated by an angle $\theta = 30°$ with respect to each other to obtain an incommensurate system. Thus 12 peaks are present in the fluid when it arranges to match the structure of *both* walls simultaneously, each peak being separated by 15°, as we observe in Fig. 5.5. At this low covering, the wall corrugation completely dominates the fluid structure, only weak indications of a ring corresponding to a nearest neighbor shell around $q \approx 5.5$ are visible. By adjusting to both walls simultaneously, the fluid locks together the two surfaces, resulting in finite (static) friction, which would be absent in *incommensurate* systems without lubricant. If there is no static friction, the kinetic friction must vanish for $v_0 \to 0^+$, too. This underlines the crucial role of surfactants for the occurrence of friction.

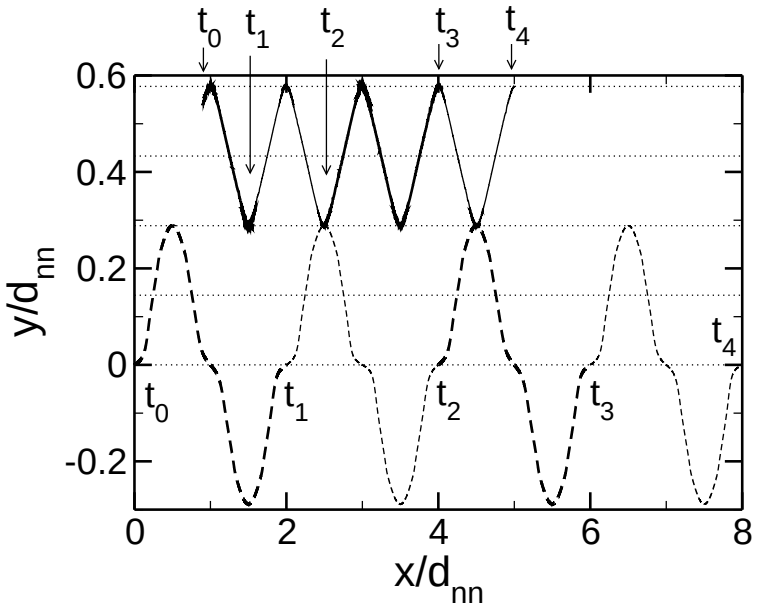

Figure 5.6: Trajectory of a tagged particle (solid line) and of the upper wall (dashed line) for a commensurate system. The upper wall is moved parallel to x at constant velocity. Horizontal lines are drawn at intervals $1/4\sqrt{3}$. For integer values of $x/d_{\rm nn}$ the configurations can be identified as hcp, for half-integer values as fcc configurations.

5.4.3.3 Effect of Commensurability on PDs

In addition to the static structure, the dynamic properties also change dramatically with commensurability. Instead of an isolated impurity, we include here the interaction between the lubricant atoms and have a different potential and finite temperature in the full 3-d model. However, as the coverage is only a quarter layer, the results remain almost identical to the ideal impurity limit. Despite the changes with respect to section 5.4.2, all arguments discussed there remain valid under the new conditions. For instance, Fig. 5.6 shows the expected dynamical behavior of two *commensurate* walls separated by lubricant impurities in sliding motion, i.e., an alteration of hcp and fcc type configurations closely resembling the line of lowest energy in Fig. 5.2. Most importantly, the trajectories of lubricant atoms become continuous for commensurate walls. Trajectories in the commensurate system are completely different from incommensurate systems, cf. Fig. 5.1. Although the average fluid-particle velocity in x is $v_0/2$,we see that the motion in x is not uniform. This is because the particle can stick randomly to each wall with the same probability. For completeness, we mention that the simulations in Figs. 5.1 and 5.6 were both done for our standard system at an external load of $L = 30$ per top wall atom, a thermal energy of $T = 0.01$, and relative sliding velocity of $v_0 = 10^{-3}$.

The different trajectories of the mechanically stable states result in qualitatively different velocity distributions, even in the presence of thermal fluctuations, which is shown in Fig. 5.7 for the same system parameters as the trajectories, but various v_0. It can be seen that the velocity PDs of impurities between incommensurate walls can indeed be described with the non-equilibrium PD suggested in Eq. (5.16), i.e.

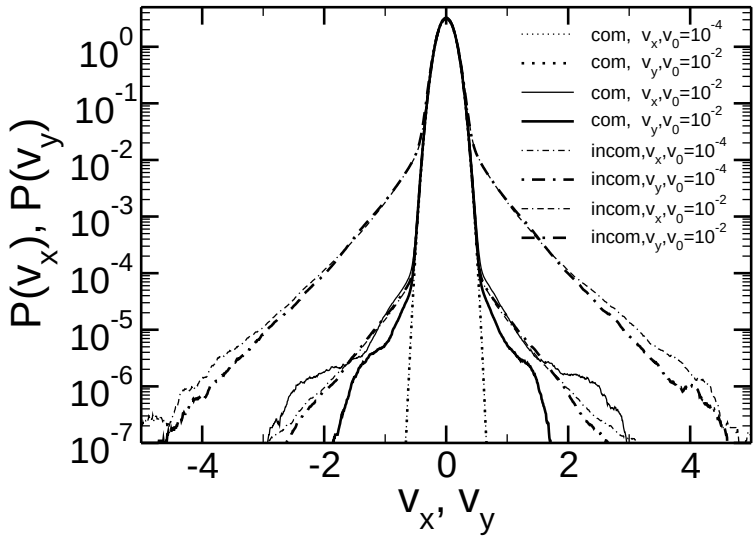

Figure 5.7: Probability distribution (PD) of the fluid particles' x- and y-velocity components for shearing velocities $v_0 = 10^{-4}$ and 10^{-2} for incommensurate and commensurate wall orientations at $T = 10^{-2}$ and $L = 30$ for our standard system. Around the central MAXWELL-BOLTZMANN PD wide tails develop upon shearing. The tails follow an exponential PD, see Eq. (5.16), which have similar magnitude parallel and transversal to the sliding direction. For commensurate walls, the tails are suppressed by two orders of magnitude at $v_0 = 10^{-2}$ and disappear completely for the lower shear velocity $v_0 = 10^{-4}$, when the PDs becomes almost indistinguishable from the MAXWELL-BOLTZMANN PD (not included).

a central MAXWELL-BOLTZMANN peak with exponential tails. It turns out that the PDs longitudinal (x) and transverse (y) to the sliding direction (x) are almost identical justifying the assumption of isotropy. We note in passing that the velocity PD normal to the interface (z direction) is affected much less than the in-plane PDs for the sub-monolayers we discussed so far.

5.4.3.4 Effect of Sliding Velocity and Temperature on PDs

As the relative sliding velocity between the walls is changed by a factor of 100, the prefactor of the exponential tail scales with the same factor, as suggested in Eq. (5.16). The commensurate walls behave differently. First, the non-equilibrium velocity distribution $P(v)$ deviates from equilibrium much less than for incommensurate walls and it does not obey Eq. (5.16) as well. More importantly, the tails of $P(v)$ behave differently from those of incommensurate surfaces under a change of sliding velocity. This difference is due to the absence of instabilities for the commensurate system. At $v_0 = 10^{-4}$, the velocity PD for commensurate walls is almost identical to the equilibrium MAXWELL-BOLTZMANN PD, while at the same v_0, the PDs for incommensurate walls show distinct non-equilibrium tails.

Further examination of the distribution functions for incommensurate surfaces as shown in Fig. 5.8 reveals that the coefficients A and B in Eq. (5.16) are approximately constant for a wide range of velocities and temperatures. The parameters can be easily read off the graphs: The slope of the tails equals B and the exponential

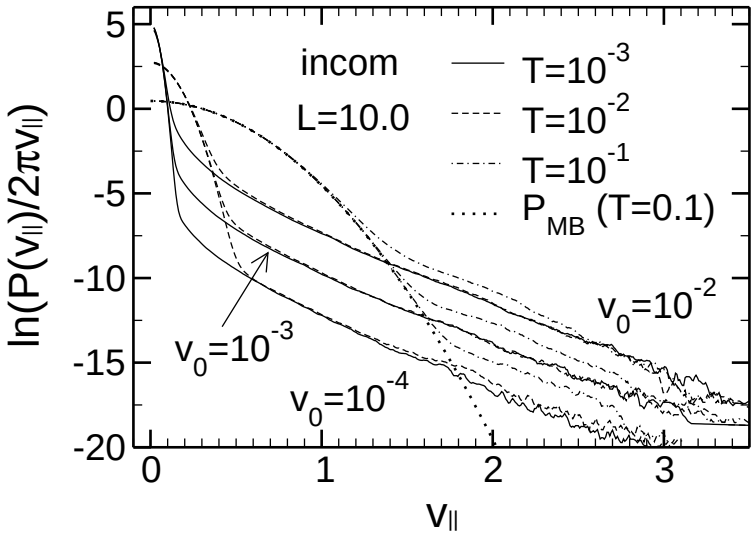

Figure 5.8: Probability distribution $\ln\left[P(v_\parallel)/2\pi v_\parallel\right]$ at load $L = 10.0$ for temperatures $T = 10^{-3}$, $T = 10^{-2}$, $T = 10^{-1}$ and sliding speeds $v_0 = 10^{-3}$, $v_0 = 10^{-2}$, and $v_0 = 10^{-1}$. At low in-plane velocities $v_\parallel$ a thermal peak described by the MAXWELL-BOLTZMANN PD (at $T = 0.1$ exemplified by a thick dotted line) dominates, before the PD crosses over to exponentially distributed tails described in Eq. (5.16). The slope of the tails, B, is independent of both T and v_0. The prefactor of the tail distribution is proportional to v_0 and changes at large temperatures.

of the y-axis intercept of a fitted line through the tails equals Av_0. The data for Fig. 5.8 were produced with load $L = 10.0$ for temperatures $T = 10^{-3} \ldots 10^{-1}$ and sliding speeds $v_0 = 10^{-3} \ldots 10^{-1}$.

The present discussion is valid when the non-equilibrium tails are clearly visible such as in Fig. 5.8. It becomes invalid when v_0 reaches extremely small values, i.e., when the tails are starting to disappear under the central MAXWELL-BOLTZMANN peak. Eq. (5.16) then ceases to be a good description of the PDs in that limit and Eqs. (5.16) and (5.18) are no longer applicable. However, the equation describing the heat-flow balance between thermostat and confined system, Eq. (5.13), is unaffected by this argument and remains valid even in the limit $v_0 \to 0$.

5.4.3.5 Effect of Load on PDs

The load dependence of the coefficients A and B was investigated as well. We show the effect of load on the PDs for one of our model systems exemplarily in Fig. 5.9. Many similar calculations were done for other loads, coverages, and sliding velocities with similar results for incommensurate surfaces. In all cases, we found that A is roughly proportional to $L^{-0.8}$, while B is quite accurately proportional to $L^{-0.4}$.

From the normalization factor of the central equilibrium peak in Eq. (5.16), one may infer that the ratio A/B^2 is a measure for the number of atoms far out of equilibrium and hence for the number of invoked instabilities. Given the proportionalities $A \propto L^{-0.8}$ and $B \propto L^{-0.4}$, this number remains constant when L is increased. Inserting the proportionalities $A \propto L^{-0.8}$ and $B \propto L^{-0.4}$ into Eq. (5.18) results in a small deviation from AMONTONS' law $F_\mathrm{k} \propto L$ at the microscopic level.

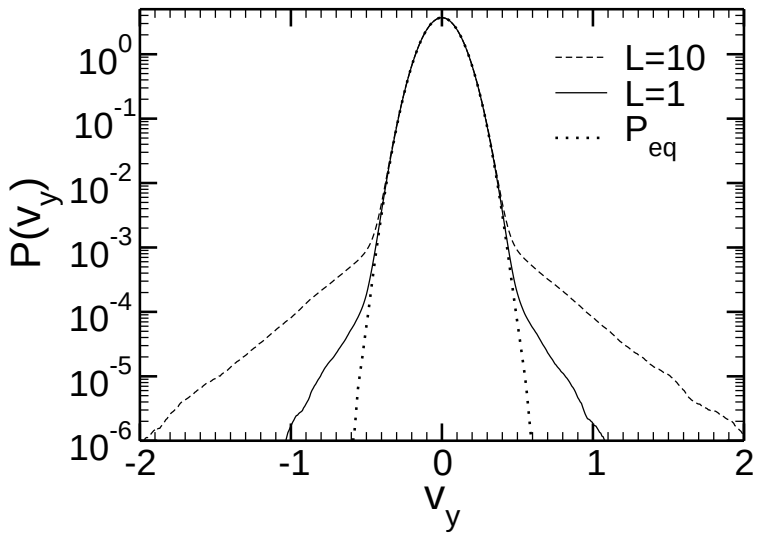

Figure 5.9: Probability distribution $P(v_y)$ at $T = 0.01$, $v_0 = 10^{-3}$ and two different loads $L = 1.0$ and $L = 10$. The thermal equilibrium distribution P_{eq} is inserted for comparison.

Potential differences scale with L in lowest order, thus we obtain for the energy dissipated in a pop $\Delta E_{\mathrm{diss}} \propto L \propto v^2$. Hence, for exact proportionality, the width of the non-equilibrium tails was $\propto L^{0.5}$, respectively $B \propto L^{-0.5}$, yielding AMONTONS' law. This shows that $B \propto L^{-\lambda}$, $\lambda \approx 0.5$ is to be expected, while the precise value of the exponent λ will depend on the specific system potentials.

The deviation in our system is due to a shift of the relative significance of lower- and higher-order harmonics. This shift would presumably be smaller if the repulsive forces were modeled with (slightly more realistic, cf. Ref. [156]) exponentially repulsive forces [32].

Fig. 5.9 reveals that the exponential tails fall off less slowly when the pressure is increased. Thus, large pressures in sliding contacts can dramatically increase the probability of large velocities, even though the lubricant's average kinetic energy $\langle T_{\mathrm{kin}} \rangle$ (or effective thermal energy) may barely change. This favors the occurrence of rare events such as chemical bond breaking, as it becomes much more likely that a bond is hit by high-velocity atoms than expected from a MAXWELL-BOLTZMANN PD. It will thus be difficult to assign a unique effective temperature that reflects at the same time the reactivity of the molecules in the junction and the energy contained in their motion.

5.4.3.6 Comparison Between Calculated and Measured Friction Coefficients

The fit of curves equivalent to those shown in Figs. 5.7 and 5.8 allows one to estimate the kinetic friction force F_{k} with the help of Eq. (5.18). This result can then be compared to the friction force that is measured directly in the simulation. It turns out that such a comparison typically leads to an agreement within approximately 25 % accuracy, which can be improved by also taking into account the effects of instabilities on the motion normal to the surfaces. The remaining deviation between

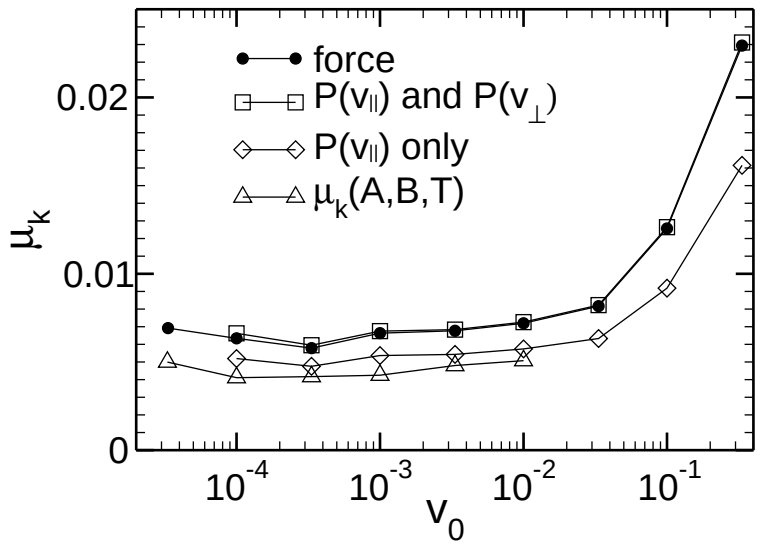

Figure 5.10: Comparison between the friction coefficient μ_k as measured directly at the wall and calculated indirectly through the non-equilibrium velocity distributions. In two cases, the true distributions $P(v)$ were used. Taking into account both in-plane velocities $v_\parallel$ and velocities $v_\perp$ normal to the interface results in perfect agreement, neglecting the normal contribution $P(v_\perp)$ underestimates μ_k. We also first fitted the PDs to Eq. (5.16), determined the coefficients A and B from the simulations and then calculated the kinetic friction force with Eq. (5.18). Quarter layer of lubricant, $T = 0.001$ and $L = 30$.

the 'predicted' F_k's and the directly measured F_k's is due to the fact that the tails are not exactly exponential. This is particular important when the temperature is large, load is small or v_0 extremely small, because our theory does not account for activated processes which become more important in these limits. If we accumulate the correct $P(v)$'s in the simulation and use Eq. (5.13) to predict F_k, the agreement between predicted and observed kinetic friction is almost perfect, also when v_0 is small. At very small v_0 the evaluation of F_k from the PDs of the simulation is however hampered by very bad statistics and shows strong fluctuations.

Fig. 5.10 shows the degree of agreement for one particular model system. One can see that the kinetic friction coefficients μ_k as obtained from the full velocity PD, see Eqs. (5.13) and (5.14) agree very well with the directly measured μ_k. Neglecting the contribution of the motion normal to the surface results in an $\mathcal{O}(20\%)$ underestimation of the friction force. Estimating μ_k indirectly with the help of Eqs. (5.16) and (5.18) leads to an underestimation of about 25%.

5.4.3.7 Effect of Temperature

It was shown by He and Robbins [40] that a model system similar to ours yields logarithmic velocity corrections to the friction force F_k for *incommensurate* surfaces, provided the temperature is positive and the sliding velocity is not too small, see also the discussion in Ref. [154]. Our simulation results of the v_0 dependence of μ_k for incommensurate, cf. Fig. 5.10, are compatible to an ansatz $\mu_k(v_0) = \mu_k(v_0 = v_{ref}) + C(T)\ln(v_0/v_{ref})$, with v_{ref} denoting a reference velocity.

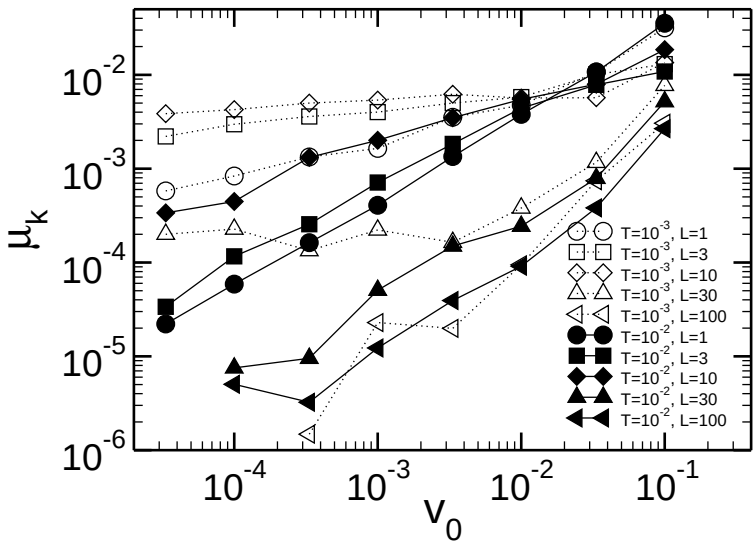

Figure 5.11: $\mu_{\rm k}$ of the commensurate standard system versus pulling velocity v_0 at different normal loads L and temperatures T. Note the logarithmic scale for the y-axis. In all cases $\mu_{\rm k}$ vanishes with a power law v^β as $v \to 0$, except for $T = 10^{-3}$, $L = 30$ where a constant, small value seems to be reached.

The prefactor $C(T)$ depended irregularly on the system parameters and the quality of the data was not sufficient to rule out other functional dependencies, however.

The basic reason for a logarithmic-type correction had already been recognized by PRANDTL [140]. Due to thermal fluctuations, the embedded atoms can jump over local energy barriers and the instabilities will be ignited prematurely. This reduces the necessary external force to maintain sliding, because it does not need to move the embedded atom all the way to the top of the energy barrier. It is conceivable, but difficult due to the two-dimensionality of the potential landscape to calculate them in our model using KRAMERS-theory [160].

For *commensurate* surfaces, discontinuous instabilities are absent and therefore the effect of thermal fluctuations must be different. This issue is investigated in Fig. 5.11. Due to the large loads and the small temperatures employed, the linear-response regime is not necessarily reached at the sliding velocities v_0 accessible to the simulations, i.e., $v_0 \approx 10^{-5}$. Therefore, we obtain kinetic friction coefficients $\mu_{\rm k}$ that apparently vanish according to

$$\mu_k \overset{v_0 \to 0}{\propto} v_0^\beta \,, \tag{5.23}$$

with exponents $0.25 \lesssim \beta \lesssim 1$.

It is remarkable that a small change in temperature has a rather strong effect on $F_{\rm k}$. For the small load $L = 1$, the exponent β is approximately unity at temperature $T = 10^{-2}$ and one may argue that the corresponding $F_{\rm k}(v_0)$ reflects a linear response curve. As T is lowered to $T = 10^{-3}$, a different exponent β is obtained, reflecting non-equilibrium behavior. When the load is now increased by a factor of ten, the energy barriers also increase approximately by a factor of ten. Therefore the $F_{\rm k}(v_0)$ curves belonging to masses $L \gtrsim 10$ should be considered far from equilibrium, i.e athermal. This would favor exponents β less than unity. However, this expectation

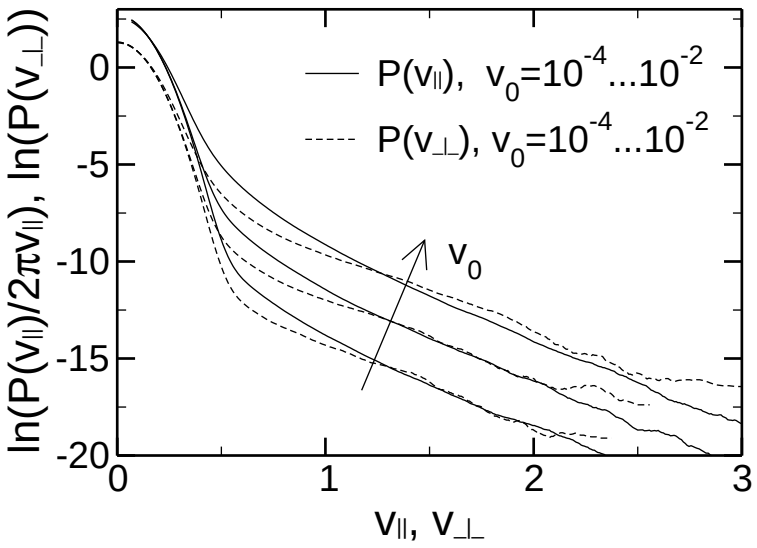

Figure 5.12: Distribution of the fluid particles velocity in plane, ($P(v_\parallel)$), and perpendicular to it, ($P(v_\perp)$), for an incommensurate system with two monolayers coverage at sliding velocities $v_0 = 10^{-4}$, 10^{-3}, and 10^{-2}. The central MAXWELL-BOLTZMANN parts are shifted because of the normalization $1/2\pi v_\parallel$ of $P(v_\parallel)$.

is not true. Instead a STOKES-type friction is observed. The almost linear relation of F_k and v_0 for these largest loads ($L = 100$) may thus be an effect due to higher harmonics in the lubricant-wall potential. The friction-velocity relationship can change qualitatively at certain critical values of the higher-order harmonics as reported for 1-d systems in Ref. [154].

5.5 Beyond the Impurity Limit

So far, we have neglected the *direct* interactions between the impurities or the coverages were small enough in order to render the direct interactions negligible. This approximation is reasonable when the coverage is small and when the lubricant particles are simple spherical units without inner degrees of freedom. When either condition is violated, the energy landscape and hence the detailed characteristics of the instabilities will change. This in turn might lead to a qualitative change in the tribological behavior of the junction. In this section, we will study the applicability, the limitations and corrections of the impurity limit model that are due to the interactions between lubricant atoms.

5.5.1 Coverage Effects

When the lubricant coverage is close to or greater than one monolayer and the junction is sheared, particles will have to move in a correlated fashion. In order for one atom to jump to another mechanically stable site, its neighbor has to jump as well, etc. A detailed description of the dynamics will be very complicated, i.e., it may involve sliding of correlated blocks along grain boundaries and the formation of dislocation-type structures [44]. Yet, the argument persists that instabilities and

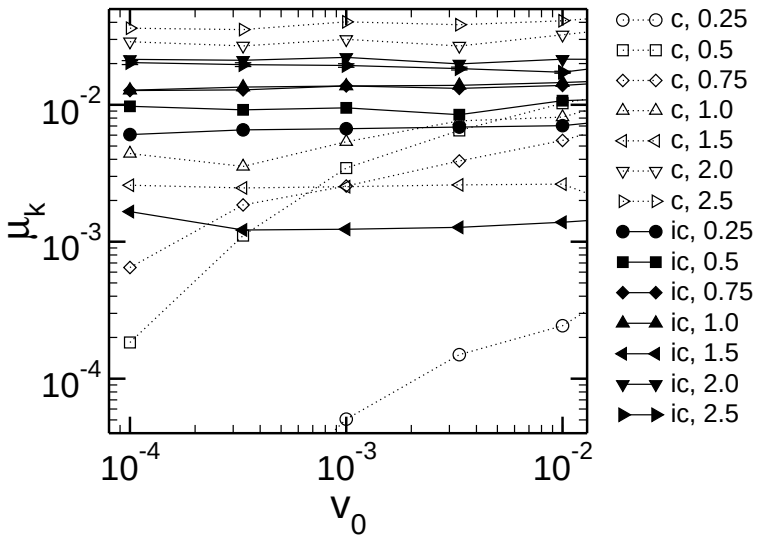

Figure 5.13: Coverage dependence of the dynamic friction coefficient μ_k of a system containing $0.25 \ldots 2.5$ monolayers of simple liquid at $T = 10^{-2}$ and $L = 30$. Commensurate systems (c) are denoted with open symbols, incommensurate walls (ic) with full symbols.

sliding induced deviations from the equilibrium velocity distribution function lead to friction.

Besides the correlated motion, some more details change when the coverage is increased. For example, pops also occur in the direction normal to the interface with a similar magnitude as parallel to the interface. This is reflected in the probability distributions $P(v)$ for the in-plane velocity $v_\parallel = \sqrt{v_x^2 + v_y^2}$ and the normal component $v_\perp = v_z$ of the fluid particles, see Fig. 5.12. The system under consideration is incommensurate, the walls are separated by a double layer and the externally imposed load per wall atom is $L = 30$. Although the detailed dynamics of the lubricant atoms must be very different from those in the impurity limit, Eq. (5.16) provides again a reasonable description for respectively $P(v_\perp)$ and $P(v_\parallel)$, i.e., a central MAXWELL-BOLTZMANN peak and a non-equilibrium exponential tail. Similar curves, which are not shown explicitly, were obtained for a coverage up to 5 monolayers.

As before, the kinetic friction force F_k could be obtained the integral over the deviation of the $P(v)$'s from the MAXWELL-BOLTZMANN distribution, as stated in Eq. (5.13). F_k from simulations is shown for various coverages and sliding velocities in Fig. 5.13. Both commensurate and incommensurate systems are investigated and again their behavior is strikingly different.

We start our discussion with the commensurate system. At a coverage of $C = 0.25$, results are very close to the impurity limit. F_k decays to zero with a power-law v^β where the exponent β is less than one. As the coverage is increased to $C = 0.5$ or even $C = 0.75$, F_k decreases considerably less quickly with decreasing v_0 than in the impurity limit. The behavior remains strikingly different from COULOMB friction. This changes when the coverage reaches and exceeds one full monolayer. For coverages beyond double layers, the kinetic friction force even exceeds that of

incommensurate systems. The prediction in Ref. [154] that commensurate systems should show smaller kinetic friction than incommensurate system must thus be limited to extreme boundary lubrication. Above one monolayer lubrication, this trends seemingly turns around. Experiments suggest that commensurability leads to enhanced friction between mica surfaces lubricated by a double layer or more [161]. Unfortunately, no study is known to us in which a monolayer of lubrication or less was used between two (smoothly) sliding *commensurate* walls.

At the smallest velocity investigated, μ_{k} increases by a factor greater than 200 for the *commensurate* case, when we increase the coverage from $C = 0.5$ to $C = 2$. The same change in coverage for incommensurate surfaces only yields a factor of two. Hence, incommensurate surfaces show much weaker coverage dependence than commensurate interfaces. Overall, there is relatively little change of F_{k} with coverage for incommensurate walls with the exception of $C = 1.5$. Due to the large load employed, the 1.5 monolayers are squeezed into a single, crystalline layer, which then essentially acts like a solid. This situation would not occur – or at least occur only for a short period of time – if the lubricant could flow out of the junction. We conclude that the coverage dependence is weak for incommensurate walls.

5.5.2 Effects due to Molecular Bonds

Most lubricant particles possess an inner structure. Here we will focus on the most simple generalization of the spherical molecules considered so far, namely dimers, and hexamers (6-mers). Dimers would represent small linear molecules such as C_2H_6, while hexamers are representative of short, linear alkane chains. The dynamics of the lubricant particles will change due to the additional internal degrees of freedom. Alternatively, one may argue that the dynamics of monomers is restricted because every monomer is constraint by at least one chemically bonded neighbor.

While monomers only have translational degrees of freedom, dimers also have *rotational* degrees of freedom. It is tempting to speculate that 'rotational' instabilities can occur in addition to the 'translational' instabilities. Therefore, one might expect F_{k} to be larger for dimers than for monomers. However, the rotational and translation motion will not be independent of each other and the coupling between them might reduce the effect of a 'translational' instability. The question which effect dominates can only be answered by analytical calculations or by molecular dynamics simulations. Simulation results for the kinetic friction force in a boundary-lubricated interface are shown in Fig. 5.14.

In Fig. 5.14, one can learn that the ratio $d_{\mathrm{nn}}/d_{\mathrm{mol}}$ of the next neighbor spacing d_{nn} in the walls and the intra-molecular bond length d_{mol} plays an important role, particularly for commensurate surfaces. When the intra-molecular bond length is close to the next-neighbor distance of wall atoms d_{nn}, F_{k} disappears as a power law with sliding velocity v_0. This means that the 'interference' effects between commensurate walls persist and that no instabilities occur. Surprisingly, this is even observed for hexamers. However, if d_{mol} differs from d_{nn}, instabilities also occur in boundary-lubricated systems, even for *commensurate* walls. These instabilities are invoked through the rotational degrees of freedom. While the misfit between d_{mol} and d_{nn} leads to COULOMB friction between commensurate walls, its value of F_{k} remains small as compared to the incommensurate case.

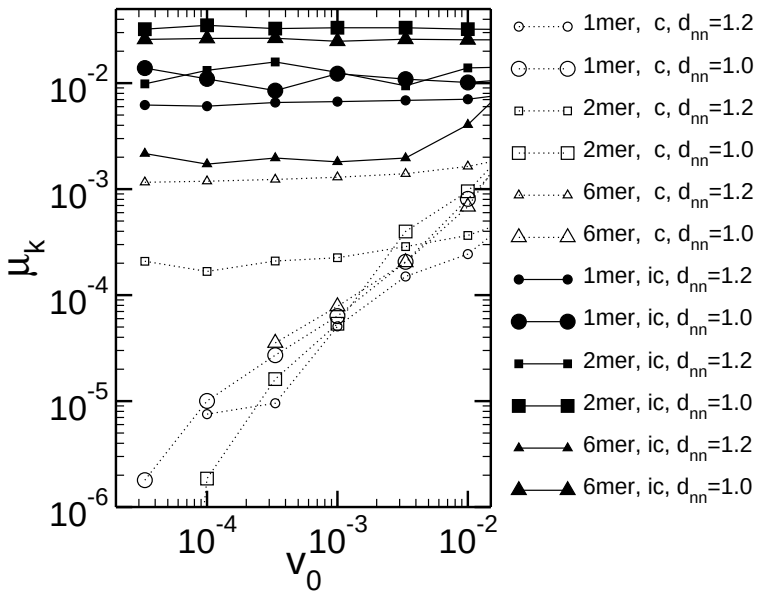

Figure 5.14: Dependence of the friction coefficient $\mu_{\rm k}$ on the wall lattice constant $d_{\rm nn}$ for a quarter monolayer of a dimer and a 6-mer with bond-length $d_{\rm mol} = 0.967$ for commensurate (c, open symbols) and incommensurate (ic, full symbols) orientation at small temperatures $T = 10^{-2}$ and large loads $L \approx 30$ (adjusted to yield identical pressures).

We now turn to the incommensurate walls. Interestingly, the smoother walls with $d_{\rm nn} = 1.0$ produce higher kinetic friction than the walls with $d_{\rm nn} = 1.2$, while the opposite is true in the case of static friction $F_{\rm s}$, as the potential wells are deeper in a system with greater $d_{\rm nn}$. The reason for the effect in $F_{\rm k}$ is that the reduced nearest neighbor spacing leads to a higher rate of popping processes, as more atoms sliding past each other at a given v_0 while the energy gain in the pops is only slightly decreasing. This effect can be verified by comparing the distributions of the particle velocities. It remains stable for all degrees of polymerization. The high friction of a dimer is caused by the contribution of their fast rotations. This is revealed by the distribution of the bonds' angular velocities. Despite these trends, incommensurate systems prove again to be less qualitatively susceptible to quantitative changes in the parameters that determine the details of the model than commensurate systems.

The quantitative effect of the chain length on friction in the generic case of incommensurate systems with different bond-length and $d_{\rm nn}$ needs more consideration. In Fig. 5.14 the hexamers exhibit the lowest friction. On the contrary, at lower loads and otherwise unchanged conditions we find that chain molecules show higher friction. Further studies are needed to clarify the interplay of surface corrugation, temperature, load and the different degrees of freedom.

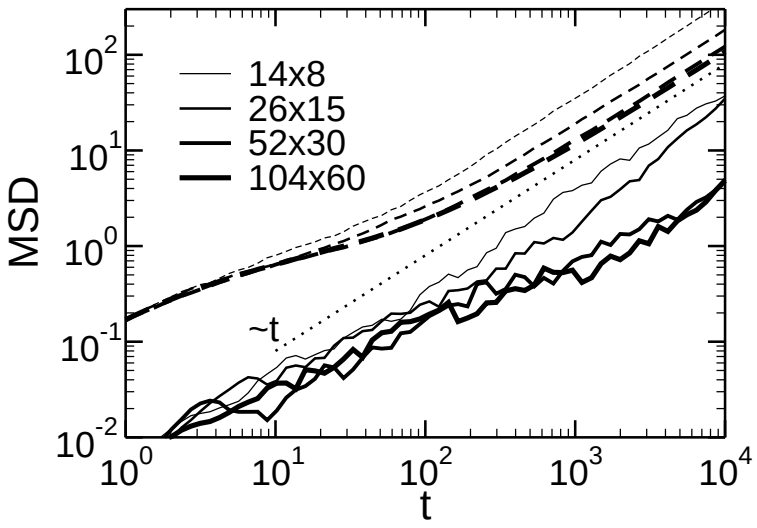

Figure 5.15: Mean square displacement (MSD) of the fluid atoms in a quarter layer of simple lubricant (dashed lines) and for the top wall (full lines) for different system sizes (increasing with line thickness), measured in wall cells (each containing two atoms), at $T = 0.6$ and $L = 3$. The MSD's of the relative wall position are multiplied by the number of wall atoms divided by the number of wall atoms of the smallest system to compare effects not due to the trivial mass dependence. The dotted line indicates free diffusion (MSD $\propto t$).

5.6 Wall Diffusion with Molecular Boundary Lubricant

While we have so far concentrated on kinetic friction with constant externally imposed sliding speed, this section presents results on the wall diffusion. A large diffusion constant for the wall motion indicates low static friction. Constant velocity kinetic friction is a non-equilibrium process, whereas diffusion is an equilibrium process. Thus, kinetic friction cannot be related immediately with diffusion and static friction. However, in a model similar to ours, F_{k} was found to be constantly $20\% \pm 5\%$ lower than F_{s} for *incommensurate* systems [40]. As our discussion in sections 5.4.3.2 and 5.4.3.3 shows, F_{s} is larger for *commensurate* systems, while F_{k} vanishes for sub-monolayer coverings (unless bonds introduce incommensurability, in which case F_{k} is still much smaller). Hence, we can only connect diffusion and kinetic friction for *incommensurate* systems.

Moreover, previous work [39] suggested that the coupling of the fluid atoms to the walls might lead to walls which are pinned to each other although the fluid atoms are still highly mobile. If such a feedback mechanism due to the walls was present, we expect to see a non-trivial dependence on system size of the wall diffusion as measured by the mean square displacement (MSD) while the fluid MSD is unaffected. A pinning would lead to a constant MSD which was reported in Fig. 11 of Ref. [39] for a quarter-layer of hexamers. As Fig. 5.15 shows for a simple liquid, we could not find such a behavior in our simulations. Although this figure was produced at higher load, lower temperature and for bigger systems, no wall pinning was observed.

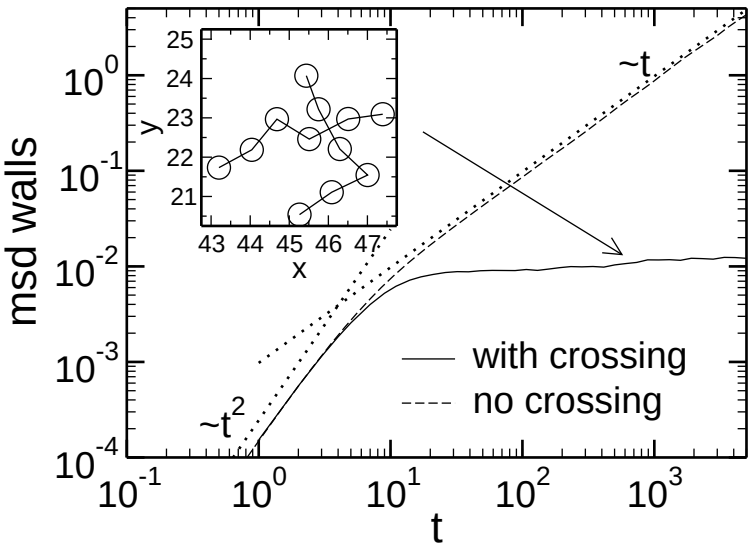

Figure 5.16: Mean square displacement for walls confining a quarter layer of hexamers at $T = 0.8$ and $L = 2$ (wall size 62×36 unit cells). If two chains cross (inset) the MSD of the walls reaches a constant, indicating pinning of the walls (solid line). If no crossed chains are present, the MSD changes from ballistic (MSD $\propto t^2$) to free diffusion (MSD $\propto t$) at $t \approx 10$.

Each of the aforementioned parameter changes lowers diffusion and should therefore increase the tendency to pin. The two smallest systems show finite size effects, but the curves seem to converge for increasing system size. In all cases free diffusion is reached at the latest times. Similar results were obtained for all investigated temperature-load combinations, regardless of the degree of polymerization of the lubricant. This suggests that non-trivial coupling effects leading to wall pinning and high static friction are not present. Consequently, we expect that kinetic friction is not enhanced by such effects if we assume $F_\mathrm{k} \propto F_\mathrm{s}$.

The apparent pinning for the system studied in Ref. [39] could instead be linked to crossed chains present in the simulated configuration but absent in our simulations, cf. section 2.6. We show in Fig. 5.16 a comparison of a system with just one occurrence of crossed chains (inset, view normal to wall plane) and of a system without such crossings using the same system parameters as in Ref. [39], $T = 0.8$, $L = 2$, and walls of size 62×36 unit cells. One clearly sees that the crossing leads to pinning and consequently to high static friction which is in turn an indication for high kinetic friction. These crossings are very stable because they represent a thick 'knot' carrying a huge load. The presence of such crossed chains is not an equilibrium configuration, but is metastable.

For molecules without crossings in the wall plane, the diffusive behavior is very similar to simple fluids as we can see in Fig. 5.17 where we plotted the same quantities as in Fig. 5.15, but for 10-mers. A comparison of the absolute magnitudes of the MSD's shows that for 10-mers both monomers and wall diffuse more slowly indicating a higher friction for molecules for this coverage, T, and L combination.

If two surfaces with surfactants consisting of molecules are brought in contact, it is indeed likely that two chains residing on facing surfaces will cross in the resulting

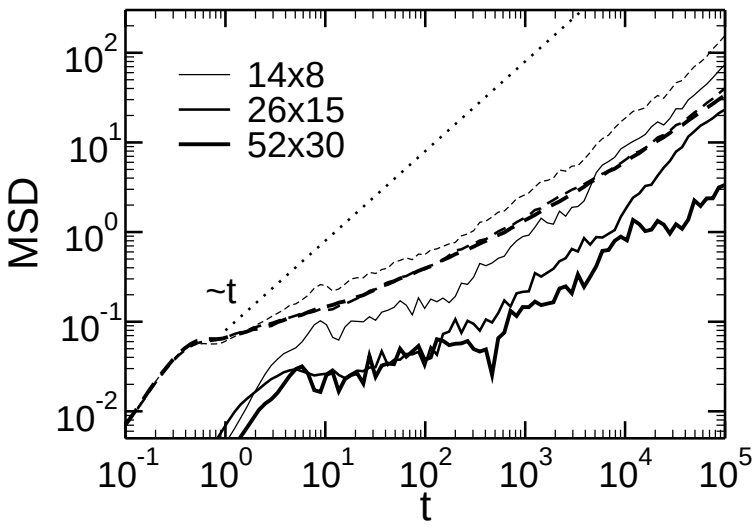

Figure 5.17: Mean square displacement (MSD) of the monomers of a quarter lubricant layer of 10-mers (dashed lines) and for the top wall (full lines) for different system sizes (increasing with line thickness) as in Fig. 5.15. The MSD's of the relative wall position are normalized to the wall size. The dotted line indicates free diffusion (MSD $\propto t$) with the same prefactor as in Fig. 5.15 for comparison.

contact leading to high friction. However, it is conceivable that the chemical bonds would break due to the high load they have to bear, or that the chain crossing is removed by the sliding motion. This emphasizes that the frictional behavior strongly depends on boundary lubricants and their structure.

5.7 Summary and Conclusions

Kinetic friction requires the prevailance of instabilities (mechanical hysteresis) in a system. In this study, we have focused on instabilities in the trajectories of particles confined between two walls, which are sheared against each other. When an instability is reached, the particle does not find a local potential energy minimum in its vicinity anymore and is thus forced to 'pop' into the next local minimum it sees. At small sliding velocity v_0, this will lead to a high velocity, which depends solely on the energy landscape. The kinetic energy is gradually dissipated, resulting in a frictional force. We derived a relationship between the (non-equilibrium) velocity distribution function $P(v)$ and the friction force F_k. The characteristics of $P(v)$ and thus F_k depend only weakly on coverage, sliding velocity, load and other parameters for incommensurate surfaces.

In a generic setup, we first used two Steele potentials reflecting two two-dimensional, triangular walls, which could be rotated with respect to each other to achieve an incommensurate system. We then computed numerically the adiabatic trajectory of a test particle. It was found that instabilities were a robust feature of the incommensurate system. Different off-symmetry wall rotations and inclusion of higher-order contributions to the Steele potential as well asymmetric interaction strengths of the walls did not alter the occurrence of the instabilities,

but only affected their frequency.

Including interactions between lubricant atoms does not change the existence of instabilities and hence the presence of COULOMB friction either. In contrast, the commensurability of the walls allowed for especially smooth trajectories of impurity atoms. The trajectories remain smooth when interactions between lubricant atoms are included up to a coverage of one monolayer. Above one monolayer, the lubricant atoms do not move coherently any longer and instabilities are starting to occur within the film. Kinetic friction rises dramatically as a consequence.

We speculate that coherent motion similar to the one just described also lead to the behavior observed in a pioneering quartz crystal oscillator study by KRIM and CHIARELLO [162, 163]. They found that the friction between a *solid* monolayer and a smooth surface was much smaller than the friction between a fluid monolayer and the same surface. The reverse was reported for a rough surface, in which case collective motion of the film can no longer occur. While KRIM and CHIARELLO could not make a final statement concerning the commensurability between the adsorbed Krypton film and the smooth gold surface, their results clearly support the following picture: Coherent motion of a layer, be it adsorbed or "between-sorbed", suppresses instabilities. This in turn reduces kinetic friction, irrespective of commensurability. Static friction, on the other hand, would certainly be increased for commensurate alignments.

We turn back to the discussion of the non-equilibrium velocity distributions. For incommensurate walls, the distribution consists of a central peak which is essentially identical to the equilibrium velocity distribution and additional non-equilibrium tails. These tails fall off only exponentially with v, which is slower than the exponential decay with v^2 in equilibrium systems. This observation is rather generic for incommensurate systems and independent of the lubricant coverage. The load dependence of the non-equilibrium tails provides a microscopic explanation for AMONTONS' law.

As the real velocity distribution function is qualitatively different from GAUSSians, it seems futile to describe the interface in terms of an effective temperature. We argued that given a specific kinetic energy associated with a lubricant (which could be used to define an effective temperature), the non-equilibrium system would be more likely to invoke chemical bond breaking or other chemical reactions.

Overall, the impurity model provides a good description of the typical characteristics of a boundary-lubricated system. However, it is essential to study two-dimensional interfaces and incommensurate surfaces. One-dimensional and/or commensurate surfaces lead to untypical behavior, i.e. rather large sensitivity of the friction force with respect to small changes in the model (details of interaction potential) or in the external parameters (sliding velocity, load, temperature, etc.). This is unfortunate, because incommensurate walls are much more common than commensurate walls, which leaves us with fewer possibilities to control friction.

A surprising result of our study for incommensurate walls is that increasing the atomic scale roughness of the walls may actually sometimes reduce the kinetic friction force.

Chains that lie over each other in a confined molecular film lead to very stable crossed bond geometries carrying a high load resulting in high friction. If these cross-

ings are absent, the quantitative influence of the chain length depends on external parameters.

It would be interesting to compare our predictions concerning the velocity distributions to experimental data. While scattering data from small, confined volumes is certainly notoriously difficult to obtain, recent advances have been made. Using fluorescence correlation spectroscopy, MUKHOPADHYAY et al. [164] measured translational diffusion in molecularly thin liquids confined within a surface forces apparatus. In the future, it might be possible to extend these studies to sliding situations where velocity distributions can be measured.

Further promising studies could be done on the dynamics of boundary lubricants, like the role of surface corrugation on relaxation processes such as spatially heterogeneous diffusion as reported in Ref. [164]. The transition from the 'instability picture' to the viscous friction regime in thick films would be another interesting topic.

Appendix A

Two-Dimensional Polymer Films

The efficient construction of chains, or more general, self-avoiding random walks (SARW) in two dimensions is a challenging problem itself as discussed in section 2.6.1. We investigate the construction of polymers with fixed bond-length r_{bond} and purely repulsive LENNARD-JONES interaction in the KREMER-GREST model as defined in section 2.3. All simulations were carried out at a reduced temperature of $T = 1.0$.

To our knowledge this is the first time the "Recoil-Growth" (RG) algorithm we introduced in section 2.6.1 was employed for a study of 2-d polymers. We will therefore investigate the properties of the RG-algorithm. For the single chain we demonstrate a strong dependence of the acceptance rate on the parameter choice. Results for the static structure of single chains are compared with theoretical predictions.

Two-dimensional polymer systems are furthermore interesting themselves as they provide the limit for (3-d) thin films. We give a table with optimal parameter choices for the RG-algorithm when discussing multi-polymer systems. To close we propose some possible investigations for these dense 2-d films.

A.1 Single Chain Results

We first concentrate on single chains. Each accepted MC-move of the RG-algorithm thus creates a completely new configuration. In Fig. A.1 we show an example of a chain of length $N = 2048$, which is the biggest length we investigated. One sees that the structure displays self-similarity. In addition we see that the chain folds back several times. These are events where configurational bias monte carlo (CBMC) is likely to fail because it gets 'trapped' in a 'dead-end' (cf. section 2.6.1).

The RG-algorithm has two adjustable parameters, N_{choice} which defines the number of possible 'branches' at each monomer where new monomers can be grown and N_{recoil}, being the limit of steps the RG-algorithm can go back when has entered a dead-end. Note that N_{choice} does not need to be an integer, but can denote the *average* number of branches (e.g. $N_{\text{choice}} = 2.5$ means that there are alternatingly 2 and 3 branches). The dependence of N_{recoil} was seen to be moderate, the dependence on N_{choice} however becomes dramatic for long chains. As we learn from Fig. A.2, a pronounced maximum of the acceptance probability for chain moves P_{acc} develops for long chains. For $N \leq 64$ it suffices to roughly adjust $N_{\text{choice}} \approx 1.9$. For the longest chains simulated, we get an optimal $N_{\text{choice}} = 1.838$. As $1/(2 - 1.838) = 6.173$

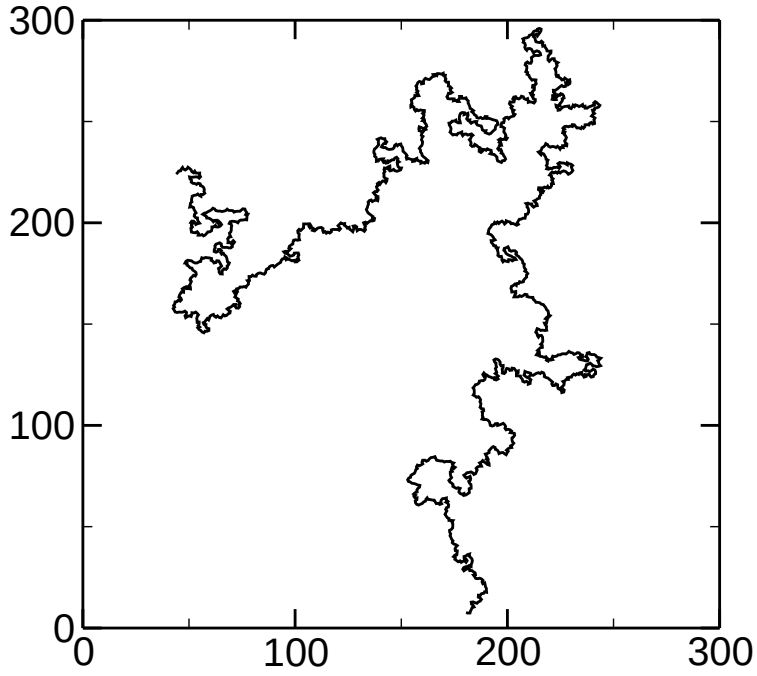

Figure A.1: Example for a single chain of length $N = 2048$ in two dimensions. The structure is self-similar.

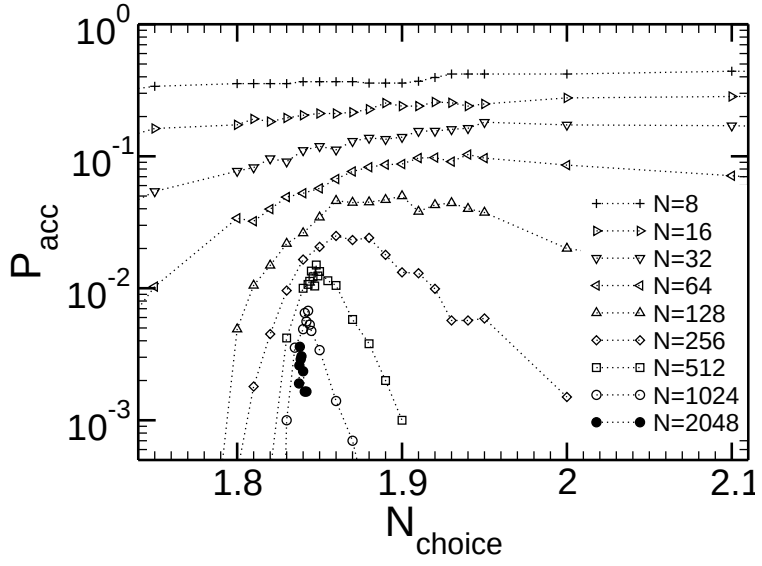

Figure A.2: Acceptance rate $P_{\rm acc}$ dependence of the RG-algorithm on $N_{\rm choice}$, with fixed $N_{\rm recoil} = N/2$ for various chain-lengths N. The sharp peak in $P_{\rm acc}$ lies at $N_{\rm choice} = 1.838$ for $N = 2048$.

this means, that there are 5 (or 6) monomers with two possible branches for new bonds, interrupted by a monomer where only one direction for a new bond is tried out. One may speculate that 6.173 is a special length for our polymer model. We also deduce from the figure that the acceptance rate falls linearly with the chain-length, $P_{\rm acc} \propto 1/N$ for optimal parameter choices. The energy computation of each monomer constructed scales with $\mathcal{O}(N^2)$ (monomer pairs in a chain) in our imple-

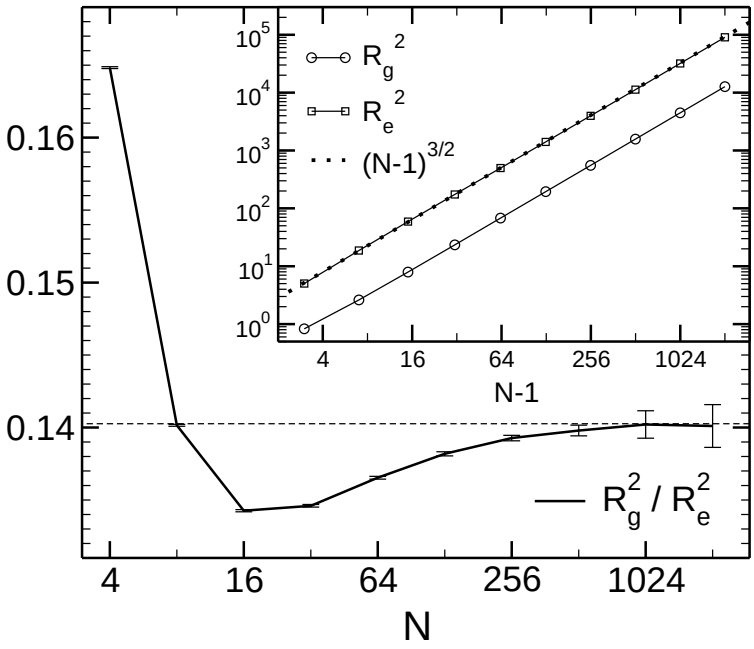

Figure A.3: R_e^2 and R_g^2 versus $N-1$ (inset) and the ratio of both quantities. The predicted universal scaling laws for the radii with an exponent 3/2 are reached quite early. The dashed line in the main figure is the predicted large N limit for the ratio R_e^2/R_g^2 from Ref. [165].

mentation. This has to be done for $\mathcal{O}(N)$ monomers in the chain, so that we arrive at a complexity of $\mathcal{O}(N^4)$ for the construction of a completely new chain which is accepted. This is only an upper bound, as some chains are rejected earlier and operations are saved and there might be further corrections due the tree structure of the algorithm. Nonetheless, this shows that it becomes difficult to reach much higher N with the RG-algorithm. We note that the so-called "Pivot"-algorithm has a complexity of $\mathcal{O}(N)$ for SARW (a single chain) [56]. The RG-algorithm however works also in dense systems and outperforms conventional CBMC by far even in 3-d [34].

There are many theoretical predictions for 2-d self-avoiding random walks [56]. First, the end-to-end distance R_e and the radius of gyration R_g follow in leading order

$$R_g \propto (N-1)^\nu \,, \quad R_e \propto (N-1)^\nu \,, \quad N \to \infty \tag{A.1}$$

with an (exact) exponent $\nu = 3/4$ for 2-d. Note that the "N-1" arises from our convention to number monomers from 1 to N. In the theory of random walks one considers steps from the origin, i.e. $N-1$ steps from the fixed first monomer. Furthermore, the ratio of the quantities is [165]

$$\lim_{N\to\infty} \frac{R_g^2}{R_e^2} = 0.14026 \pm 0.00011 \,. \tag{A.2}$$

Figure A.3 shows us that both predictions are beautifully borne out in our data. The large N scaling of R_e and R_e with ν is reached surprisingly early (on this log-log plot). Deviations are only visible when investigating the ratio of both. The data for

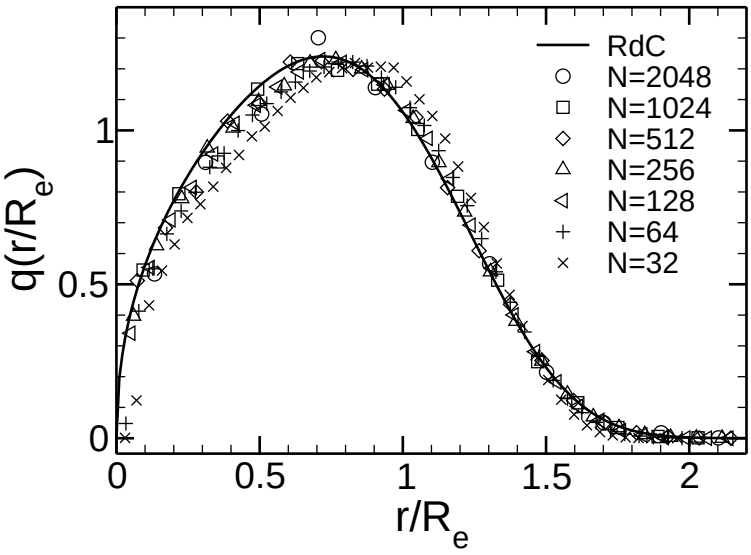

Figure A.4: Distribution of the end-to-end distance at various chainlengths N (symbols) compared with the Redner-des Cloizeaux (RdC) formula $q(r/R_\mathrm{e})$ (see text). The largest chains fulfill the RdC prediction.

$R_\mathrm{e}^2/R_\mathrm{g}^2$ seems to reach the large N value for $N \gtrsim 1024$. When performing averages over configurations it is important to weigh the configurations with the number of MC steps during which they rested unchanged (in other words, to always use the configuration at the present MC step, regardless of acceptance). Otherwise small systematic deviations are introduced in the data, cf. Ref. [36, Sect. 3.4].

Furthermore, there is a renormalization group prediction for the end-to-end distance distribution by Redner and des Cloizeaux (RdC), namely [166]

$$q(\tilde{r} = r/R_\mathrm{e}) = K_1 \tilde{r}^\theta \exp\left[-(K_2\tilde{r})^t\right] \ , \tag{A.3}$$

where the constants K_1 and K_2 are given by the following expressions

$$K_1 = t\,\frac{\Gamma^{(\theta+d)/2}[(\theta+d+2)/t]}{\Gamma^{(\theta+d+2)/2}[(\theta+d)/t]} \ , \quad K_2^2 = \frac{\Gamma[(\theta+d+2)/t]}{\Gamma[(\theta+d)/t]} \ , \tag{A.4}$$

with $d = 2$, $t(d=2) = 4$, $\theta(d=2) = 11/24$, and Γ denoting the gamma-function.

We observe in Fig. A.4 that the RdC prediction is reached for our longest chains. A Gaussian distribution would have a maximum at distance zero, and the RdC formula states that on the contrary, the probability for distance zero vanishes and the maximum is located at a value in the vicinity of the averaged end-to-end distance R_e.

These findings can be regarded as a test for the theoretical predictions and vice versa validate our implementation of the algorithm.

ρ_m	0 N_{recoil}, N_{choice}	1/8 N_{recoil}, N_{choice}	2/8 N_{recoil}, N_{choice}	3/8 N_{recoil}, N_{choice}	4/8 N_{recoil}, N_{choice}
$N = 32$	16, 1.95	20, 1.94	20, 2.08	12, 2.51	14, 3.4
$N = 64$	32, 1.94	32, 1.96	26, 2.08	20, 2.44	8, 5.5
$N = 128$	64, 1.86	64, 1.93	64, 2.04	40, 3.3	–
$N = 256$	128, 1.86	128, 1.90	–	–	–
$N = 512$	256, 1.845	–	–	–	–
$N = 1024$	512, 1.841	–	–	–	–
$N = 2048$	1024, 1.838	–	–	–	–

Table A.1: Optimal parameters for the RG-algorithm for the KREMER-GREST model maximizing the acceptance rate. $\rho_m = 0$ is used to denote the single chain.

A.2 Dense Systems

The RG-algorithm was successfully used to initialize configurations up to a number density nN/L^2 of $\approx 1/2$ in equilibrium (L being the linear dimension of the square simulation box). For high densities and long chains the acceptance rate was becoming to be so low that the LJ-radius of the monomers were shrunk to achieve a lower effective density and equilibration was done using molecular dynamics (MD) simulation methods as described in section 2.6. The optimal parameters for optimizing P_{acc} are listed in Table A.2. The CPU-time corrected optimum P_{acc}/t_{CPU} is very close to the optimum P_{acc}, and is thus not listed additionally.

With this combination of RG and MD we were able to initialize and equilibrate systems with polymers of length $N = 512$ up to very high number densities ρ_m approaching unity. As an example of such a system we present in Fig. A.5 a system with $N = 256$, $n = 24$, and $\rho_m = 7/8$ which was set up using RG at reduced LJ$-\sigma$ and subsequently equilibrated by MD runs. Some chains are highlighted and one identifies a variety of different shapes.

Investigations of the statics and dynamics of the 2-d films are currently underway. Note that in the MD-simulation the bond-length fluctuates around r_{bond} at finite temperature, but is fixed in the MC simulations for efficiency. This has to be taken into account by rescaling the data for fixed bonds by the BOLTZMANN-weight averaged bond-length to match the simulation data. Because the relaxation times grows with N and ρ_m the equilibration of the systems becomes very CPU-intensive for long chains and high densities.

A variety of interesting questions can be discussed once simulation data is ready. To name some concerning the statics which are immediately accessible:

- Shape of the polymers by analyzing the gyration tensor [167].
- Exponents for R_g and R_e as a function of density ρ_m and N.
- Pressure as a function of N and ρ_m.
- Static structure analysis like in section 3.9.

For the dynamics it would be very interesting to investigate deviations from the ROUSE model [20] which assumes that chains move in a viscous heat bath.

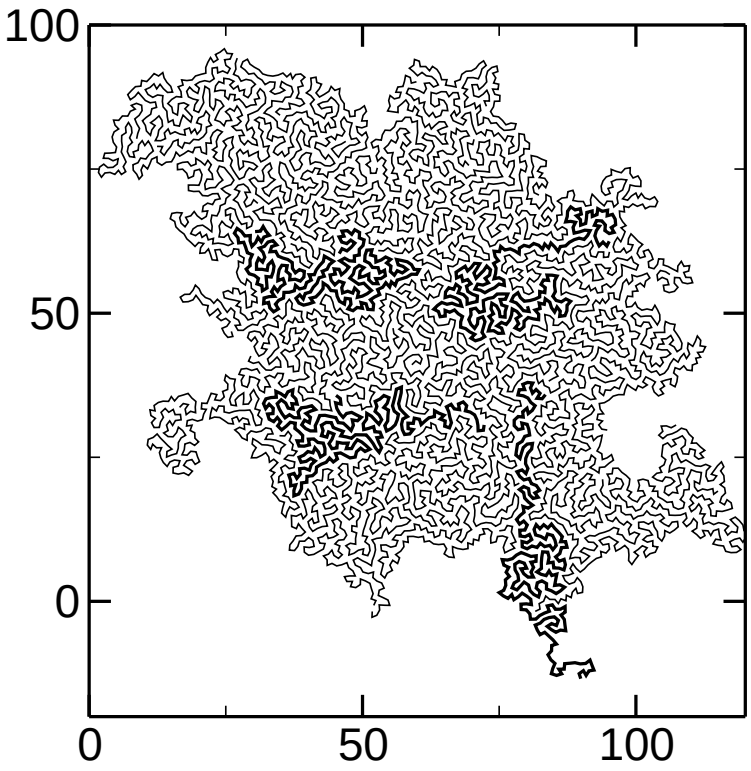

Figure A.5: Example for a dense polymer melt in two dimensions. $N = 256$, number density $\rho_{\mathrm{m}} = 7/8$ at $T = 1.0$. The simulated system is comprised of $n = 24$ chains in a simulation box of area 83.8^2 leading to a pressure of 4.15. Four chains are highlighted.

Since in 2-d chains cannot interpenetrate as easily as in 3-d, especially for dense systems, deviations should become visible. Such questions were already raised in the pioneering work by Carmesin and Kremer [168] some years ago using the "bond fluctuation model" [2] for shorter chains at lower densities. Other investigations of the dynamics and its temperature dependence could follow.

Appendix B

Clustering in Boundary Lubricated Systems with Long-Range Elastic Wall Interaction

In this chapter we want to demonstrate the effect of including long-range elastic interactions in the walls which confine a sub-monolayer of lubricant. It can be summarized as follows. Due to long range elastic coupling, a wall atom which is shifted upwards by a fluid atom will push its neighbors upwards, too. This will create more room and a pocket forms which becomes gradually filled by additional fluid atoms. This can be seen in Fig. B.1, where a half monolayer is confined between two incommensurate walls. While at the start of the simulation the fluid atoms are almost evenly distributed, they collect in one large cluster at the end of the simulation. The TOMLINSON spring constant is $k_{\mathrm{T}} = 50$ (fixing the wall atoms to their equilibrium sites) and the FRENKEL-KONTOROVA stiffness $k_{\mathrm{FK}} = 58.3$ (coupling the relative neighbor distance to the equilibrium distance) as defined in section 2.4.2. The interactions are purely repulsive ($r_{\mathrm{cut}} = 2^{1/6}$) between all particle species, temperature $T = 0.7$, and load per wall atom $L = 10$ in reduced LJ-units. As all particle interactions are repulsive, the clustering must be a pure elasticity effect.

This clustering does not happen for walls with fixed wall atoms, neither for walls without FRENKEL-KONTOROVA coupling. Our model shows effects similar to the ones discussed in Refs. [43, 44] using a similar, more detailed wall model. In Fig. B.1 the cluster spans the whole simulation box. For lower covering, we observed small pockets of trapped liquid, as observed in experiments [169].

We additionally found

- Shear facilitates clustering, as it helps to sweep fluid particles into the pockets.
- Clustering is also present for commensurate systems where clusters have a crystalline structure.
- Short chain molecules $N \leq 10$ show very similar behavior.

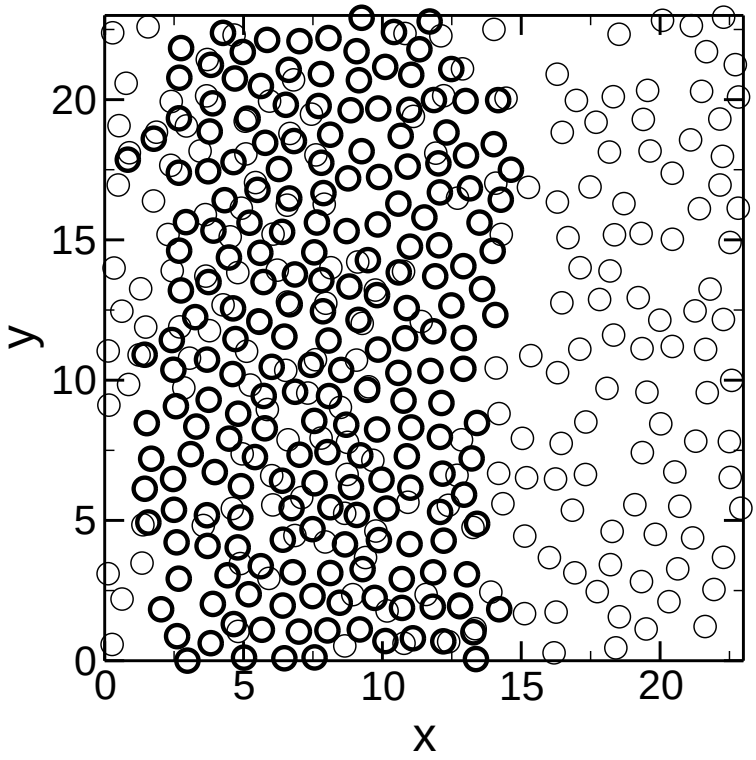

Figure B.1: Clustering of a half-layer of simple liquid confined between two incommensurate walls (defining the xy-plane) with long-range elastic coupling and purely repulsive interaction. $T = 0.7$, $L = 10$ in reduced units. Periodic boundary conditions are used in x and y. Light circles denote particle positions in the start configuration, the bold circles in a configuration in steady state while shearing with a velocity $v_0 = 3 \times 10^{-4}$.

It would be very interesting to extend these simulations to longer chains to see if a network is formed by chains with ends in two different clusters or if the behavior remains simple liquid like.

Appendix C

Simulation Code

In this chapter we list the code for the complete MD-simulation program used for the simulation of friction between interfaces. The analysis routines for the computation of the fluid particle velocity distribution is also listed.

We first print the parameter files for the simulation. `params_rheosim` controls the length of the simulation, system parameters and the simulation mode. The file `params_mc_fluid` is only needed when the recoil-growth algorithm is used. Most other system parameters like particle numbers and the order of the integration algorithm are set at compile time in the file holding the global variables. The Makefile explains the dependencies of the different program modules. We list the code in a very tiny font (so, for an intensive study of a particular routine we recommend a magnification).

params_rheosim

```
           0 initial integer starting time
           0 number of relaxation steps
      100000 number of observation steps
    10000000 number of steps between full configuration backup
         200 time constant for filtering output
       50000 linear position saving timestep
           T switch for writing particle positions
           F switch for writing particle velocities
           F save at times given in list
sample_list

  0.500000E-02 time step increment

  0.500000E+01 time to reach final temperature
  0.100000E+01 initial temperature
  0.100000E+01 final temperature

  0.500000E+02 time to reach real potential
  0.900000E+00 squared initial minimum effective distance between particles

  0.100000E+01 mass                (type 1)
  0.100000E+01 Lenard Jones epsilon (type 1)
  0.100000E+01 Lenard Jones sigma   (type 1)
  0.500000E+00 friction constant    (type 1)
  0.100000E+01 mass                (type 2)
  0.100000E+01 Lenard Jones epsilon (type 2)
  0.100000E+01 Lenard Jones sigma   (type 2)
  0.500000E+00 friction constant    (type 2)

     20030409 seed for random number generator
```

```
  0.600000E+00 skin thickness

6.40000000E+01 spacing between wall units in x direction
             F switch: walls are commensurate(T) / incommensurate(F)
  2.000000E+00 initial inter-wall spacing
  0.400000E+03 Tomlinson spring constant to wall equililibrium sites
  0.000000E+00 Linear Frenkel-Kontorova spring constant between wall particles
  0.000000E+00 Vector Frenkel-Kontorova spring constant between wall particles

             1 flag for constraining wall atoms
             0 cut-off flag: (0) short (1) long range potential
             7 flag for thermostat mode
             0 flag to relax to next minimum (will reset friction constants)
             0 flag for turning on friction force on top wall
  0.050000E+00 friction constant for friction on top wall

 flags and variables controlling boundary conditions:
   1-direction   2-direction   3-direction
             2             2             2
  0.000000E+00  0.000000E+00  0.000000E+00
  0.000000E+00  0.000000E+00  0.000000E+00

             F switch l_compute_fluid_distribution
             F switch l_compute_fluid_vel_dist
             T switch l_compute_press_tens
```

params_mc_fluid

```
             0 initial MC step
             0 number of MC relaxation steps
          1000 number of MC observation steps
      10000000 number of MC steps between full configuration storage
          1000 linear position saving timestep
             T switch for writing particle positions
             F switch for saving at times given in list
sample_list_mc

         10000 number of tries to build a complete chain
            64 depth of recoiling
       5000000 maximal tree search depth
  1.950000E+00 number of trial directions at first monomer
  1.950000E+00 number of trial directions at last monomer
  1.800000E+02 opening angle for bonds (degrees)
  0.000000E+00 minimum initial wall distance in z
  1.000000E+00 prefactor for rescaling bond-length
  1.000000E+00 initial LJ-sigma of fluid

             T switch for writing energy
             T switch for writing gyration tensor and end-to-end distance
             T switch for writing positions of the chain ends
             T switch for writing bond angle information
```

Makefile

```
# makefile for the rheology simulation program
# Martin Aichele, 2000-11-23
# last modified 2001-12-11
# select identifier for compiler
# Intel, NAG, Compaq, Absoft, PGI, G95
IDENT=Intel
# debugging on? (YES/NO)
# The Intel debugger is 'ldb'
DEBUGGING=NO
# C-preprocessor command, in cpp, the -traditional is neccessary for Fortran
PREPRO=cpp -P -traditional -pedantic-errors
# filename of pre-processed file globals$(VER).f90
PP_GLOBALS=PP_globals.f90
# Version numbering.  New versions can be introduced by 'make versionmove'
OLDVER=V1.8
# general version
VER=V1.9
# mcfluid version
VERMCF=V1.91
################################################################################
# Absoft f90
################################################################################
ifeq ($(IDENT),Absoft)
FC=f90
# the linker
LINKER=ld
# linker options
#LINKOPS=-v -V
# -m0: lots of warnings,..., -m4: no warnings
WARNOPS=-m1
ifeq ($(DEBUGGING),YES)
DEBUG=-g
endif
endif
################################################################################
# Compaq f90 (on Alphas)
################################################################################
# use the gnumake command instead of make o nthe ZDV Alpha cluster
# otherwise errors will result
ifeq ($(IDENT),Compaq)
FC=f90
# the linker
LINKER=ld
# linker options
#LINKOPS=-v -V
# -std: warn about non-standard F90
WARNOPS=-warn general -warn unused -std
#CHECKS=-check bounds -check overflow -check underflow
#FLAGS=-math_library accurate # accurate math
FLAGS=-math_library fast     # fast math
#OPTOPS=-O0
#OPTOPS=-O2
#OPTOPS=-O3
```

```
OPTOPS=-O4 -inline speed  # Compaq f90
ifeq ($(DEBUGGING),YES)
DEBUG=-g
endif
endif
#############################################################################
# Intel f90
#############################################################################
ifeq ($(IDENT),Intel)
FC=ifc
# the linker
LINKER=ld
# -e90: issue warnings for non F90 standard compliant code
WARNOPS=-e90
# some of the checks could still be somewhat broken (check documentation)
#CHECKS=-C -d1
#CHECKS=-WB -CB -d1
# CHECKS=-WB # warn about overrun array bounds at compile time
# -nbs: treat backslash as character
#FLAGS=-nbs -opt_report -opt_report_levelmin
FLAGS=-nbs
# -ip: interprocedural optimization
# -prefetch: cache data prefetching
# -xK: code for Pentium III and higher
# -axX: code optimized for processor type X and higher, but runs on every i386
# -rcd: fast float to int conversion (does not make code faster)
# -tpp6: code for Pentium 6 family
#OPTOPS=-O0
#OPTOPS=-O2
# seems to give the fastest code on PIII
# ifc V6.0 with -O3 gives wrong results
# ICS, ifc V6.0
OPTOPS=-O2 -tpp6 -axK -ipo
# KOMA, ifc V7.0
# using -(a)xM causes incorrect results on Athlons
# using -axK runs okay, but is slightly slower on Athlons
# -prefetch requires -O3 and does not improve performance
#OPTOPS=-O2 -tpp6 -ipo
#OPTOPS=-prof_gen -O1
# Xeon P4
# -xW caused a bus error after a short run time.
#OPTOPS=-O2 -tpp7 -axW -ipo
#OPTOPS=-O2 -xK -ip -prefetch
#OPTOPS=-O2 -ip
# linker options
#LINKOPS=-v -V
LINKOPS=$(OPTOPS) $(CHECKS) $(FLAGS) -static #-prof_gen
ifeq ($(DEBUGGING),YES)
DEBUG=-g
endif
endif
#############################################################################
# NAG f95
#############################################################################
ifeq ($(IDENT),NAG)
FC=f95
# the linker
LINKER=ld
# linker options
#LINKOPS=-v -V
CHECKS=-C=all
# -kind=byte: kind numbering is done by bytes
# -nan: initialize variables to NaN to cause run-time error if these variables
#       are used before initializing
# -float-store: (Gnu C based systems only) Do not store floating-point
#               variables in registers on machines with floating-point
#               registers wider than 64 bits.  This can avoid problems with
#               excess precision.
#            My experience is that this option makes the code slower on IA32!
FLAGS=-kind=byte -nan #-float-store
# -gc: garbage collection
#OPTOPS=-O0
#OPTOPS=-O1
OPTOPS=-O3 # fastest
#OPTOPS=-O3 -gc
#OPTOPS=-O4 # not faster than -O3
ifeq ($(DEBUGGING), YES)
DEBUG=-gline #show traceback upon run-time error
endif
endif

#############################################################################
# GNU g95
#############################################################################
ifeq ($(IDENT),G95)
FC=g95
# the linker
LINKER=ld
# linker options
#LINKOPS=-v -V
WARNOPS=-Wline-truncation
OPS=-ffree-form
ifeq ($(DEBUGGING),YES)
DEBUG=-g
endif
endif
#############################################################################
#############################################################################
# Options for compiler
ifeq ($(DEBUGGING),YES)
OPS=$(WARNOPS) $(CHECKS) $(FLAGS) $(DEBUG)
else
OPS=$(WARNOPS) $(CHECKS) $(FLAGS) $(OPTOPS)
endif
#############################################################################
#############################################################################

all : rheosim
.phony : clean
versionmove : globals$(OLDVER).f90 interaction$(OLDVER).f90 \
mdroutines$(OLDVER).f90 initialization$(OLDVER).f90 \
rheosim$(OLDVER).f90 ftdensity$(OLDVER).f90 \
velautocorr$(OLDVER).f90 fluid_density$(OLDVER).f90 \
popping$(OLDVER).f90 mcfluid$(OLDVER).f90
cp -i globals$(OLDVER).f90        globals$(VER).f90
cp -i interaction$(OLDVER).f90    interaction$(VER).f90
cp -i initialization$(OLDVER).f90 initialization$(VER).f90
cp -i mdroutines$(OLDVER).f90     mdroutines$(VER).f90
cp -i rheosim$(OLDVER).f90        rheosim$(VER).f90
cp -i ftdensity$(OLDVER).f90      ftdensity$(VER).f90
cp -i velautocorr$(OLDVER).f90    velautocorr$(VER).f90
cp -i fluid_density$(OLDVER).f90  fluid_density$(VER).f90
cp -i popping$(OLDVER).f90        popping$(VER).f90
cp -i utilities$(OLDVER).f90      utilities$(VER).f90
cp -i mcfluid$(OLDVER).f90        mcfluid$(VERMCF).f90
cp -i polymer$(OLDVER).f90        polymer$(VER).f90
luxury.o : luxury.f90
$(FC) $(OPS) -c luxury.f90 -o luxury.o
globals.o : globals$(VER).f90
rm -f $(PP_GLOBALS); \
$(PREPRO) globals$(VER).f90 -o $(PP_GLOBALS); \
$(FC) $(OPS) -c  $(PP_GLOBALS) -o globals.o
interaction.o : globals.o utilities.o interaction$(VER).f90
$(FC) $(OPS) -c interaction$(VER).f90 -o interaction.o
polymer.o : globals.o polymer$(VER).f90
$(FC) $(OPS) -c polymer$(VER).f90 -o polymer.o
mcfluid.o : globals.o luxury.o utilities.o polymer.o mcfluid$(VERMCF).f90
$(FC) $(OPS) -c mcfluid$(VERMCF).f90 -o mcfluid.o
init.o : globals.o luxury.o mdroutines.o mcfluid.o utilities.o \
  initialization$(VER).f90
$(FC) $(OPS) -c initialization$(VER).f90 -o init.o
utilities.o : globals.o luxury.o utilities$(VER).f90
$(FC) $(OPS) -c utilities$(VER).f90 -o utilities.o
mdroutines.o : globals.o luxury.o utilities.o mdroutines$(VER).f90
$(FC) $(OPS) -c mdroutines$(VER).f90 -o mdroutines.o
# there is a problem with this module used for big systems, as
# it allocates huge amounts of memory
ftdensity.o : globals.o ftdensity$(VER).f90
$(FC) $(OPS) -c ftdensity$(VER).f90 -o ftdensity.o
vacf.o : globals.o luxury.o velautocorr$(VER).f90
$(FC) $(OPS) -c velautocorr$(VER).f90 -o vacf.o
fluiddens.o : globals.o fluid_density$(VER).f90
$(FC) $(OPS) -c fluid_density$(VER).f90 -o fluiddens.o
popping.o : globals.o popping$(VER).f90
$(FC) $(OPS) -c popping$(VER).f90 -o popping.o
rheosim.o : globals.o popping.o rheosim$(VER).f90
$(FC) $(OPS) -c rheosim$(VER).f90 -o rheosim.o
# include ftdensity.o if needed only.
rheosim : luxury.o globals.o interaction.o mdroutines.o \
init.o vacf.o fluiddens.o utilities.o popping.o mcfluid.o \
polymer.o rheosim.o
$(FC) $(LINKOPS) luxury.o globals.o \
interaction.o mdroutines.o init.o vacf.o fluiddens.o \
utilities.o popping.o mcfluid.o polymer.o \
rheosim.o -o rheosim$(VER).exe
clean  :
rm -f $(PP_GLOBALS) *.o *.vo *.mod *.d *.bif V*.inc *.il \
V*.f work.p* a.out core ifc??????
```

Main program rheosimV1.9.f90

```
! rheosimVx.y.f90
! Molecular dynamics simulation of liquids and polymers confined between two
! walls.  Uses predictor - corrector algorithm up to fifth order.
! Based on mfa_prog.f by Dr. Martin H. Mueser.
! Martin Aichele, 2000-11-23
! V1.8 last changed 2002-03-26
! V1.9 last changed 2003-02-17
! Note: cpu_time() is Fortran 95
!#############################################################################
program rheosim
  ! use the variables declared in globals in this scope
  use globals
  ! routines for initialization of the simulation
  use initialization
  ! routines for particle interactions
  use interaction
  ! the core md routines
  use mdroutines
  ! module for the velocity autocorrelation function
  use vacf
  ! module for the fluid density distribution
  use fluid_density
  ! module for pops
  use popping
  ! module for evaluating polymer quantities
  use polymer
  ! no implicit variables
  implicit none
  !----------------------------------------------------------------------------
  ! debug switches      |-------- 31 characters ------|
  !----------------------------------------------------------------------------
  logical, parameter :: l_debug_main                    = .FALSE.
  !----------------------------------------------------------------------------
  !#############################################################################
  ! get the CPU-time at the start
  if(l_cpu_timing) call cpu_time(cpu_time_start)
  write(unit=*, fmt='(2a)') "MESSAGE: This is ", program_name
  write(unit=*, fmt='(a,i1,a)') "MESSAGE: We are simulating a ", n_dim, &
       "-d system"
  ! initialize global arrays
  call init_arrays
  ! try to read parameters
  ! if a parameter file exists, read it
  ! else use default parameters
  call init_parameters("params_rheosim")
  ! initialize random number generator which might be needed in case a
  ! configuration has to be generated, not only for the thermostat.
  call rluxgo(rngQualityLevel, iseed, 0, 0)
  ! call rluxat(level, iseed, k1, k2) gets the values of the variables to
  ! restart the random number generator where it stopped
  ! write out parameters
  call write_parameters("params_rheosim_new")
  ! Read configuration if it exists, otherwise create configuration
  ! The random number generator is restarted from an old configuration
  call init_config("conf_old"//version)
  ! write out information about the simulated system
  call write_config_infos("config_info"//version)
  ! find the minimal distance between two particles
  call min_distance
  ! initialize observation variables
  call init_variables
  ! read sample list (the times are needed to initialize the next time to write
  ! and to check if t he simulation is long enough)
  if(l_read_sample_list) call read_sample_list(file_sample_list, sample_times,&
       s_time, n_relax, n_obser)
  ! initialize energy logging
  if(l_write_energy_every_mds) &
       call init_energy_file("energy_log"//version//".dat")
  if(l_compute_fluid_distribution) then
```

```
   ! initialize fluid density histograms
   if(n_mon_tot .gt. 0) call init_density_histogram(5, 5)
  end if
  ! this is the step before the main loop.  It is included to write out the
  ! starting configuration of the simulation like at other times.
  i_time = s_time
  call pos_and_vel_out
  !###################  M A I N   L O O P  ################################
  write(unit=*, fmt='(a)') "MESSAGE: Entering main MD loop"
  ! s_time is the number of steps already completed, so the loop starts at the
  ! next step, thus `+1'
  do i_time = s_time +1, n_relax + n_obser
    if(l_debug_main) call simulationstate
    r_time = i_time * dt ! the model time
    ! propagate coordinates and its derivatives one time unit
    call predict
    ! if there is at least one particle which has moved farther then half the
    ! skin we have to create new neighborhood lists
    if( .not. check_skin() ) call binning3d
    ! set temperature and effective minimum distance, reinitialize forces
    call thermostat
    ! compute new forces with predicted positions and derivatives
    if(l_fluid_fluid_interaction) call fluid_fluid
    if(l_fluid_wall_interaction)  call fluid_wall
    if(l_allow_wall_wall_interaction) then
      ! in the binning routine we tested if there was possible wall-wall
      ! interaction and set l_wall_wall_interaction accordingly
      if(l_wall_wall_interaction) then
        call wall_wall
      else ! we reset v_wall_wall here
        v_wall_wall = 0.0_dp
      end if
    end if
    if(l_intra_molec_interaction) call intra_molec
    if(l_intra_wall_interaction)  call intra_wall
    ! correct positions and derivatives proportional to deviation between
    ! predicted and real acceleration
    call corrector
    !=========================================================================
    ! progagation complete
    !=========================================================================
    ! write out a short message for every 10% of the simulation
    if((i_time.gt.s_time) .and. (n_relax + n_obser - s_time .gt. 10) .and. &
         mod(i_time - s_time, (n_relax + n_obser - s_time)/10) .eq. 0) &
         write(unit=*, fmt='(a, i12)') "MESSAGE: Finished MDS", i_time
    ! check if particles have penetrated a wall
    if(n_dim .eq. n_dim_max+1 .and. l_check_wall_penetration) &
         call wall_penetration
    ! check for presence of bond crossings in xy-plane.
    if(l_forbid_xy_bond_crossings .and. n_dim_pbc .eq. 2) then
      if((i_time.gt.s_time) .and. &
           (mod(i_time, n_linear_out) .eq. 0)) then
        write(unit=*, fmt='(a, i12)') "MESSAGE: &
             &Checking for xy-bond crossings at MDS", i_time
        if(.not. check_xy_bond_crossings_all()) then
          write(unit=*, fmt='(a, i12)') &
               "ERROR: xy-plane bond crossings detected at MDS", i_time
          ! for some reason SR fluid_positions_out can't be called
          call particle_positions_out("xy-bonds-cross_t", i_time, &
               1, n_mon_tot, .FALSE.)
          stop
        end if
      end if
    end if
    ! low pass filtering of output and control of force ramp
    call control
    ! compute energies (and write out every MDS if l_write_energy_every_mds is
    ! set to .TRUE.)
    call energy("energy_log"//version//".dat")
    if(l_compute_wall_wall_quantities) call wall_wall_quantities
    ! calculate gyration tensor
    if(n_mon .gt. 1) call gyration_tensor(n_time_ave)
    ! calculate pressure tensor
    if(l_compute_press_tens) call pressure_tensor(n_time_ave)
    ! writing of positions and velocities
    call pos_and_vel_out
    ! observe system after equilibration
    if(i_time.ge.n_relax) then
      ! HACK
      if(.FALSE. .and. (mod(i_time, 100) .eq. 0)) then
        ! write x,y position of first particle
        write(unit=60, fmt='(es12.6, 2f11.3)') r_time, r0_unfolded(1, 1:2)
      end if
      if(l_compute_vacf) call velocity_autocorr
      if(l_compute_fluid_distribution) &
           call write_dens_histogram("fl_dens_histogram"//version//".dat", &
           n_time_ave, "var_only")
      if(l_compute_fluid_displ_dist) &
           call fluid_displ_p_dist("fl_displ_pdist_"//version//".dat", &
           n_time_ave, "moments_only")
      if(l_compute_fluid_vel_dist) then
        call fluid_vel_p_dist("fl_vel_pdist_t_"//version//".dat", &
             n_time_ave, "ave_prob_dist")
        call fluid_vel_xy_pl_pdist("fl_vel_xy_pl_pdist_t_"//version//".dat", &
             n_time_ave, "ave_prob_dist")
        call fluid_vel_xyz_p_dist("fl_vel_x_pdist_t_"//version//".dat", &
             "fl_vel_y_pdist_t_"//version//".dat", &
             "fl_vel_z_pdist_t_"//version//".dat", n_time_ave, "ave_prob_dist")
        if(n_mon .gt. 1) then
          call bond_ang_vel_p_dist("bond_ang_vel_pdist_t_"//version//".dat", &
               n_time_ave, "ave_prob_dist")
        end if
      end if
      if(l_compute_fluid_accel_dist) &
           call fluid_accel_p_dist("fl_accel_pdist_t_"//version//".dat", &
           n_time_ave, "moments_only")
    end if !(i_time .ge. n_relax)
    ! write out safety copy every n_save MDS
    if((i_time.gt.s_time) .and. (mod(i_time, n_save) .eq. 0)) then
      if(l_use_config_rng) then
        call store_configuration_rng("last_save_file", .TRUE.)
      else
        call store_configuration("last_save_file")
      end if
    end if
    ! check if CPU time limit reached.  If so, exit MD loop
    if(l_cpu_timing) then
      if(mod(i_time, n_call_cpu_time) .eq. 0) then
        call cpu_time(cpu_time_now)
        if((cpu_time_now - cpu_time_start) .ge. cpu_time_available) then
          write(unit=*, fmt='(a, f16.1, 2a, i12)') &
               "MESSAGE: CPU time limit=", cpu_time_available, &
               " sec reached at", " i_time=", i_time
          l_cpu_time_limit_reached = .TRUE.
          exit
        end if
      end if !(mod(i_time, n_call_cpu_time) .eq. 0)
    end if ! l_cpu_timing
  end do
  write(unit=*, fmt='(a)') ""
  !###################   E N D   M A I N   L O O P   ######################
  if(l_cpu_time_limit_reached) then
    n_mds_after_relax = i_time - n_relax - s_time
    ! write new parameter file for continuation run
    s_time = i_time
    call write_parameters("params_rheosim"//version//"_new")
  else ! main loop completed
    ! we set i_time back by one if the main loop was completed,
    ! because i_time is incremented by one after the do-loop
    i_time = n_relax + n_obser
    n_mds_after_relax = n_obser - s_time
  end if
  ! write out final configuration
  if(l_use_config_rng) then
    call store_configuration_rng("conf_new"//version, .TRUE.)
  else
    call store_configuration("conf_new"//version)
  end if
  ! write out observables at the end of the simulation
  if(n_mds_after_relax.gt.0) then
    call simulation_log_out("simulation_log"//version//".dat")
    if(l_compute_vacf) call write_vacf("vacf"//version//".dat")
    if(l_compute_fluid_vel_dist) then
      ! Note: it is necessary to pass the histogram with the (:), otherwise
      !       the program crashes.  (Compiler bug?)
      call p_dist_write_out("fl_vel_pdist_"//version//".dat", &
           "fluid velocity probability distribution", &
           "bin-center | probability", fluid_vel_p_dist_ave(:), &
           fluid_vel_bin_start, fluid_vel_bin_end, &
           fluid_vel_bin_width)
      call p_dist_write_out("fl_vel_xy_pl_pdist_"//version//".dat", &
           "fluid velocity in xy-plane probability distribution", &
           "bin-center | probability", fluid_vel_xy_pl_pdist_ave(:), &
           fluid_vel_xy_pl_bin_start, fluid_vel_xy_pl_bin_end, &
           fluid_vel_xy_pl_bin_width)
      call p_dist_write_out("fl_vel_x_pdist_"//version//".dat", &
           "fluid velocity probability distribution in x", &
           "bin-center | probability", fluid_vel_x_p_dist_ave(:), &
           fluid_vel_x_bin_start, fluid_vel_x_bin_end, &
           fluid_vel_x_bin_width)
      call p_dist_write_out("fl_vel_y_pdist_"//version//".dat", &
           "fluid velocity probability distribution in y", &
           "bin-center | probability", fluid_vel_y_p_dist_ave(:), &
           fluid_vel_y_bin_start, fluid_vel_y_bin_end, &
           fluid_vel_y_bin_width)
      call p_dist_write_out("fl_vel_z_pdist_"//version//".dat", &
           "fluid velocity probability distribution in y", &
           "bin-center | probability", fluid_vel_z_p_dist_ave(:), &
           fluid_vel_z_bin_start, fluid_vel_z_bin_end, &
           fluid_vel_z_bin_width)
      if(n_mon .gt. 1) then
        call p_dist_write_out("bond_ang_vel_pdist_"//version//".dat", &
             "bond angular velocity probability distribution", &
             "bin-center | probability", bond_ang_vel_p_dist_ave(:), &
             bond_ang_vel_bin_start, bond_ang_vel_bin_end, &
             bond_ang_vel_bin_width)
      end if
    end if
  else
    write(unit=*, fmt='(a, i12, a)') &
         "MESSAGE: n_mds_after_relax=", n_mds_after_relax, &
         " .le. 0, so simulation log not written"
  end if
  ! close output channels used (it seems that this reduces the occurence of
  ! incompletely written last records)
  close(41)
  close(42)
  close(50)
  close(51)
  close(52)
  ! free memory.  The freeing functions check if the memory was really
  ! allocated before deallocating
  call free_bin_memory
  call free_dens_bin_memory
  write(unit=*, fmt='(3a)') "MESSAGE: ", program_name, " says goodbye."
  stop
  !=============================================================================
end program rheosim
!=============================================================================
```

globalsV1.9.f90

```
! global variables for rheosimVx.y
!-------------------------------------------------------------------------------
! Rheosim is a simulation program for doing molecular dynamics simulations of
! simple or polymeric liquids confined between to walls in 1-d and 2-d.
! Walls can be totally rigid, coupled harmonically to their equililibrium sites
! (Tomlinson walls) and neighboring wall atoms can be coupled harmonically to
! introduce long range elastic coupling in the walls
! (Frenkel-Kontorova-Tomlinson walls).
! The program is based on the program mfa.f by Dr. Martin H. Mueser.
! Developed by Martin Aichele, starting June 2000.
!-------------------------------------------------------------------------------
! most program parameters which are not read from a file should be changed here
! Martin Aichele, 2000-11-27
! last changed 2003-02-25
!-------------------------------------------------------------------------------
! references are given as
! * M/D :
!   Michael Metcalf and John Reid,
!   Fortran 90/95 explained, 2nd edition
!   Oxford University Press, 1999
! * E/P/L :
!   T. M. R. Ellis, Ivor R. Phillips, Thomas M. Lahey,
!   Fortran 90 Programming
!   Addison-Wesley, 1994 (reprint 1998)
!-------------------------------------------------------------------------------
! * F90 does not initialize variables to anything by default for efficiency
! * a word for the C programmer: initializing a variable in the declaration
!   gives the variable the SAVE attribute, which might not be what you want,
!   because the variable is *not* re-initialized upon each call.
! Note on initialization: Initializing large arrays during compiling needs a
! lot of memory and is very slow.  These initializations are done in
! SR init_arrays.
```

```
! Because C preprocessing is not standard in Fortran, we call the
! C-preprocessor before compiling.  (See makefile)
module globals
  !---------------------------------------------------------------------------
  ! conventions about variables:
  !---------------------------------------------------------------------------
  ! # no variables with just one letter
  !   (hard to find in editor and hard to replace as well)
  !   an identifier can have 31 characters in F90
  ! # variables starting with f_ :
  !   flags
  ! # variables starting with l_ :
  !   logical values (mainly used as on/off switches)
  ! # variables starting with n_ :
  !   dimensions of arrays and lengths of loops
  ! # variables ending with: _1 _2 _3 _4
  !   1., 2., ... power or cumulant of a variable (cum grano salis)
  ! no implicit declaration of variables
  implicit none
  ! because this module is USEd in the main program, the SAVE statement is
  ! not necessary, see E/P/L, p. 363 (but it won't hurt and is good style)
  save
  !---------------------------------------------------------------------------
  ! Version numbering, program name, dimensions
  !---------------------------------------------------------------------------
  ! program version
  character(len=*), parameter :: version = "V1.9"
  ! compile date (useful for comparing program results when code was changed)
  ! I love the C-preprocessor :) )
  character(len=*), parameter :: date = __DATE__
  ! program name
  character(len=*), parameter :: &
       program_name="rheosim"//version//" ("//date//")"
  ! maximal number of dimensions (for allocation of arrays)
  ! never change from 3.
  integer, parameter :: n_dim_max = 3
  ! dimension of the system
  integer, parameter :: n_dim = 2
  ! number of dimensions with periodic boundary conditions
  integer, parameter :: n_dim_pbc = 2
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! KINDS of variables
  !---------------------------------------------------------------------------
  ! on Intel 32 bit processors (Pentiums) the default real has only 6 digit
  ! PRECISION, which is not enough.  That's why we inquire the KIND of real
  ! giving at least 15 digits PRECISION.  On most machines this is a 64 bit
  ! floating point number and corresponds to DOUBLE PRECISION
  integer, parameter :: dp = SELECTED_REAL_KIND(15, 150)
  ! when assigning constants to these reals, it it important that they have
  ! the suffix "_dp", otherwise the constant is converted to default real and
  ! only after that assigned to the variable, thus precision is lost.
  ! Some compilers provide a switch to do that, but it is not standart behavior
  ! the random number generator requires special KINDs
  ! make sure this KIND is the same as in the random number generator module
  integer, parameter :: rngRealKind = 4
  !---------------------------------------------------------------------------
  ! constants
  !---------------------------------------------------------------------------
  ! constants (it is permitted to give more digits than are actually used, cf.
  ! M/R, page 16), some compilers will issue warnings which can be ignored.
  real(kind=dp), parameter :: sqrt2  = 1.41421356237309504880_dp
  real(kind=dp), parameter :: sqrt3  = 1.73205080756887729352_dp
  real(kind=dp), parameter :: pi     = 3.14159265358979323846_dp
  real(kind=dp), parameter :: twopi  = 6.28318530717958647692_dp
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! switches for checks performed during simulation.  They do not increase CPU
  ! time significantly.
  !---------------------------------------------------------------------------
  logical, parameter :: l_predict_check_pbc             = .FALSE.
  logical, parameter :: l_binning3d_check_range         = .TRUE.
  logical, parameter :: l_binning3d_check_bin_overrun   = .TRUE.
  logical, parameter :: l_binning3d_check_number        = .FALSE.
  logical, parameter :: l_binning3d_check_list_overrun  = .TRUE.
  logical, parameter :: l_check_wall_penetration        = .FALSE.
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! switches for analysis computations
  !---------------------------------------------------------------------------
  ! velocity autocorrelation function
  logical, parameter :: l_compute_vacf                  = .FALSE.
  ! Fourier transforms of the particle density
  logical, parameter :: l_compute_ft_density            = .FALSE.
  ! histograms of fluid particle density
  logical            :: l_compute_fluid_distribution    = .FALSE.
  ! wall - wall quantities (potential, correlations, ...)
  logical, parameter :: l_compute_wall_wall_quantities  = .FALSE.
  ! the probability distribution of the fluid particles' displacements
  logical, parameter :: l_compute_fluid_displ_dist      = .FALSE.
  ! the probability distribution of the fluid particles' velocities
  logical            :: l_compute_fluid_vel_dist        = .FALSE.
  ! the probability distribution of the fluid particles' accelerations
  logical, parameter :: l_compute_fluid_accel_dist      = .FALSE.
  ! the pressure tensor
  logical            :: l_compute_press_tens            = .TRUE.
  ! if the wall--wall and intra-wall contributions to the
  ! pressure tensor should be included
  logical, parameter :: l_compute_press_tens_wall_contr = .FALSE.
  ! center of mass of fluid
  logical, parameter :: l_compute_cm_fluid              = .FALSE.
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! file writing and file format switches
  !---------------------------------------------------------------------------
  ! switch for writing energies to disk at every MDS
  logical, parameter :: l_write_energy_every_mds        = .FALSE.
  ! switch for deciding if the laboratory system is to be used for all
  ! kinetic energy calculations.  Use this to check energy conservation.
  logical, parameter :: l_use_lab_system_for_energies   = .FALSE.
  ! switch for writing the particle positions
  logical            :: l_write_particle_positions      = .FALSE.
  ! switch for writing the particle velocities
  logical            :: l_write_particle_velocities     = .FALSE.
  ! switch for deciding which format the configuration i/o routines should use
  ! (if the state of the random number generator is saved or not)
  ! The format witout random numbers is the format used by M. H. Mueser.
  logical, parameter :: l_use_config_rng                = .TRUE.
  ! if we restart the random number generator from the configuration.
  ! If set to .FALSE. the seed in the parameter file is used.
  ! (meaningless if l_use_config_rng == .FALSE.)
  logical, parameter :: l_read_conf_rng_reinit          = .TRUE.
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! switches for deciding which interaction is turned on.
  !---------------------------------------------------------------------------
  logical, parameter :: l_fluid_fluid_interaction       = .TRUE.
  logical, parameter :: l_fluid_wall_interaction        = .FALSE.
  ! we adjust the interaction between fluid and bottom wall by a factor
  ! bw_fl_ia_adjust if l_bottom_wall_fluid_ia_differ == .TRUE.
  ! Not fully implemented: just changes the interactions SR fluid_wall
  ! It would be better to introduce different particle types for both walls
  logical, parameter :: l_bottom_wall_fluid_ia_differ   = .FALSE.
  real(kind=dp), parameter :: bw_fl_ia_adjust = 1.0_dp
  ! if there is a lot of fluid between the walls or the walls are very far away
  ! then the wall-wall interaction is negligible.  The creation of neighbor
  ! lists of wall-wall pairs is also switched off in this case.
  ! With this switch we can switch off the wall-wall interaction
  ! unconditionally (if set to .FALSE.).  The checks for wall-wall interaction
  ! are not carried out.
  logical, parameter :: l_allow_wall_wall_interaction   = .FALSE.
  ! this switch is reset in SR BINNING3D if there is possible interaction
  logical :: l_wall_wall_interaction                    = .TRUE.
  logical, parameter :: l_intra_wall_interaction        = .FALSE.
  ! if n_mon .eq. 1 then there are no bonds and no intramolecular interaction
  logical, parameter :: l_intra_molec_interaction       = .TRUE.
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! fundamental program parameters
  !---------------------------------------------------------------------------
  ! the quality level for the random number generator.  0: bad but fast,
  ! 4: very good quality but slower.  1 or 2 seem to be a good choices.
  ! Do not use level 0!
  ! By going from level 1 to 2 the program runs about 3% slower.
  integer, parameter :: rngQualityLevel = 2
  ! the order of the algorithm which is actually used
  ! n_order = 2 is the velocity-Verlet algorithm
  integer, parameter :: n_order = 5 ! must be .leq. n_order_max
  ! maximal order of predictor - corrector algorithm (the program has
  ! structures for using order 5 at most)
  integer, parameter :: n_order_max = 5
  ! switch for the method of the minimum image convention calculation in
  ! routines calculating the distance of particles.
  ! On Intel (Pentium and Celeron) systems .FALSE. is about 10% faster.
  ! On Alpha-systems the difference is only small, but .FALSE. is still faster.
  ! The results are slightly different, because there are other operations
  ! performed.
  logical, parameter :: l_mic_use_int_cast = .FALSE.
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! time variables
  !---------------------------------------------------------------------------
  integer       :: i_time = -1  ! time in MDS used throughout the program
  real(kind=dp) :: dt           ! timestep
  real(kind=dp) :: dt_2, dt_inv ! square of timestep and inverse
  real(kind=dp) :: r_time       ! r_time = i_time * dt
  integer       :: s_time       ! starting integer time
  integer       :: n_relax      ! number of relaxation steps
  integer       :: n_obser      ! number of observation steps
  integer       :: n_save       ! number of steps between safety storages
  integer       :: n_linear_out ! number of steps between writing of positions
  ! MD steps completed in the observation time range
  integer :: n_mds_after_relax
  ! switch for CPU-timing
  logical, parameter :: l_cpu_timing = .FALSE.
  ! call cpu_time every n_call_cpu_time
  integer, parameter :: n_call_cpu_time = 100000
  ! total allowed CPU time in seconds
  ! 241920 secs are 4 weeks
  !  real, parameter :: cpu_time_available = 241920.0
  real, parameter :: cpu_time_available = 120.0 ! for testing
  ! switch for noting when program is stopped due to CPU time limit
  logical :: l_cpu_time_limit_reached = .FALSE.
  ! processor real time in seconds at begin of the simulation and after each
  ! MD step
  real :: cpu_time_start, cpu_time_now
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! switch and file for writing at a specified list of times
  logical :: l_read_sample_list = .FALSE.
  character(len=400) :: file_sample_list = 'file_name_of_sample_list'
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! flags
  !---------------------------------------------------------------------------
  ! f_wall_fix: wall atoms constrained (1) or vibrating (0)
  integer :: f_wall_fix = huge(1)
  ! flag to relax the system to the next energy minimum, (1): yes, (0): no
  ! sets forces to 0 and chooses a special  friction constant
  integer :: f_minimize = huge(1)
  ! flag for selecting the range of the LJ-potentials:
  ! f_cut_off = 0: range = 2^(1/6) \sigma (minimum of LJ-potential)
  ! f_cut_off = 1: range = 2.2 \sigma
  integer :: f_cut_off = huge(1)
  ! thermostatting flag:
  !--------------------
  ! the initial value is changed to 1 if we freeze the system.
  integer :: f_thermostat_mode = huge(1)
  ! number of thermostat modes (no thermostatting at all is mode 0)
  integer, parameter :: n_thermostat_modes = 8
  character(len=*), dimension(0:n_thermostat_modes-1), parameter :: &
       thermostat_mode_strings = &
       (/'no thermostatting at all                                    ', &
         'only apply friction on all unfixed particles                ', &
         'thermostatting on all unfixed particles in inertial (cm) system ', &
         'thermostatting on wall particles in inertial (cm) system      ', &
         'thermostatting on all unfixed particles in directions w/o strain', &
         'thermostatting on wall particles in inertial (rx_twall) system  ', &
         'thermostatting on wall particles in local inertial system     ', &
         'thermostatting on fluid particles in cm system of walls       '/)
  ! flag for turning on friction force on top wall
  integer :: f_friction_on_twall = huge(1)
  ! friction constant for friction on top wall
  real(kind=dp) :: friction_constant_twall = 1.0E100_dp
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! system and system size parameters
  !---------------------------------------------------------------------------
  ! switch for the desired wall geometry
  ! what it basically does is choosing a different setup for the bottom wall by
  ! (x-y) mirroring the top wall with respect to the bottom wall
  logical :: l_walls_are_commensurate = .FALSE.

  ! ONEDIM
  ! for one-dimensional walls the setup is different: a cell is just a particle
  ! the switch l_walls_are_commensurate has no meaning anymore.
  ! As the top wall is set up with x_space, and
  ! y_space = x_space * n_cell_w_x/n_cell_w_y, we have to choose
  ! n_cell_w_x smaller than n_cell_w_y
  ! spiral mean: 0.755
  ! integer, parameter :: n_cell_w_x=52, n_cell_w_y=68
  ! 517 = 11*47, 683 is a prime number
  ! integer, parameter :: n_cell_w_x=517, n_cell_w_y=683
  ! integer, parameter :: n_cell_w_x=60, n_cell_w_y=60
  !integer, parameter :: n_cell_w_x=450, n_cell_w_y=600
  !integer, parameter :: n_cell_w_x=600, n_cell_w_y=600
  !integer, parameter :: n_top_wall = n_cell_w_x, n_bottom_wall = n_cell_w_y
  ! n_cell_w_x, n_cell_w_y : number of cells for wall in x, y direction
  ! choose only one combination from below
```

```
! for simulations in the xy plane without wall interaction, we set
! n_cell_w_x=1, n_cell_w_y=1.
! n_cell_w_x / n_cell_w_y should be close to sqrt(3) for incommensurate walls
! !MHM denotes combinations used frequently by M. H. Mueser
integer, parameter :: n_cell_w_x=1, n_cell_w_y=1
! integer, parameter :: n_cell_w_x=12, n_cell_w_y=7
! integer, parameter :: n_cell_w_x=14, n_cell_w_y=8
! integer, parameter :: n_cell_w_x=28, n_cell_w_y=16
! integer, parameter :: n_cell_w_x=17, n_cell_w_y=10 ! MHM
! integer, parameter :: n_cell_w_x=19, n_cell_w_y=11
! integer, parameter :: n_cell_w_x=26, n_cell_w_y=15 ! very close to  sqrt(3)
! integer, parameter :: n_cell_w_x=31, n_cell_w_y=18 ! MHM
! integer, parameter :: n_cell_w_x=33, n_cell_w_y=19
! integer, parameter :: n_cell_w_x=38, n_cell_w_y=22
! integer, parameter :: n_cell_w_x=45, n_cell_w_y=26
! integer, parameter :: n_cell_w_x=52, n_cell_w_y=30
! integer, parameter :: n_cell_w_x=104, n_cell_w_y=60 ! for scaling
! integer, parameter :: n_cell_w_x=59, n_cell_w_y=34
! integer, parameter :: n_cell_w_x=62, n_cell_w_y=36 ! MHM
! integer, parameter :: n_cell_w_x=64, n_cell_w_y=37
! integer, parameter :: n_cell_w_x=71, n_cell_w_y=41 ! dev. = 3.4*10^-4
! integer, parameter :: n_cell_w_x=78, n_cell_w_y=45
! integer, parameter :: n_cell_w_x=97, n_cell_w_y=56 ! dev. = 9.2*10^-5
! integer, parameter :: n_cell_w_x=116, n_cell_w_y=67
! integer, parameter :: n_cell_w_x=123, n_cell_w_y=71
! integer, parameter :: n_cell_w_x=142, n_cell_w_y=82
! integer, parameter :: n_cell_w_x=149, n_cell_w_y=86
! integer, parameter :: n_cell_w_x=161, n_cell_w_y=93
! integer, parameter :: n_cell_w_x=168, n_cell_w_y=97 ! dev. = 9.2*10^-5
! integer, parameter :: n_cell_w_x=175, n_cell_w_y=101
! integer, parameter :: n_cell_w_x=187, n_cell_w_y=108
! integer, parameter :: n_cell_w_x=194, n_cell_w_y=112 ! dev. = 9.2*10^-5
! integer, parameter :: n_cell_w_x=201, n_cell_w_y=116
! integer, parameter :: n_cell_w_x=206, n_cell_w_y=119
! integer, parameter :: n_cell_w_x=213, n_cell_w_y=123
! integer, parameter :: n_cell_w_x=220, n_cell_w_y=127
! integer, parameter :: n_cell_w_x=227, n_cell_w_y=131
! integer, parameter :: n_cell_w_x=239, n_cell_w_y=138
! integer, parameter :: n_cell_w_x=246, n_cell_w_y=142
! integer, parameter :: n_cell_w_x=258, n_cell_w_y=149
! integer, parameter :: n_cell_w_x=265, n_cell_w_y=153 ! dev. = -2.5*10^-5
! integer, parameter :: n_cell_w_x=272, n_cell_w_y=157
! integer, parameter :: n_cell_w_x=277, n_cell_w_y=160 ! dev. = -8.0*10^-4
! integer, parameter :: n_cell_w_x=279, n_cell_w_y=161 ! dev. =  8.7*10^-4
! integer, parameter :: n_cell_w_x=284, n_cell_w_y=164
! integer, parameter :: n_cell_w_x=291, n_cell_w_y=168 ! dev. = 9.2*10^-5
! integer, parameter :: n_cell_w_x=298, n_cell_w_y=172
! (... some ratios skipped ...)
! integer, parameter :: n_cell_w_x=362, n_cell_w_y=209 ! dev. = 6.6*10^-6
! integer, parameter :: n_cell_w_x=627, n_cell_w_y=362 ! dev. = 6.6*10^-6
! integer, parameter :: n_cell_w_x=1086, n_cell_w_y=627 ! dev. = 6.6*10^-6
! bigger walls probably won't fit into memory (here we need about 1GB)
! old commensurate settings MHM
! integer, parameter :: n_cell_w_x=16, n_cell_w_y=8
! integer, parameter :: n_cell_w_x=32, n_cell_w_y=16
! for testing of the Frenkel-Kontorova neighbor lists
! integer, parameter :: n_cell_w_x=4, n_cell_w_y=3
! integer, parameter :: n_cell_w_x=8, n_cell_w_y=6
! number of particles in top and bottom wall (a cell contains two particles)
integer, parameter :: n_top_wall    = 2 * n_cell_w_x * n_cell_w_y
integer, parameter :: n_bottom_wall = 2 * n_cell_w_x * n_cell_w_y
! number of particles in both walls
integer, parameter :: n_wall = n_top_wall + n_bottom_wall
! monomers per chain
! if n_mon > 0, l_intra_molec_interaction has to be set to .TRUE., otherwise
! the intra molecular interaction is turned off
integer, parameter :: n_mon = 128
! number of chains in the fluid between the walls (unit: layers)
! example: n_chain = n_wall/(4*2*n_mon) is a quarter layer
! integer, parameter :: n_chain = (n_wall)/(2*n_mon)
integer, parameter :: n_chain = 32
! switch for deciding if we use biased monte carlo for
! the setup of the fluid particles (for chains lengths greater 50 this is
! basically a must, because a simple random walk is not self avoiding enough)
logical, parameter :: l_use_mc_fluid_setup            = .TRUE.
! if a 2-d interface should be set up exactly in the n_dim_pbc if there are
! walls (otherwise l_2d_flat_setup is ignored)
logical, parameter :: l_2d_flat_setup                 = .FALSE.
! if bond crossings in the x-y plane should be forbidden
! (requires n_dim_pbc .eq. 2)
logical, parameter :: l_forbid_xy_bond_crossings      = .TRUE.
! total number of monomers in the fluid
integer, parameter :: n_mon_tot = n_mon * n_chain
! total number of particles in the walls and the fluid
integer, parameter :: n_part = n_mon_tot + n_wall
! upper loop boundary,  n_moving = n_mon_tot or n_part:
! if(f_wall_fix.eq.1) n_moving = n_mon_tot
! else  n_moving = n_part
integer n_moving
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! positions, derivatives, forces, predictor-corrector coefficients
!------------------------------------------------------------------------------
! It seems that initializing the arrays here causes slow compiles with some
! compilers, thus only some arrays are initialized
! actual positions of particles, folded back in simulation box
! (present MD step)
real(kind=dp) :: r0(n_part, n_dim)
! actual positions of particles, not folded back in simulation box
! (present MD step), used for analysis
! r0_unfolded(:, :) is defined after SR corrector
real(kind=dp) :: r0_unfolded(n_part, n_dim)
! positions before last binning, folded back in simulation box
real(kind=dp) :: r0_old(n_part, n_dim)
! time averaged positions of particles
! real(kind=dp) :: r0_ave(n_part, n_dim)
! forces acting on the particles, re-initialized in subroutine thermostat at
! each MD step
real(kind=dp) :: force_determ(n_part, n_dim)
! the random force has to be initialized, because subroutine predict is
! called before thermostat
real(kind=dp) :: force_random(n_part, n_dim)
! equilibrium sites of wall atoms (both top and bottom wall)
real(kind=dp) :: r_wall_equi(n_wall, n_dim)
! number of neighbors for the Frenkel-Kontorova interaction
integer :: n_fk_neighbors
! equilibrium vectors between nearest neighbors for Frenkel-Kontorova model
! top wall
real(kind=dp), dimension(:, :), allocatable :: fk_neigh_vec_tw
! bottom wall
real(kind=dp), dimension(:, :), allocatable :: fk_neigh_vec_bw
! equilibrium distances between nearest neighbors for Frenkel-Kontorova model
real(kind=dp), dimension(:), allocatable :: fk_neigh_eq_dist_tw
real(kind=dp), dimension(:), allocatable :: fk_neigh_eq_dist_bw
! neighbor lists of walls for Frenkel-Kontorova model
integer, dimension(:, :), allocatable :: fk_neighbors
! velocities, accelerations and higher derivatives of the positions of the
! particles
real(kind=dp) :: rx(n_part, n_dim, n_order)
! predictor corrector coefficients
real(kind=dp) :: predict_coef(0:n_order-1, 0:n_order) = 1.0E100_dp
real(kind=dp) :: correct_coef(0:n_order) = 1.0E100_dp
! the potential contribution of the pressure tensor
! (averaged over the system)
real(kind=dp), dimension(n_dim, n_dim) :: press_tens_pot       = 0.0_dp
! wall-wall and intra-wall contribution
real(kind=dp), dimension(n_dim, n_dim) :: press_tens_pot_walls = 0.0_dp
! x-y dimensions of wall
real(kind=dp) :: x_space = 1.0E100_dp, y_space = 1.0E100_dp
! initial inter-wall spacing
real(kind=dp) :: z_space_wall = 1.0E100_dp
! boundaries, half boundaries
real(kind=dp) :: boundary(n_dim) = 1.0E100_dp, half_bound(n_dim) = 1.0E100_dp
! volume in periodic boundaries
real(kind=dp) :: pbc_volume = 1.0E100_dp
! upper and lower bound of the fluid in z-direction shared in different
! routines.  (Because of dependency problems, these variables are global)
real(kind=dp) :: highest_fluid_z=1.0E100_dp, lowest_fluid_z=1.0E100_dp
! maximally encountered correction of coordinates
real(kind=dp) :: max_encountered_correction = 0.0_dp
!------------------------------------------------------------------------------
!------------------------------------------------------------------------------
! definitions of particle parameters
!------------------------------------------------------------------------------
! FENE polymer backbone potential parameters
real(kind=dp), parameter :: r_chain = 1.5_dp, k_chain = 30.0_dp, &
     r_chain_2 = r_chain * r_chain
integer, parameter :: n_type = 2            ! number of particle types
real(kind=dp) :: friction(n_type)           ! friction constant of each type
real(kind=dp) :: mass_type(n_type)          ! mass of each type
real(kind=dp) :: epsil(n_type, n_type)      ! LJ-epsilon of pairs
real(kind=dp) :: e_shift(n_type, n_type)    ! LJ-energy shift of pairs
real(kind=dp) :: four_epsil(n_type, n_type) ! 4 * LJ-epsilon of pairs
real(kind=dp) :: sigma(n_type, n_type)      ! LJ-sigma of pairs
real(kind=dp) :: sigma_2(n_type, n_type)    ! LJ-sigma squared
! prefactors of FENE-force
real(kind=dp) :: epsil_k_chain_over_sigma_2(n_type, n_type)
real(kind=dp) :: range_1(n_type, n_type) ! LJ-range of pairs
real(kind=dp) :: range_2(n_type, n_type) ! LJ-range squared of pairs
! neighbor list ranges squared
real(kind=dp) :: neigh_list_rad_2(n_type, n_type)
real(kind=dp) :: rf_width(n_type)        ! random force distribution width
integer       :: type(n_part)            ! type of each particle
real(kind=dp) :: mass(n_part)            ! mass of each particle
real(kind=dp) :: k_spring_wall_tom       ! Tomlinson spring constant
! Frenkel-Kontorova spring constants
real(kind=dp) :: k_spring_wall_fre_kon_linear
real(kind=dp) :: k_spring_wall_fre_kon_vector
! the maximum of the interaction ranges
real(kind=dp) :: max_interaction_range = 0.0_dp
! bond-lengths.  Depend on the FENE parameters and on LJ sigmas
real(kind=dp) :: bond_length(n_type, n_type)
!------------------------------------------------------------------------------
! variables for external side conditions, including external spring
!------------------------------------------------------------------------------
! limit we impose on the sliding velocity of the top wall in each direction
real(kind=dp), parameter :: max_sliding_vel = 1.5_dp
! flags controlling mode
integer :: f_twall(n_dim_max)
! symbolic constants for the modes.  Modes can be different in different
! directions, e.g. the top wall is pulled in x while it is stabilized with
! a spring in y
! top wall moves with constant velocity
integer, parameter :: velocity_mode = 1
! force is excerted on the top wall (in z-direction giving a pressure
! a constant pressure), which can be ramped to see depinning
integer, parameter :: force_mode    = 2
! an external spring is pulling on the top wall
integer, parameter :: spring_mode   = 3
! the other way round
character(len=*), dimension(1:3), parameter :: &
     mode_strings = (/'velocity_mode ', 'force_mode    ', 'spring_mode   '/)
! f_twall: 1  top wall constr veloc. (v_0),   velocity mode
! f_twall: 2  top wall const. accel. (a_0),   force_mode
! f_twall: 3  top wall pulled by ext. spring, spring mode
! f_twall = 1 = velocity_mode
!--------------------------
! v_spring_twall: velocity with which the upper wall is moved
! k_spring_twall has no meaning in this case (think k = \infty)
! f_twall = 2 = force_mode
!-------------------------
! ext_force_twall: externally applied force that each individual atom
!                  in upper wall experiences
! ramp_force_twall: change of ext_force_twall per time unit
! f_twall = 3 = spring_mode (external spring pulling top wall)
!-------------------------------------------------------------
! v_spring_twall: velocity with which the spring is moved
! k_spring_twall: spring constant of spring
!                 with which upper wall is moved (normalized to
!                 number of atoms in upper wall)
real(kind=dp) :: ext_force_twall(n_dim)  = 1.0E100_dp
real(kind=dp) :: ramp_force_twall(n_dim) = 1.0E100_dp
real(kind=dp) :: v_spring_twall(n_dim)   = 1.0E100_dp
real(kind=dp) :: k_spring_twall(n_dim)   = 1.0E100_dp
! total force acting on top wall (from fluid, bottom wall and external
! spring)
! (re)initialized in subroutine thermostat
real(kind=dp) :: total_force_twall(n_dim) = 1.0E100_dp
! position of the top wall's center of mass
real(kind=dp) :: r0_twall(n_dim) = 1.0E100_dp
! position of the bottom wall's center of mass
! (currently the initial values are not changed)
real(kind=dp) :: r0_bwall(n_dim) = 0.0_dp
real(kind=dp) :: mass_twall = 1.0E100_dp ! mass of the top wall
real(kind=dp) :: mass_bwall = 1.0E100_dp ! mass of the bottom wall
real(kind=dp) :: mass_fluid_part = 1.0E100_dp ! mass of the fluid atoms
real(kind=dp) :: mass_twall_part = 1.0E100_dp ! mass of the top wall atoms
real(kind=dp) :: mass_bwall_part = 1.0E100_dp ! mass of the bottom wall atoms
! derivatives of the position of the top wall's center of mass
real(kind=dp) :: rx_twall(n_dim, n_order) = 1.0E100_dp
! tracking of periodic boundary conditions for top and bottom wall
! (currently not used)
integer :: pbc_twall(n_dim) = 0, pbc_bwall(n_dim) = 0
! counters for periodic boundary conditions (how many times the boundaries
! have been added or substracted)
integer :: pbc_count(n_part, n_dim)     ! present MD step
integer :: pbc_count_old(n_part, n_dim) ! before last binning
! position of spring pulling top wall (if applicable)
real(kind=dp) :: r0_spring_twall(n_dim) = 1.0E100_dp
! position of top wall where present and previous pinning occured
real(kind=dp) :: r0_pinned_twall(n_dim) = 1.0E100_dp
real(kind=dp) :: r0_last_pinned(n_dim)  = 1.0E100_dp
! array for deciding if force ramp is going up or down and with which
```

```
  ! multiplier
  integer :: s_force_grad(n_dim) = 0

  !---------------------------------------------------------------------------
  ! energy contributions at each MD step
  ! the initialization is important, because not all interactions might be
  ! turned on, so that these variables are not initialized at all other than
  ! here
  real(kind=dp) :: v_intra_molec = 0.0_dp ! potential energy of the bonds
  real(kind=dp) :: v_wall_wall   = 0.0_dp ! potential energy between walls
  ! potential energy of Tomlinson springs
  real(kind=dp) :: v_wall_harm_tom     = 0.0_dp
  ! potential energies of Frenkel-Kontorova springs
  real(kind=dp) :: v_wall_harm_fre_kon_linear = 0.0_dp
  real(kind=dp) :: v_wall_harm_fre_kon_vector = 0.0_dp
  real(kind=dp) :: v_fluid_wall  = 0.0_dp ! potential energy wall-fluid
  real(kind=dp) :: v_fluid_fluid = 0.0_dp ! potential energy of fluid pairs
  real(kind=dp) :: t_fluid       = 0.0_dp ! kinetic energy of fluid atoms
  real(kind=dp) :: t_wall        = 0.0_dp ! kinetic energy of wall atoms
  ! sum of the different contributions (wall -- wall, fluid -- fluid, ...)
  ! here the initialization is merely used to catch bugs.
  real(kind=dp) :: v_total=1.0E100_dp, t_total=1.0E100_dp, e_total=1.0E100_dp
  ! fileunit for logging energies
  integer, parameter :: energy_log_file_unit = 9
  !---------------------------------------------------------------------------
  ! observation variables
  !---------------------------------------------------------------------------
  ! variables for summing up the values of each MD step
  real(kind=dp) :: v_wall_wall_1 = 0.0_dp, v_wall_wall_2 = 0.0_dp
  real(kind=dp) :: t_wall_1      = 0.0_dp, t_wall_2      = 0.0_dp
  real(kind=dp) :: e_wall_wall_1 = 0.0_dp, e_wall_wall_2 = 0.0_dp
  ! fluid wall observables, analog to wall--wall interaction
  real(kind=dp) :: v_fluid_wall_1 = 0.0_dp, v_fluid_wall_2 = 0.0_dp
  ! variables for summing up the values of each MD step
  real(kind=dp) :: v_fluid_fluid_1 = 0.0_dp, v_fluid_fluid_2 = 0.0_dp
  real(kind=dp) :: t_fluid_1       = 0.0_dp, t_fluid_2       = 0.0_dp
  real(kind=dp) :: e_fluid_fluid_1 = 0.0_dp, e_fluid_fluid_2 = 0.0_dp
  ! sum over all MD steps (resp. the average)
  real(kind=dp) :: v_total_1 = 0.0_dp, v_total_2 = 0.0_dp
  real(kind=dp) :: t_total_1 = 0.0_dp, t_total_2 = 0.0_dp
  real(kind=dp) :: e_total_1 = 0.0_dp, e_total_2 = 0.0_dp
  ! observation variables, will be low pass filtered with a time constant of
  ! n_time_ave MD steps
  integer       :: n_time_ave                     = 0
  real(kind=dp) :: r0_twall_1(n_dim)              = 1.0E100_dp
  real(kind=dp) :: velocity_twall_1(n_dim)        = 1.0E100_dp
  real(kind=dp) :: ext_force_twall_1(n_dim)       = 1.0E100_dp
  real(kind=dp) :: total_force_twall_1(n_dim)     = 1.0E100_dp
  real(kind=dp) :: simulated_wall_wall_potential_1 = 1.0E100_dp
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! variables for binning
  !---------------------------------------------------------------------------
  ! number of fluid bins in x, y, z direction (we always need n_dim_max=3
  ! values, because the bin arrays have 4 dimensions)
  integer, dimension(n_dim_max) :: n_bin, n_bin_used
  ! selectors for the numbers of bins to look at (depending on number of bins
  ! in x, y direction)
  integer, dimension(n_dim_max) :: delta_bin
  ! widths of bins in x, y, z direction
  real(kind=dp), dimension(n_dim) :: r_bin
  ! the maximal number of bin entries of each (x,y,z)-bin
  ! allocation is done with indices 0:n_bin_XX, so 2^n-1 are good values for
  ! byte boundaries.
  ! integer, parameter :: n_bin_wa = 15, n_bin_fl = 31
  ! for thin films
  integer, parameter :: n_bin_wa = 15, n_bin_fl = 15
  ! with these variables we keep track of the maximal use of bins.
  ! n_bin_(wa/fl)_max > n_bin_(wa/fl) is fatal!
  integer :: n_bin_wa_max = 0, n_bin_fl_max = 0
  ! memory for the bins
  integer, dimension(:, :, :, :), allocatable :: bin_fluid
  ! for top and bottom wall only one bin layer in z-direction
  integer, dimension(:, :, :), allocatable :: bin_twall, bin_bwall
  ! skin thickness.  skin = 0.4 is pretty good for most situations.
  real(kind=dp) :: skin = 1.0E100_dp
  !---------------------------------------------------------------------------
  ! variables for neighbor lists
  !---------------------------------------------------------------------------
  ! maximal number of neighbors in the neighbor lists (due to the counter)
  ! in other words:  the lists can hold n_neigh_?? particle indices
  ! For special cases, the numbers below can be decreased to save memory.
  ! integer, parameter :: n_neigh_ff=47, n_neigh_fw=47, n_neigh_ww=31
  ! this only works for short range interaction and thin films
  integer, parameter :: n_neigh_ff=23, n_neigh_fw=15, n_neigh_ww=15
  ! the maximal number of entries in the neighbor lists
  integer :: n_neigh_ff_max = 0, n_neigh_fw_max = 0, n_neigh_ww_max = 0
  ! neighbor lists
  ! the max() is there to guarantee allocation if no fluid is present.
  ! fluid - fluid
  integer, dimension(:, :), allocatable :: ff_list
  ! fluid - wall
  integer, dimension(:, :), allocatable :: fw_list
  ! top wall - bottom wall
  integer, dimension(:, :), allocatable :: ww_list
  ! counts how many times the neighbor lists have been updated
  integer :: counter_list_updates = 0
  !---------------------------------------------------------------------------
  ! variables for ramping interaction and temperature
  !---------------------------------------------------------------------------
  ! r_2_min is the squared effective minimum distances of two particles.
  ! It is ramped down from r_2_min_init to 0.0 linearly in r_2_min_time
  ! if a new configuration is generated
  real(kind=dp) :: r_2_min, r_2_min_init, r_2_min_time
  ! if r_2_min is finite
  logical :: l_r_2_min_finite
  ! temperature variables
  ! temp is the present temperature.  If the initial temperature temp_init is
  ! not equal to the final temperature temp_final, then the final temperature
  ! is reached in temp_time
  real(kind=dp) :: temp      = 1.0E100_dp, temp_time  = 1.0E100_dp, &
                   temp_init = 1.0E100_dp, temp_final = 1.0E100_dp
  ! inverse temperature
  real(kind=dp) :: temp_inv  = 1.0E100_dp
  !---------------------------------------------------------------------------
  ! random number variables
  !---------------------------------------------------------------------------
  ! array for holding random numbers (only 32 bit pseudo random numbers are
  ! generated)
  ! check also the random number module luxury.f90 for more information
  real(kind=rngRealKind) :: random(n_part)
  ! seed for the random number generator
  integer :: iseed = 0
end module globals
!##############################################################################
```

initializationV1.9.f90

```
! module for initialization of the simulation:
! reading and writing of parameter files and configurations,
! initialization of binning
! Martin Aichele, 2001-02-07
! V1.9 real dimension switch
! Martin Aichele, 2003-02-24
! last modified 2003-02-28
! this module only exists to make the chosen KINDs visible in INTERFACE blocks
module mykinds
  use globals, only : dp
  implicit none
end module mykinds
module initialization
  ! module containing global variables
  use globals
  ! module for the interaction of the particles
  ! (needed for creating a default configuration if no configuration file
  ! exists)
  use interaction
  ! module containing the core MD-routines
  use mdroutines
  ! module for setting up chains using monte carlo methods
  use mcfluid
  use utilities
  implicit none
  !---------------------------------------------------------------------------
  ! debug switches      |-------- 31 characters ------|
  !---------------------------------------------------------------------------
  logical, parameter :: l_debug_init_parameters         = .FALSE.
  logical, parameter :: l_debug_init_binning            = .FALSE.
  logical, parameter :: l_debug_conf_default            = .FALSE.
  logical, parameter :: l_debug_rwc_3d                  = .FALSE.
  logical, parameter :: l_debug_rwc_2d                  = .FALSE.
  !---------------------------------------------------------------------------
  ! the mode parameters read from the parameter file which will be assigned
  ! to the appropriate variables
  real(kind=dp), dimension(1:n_dim_max), save :: &
       mode_parameter_one = 1.0E100_dp, mode_parameter_two = 1.0E100_dp
contains
  !---------------------------------------------------------------------------
  ! subroutine for initializing (large) arrays and allocating neighbor lists
  ! Initializing them during compilation is slow and takes a lot of memory.
  subroutine init_arrays
    ! periodic boundary condition counters
    pbc_count(:, :)     = 0
    pbc_count_old(:, :) = 0
    ! the random force has to be initialized, because subroutine predict is
    ! called before thermostat
    force_random(:, :) = 0.0_dp
    ! arrays here are initialized only for bug catching
    force_determ(:, :) = 1.0E100_dp
    r0(:, :)           = 1.0E100_dp
    r0_unfolded(:, :)  = 1.0E100_dp
    r0_old(:, :)       = 1.0E100_dp
    r_wall_equi(:, :)  = 1.0E100_dp
    rx(:, :, :)        = 1.0E100_dp
    random(:) = 1.0E10
    ! neighbor lists
    if(n_mon_tot .gt. 0) then
      ! fluid - fluid
      if(l_fluid_fluid_interaction) then
        allocate(ff_list(1:n_mon_tot, 0:n_neigh_ff))
        ff_list(:, :) = huge(1)
      end if
      ! fluid - wall
      if(l_fluid_wall_interaction) then
        allocate(fw_list(1:n_mon_tot, 0:n_neigh_fw))
        fw_list(:, :) = huge(1)
      end if
    end if
    ! top wall - bottom wall
    if(l_allow_wall_wall_interaction) then
      allocate(ww_list(1:n_top_wall, 0:n_neigh_ww))
      ww_list(:, :) = huge(1)
    end if
  end subroutine init_arrays
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! read parameters if parameter file can be read, else use default values.
  ! do some initialization and some checks.
  subroutine init_parameters(old_parameter_file)
    character(len=*), intent(in) :: old_parameter_file
    integer :: i_dim, i_fluid
    integer, parameter :: in_file = 10
    integer            :: io_status
    ! coefficients for the predictor corrector algorithm (helpers)
    real(kind=dp) :: pred_coef(0:n_order_max-1, 0:n_order_max)
    real(kind=dp) :: corr_coef(0:n_order_max)
    real(kind=dp) :: r_dummy
    ! loops over predictor-corrector coefficients
    integer :: i_order, j_order
    ! loops over particle types
    integer :: i_type, j_type
    ! loop over nearest neighbors
    integer :: i_neigh
    if(l_debug_init_parameters) then
      print *, "DEBUG: Entering SR init_parameters"
    end if
    ! try to open file containing parameters
    open(unit=in_file, file=old_parameter_file, &
         iostat=io_status, action="read", status="old")
    if(io_status /= 0) then
      write(unit=*, fmt='(3a)') "CAUTION: Could not read parameter file >>", &
           old_parameter_file, "<<"
      write(unit=*, fmt='(a)')  "         using default parameters."
      s_time  = 0            ! initial integer starting time
      n_relax = 10000        ! number of relaxation steps
      n_obser = 50000        ! number of observation steps
      n_save  = 100000       ! # of steps between backup configuration storage
      n_linear_out = 20000   ! # of steps for configuration saving
      n_time_ave = 100       ! time constant for output filtering
```

```
  dt = 0.005_dp          ! time step increment
  temp_time =  50.0_dp   ! time to reach final temperature
  temp_init =  0.5_dp    ! initial temperature
  temp_final = 0.5_dp    ! final temperature
  r_2_min_time = n_relax*dt ! time during which r_2_min goes to 0.0
  ! squared effective minimum distance
  ! 2.0_dp**(1.0_dp/6.0_dp) is the potential minimum of the LJ-potential
  ! r_2_min_init = 0.81 seems to work quite well.  When setting up
  ! polymers, it seems that increasing r_2_min_init helps to avoid
  ! overstretching the bonds.
  r_2_min_init = 0.65_dp * 2.0_dp**(1.0_dp/3.0_dp)
  ! spacing between wall units in x direction
  ! 1.2091356_dp was often used by MHM.
  x_space = 1.2091356_dp
  ! select commensurability
  l_walls_are_commensurate = .FALSE.
  ! y_space is defined later.
  ! this is a list of default values
  !---------------------------------
  ! initial inter-wall spacing (= z_space_wall).  For more fluid layers we
  ! have to leave more room
  z_space_wall = 2.0_dp * (real(2*n_mon_tot, kind=dp) &
       / real(n_wall, kind=dp)) + 1.25_dp
  ! Tomlinson spring constant to wall equililibrium sites
  k_spring_wall_tom = 50.0_dp
  ! Frenkel-Kontorova spring constants
  k_spring_wall_fre_kon_linear = 0.0_dp
  k_spring_wall_fre_kon_vector = 58.3333333_dp
  mass_type(1) = 1.0_dp ! masses
  mass_type(2) = 1.0_dp
  epsil(1,1) = 1.0_dp   ! LJ-Epsilons
  epsil(2,2) = 1.0_dp
  sigma(1,1) = 1.0_dp   ! LJ-Sigmas
  sigma(2,2) = 1.0_dp
  friction(1) = 0.5_dp  ! friction constants.  2.0 is high.
  friction(2) = 0.5_dp  ! for production runs use 0.5 -- 0.1
  iseed = 1000000       ! seed for random number generator
  skin=0.5_dp           ! skin thickness.  For sliding simluations this a
                        ! good value, for equilibrium simulations 0.25.
  f_wall_fix = 0 ! flag for constraining wall atoms: 0: FALSE, 1: TRUE
  f_cut_off = 0  ! cut-off flag
  f_thermostat_mode = 4 ! thermostat mode flag
  f_minimize = 0 ! flag to relax to next minimum
  ! flag for turning on friction force on top wall in direction of pulling
  f_friction_on_twall = 0
  ! frictiom constant for friction on top wall
  friction_constant_twall = 0.05_dp
  ! flags and variables controlling boundary conditions
  f_twall(1) = spring_mode ! = 3
  f_twall(2) = spring_mode ! = 3
  f_twall(3) = force_mode  ! = 2
  mode_parameter_one(1) =  0.0_dp
  mode_parameter_one(2) =  0.0_dp
  mode_parameter_one(3) = -5.0_dp
  mode_parameter_two(1) = 5.0_dp
  mode_parameter_two(2) = 5.0_dp
  mode_parameter_two(3) = 0.0_dp
else ! file is readable
  write(unit=*, fmt='(3a)') "MESSAGE: Reading parameters from file >>", &
       old_parameter_file, "<<"
  ! read parameters in this IO unit
  call read_parameters(in_file)
  ! close input file
  close(unit=in_file)
end if
! if wall atoms are fixed, we do not need to loop over them for propagation
if(f_wall_fix.eq.1) then
  n_moving = n_mon_tot
else
  n_moving = n_part
end if
! Now we have to initialize the parameters defining the boundary
! conditions.  That is, the values read to mode_parameter_one(:) and
! mode_parameter_two(:) have to be assigned to the approbriate arrays
do i_dim = 1, n_dim
  select case(f_twall(i_dim))
  case(velocity_mode)
    v_spring_twall(i_dim) = mode_parameter_one(i_dim)
    ! k_spring_twall(i_dim) is redundant in velocity_mode, so warn
    k_spring_twall(i_dim) = 1.0E100_dp ! help catch bugs
    if(mode_parameter_two(i_dim) .ne. 0.0_dp) then
      write(unit=*, fmt='(a)') &
           "WARNING (SR init_parameters): In velocity mode, it is not &
           &allowed to set the spring stiffness."
    end if
    ! however, it makes a good test to compare velocity mode and spring
    ! mode, because spring mode becomes velocity mode for infinite spring
    ! stiffness.  However, putting 10^99 as spring stiffness causes the
    ! program to crash due to numerical accuracy problems.  One has to set
    ! the spring stiffness to something like 10^9 (depending on the
    ! supported accuracy), then one sees the system behaves like in
    ! velocity mode.  Hopefully ;)
  case(force_mode)
    ext_force_twall(i_dim)  = mode_parameter_one(i_dim)
    ramp_force_twall(i_dim) = mode_parameter_two(i_dim)
  case(spring_mode)
    v_spring_twall(i_dim) = mode_parameter_one(i_dim)
    k_spring_twall(i_dim) = mode_parameter_two(i_dim)
  case default
    write(unit=*, fmt='(a, i1, a, i1, a)') &
         "ERROR (SR init_parameters): mode flag f_twall(", i_dim, ") =", &
         f_twall(i_dim), " not recognized."
    stop
  end select
end do ! loop over all directions
! loop over dimensions not simulated
do i_dim = n_dim+1, n_dim_max
  if(f_twall(i_dim) .ne. 0) then
    write(unit=*, fmt='(a, i1, a, i1, a)') &
         "ERROR (SR init_parameters): mode flag f_twall(", i_dim, ") not 0"
    stop
  end if
  if(mode_parameter_one(i_dim) .ne. 0.0_dp) then
    write(unit=*, fmt='(a,i1,a,i1)') "WARNING (SR init_parameters): &
         &Ignoring mode parameter one in dimension ", &
         i_dim, " > n_dim=", n_dim
  end if
  if(mode_parameter_two(i_dim) .ne. 0.0_dp) then
    write(unit=*, fmt='(a,i1,a,i1)') "WARNING (SR init_parameters): &
         &Ignoring mode parameter two in dimension ", &
         i_dim, " > n_dim=", n_dim
  end if
end do
! write out the modes and values in each direction
do i_dim = 1, n_dim
  write (unit=*, fmt='(a,i1,3a,2e12.4)') &
       "MESSAGE: In ", i_dim, "-direction we are in ", &
       mode_strings(f_twall(i_dim)), "with values", &
       mode_parameter_one(i_dim), mode_parameter_two(i_dim)
end do
```

```
! write the thermostatting mode to screen
write(unit=*, fmt='(2a)') "MESSAGE: We do ", &
     thermostat_mode_strings(f_thermostat_mode)
if(f_thermostat_mode .eq. 4) then
  ! print the thermostatted directions on screen
  write(unit=*, fmt='(a)') "         these are directions:"
  do i_dim =1, n_dim
    if( & ! velocity_mode and velocity is zero
         ((f_twall(i_dim).eq.velocity_mode) &
         .and.(v_spring_twall(i_dim).eq.0.0_dp))&
         .or. & ! just a force, no force ramp
         ! or negative force ramp in z-direction (pressure increase)
         ((f_twall(i_dim).eq.force_mode) &
         .and.(ramp_force_twall(i_dim).eq.0.0_dp &
         .or. ((ramp_force_twall(i_dim).lt.0.0_dp).and.(i_dim.eq.n_dim))))&
         .or. & ! spring mode when spring is not pulled
         ((f_twall(i_dim).eq.spring_mode) &
         .and.(v_spring_twall(i_dim).eq.0.0_dp)) &
         ) then
      write(unit=*, fmt='(a, i1)') "         ", i_dim
    end if
  end do
end if
! calculate approximate equivalent Tomlinson stiffness for FK-model
if((k_spring_wall_fre_kon_linear .ne. 0.0_dp) &
     .or. (k_spring_wall_fre_kon_vector .ne. 0.0_dp)) then
  write(unit=*, fmt='(a, g13.6)') &
       "MESSAGE: Approximate equivalent Tomlinson spring stiffness=", &
       k_spring_wall_tom + (8.0_dp/3.0_dp)*k_spring_wall_fre_kon_linear &
       + 6.0_dp * k_spring_wall_fre_kon_vector
end if
! initialize temperature
if(temp_init.ne.temp_final) then
  if(temp_time.gt.n_relax*dt) then
    write(unit=*, fmt='(a)') &
         "WARNING: Temperature ramp might take too long."
  end if
  if(temp_time.le.0.0_dp) then
    write(unit=*, fmt='(2a)') &
         "ERROR: (temp_init .ne. temp_final).and.(temp_time.le.0.0_dp)", &
         " is not sensible."
    stop
  end if
  temp = temp_init
else ! no temperature change intended
  temp = temp_final
end if
! initialize variables that immediately follow from input variables
!=================================================================
! squared timestep
dt_2 = dt * dt
! inverse timestep
dt_inv = 1.0_dp / dt
! total mass of the fluid particles
! fluid particle have type 1
mass_fluid_part = real(n_mon_tot, kind=dp) * mass_type(1)
! total mass of the top wall particles.
! Wall particles always have type n_type
mass_twall_part = real(n_top_wall, kind=dp) * mass_type(n_type)
! total mass of the bottom wall particles.
mass_bwall_part = real(n_bottom_wall, kind=dp) * mass_type(n_type)
! mass of the walls thought as rigid lattices.  These can be thought of
! particles which are propagated.  These masses need not be the same.  If
! mass_twal is 2*mass_twall_part then this corresponds to two layers of
! particles, only one of which is treated explicitely.  For the propagation
! of the walls mass_twall is used.  mass_twall_part is needed for kinetic
! energy calculation.
mass_twall = mass_twall_part
mass_bwall = mass_bwall_part
! apply standard sum rules to Lennard Jones parameters
do i_type = 1, n_type-1
  do j_type = i_type+1,n_type
    epsil(i_type,j_type) &
         = sqrt(epsil(i_type,i_type)*epsil(j_type,j_type))
    epsil(j_type,i_type) = epsil(i_type,j_type)
    sigma(i_type,j_type) &
         = (sigma(i_type,i_type)+sigma(j_type,j_type))/2.0_dp
    sigma(j_type,i_type) = sigma(i_type,j_type)
  end do
end do
! initialize sigma_2
! initialize prefactors for bond-potentials
! absorb prefactor 4 in LJ-epsilon
do i_type = 1,n_type
  do j_type = 1,n_type
    sigma_2(i_type,j_type) = sigma(i_type,j_type)**2
    epsil_k_chain_over_sigma_2(i_type, j_type) = &
         (epsil(i_type,j_type) * k_chain)/sigma_2(i_type,j_type)
    four_epsil(i_type,j_type) = 4.0_dp * epsil(i_type,j_type)
  end do
end do
! define cutoffs and shifts in energy, find maximal interaction range
do i_type = 1, n_type
  do j_type = 1, n_type
    select case(f_cut_off)
    case(0)
      ! purely repulsive potential
      ! 2**(1/6) = 1.122462
      range_2(j_type,i_type) = (2.0_dp**(1.0_dp/6.0_dp) &
           * sigma(j_type,i_type))**2
    case(1)
      ! frequently used by M. H. Mueser and M. O. Robbins
      range_2(j_type,i_type) = (2.2_dp*sigma(j_type,i_type))**2
    case(2)
      ! long range interaction as used by C. Bennemann, F. Varnik
      ! the difference to 2.2 is not very important.
      ! 2 * 2**(1/6) = 2.244924
      range_2(j_type,i_type) = (2.0_dp * 2.0_dp**(1.0_dp/6.0_dp) &
           * sigma(j_type,i_type))**2
    case(3)
      ! purely repulsive walls and fluid particles with long range
      ! LJ-interaction like in case 2
      range_2(j_type,i_type) = (2.0_dp**(1.0_dp/6.0_dp) &
           * sigma(j_type,i_type))**2
      ! fluid particles are always defined as type 1 (see SR conf_default)
      if((j_type .eq. 1) .and. (i_type .eq. 1)) then
        range_2(j_type,i_type) = (2.0_dp * 2.0_dp**(1.0_dp/6.0_dp) &
             * sigma(j_type,i_type))**2
      end if
    case default ! fall-through
      write(unit=*, fmt='(a, i2, a)') &
           "ERROR (SR init_parameters): case(f_cut_off=",&
           f_cut_off, ") not recognized."
      stop
    end select
    r_dummy = (sigma(j_type,i_type)**2/range_2(j_type,i_type))**3
    ! Epsilon does not need to be attached at this point.  Epsilon enters
    ! in the interaction routines.
    e_shift(j_type, i_type) = r_dummy*(r_dummy-1.0_dp)
  end do
```

```
end do
! LJ-ranges
do i_type = 1, n_type
  do j_type = 1, n_type
    range_1(i_type, j_type) = sqrt(range_2(j_type, i_type))
  end do
end do
! define maximal interaction range.  Needed for finding the minimal and
! optimal bin width.
max_interaction_range = sqrt(maxval(range_2(:, :)))
! A particle is put in the neighbor list, if
! r_ij^2 <= r_cutoff^2 + 2*r_cutoff*skin + skin^2,
! the right hand side is put in an array.
do i_type = 1, n_type
  do j_type = 1, n_type
    neigh_list_rad_2(j_type, i_type) = &
         range_2(j_type, i_type) &
         +  2.0_dp*sqrt(range_2(j_type, i_type))*skin + skin**2
  end do
end do
! find bond-lengths of all pairs
call bisection(bond_potential_derivative, 0.75_dp, 0.75_dp*r_chain, &
     10.0_dp**(-precision(1.0_dp)+3), r_dummy)
do i_type = 1, n_type
  do j_type = 1, n_type
    ! the result is in units of sigma, so multiply with sigma
    bond_length(j_type, i_type) = sigma(j_type, i_type) * r_dummy
    write(unit=*, fmt='(a, i2, a, i2, a, g13.6, a, g13.6)') &
         "MESSAGE: bond_length(", j_type,",", i_type, ")=", &
         r_dummy, " sigma =", bond_length(j_type, i_type)
  end do
end do
! define coefficients for the predictor-corrector algorithm
!============================================================
! the predictor coefficients are just the Taylor expansion coefficients
! multiplied by prefactors stemming from the scaling of rx(:, :, :)
! with (dt**n)/n!:
! predict_coeff(i, j) = j! / (i! * (j-i)!)
! The corrector coefficients were chosen by Gear to make the local
! trunction error of O(dt**(n_order+1)) for linear differential equations.
! We follow J. M. Haile, "Molecular Dynamics Simulation: Elementary
! methods", Wiley Professional Paperback Series, 1997, pp. 160
! for the coefficients.
! initialize predictor coefficients (write to helpers, as this is easier)
do i_order = 0,n_order_max-1
  do j_order = 0,n_order_max
    pred_coef(i_order,j_order) = 0.0_dp
  end do
  pred_coef(0, i_order+1) = 1.0_dp
  if(i_order.ge.1) pred_coef(1, i_order+1) = i_order+1
  pred_coef(i_order, i_order) = 1.0_dp
end do
pred_coef(2,3) =  3.0_dp
pred_coef(2,4) =  6.0_dp
pred_coef(2,5) = 10.0_dp
pred_coef(3,4) =  4.0_dp
pred_coef(3,5) = 10.0_dp
pred_coef(4,5) =  5.0_dp
! now write to the matrix holding the predictor coefficients for
! computation
do i_order = 0, n_order-1
  do j_order = 0, n_order
    predict_coef(i_order,j_order) = 0.0_dp
  end do
  do j_order = i_order, n_order
    predict_coef(i_order, j_order) = pred_coef(i_order, j_order)
  end do
end do
! initialize corrector
do i_order = 1, n_order
  corr_coef(i_order) = 0.0_dp
end do
if(n_order.eq.2) then ! velocity Verlet algorithm
  corr_coef(0) = 0.0_dp
  corr_coef(1) = 1.0_dp
  corr_coef(2) = 1.0_dp  ! this is true in any order
else if (n_order.eq.3) then
  corr_coef(0) = 1.0_dp/6.0_dp
  corr_coef(1) = 5.0_dp/6.0_dp
  corr_coef(2) = 1.0_dp
  corr_coef(3) = 1.0_dp/3.0_dp
else if (n_order.eq.4) then
  corr_coef(0) = 19./120.0_dp
  ! corr_coef(0) = 19.0_dp/90.0_dp ! used by MHM
  ! corr_coef(0) = 19./120.0_dp    ! MHM: only if thermostat absent
  corr_coef(1) = 3.0_dp/4.0_dp
  corr_coef(3) = 1.0_dp/2.0_dp
  corr_coef(2) = 1.0_dp
  corr_coef(4) = 1.0_dp/12.0_dp
else if (n_order.eq.5) then
  corr_coef(0) = 3.0_dp/16.0_dp
  ! corr_coef(0) = 3.0_dp/16.0_dp ! used by MHM
  ! corr_coef(0) = 3.0_dp/20.0_dp ! MHM: only if thermostat absent
  corr_coef(1) = 251.0_dp/360.0_dp
  corr_coef(2) = 1.0_dp
  corr_coef(3) = 11.0_dp/18.0_dp
  corr_coef(4) = 1.0_dp/6.0_dp
  corr_coef(5) = 1.0_dp/60.0_dp
else
  write(unit=*, fmt='(a, i3, a)') &
       "ERROR (SR init_params): Order ", n_order, &
       " for predictor corrector unavailable"
  stop
end if
! now write to array holding the corrector coefficients for computation
do i_order = 0, n_order
  correct_coef(i_order) = corr_coef(i_order)
end do
! write out the coefficients if desired
if(.FALSE.) then
  write(unit=*, fmt='(a)') &
    "MESSAGE (SR init_params): Printing predictor-corrector coefficients:"
  do i_order = 0, n_order-1
    write(unit=*, fmt='(a, i1, a)') &
         "MESSAGE (SR init_params): predict_coef(", i_order, ")="
    do j_order = 0, n_order
      write(unit=*, fmt='(a, f10.3)') &
           "             ", predict_coef(i_order, j_order)
    end do
  end do
  do i_order = 0, n_order
    write(unit=*, fmt='(a, i1, a, f10.3)') &
         "MESSAGE (SR init_params): correct_coef(", i_order, ")=", &
         correct_coef(i_order)
  end do
end if ! if we print the predictor-corrector coeeficients
! initialization of the counters for the minimum image convention is done
! in SR init_arrays.
! define y_space depending on walls
! because y_space follows from x_space it is not read from the parameter
! file, but computed here.
  ! force pressure conversion factor due to the size of the wall
  ! unit cells is 1/2 * \sqrt(3) x_space^2
  ! (convert from Load/particle to Load/area, and one particle occupies an
  ! area of 1/2 * x_space * y_space)
  ! define y_space depending on wall type and dimension
  if(l_walls_are_commensurate .and. n_dim .eq. 3 .and. n_dim_pbc .eq. 2) then
    ! undistorted lattice
    y_space = x_space*sqrt3
  else if((.not.l_walls_are_commensurate) .and. &
       n_dim .eq. 3 .and. n_dim_pbc .eq. 2) then
    ! one tries to have a wall cell ratio close to sqrt(3) to get nearly
    ! quadratic walls without distorting the lattice too much.
    y_space = &
         (real(n_cell_w_x, kind=dp)/real(n_cell_w_y, kind=dp))*x_space
  else if(n_dim .eq. 2 .and. n_dim_pbc .eq. 1) then
    ! for one-dimensional walls there is no lattice and y_space is always
    ! defined like this.
    y_space = &
         (real(n_top_wall, kind=dp)/real(n_bottom_wall, kind=dp))*x_space
  else if(n_dim .eq. n_dim_pbc) then
    y_space = x_space
  else
    write(unit=*, fmt='(a)') &
         "ERROR (SR init_parameters): Fallthrough when setting up y_space"
    stop
  end if
  write(unit=*, fmt='(a, f16.8, a, f16.8)') &
       "MESSAGE (SR init_parameters): x_space=",x_space,", y_space=",y_space
  if(n_dim .eq. 3 .and. n_dim_pbc .eq. 2) then
    write(unit=*, fmt='(a, f16.8)') &
         "MESSAGE (SR init_parameters): Load--pressure conversion factor=", &
        0.5_dp * x_space * y_space
  end if
  ! boundaries in x,y,z direction
  ! boundary(n_dim) will be changed if configuration is read from file
  ! For 1-d interfaces, boundary(2) = z_space_wall
  boundary(1) = x_space * real(n_cell_w_x, kind=dp)
  boundary(2) = y_space * real(n_cell_w_y, kind=dp)
  ! will be re-initialized later if there is a configuration
  if(n_dim .eq. 3) then
    boundary(n_dim) = z_space_wall
  end if
  ! call checking routine
  call checks
  if(l_debug_init_parameters) then
    print *, "DEBUG: Leaving SR init_parameters"
  end if
end subroutine init_parameters
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! initializes the Frenkel-Kontorova wall model
subroutine init_fk_model
  integer :: i_neigh
  if(n_dim .ne. n_dim_pbc+1) then
    write(unit=*, fmt='(a)') "ERROR (SR init_fk_model): &
         &n_dim .ne. n_dim_pbc+1"
    stop
  end if
  ! number of neighbors for the Frenkel-Kontorova interaction
  select case(n_dim)
  case(3)
    n_fk_neighbors = 6
  case(2)
    n_fk_neighbors = 2
  case default
    write(unit=*, fmt='(a, i1, a)') "ERROR (SR init_fk_model): &
         &n_dim=", n_dim, " for FK-walls not implemented"
    stop
  end select
  ! allocate
  allocate(fk_neigh_vec_tw(n_dim, 1:n_fk_neighbors))
  allocate(fk_neigh_vec_bw(n_dim, 1:n_fk_neighbors))
  allocate(fk_neigh_eq_dist_tw(1:n_fk_neighbors))
  allocate(fk_neigh_eq_dist_bw(1:n_fk_neighbors))
  ! neighbor lists of walls for Frenkel-Kontorova model.  Each wall particle
  ! is connected with its six/two neighbors.  (2-dim or 1-dim walls)
  ! These neighbors do not change during the simulation, but we have to take
  ! care of the periodic boundary conditions and select the right periodic
  ! image.  (A particle is just a representative for an equivalence class of
  ! particles)
  ! The n_fk_neighbors neighbors are saved as particle indices, whereas the
  ! second enumeration index uses wall particle indexing.
  allocate(fk_neighbors(n_fk_neighbors, n_wall))
  fk_neighbors(:, :) = huge(1)
  if(n_dim .eq. 3 .and. n_dim_pbc .eq. 2) then
    ! define equilibrium nearest neighbor vectors and distances in
    ! the walls for the Frenkel-Kontorova model
    ! layout:
    !--------
    ! top wall          x-y mirrored (incommensurate bottom wall)
    !    1   2           4   3
    !     \ /             \ /
    ! 6 -center- 3     5 -center- 2
    !     / \             / \
    !    5   4           6   1
    !
    ! distances should be quite close to x_space
    ! top wall
    !---------
    fk_neigh_vec_tw(1, 1) = -0.5_dp * x_space
    fk_neigh_vec_tw(2, 1) =  0.5_dp * y_space
    fk_neigh_vec_tw(1, 2) =  0.5_dp * x_space
    fk_neigh_vec_tw(2, 2) =  0.5_dp * y_space
    fk_neigh_vec_tw(1, 3) =  x_space
    fk_neigh_vec_tw(2, 3) =  0.0_dp
    fk_neigh_vec_tw(1, 4) =  0.5_dp * x_space
    fk_neigh_vec_tw(2, 4) = -0.5_dp * y_space
    fk_neigh_vec_tw(1, 5) = -0.5_dp * x_space
    fk_neigh_vec_tw(2, 5) = -0.5_dp * y_space
    fk_neigh_vec_tw(1, 6) = -x_space
    fk_neigh_vec_tw(2, 6) =  0.0_dp
    ! bottom wall
    !------------
    ! for incommensurate walls, x- and y- are interchanged, so we have to
    ! change the vectors, too.
    ! Look in SR conf_default for the construction of the walls.
    if(l_walls_are_commensurate) then
      fk_neigh_vec_bw(1, 1) = -0.5_dp * x_space
      fk_neigh_vec_bw(2, 1) =  0.5_dp * y_space
      fk_neigh_vec_bw(1, 2) =  0.5_dp * x_space
      fk_neigh_vec_bw(2, 2) =  0.5_dp * y_space
      fk_neigh_vec_bw(1, 3) =  x_space
      fk_neigh_vec_bw(2, 3) =  0.0_dp
      fk_neigh_vec_bw(1, 4) =  0.5_dp * x_space
      fk_neigh_vec_bw(2, 4) = -0.5_dp * y_space
      fk_neigh_vec_bw(1, 5) = -0.5_dp * x_space
      fk_neigh_vec_bw(2, 5) = -0.5_dp * y_space
      fk_neigh_vec_bw(1, 6) = -x_space
      fk_neigh_vec_bw(2, 6) =  0.0_dp
```

```
    else
      fk_neigh_vec_bw(1, 1) =  0.5_dp * y_space
      fk_neigh_vec_bw(2, 1) = -0.5_dp * x_space
      fk_neigh_vec_bw(1, 2) =  0.5_dp * y_space
      fk_neigh_vec_bw(2, 2) =  0.5_dp * x_space
      fk_neigh_vec_bw(1, 3) =  0.0_dp
      fk_neigh_vec_bw(2, 3) =  x_space
      fk_neigh_vec_bw(1, 4) = -0.5_dp * y_space
      fk_neigh_vec_bw(2, 4) =  0.5_dp * x_space
      fk_neigh_vec_bw(1, 5) = -0.5_dp * y_space
      fk_neigh_vec_bw(2, 5) = -0.5_dp * x_space
      fk_neigh_vec_bw(1, 6) =  0.0_dp
      fk_neigh_vec_bw(2, 6) = -x_space
    end if
    ! walls are flat, so difference in z-component is always 0
    fk_neigh_vec_tw(n_dim, :) = 0.0_dp
    fk_neigh_vec_bw(n_dim, :) = 0.0_dp
    ! the vectors above are from the center to the neighbors
    ! we switch signs for getting the vectors from the neighbors to the
    ! center, this is just a convention chosen in the interaction routines.
    fk_neigh_vec_tw(:, :) = -1.0_dp * fk_neigh_vec_tw(:, :)
    fk_neigh_vec_bw(:, :) = -1.0_dp * fk_neigh_vec_bw(:, :)
    ! equilibrium neighbor distances (no distinction between top and
    ! bottom wall necessary here)
    do i_neigh = 1, n_fk_neighbors
      fk_neigh_eq_dist_tw(i_neigh) = &
           sqrt(dot_product(fk_neigh_vec_tw(:, i_neigh), &
           fk_neigh_vec_tw(:, i_neigh)))
    end do
    fk_neigh_eq_dist_bw(:) = fk_neigh_eq_dist_tw(:)
    ! print *, fk_neigh_eq_dist_tw
    ! print *, "x_space=", x_space
  else if(n_dim .eq. 2 .and. n_dim_pbc .eq. 1) then ! one-dimensional walls
    ! ONEDIM
    ! layout:
    !--------
    !
    ! (commensurate and incommensurate)
    !
    ! 1 - center - 2
    !
    fk_neigh_vec_tw(1, 1) = -x_space
    fk_neigh_vec_tw(2, 1) = 0.0_dp
    fk_neigh_vec_tw(1, 2) = x_space
    fk_neigh_vec_tw(2, 2) = 0.0_dp
    ! bottom wall
    fk_neigh_vec_bw(1, 1) = -y_space
    fk_neigh_vec_bw(2, 1) = 0.0_dp
    fk_neigh_vec_bw(1, 2) = y_space
    fk_neigh_vec_bw(2, 2) = 0.0_dp
    ! the vectors above are from the center to the neighbors
    ! we switch signs for getting the vectors from the neighbors to the
    ! center, this is just a convention chosen in the interaction routines.
    fk_neigh_vec_tw(:, :) = -1.0_dp * fk_neigh_vec_tw(:, :)
    fk_neigh_vec_bw(:, :) = -1.0_dp * fk_neigh_vec_bw(:, :)
    ! equilibrium neighbor distances
    do i_neigh = 1, n_fk_neighbors
      fk_neigh_eq_dist_tw(i_neigh) = x_space
      fk_neigh_eq_dist_bw(i_neigh) = y_space
    end do
  end if
  ! set up neighbor lists for Frenkel-Kontorova walls if needed
  select case(n_dim)
  case(3)
    call create_fk_neigh_list
  case(2)
    call create_fk_neigh_list_1d
  case default
    write(unit=*, fmt='(a, i1, a)') "ERROR (SR init_fk_model): &
         &n_dim=", n_dim, " for FK-walls not implemented"
    stop
  end select
  ! routine that checks that actual positions and inter-particle-differences
  ! according to the Frenkel-Kontorova neighbor-lists match
  call compare_fk_positions
end subroutine init_fk_model
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! checks validity and consistency of flags and performs some sanity tests
! Because we have so many choices of what system to simulate and so many
! switches and parameters, this subroutine is quite important.
subroutine checks
  ! n_mds_steps_compat_r_time_prec is the number of MD steps compatible
  ! with the precision for writing out the real time.
  integer :: counter, i_dim, n_mds_steps_compat_r_time_prec = huge(1)
  ! check dimension values
  if(n_dim_max .ne. 3) then
    write(unit=*, fmt='(a)') "ERROR (SR checks): n_dim_max .ne. 3"
    stop
  end if
  if(n_dim .lt. 1 .or. n_dim .gt. 3) then
    write(unit=*, fmt='(a)') "ERROR (SR checks): &
         &n_dim .lt. 1 .or. n_dim .gt. 3"
    stop
  end if
  if(n_dim .lt. n_dim_pbc) then
    write(unit=*, fmt='(a)') "ERROR (SR checks): &
         &n_dim .lt. n_dim_pbc"
    stop
  end if
  if((n_dim - n_dim_pbc) .gt. 1) then
    write(unit=*, fmt='(a)') "ERROR (SR checks): &
         &(n_dim - n_dim_pbc) .gt. 1 not implemented"
    stop
  end if
  ! check sensibility of time constants
  if(n_time_ave .le. 0) then
    write(unit=*, fmt='(a, i12, a)') "ERROR (SR checks): n_time_ave=", &
         n_time_ave, " .le. 0"
    stop
  end if
  if(n_time_ave .gt. n_obser) then
    write(unit=*, fmt='(2(a, i12))') "WARNING (SR checks): n_time_ave=", &
         n_time_ave, " > n_obser=", n_obser
  end if
  if(n_save .le. 0) then
    write(unit=*, fmt='(a, i12, a)') &
         "ERROR (SR checks): n_save=", n_save, " .le. 0"
    stop
  end if
  if(n_linear_out .le. 0) then
    write(unit=*, fmt='(a, i12, a)') &
         "ERROR (SR checks): n_linear_out=", n_linear_out, " .le. 0"
    stop
  end if
  ! check if precision for writing r_time is sufficient
  ! (Assuming that the timestep dt is of order 0.001)
  ! note: huge(1)=  2147483647 on 32-bit machines
  !       ln(10**12) = 27.63
  if(mod(n_time_ave, 1000) .eq. 0) then
    if(log(real(huge(1))) .gt. 27.63) then
      n_mds_steps_compat_r_time_prec = 10**12-1
    else
      n_mds_steps_compat_r_time_prec = huge(1)
    end if
  else if(mod(n_time_ave, 100) .eq. 0) then
    if(log(real(huge(1))) .gt. 25.328) then
      n_mds_steps_compat_r_time_prec = 10**11-1
    else
      n_mds_steps_compat_r_time_prec = huge(1)
    end if
  else if(mod(n_time_ave, 10) .eq. 0) then
    n_mds_steps_compat_r_time_prec = 10**10 - 1
  else
    n_mds_steps_compat_r_time_prec = 10**9 - 1
  end if
  if(n_relax + n_obser .gt. n_mds_steps_compat_r_time_prec) then
    write(unit=*, fmt='(a)') "ERROR (SR checks): &
         &Number of MD steps too big for chosen precision for r_time &
         &(format es17.11)"
    stop
  end if
  ! check if there's something to simulate
  if(n_moving .le. 0) then
    write(unit=*, fmt='(a, i12, a)') &
         "ERROR (SR checks): n_moving=", n_moving, " .le. 0"
    write(unit=*, fmt='(a)') "       That would be a boring simulation."
    stop
  end if
  if((f_minimize .ne. 1) .and. (f_thermostat_mode .eq. 1)) then
    write(unit=*, fmt='(a)') &
         "ERROR (SR checks): f_minimize .ne. 1 and f_thermostat_mod&
         &e=1 does not make sense"
    stop
  end if
  if((f_wall_fix .eq. 1) .and. (f_thermostat_mode .eq. 3)) then
    write(unit=*, fmt='(a)') &
         "ERROR (SR checks): f_wall_fix .eq. 1, so it makes no &
         &sense to thermostat walls only (f_thermostat_mode .eq. 3)"
    stop
  end if
  if((f_wall_fix .eq. 1) .and. (f_thermostat_mode .eq. 5)) then
    write(unit=*, fmt='(a)') &
         "ERROR (SR checks): f_wall_fix .eq. 1, so it makes no &
         &sense to thermostat walls only (f_thermostat_mode .eq. 5)"
    stop
  end if
  if((n_mon_tot .eq. 0) .and. (f_thermostat_mode .eq. 2)) then
    write(unit=*, fmt='(a)') &
         "CAUTION (SR checks): (n_mon_tot .eq. 0) .and. (f_thermost&
         &at_mode .eq. 2)."
    write(unit=*, fmt='(a)') &
         "        f_thermostat_mode = 3 is clearer."
  end if
  ! check validity of flags
  !========================
  if(.not.((f_wall_fix .eq. 0) .or. (f_wall_fix .eq. 1))) then
    write(unit=*, fmt='(a, i12, a)') &
         "ERROR (SR checks): Case for f_wall_fix=", f_wall_fix, &
         " not recognized."
    stop
  end if
  if(.not.((f_minimize .eq. 0) .or. (f_minimize .eq. 1))) then
    write(unit=*, fmt='(a, i12, a)') &
         "ERROR (SR checks): Case for f_minimize=", f_minimize, &
         " not recognized."
    stop
  end if
  if((f_cut_off .lt. 0) .or. (f_cut_off .gt. 3)) then
    write(unit=*, fmt='(a, i12, a)') &
         "ERROR (SR checks): Case for f_cut_off=", f_cut_off, &
         " not recognized."
    stop
  end if
  if((f_thermostat_mode .lt. 0) &
       .or. (f_thermostat_mode .gt. n_thermostat_modes-1)) then
    write(unit=*, fmt='(a, i12, a)') &
         "ERROR (SR checks): Case for f_thermostat_mode=", &
         f_thermostat_mode, " not recognized."
    stop
  end if
  if(.not.((f_friction_on_twall.eq.0).or.(f_friction_on_twall.eq.1))) then
    write(unit=*, fmt='(a, i12, a)') &
         "ERROR (SR checks): Case for f_friction_on_twall=", &
         f_friction_on_twall, " not recognized."
    stop
  end if
  ! sanity and consistency checks
  !==============================
  ! by convention n_cell_w_x >= n_cell_w_y
  if(n_cell_w_x .lt. n_cell_w_y) then
    write(unit=*, fmt='(a,i12,a,i12)') &
         "ERROR (SR checks): n_cell_w_x=", n_cell_w_x, &
         " .lt. n_cell_w_y=", n_cell_w_y
    write(unit=*, fmt='(a)') &
         "                   By convention we do not allow this"
    stop
  end if
  ! check the wall layout for the incommensurate case.  The ratio of
  ! n_cell_w_x / n_cell_w_y should be close to sqrt(3) in order not
  ! to distort the lattice too much when setting up the incommensurate walls.
  ! For commensurate walls any geometry is possible
  if(n_dim .eq. 3 .and. n_dim_pbc .eq. 2 &
       .and. abs(real(n_cell_w_x, kind=dp) &
       / real(n_cell_w_y, kind=dp) - sqrt3) .gt. 0.1_dp) then
    write(unit=*, fmt='(a)') &
         "CAUTION (SR checks): Walls are not very quadratic:"
    write(unit=*, fmt='(a, g13.6, a, g13.6)') &
         "        n_cell_w_x/n_cell_w_y=", &
         real(n_cell_w_x)/real(n_cell_w_y), &
         " n_cell_w_x/n_cell_w_y - sqrt(3)=", &
         real(n_cell_w_x)/real(n_cell_w_y) - sqrt(3.0)
    ! switch set correctly ?
    if(.not. l_walls_are_commensurate) then
      write(unit=*, fmt='(a, i4, a, i4)') "WARNING (SR checks): &
           &Reluctant to create walls with n_cell_w_x= ", &
           n_cell_w_x, " and n_cell_w_y= ", n_cell_w_y
      write(unit=*, fmt='(a)') &
           "      because they are not incommensurate enough for &
           &l_walls_are_commensurate = .FALSE."
    end if
  end if ! (good cell combination for incommensurate walls)
  ! check for the matching of presence of fluid and fluid interaction
  ! switches
  if( (n_mon_tot .eq. 0) .and. l_fluid_fluid_interaction) then
    write(unit=*, fmt='(a)') &
         "ERROR (SR checks): (n_mon_tot .eq. 0) .and. &
         &(l_fluid_fluid_interaction .eq. .TRUE.)"
    stop
  end if
  if( (n_mon_tot .eq. 0) .and. l_fluid_wall_interaction) then
    write(unit=*, fmt='(a)') &
```

```
       "ERROR (SR checks): (n_mon_tot .eq. 0) .and. &
       &(l_fluid_wall_interaction .eq. .TRUE.)"
  stop
end if
! check for calls to analysis routines
if( (n_mon_tot .eq. 0) .and. l_compute_fluid_displ_dist) then
  write(unit=*, fmt='(a)') &
       "ERROR (SR checks): (n_mon_tot .eq. 0) .and. &
       &(l_compute_fluid_displ_dist .eq. .TRUE.)"
  stop
end if
if( (n_mon_tot .eq. 0) .and. l_compute_fluid_vel_dist) then
  write(unit=*, fmt='(a)') &
       "ERROR (SR checks): (n_mon_tot .eq. 0) .and. &
       &(l_compute_fluid_vel_dist .eq. .TRUE.)"
  stop
end if
if( (n_mon_tot .eq. 0) .and. l_compute_fluid_accel_dist) then
  write(unit=*, fmt='(a)') &
       "ERROR (SR checks): (n_mon_tot .eq. 0) .and. &
       &(l_compute_fluid_accel_dist .eq. .TRUE.)"
  stop
end if
! sometimes we want to perform simulations where the fluid-fluid
! interaction is turned off for a reason
if( (n_mon_tot .ge. 2) .and. (.not. l_fluid_fluid_interaction) ) then
  write(unit=*, fmt='(a)') &
       "WARNING (SR checks): (n_mon_tot .ge. 2) .and. &
       &(.not. l_fluid_fluid_interaction)"
end if
if( (n_mon_tot .ge. 1) .and. (.not. l_fluid_wall_interaction) &
     .and. (n_dim .eq. n_dim_pbc +1)) then
  write(unit=*, fmt='(a)') &
       "ERROR (SR checks): (n_mon_tot .ge. 1) .and. &
       &(.not. l_fluid_wall_interaction) .and. &
       &(n_dim .eq. n_dim_pbc +1)"
end if
! check matching of intramolecular interaction switch and chainlength
if( (n_chain .ge. 1) .and. (n_mon .ge. 2) .and. &
     (.not. l_intra_molec_interaction) ) then
  write(unit=*, fmt='(a)') &
       "ERROR (SR checks): (n_chain .ge. 1) .and. (n_mon .ge. 2) &
       &.and. (.not. l_intra_molec_interaction)"
  write(unit=*, fmt='(a)') &
       "       Switch on intra-molecular interaction."
  stop
end if
if( (n_mon .le. 1) .and. (l_intra_molec_interaction) ) then
  write(unit=*, fmt='(a)') &
       "CAUTION (SR checks): (n_mon .le. 1) .and. (l_intra_molec&
       &_interaction)"
  write(unit=*, fmt='(a)') &
       "         intra-molecular interaction can be switched off"
end if
! check if there are two or more force ramps
counter = 0
do i_dim = 1, n_dim
  if(f_twall(i_dim).eq.force_mode &
       .and. ramp_force_twall(i_dim).ne.0.0_dp) then
    counter = counter + 1
  end if
end do
if(counter .gt. 1) then
  write(unit=*, fmt='(a)') &
       "ERROR (SR checks): There's a force ramp in more than one &
       &direction."
  write(unit=*, fmt='(a)') &
       "       The output routines in SR control have to be modified."
  stop
end if
if(n_dim_pbc .ne. 2 .and. l_compute_ft_density) then
  write(unit=*, fmt='(a)') &
       "ERROR (SR checks): n_dim_pbc .ne. 2 .and. l_compute_ft_density"
  write(unit=*, fmt='(a)') &
       "       Fourier transformation of particle densities does not work &
       &for 1-d presently"
  stop
end if
if((n_mon .eq. 0) .and. l_compute_press_tens) then
  write(unit=*, fmt='(a)') "ERROR (SR checks): &
       &(n_mon .eq. 0) .and. l_compute_press_tens):"
  write(unit=*, fmt='(a)') &
       "       Can't compute pressure tensor of non-existing fluid"
  stop
end if
if(.not. l_compute_press_tens .and. l_compute_press_tens_wall_contr) then
  write(unit=*, fmt='(a)') "ERROR (SR checks): &
       &.not. l_compute_press_tens .and. l_compute_press_tens_wall_contr"
  stop
end if
if(l_compute_cm_fluid .and. (n_mon_tot .eq. 0)) then
  write(unit=*, fmt='(a)') "ERROR (SR checks): &
       &l_compute_cm_fluid .and. (n_mon_tot .eq. )"
  stop
end if
if(l_use_lab_system_for_energies .and. (f_thermostat_mode .ne. 0 &
     .or. f_thermostat_mode .ne. 1 .or. f_thermostat_mode .ne. 4 )) then
  write(unit=*, fmt='(a)') "ERROR (SR checks): &
       &l_use_lab_system_for_energies .and. (f_thermostat_mode .ne. 0 &
       &.or. f_thermostat_mode .ne. 1 .or. f_thermostat_mode .ne. 4 )"
  write(unit=*, fmt='(a)') "      This might create problems"
  stop
end if
if(l_write_energy_every_mds .and. .not. l_use_lab_system_for_energies) then
  write(unit=*, fmt='(a)') "ERROR (SR checks): &
       &l_write_energy_every_mds .and. .not. l_use_lab_system_for_energies"
  write(unit=*, fmt='(a)') "      This might create problems"
  stop
end if
! we don't want to blow the disk with the energies at every MD step for
! long simulations
if(l_write_energy_every_mds .and. &
     (n_obser + n_relax - s_time) .gt. 10**6) then
  write(unit=*, fmt='(a)') "ERROR (SR checks): &
       &l_write_energy_every_mds .and. more then 10**6 MDS"
  write(unit=*, fmt='(a)') &
       "       Go into SR checks and remove this hook if you feel like it."
  stop
end if
 if((n_dim .eq. n_dim_pbc +1) .and. &
     .not. l_intra_wall_interaction) then
  write(unit=*, fmt='(a)') "ERROR (SR checks): &
       & (n_dim .eq. n_dim_pbc +1) .and. &
       &.not. l_intra_wall_interaction"
  stop
end if
! checks for MD simulation without walls
if(n_dim .eq. n_dim_pbc) then

  if(n_cell_w_x .ne. 1 .or. n_cell_w_y .ne. 1) then
    write(unit=*, fmt='(a)') "ERROR (SR checks): &
         & n_dim .eq. n_dim_pbc), so there must be only one wall &
         &cell, but"
    write(unit=*, fmt='(2(a,i12))') "       n_cell_w_x=", n_cell_w_x, &
         ", n_cell_w_y=", n_cell_w_y
    stop
  end if
  if(f_wall_fix .ne. 1) then
    write(unit=*, fmt='(a)') "ERROR (SR checks): &
         &(n_dim .eq. n_dim_pbc), but f_wall_fix.ne.1 (as our convention)"
    stop
  end if
  if(n_mon_tot .le. 0) then
    write(unit=*, fmt='(a)') "ERROR (SR checks): &
         &(n_dim .eq. n_dim_pbc), but there's no fluid to simulate"
    stop
  end if
  if(l_2d_flat_setup) then
    write(unit=*, fmt='(a)') "ERROR (SR checks): &
         &(n_dim .eq. n_dim_pbc) .and. l_2d_flat_setup makes no sense"
    write(unit=*, fmt='(a)') "       (l_2d_flat_setup is ignored)"
    stop
  end if
  if(l_allow_wall_wall_interaction) then
    write(unit=*, fmt='(a)') "ERROR (SR checks): &
         &(n_dim .eq. n_dim_pbc) .and. l_allow_wall_wall_interaction"
    stop
  end if
  if(l_fluid_wall_interaction) then
    write(unit=*, fmt='(a)') "ERROR (SR checks): &
         &(n_dim .eq. n_dim_pbc) .and. l_fluid_wall_interaction"
    stop
  end if
  if(l_intra_wall_interaction) then
    write(unit=*, fmt='(a)') "ERROR (SR checks): &
         &(n_dim .eq. n_dim_pbc) .and. l_intra_wall_interaction"
    stop
  end if
  ! we don't want the top wall to be expected to be moved
  if(maxval(abs(mode_parameter_one(1:n_dim))) .gt. 0.0_dp) then
    write(unit=*, fmt='(a)') "ERROR (SR checks): &
         &(n_dim .eq. n_dim_pbc) .and. mode_parameters_one(:) not 0"
    stop
  end if
  if(maxval(abs(mode_parameter_two(1:n_dim))) .gt. 0.0_dp) then
    write(unit=*, fmt='(a)') "ERROR (SR checks): &
         &(n_dim .eq. n_dim_pbc) .and. mode_parameters_two(:) not 0"
    stop
  end if
end if ! (n_dim .eq. n_dim_pbc)
if(n_dim_pbc .ne. 2 .and. l_forbid_xy_bond_crossings) then
  write(unit=*, fmt='(a)') "WARNING (SR checks): &
       &n_dim_pbc .ne. 2 .and. l_forbid_xy_bond_crossings."
  write(unit=*, fmt='(a)') &
       "          Will ignore l_forbid_xy_bond_crossings "
end if
if((n_mon .lt. 2) .and. l_forbid_xy_bond_crossings) then
  write(unit=*, fmt='(a)') "ERROR (SR checks): &
       &n_mon .lt. 2 .and. l_forbid_xy_bond_crossings (there are no bonds)"
  stop
end if
end subroutine checks
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! initializes the binning:
! checks if skin thickness is compatible with interaction ranges, computes
! optimal number of bins and allocates memory
subroutine init_binning(l_call_before_conf)
  ! if this SR was called before reading a configuration from disk
  logical, intent(in) :: l_call_before_conf
  integer :: i_dim
  ! loops over particle types
  integer :: i_type, j_type
  if(l_debug_init_binning) then
    print *, "DEBUG: Entering subroutine init_binning"
  end if
  ! if there are walls
  if(n_dim .eq. n_dim_pbc+1) then
    if(.not. l_call_before_conf) then
      ! obtain film thickness from configuration
      ! the fluid film might be quite thick, so we try to find the optimal
      ! number of binning boxes for the fluid in z-direction.
      ! The walls are "flat" anyways and fit in one layer of binning boxes
      ! In case there is no fluid at all, we don't allocate fluid bins.
      ! find the range of z-positions of the fluid particles
      if(n_mon_tot .gt. 0) then
        highest_fluid_z = maxval(r0(1:n_mon_tot, n_dim))
        lowest_fluid_z = minval(r0(1:n_mon_tot, n_dim))
        ! sanity check
        ! for one fluid atom the equality is okay.
        if(highest_fluid_z .lt. lowest_fluid_z) then
          write(unit=*, fmt='(a)') &
               "ERROR (SR init_binning): highest_fluid_z < lowest_fluid_z:"
          write(unit=*, fmt='(2(a, g20.13))') &
               "       highest_fluid_z =", highest_fluid_z, &
               " lowest_fluid_z= ", lowest_fluid_z
          stop
        end if
        write(unit=*, fmt='(a,g13.6,a,g13.6)') &
             "MESSAGE (SR init_binning): highest_fluid_z= ", &
             highest_fluid_z, "lowest_fluid_z= ", lowest_fluid_z
      else ! n_mon_tot .le. 0
        write(unit=*, fmt='(a)') &
             "MESSAGE (SR init_binning): No fluid present"
        highest_fluid_z = 0.0_dp
        lowest_fluid_z = 0.0_dp
      end if
    else
      ! take initial values from parameter file
      highest_fluid_z = z_space_wall
      lowest_fluid_z  = 0.0_dp
    end if
    ! boundary(n_dim) is in this case the film thickness
    boundary(n_dim) = highest_fluid_z - lowest_fluid_z
  end if ! there are walls
  ! find number of bins in directions with periodic boundary conditions
  do i_dim = 1, n_dim_pbc
    n_bin(i_dim) = int(boundary(i_dim)/(max_interaction_range + skin))
    if(n_bin(i_dim) .eq. 0) then
      n_bin(i_dim) = 1
      r_bin(i_dim) = max_interaction_range + skin
    else ! everything is normal
      ! widths of bin can be computed from number of bins
      r_bin(i_dim) = boundary(i_dim)/real(n_bin(i_dim), kind=dp)
    end if
  end do
  ! volume in periodic boundaries
  pbc_volume = 1.0_dp
  do i_dim = 1, n_dim_pbc
    pbc_volume = pbc_volume * boundary(i_dim)
  end do
  if(n_dim .eq. n_dim_pbc+1) then
```

```
    if(n_mon_tot .gt. 0) then
      ! in "z"-direction there are no periodic boundary conditions so we can
      ! always use the smallest possible bin width
      ! (max_interaction_range + skin).  If we prefer more evenly filled bins
      ! (see SR binning3d) the extra allocated bin layer in "z" won't hurt.
      n_bin(n_dim) = int(boundary(n_dim)/(max_interaction_range + skin)) + 1
      r_bin(n_dim) = max_interaction_range + skin
    else
      n_bin(n_dim) = 1
      r_bin(n_dim) = 0.0_dp
    end if
  end if
  ! we need at least one bin in each dimension
  n_bin(n_dim+1:n_dim_max) = 1
  ! n_bin_used might get changed in SR binning3d (but we need to initialize
  ! in case we need the values in module mcfluid)
  n_bin_used(:) = n_bin(:)
  ! half boundaries
  half_bound(:) = 0.5_dp * boundary(:)
  ! select flags to decide which binning boxes to look at
  do i_dim = 1, n_dim_max
    if(n_bin(i_dim).ge.3) then
      delta_bin(i_dim) = 1
    else if (n_bin(i_dim).eq.2) then
      delta_bin(i_dim) = 0
    else if (n_bin(i_dim).eq.1) then
      delta_bin(i_dim) = -1
    else
      write(unit=*, fmt='(a,i1,a,i10)') &
           "ERROR: (SR init_binning): n_bin(", i_dim, ")= ", n_bin(i_dim)
      stop
    end if
  end do
  write(unit=*, fmt='(a, g22.15)') &
       "MESSAGE (SR init_binning): max_interaction_range + skin=", &
       max_interaction_range + skin
  write(unit=*, fmt='(a)') &
       "MESSAGE (SR init_binning): Number of binning boxes"
  write(unit=*, fmt='(a, 3i5)') &
       "                           ", n_bin(1: n_dim_max)
  write(unit=*, fmt='(a)') &
       "MESSAGE (SR init_binning): boundary(:):"
  write(unit=*, advance="no", fmt='(a)') "                                "
  do i_dim=1, n_dim
    write(unit=*, advance="no", fmt='(g13.6)') boundary(i_dim)
  end do
  write(unit=*, fmt=*) ""
  write(unit=*, fmt='(a, g13.6, a)') &
       "MESSAGE (SR init_binning): Fluid density in periodic volume:", &
       real(n_mon_tot, kind=dp) / pbc_volume
  write(unit=*, fmt='(a)') &
       "MESSAGE (SR init_binning): Widths of binning boxes"
  write(unit=*, advance="no", fmt='(a)') "                                "
  do i_dim=1, n_dim
    write(unit=*, advance="no", fmt='(g13.6)') r_bin(i_dim)
  end do
  write(unit=*, fmt=*) ""
  ! check if interaction range plus skin are not bigger than width of bin
  ! if everything works fine this test should never complain
  do i_type = 1, n_type
    do j_type = i_type, n_type
      do i_dim=1, n_dim
        if(neigh_list_rad_2(j_type, i_type).gt.r_bin(i_dim)**2 &
             + 10.0_dp**(-precision(1.0_dp)+1))then
          if((i_dim .le. 2) .or. (n_mon_tot .gt. 0)) then
            write(unit=*, fmt='(a, i1, a, i1, a)') &
                 "WARNING (SR init_binning): For particle pairs of type (", &
                 j_type, ", ", i_type, ") binning problematic:"
            write(unit=*, fmt='(a, i2, a, i2, a, g22.15, a, i2, a, g22.15)')&
                 "neigh_list_rad_2(", j_type, ",", i_type, ")=", &
                 neigh_list_rad_2(j_type, i_type), ".gt.r_bin(", &
                 i_dim, ")**2=", r_bin(i_dim)**2
          end if
        end if
      end do
    end do
  end do

  ! allocate memory for bins
  !-------------------------
  ! if there's no fluid we don't allocate fluid bins
  if(n_mon_tot .gt. 0) then
    allocate(bin_fluid(0:n_bin(1)-1, 0:n_bin(2)-1, 0:n_bin(3)-1, 0:n_bin_fl))
  end if
  if(n_dim .eq. n_dim_pbc +1) then
    ! only one bin layer in z-direction for wall binning
    allocate(bin_twall(0:n_bin(1)-1, 0:n_bin(2)-1, 0:n_bin_wa))
    allocate(bin_bwall(0:n_bin(1)-1, 0:n_bin(2)-1, 0:n_bin_wa))
  end if
  !=================================================================!
  ! bins are initialized in subroutine binning or in SR conf_default !
  !=================================================================!
  if(l_debug_init_binning) then
    print *, "DEBUG: Leaving subroutine init_binning"
  end if
end subroutine init_binning
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! frees the memory taken by the bins and the neighbor lists
subroutine free_bin_memory
  if(allocated(bin_twall)) deallocate(bin_twall)
  if(allocated(bin_bwall)) deallocate(bin_bwall)
  if(allocated(bin_fluid)) deallocate(bin_fluid)
  if(allocated(ff_list))   deallocate(ff_list)
  if(allocated(fw_list))   deallocate(fw_list)
  if(allocated(ww_list))   deallocate(ww_list)
end subroutine free_bin_memory
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! reads a configuration if it exists or calls a subroutine to create a con-
! figuration
subroutine init_config(old_configuration_file)
  character(len=*), intent(in) :: old_configuration_file
  integer :: i_dim
  integer, parameter :: in_file = 10
  integer            :: io_status
  ! loops
  integer :: i_type, i_part, i_order
  ! try to open file containing configuration data
  open(unit=in_file, file=old_configuration_file, &
       iostat=io_status, action="read", status="old");
  ! close file
  close(in_file)
  if(io_status .ne. 0) then
    write(unit=*, fmt='(3a)') &
         "CAUTION: Could not read configuration file >>", &
         old_configuration_file, "<<"
    ! initialize ramp for effective minimum distance for creating a new
    ! configuration.  If we read a configuration we assume that the stored
    ! configuration doesn't need a r_2_min > 0
    if(r_2_min_time .gt. n_relax * dt) then
      write(unit=*, fmt='(a)') &
           "CAUTION (SR init_config):  Ramp for effective minimum distance &
           &longer than relaxation time:"
      write(unit=*, fmt='(a, e13.6, a, e13.6)') &
           "                           r_2_min_time=", r_2_min_time, &
           " .gt. n_relax * dt=", n_relax * dt
    end if
    r_2_min = r_2_min_init
    l_r_2_min_finite = .TRUE.
    ! create default configuration
    write(unit=*, fmt='(a)') "MESSAGE (SR init_config): &
         &Creating default configuration"
    ! initialize binning
    call init_binning(.TRUE.)
    ! create new configuration from scratch
    call conf_default
  else ! there's a configuration
    if(l_use_config_rng) then
      ! use read_configuration_rng_2d or 3d for compatibility
      call read_configuration_rng(old_configuration_file)
    else
      call read_configuration(old_configuration_file)
    end if
    ! if we start a new simulation with an old configuration, we shift the
    ! wall atomes back in the simulation box and also the chains centers of
    ! mass by changing the pbc counters
    if(s_time .eq. 0) call shift_chains_in_box
    ! check if chains were ripped by folding
    call chain_sens_fold
    ! initialize binning
    call init_binning(.FALSE.)
    ! create neighbor lists so that the interaction routines work properly
    call binning3d
    ! reset ramp for r_2_min, because we just read a configuration
    r_2_min_init = 0.0_dp
    r_2_min      = 0.0_dp
    r_2_min_time = 0.0_dp
    l_r_2_min_finite = .FALSE.
    ! check for presence of bond crossings in xy-plane.
    if(l_forbid_xy_bond_crossings .and. n_dim_pbc .eq. 2) then
      if(.not. check_xy_bond_crossings_all()) then
        write(unit=*, fmt='(a, i12)') "ERROR (SR init_config): &
             &xy-plane bond crossings detected at MDS", i_time
        call fluid_positions_out("xy-bonds-cross_t", i_time)
        stop
      else
        write(unit=*, fmt='(a)') &
             "MESSAGE (SR init_config): No xy-plane bond crossings detected."
      end if
    end if
  end if ! old configuration not readable
  !-----------------------------------------------------------------------
  ! at this point we have a configuration
  !-----------------------------------------------------------------------
  ! print effective potential ramping status (could have been changed by the
  ! MC fluid setup routines)
  write(unit=*, fmt='(a, 12)') &
       "MESSAGE (SR init_config): MD-potential ramp switch &
       &l_r_2_min_finite =", l_r_2_min_finite
  ! reinitialize derivatives of top wall position if in velocity mode
  do i_dim = 1, n_dim
    if(f_twall(i_dim).eq.velocity_mode) then
      rx_twall(i_dim, 1) = v_spring_twall(i_dim)*dt
      do i_order = 2, n_order
        rx_twall(i_dim, i_order) = 0.0_dp
      end do
    end if
  end do
  ! store masses in array
  do i_part = 1, n_part
    mass(i_part) = mass_type(type(i_part))
  end do
  if(f_minimize.eq.1) then ! freeze the system
    write(unit=*, fmt='(a)') &
         "MESSAGE (SR init_config): Freezing the system, maybe overriding &
         &parameters."
    ! set all derivatives to 0.0
    rx(:, :, :)    = 0.0_dp
    rx_twall(:, :) = 0.0_dp
    if(f_thermostat_mode .ne. 1) then
      write(unit=*, fmt='(a, i2, a)') &
           "MESSAGE (SR init_config): Set thermostat mode &
           &f_thermostat_mode=", f_thermostat_mode, " to 1"
      write(unit=*, fmt='(a)') &
           "        (no random forces, just friction acting on all moving &
           &particles)"
      ! switch to thermostat mode for this case
      f_thermostat_mode = 1
    end if
    ! chose a different friction constant
    do i_type = 1, n_type
      friction(i_type) = 0.05_dp / dt
      ! This is the maximum amount of drag friction for which
      ! the trajectories can be integrated in a stable fashion.
      ! Some modes will be underdamped!
    end do
    ! check if top wall is released, if not do so
    do i_dim = 1, n_dim
      select case (f_twall(i_dim))
      case(velocity_mode)
        if(v_spring_twall(i_dim) .ne. 0.0_dp) then
          write(unit=*, fmt='(a, i1, a, e14.6, a)') &
               "CAUTION (SR init_config): v_spring_twall(", i_dim, &
               ") =", v_spring_twall(i_dim), &
               " not 0.0, so set to 0.0 for f_minimize=1"
          v_spring_twall(i_dim) = 0.0_dp
        end if
        if(k_spring_twall(i_dim) .ne. 0.0_dp) then
          write(unit=*, fmt='(a, i1, a, e14.6, a)') &
               "CAUTION (SR init_config): k_spring_twall(", i_dim, &
               ") =", k_spring_twall(i_dim), &
               ", which is meaningless in this mode, not 0.0, so set to 0.0"
          k_spring_twall(i_dim) = 0.0_dp
        end if
      case(force_mode)
        if(ramp_force_twall(i_dim) .ne. 0.0_dp) then
          write(unit=*, fmt='(a, i1, a, e14.6, a)') &
               "CAUTION (SR init_config): ramp_force_twall(", i_dim, &
               ") =", ramp_force_twall(i_dim), &
               " not 0.0, so set to 0.0 for f_minimize=1"
          ramp_force_twall(i_dim) = 0.0_dp
        end if
        ! a force in x or y direction would prevent equilibrium
        if((i_dim .ne. n_dim).and.(ext_force_twall(i_dim) .ne. 0.0_dp)) then
          write(unit=*, fmt='(a, i1, a, e14.6, a)') &
               "CAUTION (SR init_config): ext_force_twall(", i_dim, &
               ") =", ext_force_twall(i_dim), &
               " not 0.0, so set to 0.0 for f_minimize=1"
          ext_force_twall(i_dim) = 0.0_dp
        end if
      case(spring_mode)
```

```
      if(v_spring_twall(i_dim) .ne. 0.0_dp) then
        write(unit=*, fmt='(a, i1, a, e14.6, a)') &
             "CAUTION (SR init_config): v_spring_twall(", i_dim, &
             ") =", v_spring_twall(i_dim), &
             " not 0.0, so set to 0.0 for f_minimize=1"
        v_spring_twall(i_dim) = 0.0_dp
      end if
      ! a force in x or y direction would prevent equilibrium, but we allow
      ! fixed springs to stabilize the system.
      if((i_dim .ne. n_dim) .and. k_spring_twall(i_dim) .ne. 0.0_dp) then
        write(unit=*, fmt='(a, i1, a, e14.6, a)') &
             "CAUTION (SR init_config): k_spring_twall(", i_dim, &
             ") =", k_spring_twall(i_dim), &
             ", not 0.0, so system might be shifted by springs."
      end if
    case default
      write(unit=*, fmt='(a, i1, a, i2, a)') &
           "ERROR (SR init_config): case f_twall(", i_dim, ")=", &
           f_twall(i_dim), " not recognized."
      stop
    end select
  end do
end if ! (f_minimize.eq.1)

if(n_dim .eq. n_dim_pbc) then
  if(maxval(abs(rx_twall(:, :))) .gt. 0.0_dp) then
    write(unit=*, fmt='(a)') "WARNING (SR checks): &
         & (n_dim .eq. n_dim_pbc) .and. maxval(abs(rx_twall(:, :))) > 0"
    write(unit=*, fmt='(a)') &
         "                              Resetting rx_twall(:, :) = 0"
    rx_twall(:, :) = 0.0_dp
  end if
end if

end subroutine init_config
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! creates default configuration.  Calls a lot of other functions...
subroutine conf_default
  ! loop variables
  integer :: i_chain, i_cell_w_x, i_cell_w_y, i_wall, i_part
  integer :: i_dim
  ! accounting of chains
  logical :: l_new_chain_inserted
  integer :: call_chain_insert_counter
  ! maximal number of calls to CBMC insertion of chain before giving up
  integer, parameter :: n_max_insert_calls = 10000
  if(l_debug_conf_default) then
    print *, "DEBUG: Entering subroutine conf_default"
  end if
  ! initialize particle and wall positions
  r0(:, :) = 1.0E100_dp
  ! initialize lower left corner of top wall
  r0_twall(:) = 0.0_dp
  ! initialize position of spring
  r0_spring_twall(:) = 0.0_dp
  if(n_dim .eq. n_dim_pbc+1) then
    ! top wall has initial distance of z_space_wall
    r0_twall(n_dim) = z_space_wall
  end if
  ! initialize force ramp gradient
  s_force_grad(:) = 1
  ! define which particle belongs to which species
  type(1:n_mon_tot) = 1 ! fluid is type 1
  ! define wall atoms always to be of type "n_type"
  type(n_mon_tot+1:n_part) = n_type
  ! set all derivatives to 0.0
  rx(:, :, :)    = 0.0_dp
  rx_twall(:, :) = 0.0_dp
  ! define equilibrium and start up sites of wall atoms
  !=====================================================
  if(n_dim .eq. 3 .and. n_dim_pbc .eq. 2) then
    call wall_setup_2d
  else if(n_dim .eq. 2 .and. n_dim_pbc .eq. 1) then
    call wall_setup_1d
  else
    ! set all wall particles at the origin
    r0(n_mon_tot+1:n_part, :) = 0.0_dp
    r_wall_equi(1:n_wall, :)  = 0.0_dp
  end if
  if(n_dim .eq. n_dim_pbc+1) then
    ! bin the wall particles to speed up interaction computations
    call bin_wall_particles
  end if
  ! define start up sites of monomers
  !==================================
  ! maybe we don't have fluid particles (and fluid bins are not allocated)
  ! so we can't initialize bin_fluid here
  if(l_use_mc_fluid_setup) then
    call mc_fluid_setup
    ! uncomment for using the V1.7 CBMC fluid setup without MC equilibration
    ! call mc_fluid_setup_cbmc
  else
    ! create each chain as a simple random walk
    do i_chain = 1, n_chain
      select case(n_dim)
      case(3)
        call random_walk_chain_3d(type(1+(i_chain-1)*n_mon:i_chain*n_mon), &
             r0(1+(i_chain-1)*n_mon:i_chain*n_mon, :))
      case(2)
        call random_walk_chain_2d(type(1+(i_chain-1)*n_mon:i_chain*n_mon), &
             r0(1+(i_chain-1)*n_mon:i_chain*n_mon, :))
      case default
        write(unit=*, fmt='(a,i1,a)') "ERROR (SR conf_default): &
             &Random walk chain setup for n_dim=", n_dim, " not implemented"
        stop
      end select
      ! apply periodic boundary conditions if necessary for the whole chain.
      ! If we applied the pbc during creation of the chain, and particle i
      ! gets folded in the simulation box, particle i+1 would not be folded.
      ! This seems a bit artificial, although the simulation runs identically
      ! with either method, as we are working with equivalence classes of
      ! particles anyways.
      do i_part = 1 + (i_chain-1)*n_mon, i_chain*n_mon
        do i_dim = 1, n_dim_pbc
          if(r0(i_part,i_dim).gt.boundary(i_dim)) then
            r0(i_part,i_dim) = r0(i_part,i_dim) - boundary(i_dim)
            pbc_count(i_part,i_dim) = pbc_count(i_part,i_dim) + 1
          else if(r0(i_part,i_dim).lt.0.0_dp) then
            r0(i_part,i_dim) = r0(i_part,i_dim) + boundary(i_dim)
            pbc_count(i_part,i_dim) = pbc_count(i_part,i_dim) - 1
          end if
        end do
      end do
      ! define unfolded coordinates of this chain
      do i_dim = 1, n_dim_pbc
        r0_unfolded((i_chain -1) * n_mon +1:i_chain * n_mon, i_dim) &
             = r0((i_chain -1) * n_mon +1:i_chain * n_mon, i_dim) &
             + real(pbc_count((i_chain -1) * n_mon +1:i_chain*n_mon,i_dim), &
             kind=dp) * boundary(i_dim)
      end do
      r0_unfolded((i_chain -1) * n_mon +1:i_chain*n_mon, n_dim_pbc+1:n_dim) &
           = r0((i_chain -1) * n_mon +1:i_chain*n_mon, n_dim_pbc+1:n_dim)
    end do ! i_chain = 1, n_chain
  end if ! l_use_mc_fluid_setup
  ! check for presence of bond crossings in xy-plane.
  if(l_forbid_xy_bond_crossings .and. n_dim_pbc .eq. 2) then
    if(.not. check_xy_bond_crossings_all()) then
      write(unit=*, fmt='(a, i12)') "ERROR (SR conf_default): &
           &xy-plane bond crossings detected at MDS", i_time
      call fluid_positions_out("xy-bonds-cross_t", i_time)
      stop
    else
      write(unit=*, fmt='(a)') &
           "MESSAGE (SR conf_default): No xy-plane bond crossings detected."
    end if
  end if
  ! thermal velocities will not be attributed to monomers here.
  ! The thermostat will do that later.
  ! initialize algorithm
  !=====================
  ! if there is Frenkel-Kontorova interaction, initialize
  if(((k_spring_wall_fre_kon_linear .ne. 0.0_dp) .or. &
       (k_spring_wall_fre_kon_vector .ne. 0.0_dp) .or. &
       (f_thermostat_mode .eq. 6)) .and. (n_dim .eq. n_dim_pbc+1)) then
    call init_fk_model
  end if
  call binning3d
  call thermostat
  ! compute forces
  if(l_fluid_fluid_interaction) call fluid_fluid
  if(l_fluid_wall_interaction)  call fluid_wall
  if(l_allow_wall_wall_interaction .and. l_wall_wall_interaction) &
       call wall_wall
  if(l_intra_molec_interaction) call intra_molec
  if(l_intra_wall_interaction)  call intra_wall
  ! convert (deterministic) forces into accelerations
  do i_dim = 1, n_dim
    do i_part = 1, n_moving
      rx(i_part, i_dim, 2) = &
           (force_determ(i_part,i_dim)/mass_type(type(i_part))) &
           * (dt_2/2.0_dp)
    end do
    rx_twall(i_dim, 2) = &
         (total_force_twall(i_dim)/mass_twall) * (dt_2/2.0_dp)
  end do
  if(l_debug_conf_default) then
    print *, "DEBUG: Leaving subroutine conf_default"
  end if
  !----------------------------------------------------------------------------
contains
  !----------------------------------------------------------------------------
  subroutine wall_setup_2d
    ! give error if top and bottom wall have different size
    if(n_top_wall .ne. n_bottom_wall) then
      write(unit=*, fmt='(a)') &
           "ERROR (SR wall_setup_2d): Different wall sizes not implemented"
      stop
    end if
    i_wall = 1
    ! create top wall
    do i_cell_w_x = 0, n_cell_w_x-1
      do i_cell_w_y = 0, n_cell_w_y-1
        ! x coordinates
        r_wall_equi(i_wall,  1) = real(i_cell_w_x, kind=dp) * x_space
        r_wall_equi(i_wall+1,1) = r_wall_equi(i_wall,1) + x_space/2.0_dp
        ! y coordinates
        r_wall_equi(i_wall,  2) = real(i_cell_w_y, kind=dp) * y_space
        r_wall_equi(i_wall+1,2) = r_wall_equi(i_wall,2) + y_space/2.0_dp
        ! each cell contains two particles
        i_wall = i_wall + 2
      end do
    end do
    if(l_walls_are_commensurate) then
      ! bottom wall is identical
      i_wall = n_top_wall+1
      do i_cell_w_x = 0, n_cell_w_x-1
        do i_cell_w_y = 0, n_cell_w_y-1
          r_wall_equi(i_wall,  1) = real(i_cell_w_x, kind=dp) * x_space
          r_wall_equi(i_wall+1,1) = r_wall_equi(i_wall,1) + x_space/2.0_dp
          r_wall_equi(i_wall,  2) = real(i_cell_w_y, kind=dp) * y_space
          r_wall_equi(i_wall+1,2) = r_wall_equi(i_wall,2) + y_space/2.0_dp
          ! each cell contains two particles
          i_wall = i_wall + 2
        end do
      end do
    else ! incommensurate surfaces
      ! i.e. identical surfaces, but the bottom wall is x-y mirrored.
      ! This is why we want n_cell_w_x / n_cell_w_y close to sqrt(3) to get
      ! nearly quadratic surfaces.
      i_wall = n_top_wall+1
      do i_cell_w_x = 0, n_cell_w_x-1
        do i_cell_w_y = 0, n_cell_w_y-1
          ! mirror x-y for incommensurate bottom wall
          r_wall_equi(i_wall,  1) = real(i_cell_w_y, kind=dp) * y_space
          r_wall_equi(i_wall+1,1) = r_wall_equi(i_wall, 1) + y_space/2.0_dp
          r_wall_equi(i_wall,  2) = real(i_cell_w_x, kind=dp) * x_space
          r_wall_equi(i_wall+1,2) = r_wall_equi(i_wall, 2) + x_space/2.0_dp
          i_wall = i_wall + 2
        end do
      end do
    end if
    ! define z-components
    do i_wall = 1, n_top_wall ! top wall
      r_wall_equi(i_wall, n_dim) = z_space_wall
    end do
    do i_wall = n_top_wall+1, n_wall ! bottom wall
      r_wall_equi(i_wall, n_dim) = 0.0_dp
    end do
    ! write equilibrium positions onto actual positions
    do i_wall = 1, n_wall
      i_part = i_wall + n_mon_tot
      r0(i_part, :) = r_wall_equi(i_wall, :)
    end do
  end subroutine wall_setup_2d
  !----------------------------------------------------------------------------
  !----------------------------------------------------------------------------
  subroutine wall_setup_1d
    ! y_space = n_top_wall / n_bottom_wall
    do i_wall = 1, n_top_wall
      ! x coordinates
      r_wall_equi(i_wall, 1) = real(i_wall -1, kind=dp) * x_space
    end do
    do i_wall = n_top_wall +1, n_wall
      ! x coordinates
      r_wall_equi(i_wall, 1) = real(i_wall - n_top_wall-1, kind=dp) * y_space
    end do
    ! define "z"-components
    r_wall_equi(1:n_top_wall, n_dim) = z_space_wall   ! top wall
    r_wall_equi(n_top_wall +1:n_wall, n_dim) = 0.0_dp ! bottom wall
    ! write equilibrium positions onto actual positions
    do i_wall = 1, n_wall
      i_part = i_wall + n_mon_tot
      r0(i_part, :) = r_wall_equi(i_wall, :)
    end do
  end subroutine wall_setup_1d
  !----------------------------------------------------------------------------
```

```
end subroutine conf_default
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! creates a single chain (or simply positions a monomer) as random walk in
! three dimensions.
subroutine random_walk_chain_3d(monomer_types, chain_positions)
  integer, dimension(n_mon), intent(in) :: monomer_types
  real(kind=dp), dimension(n_mon, n_dim), intent(out) :: chain_positions
  integer :: i_dim
  ! random walk jump steps in each direction
  real(kind=dp), dimension(n_dim) :: rw_step
  ! angles describing direction of random walk step and step length
  real(kind=dp) :: rw_phi, rw_theta, rw_step_length
  ! monomer position in chain
  integer :: i_mon_pos
  ! minimal absolute distance from the wall atoms we enforce when setting
  ! up configuration
  real(kind=dp) :: min_initial_wall_distance
  ! measure minimal initial wall distace in units of sigma
  min_initial_wall_distance = 0.8_dp * minval(sigma(:, :))
  if(n_dim .eq. n_dim_pbc+1) then
    ! sanity check
    if(2.0_dp * min_initial_wall_distance .ge. boundary(n_dim)) then
      write(unit=*, fmt='(2(a, g13.6))') &
           "ERROR (SR random_walk_chain): 2*min_initial_wall_distance=", &
           2.0_dp * min_initial_wall_distance, &
           " .le. boundary(n_dim)=", boundary(n_dim)
      stop
    end if
    if(l_debug_rwc_3d) then
      write(unit=*, fmt='(a, g13.6)') &
           "DEBUG: (SR random_walk_chain): min_initial_wall_distance = ", &
           min_initial_wall_distance
    end if
  else
    min_initial_wall_distance = 0.0_dp
  end if
  ! specify position of first monomer in chain
  i_mon_pos = 1
  call ranlux(random, n_dim)
  ! x-y coordinates
  do i_dim = 1, n_dim-1
    chain_positions(i_mon_pos, i_dim) = random(i_dim) * boundary(i_dim)
  end do
  ! boundary(n_dim) is initialized with the initial wall separation
  chain_positions(i_mon_pos, n_dim) = min_initial_wall_distance &
       + (boundary(n_dim) - 2.0_dp * min_initial_wall_distance) &
       * random(n_dim)
  ! generate random walk for next monomers on chain
  do i_mon_pos = 2, n_mon
    if(l_debug_rwc_3d) then
      write(unit=*, fmt='(2(a, i5), a, 3g10.3)') &
           "DEBUG: Starting random walk step from i_mon_pos=", i_mon_pos-1, &
           " to i_mon_pos=", i_mon_pos, " starting from: ", &
           chain_positions(i_mon_pos-1, :)
    end if
    do ! try random walk steps unless z-coordinate is in right range
      ! define random walk direction
      call ranlux(random, n_dim-1)
      rw_phi   = twopi * random(1)
      rw_theta =    pi * random(2)
      ! define the random walk step
      rw_step_length = bond_length(monomer_types(i_mon_pos-1), &
           monomer_types(i_mon_pos))
      rw_step(1)     = rw_step_length * cos(rw_phi) * sin(rw_theta)
      rw_step(2)     = rw_step_length * sin(rw_phi) * sin(rw_theta)
      rw_step(n_dim) = rw_step_length * cos(rw_theta)
      if(l_debug_rwc_3d) then
        write(unit=*, fmt='(a, 3g12.3, a, g10.3, a, g10.3)') &
             "DEBUG:  Trying step with:", rw_step(:), ", length=", &
             sqrt(dot_product(rw_step(:), rw_step(:))), ", bond=", &
             bond_length(monomer_types(i_mon_pos-1), &
             monomer_types(i_mon_pos))
      end if
      ! define the new position
      chain_positions(i_mon_pos, :) = &
           chain_positions(i_mon_pos-1, :) + rw_step(:)
      ! check whether new z coordinates is allright.  If so, exit loop
      if(((chain_positions(i_mon_pos, n_dim) &
           .gt. min_initial_wall_distance) .and. &
           (chain_positions(i_mon_pos, n_dim) .lt. (boundary(n_dim) &
           - min_initial_wall_distance))) .or. (n_dim_pbc .eq. 3)) exit
    end do ! end loop over random walk tries
    if(l_debug_rwc_3d) then
      write(unit=*, fmt='(2(a, i5))') &
           "DEBUG: Ended random walk step for i_mon_pos=", i_mon_pos-1, &
           " and i_mon_pos=", i_mon_pos
    end if
  end do ! i_mon_pos = 2, n_mon
end subroutine random_walk_chain_3d
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
subroutine random_walk_chain_2d(monomer_types, chain_positions)
  integer, dimension(n_mon), intent(in) :: monomer_types
  real(kind=dp), dimension(n_mon, n_dim), intent(out) :: chain_positions
  real(kind=dp), dimension(n_dim) :: rw_step
  real(kind=dp) :: rw_phi, rw_step_length
  integer :: i_mon_pos
  if(n_dim .eq. n_dim_pbc+1) then
    ! measure minimal initial wall distace in units of sigma
    min_initial_wall_distance = 0.8_dp * maxval(sigma(:, :))
    ! sanity check
    if(2.0_dp * min_initial_wall_distance .ge. boundary(n_dim)) then
      write(unit=*, fmt='(2(a, g13.6))') &
           "ERROR (SR random_walk_chain_2d): 2*min_initial_wall_distance=", &
           2.0_dp * min_initial_wall_distance, &
           " .le. boundary(n_dim)=", boundary(n_dim)
      stop
    end if
    write(unit=*, fmt='(a, g13.6)') &
         "MESSAGE (SR random_walk_chain_2d): min_initial_wall_distance = ", &
         min_initial_wall_distance
  else
    min_initial_wall_distance = 0.0_dp
  end if
  call ranlux(random, n_dim)
  ! specify position of first monomer in chain
  ! x coordinate
  i_mon_pos = 1
  chain_positions(i_mon_pos, 1) = random(1)*boundary(1)
  ! boundary(n_dim) is initialized with the initial wall separation
  chain_positions(i_mon_pos, n_dim) = min_initial_wall_distance &
       + (boundary(n_dim) - 2.0_dp * min_initial_wall_distance) &
       * random(n_dim)
  ! we don't need to check periodic boundary conditions here, as the
  ! particles are set up to conform to them
  ! generate random walk for next monomers on chain
     do i_mon_pos = 2, n_mon
       if(l_debug_rwc_2d) then
         write(unit=*, fmt='(2(a, i5), a, 3g10.3)') &
              "DEBUG: Starting random walk step from i_mon_pos=", &
              i_mon_pos, " starting from: ",chain_positions(i_mon_pos-1, :)
       end if
       do ! try random walk steps unless z-coordinate is in right range
         ! define random walk direction
         call ranlux(random, 1)
         rw_phi   = twopi * random(1)
         ! define the random walk step
         rw_step_length = bond_length(monomer_types(i_mon_pos-1), &
              monomer_types(i_mon_pos))
         ! in the one-dimensional system we can only step in one direction
         ! therefore we take the absolute value of the x-step
         rw_step(1) = rw_step_length * abs(cos(rw_phi))
         rw_step(n_dim) = rw_step_length * sin(rw_phi)
         if(l_debug_rwc_2d) then
           write(unit=*, fmt='(a, 2g10.3, a, g10.3, a, g10.3)') &
                "DEBUG:  Trying step with:", rw_step(:), ", length=", &
                sqrt(dot_product(rw_step(:), rw_step(:))), ", bond=", &
                bond_length(monomer_types(i_mon_pos-1), &
                monomer_types(i_mon_pos))
         end if
         ! define the new position
         chain_positions(i_mon_pos, :) = chain_positions(i_mon_pos-1, :) &
              + rw_step(:)
         ! check whether new z coordinates is alright.  If so, exit loop
         if(((chain_positions(i_mon_pos, n_dim) &
              .gt. min_initial_wall_distance) .and. &
              (chain_positions(i_mon_pos, n_dim) .lt. (boundary(n_dim) &
              - min_initial_wall_distance))) .or. (n_dim_pbc .eq. 2)) exit
       end do ! end loop over random walk tries
       if(l_debug_rwc_2d) then
         write(unit=*, fmt='(a,i5)') &
              "DEBUG: Ended random walk step for i_mon_pos=", i_mon_pos
       end if
     end do ! i_mon = 1, n_mon-1
   end subroutine random_walk_chain_2d
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! test the actual particle positions against the predicted particle positions
! from the Frenkel-Kontorova neighbor lists
subroutine compare_fk_positions
  ! tolerance when comparing theoretical and actual Frenkel-Kontorova
  ! neighbor positions.  If the program spits out a lot of errors in this
  ! subroutine although the positions seem right, then there might be a
  ! precision problem and fk_tolerance should be increased.  This can happen
  ! for large systems because substracting two numbers is numerically
  ! unfavourable.
  real(kind=dp), parameter :: fk_tolerance = 10.0_dp**(-precision(1.0_dp)+3)
  integer :: i_dim
  real(kind=dp) :: r_dummy
  real(kind=dp), dimension(3) :: delta_r
  integer :: i_wall, i_part, i_neigh
  ! checking counter
  integer :: error_counter
  ! loop over top wall particles
  error_counter = 0
  do i_wall = 1, n_top_wall
    i_part = i_wall + n_mon_tot
    ! loop over all neighbors
    do i_neigh=1, n_fk_neighbors
      call calc_distance(i_part, fk_neighbors(i_neigh, i_wall), delta_r, &
           r_dummy)
      do i_dim=1, n_dim
        if(abs(delta_r(i_dim) - fk_neigh_vec_tw(i_dim, i_neigh)) .gt. &
             fk_tolerance) then
          error_counter = error_counter + 1
          write(unit=*, fmt='(a, i6, a, i2, a, i2, a)') &
               "ERROR (SR conf_default): Wall particle", i_wall, &
               ", problem with fk_neigh_vec_tw(", i_dim, ",", i_neigh, ")"
          write(unit=*, fmt='(a, i2, a, 3(g13.6))') &
               "fk_neigh_vec_tw(:,", i_neigh, ") =       ", &
               fk_neigh_vec_tw(:, i_neigh)
          write(unit=*, fmt='(a, i2, a, 3(g13.6))') &
               "Computed vector to neighbor", i_neigh, ":", delta_r(:)
        end if
      end do
    end do ! end loop over neighbors
  end do ! end loop over wall particles
  ! loop over bottom wall particles
  do i_wall = n_top_wall + 1, n_wall
    i_part = i_wall + n_mon_tot
    ! loop over all neighbors
    do i_neigh=1, n_fk_neighbors
      call calc_distance(i_part, fk_neighbors(i_neigh, i_wall), delta_r, &
           r_dummy)
      do i_dim=1, n_dim
        if(abs(delta_r(i_dim) - fk_neigh_vec_bw(i_dim, i_neigh)) .gt. &
             fk_tolerance) then
          error_counter = error_counter + 1
          write(unit=*, fmt='(a, i6, a, i2, a, i2, a)') &
               "ERROR (SR conf_default): Wall particle", i_wall, &
               ", problem with fk_neigh_vec_bw(", i_dim, ",", i_neigh, ")"
          write(unit=*, fmt='(a, i2, a, 3(g22.15))') &
               "fk_neigh_vec_bw(:,", i_neigh, ") =       ", &
               fk_neigh_vec_bw(:, i_neigh)
          write(unit=*, fmt='(a, i2, a, 3(g22.15))') &
               "Computed vector to neighbor", i_neigh, ":", delta_r(:)
        end if
      end do
    end do ! end loop over neighbors
  end do ! end loop over wall particles
  if(error_counter .gt. 0) then
    write(unit=*, fmt='(a, i6, a)') "ERROR (SR conf_default): Found", &
         error_counter, " errors in neighbor positions.  Exiting."
    write(unit=*, fmt='(a)') &
         "Vectors to nearest neighbors for Frenkel-Kontorova interaction"
    write(unit=*, fmt='(a)') "top wall:"
    do i_neigh=1, n_fk_neighbors
      write(unit=*, fmt='(a, i2, a, 3(g22.15))') &
           "fk_neigh_vec_tw(:,", i_neigh, ") =       ", &
           (fk_neigh_vec_tw(i_dim, i_neigh), i_dim = 1, n_dim)
    ! first wall particle sits at (0,0,0)
      call calc_distance(n_mon_tot + 1, fk_neighbors(i_neigh, 1), &
           delta_r, r_dummy)
      write(unit=*, fmt='(a, i2, a, 3(g22.15))') &
           "Computed vector to neighbor", i_neigh, ":", &
           (delta_r(i_dim), i_dim = 1, n_dim)
    end do
    write(unit=*, fmt='(a)') ""
    write(unit=*, fmt='(a)') "bottom wall:"
    do i_neigh=1, n_fk_neighbors
      write(unit=*, fmt='(a, i2, a, 3(g22.15))') &
```

```
          "fk_neigh_vec_bw(:,", i_neigh, ") =       ", &
          (fk_neigh_vec_bw(i_dim, i_neigh), i_dim = 1, n_dim)
     ! first wall particle sits at (0,0,0)
     call calc_distance(n_mon_tot + n_top_wall + 1, &
          fk_neighbors(i_neigh, n_top_wall + 1), delta_r, &
          r_dummy)
     write(unit=*, fmt='(a, i2, a, 3(g22.15))') &
          "Computed vector to neighbor", i_neigh, ":", &
          (delta_r(i_dim), i_dim = 1, n_dim)
   end do
   stop
 end if ! errors encountered
end subroutine compare_fk_positions
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! reads the parameters of the simulation
subroutine read_parameters(in_file)
 integer, intent(in) :: in_file
 integer             :: io_status, counter
 character(len=65)   :: string
 integer :: i_dim
 ! loops over particle types
 integer :: i_type
 ! initialize
 counter = 0
 read(in_file, iostat=io_status, fmt='(i14, a)') s_time, string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, i14, a)') "MESSAGE: Read ", s_time, string
 end if
 read(in_file, iostat=io_status, fmt='(i14, a)') n_relax, string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, i14, a)') "MESSAGE: Read ", n_relax, string
 end if
 read(in_file, iostat=io_status, fmt='(i14, a)') n_obser, string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, i14, a)') "MESSAGE: Read ", n_obser, string
 end if
 read(in_file, iostat=io_status, fmt='(i14, a)') n_save, string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, i14, a)') "MESSAGE: Read ", n_save, string
 end if
 read(in_file, iostat=io_status, fmt='(i14, a)') n_time_ave, string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, i14, a)') "MESSAGE: Read ", n_time_ave, string
 end if
 read(in_file, iostat=io_status, fmt='(i14, a)') n_linear_out, string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, i14, a)') "MESSAGE: Read ", n_linear_out, string
 end if
 read(in_file, iostat=io_status, fmt='(l14, a)') &
      l_write_particle_positions, string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, l14, a)') "MESSAGE: Read ", &
        l_write_particle_positions, string
 end if
 read(in_file, iostat=io_status, fmt='(l14, a)') &
      l_write_particle_velocities, string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, l14, a)') "MESSAGE: Read ", &
        l_write_particle_velocities, string
 end if
 read(in_file, iostat=io_status, fmt='(l14, a)') l_read_sample_list, string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, l14, a)') "MESSAGE: Read ", l_read_sample_list, string
 end if
 read(in_file, iostat=io_status, fmt='(a)') file_sample_list
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, a)') "MESSAGE: Read ", trim(file_sample_list)
 end if
 read(in_file, iostat=io_status, fmt='(a)') string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, a)') "MESSAGE: Read ", string
 end if
 read(in_file, iostat=io_status, fmt='(e14.6, a)') dt, string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, e14.6, a)') "MESSAGE: Read ", dt, string
 end if
 read(in_file, iostat=io_status, fmt='(a)') string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, a)') "MESSAGE: Read ", string
 end if
 read(in_file, iostat=io_status, fmt='(e14.6, a)') temp_time, string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, e14.6, a)') "MESSAGE: Read ", temp_time, string
 end if
 read(in_file, iostat=io_status, fmt='(e14.6, a)') temp_init, string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, e14.6, a)') "MESSAGE: Read ", temp_init, string
 end if
 read(in_file, iostat=io_status, fmt='(e14.6, a)') temp_final, string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, e14.6, a)') "MESSAGE: Read ", temp_final, string
 end if
 read(in_file, iostat=io_status, fmt='(a)') string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, a)') "MESSAGE: Read ", string
 end if
 read(in_file, iostat=io_status, fmt='(e14.6, a)') r_2_min_time, string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, e14.6, a)') "MESSAGE: Read ", r_2_min_time, string
 end if
 read(in_file, iostat=io_status, fmt='(e14.6, a)') r_2_min_init, string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, e14.6, a)') "MESSAGE: Read ", r_2_min_init, string
 end if
 read(in_file, iostat=io_status, fmt='(a)') string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, a)') "MESSAGE: Read ", string
 end if
 do i_type=1, n_type
   read(in_file, iostat=io_status, fmt='(e14.6, a)')&
        mass_type(i_type), string
   counter = counter + 1
   if(io_status /= 0) then
     print *, "ERROR: Wrong format when reading ", counter, "th format."
     stop
   else
     write(*, '(a, e14.6, a)') "MESSAGE: Read ", mass_type(i_type), string
   end if
   read(in_file, iostat=io_status, fmt='(e14.6, a)')&
        epsil(i_type,i_type), string
   counter = counter + 1
   if(io_status /= 0) then
     print *, "ERROR: Wrong format when reading ", counter, "th format."
     stop
   else
     write(*, '(a, e14.6, a)') "MESSAGE: Read ",&
          epsil(i_type,i_type), string
   end if
   read(in_file, iostat=io_status, fmt='(e14.6, a)')&
        sigma(i_type,i_type), string
   counter = counter + 1
   if(io_status /= 0) then
     print *, "ERROR: Wrong format when reading ", counter, "th format."
     stop
   else
     write(*, '(a, e14.6, a)') "MESSAGE: Read ",&
          sigma(i_type,i_type), string
   end if
   read(in_file, iostat=io_status, fmt='(e14.6, a)')&
        friction(i_type), string
   counter = counter + 1
   if(io_status /= 0) then
     print *, "ERROR: Wrong format when reading ", counter, "th format."
     stop
   else
     write(*, '(a, e14.6, a)') "MESSAGE: Read ",&
          friction(i_type), string
   end if
 end do
 read(in_file, iostat=io_status, fmt='(a)') string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, a)') "MESSAGE: Read ", string
 end if
 read(in_file, iostat=io_status, fmt='(i14, a)') iseed, string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, i14, a)') "MESSAGE: Read ", iseed, string
 end if
 read(in_file, iostat=io_status, fmt='(a)') string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, a)') "MESSAGE: Read ", string
 end if
 read(in_file, iostat=io_status, fmt='(e14.6, a)')  skin, string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, e14.6, a)') "MESSAGE: Read ", skin, string
 end if
 read(in_file, iostat=io_status, fmt='(a)') string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
 else
   write(*, '(a, a)') "MESSAGE: Read ", string
 end if
 read(in_file, iostat=io_status, fmt='(e14.6, a)') x_space, string
 counter = counter + 1
 if(io_status /= 0) then
   print *, "ERROR: Wrong format when reading ", counter, "th format."
   stop
```

```
else
  write(*, '(a, e14.6, a)') "MESSAGE: Read ", x_space, string
end if
read(in_file, iostat=io_status, fmt='(l14, a)') &
     l_walls_are_commensurate, string
counter = counter + 1
if(io_status /= 0) then
  print *, "ERROR: Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, l14, a)') "MESSAGE: Read ", &
       l_walls_are_commensurate, string
end if

read(in_file, iostat=io_status, fmt='(e14.6, a)') z_space_wall, string
counter = counter + 1
if(io_status /= 0) then
  print *, "ERROR: Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, e14.6, a)') "MESSAGE: Read ", z_space_wall, string
end if
read(in_file, iostat=io_status, fmt='(e14.6, a)') k_spring_wall_tom, string
counter = counter + 1
if(io_status /= 0) then
  print *, "ERROR: Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, e14.6, a)') "MESSAGE: Read ", k_spring_wall_tom, string
end if
read(in_file, iostat=io_status, fmt='(e14.6, a)') &
     k_spring_wall_fre_kon_linear, string
counter = counter + 1
if(io_status /= 0) then
  print *, "ERROR: Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, e14.6, a)') "MESSAGE: Read ", &
       k_spring_wall_fre_kon_linear, string
end if
read(in_file, iostat=io_status, fmt='(e14.6, a)') &
     k_spring_wall_fre_kon_vector, string
counter = counter + 1
if(io_status /= 0) then
  print *, "ERROR: Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, e14.6, a)') "MESSAGE: Read ", &
       k_spring_wall_fre_kon_vector, string
end if
read(in_file, iostat=io_status, fmt='(a)') string
counter = counter + 1
if(io_status /= 0) then
  print *, "ERROR: Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, a)') "MESSAGE: Read ", string
end if

read(in_file, iostat=io_status, fmt='(i14, a)') f_wall_fix, string
counter = counter + 1
if(io_status /= 0) then
  print *, "ERROR: Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, i14, a)') "MESSAGE: Read ", f_wall_fix, string
end if
read(in_file, iostat=io_status, fmt='(i14, a)') f_cut_off, string
counter = counter + 1
if(io_status /= 0) then
  print *, "ERROR: Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, i14, a)') "MESSAGE: Read ", f_cut_off, string
end if
read(in_file, iostat=io_status, fmt='(i14, a)') f_thermostat_mode, string
counter = counter + 1
if(io_status /= 0) then
  print *, "ERROR: Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, i14, a)') "MESSAGE: Read ", f_thermostat_mode, string
end if
read(in_file, iostat=io_status, fmt='(i14, a)') f_minimize, string
counter = counter + 1
if(io_status /= 0) then
  print *, "ERROR: Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, i14,a)') "MESSAGE: Read ", f_minimize, string
end if
read(in_file, iostat=io_status, fmt='(i14, a)') f_friction_on_twall, string
counter = counter + 1
if(io_status /= 0) then
  print *, "ERROR: Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, i14,a)') "MESSAGE: Read ", f_friction_on_twall, string
end if
read(in_file, iostat=io_status, fmt='(e14.6, a)') &
     friction_constant_twall, string
counter = counter + 1
if(io_status /= 0) then
  print *, "ERROR: Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, e14.6,a)') "MESSAGE: Read ", friction_constant_twall,string
end if
read(in_file, iostat=io_status, fmt='(a)') string
counter = counter + 1
if(io_status /= 0) then
  print *, "ERROR: Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, a)') "MESSAGE: Read ", string
end if
read(in_file, iostat=io_status, fmt='(a)') string
counter = counter + 1
if(io_status /= 0) then
  print *, "ERROR: Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, a)') "MESSAGE: Read ", string
end if
read(in_file, iostat=io_status, fmt='(a)') string
counter = counter + 1
if(io_status /= 0) then
  print *, "ERROR: Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, a)') "MESSAGE: Read ", string
end if
! read the mode flags in each direction
! depending on the mode flag, the values in mode_parameter_one(i_dim)
! and mode_parameter_two(i_dim) are interpreted in subroutine init_params
read(in_file, iostat=io_status, fmt='(3i14)') &
     (f_twall(i_dim), i_dim=1, n_dim)
counter = counter + 1
if(io_status /= 0) then
  print *, "ERROR: Wrong format when reading ", counter, "th format."
    stop
  else
    write (unit=*, fmt='(a, 3i12)') "MESSAGE: Read ", f_twall(1:n_dim)
  end if
  read(in_file, iostat=io_status, fmt='(3e14.6)') &
       (mode_parameter_one(i_dim), i_dim=1, n_dim)
  counter = counter + 1
  if(io_status /= 0) then
    print *, "ERROR: Wrong format when reading ", counter, "th format."
    stop
  else
    write (unit=*, fmt='(a, 3e12.4)') &
         "MESSAGE: Read ", mode_parameter_one(1:n_dim)
  end if
  read(in_file, iostat=io_status, fmt='(3e14.6)') &
       (mode_parameter_two(i_dim), i_dim=1, n_dim)
  counter = counter + 1
  if(io_status /= 0) then
    print *, "ERROR: Wrong format when reading ", counter, "th format."
    stop
  else
    write (unit=*, fmt='(a, 3e12.4)') &
         "MESSAGE: Read ", mode_parameter_two(1:n_dim)
  end if
  read(in_file, iostat=io_status, fmt='(a)') string
  counter = counter + 1
  if(io_status /= 0) then
    print *, "ERROR: Wrong format when reading ", counter, "th format."
    stop
  else
    write(*, '(a, a)') "MESSAGE: Read ", string
  end if
  ! switches for computation
  call read_computation_switch(l_compute_fluid_distribution)
  call read_computation_switch(l_compute_fluid_vel_dist)
  call read_computation_switch(l_compute_press_tens)
  write(unit=*, fmt='(a)') &
       "MESSAGE: Successfully read parameters."
contains
  !------------------------------------------------------------------------
  ! reads a computation switch
  subroutine read_computation_switch(l_switch)
    logical, intent(inout) ::l_switch
    read(in_file, iostat=io_status, fmt='(l14, a)') l_switch, string
    counter = counter + 1
    if(io_status /= 0) then
      write(unit=*, fmt='(a, i3, a)') &
           "ERROR (SR read_computation_switch): Wrong format when reading", &
           counter, "th format."
      stop
    else
      write(unit=*, fmt='(a, l14, a)') "MESSAGE: Read ", l_switch, string
    end if
  end subroutine read_computation_switch
  !------------------------------------------------------------------------
end subroutine read_parameters
!--------------------------------------------------------------------------

!--------------------------------------------------------------------------
! writes out the parameters of the current simulation
subroutine write_parameters(param_out_file)
  character(len=*), intent(in) :: param_out_file
  integer, parameter :: out_file = 20
  integer :: io_status
  integer :: i_dim
  ! loops over particle types
  integer :: i_type
  ! check if the file we want to write already exists
  open(unit=out_file, file=param_out_file, status="old", &
       iostat=io_status)
  if(io_status == 0) then
    write(unit=*, fmt='(3a)') "CAUTION: File >>", param_out_file, "<<"
    write(unit=*, fmt='(a)') &
         "          already exists and will be overwritten."
    close(unit=out_file)
  end if
  ! open file for replacement
  open(unit=out_file, file=param_out_file, status="replace", &
       action="write", iostat=io_status)
  if(io_status .ne. 0) then
    write (unit=*, fmt='(3a)') &
         "ERROR: Couldn't open file >>", param_out_file, &
         "<< for writing new parameters."
    stop
  else
    write (unit=*, fmt='(3a)') &
         "MESSAGE: Writing parameters to >>", param_out_file, "<<"
    write(unit=out_file, fmt='(i14, a)')   s_time, &
         " initial integer starting time"
    write(unit=out_file, fmt='(i14, a)')   n_relax, &
         " number of relaxation steps"
    write(unit=out_file, fmt='(i14, a)')   n_obser, &
         " number of observation steps"
    write(unit=out_file, fmt='(i14, a)')   n_save, &
         " number of steps between full configuration backup"
    write(unit=out_file, fmt='(i14, a)')   n_time_ave, &
         " time constant for filtering output"
    write(unit=out_file, fmt='(i14, a)')   n_linear_out, &
         " linear position saving timestep"
    write(unit=out_file, fmt='(l14, a)')   l_write_particle_positions, &
         " switch for writing particle positions"
    write(unit=out_file, fmt='(l14, a)')   l_write_particle_velocities, &
         " switch for writing particle velocities"
    write(unit=out_file, fmt='(l14, a)')   l_read_sample_list, &
         " switch for reading a sample list in file"
    write(unit=out_file, fmt='(a)') trim(file_sample_list)
    write(unit=out_file, fmt='(a)') ""
    write(unit=out_file, fmt='(e14.6, a)') dt, &
         " time step increment"
    write(unit=out_file, fmt='(a)') ""
    write(unit=out_file, fmt='(e14.6, a)') temp_time, &
         " time to reach final temperature"
    write(unit=out_file, fmt='(e14.6, a)') temp_init, &
         " initial temperature"
    write(unit=out_file, fmt='(e14.6, a)') temp_final, &
         " final temperature"
    write(unit=out_file, fmt='(a)') ""
    write(unit=out_file, fmt='(e14.6, a)') r_2_min_time, &
         " time to reach real potential"
    write(unit=out_file, fmt='(e14.6, a)') r_2_min_init, &
         " squared initial minimum effective distance between particles"
    write(unit=out_file, fmt='(a)') ""
    do i_type = 1, n_type
      write(unit=out_file, fmt='(e14.6, a, i1, a)') mass_type(i_type), &
           " mass                 (type ", i_type, ")"
      write(unit=out_file, fmt='(e14.6, a, i1, a)') &
           epsil(i_type,i_type), &
           " Lenard Jones epsilon (type ", i_type, ")"
      write(unit=out_file, fmt='(e14.6, a, i1, a)') sigma(i_type,i_type), &
           " Lenard Jones sigma   (type ", i_type, ")"
      write(unit=out_file, fmt='(e14.6, a, i1, a)') friction(i_type), &
           " friction constant    (type ", i_type, ")"
    end do
    write(unit=out_file, fmt='(a)') ""
```

```
    ! iseed + 1 is here to get new random numbers the next time
    write(unit=out_file, fmt='(i14, a)') iseed + 1, &
         " seed for random number generator"
    write(unit=out_file, fmt='(a)')
    write(unit=out_file, fmt='(e14.6, a)') skin , &
         " skin thickness"
    write(unit=out_file, fmt='(a)') ""
    write(unit=out_file, fmt='(e14.6, a)') x_space, &
         " spacing between wall units in x direction"
    write(unit=out_file, fmt='(l14, a)') l_walls_are_commensurate, &
         " switch: walls are commensurate(T) / incommensurate(F)"
    write(unit=out_file, fmt='(e14.6, a)') z_space_wall , &
         " initial inter-wall spacing"
    write(unit=out_file, fmt='(e14.6, a)') k_spring_wall_tom, &
         " Tomlinson spring constant to wall equililibrium sites"
    write(unit=out_file, fmt='(e14.6, a)') k_spring_wall_fre_kon_linear, &
         " Linear Frenkel-Kontorova spring constant between wall particles"
    write(unit=out_file, fmt='(e14.6, a)') k_spring_wall_fre_kon_vector, &
         " Vector Frenkel-Kontorova spring constant between wall particles"
    write(unit=out_file, fmt='(a)') ""
    write(unit=out_file, fmt='(i14, a)') f_wall_fix, &
         " flag for constraining wall atoms"
    write(unit=out_file, fmt='(i14, a)') f_cut_off, &
         " cut-off flag"
    write(unit=out_file, fmt='(i14, a)') f_thermostat_mode, &
         " flag for thermostat mode"
    write(unit=out_file, fmt='(i14, a)') f_minimize, &
         " flag to relax to next minimum (will reset friction constants)"
    write(unit=out_file, fmt='(i14, a)') f_friction_on_twall, &
         " flag for turning on friction force on top wall"
    write(unit=out_file, fmt='(e14.6, a)') friction_constant_twall, &
         " friction constant for friction on top wall"
    write(unit=out_file, fmt='(a)') ""
    write(unit=out_file, fmt='(a)') &
         " flags and variables controlling boundary conditions:"
    write(unit=out_file, fmt='(a)') &
         "   1-direction   2-direction   3-direction"
    ! f_twall: 1  top wall constr veloc. (v_0),   velocity mode
    ! f_twall: 2  top wall const. accel. (a_0),   force_mode
    ! f_twall: 3  top wall pulled by ext. spring, spring mode
    ! f_twall = 1 = velocity_mode
    !-----------------------------
    ! v_spring_twall: velocity with which the upper wall is moved
    ! k_spring_twall has no meaning in this case
    ! f_twall = 2 = force_mode
    !-------------------------
    ! ext_force_twall: externally applied force that each individual atom
    !              in upper wall experiences
    ! ramp_force_twall: change of ext_force_twall per unit time
    ! f_twall = 3 = spring_mode (external spring pulling top wall)
    !-----------------------------------------------------------
    ! v_spring_twall: velocity with which the spring is moved
    ! k_spring_twall: spring constant of spring
    !                 with which upper wall is moved (normalized on
    !                 number of atoms in upper wall)
    write(unit=out_file, fmt='(3i14)')   (f_twall(i_dim),i_dim=1,n_dim)
    write(unit=out_file, fmt='(3e14.6)') &
        (mode_parameter_one(i_dim),i_dim=1,n_dim)
    write(unit=out_file, fmt='(3e14.6)') &
        (mode_parameter_two(i_dim),i_dim=1,n_dim)
    write(unit=out_file, fmt='(a)') ""
    write(unit=out_file, fmt='(l14, a)') l_compute_fluid_distribution, &
         " switch l_compute_fluid_distribution"
    write(unit=out_file, fmt='(l14, a)') l_compute_fluid_vel_dist, &
         " switch l_compute_fluid_vel_dist"
    write(unit=out_file, fmt='(l14, a)') l_compute_press_tens, &
         " switch l_compute_press_tens"
    write(unit=out_file, fmt='(a)') ""
    close(unit=out_file)
  end if
end subroutine write_parameters
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! initialize observation variables
subroutine init_variables
  integer :: i_dim
  ! initialize low pass filtered top wall observation variables
  r0_twall_1(:)          = r0_twall(:)        ! position of top wall
  velocity_twall_1(:)    = rx_twall(:, 1)     ! velocity of top wall (*dt)
  if(l_compute_cm_fluid) then
    ! center of mass fluid position (normalize later)
    do i_dim = 1, n_dim
      cm_fluid_1(i_dim) = sum(r0_unfolded(1:n_mon_tot, i_dim))
    end do
    ! note: the center of mass of the fluid will be close to the middle of
    ! the walls, but r0_twall is the lower left corner.
    ! When comparing positions 0.5*boundary has to be added to the top wall
    ! position
  end if
  ! r0_ave(:, :)           = r0(:, :)         ! positions of particles
  do i_dim = 1, n_dim
    select case(f_twall(i_dim))
    case(velocity_mode)
      ext_force_twall_1(i_dim) = 0.0_dp
    case(spring_mode)
      ext_force_twall_1(i_dim) = 0.0_dp
    case(force_mode)
      ext_force_twall_1(i_dim) = ext_force_twall(i_dim)
    end select
  end do
  ! Total force on the top wall.
  ! Additional forces (from particles) will be added later.
  ! Note that ext_force_twall(:) is the force per particle, but
  ! total_force_twall is the sum of all forces.
  total_force_twall_1(:) = real(n_top_wall, kind=dp) * ext_force_twall_1(:)
  r0_last_pinned(:)      = r0_twall(:)          ! last pinning position
end subroutine init_variables
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! subroutine for setting up the energy log file
subroutine init_energy_file(energy_log_file)
  character(len=*), intent(in) :: energy_log_file
  integer :: iostatus
  open(unit=energy_log_file_unit, file=energy_log_file, &
       status="unknown", action="write", &
       iostat=iostatus)
  if(iostatus .ne. 0) then
    write(unit=*, fmt='(3a)') &
        "ERROR: Couldn't open file >>", energy_log_file, "<< for writing."
    stop
  else
    write(unit=*, fmt='(3a)') &
         "MESSAGE: Writing energies to >>", energy_log_file, "<<"
    ! write file header depending on the simulation
    write (unit=energy_log_file_unit, fmt='(a)') &
         "# energy log data"
    write (unit=energy_log_file_unit, fmt='(a)') &
         "# ==============="
    if(n_mon_tot .gt. 0) then
      write (unit=energy_log_file_unit, fmt='(a)') &
           "# time | sum_all_energies | static_energy_top_wall | &
           &t_cms | t_wall | t_fluid | v_wall_wall | &
           &v_wall_harm_tom | v_wall_harm_fre_kon_linear | &
           &v_wall_harm_fre_kon_vector | v_fluid_wall | &
           &v_fluid_fluid | v_intra_molec"
    else
      write (unit=energy_log_file_unit, fmt='(a)') &
           "# time | sum_all_energies | static_energy_top_wall | &
           &t_cms | t_wall | v_wall_wall | &
           &v_wall_harm_tom | v_wall_harm_fre_kon_linear | &
           &v_wall_harm_fre_kon_vector"
    end if
    close(energy_log_file_unit)
  end if
end subroutine init_energy_file
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! calculates the derivative of the bond-potential in units of epsilon/sigma
! distance is measured in units of sigma
! This version includes attractive LJ-interaction.
function bond_potential_derivative(distance)
  real(kind=dp)             :: bond_potential_derivative
  real(kind=dp), intent(in) :: distance
  if(distance .le. 0.0_dp) then
    write(unit=*, fmt='(a, g13.6, a)') &
         "ERROR (function bond_potential_derivative): distance=", distance, &
         " < 0"
    stop
  end if
  if(distance .ge. r_chain) then
    write(unit=*, fmt='(2(a, g13.6))') &
         "ERROR (function bond_potential_derivative): distance=", distance, &
         " > r_chain=", r_chain
    stop
  end if
  bond_potential_derivative = &
       (k_chain * distance)/(1.0_dp - (distance/r_chain)**2) &
       + 4.0_dp * (6.0_dp * distance**(-7) - 12.0_dp*distance**(-13))
end function bond_potential_derivative
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! given a function func, find the root of a function known to lie between a
! and b using the bisection method.
! The root is refined until a certain accuracy has been reached.
! The number of function evaluations is recorded on n_func_evals
subroutine bisection(func, start_a, start_b, accuracy, bsroot, n_func_evals)
  implicit none
  interface
    function func(x)
      use mykinds
      implicit none
      real(kind=dp), intent(in) :: x
      real(kind=dp)             :: func
    end function func
  end interface
  real(kind=dp), intent(in)      :: start_a, start_b
  real(kind=dp), intent(in)      :: accuracy
  real(kind=dp), intent(out)     :: bsroot
  integer, optional, intent(out) :: n_func_evals
  ! function arguments and values
  real(kind=dp) a, b, fa, fb
  ! step width
  real(kind=dp) :: dx
  ! maximal number of iterations
  integer, parameter :: n_max_iterations = 1000
  ! loop variable
  integer :: i_loop
  ! sanity check
  if(accuracy .le. 10.0_dp * spacing(max(start_a,start_b))) then
    write(unit=*, fmt='(a, g21.14, a, g21.14)') &
         "ERROR (SR bisection): accuracy=", accuracy, &
         " < 10.0_dp * spacing(max(start_a,start_b))=", &
         10.0_dp * spacing(max(start_a,start_b))
    stop
  end if
  ! reset counter
  if(present(n_func_evals)) n_func_evals = 0
  ! get first values
  a = start_a
  b = start_b
  fa = func(a)
  fb = func(b)
  if(present(n_func_evals)) n_func_evals = n_func_evals + 2
  ! check if a and b are bracketing
  if((fa < 0.0_dp .and. fb < 0.0_dp) &
       .or. (fa > 0.0_dp .and. fb > 0.0_dp)) then
    write(unit=*, fmt='(a)') &
         "ERROR (SR bisection): Root must be bracketed"
    stop
  end if
  ! orient values so that fa < fb
  if(fa < 0.0_dp) then
    bsroot = a
    dx = b - a
  else
    bsroot = b
    dx = a - b
  end if
  ! bisection loop
  do i_loop = 1, n_max_iterations
    dx = 0.5_dp * dx
    a = b ! a is just the last value of b
    b = bsroot + dx
    fa = fb
    fb = func(b)
    if(present(n_func_evals)) n_func_evals = n_func_evals + 1
    ! find out to which half the root belongs
    if(fb .le. 0.0_dp) bsroot = b
    ! test if we are done
    if((abs(dx) < accuracy) .or. (fb .eq. 0.0_dp)) return
  end do
  write(unit=*, fmt='(a, i11, a)') &
       "WARNING (SR bisection): Maximum number of iterations, ", &
       n_max_iterations, " exceeded"
end subroutine bisection
!-------------------------------------------------------------------------------
end module initialization
```

mdroutinesV1.9.f90

```
! this module contains the core routines for the simulation:
! subroutine binning3d:  binning and neighbor lists
! subroutine thermostat: Langevin thermostat, force initialization
! subroutine predict:    the predictor of the predictor-corrector algorithm
! subroutine correct:    the corrector of the predictor-corrector algorithm
! In addition there are some helper routines which are not so important.
! Martin Aichele, 2001-02-07
! last changed 2002-04-19
! V1.9 with real dimension switch
! Martin Aichele, 2003-02-24
! last modified 2003-02-25
module mdroutines
  ! module containing global variables
  use globals
  ! random number module
  use luxury
  ! interaction module, used for the Frenkel-Kontorova neighbor lists
  use interaction, only: fk_neighbors
  ! miscelaneous helper functions
  use utilities
  implicit none
  !-------------------------------------------------------------------------
  ! debug switches      |-------- 31 characters ------|
  !-------------------------------------------------------------------------
  logical, parameter :: l_debug_binning3d               = .FALSE.
  logical, parameter :: l_debug_predict                 = .FALSE.
  logical, parameter :: l_debug_corrector               = .FALSE.
  logical, parameter :: l_debug_thermostat              = .FALSE.
  logical, parameter :: l_debug_check_skin              = .FALSE.
  logical, parameter :: l_debug_max_distance            = .FALSE.
  logical, parameter :: l_debug_min_distance            = .FALSE.
  !-------------------------------------------------------------------------
  !-------------------------------------------------------------------------
  ! maximal allowed correction of coordinates before issuing a warning
  real(kind=dp), parameter :: max_allowed_correction    = 0.001_dp
  !-------------------------------------------------------------------------
contains
  !-------------------------------------------------------------------------
  ! calculate the squared distance of particles j_part and i_part.
  ! this function is called very often, so speed is important here.
  ! this function is identical to calc_distance concerning the return value.
  real(kind=dp) function distance_squared(i_part, j_part)
    integer, intent(in) :: i_part, j_part
    integer :: i_dim
    ! test if speed is gained if the local variables have the SAVE attribute.
    real(kind=dp), dimension(n_dim) :: delta_r_vec
    delta_r_vec(:) = r0(i_part,:) - r0(j_part,:)
    ! minimum image convention in n_dim_pbc dimensions only
    ! here one can save a lot of cpu time if the right method is chosen
    if(l_mic_use_int_cast) then
      ! delta_r_vec(i_dim)/half_bound(i_dim) is .le. 1.0, as r0 are
      ! folded coordinates.
      do i_dim = 1, n_dim_pbc
        delta_r_vec(i_dim) = delta_r_vec(i_dim) - boundary(i_dim) &
             * aint(delta_r_vec(i_dim)/half_bound(i_dim))
      end do
    else ! faster on Pentiums
      do i_dim = 1, n_dim_pbc
        if(delta_r_vec(i_dim) > half_bound(i_dim)) then
          delta_r_vec(i_dim) = delta_r_vec(i_dim) - boundary(i_dim)
        else if(delta_r_vec(i_dim) < -half_bound(i_dim)) then
          delta_r_vec(i_dim) = delta_r_vec(i_dim) + boundary(i_dim)
        end if
      end do
    end if
   distance_squared = dot_product(delta_r_vec(:), delta_r_vec(:))
  end function distance_squared
  !-------------------------------------------------------------------------

  !-------------------------------------------------------------------------
  ! checks if the coordinates are in the proper range.  Its main use is for
  ! testing and debugging.
  subroutine binning3d_check_range(i_part)
    integer, intent(in) :: i_part
    integer :: i_dim
    do i_dim=1, n_dim_pbc
      if((r0(i_part, i_dim).lt.0.0_dp) &
           .or. (r0(i_part, i_dim).gt.boundary(i_dim))) then
        write (unit=*, fmt='(a,i8,a,i1,a,e12.5)') &
             "ERROR (SR binning3d_check_range): r0(", i_part, ",", i_dim, &
             ")=", r0(i_part, i_dim)
        write (unit=*, fmt='(a,i1,a,e12.5,a,i12)') &
             "      out of range 0.0 -- boundary(", i_dim, ")=", &
             boundary(i_dim), " at MD step ", i_time
        stop
      end if
    end do
    ! check the z-range of fluid particles only
    if(n_dim .eq. n_dim_pbc+1) then
      if(i_part .le. n_mon_tot) then
        ! z-direction
        if((r0(i_part, n_dim).lt.lowest_fluid_z) &
             .or. (r0(i_part, n_dim).gt.highest_fluid_z)) then
          write (unit=*, fmt='(a,i8,a,i1,a,e12.5)') &
               "ERROR (SR binning3d_check_range): r0(", i_part, ",", n_dim, &
               ")=", r0(i_part, n_dim)
          write (unit=*, fmt='(a,e12.5,a,e12.5,a,i12)') &
               "      out of range lowest_fluid_z=", lowest_fluid_z, &
               " -- highest_fluid_z=", highest_fluid_z, &
               " at MD step ", i_time
          stop
        end if
      end if
    end if
  end subroutine binning3d_check_range
  !-------------------------------------------------------------------------

  !-------------------------------------------------------------------------
  ! checks if there is at least one particle which has moved more than half
  ! the skin, if so, the function returns .FALSE., .TRUE. otherwise
  logical function check_skin()
    real(kind=dp), dimension(n_dim) :: delta_r_vec
    integer :: i_part
    if(l_debug_check_skin) then
      print *, "DEBUG: Entering function check_skin()"
    end if
    do i_part = 1, n_part
      ! calculate how far the particle has moved after last binning
      delta_r_vec(1:n_dim_pbc) = &
           r0_old(i_part, 1:n_dim_pbc) - r0(i_part, 1:n_dim_pbc) &
           + real(pbc_count_old(i_part, 1:n_dim_pbc) &
           - pbc_count(i_part, 1:n_dim_pbc),kind=dp) &
           * boundary(1:n_dim_pbc)
      delta_r_vec(n_dim_pbc+1:n_dim) &
           = r0_old(i_part, n_dim_pbc+1:n_dim) - r0(i_part, n_dim_pbc+1:n_dim)
      if (dot_product(delta_r_vec, delta_r_vec) .ge. 0.25_dp * skin**2) then
        check_skin = .FALSE.
        if(l_debug_check_skin) then
          write(unit=*, fmt='(a, g13.6, a)') &
               "DEBUG: At r_time=", r_time, &
               " leaving function check_skin() with .FALSE."
          write(unit=*, fmt='(a, f9.6, a, f9.6)') &
               "       |delta_r| =", &
               sqrt(dot_product(delta_r_vec, delta_r_vec)), &
               " > skin/2 = ", skin/2.0_dp
        end if
        ! once we have found one particle which has moved this far, we can
        ! leave the function.
        return
      end if
    end do
    ! if no particle has moved farther than skin/2 then return .TRUE.
    check_skin = .TRUE.
    if(l_debug_check_skin) then
      print *, "DEBUG: At r_time=", r_time, &
           " leaving function check_skin() with .TRUE."
    end if
  end function check_skin
  !-------------------------------------------------------------------------

  !-------------------------------------------------------------------------
  ! bin the particles into boxes and create neighbor lists
  ! this routine bins in three dimensions
  subroutine binning3d
    integer :: i_dim
    ! fine grained bin usage
    integer, dimension(n_dim_max) :: bin_lowest_fl_tw, bin_highest_fl_bw
    ! number of fluid bins in z-direction in use (might change when walls are
    ! moved)
    integer :: n_bin_z_used
    ! bin looping variables
    integer, dimension(n_dim_max) :: i_bin
    ! note: do loop variables can't have subscripts
    integer :: i_bin_x, i_bin_y, i_bin_z, j_bin_x, j_bin_y, j_bin_z, &
         k_bin_x, k_bin_y, k_bin_z
    ! n(1/2)_bin_part: number of particles in bin
    ! i(1/2)_bin_part: for looping over particles in bin
    integer :: i1_bin_part, i2_bin_part, n1_bin_part, n2_bin_part
    integer :: i_wall        ! loop over wall particles
    integer :: i_part, j_part ! loop over all particles
    ! loops over particle types
    integer :: i_type, j_type
    ! lowest and highest wall coordinates of top and bottom wall
    real(kind=dp) :: lowest_twall_z, highest_bwall_z
    ! lowest and highest indices of top and bottom wall z-bin layer interacting
    ! with the fluid
    integer :: bin_z_lowest_fl_twall, bin_z_highest_fl_bwall
    ! minimal distance of wall particles
    real(kind=dp) :: minimal_wall_distance
    if(l_debug_binning3d) then
      write(unit=*, fmt='(a)') "DEBUG: Entering subroutine binning3d"
    end if
    ! count only calls after we have entered the main MD loop and are not
    ! relaxing
    if(i_time .ge. n_relax) counter_list_updates = counter_list_updates + 1
    ! save positions and periodic boundary condition counters
    ! in order to find the displacement of particles
    r0_old(:, :)        = r0(:, :)
    pbc_count_old(:, :) = pbc_count(:, :)
    ! zero all binning entries, at places for particle indices, write huge(1)
    ! which will lead to segmentation faults if there's a bug
    !=========================================================================
    ! if no fluid is present, this array is not allocated
    if(n_mon_tot .gt. 0) then
      bin_fluid(:, :, :, 0)  = 0
      bin_fluid(:, :, :, 1:) = huge(1)
    end if
    if(n_dim .eq. n_dim_pbc+1) then
      bin_twall(:, :, 0)  = 0
      bin_bwall(:, :, 0)  = 0
      bin_twall(:, :, 1:) = huge(1)
      bin_bwall(:, :, 1:) = huge(1)
    end if
    ! reset neighbor lists
    if(l_fluid_fluid_interaction) then
      ff_list(:, 0)  = 0
      ff_list(:, 1:) = huge(1)
    end if
    if(l_fluid_wall_interaction) then
      fw_list(:, 0)  = 0
      fw_list(:, 1:) = huge(1)
    end if
    if(l_allow_wall_wall_interaction) then
      ww_list(:, 0)  = 0
      ww_list(:, 1:) = huge(1)
    end if
    ! find the range of z-positions of the fluid particles
    if((n_mon_tot .gt. 0) .and. (n_dim .eq. n_dim_pbc+1)) then
      highest_fluid_z = maxval(r0(1:n_mon_tot, n_dim))
      lowest_fluid_z = minval(r0(1:n_mon_tot, n_dim))
      if(l_debug_binning3d) then
        write(unit=*, fmt='(2(a,g13.6))') &
             "MESSAGE (SR binning3d): highest_fluid_z=", highest_fluid_z, &
             "lowest_fluid_z=", lowest_fluid_z
      end if
      ! if the walls moved farther apart, the width of the bins in z-direction
      ! must be increased if
      ! (max_interaction_range+skin)*n_bin(3) &
      !      < highest_fluid_z-lowest_fluid_z
      !
      ! if the walls move closer, then the bin width can only be decreased down
      ! to max_interaction_range + skin.  The number of bins allocated is
      ! fixed, but not all of them might be filled.
      if(highest_fluid_z - lowest_fluid_z &
           .ge. (max_interaction_range + skin)*real(n_bin(n_dim),kind=dp)) then
        n_bin_z_used = n_bin(n_dim) ! all bins are used
        ! this is the optimal bin width
        r_bin(n_dim) = &
             (highest_fluid_z-lowest_fluid_z)/real(n_bin(n_dim),kind=dp)
        if(l_debug_binning3d) then
          write(unit=*, fmt='(a,g13.6,a,g13.6,a)') &
               "MESSAGE (SR binning3d): r_bin(n_dim)= ", r_bin(n_dim), &
               "(max_interaction_range + skin= ", max_interaction_range+skin,&
               ")"
        end if
      else
        ! we try to fill the bins evenly and not to have a only slighly
        ! filled one (if we have a fluid film thicker than the minimal
        ! allowed bin thickness).
        ! Find out how many bins are actually used, we only loop only over
        ! these.
        n_bin_z_used = int((highest_fluid_z - lowest_fluid_z) &
             / (max_interaction_range + skin))
        if(n_bin_z_used .eq. 0) then
          ! there is a fluid layer thinner than max_interaction_range + skin
          r_bin(n_dim) = max_interaction_range + skin
          n_bin_z_used = 1
          if(l_debug_binning3d) then
            write(unit=*, fmt='(a,i4,a)') &
                 "MESSAGE (SR binning3d): r_bin(n_dim) set to minimal value, &
```

```
            &n_bin_z_used set to 1 (n_bin(n_dim)=",n_bin(n_dim),")"
      end if
    else
      ! this is the thickness bigger than max_interaction_range + skin
      ! leading to completely filled bins
      r_bin(n_dim) = (highest_fluid_z - lowest_fluid_z) &
           /real(n_bin_z_used, kind=dp)
      if(l_debug_binning3d) then
        write(unit=*, fmt='(a,g13.6,a,i4,a,i4,a)') &
             "MESSAGE (SR binning3d): r_bin(n_dim)=", r_bin(n_dim), &
             ", n_bin_z_used=",n_bin_z_used,&
             " (n_bin(n_dim)=",n_bin(n_dim),")"
      end if
    end if !((n_bin_z_used .eq. 0)
  end if !(highest_fluid_z - lowest_fluid_z &
  !       .gt.(max_interaction_range + skin) * real(n_bin(n_dim), kind=dp))
end if !((n_mon_tot .gt. 0) .and. (n_dim .eq. n_dim_pbc+1))
! get extremal coordinates of walls in z
if(n_dim .eq. n_dim_pbc+1) then
  lowest_twall_z  = minval(r0(n_mon_tot+1:n_mon_tot+n_top_wall, n_dim))
  highest_bwall_z = maxval(r0(n_mon_tot+n_top_wall+1:n_part, n_dim))
end if
if(l_allow_wall_wall_interaction) then
  ! check if there is possible interaction between the two walls
  ! this is rather a lower bound, which might be negative
  minimal_wall_distance = lowest_twall_z - highest_bwall_z
  if(minimal_wall_distance .gt. (max_interaction_range + skin)) then
    l_wall_wall_interaction = .FALSE.
  else
    l_wall_wall_interaction = .TRUE.
  end if
  if(l_debug_binning3d) then
    write(unit=*, fmt='(2(a, g13.6))') &
         "MESSAGE (SR binning3d): minimal_wall_distance=", &
         minimal_wall_distance, " max_interaction_range + skin=", &
         max_interaction_range + skin
    write(unit=*, fmt='(a, l2)') &
         "            so l_wall_wall_interaction=", l_wall_wall_interaction
  end if
end if !(l_allow_wall_wall_interaction)
if((n_mon_tot .gt. 0) .and. (n_dim .eq. n_dim_pbc+1)) then
  ! check which fluid z-bin layers can interact with the walls
  ! we need to allow for a skin since we don't do binning every MD step
  ! the lowest index of the fluid bins which can interact with the lowest
  ! particle in the top wall.  lowest_fluid_z is the z-coordinate from
  ! where binning is started, i.e. the bottom of bin(x, y, 0, :)
  bin_z_lowest_fl_twall = max(int((lowest_twall_z - lowest_fluid_z &
       - max_interaction_range - skin) / r_bin(n_dim)), 0)
  ! make sure that bin_z_lowest_fl_twall doesn't become too large, so
  ! that we have at least interaction with the highest fluid z-layer
  if(bin_z_lowest_fl_twall .gt. n_bin_z_used -1) &
       bin_z_lowest_fl_twall = n_bin_z_used -1
  ! the highest index of the fluid bins which can interact with the highest
  ! particle in the bottom wall.
  bin_z_highest_fl_bwall = min(int((highest_bwall_z - lowest_fluid_z &
       + max_interaction_range + skin)/ r_bin(n_dim)), n_bin_z_used -1)
  ! make sure that bin_z_highest_fl_bwall doesn't become negative, so
  ! that we have at least interaction with the lowest fluid z-layer
  if(bin_z_highest_fl_bwall .lt. 0) bin_z_highest_fl_bwall = 0
  if(l_debug_binning3d) then
    write(unit=*, fmt='(2(a, i4))')&
         "MESSAGE (SR binning3d): bin_z_lowest_fl_twall=", &
         bin_z_lowest_fl_twall, " bin_z_highest_fl_bwall=", &
         bin_z_highest_fl_bwall
  end if
end if !(n_mon_tot .gt. 0) .and. (n_dim .eq. n_dim_pbc+1)

! which bins are actually used
n_bin_used(1:n_dim_pbc) = n_bin(1:n_dim_pbc)
if(n_dim .eq. n_dim_pbc+1) n_bin_used(n_dim) = n_bin_z_used
n_bin_used(n_dim+1:n_dim_max) = 1
bin_lowest_fl_tw(1:n_dim_pbc) = 0
if(n_dim .eq. n_dim_pbc+1) bin_lowest_fl_tw(n_dim) = bin_z_lowest_fl_twall
bin_lowest_fl_tw(n_dim+1:n_dim_max) = 0
bin_highest_fl_bw(1:n_dim_pbc) = n_bin(1:n_dim_pbc) -1
if(n_dim .eq. n_dim_pbc+1) bin_highest_fl_bw(n_dim)=bin_z_highest_fl_bwall
bin_highest_fl_bw(n_dim+1:n_dim_max) = 0
! check
if(minval(n_bin_used(:)) .le. 0 .or. &
     maxval(n_bin_used(:) - n_bin(:)) .gt. 0) then
  write(unit=*, fmt='(a)') "ERROR (SR binning3d): &
       &n_bin_used(:) out of range:)"
  print *, "n_bin(:)=", n_bin(:)
  print *, "n_bin_used(:)=", n_bin_used(:)
  stop
end if
if(minval(bin_lowest_fl_tw(:)) .lt. 0 .or. &
     maxval(bin_lowest_fl_tw(:) - n_bin_used(:)) .ge. 0) then
  write(unit=*, fmt='(a)') "ERROR (SR binning3d): &
       &bin_lowest_fl_tw(:) out of range:"
  print *, "n_bin_used(:)=", n_bin_used(:)
  print *, "bin_lowest_fl_tw(:)=", bin_lowest_fl_tw(:)
  stop
end if
if(minval(bin_highest_fl_bw(:)) .lt. 0 .or. &
     maxval(bin_highest_fl_bw(:) - n_bin_used(:)) .ge. 0) then
  write(unit=*, fmt='(a)') "ERROR (SR binning3d): &
       &bin_highest_fl_bw(:) out of range:"
  print *, "n_bin_used(:)=", n_bin_used(:)
  print *, "bin_highest_fl_bw(:)=", bin_highest_fl_bw(:)
  stop
end if
!------------------------------------------------------------------------
! bin the particles into the boxes
!------------------------------------------------------------------------
! As r_bin(1) = boundary(1)/n_bin(1) one could write
! i_bin(1) = int(r0(i_part,1)/r_bin(1)) if 0 <= position < L and if we had
! infinite precision.
! But because of finite precision int(r0(i_part,1)/r_bin(1)) could become
! n_bin(1) (if r_bin(1) was rounded towards zero or position == L)
! outside the valid range.
! To prevent this we use the if() statements after the int cast.
! In every MD-step the positions are folded back, so we know
! 0 <= position <= L, (but not < L) for sure.
! fluid
do i_part = 1, n_mon_tot
  ! bins in dimensions which are not in use get i_bin = 0
  i_bin(n_dim+1:n_dim_max) = 0
  ! check range of coordinates
  if(l_binning3d_check_range) call binning3d_check_range(i_part)
  ! find indices of bins
  do i_dim = 1, n_dim_pbc
    i_bin(i_dim)= int(r0(i_part, i_dim)/r_bin(i_dim))
    if(i_bin(i_dim) .eq. n_bin(i_dim)) i_bin(i_dim) = n_bin(i_dim) - 1
  end do
  if(n_dim .eq. n_dim_pbc+1) then
    i_dim = n_dim
    i_bin(i_dim)= int((r0(i_part, i_dim) - lowest_fluid_z) /r_bin(i_dim))
    if(i_bin(i_dim) .eq. n_bin(i_dim)) i_bin(i_dim) = n_bin(i_dim) - 1
  end if
  ! increment counter
  bin_fluid(i_bin(1), i_bin(2), i_bin(n_dim_max), 0) = &
       bin_fluid(i_bin(1), i_bin(2), i_bin(n_dim_max), 0) + 1
  ! check overrun
  if(l_binning3d_check_bin_overrun) then
    if(bin_fluid(i_bin(1), i_bin(2), i_bin(n_dim_max), 0).gt.n_bin_fl) then
      write(unit=*, fmt='(a,i4,a,i4,a,i4,a,i3,a,i3)') &
           "ERROR (SR binning3d): bin_fluid(", &
           i_bin(1),",",i_bin(2),",",i_bin(n_dim_max),",0)=", &
           bin_fluid(i_bin(1), i_bin(2), i_bin(n_dim_max), 0), &
           " > n_bin_fl=", n_bin_fl
      stop
    end if
  end if !(l_binning3d_check_bin_overrun)
  ! put particle into bin
  bin_fluid(i_bin(1), i_bin(2), i_bin(n_dim_max), &
       bin_fluid(i_bin(1), i_bin(2), i_bin(n_dim_max), 0)) = i_part
end do ! end loop over fluid particles
if(n_dim .eq. n_dim_pbc+1) then
  ! bins in dimensions which are not in use get i_bin = 0
  i_bin(n_dim_pbc+1:n_dim_max) = 0
  ! top wall
  do i_part = 1+n_mon_tot,n_mon_tot+n_top_wall
    ! check range of coordinates
    if(l_binning3d_check_range) call binning3d_check_range(i_part)
    ! find indices of bins
    do i_dim = 1, n_dim_pbc
      i_bin(i_dim)= int(r0(i_part, i_dim)/r_bin(i_dim))
      if(i_bin(i_dim) .eq. n_bin(i_dim)) i_bin(i_dim) = n_bin(i_dim) - 1
    end do
    ! increment counter
    bin_twall(i_bin(1), i_bin(n_dim_max-1), 0) = &
       bin_twall(i_bin(1), i_bin(n_dim_max-1), 0) + 1
    ! check overrun
    if(l_binning3d_check_bin_overrun) then
      if(bin_twall(i_bin(1), i_bin(2), 0) .gt. n_bin_wa) then
        write(unit=*, fmt='(a,i1,a,i1,a,i3,a,i3)') &
             "ERROR (SR binning3d): bin_twall(", &
             i_bin(1),",",i_bin(2),",0)=", &
             bin_twall(i_bin(1), i_bin(2), 0), &
          " > n_bin_wa=", n_bin_wa
        stop
      end if
    end if !(l_binning3d_check_bin_overrun)
    ! put particle into bin
    bin_twall(i_bin(1), i_bin(2), &
         bin_twall(i_bin(1), i_bin(2), 0)) = i_part
  end do ! loop over top wall
  ! bottom wall
  do i_part = 1+n_mon_tot+n_top_wall, n_part
    ! check range of coordinates
    if(l_binning3d_check_range) call binning3d_check_range(i_part)

    ! find indices of bins
    do i_dim = 1, n_dim_pbc
      i_bin(i_dim)= int(r0(i_part, i_dim)/r_bin(i_dim))
      if(i_bin(i_dim) .eq. n_bin(i_dim)) i_bin(i_dim) = n_bin(i_dim) - 1
    end do
    ! increment counter
    bin_bwall(i_bin(1), i_bin(n_dim_max-1), 0) = &
        bin_bwall(i_bin(1), i_bin(n_dim_max-1), 0) + 1
    ! check overrun
    if(l_binning3d_check_bin_overrun) then
      if(bin_bwall(i_bin(1), i_bin(2), 0) .gt. n_bin_wa) then
        write(unit=*, fmt='(a,i1,a,i1,a,i3,a,i3)') &
             "ERROR (SR binning3d): bin_bwall(", &
             i_bin(1),",",i_bin(2),",0)=", &
             bin_bwall(i_bin(1), i_bin(2), 0), &
             " > n_bin_wa=", n_bin_wa
        stop
      end if
    end if !(l_binning3d_check_bin_overrun)
    ! put particle into bin
    bin_bwall(i_bin(1), i_bin(2), &
         bin_bwall(i_bin(1), i_bin(2), 0)) = i_part
  end do !loop over bottom wall
  n_bin_wa_max = max(maxval(bin_twall(:, :, 0)), &
       maxval(bin_bwall(:, :, 0)), n_bin_wa_max)
end if ! (n_dim .eq. n_dim_pbc+1)
! record the maximum number of entries to optimize memory usage
if(n_mon_tot .gt. 0) then
  n_bin_fl_max = max(maxval(bin_fluid(:, :, :, 0)), n_bin_fl_max)
end if
! check if we have all particles
if(l_binning3d_check_number) then
  if(n_mon_tot .gt. 0) then
    if(sum(bin_fluid(:, :, :, 0)) .ne. n_mon_tot) then
      write(unit=*, fmt='(a)') "ERROR (SR binning3d): 7
      &Incorrect number of particles in bin_fluid:"
      write(unit=*, fmt='(2(a, i11))') &
           "                         Found",  sum(bin_fluid(:, :, :, 0)), &
           ", but n_mon_tot=", n_mon_tot
      stop
    end if
  end if
  if(n_dim .eq. n_dim_pbc+1) then
    if(sum(bin_twall(:, :, 0)) .ne. n_top_wall) then
      write(unit=*, fmt='(a)') "ERROR (SR binning3d): &
           &Incorrect number of particles in bin_twall:"
      write(unit=*, fmt='(2(a, i11))') &
           "                         Found",  sum(bin_twall(:, :, 0)), &
           ", but n_top_wall= ", n_top_wall
      stop
    end if
    if(sum(bin_bwall(:, :, 0)) .ne. n_bottom_wall) then
      write(unit=*, fmt='(a)') "ERROR (SR binning3d): &
           &Incorrect number of particles in bin_bwall:"
      write(unit=*, fmt='(2(a, i11))') &
           "                         Found",  sum(bin_bwall(:, :, 0)), &
           ", but n_bottom_wall= ", n_bottom_wall
      stop
    end if
  end if
end if !(l_binning3d_check_number)
!------------------------------------------------------------------------
! create neighbor lists
!------------------------------------------------------------------------
! rearranging the bin data structure to
! bin(particles, z-index, y-index, x-index) does not improve performance
! For taking periodic boundary conditions into account we prefer if()
! statements over mod() operations, which are slower.
if(n_mon_tot .gt. 0) then
  if(l_fluid_fluid_interaction) then
    ! fluid fluid list
    do i_bin_x = 0, n_bin_used(1)-1 ! loop over all bins in x-direction
      do i_bin_y = 0, n_bin_used(2)-1 ! loop over all bins in y-direction
        ! loop over all bins in z-direction
        do i_bin_z = 0, n_bin_used(n_dim_max)-1
          ! # of particles in bin
          n1_bin_part = bin_fluid(i_bin_x, i_bin_y, i_bin_z, 0)
          do i1_bin_part = 1, n1_bin_part ! loop over all particles in bin
            ! get the index of the i1_bin_part th particle in the bin
```

```
          i_part = bin_fluid(i_bin_x, i_bin_y, i_bin_z, i1_bin_part)
          i_type = type(i_part) ! get particle type
          ! loop over approbriate neighbor bins as set by delta_bin(X)
          do j_bin_x = i_bin_x-1, i_bin_x + delta_bin(1)
              if(n_dim_pbc .ge. 1) then
                ! do not forget periodic boundary conditions
                if(j_bin_x .lt. 0) then
                  k_bin_x = n_bin(1) - 1
                else if(j_bin_x .eq. n_bin(1)) then
                  k_bin_x = 0
                else
                  k_bin_x = j_bin_x
                end if
              else if(n_dim .ge. 1) then
                ! no pbc
                if(j_bin_x.lt.0 .or. j_bin_x.gt.n_bin_used(1)-1) cycle
                k_bin_x = j_bin_x
              else
                ! direction not used
                k_bin_x = 0
              end if
            do j_bin_y = i_bin_y-1, i_bin_y + delta_bin(2)
              if(n_dim_pbc .ge. 2) then
                if(j_bin_y .lt. 0) then
                  k_bin_y = n_bin(2) - 1
                else if(j_bin_y .eq. n_bin(2)) then
                  k_bin_y = 0
                else
                  k_bin_y = j_bin_y
                end if
              else if(n_dim .ge. 2) then
                if(j_bin_y.lt.0 .or. j_bin_y.gt.n_bin_used(2)-1) cycle
                k_bin_y = j_bin_y
              else
                k_bin_y = 0
              end if
              do j_bin_z = i_bin_z-1, i_bin_z + delta_bin(n_dim_max)
                if(n_dim_pbc .eq. n_dim_max) then
                  if(j_bin_z .lt. 0) then
                    k_bin_z = n_bin(n_dim_max) - 1
                  else if(j_bin_z .eq. n_bin(n_dim_max)) then
                    k_bin_z = 0
                  else
                    k_bin_z = j_bin_z
                  end if
                else if(n_dim .eq. n_dim_max) then
                  if(j_bin_z.lt.0 .or. &
                       j_bin_z.gt.n_bin_used(n_dim_max)-1) cycle
                  k_bin_z = j_bin_z
                else
                  k_bin_z = 0
                end if
                ! # of particles in bin
                n2_bin_part = bin_fluid(k_bin_x, k_bin_y, k_bin_z, 0)
                ! loop over all possible neighbors
                do i2_bin_part = 1, n2_bin_part
                  ! get the particle index
                  j_part = bin_fluid(k_bin_x,k_bin_y,k_bin_z,i2_bin_part)
                  ! look at every pair only once
                  if(i_part .lt. j_part) then
                    j_type = type(j_part)
                    ! check whether it is worth making list entry
                    if(distance_squared(i_part, j_part) &
                         .lt. neigh_list_rad_2(i_type, j_type)) then
                      if(ff_list(i_part,0).le.ff_list(j_part,0)) then
                        ff_list(i_part,0) = ff_list(i_part,0) + 1
                        ff_list(i_part,ff_list(i_part,0)) = j_part
                      else
                        ff_list(j_part,0) = ff_list(j_part,0) + 1
                        ff_list(j_part,ff_list(j_part,0)) = i_part
                      end if
                    end if
                  end if
                end do
              end do
            end do
          end do
        end do
      end do
    end do
  end if !(l_fluid_fluid_interaction)
  if(l_fluid_wall_interaction) then
    ! fluid - top wall list entries
    j_type = n_type ! wall particles are of type n_type
    do i_bin_x = n_bin_used(1)-1, bin_lowest_fl_tw(1), -1
      do i_bin_y = n_bin_used(2)-1, bin_lowest_fl_tw(2), -1
        do i_bin_z = n_bin_used(n_dim_max)-1,bin_lowest_fl_tw(n_dim_max),-1
          n1_bin_part = bin_fluid(i_bin_x, i_bin_y, i_bin_z, 0)
          do i1_bin_part = 1, n1_bin_part
            i_part = bin_fluid(i_bin_x, i_bin_y, i_bin_z, i1_bin_part)
            ! loop over approbriate neighbor bins as set by delta_bin(X)
            do j_bin_x = i_bin_x-1, i_bin_x + delta_bin(1)
              if(n_dim_pbc .ge. 1) then
                ! do not forget periodic boundary conditions
                if(j_bin_x .lt. 0) then
                  k_bin_x = n_bin(1) - 1
                else if(j_bin_x .eq. n_bin(1)) then
                  k_bin_x = 0
                else
                  k_bin_x = j_bin_x
                end if
              else ! note: the wall has only directions with pbc
                ! direction not used
                k_bin_x = 0
              end if
              do j_bin_y = i_bin_y-1, i_bin_y + delta_bin(2)
                if(n_dim_pbc .ge. 2) then
                  if(j_bin_y .lt. 0) then
                    k_bin_y = n_bin(2) - 1
                  else if(j_bin_y .eq. n_bin(2)) then
                    k_bin_y = 0
                  else
                    k_bin_y = j_bin_y
                  end if
                else
                  k_bin_y = 0
                end if
                n2_bin_part = bin_twall(k_bin_x, k_bin_y, 0)
                do i2_bin_part = 1, n2_bin_part
                  j_part = bin_twall(k_bin_x, k_bin_y, i2_bin_part)
                  if(distance_squared(i_part, j_part) &
                       .lt. neigh_list_rad_2(i_type, j_type)) then
                    fw_list(i_part,0) = fw_list(i_part,0) + 1
                    fw_list(i_part,fw_list(i_part,0)) = j_part
                  end if
                end do
              end do
            end do
          end do
        end do
      end do
    end do
    ! fluid - bottom wall list entries
    j_type = n_type ! wall particles are of type n_type
    do i_bin_x = 0, bin_highest_fl_bw(1)
      do i_bin_y = 0, bin_highest_fl_bw(2)
        do i_bin_z = 0, bin_highest_fl_bw(n_dim_max)
          n1_bin_part = bin_fluid(i_bin_x, i_bin_y, i_bin_z, 0)
          do i1_bin_part = 1, n1_bin_part
            i_part = bin_fluid(i_bin_x, i_bin_y, i_bin_z, i1_bin_part)
            ! loop over approbriate neighbor bins as set by delta_bin(X)
            do j_bin_x = i_bin_x-1, i_bin_x + delta_bin(1)
              if(n_dim_pbc .ge. 1) then
                ! do not forget periodic boundary conditions
                if(j_bin_x .lt. 0) then
                  k_bin_x = n_bin(1) - 1
                else if(j_bin_x .eq. n_bin(1)) then
                  k_bin_x = 0
                else
                  k_bin_x = j_bin_x
                end if
              else ! note: the wall has only directions with pbc
                ! direction not used
                k_bin_x = 0
              end if
              do j_bin_y = i_bin_y-1, i_bin_y + delta_bin(2)
                if(n_dim_pbc .ge. 2) then
                  if(j_bin_y .lt. 0) then
                    k_bin_y = n_bin(2) - 1
                  else if(j_bin_y .eq. n_bin(2)) then
                    k_bin_y = 0
                  else
                    k_bin_y = j_bin_y
                  end if
                else
                  k_bin_y = 0
                end if
                n2_bin_part = bin_bwall(k_bin_x, k_bin_y, 0)
                do i2_bin_part = 1, n2_bin_part
                  j_part = bin_bwall(k_bin_x, k_bin_y, i2_bin_part)
                  if(distance_squared(i_part, j_part) &
                       .lt. neigh_list_rad_2(i_type, j_type)) then
                    fw_list(i_part,0) = fw_list(i_part,0) + 1
                    fw_list(i_part,fw_list(i_part,0)) = j_part
                  end if
                end do
              end do
            end do
          end do
        end do
      end do
    end do
  end if !(l_fluid_wall_interaction)
end if !(n_mon_tot .gt. 0)

! generate top-wall bottom-wall list only if interaction between the walls
if(l_allow_wall_wall_interaction) then
  if(l_wall_wall_interaction) then
    i_type = n_type
    j_type = n_type
    do i_bin_x = 0, n_bin(1)-1
      do i_bin_y = 0, n_bin(2)-1
        n1_bin_part = bin_twall(i_bin_x, i_bin_y, 0)
        do i1_bin_part = 1, n1_bin_part
          i_part = bin_twall(i_bin_x, i_bin_y, i1_bin_part)
          ! wall lists are indexed with wall particle index
          i_wall = i_part - n_mon_tot
          ! loop over approbriate neighbor bins as set by delta_bin(X)
          do j_bin_x = i_bin_x-1, i_bin_x + delta_bin(1)
            if(n_dim_pbc .ge. 1) then
              ! do not forget periodic boundary conditions
              if(j_bin_x .lt. 0) then
                k_bin_x = n_bin(1) - 1
              else if(j_bin_x .eq. n_bin(1)) then
                k_bin_x = 0
              else
                k_bin_x = j_bin_x
              end if
            else ! note: the wall has only directions with pbc
              ! direction not used
              k_bin_x = 0
            end if
            do j_bin_y = i_bin_y-1, i_bin_y + delta_bin(2)
              if(n_dim_pbc .ge. 2) then
                if(j_bin_y .lt. 0) then
                  k_bin_y = n_bin(2) - 1
                else if(j_bin_y .eq. n_bin(2)) then
                  k_bin_y = 0
                else
                  k_bin_y = j_bin_y
                end if
              else
                k_bin_y = 0
              end if
              n2_bin_part = bin_bwall(k_bin_x, k_bin_y, 0)
              do i2_bin_part = 1, n2_bin_part
                j_part = bin_bwall(k_bin_x, k_bin_y, i2_bin_part)
                if(distance_squared(i_part, j_part) &
                     .lt. neigh_list_rad_2(i_type, j_type)) then
                  ww_list(i_wall, 0) = ww_list(i_wall, 0) + 1
                  ww_list(i_wall, ww_list(i_wall, 0)) = j_part
                end if
              end do
            end do
          end do
        end do
      end do
    end do
  end if ! (l_wall_wall_interaction)
end if ! (l_allow_wall_wall_interaction)
! catch overruns of neighbor lists: check if the allocated size for the
! neighbor lists is sufficient.  In fact, here is too late to do that, but
! doing it inside all the nested loops is too expensive.  Some compilers
! provide run time array bound checking.
if(l_binning3d_check_list_overrun) then
  do i_part = 1, n_mon_tot
    ! fluid - fluid
    if(l_fluid_fluid_interaction &
         .and. ff_list(i_part, 0) .gt. n_neigh_ff) then
      write(unit=*, fmt='(a,i5,a,i5,a,i5)') &
           "ERROR (SR binning3d): ff_list(", i_part, ", 0)=", &
           ff_list(i_part, 0), " .gt. n_neigh_ff=", n_neigh_ff
      write(unit=*, fmt='(a, i5, a)') &
           "Dumping ff_list(", i_part, "):"
      print *, ff_list(i_part, :)
      stop
    end if
    ! fluid - wall
    if(l_fluid_wall_interaction &
         .and. fw_list(i_part, 0) .gt. n_neigh_fw) then
      write(unit=*, fmt='(a,i5,a,i5,a,i5)') &
           "ERROR (SR binning3d): fw_list(", i_part, ", 0)=", &
           fw_list(i_part, 0), " .gt. n_neigh_fw=", n_neigh_fw
      write(unit=*, fmt='(a, i5, a)') &
           "Dumping fw_list(", i_part, "):"
      print *, fw_list(i_part, :)
      stop
    end if
  end do ! loop over fluid
  if(l_allow_wall_wall_interaction &
       .and. l_wall_wall_interaction) then
    do i_wall = 1, n_top_wall
      ! top wall - bottom wall
      if(ww_list(i_wall, 0) .gt. n_neigh_ww) then
        write(unit=*, fmt='(a,i5,a,i5,a,i5)') &
             "ERROR (SR binning3d): ww_list(", i_wall, ", 0)=", &
             ww_list(i_wall, 0), " .gt. n_neigh_ww=", n_neigh_ww
```

```
          write(unit=*, fmt='(a, i5, a)') &
               "Dumping ww_list(", i_wall, "):"
          print *, ww_list(i_wall, :)
          stop
        end if
      end do
    end if !(l_wall_wall_interaction)
  end if !(l_binning3d_check_list_overrun)
  ! find maximum entry for optimizing memory usage
  do i_part = 1, n_mon_tot
    if(l_fluid_fluid_interaction) &
         n_neigh_ff_max = max(n_neigh_ff_max, ff_list(i_part,0))
    if(l_fluid_wall_interaction) &
         n_neigh_fw_max = max(n_neigh_fw_max, fw_list(i_part,0))
  end do
  if(l_allow_wall_wall_interaction .and. l_wall_wall_interaction) then
    do i_wall = 1, n_top_wall
      n_neigh_ww_max = max(n_neigh_ww_max, ww_list(i_wall, 0))
    end do
  end if
  if(l_debug_binning3d) then
    write(unit=*, fmt='(a)') "DEBUG: Leaving subroutine binning3d"
  end if
end subroutine binning3d
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! checks if there are particles which penetrated the walls, i.e. if there is
! a fluid particle under the bottom wall or above the top wall in case there
! is fluid.  If there's no fluid we test wall particles only.
! these tests are heuristic and should be modified for special simulations.
subroutine wall_penetration
  ! signal error and exit flag
  logical :: l_exit
  ! highest z-coordinate of top and lowest z-coordinate of bottom wall
  real(kind=dp) :: highest_twall_z, lowest_twall_z, &
                   highest_bwall_z, lowest_bwall_z
  if(n_dim .ne. n_dim_pbc+1) then
    write(unit=*, fmt='(a)') "ERROR (SR wall_penetration): &
         &There are no walls"
    stop
  end if
  l_exit = .FALSE.
  highest_twall_z = maxval(r0(n_mon_tot+1:n_mon_tot+n_top_wall, n_dim))
  lowest_twall_z  = minval(r0(n_mon_tot+1:n_mon_tot+n_top_wall, n_dim))
  highest_bwall_z = maxval(r0(n_mon_tot+n_top_wall+1:n_part, n_dim))
  lowest_bwall_z  = minval(r0(n_mon_tot+n_top_wall+1:n_part, n_dim))
  if(n_mon_tot .gt. 0) then
    if(highest_fluid_z .gt. highest_twall_z) then
      write(unit=*, fmt='(2(a, g13.6))') &
           "ERROR (SR wall_penetration): highest_fluid_z= ", &
           highest_fluid_z, " > highest_twall_z = ", highest_twall_z
      l_exit = .TRUE.
    else if(lowest_fluid_z .lt. lowest_bwall_z) then
      write(unit=*, fmt='(2(a, g13.6))') &
           "ERROR (SR wall_penetration): lowest_fluid_z= ", &
           lowest_fluid_z, " < lowest_bwall_z = ", lowest_bwall_z
      l_exit = .TRUE.
    end if
  else ! no fluid
    if(highest_bwall_z .gt. highest_twall_z) then
      write(unit=*, fmt='(2(a, g13.6))') &
           "ERROR (SR wall_penetration): highest_bwall_z= ", &
           highest_bwall_z, " > highest_twall_z = ", highest_twall_z
      l_exit = .TRUE.
    else if(lowest_twall_z .lt. lowest_bwall_z) then
      write(unit=*, fmt='(2(a, g13.6))') &
           "ERROR (SR wall_penetration): lowest_twall_z= ", &
           lowest_twall_z, " < lowest_bwall_z = ", lowest_bwall_z
      l_exit = .TRUE.
    end if
  end if !(n_mon_tot .gt. 0)
  ! print some hopefully useful information on exit
  if(l_exit) then
    write(unit=*, fmt='(a)') &
         "      Penetration of walls is likely."
    write(unit=*, fmt='(3a)') &
         "MESSAGE: In n_dim-direction we are in ", &
         mode_strings(f_twall(n_dim)), "with values"
    select case(f_twall(n_dim))
    case(velocity_mode)
      write(unit=*, fmt='(a, g13.6)') &
           "         v_spring_twall(n_dim)=", v_spring_twall(n_dim)
    case(force_mode)
      write(unit=*, fmt='(2(a, g13.6))') &
           "         ext_force_twall(n_dim)=", ext_force_twall(n_dim), &
           ", ramp_force_twall(n_dim)=", ramp_force_twall(n_dim)
    case(spring_mode)
      write(unit=*, fmt='(2(a, g13.6))') &
           "         v_spring_twall(n_dim)=", v_spring_twall(n_dim), &
           ", k_spring_twall(n_dim)=", k_spring_twall(n_dim)
    end select
    write(unit=*, fmt='(a, g13.6)') &
         "         r0_twall(n_dim) - r0_bwall(n_dim)=", &
         r0_twall(n_dim) - r0_bwall(n_dim)
    write(unit=*, fmt='(a, i11, a, g13.6)') &
         "         i_time=", i_time, ", r_2_min=", r_2_min
    write(unit=*, fmt='(a, g13.6)') &
         "         max_encountered_correction=", max_encountered_correction
    ! dump configuration
    call particle_positions_out("wall_penetration_", i_time)
    stop
  end if ! penetration
end subroutine wall_penetration
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! the predictor routine of the predictor corrector algorithm
subroutine predict
  integer :: i_wall ! loop over wall particles
  integer :: i_part ! loop over particles
  integer :: i_dim
  ! loops over predictor-corrector coefficients
  integer :: i_order, j_order
  ! helper vector
  real(kind=dp), dimension(n_dim) :: delta_r
  if(l_debug_predict) then
    print *, "DEBUG: Entering subroutine predict"
  end if
  ! calculate the spring position if in spring mode
  do i_dim = 1, n_dim
    if(f_twall(i_dim).eq.spring_mode) then
      r0_spring_twall(i_dim) = r0_spring_twall(i_dim) &
           + v_spring_twall(i_dim) * dt
    end if
  end do
  ! predict coordinates
  do j_order = 1, n_order
    do i_part = 1, n_moving
      r0(i_part, :) = r0(i_part, :) &
           + predict_coef(0, j_order) * rx(i_part, :, j_order)
    end do
  end do

  ! predict derivatives
  do i_order = 1, n_order-1
    do j_order = i_order + 1, n_order
      do i_part = 1, n_moving
        rx(i_part, :, i_order) = rx(i_part, :, i_order) &
             + predict_coef(i_order, j_order) * rx(i_part, :, j_order)
      end do
    end do
  end do
  if(n_dim .eq. n_dim_pbc+1) then
    ! initialize
    delta_r(:) = 0.0_dp
    ! in velocity mode, the velocity of the top wall is stored in
    ! rx_twall(:, :).
    do j_order = 1, n_order
      ! delta_r has meaning of (differential) displacement of top wall
      ! (we need the displacement later on for the prediction of the
      ! equilibrium positions, so we store it)
      delta_r(:) = delta_r(:) &
           + predict_coef(0, j_order) * rx_twall(:, j_order)
    end do
    ! predict equilibrium position of top wall
    r0_twall(:) = r0_twall(:) + delta_r(:)
    ! predict equilibrium particle positions of top wall
    do i_wall = 1, n_top_wall
      i_part = n_mon_tot + i_wall
      r_wall_equi(i_wall, :) = r_wall_equi(i_wall, :) + delta_r(:)
    end do
    ! predict derivatives
    do i_order = 1, n_order-1
      do j_order = i_order + 1, n_order
        rx_twall(:, i_order) = rx_twall(:, i_order) &
             + predict_coef(i_order, j_order) * rx_twall(:, j_order)
      end do
    end do
    ! if the wall particles are fixed, then the actual positions are the
    ! equilibrium positions
    if(f_wall_fix.eq.1) then
      r0(n_mon_tot+1:n_mon_tot+n_top_wall, :) = r_wall_equi(1:n_top_wall, :)
    end if
    ! the bottom wall equilibrium particle positions are not predicted,
    ! because they are not moved in any direction.
    ! There is just a thermostatting force acting upon them (if they are not
    ! fixed anyways)
  end if ! walls
  ! add contribution from random forces from thermostat
  !-----------------------------------------------------
  if(f_thermostat_mode .gt. 1) then
    ! if(f_wall_fix.eq.1) then the wall particles are not touched
    do i_part = 1, n_moving
      ! convert random force into acceleration times (dt_2/2.0_dp)
      force_random(i_part, :) = &
           (force_random(i_part, :) / mass(i_part)) * (dt_2/2.0_dp)
      ! positions
      r0(i_part, :) = r0(i_part, :) + force_random(i_part, :)
      ! velocity * dt of particles
      rx(i_part, :, 1) = rx(i_part, :, 1) + 2.0_dp*force_random(i_part, :)
    end do ! loop over moving particles
  end if !(l_thermostat_on)
  ! impose periodic boundary conditions for fluid particles in x, y
  !================================================================
  ! Note that after applying the periodic boundary conditions the coordinates
  ! lie in the closed interval [0, L], L being the box-length in this
  ! direction.
  ! Suppose there is a coordinate r = -10^-30 and we wanted to enforce
  ! r \in [0, L[ (halfopen):
  ! Clearly, r .lt. 0.0_dp is .TRUE., so L gets added to r, but the result is
  ! seen as L (if L is finite), so r .eq. L is true afterwards, if we only
  ! have 15 digits precision.
  ! We can distuinguish tiny() = 10^-308 from 0.0, but only epsilon() from L.
  ! The problem would disappear if r \in [a, b[ and epsilon(a)==epsilon(b).
  ! If one tested r \in [0, L[ (halfopen) one would get an error message.
  ! That's why we allow the closed interval.  Both borders of the interval
  ! are thus equivalent.
  do i_dim = 1, n_dim_pbc
    do i_part = 1, n_mon_tot
      if(r0(i_part,i_dim).gt.boundary(i_dim)) then
        r0(i_part,i_dim) = r0(i_part,i_dim) - boundary(i_dim)
        pbc_count(i_part,i_dim) = pbc_count(i_part,i_dim) + 1
      else if(r0(i_part,i_dim).lt.0.0_dp) then
        r0(i_part,i_dim) = r0(i_part,i_dim) + boundary(i_dim)
        pbc_count(i_part,i_dim) = pbc_count(i_part,i_dim) - 1
      end if
    end do
  end do
  ! impose periodic boundary conditions for wall particles in x, y
  do i_dim = 1, n_dim_pbc
    do i_wall = 1, n_wall
      i_part = i_wall + n_mon_tot
      if(r0(i_part,i_dim).gt.boundary(i_dim)) then
        r0(i_part,i_dim) = r0(i_part,i_dim) - boundary(i_dim)
        r_wall_equi(i_wall,i_dim) = &
             r_wall_equi(i_wall,i_dim) - boundary(i_dim)
        pbc_count(i_part,i_dim) = pbc_count(i_part,i_dim) + 1
      else if(r0(i_part,i_dim).lt.0.0_dp) then
        r0(i_part,i_dim) = r0(i_part,i_dim) + boundary(i_dim)
        r_wall_equi(i_wall,i_dim) = &
             r_wall_equi(i_wall,i_dim) + boundary(i_dim)
        pbc_count(i_part,i_dim) = pbc_count(i_part,i_dim) - 1
      end if
    end do
  end do
  ! check if the coordinates really are in the proper range.  If that's not
  ! true, then a particle has moved farther than one boundary, which is bad.
  if(l_predict_check_pbc) then
    ! fluid
    do i_dim = 1, n_dim_pbc
      do i_part = 1, n_mon_tot
        if(r0(i_part,i_dim).gt.boundary(i_dim)) then
          write(unit=*, fmt='(a,i5,a,i2,a,e14.6,a,i1,a,g14.6,a,i11)') &
               "ERROR (SR predict): Fluid position r0(", i_part, ",", &
               i_dim, ")=", r0(i_part,i_dim), " > boundary(", i_dim, ")=", &
               boundary(i_dim), " at MD step ", i_time
          stop
        else if(r0(i_part,i_dim).lt.0.0_dp) then
          write(unit=*, fmt='(a, i5, a, i2, a, g14.6, a, a, i11)') &
               "ERROR (SR predict): Fluid position r0(", i_part, ",", &
               i_dim, ")=", r0(i_part,i_dim), " < 0.0", &
               " at MD step ", i_time
          stop
        end if
      end do
    end do
    ! wall
    do i_dim = 1, n_dim_pbc
      do i_wall = 1, n_wall
        i_part = i_wall + n_mon_tot
        if(r0(i_part,i_dim).gt.boundary(i_dim)) then
          write(unit=*, fmt='(a,i5,a,i2,a,e14.6,a,i1,a,g14.6,a,i11)') &
               "ERROR (SR predict): Wall position r0(", i_part, ",", &
               i_dim, ")=", r0(i_part,i_dim), " > boundary(", i_dim, ")=", &
```

```
              boundary(i_dim), " at MD step ", i_time
            stop
          else if(r0(i_part,i_dim).lt.0.0_dp) then
            write(unit=*, fmt='(a, i5, a, i2, a, g14.6, a, a, i11)') &
                 "ERROR (SR predict): Wall position r0(", i_part, ",", &
                 i_dim, ")=", r0(i_part,i_dim), " < 0.0", &
                 " at MD step ", i_time
            stop
          end if
        end do
      end do
    end if !(l_predict_check_pbc)
    if(l_debug_predict) then
      print *, "DEBUG: Leaving subroutine predict"
    end if
  end subroutine predict
  !-----------------------------------------------------------------------------

  !-----------------------------------------------------------------------------
  ! thermostatting subroutine.
  ! Sets the current temperature and effective minimum distance.
  ! Uses the Langevin thermostat for thermostatting.
  subroutine thermostat
    integer :: i_part, i_wall ! loops over particles
    integer :: i_neigh        ! loop over nearest neighbors
    integer :: i_type         ! loop over particle types
    integer :: i_dim
    ! center of mass velocities
    real(kind=dp), dimension(1:n_dim) :: velocity_cm_twall, velocity_cm_bwall,&
         velocity_cm_fluid, vel_cm_t_b_wall
    ! average velocity of nearest neighbors
    real(kind=dp) :: velocity_local_neighbors
    if(l_debug_thermostat) then
      print *, "DEBUG: Entering subroutine thermostat"
    end if
    ! initialize for safety
!    velocity_cm_twall(:)        = 1.0E100_dp
!    velocity_cm_bwall(:)        = 1.0E100_dp
!    velocity_cm_fluid(:)        = 1.0E100_dp
!    velocity_local_neighbors(:) = 1.0E100_dp
!    vel_cm_t_b_wall(:)          = 1.0E100_dp
    ! define temperature
    if((r_time.lt.temp_time).and.(temp_init.ne.temp_final)) then
      temp = temp_init + r_time*(temp_final-temp_init)/temp_time
    else
      temp = temp_final
    end if
    ! define effective minimium distance (to be used in the interaction subrou-
    ! tines)
    if(l_r_2_min_finite) then
      if(r_time.lt.r_2_min_time) then
        r_2_min = (1.0_dp-r_time/r_2_min_time)*r_2_min_init
      else
        l_r_2_min_finite = .FALSE.
        r_2_min = 0.0_dp
      end if
    end if
    ! get distribution width of random force (not random acceleration)
    ! as temp is constant most of the time, it would be nicer not to compute
    ! the width every step
    do i_type = 1, n_type
      ! Gaussian distribution of random variable
      rf_width(i_type) = sqrt(2.0_dp * temp * mass_type(i_type) &
           * friction(i_type)/dt)
      ! for uniformly distributed -1 < r.v. < 1 (instead of Gaussians)
      rf_width(i_type) = sqrt3*rf_width(i_type)
    end do
    ! set all forces to zero first.  Then in the next step, thermostatting
    ! forces are applied where appropriate.
    total_force_twall(:) = 0.0_dp
    force_determ(:, :)   = 0.0_dp
    force_random(:, :)   = 0.0_dp
    ! apply friction force on the top wall, if selected
    ! The reference frame doesn't matter here, as velocity enters linearly
    if(f_friction_on_twall.eq.1) then
      total_force_twall(:) = &
           - friction_constant_twall * mass_twall * (rx_twall(:, 1)/dt)
    end if
    ! we have different thermostatting modes
    select case (f_thermostat_mode)
    case(0)
      ! no thermostatting at all
    case(1)
      ! just friction acting on all unfixed particles.
      ! if we are freezing the system, the are no forces pulling the system
      ! in x or y-direction, no force ramp and no moved external spring
      ! we don't go to the inertial system and just apply the friction force
      ! to slow down the system
      do i_part = 1, n_moving
        force_determ(i_part, :) = - mass(i_part) &
             * friction(type(i_part)) * (rx(i_part,: ,1)/dt)
      end do
    case(2)
      ! thermostat all unfixed particles (fluid and unfixed wall particles)
      ! in the inertial system defined by the center of mass motion.
      ! For thicker films one should define layer velocities and thermostat
      ! each layer in the inertial system of the layer.
      do i_dim = 1, n_dim
        ! draw random numbers for this dimension
        call ranlux(random, n_moving)
        ! find out averaged velocity of the fluid and top and bottom wall
        ! particles, that is the velocities of the centers of mass.
        if(n_mon_tot .gt. 0) then
          velocity_cm_fluid(i_dim) = sum(rx(1:n_mon_tot, i_dim, 1)) &
               / (dt * real(n_mon_tot, kind=dp))
        end if
        if(f_wall_fix .ne. 1) then
          velocity_cm_twall(i_dim) = sum(rx(n_mon_tot+1:n_mon_tot+n_top_wall, &
               i_dim, 1)) / (dt * real(n_top_wall, kind=dp))
          velocity_cm_bwall(i_dim) = &
               sum(rx(n_mon_tot+n_top_wall+1:n_mon_tot+n_wall , &
               i_dim, 1)) / (dt * real(n_bottom_wall, kind=dp))
        else
          velocity_cm_twall(i_dim) = rx_twall(i_dim, 1)
          ! bottom wall doesn't move anyways
          velocity_cm_bwall(i_dim) = 0.0_dp
        end if
        ! fluid
        do i_part = 1, n_mon_tot
          ! random force
          force_random(i_part, i_dim) = (2.0_dp*random(i_part)-1.0_dp) &
               * rf_width(type(i_part))
          ! friction force (as one part of the deterministic force acting
          ! upon a particle)
          force_determ(i_part, i_dim) = &
               - mass(i_part) * friction(type(i_part)) &
               * (rx(i_part,i_dim,1)/dt - velocity_cm_fluid(i_dim))
        end do
        ! if the wall particles are not fixed, thermostat them
        if(f_wall_fix .ne. 1) then
          ! top wall
          do i_part = n_mon_tot+1, n_mon_tot+n_top_wall
            ! random force
            force_random(i_part, i_dim) = (2.0_dp*random(i_part)-1.0_dp) &
                 * rf_width(type(i_part))
            ! friction force (as one part of the deterministic force acting
            ! upon a particle)
            ! do thermostatting in the inertial system
            force_determ(i_part, i_dim) = &
                 -mass(i_part) * friction(type(i_part)) &
                 * (rx(i_part,i_dim,1)/dt - velocity_cm_twall(i_dim))
          end do
          ! bottom wall
          do i_part = n_mon_tot+n_top_wall+1, n_part
            ! random force
            force_random(i_part, i_dim) = (2.0_dp*random(i_part)-1.0_dp) &
                 * rf_width(type(i_part))
            ! friction force (as one part of the deterministic force acting
            ! upon a particle)
            force_determ(i_part, i_dim) = &
                 -mass(i_part) * friction(type(i_part)) &
                 * (rx(i_part,i_dim,1)/dt - velocity_cm_bwall(i_dim))
          end do
        end if !(f_wall_fix .ne. 1)
      end do ! loop over spacial dimensions
    case(3)
      ! thermostat only wall particles in inertial system, as defined by the
      ! motions of the center of mass
      ! (makes sense only if wall atoms are not fixed)
      do i_dim = 1, n_dim
        ! draw random numbers for this direction
        call ranlux(random, n_wall)
        ! find out averaged velocity of the top and bottom wall particles,
        ! that is the velocity of the centers of mass
        velocity_cm_twall(i_dim) = sum(rx(n_mon_tot+1:n_mon_tot+n_top_wall, &
             i_dim, 1)) / (dt * real(n_top_wall, kind=dp))
        velocity_cm_bwall(i_dim) = &
             sum(rx(n_mon_tot+n_top_wall+1:n_mon_tot+n_wall , &
             i_dim, 1)) / (dt * real(n_top_wall, kind=dp))
        ! top wall
        do i_wall = 1, n_top_wall
          ! particle index
          i_part = i_wall + n_mon_tot
          ! random force
          force_random(i_part, i_dim) = &
               (2.0_dp*random(i_wall)-1.0_dp) * rf_width(type(i_part))
          ! friction force (as one part of the deterministic force acting
          ! upon a particle)
          force_determ(i_part, i_dim) = &
               -mass(i_part) * friction(type(i_part)) &
               * (rx(i_part,i_dim ,1)/dt - velocity_cm_twall(i_dim))
        end do
        ! bottom wall
        do i_wall = n_top_wall+1, n_wall
          ! particle index
          i_part = i_wall + n_mon_tot
          ! random force
          force_random(i_part, i_dim) = (2.0_dp*random(i_wall)-1.0_dp) &
               * rf_width(type(i_part))
          ! friction force (as one part of the deterministic force acting
          ! upon a particle)
          force_determ(i_part, i_dim) = &
               -mass(i_part) * friction(type(i_part)) &
               * (rx(i_part,i_dim ,1)/dt - velocity_cm_bwall(i_dim))
        end do
      end do ! loop over spacial directions
    case(4)
      ! thermostat in directions with no strain in the laboratory system.
      ! The problem with this method is that the thermostat is not effective
      ! enough at "high" shear rates.
      do i_dim = 1, n_dim
        if( & ! velocity_mode and velocity is zero
             ((f_twall(i_dim).eq.velocity_mode) &
             .and.(v_spring_twall(i_dim).eq.0.0_dp))&
             .or. & ! just a force, no force ramp
             ! or negative force ramp in z-direction (pressure increase)
             ((f_twall(i_dim).eq.force_mode) &
             .and.(ramp_force_twall(i_dim).eq.0.0_dp &
             .or. ((ramp_force_twall(i_dim).lt.0.0_dp).and.(i_dim.eq.n_dim))))&
             .or. & ! spring mode when spring is not pulled
             ((f_twall(i_dim).eq.spring_mode) &
             .and.(v_spring_twall(i_dim).eq.0.0_dp)) &
             ) then
          call ranlux(random, n_moving)
          do i_part = 1, n_moving
            ! random force
            force_random(i_part, i_dim) = &
                 (2.0_dp*random(i_part)-1.0_dp) * rf_width(type(i_part))
            ! friction force (as one part of the deterministic force
            ! acting upon a particle)
            force_determ(i_part, i_dim) = - mass(i_part) &
                 * friction(type(i_part)) * (rx(i_part,i_dim,1)/dt)
          end do
        end if !(thermostat in this direction)
      end do ! loop over spacial directions
    case(5)
      ! thermostat wall particles in inertial system, as defined by the
      ! motion of the lattice equilibrium positions, in all directions.
      ! (makes sense only if wall atoms are not fixed)
      do i_dim = 1, n_dim
        ! draw random numbers for this direction
        call ranlux(random, n_wall)
        ! top wall
        do i_wall = 1, n_top_wall
          ! particle index
          i_part = i_wall + n_mon_tot
          ! random force
          force_random(i_part, i_dim) = &
               (2.0_dp * random(i_wall) - 1.0_dp) &
               * rf_width(type(i_part))
          ! friction force (as one part of the deterministic force acting
          ! upon a particle)
          force_determ(i_part, i_dim) = &
               -mass(i_part) * friction(type(i_part)) &
               * ((rx(i_part,i_dim, 1) - rx_twall(i_dim, 1))/dt)
        end do
        ! bottom wall
        do i_wall = n_top_wall+1, n_wall
          ! particle index
          i_part = i_wall + n_mon_tot
          ! random force
          force_random(i_part, i_dim) = &
               (2.0_dp * random(i_wall) - 1.0_dp) &
               * rf_width(type(i_part))
          ! friction force (as one part of the deterministic force acting
          ! upon a particle)
          ! rx_bwall(:) is constantly 0.0 anyways at present.
          force_determ(i_part, i_dim) = &
               -mass(i_part) * friction(type(i_part)) &
               * (rx(i_part,i_dim, 1)/dt)
```

```
    end do
  end do ! end loop over spacial dimensions
case(6)
  ! thermostat top and bottom wall only in "local" inertial system.
  ! Use the six nearest neighbors to define a "local" inertial system for
  ! each particle.  Problem: This choice of "local" has to be justified.
  do i_dim = 1, n_dim
    ! draw random numbers for this direction
    call ranlux(random, n_wall)
    ! top wall
    do i_wall = 1, n_wall
      ! particle index
      i_part = i_wall + n_mon_tot
      ! find out averaged velocity of the six nearest neighbors
      velocity_local_neighbors = 0.0_dp
      do i_neigh = 1, 6
        velocity_local_neighbors = velocity_local_neighbors &
             + rx(fk_neighbors(i_neigh, i_wall), i_dim, 1)
      end do
      velocity_local_neighbors = velocity_local_neighbors / (6.0_dp * dt)
      ! random force
      force_random(i_part, i_dim) = &
           (2.0_dp*random(i_wall)-1.0_dp) * rf_width(type(i_part))
      force_determ(i_part, i_dim) = &
           -mass(i_part) * friction(type(i_part)) &
           * (rx(i_part,i_dim ,1)/dt - velocity_local_neighbors)
    end do ! loop over all wall particles
  end do ! loop over spacial directions
case(7)
  ! thermostat fluid particles in the center of mass inertial system of
  ! top and bottom wall.
  ! For thermostatting of thin films with fixed walls.
  ! find out averaged velocity of the fluid and top and bottom wall
  ! Note: the bottom wall is fixed, therefore we divide by 2.
  vel_cm_t_b_wall(:) = rx_twall(:, 1)/(2.0_dp*dt)
  do i_dim = 1, n_dim
    ! draw random numbers for this dimension
    call ranlux(random, n_mon_tot)
    do i_part = 1, n_mon_tot
      ! random force
      force_random(i_part, i_dim) = (2.0_dp*random(i_part)-1.0_dp) &
           * rf_width(type(i_part))
      ! friction force (as one part of the deterministic force acting
      ! upon a particle)
      force_determ(i_part, i_dim) = &
           - mass(i_part) * friction(type(i_part)) &
           * (rx(i_part,i_dim,1)/dt - vel_cm_t_b_wall(i_dim))
    end do
  end do ! dimensions
case default
  write(unit=*, fmt='(a,i2,a)') &
       "ERROR (SR thermostat): Mode f_thermostat_mode=", &
       f_thermostat_mode, "not recognized"
  stop
end select
! DEBUG
! write out the average velocity of the top wall atoms and compare to the
! rigid top wall lattice velocity.  For some thermostat modes (with no
! fluid between the walls) they can oscillate against each other which
! gives rise to artefacts.
if(.FALSE.) then
  do i_dim = 1, n_dim
    velocity_cm_twall(i_dim) = sum(rx(n_mon_tot+1:n_mon_tot+n_top_wall, &
         i_dim, 1)) / (dt * real(n_top_wall, kind=dp))
  end do
  write(69, '(7e14.6)') r_time, (velocity_cm_twall(i_dim),i_dim=1,n_dim), &
       (rx_twall(i_dim, 1)/dt, i_dim=1, n_dim)
end if
if(l_debug_thermostat) then
  print *, "DEBUG: Leaving subroutine thermostat"
end if
end subroutine thermostat
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! the corrector routine of the predictor corrector algorithm
subroutine corrector
  integer :: i_wall ! loop over wall particles
  integer :: i_part ! loop over particles
  integer :: i_dim
  ! loop over predictor-corrector coefficients
  integer :: i_order
  ! helper vector
  real(kind=dp), dimension(n_dim) :: delta_r
  ! helper variable holding the difference of the predicted and real
  ! acceleration of the top wall.  This is done to preserve total_force_twall
  ! as force.
  real(kind=dp), dimension(n_dim) :: diff_pred_real_acc_tw
  ! helper variable holding the difference of the predicted and real
  ! acceleration of the particles.
  ! This is done to preserve force_determ(:, :) as force.
  real(kind=dp), dimension(1:n_moving, 1:n_dim) :: diff_pred_real_acc
  real(kind=dp) :: max_correction
  if(l_debug_corrector) then
    print *, "DEBUG: Entering subroutine corrector"
  end if
  ! field diff_pred_real_acc(:, :) gets meaning of
  ! predicted acceleration - real acceleration (times dt_2 / 2.0_dp)
  do i_part = 1, n_moving
    diff_pred_real_acc(i_part, :) = - rx(i_part, :, 2) &
         + (force_determ(i_part, :) / mass(i_part)) * (dt_2 / 2.0_dp)
  end do
  ! get maximal value of the particle position corrections
  if(i_time .gt. n_relax) then
    max_correction = &
         maxval(abs(diff_pred_real_acc(1:n_moving, :))) * correct_coef(0)
    max_encountered_correction = max(max_encountered_correction, &
         max_correction)
    if(max_correction .gt. max_allowed_correction) then
      write(unit=*, fmt='(a)') &
           "WARNING (SR corrector): Maximal correction of particle position"
      write(unit=*, fmt='(2(a, g13.6, a, i12))') &
           "        ", max_correction, " > max_allowed_correction=", &
           max_allowed_correction, " at MDS", i_time
    end if
  end if
  ! correct coordinates
  do i_part = 1, n_moving
    r0(i_part, :) = r0(i_part, :) &
         + correct_coef(0) * diff_pred_real_acc(i_part, :)
  end do
  ! correct derivatives
  do i_order = 1, n_order
    do i_dim = 1, n_dim
      do i_part = 1, n_moving
        rx(i_part, i_dim, i_order) = rx(i_part, i_dim, i_order) &
             + correct_coef(i_order) * diff_pred_real_acc(i_part, i_dim)
      end do
    end do
  end do
  if(n_dim .eq. n_dim_pbc+1) then
    ! do the same for the top wall
    diff_pred_real_acc_tw(:) = - rx_twall(:, 2) &
         + (total_force_twall(:) / mass_twall) * (dt_2 / 2.0_dp)
    do i_order = 1, n_order
      do i_dim = 1, n_dim
        ! do not correct (i.e. change) derivative when in velocity mode,
        !  which says that acceleration is zero, and velocity is constant.
        if(f_twall(i_dim) .ne. velocity_mode) then
          !================================
          rx_twall(i_dim, i_order) = rx_twall(i_dim, i_order) &
               + correct_coef(i_order) * diff_pred_real_acc_tw(i_dim)
        end if
      end do
    end do
    ! delta_r is the correction of the (differential) displacement of the
    ! top wall.
    ! If we are in velocity mode, the top wall is pulled with a constant
    ! velocity anyways, and derivatives are not changed.
    do i_dim = 1, n_dim
      if(f_twall(i_dim) .ne. velocity_mode) then
        delta_r(i_dim) = correct_coef(0) * diff_pred_real_acc_tw(i_dim)
      else
        delta_r(i_dim) = 0.0_dp
      end if
    end do
    ! correct equilibrium positions of top wall
    r0_twall(:) = r0_twall(:) + delta_r(:)
    do i_wall = 1, n_top_wall
      r_wall_equi(i_wall, :) = r_wall_equi(i_wall, :) + delta_r(:)
    end do
    ! if the wall particles are fixed, then the actual positions are the
    ! equilibrium positions.  The "else" part is treated implicitely in
    ! the correction of the n_moving particles.  If the wall atoms are
    ! fixed, we have to correct the equilibrium positions of the top wall
    ! particles here.
    if(f_wall_fix.eq.1) then
      r0(n_mon_tot+1:n_mon_tot+n_top_wall, :) = r_wall_equi(1:n_top_wall, :)
    end if
  end if ! walls
  ! define unfolded coordinates
  do i_dim = 1, n_dim_pbc
    r0_unfolded(:, i_dim) = r0(:, i_dim) &
         + real(pbc_count(:, i_dim), kind=dp) * boundary(i_dim)
  end do
  r0_unfolded(:, n_dim_pbc+1:n_dim) = r0(:, n_dim_pbc+1:n_dim)
  if(l_debug_corrector) then
    print *, "DEBUG: Leaving subroutine corrector"
  end if
end subroutine corrector
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! computes the minimal distance a particle has from any neighbor, writes
! out the distance, the particle index, and the kind of particle
subroutine min_distance
  real(kind=dp):: minimum, average, r_dummy
  integer :: i_part, j_part, min_index_i, min_index_j, counter
  character(len=16) :: string
  ! switch to do the same calculation without the neighbor lists.
  ! can be used as a test for the neighbor lists (but takes a lot of time)
  logical, parameter :: l_no_lists = .FALSE.
  if(l_debug_min_distance) then
    write(unit=*, fmt='(a)') "DEBUG: Entering subroutine min_distance"
  end if
  if(n_mon_tot .eq. 0) then
    write(unit=*, fmt='(a)') "MESSAGE (SR min_distance): There's no fluid, &
       &so nothing to be done here."
    return
  end if
  ! initialize
  minimum   = 1.0E100_dp
  average   = 0.0_dp
  counter   = 0
  min_index_i = -1
  min_index_j = -1
  ! loop over fluid particles
  do i_part = 1, n_mon_tot
    if(l_fluid_fluid_interaction) then
      ! neighbors in the fluid
      counter = counter + ff_list(i_part, 0)
      do j_part = 1, ff_list(i_part, 0)
        r_dummy = sqrt(distance_squared(i_part, ff_list(i_part, j_part)))
        average = average + r_dummy
        if(r_dummy.lt.minimum) then
          minimum = r_dummy
          min_index_i = i_part
          min_index_j = ff_list(i_part, j_part)
        end if
      end do
    end if
    if(l_fluid_wall_interaction) then
      ! neighbors in the wall
      counter = counter + fw_list(i_part, 0)
      do j_part = 1, fw_list(i_part, 0)
        r_dummy = sqrt(distance_squared(i_part, fw_list(i_part, j_part)))
        average = average + r_dummy
        if(r_dummy.lt.minimum) then
          minimum = r_dummy
          min_index_i = i_part
          min_index_j = fw_list(i_part, j_part)
        end if
      end do
    end if
  end do
  if(counter .gt. 0) then
    average = average / real(counter, kind=dp)
    write(unit=*, fmt='(a, g11.4, a, g11.4)') &
         "MESSAGE (SR min_distance): Distances in neighbor lists: min =", &
         minimum, ", avg =", average
    if(min_index_i .le. 0) then
      string = "nowhere"
    else if((0 .lt. min_index_i) &
         .and. (min_index_i .le. n_mon_tot) ) then
      string = "fluid"
    else if((n_mon_tot .lt. min_index_i) &
         .and. (min_index_i .le. n_mon_tot + n_top_wall)) then
      string = "top wall"
    else if((n_mon_tot + n_top_wall .lt. min_index_i) &
         .and. (min_index_i .le. n_part)) then
      string = "bottom wall"
    else
      string = "forbidden range"
    end if
    write(unit=*, fmt='(a, i11, 3a)') &
         "         (part. index i", min_index_i, " in ", string, ")"
    if(min_index_j .le. 0) then
      string = "nowhere"
    else if((0 .lt. min_index_j) &
         .and. (min_index_j .le. n_mon_tot) ) then
      string = "fluid"
    else if((n_mon_tot .lt. min_index_j) &
         .and. (min_index_j .le. n_mon_tot + n_top_wall)) then
      string = "top wall"
    else if((n_mon_tot + n_top_wall .lt. min_index_j) &
```

```
          .and. (min_index_j .le. n_part)) then
        string = "bottom wall"
      else
        string = "forbidden range"
      end if
      write(unit=*, fmt='(a, i11, 3a)') &
           "         (part. index j", min_index_j, " in ", string, ")"
    else
      write(unit=*, fmt='(a)') "CAUTION (SR min_distance): &
           &Could not compute minimal distance from fl-fl and fl-wa &
           &neighbor lists."
    end if
    if(l_no_lists) then
      ! do a check without using neighbor lists
      ! initialize
      minimum   = 1.0E100_dp
      min_index_i = -1
      min_index_j = -1
      do i_part = 1, n_mon_tot
        do j_part = 1, n_part
          if(i_part .ne. j_part) then
            r_dummy = sqrt(distance_squared(i_part, j_part))
            average = average + r_dummy
            if(r_dummy.lt.minimum) then
              minimum = r_dummy
              min_index_i = i_part
              min_index_j = j_part
            end if
          end if
        end do
      end do
      ! it makes no sense to compute an average, as non interacting particles
      ! can be very far away
      write(unit=*, fmt='(a, g12.6)') &
           "         without neighbor lists: min. dist. =", minimum
      if(min_index_i .le. 0) then
        string = "nowhere"
      else if((0 .lt. min_index_i) &
           .and. (min_index_i .le. n_mon_tot) ) then
        string = "fluid"
      else if((n_mon_tot .lt. min_index_i) &
           .and. (min_index_i .le. n_mon_tot + n_top_wall)) then
        string = "top wall"
      else if((n_mon_tot + n_top_wall .lt. min_index_i) &
           .and. (min_index_i .le. n_part)) then
        string = "bottom wall"
      else
        string = "forbidden range"
      end if
      write(unit=*, fmt='(a, i11, 3a)') &
           "         (part. index i", min_index_i, " in ", string, ")"
      if(min_index_j .le. 0) then
        string = "nowhere"
      else if((0 .lt. min_index_j) &
           .and. (min_index_j .le. n_mon_tot) ) then
        string = "fluid"
      else if((n_mon_tot .lt. min_index_j) &
           .and. (min_index_j .le. n_mon_tot + n_top_wall)) then
        string = "top wall"
      else if((n_mon_tot + n_top_wall .lt. min_index_j) &
           .and. (min_index_j .le. n_part)) then
        string = "bottom wall"
      else
        string = "forbidden range"
      end if
      write(unit=*, fmt='(a, i11, 3a)') &
           "         (part. index j", min_index_j, " in ", string, ")"
    end if !(l_no_lists)

    if(l_debug_min_distance) then
      print *, "DEBUG: Leaving subroutine min_distance"
    end if
  end subroutine min_distance
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! computes the maximal distance a particle has travelled, writes out the
  ! distance, the particle index, and the kind of particle
  subroutine max_distance
    real(kind=dp), dimension(n_dim) :: delta_r_vec
    real(kind=dp):: distance, maximum, average
    integer :: i_part, max_index
    character(len=16) :: string
    if(l_debug_max_distance) then
       print *, "DEBUG: Entering subroutine max_distance"
    end if
    ! initialize
    distance  = 0.0_dp
    maximum   = 0.0_dp
    average   = 0.0_dp
    max_index = -1
    ! loop only over particles which are moving.
    do i_part = 1, n_moving
       ! calculate how far the particle has moved from the last step
      delta_r_vec(1:n_dim_pbc) = &
           r0_old(i_part, 1:n_dim_pbc) - r0(i_part, 1:n_dim_pbc) &
           + real(pbc_count_old(i_part, 1:n_dim_pbc) &
           - pbc_count(i_part, 1:n_dim_pbc),kind=dp) &
           * boundary(1:n_dim_pbc)
      delta_r_vec(n_dim_pbc+1:n_dim) &
           = r0_old(i_part, n_dim_pbc+1:n_dim) - r0(i_part, n_dim_pbc+1:n_dim)
      distance = sqrt(dot_product(delta_r_vec, delta_r_vec))
      average = average + distance
      if(distance.gt.maximum) then
         maximum = distance
         max_index = i_part
      end if
    end do
    average = average / real(n_moving, kind=dp)
    if(max_index .le. 0) then
       string = "nowhere"
    else if((0 .lt. max_index) &
         .and. (max_index .le. n_mon_tot) ) then
       string = "fluid"
    else if((n_mon_tot .lt. max_index) &
         .and. (max_index .le. n_mon_tot + n_top_wall)) then
       string = "top wall"
    else if((n_mon_tot + n_top_wall .lt. max_index) &
         .and. (max_index .le. n_part)) then
       string = "bottom wall"
    else
       string = "forbidden range"
    end if
    print *, "MESSAGE: In subroutine max_distance:"
    print *, "         max. dist. =", maximum, ", average= ", average
    print *, "         (part. index ", max_index, " in ", string, ")"
    if(l_debug_max_distance) then
       print *, "DEBUG: Leaving subroutine max_distance"
    end if
  end subroutine max_distance
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! computes the maximal force a particle experiences, writes out the
  ! force, the particle index, and the kind of particle
  subroutine max_force
    real(kind=dp):: modulus_force, maximum, average
    integer :: i_part, max_index
    character(len=16) :: string
    logical, parameter :: l_debug_max_force = .FALSE.
    if(l_debug_max_force) then
       print *, "DEBUG: Entering subroutine max_force"
    end if
    ! initialize
    modulus_force = 0.0_dp
    maximum       = 0.0_dp
    average       = 0.0_dp
    max_index     = -1
    ! loop only over particles which are moving.
    do i_part = 1, n_moving
       modulus_force = sqrt(dot_product(force_determ(i_part, :), &
       force_determ(i_part, :)))
       average = average + modulus_force
       if(modulus_force.gt.maximum) then
          maximum = modulus_force
          max_index = i_part
       end if
    end do
    average = average / n_moving
    if(max_index .le. 0) then
       string = "nowhere"
    else if((0 .lt. max_index) &
         .and. (max_index .le. n_mon_tot) ) then
       string = "fluid"
    else if((n_mon_tot .lt. max_index) &
         .and. (max_index .le. n_mon_tot + n_top_wall)) then
       string = "top wall"
    else if((n_mon_tot + n_top_wall .lt. max_index) &
         .and. (max_index .le. n_part)) then
       string = "bottom wall"
    else
       string = "forbidden range"
    end if
    print *, "MESSAGE: In subroutine max_force:"
    print *, "         max. force =", maximum, ", average= ", average
    print *, "         (part. index ", max_index, " in ", string, ")"
    if(l_debug_max_force) then
       print *, "DEBUG: Leaving subroutine max_force"
    end if
  end subroutine max_force
  !---------------------------------------------------------------------------
end module mdroutines
```

interactionV1.9.f90

```
! module containing the subroutines for interactions
! Martin Aichele, 2000-11-27
! last changed: 2003-02-17
module interaction
  ! module containing global variables
  use globals
  use utilities
  implicit none
  !---------------------------------------------------------------------------
  ! debug switches      |-------- 31 characters ------|
  !---------------------------------------------------------------------------
  logical, parameter :: l_debug_calc_distance           = .FALSE.
  logical, parameter :: l_debug_fluid_fluid             = .FALSE.
  logical, parameter :: l_debug_fluid_wall              = .FALSE.
  logical, parameter :: l_debug_wall_wall               = .FALSE.
  logical, parameter :: l_debug_intra_molec             = .FALSE.
  logical, parameter :: l_debug_intra_wall              = .FALSE.
  logical, parameter :: l_debug_create_fk_neigh_list    = .FALSE.
  logical, parameter :: l_debug_create_fk_neigh_list_1d = .FALSE.
  !---------------------------------------------------------------------------
contains
  !---------------------------------------------------------------------------
  ! Calculate the vector delta_r from particle j_part to particle i_part and
  ! r_2 = |delta_r|^2
  ! This subroutine is called very often, so speed is important here.
  subroutine calc_distance(i_part, j_part, delta_r, r_2)
    integer, intent(in) :: i_part, j_part
    real(kind=dp), intent(out) :: delta_r(n_dim)
    real(kind=dp), intent(out), optional :: r_2
    integer :: i_dim
    if(l_debug_calc_distance) then
      if( (i_part .gt. n_part) .or. (j_part.gt.n_part)) then
        print *, "DEBUG: Entering subroutine calc_distance with indices ", &
             i_part, j_part, " > n_part=", n_part
        stop
      end if
    end if
    do i_dim = 1, n_dim
      delta_r(i_dim) = r0(i_part, i_dim) - r0(j_part, i_dim)
    end do
    ! minimum image convention in xy plane only
    ! here one can save a lot of cpu time if the right method is chosen
    if(l_mic_use_int_cast) then
      do i_dim = 1, n_dim_pbc
        delta_r(i_dim) = delta_r(i_dim) - boundary(i_dim) &
             * aint(delta_r(i_dim)/half_bound(i_dim))
      end do
    else ! faster on Pentiums and Alphas
      do i_dim = 1, n_dim_pbc
        if(delta_r(i_dim) > half_bound(i_dim)) then
          delta_r(i_dim) = delta_r(i_dim) - boundary(i_dim)
        else if(delta_r(i_dim) < -half_bound(i_dim)) then
          delta_r(i_dim) = delta_r(i_dim) + boundary(i_dim)
        end if
      end do
    end if
    ! calculate squared distance only if needed
    if(present(r_2)) r_2 = dot_product(delta_r(:), delta_r(:))
  end subroutine calc_distance
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
```

```
! the interaction between two fluid particles
subroutine fluid_fluid
  real(kind=dp) :: r_dummy, r_2, r_6, r_12, pot_loc
  real(kind=dp), dimension(n_dim) :: delta_r, force_loc
  integer :: i_dim
  ! loop variables
  integer :: i_neighbor, i_part, j_part
  ! loops over particle types
  integer :: i_type, j_type
  if(l_debug_fluid_fluid) then
    print *, "DEBUG: Entering subroutine fluid_fluid"
  end if
  ! initialize
  v_fluid_fluid = 0.0_dp
  do i_part = 1, n_mon_tot ! loop over all particles in the fluid
    i_type = type(i_part)
    do i_neighbor = 1, ff_list(i_part,0) ! loop over all neighbors of i_part
      j_part = ff_list(i_part,i_neighbor)
      j_type = type(j_part)
      call calc_distance(i_part, j_part, delta_r, r_2)
      if(l_debug_fluid_fluid) then
        write(unit=*, fmt='(2(a, i8), 2(a, g13.6))') &
             "fl-fl interaction: i=", i_part, ", j=", j_part, &
             ", r_2=", r_2, ", range_2=", range_2(i_type, j_type)
      end if
      ! check whether interaction takes place.  This is necessary because
      ! the lists include particles in the skin which don't interact
      if(r_2.lt.range_2(i_type, j_type)) then
        ! restrict the distance to an effective interaction distance.
        ! An if switch is faster than the comparison of two doubles.
        ! If we start with a configuration read from a file, we could
        ! set this variable as PARAMETER = .FALSE. to speed up the code
        if(l_r_2_min_finite) then
          r_2 = max(r_2, r_2_min)
        end if
        r_6 = (sigma_2(i_type, j_type)/r_2)**3
        r_12 = r_6 * r_6
        pot_loc = (r_12 - r_6) - e_shift(i_type, j_type)
        v_fluid_fluid = v_fluid_fluid + four_epsil(i_type,j_type) * pot_loc
        r_dummy = four_epsil(i_type, j_type)*(-12.0_dp*r_12+6.0_dp*r_6) / r_2
        do i_dim = 1, n_dim
          force_loc(i_dim) = r_dummy*delta_r(i_dim)
          force_determ(i_part, i_dim) = force_determ(i_part, i_dim) &
               - force_loc(i_dim)
          force_determ(j_part, i_dim) = force_determ(j_part, i_dim) &
               + force_loc(i_dim)
        end do
        if(l_compute_press_tens) &
             call add_to_press_tensor(force_loc, delta_r, press_tens_pot)
        if(l_debug_fluid_fluid) then
          print *, "force_loc=", force_loc(:)
          print *, "force(i)=", force_determ(i_part, :)
          print *, "force(j)=", force_determ(j_part, :)
        end if
      end if
    end do
  end do
  if(l_debug_fluid_fluid) then
    print *, "DEBUG: Leaving subroutine fluid_fluid"
  end if
end subroutine fluid_fluid
!------------------------------------------------------------------------------
!------------------------------------------------------------------------------
! the interaction between a fluid and a wall particle
subroutine fluid_wall
  real(kind=dp) :: r_dummy, r_2, r_6, r_12, pot_loc
  real(kind=dp), dimension(n_dim) :: delta_r, force_loc
  integer :: i_dim
  ! loop variables
  integer :: i_neighbor, i_part, j_part
  ! loops over particle types
  integer :: i_type, j_type
  if(l_debug_fluid_wall) then
    print *, "DEBUG: Entering subroutine fluid_wall"
  end if
  v_fluid_wall = 0.0_dp
  j_type = n_type
  do i_part = 1, n_mon_tot
    i_type = type(i_part)
    do i_neighbor = 1, fw_list(i_part, 0)
      j_part = fw_list(i_part, i_neighbor)
      call calc_distance(i_part, j_part, delta_r, r_2)
      ! check whether interaction takes place
      if(r_2.lt.range_2(i_type,j_type)) then
        if(l_r_2_min_finite) then
          r_2 = max(r_2, r_2_min)
        end if
        r_6 = (sigma_2(i_type,j_type)/r_2)**3
        r_12 = r_6**2
        pot_loc = (r_12 - r_6) - e_shift(i_type,j_type)
        if(l_bottom_wall_fluid_ia_differ .and. &
             (j_part .gt. n_mon_tot + n_top_wall)) then
          ! fluid -- bottom wall interaction is modified
          v_fluid_wall = v_fluid_wall &
               + four_epsil(i_type,j_type) * bw_fl_ia_adjust * pot_loc
          r_dummy = four_epsil(i_type,j_type) * bw_fl_ia_adjust &
               * (-12.0_dp*r_12+6.0_dp*r_6)/r_2
        else
          ! add up potential
          v_fluid_wall = v_fluid_wall + four_epsil(i_type,j_type)*pot_loc
          ! force_loc is force acting on wall atoms
          r_dummy = four_epsil(i_type,j_type)*(-12.0_dp*r_12+6.0_dp*r_6)/r_2
        end if
        do i_dim = 1,n_dim
          force_loc(i_dim) = r_dummy*delta_r(i_dim)
          force_determ(i_part,i_dim) = force_determ(i_part,i_dim) &
               - force_loc(i_dim)
          force_determ(j_part,i_dim) = force_determ(j_part,i_dim) &
               + force_loc(i_dim)
        end do
        if(l_compute_press_tens) &
             call add_to_press_tensor(force_loc, delta_r, press_tens_pot)
      end if
    end do
  end do
  if(l_debug_fluid_wall) then
    print *, "DEBUG: Leaving subroutine fluid_wall"
  end if
end subroutine fluid_wall
!------------------------------------------------------------------------------
!------------------------------------------------------------------------------
! the interaction of wall particles
subroutine wall_wall
  real(kind=dp) :: r_dummy, r_2, r_6, r_12, pot_loc
  real(kind=dp), dimension(n_dim) :: delta_r, force_loc
  integer :: i_dim
  ! loop variables
  integer :: i_neighbor, i_wall, i_part, j_part
  ! particle types (two wall particles)
  integer, parameter :: i_type = n_type, j_type = n_type
  if(l_debug_wall_wall) then
    print *, "DEBUG: Entering subroutine wall_wall"
  end if
  v_wall_wall = 0.0_dp
  do i_wall = 1, n_top_wall
    i_part = i_wall + n_mon_tot
    do i_neighbor = 1,ww_list(i_wall,0)
      j_part = ww_list(i_wall,i_neighbor)
      call calc_distance(i_part, j_part, delta_r, r_2)
      ! check whether interaction takes place
      if(r_2.lt.range_2(i_type,j_type)) then
        if(l_r_2_min_finite) then
          r_2 = max(r_2, r_2_min)
        end if
        r_6 = (sigma_2(i_type,j_type)/r_2)**3
        r_12 = r_6**2
        pot_loc = (r_12 - r_6) - e_shift(i_type,j_type)
        v_wall_wall = v_wall_wall + four_epsil(i_type,j_type)*pot_loc
        r_dummy = four_epsil(i_type,j_type)*(-12.0_dp*r_12+6.0_dp*r_6)/r_2
        do i_dim = 1,n_dim
          force_loc(i_dim) = r_dummy*delta_r(i_dim)
          force_determ(i_part,i_dim) = force_determ(i_part,i_dim) &
               - force_loc(i_dim)
          force_determ(j_part,i_dim) = force_determ(j_part,i_dim) &
               + force_loc(i_dim)
        end do
        if(l_compute_press_tens_wall_contr) then
          call add_to_press_tensor(force_loc,delta_r,press_tens_pot_walls)
        end if
      end if
    end do
  end do
  if(l_debug_wall_wall) then
    print *, "DEBUG: Leaving subroutine wall_wall"
  end if
end subroutine wall_wall
!------------------------------------------------------------------------------
!------------------------------------------------------------------------------
! the interaction in the polymer
subroutine intra_molec
  ! the maximal value of |r_i - r_j|^2 / r_chain^2 (the FENE - potential
  ! diverges for |r_i - r_j|^2 / r_chain^2 = 1) we accept without issuing
  ! a warning.  In this case the potential is set to the value at
  ! max_bond_extend.
  ! This is done because high forces can cause the algorithm to become
  ! unstable.
  real(kind=dp), parameter :: max_bond_extend = 0.98_dp
  integer :: i_dim
  real(kind=dp) :: r_dummy, r_2
  real(kind=dp), dimension(n_dim) :: delta_r, force_loc
  ! loops over particle types
  integer :: i_type, j_type
  integer :: i_mon, i_chain ! loop variables
  integer :: i_part, j_part ! loop variables
  if(l_debug_intra_molec) then
    print *, "DEBUG: Entering subroutine intra_molec"
  end if
  ! reset potential
  v_intra_molec = 0.0_dp
  do i_chain = 1, n_chain ! loop over all chains
    do i_mon = 1, n_mon-1 ! loop over all monomer pairs in this polymer
      i_part = (i_chain-1) * n_mon + i_mon ! particle index
      i_type = type(i_part)
      j_part = i_part+1 ! the interacting partner is the neighbor
      j_type = type(j_part)
      call calc_distance(i_part, j_part, delta_r, r_2)
      ! bonded pairs always interact, so we don't test the distance
      ! we don't want to stretch the polymer bond too far, as very strong
      ! forces would result, crashing the integration algorithm.  This
      ! cutting off of the FENE potential happens only if strong external
      ! forces are applied or we create a new configuration.
      r_dummy = r_2/(r_chain_2 * sigma_2(i_type, j_type))
      if(r_dummy .ge. 1.0_dp) then
        ! the FENE-potential diverges at r_chain_2 * sigma_2(i_type, j_type).
        ! If the bond is stretched over this length, we try to repair the
        ! bond or stop
        write(unit=*, fmt='(a, i1, a, i2, a, g13.6, a)') &
             "WARNING (SR intra_molec): r_2/(r_chain_2*sigma_2(", i_type, &
             ",", j_type, ")=", r_dummy, ".ge. 1.0"
        write(unit=*, fmt='(a, g13.6, a, f7.4, a, f7.4, a)') &
             "        distance=", sqrt(r_2), ", bond-length=", &
             bond_length(i_type, j_type), " (FENE pot. diverges at", &
             sqrt(r_chain_2 * sigma_2(i_type, j_type)), ")"
        write(unit=*, fmt='(2(a, i7), a, i12)') &
             "                for i_part=", i_part, ", j_part=" , j_part, &
             ", at MDS", i_time
        ! repair bonds in the relaxation phase
        if(l_r_2_min_finite) then
          call repair_bond(i_part, j_part)
          r_dummy = max_bond_extend
          if(l_debug_intra_molec) then
            call calc_distance(i_part, j_part, delta_r, r_2)
            r_dummy = r_2/(r_chain_2 * sigma_2(i_type, j_type))
            if(abs(r_dummy - max_bond_extend) .gt. 10.0_dp**(-10)) then
              print *, "DEBUG: r_dummy =", r_dummy, &
                   "should be max_bond_extend"
              stop
            end if
          end if
        else
          write(unit=*, fmt='(a)') &
               "ERROR (SR intra_molec):  Not repairing bond.  Giving up."
          call particle_positions_out("bond_overstretch_", i_time)
          stop
        end if
      else if(r_dummy .gt. max_bond_extend) then
        write(unit=*, fmt='(a, i1, a, i2, a, g13.6, a, f6.4)') &
             "WARNING (SR intra_molec): r_2/(r_chain_2*sigma_2(", i_type, &
             ",", j_type, ")=", r_dummy, ".gt. max_bond_extend=", &
             max_bond_extend
        write(unit=*, fmt='(a, g13.6, a, f6.4)') &
             "        distance sqrt(r_2)=", sqrt(r_2), ", bond-length=", &
             bond_length(i_type, j_type)
        write(unit=*, fmt='(2(a, i7), a, i12)') &
             "                for i_part=", i_part, ", j_part=" , j_part, &
             ", at MDS", i_time
        write(unit=*, fmt='(a)') &
             "                setting r_dummy = max_bond_extend and continue..."
        r_dummy = max_bond_extend
      end if ! (r_dummy .gt. 1.0_dp)
      ! add up potential
      v_intra_molec = v_intra_molec &
           + epsil(i_type, j_type) * log(1.0_dp - r_dummy)
      r_dummy = &
           epsil_k_chain_over_sigma_2(i_type, j_type) / (1.0_dp - r_dummy)
      force_loc(:) = r_dummy * delta_r(:)
      force_determ(i_part, :) = force_determ(i_part, :) - force_loc(:)
```

```
      force_determ(j_part, :) = force_determ(j_part, :) + force_loc(:)
      if(l_compute_press_tens) &
           call add_to_press_tensor(force_loc, delta_r, press_tens_pot)
    end do ! end loop over all monomers in a chain
  end do ! end loop over all chains
  ! multiply potential with common prefactor
  v_intra_molec = -0.5_dp * k_chain * r_chain_2 * v_intra_molec
  if(l_debug_intra_molec) then
    print *, "DEBUG: Leaving subroutine intra_molec"
  end if
contains
  subroutine repair_bond(i_part, j_part)
    integer, intent(in) :: i_part, j_part
    integer :: i_dim
    real(kind=dp), dimension(n_dim) :: delta_r, pbc_shift
    real(kind=dp) :: prefactor
    write(unit=*, fmt='(a, i7, a)') &
         "WARNING (SR repair_bond): Moving particle", j_part, &
         " to reduce bond-length to max_bond_extend"
    ! initialize
    pbc_shift(:) = 0.0_dp
    delta_r(:) = r0(i_part, :) - r0(j_part, :)
    do i_dim = 1, n_dim_pbc
      if(delta_r(i_dim) > half_bound(i_dim)) then
        delta_r(i_dim) = delta_r(i_dim) - boundary(i_dim)
        pbc_shift(i_dim) = boundary(i_dim)
      else if(delta_r(i_dim) < -half_bound(i_dim)) then
        delta_r(i_dim) = delta_r(i_dim) + boundary(i_dim)
        pbc_shift(i_dim) = -boundary(i_dim)
      end if
    end do
    ! rescale delta_r so that the bond length corresponds to max_bond_extent
    !( note: r_dummy = r_2/(r_chain_2 * sigma_2(i_type, j_type)) )
    prefactor = sqrt(max_bond_extend * r_chain_2 * sigma_2(i_type, j_type) &
         / r_2)
    r0(j_part, :) = r0(i_part, :) - prefactor * delta_r(:) + pbc_shift(:)
  end subroutine repair_bond
end subroutine intra_molec
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! This subroutine deals with the forces on the wall particles and walls:
! 1) wall-particles (attached to the wall with springs) wall interaction
!    (Tomlinson wall model)
! 2) interaction between nearest neighbors (attached to each other with
!    springs, Frenklin-Kontorova model)
! 3) the external forces on the top wall
! Loops over top and bottom wall particles are separate in case different
! walls are introduced.
subroutine intra_wall
  real(kind=dp)                       :: r_dummy, r_2
  real(kind=dp), dimension(n_dim) :: delta_r, force_loc
  integer :: i_dim
  integer :: i_part, j_part, i_wall, i_neigh
  if(l_debug_intra_wall) then
    print *, "DEBUG: Entering subroutine intra_wall"
  end if
  ! initialize sum of contributions of Tomlinson springs to potential energy
  v_wall_harm_tom = 0.0_dp
  ! initialize sum of contributions of Frenkel-Kontorova springs to
  ! potential energy
  v_wall_harm_fre_kon_linear = 0.0_dp
  v_wall_harm_fre_kon_vector = 0.0_dp
  if(f_wall_fix.eq.0) then ! wall particles not fixed
    ! Tomlinson spring interaction
    if(k_spring_wall_tom .ne. 0.0_dp) then
      ! top wall
      do i_part = n_mon_tot+1, n_mon_tot + n_top_wall
        delta_r(:) = r0(i_part, :) - r_wall_equi(i_part-n_mon_tot, :)
        force_loc(:) = k_spring_wall_tom * delta_r(:)
        ! spring connecting the wall and the wall particle excerts
        ! an opposing force on the particle -------------|
        !                                                V
        force_determ(i_part, :) = force_determ(i_part, :) - force_loc(:)
        total_force_twall(:) = total_force_twall(:) + force_loc(:)
        v_wall_harm_tom = v_wall_harm_tom &
             + dot_product(delta_r(:), delta_r(:))
        if(l_compute_press_tens_wall_contr) then
          call add_to_press_tensor(force_loc,delta_r,press_tens_pot_walls)
        end if
      end do
      ! bottom wall
      do i_part = n_mon_tot + n_top_wall +1, n_mon_tot + n_wall
        delta_r(:) = r0(i_part, :) - r_wall_equi(i_part-n_mon_tot, :)
        force_loc(:) = k_spring_wall_tom * delta_r(:)
        ! spring connecting the wall and the wall particle excerts
        ! an opposing force on the particle -------------|
        !                                                V
        force_determ(i_part, :) = force_determ(i_part, :) - force_loc(:)
        v_wall_harm_tom = v_wall_harm_tom &
             + dot_product(delta_r(:), delta_r(:))
        if(l_compute_press_tens_wall_contr) then
          call add_to_press_tensor(force_loc,delta_r,press_tens_pot_walls)
        end if
      end do
    end if !(k_spring_wall_tom .ne. 0.0_dp)
    ! Frenkel-Kontorova spring interactions
    ! first the "linear" one, i.e. the force is exerted along a spring
    ! connecting the neighbors
    ! second the "vector" one, i.e. a force is exerted proportional to the
    ! difference vector of the neighbors.
    if(k_spring_wall_fre_kon_linear .ne. 0.0_dp) then
      ! loop over top wall particles
      do i_wall = 1, n_top_wall
        i_part = i_wall + n_mon_tot
        ! loop over all nearest neighbors
        do i_neigh = 1, n_fk_neighbors
          j_part = fk_neighbors(i_neigh, i_wall)
          ! sum up each pair only once
          if(j_part .gt. i_part) then
            ! periodic boundary contitions are taken into account by
            ! SR calc_distance
            call calc_distance(i_part, j_part, delta_r, r_2)
            r_dummy = sqrt(r_2)
            force_loc(:) = k_spring_wall_fre_kon_linear &
                 * (fk_neigh_eq_dist_tw(i_neigh)/r_dummy-1.0_dp) * delta_r(:)
            force_determ(i_part, :) = force_determ(i_part, :) + force_loc(:)
            force_determ(j_part, :) = force_determ(j_part, :) - force_loc(:)
            v_wall_harm_fre_kon_linear = v_wall_harm_fre_kon_linear &
                 + (r_dummy - fk_neigh_eq_dist_tw(i_neigh))**2
            if(l_compute_press_tens_wall_contr) then
              call add_to_press_tensor(-force_loc, &
                   delta_r, press_tens_pot_walls)
            end if
            ! if there is also vector FK interaction, we can compute it
            ! almost for free (no call to SR calc_distance necessary)
            if(k_spring_wall_fre_kon_vector .ne. 0.0_dp) then
              ! now delta_r gets meaning of the difference between
              ! equilibrium and actual vector between neighbors
              delta_r(:) = fk_neigh_vec_tw(:, i_neigh) - delta_r(:)
              force_loc(:) = k_spring_wall_fre_kon_vector * delta_r(:)
              force_determ(i_part, :) = force_determ(i_part,:) + force_loc(:)
              force_determ(j_part, :) = force_determ(j_part,:) - force_loc(:)
              v_wall_harm_fre_kon_vector = v_wall_harm_fre_kon_vector &
                   + dot_product(delta_r(:), delta_r(:))
              if(l_compute_press_tens_wall_contr) then
                call add_to_press_tensor(-force_loc, delta_r, &
                     press_tens_pot_walls)
              end if
            end if !(k_spring_wall_fre_kon_vector .ne. 0.0_dp)
          end if ! (j_part .gt. i_part)
        end do ! loop over neighbors
      end do ! loop over top wall particles
      ! loop over bottom wall particles
      do i_wall = 1+ n_top_wall, n_bottom_wall + n_top_wall
        i_part = i_wall + n_mon_tot
        ! loop over all nearest neighbors
        do i_neigh = 1, n_fk_neighbors
          j_part = fk_neighbors(i_neigh, i_wall)
          ! sum up each pair only once
          if(j_part .gt. i_part) then
            ! periodic boundary contitions are taken into account by
            ! SR calc_distance
            call calc_distance(i_part, j_part, delta_r, r_2)
            r_dummy = sqrt(r_2)
            force_loc(:) = k_spring_wall_fre_kon_linear &
                 * (fk_neigh_eq_dist_bw(i_neigh)/r_dummy-1.0_dp) * delta_r(:)
            force_determ(i_part, :) = force_determ(i_part, :) + force_loc(:)
            force_determ(j_part, :) = force_determ(j_part, :) - force_loc(:)
            v_wall_harm_fre_kon_linear = v_wall_harm_fre_kon_linear &
                 + (r_dummy - fk_neigh_eq_dist_bw(i_neigh))**2
            if(l_compute_press_tens_wall_contr) then
              call add_to_press_tensor(-force_loc, &
                   delta_r, press_tens_pot_walls)
            end if
            ! if there is also vector FK interaction, we can compute it
            ! almost for free (no call to SR calc_distance necessary)
            if(k_spring_wall_fre_kon_vector .ne. 0.0_dp) then
              ! now delta_r gets meaning of the difference between
              ! equilibrium and actual vector between neighbors
              delta_r(:) = fk_neigh_vec_bw(:, i_neigh) - delta_r(:)
              force_loc(:) = k_spring_wall_fre_kon_vector * delta_r(:)
              force_determ(i_part, :) = force_determ(i_part,:) + force_loc(:)
              force_determ(j_part, :) = force_determ(j_part,:) - force_loc(:)
              v_wall_harm_fre_kon_vector = v_wall_harm_fre_kon_vector &
                   + dot_product(delta_r(:), delta_r(:))
              if(l_compute_press_tens_wall_contr) then
                call add_to_press_tensor(-force_loc, &
                     delta_r, press_tens_pot_walls)
                end if
              end if !(k_spring_wall_fre_kon_vector .ne. 0.0_dp)
            end if ! (j_part .gt. i_part)
          end do ! loop over neighbors
        end do ! loop over bottom wall particles
      else if(k_spring_wall_fre_kon_vector .ne. 0.0_dp) then
        ! no linear FK interaction  where vector FK-interaction is included
        ! but pure vector interaction
        ! loop over top wall particles
        do i_wall = 1, n_top_wall
          i_part = i_wall + n_mon_tot
          ! loop over all nearest neighbors
          do i_neigh = 1, n_fk_neighbors
            j_part = fk_neighbors(i_neigh, i_wall)
            ! sum up each pair only once
            if(j_part .gt. i_part) then
              ! compute only the difference of the positions, not the distance
              call calc_distance(i_part, j_part, delta_r)
              ! now delta_r gets meaning of the difference between equilibrium
              ! and actual vector between neighbors
              delta_r(:) = fk_neigh_vec_tw(:, i_neigh) - delta_r(:)
              force_loc(:) = k_spring_wall_fre_kon_vector * delta_r(:)
              force_determ(i_part, :) = force_determ(i_part, :) + force_loc(:)
              force_determ(j_part, :) = force_determ(j_part, :) - force_loc(:)
              v_wall_harm_fre_kon_vector = v_wall_harm_fre_kon_vector &
                   + dot_product(delta_r(:), delta_r(:))
              if(l_compute_press_tens_wall_contr) then
                call add_to_press_tensor(-force_loc, &
                     delta_r, press_tens_pot_walls)
              end if
            end if !(j_part .gt. i_part)
          end do ! loop over neighbors
        end do ! loop over top wall par1ticles
        ! loop over bottom  wall particles
        do i_wall = 1 + n_top_wall, n_bottom_wall + n_top_wall
          i_part = i_wall + n_mon_tot
          ! loop over all nearest neighbors
          do i_neigh = 1, n_fk_neighbors
            j_part = fk_neighbors(i_neigh, i_wall)
            ! sum up each pair only once
            if(j_part .gt. i_part) then
              ! compute only the difference of the positions, not the distance
              call calc_distance(i_part, j_part, delta_r)
              ! now delta_r gets meaning of the difference between equilibrium
              ! and actual vector between neighbors
              delta_r(:) = fk_neigh_vec_bw(:, i_neigh) - delta_r(:)
              force_loc(:) = k_spring_wall_fre_kon_vector * delta_r(:)
              force_determ(i_part, :) = force_determ(i_part, :) + force_loc(:)
              force_determ(j_part, :) = force_determ(j_part, :) - force_loc(:)
              v_wall_harm_fre_kon_vector = v_wall_harm_fre_kon_vector &
                   + dot_product(delta_r(:), delta_r(:))
              if(l_compute_press_tens_wall_contr) then
                call add_to_press_tensor(-force_loc, &
                     delta_r, press_tens_pot_walls)
              end if
            end if !(j_part .gt. i_part)
          end do ! loop over neighbors
        end do ! loop over bottom wall particles
    end if !(k_spring_wall_fre_kon_linear .ne. 0.0_dp)
  else ! f_wall_fix = 1
    ! add up forces acting on top wall particles to the total force on the
    ! top wall, as springs are totally stiff
    do i_part = n_mon_tot+1, n_mon_tot + n_top_wall
      total_force_twall(:) = total_force_twall(:) + force_determ(i_part, :)
    end do
  end if !(f_wall.eq.0)
  ! get correct prefactors of the potential energies
  if(k_spring_wall_tom .ne. 0.0_dp) &
       v_wall_harm_tom = &
       (k_spring_wall_tom/2.0_dp) * v_wall_harm_tom
  if(k_spring_wall_fre_kon_linear .ne. 0.0_dp) &
       v_wall_harm_fre_kon_linear = &
       (k_spring_wall_fre_kon_linear/2.0_dp)*v_wall_harm_fre_kon_linear
  if(k_spring_wall_fre_kon_vector .ne. 0.0_dp) &
```

```
      v_wall_harm_fre_kon_vector = &
       (k_spring_wall_fre_kon_vector/2.0_dp)*v_wall_harm_fre_kon_vector
  ! coupling to external spring or force
  do i_dim = 1, n_dim
    select case(f_twall(i_dim))
    case(force_mode)
      ! add external force
      r_dummy = ext_force_twall(i_dim)
      ext_force_twall(i_dim) = ext_force_twall(i_dim) &
           + dt * s_force_grad(i_dim) * ramp_force_twall(i_dim)
      if(r_dummy * ext_force_twall(i_dim) .lt. 0.0_dp) then
        ! when sign of force has changed then the force ramp is changed to
        ! slower
        s_force_grad(i_dim) = s_force_grad(i_dim)/4.0_dp
      end if
      total_force_twall(i_dim) = total_force_twall(i_dim) &
           + real(n_top_wall, kind=dp) * ext_force_twall(i_dim)
    case(spring_mode)
      ! add force from external spring
      ext_force_twall(i_dim) = &
           k_spring_twall(i_dim)*(r0_spring_twall(i_dim)-r0_twall(i_dim))
      total_force_twall(i_dim) = total_force_twall(i_dim) &
           + real(n_top_wall, kind=dp) * ext_force_twall(i_dim)
    case default
      write(unit=*, fmt=*) &
           "ERROR (SR intra_wall): mode flag not recognized"
      stop
    end select
  end do
  if(l_debug_intra_wall) then
    print *, "DEBUG: Leaving subroutine intra_wall"
  end if
end subroutine intra_wall
!------------------------------------------------------------------------------
!------------------------------------------------------------------------------
! subroutines for the implementation of the Frenkel-Kontorova wall model
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! creates the neighbor list of wall particles for the Frenkel-Kontorova model
! for 1-dimensional walls
subroutine create_fk_neigh_list_1d
  ! the neighbors of a wall particle
  integer :: a_neigh, b_neigh
  ! loop over  wall indices
  integer :: i_wall
  if(l_debug_create_fk_neigh_list_1d) then
    print *, "DEBUG: Entering SR create_fk_neigh_list_1d"
  end if
  ! top wall
  !=========
  ! first particle has periodic neighbor n_top_wall.  Because the neighbor
  ! lists are so simple, we code everything explicitely here.
  i_wall  = 1
  a_neigh = n_top_wall
  b_neigh = 2
  ! the first index is a particle index, so add offset n_mon_tot
  fk_neighbors(1, i_wall) = a_neigh + n_mon_tot
  fk_neighbors(2, i_wall) = b_neigh + n_mon_tot
  ! middle of the wall
  do i_wall = 2, n_top_wall-1
    a_neigh = i_wall -1
    b_neigh = i_wall +1
    fk_neighbors(1, i_wall) = a_neigh + n_mon_tot
    fk_neighbors(2, i_wall) = b_neigh + n_mon_tot
  end do
  i_wall  = n_top_wall
  a_neigh = n_top_wall -1
  b_neigh = 1
  fk_neighbors(1, i_wall) = a_neigh + n_mon_tot
  fk_neighbors(2, i_wall) = b_neigh + n_mon_tot

  ! bottom wall
  !============
  i_wall  = n_top_wall + 1
  a_neigh = n_top_wall + n_bottom_wall
  b_neigh = n_top_wall + 2
  ! the first index is a particle index
  fk_neighbors(1, i_wall) = a_neigh + n_mon_tot
  fk_neighbors(2, i_wall) = b_neigh + n_mon_tot
  do i_wall = n_top_wall + 2, n_top_wall + n_bottom_wall-1
    a_neigh = i_wall -1
    b_neigh = i_wall +1
    fk_neighbors(1, i_wall) = a_neigh + n_mon_tot
    fk_neighbors(2, i_wall) = b_neigh + n_mon_tot
  end do
  i_wall  = n_top_wall + n_bottom_wall
  a_neigh = n_top_wall + n_bottom_wall-1
  b_neigh = n_top_wall + 1
  fk_neighbors(1, i_wall) = a_neigh + n_mon_tot
  fk_neighbors(2, i_wall) = b_neigh + n_mon_tot
  if(l_debug_create_fk_neigh_list_1d) then
    print *, "DEBUG: Leaving SR create_fk_neigh_list_1d"
  end if
end subroutine create_fk_neigh_list_1d
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! creates the neighbor list of wall particles for the Frenkel-Kontorova model
! for 2-dimensional walls
subroutine create_fk_neigh_list
  ! the neighbors around a wall particle
  integer :: a_neigh, b_neigh, c_neigh, d_neigh, e_neigh, f_neigh
  ! loop over  wall indices
  integer :: i_wall
  if(l_debug_create_fk_neigh_list) then
    print *, "DEBUG: Entering SR create_fk_neigh_list"
  end if
  ! top wall
  !---------
  ! first row of cells
  do i_wall = 1, 2*n_cell_w_y
    call assign_neighbors(n_cell_w_y, i_wall, &
         a_neigh, b_neigh, c_neigh, d_neigh, e_neigh, f_neigh)
    if(mod(i_wall, 2) .eq. 1) then
      a_neigh = a_neigh + 2*n_cell_w_x*n_cell_w_y
      e_neigh = e_neigh + 2*n_cell_w_x*n_cell_w_y
      f_neigh = f_neigh + 2*n_cell_w_x*n_cell_w_y
    else
      f_neigh = f_neigh + 2*n_cell_w_x*n_cell_w_y
    end if
    call apply_pbc(2*n_cell_w_y, i_wall, a_neigh, b_neigh, c_neigh, d_neigh,&
         e_neigh, f_neigh)
    call write_fk_neighbors(i_wall, n_mon_tot, a_neigh, b_neigh, c_neigh, &
         d_neigh, e_neigh, f_neigh)
  end do
  do i_wall = 2*n_cell_w_y + 1, 2*n_cell_w_x*n_cell_w_y - 2*n_cell_w_y
    call assign_neighbors(n_cell_w_y, i_wall, &
         a_neigh, b_neigh, c_neigh, d_neigh, e_neigh, f_neigh)
    call apply_pbc(2*n_cell_w_y, i_wall, a_neigh, b_neigh, c_neigh, d_neigh,&
         e_neigh, f_neigh)
    call write_fk_neighbors(i_wall, n_mon_tot, a_neigh, b_neigh, c_neigh, &
         d_neigh, e_neigh, f_neigh)
  end do
  ! last row of cells
  do i_wall = 2*n_cell_w_x*n_cell_w_y - 2*n_cell_w_y + 1, &
       2*n_cell_w_x*n_cell_w_y
    call assign_neighbors(n_cell_w_y, i_wall, &
         a_neigh, b_neigh, c_neigh, d_neigh, e_neigh, f_neigh)
    if(mod(i_wall, 2) .eq. 0) then
      b_neigh = b_neigh - 2*n_cell_w_x*n_cell_w_y
      c_neigh = c_neigh - 2*n_cell_w_x*n_cell_w_y
      d_neigh = d_neigh - 2*n_cell_w_x*n_cell_w_y
    else
      c_neigh = c_neigh - 2*n_cell_w_x*n_cell_w_y
    end if
    call apply_pbc(2*n_cell_w_y, i_wall, a_neigh, b_neigh, c_neigh, d_neigh,&
         e_neigh, f_neigh)
    call write_fk_neighbors(i_wall, n_mon_tot, a_neigh, b_neigh, c_neigh, &
         d_neigh, e_neigh, f_neigh)
  end do
  ! bottom wall
  !------------
  ! first row of cells
  do i_wall = n_top_wall + 1, n_top_wall + 2*n_cell_w_y
    call assign_neighbors(n_cell_w_y, i_wall, &
         a_neigh, b_neigh, c_neigh, d_neigh, e_neigh, f_neigh)
    if(mod(i_wall, 2) .eq. 1) then
      a_neigh = a_neigh + 2*n_cell_w_x*n_cell_w_y
      e_neigh = e_neigh + 2*n_cell_w_x*n_cell_w_y
      f_neigh = f_neigh + 2*n_cell_w_x*n_cell_w_y
    else
      f_neigh = f_neigh + 2*n_cell_w_x*n_cell_w_y
    end if
    call apply_pbc(2*n_cell_w_y, i_wall, a_neigh, b_neigh, c_neigh, d_neigh,&
         e_neigh, f_neigh)
    call write_fk_neighbors(i_wall, n_mon_tot, a_neigh, b_neigh, c_neigh, &
         d_neigh, e_neigh, f_neigh)
  end do
  do i_wall = n_top_wall + 2*n_cell_w_y + 1, &
       n_top_wall + 2*n_cell_w_x*n_cell_w_y - 2*n_cell_w_y
    call assign_neighbors(n_cell_w_y, i_wall, &
         a_neigh, b_neigh, c_neigh, d_neigh, e_neigh, f_neigh)
    call apply_pbc(2*n_cell_w_y, i_wall, a_neigh, b_neigh, c_neigh, d_neigh,&
         e_neigh, f_neigh)
    call write_fk_neighbors(i_wall, n_mon_tot, a_neigh, b_neigh, c_neigh, &
         d_neigh, e_neigh, f_neigh)
  end do
  ! last row of cells
  do i_wall =  n_top_wall + 2*n_cell_w_x*n_cell_w_y - 2*n_cell_w_y + 1, &
       n_top_wall + 2*n_cell_w_x*n_cell_w_y
    call assign_neighbors(n_cell_w_y, i_wall, &
         a_neigh, b_neigh, c_neigh, d_neigh, e_neigh, f_neigh)
    if(mod(i_wall, 2) .eq. 0) then
      b_neigh = b_neigh - 2*n_cell_w_x*n_cell_w_y
      c_neigh = c_neigh - 2*n_cell_w_x*n_cell_w_y
      d_neigh = d_neigh - 2*n_cell_w_x*n_cell_w_y
    else
      c_neigh = c_neigh - 2*n_cell_w_x*n_cell_w_y
    end if
    call apply_pbc(2*n_cell_w_y, i_wall, a_neigh, b_neigh, c_neigh, d_neigh,&
         e_neigh, f_neigh)
    call write_fk_neighbors(i_wall, n_mon_tot, a_neigh, b_neigh, c_neigh, &
         d_neigh, e_neigh, f_neigh)
  end do
  if(l_debug_create_fk_neigh_list) then
    do i_wall=1, n_wall
      call print_neighbors(i_wall, fk_neighbors(:, i_wall))
    end do
    print *, "DEBUG: Leaving SR create_fk_neigh_list"
  end if
end subroutine create_fk_neigh_list
!------------------------------------------------------------------------------
!------------------------------------------------------------------------------
! writes the neighbor indices to the list.  The offset converts to the
! particle index used for enumerating the particles.
!------------------------------------------------------------------------------
subroutine write_fk_neighbors(center, offset, a, b, c, d, e, f)
  integer, intent(in) :: center, offset, a, b, c, d, e, f
  fk_neighbors(1, center) = a + offset
  fk_neighbors(2, center) = b + offset
  fk_neighbors(3, center) = c + offset
  fk_neighbors(4, center) = d + offset
  fk_neighbors(5, center) = e + offset
  fk_neighbors(6, center) = f + offset
end subroutine write_fk_neighbors
!------------------------------------------------------------------------------
!------------------------------------------------------------------------------
! assigns the neighbor indices around a particle center to a,b,c,d,e,f
! the periodic boundary conditions are treated later
subroutine assign_neighbors(n_cell_wall_y, center, a, b, c, d, e, f)
  integer, intent(in)  :: center, n_cell_wall_y
  integer, intent(out) :: a, b, c, d, e, f
  integer :: two_n_cell
  ! in case we have different walls, we can treat different n_cell_wall_y
  two_n_cell = 2 * n_cell_wall_y
  ! here's the layout:
  !
  !    a   b
  !     \ /
  ! f -center- c
  !     / \
  !    e   d
  !
  ! it helps to make a drawing showing how the particle index runs...

  if(mod(center, 2) .eq. 1) then ! center odd
    a = center - two_n_cell + 1
    b = center + 1
    c = center + two_n_cell
    d = center - 1
    e = center - two_n_cell - 1
    f = center - two_n_cell
  else ! center even
    a = center + 1
    b = center + two_n_cell + 1
    c = center + two_n_cell
    d = center + two_n_cell - 1
    e = center - 1
    f = center - two_n_cell
  end if
end subroutine assign_neighbors
!------------------------------------------------------------------------------
!------------------------------------------------------------------------------
! apply periodic boundary conditions, that is, take into account wrapping
! of wall particle indices after two_n_cell particles.  To understand the
! code, draw a picture!
subroutine apply_pbc(two_n_cell, center, a, b, c, d, e, f)
  integer, intent(in)    :: two_n_cell, center
  integer, intent(inout) :: a, b, c, d, e, f
  if(mod(center, two_n_cell) .eq. 0) then
    a = a - two_n_cell
    b = b - two_n_cell
  else if(mod(center, two_n_cell) .eq. 1) then
    d = d + two_n_cell
    e = e + two_n_cell
  end if
```

```
  end subroutine apply_pbc
  !------------------------------------------------------------------------
  !------------------------------------------------------------------------
  ! prints the neighbor indices around center
  ! Compiling note:  Some compilers (PGI, Intel) understand the backslash as
  ! an escape sequence and will complain about the "\" below.
  subroutine print_neighbors(center, neighbors)
    integer, intent(in)   :: center
    integer, dimension(n_fk_neighbors) :: neighbors
    ! the output should look like this (for 6 neighbors):
    !
    !    aaa  bbb
    !      \  /
    ! fff-center-ccc
    !      /  \
    !    eee  ddd
    !
    !     aaaaa bbbbb
    !         \   /
    ! fffff-center-ccccc
    !         /   \
    !     eeeee ddddd

    write(unit=*, fmt='(a, i5, a, i5, 6i5)') &
        "DEBUG: Particle", center, " has neighbors", neighbors
    if(n_fk_neighbors .eq. 6) then
      if(maxval(neighbors(:)) .lt. 1000) then
        write(unit=*, fmt='(a, i3, a, i3)') &
            "DEBUG:    ", neighbors(1), "  ", neighbors(2)
        write(unit=*, fmt='(a)') &
            "DEBUG:      \  /"
        write(unit=*, fmt='(a, i3, a, i4, a, i3)') &
            "DEBUG: ", neighbors(6), "- ", center, " -", neighbors(3)
        write(unit=*, fmt='(a)') &
            "DEBUG:      /  \"
        write(unit=*, fmt='(a, i3, a, i3)') &
            "DEBUG:    ", neighbors(5), "  ", neighbors(4)
      else ! large indices
        write(unit=*, fmt='(a, i5, a, i5)') &
            "DEBUG:    ", neighbors(1), "  ", neighbors(2)
        write(unit=*, fmt='(a)') &
            "DEBUG:        \    /"
        write(unit=*, fmt='(a, i5, a, i6, a, i5)') &
            "DEBUG: ", neighbors(6), "-", center, "-", neighbors(3)
        write(unit=*, fmt='(a)') &
            "DEBUG:        /    \"
        write(unit=*, fmt='(a, i5, a, i5)') &
            "DEBUG:    ", neighbors(5), "  ", neighbors(4)
      end if
    else if(n_fk_neighbors .eq. 2) then
      write(unit=*, fmt='(a, i3, a, i4, a, i3)') &
          "DEBUG: ", neighbors(1), "- ", center, " -", neighbors(2)
    end if
  end subroutine print_neighbors
  !------------------------------------------------------------------------
  !------------------------------------------------------------------------
  ! add up potential contributions to pressure tensor
  subroutine add_to_press_tensor(force, delta_r, press_tensor)
    real(kind=dp), dimension(n_dim, n_dim), intent(inout) :: press_tensor
    real(kind=dp), dimension(n_dim), intent(in) :: force
    real(kind=dp), dimension(n_dim), intent(in) :: delta_r
    integer :: index_a, index_b
    do index_a = 1, n_dim
      do index_b = index_a, n_dim
        press_tensor(index_a, index_b) = press_tensor(index_a, index_b) &
            + force(index_a) * delta_r(index_b)
      end do
    end do
!!$    print *, "DEBUG: press_tens_pot(:, :):"
!!$    print *, press_tens_pot(:, :)
!!$    print *, ""
  end subroutine add_to_press_tensor
  !------------------------------------------------------------------------
end module interaction
```

mcfluidV1.91.f90

```
! mcfluidVx.y.f90
! Recoil Growth algorithm implementation based on F77 code by Thijs J.H. Vlugt
! Martin Aichele, 2002-01-16
! last changed 2002-12-12
! V1.82 implements a faster algorithm for the calculation of the intra-chain
! potential, which is especially much faster for long chains and still faster
! for small chains.
! (by a factor of 5 at N=1024 as compared to the simple loop over all monomers)
! Martin Aichele, 2002-12-12
! V1.9 real n_dim systems code
! Martin Aichele, 2003-02-24
! last modified 2003-03-11
module mcfluid
  use globals
  use luxury
  use utilities
  use polymer
  ! for debugging only
  ! use mdroutines, only: binning3d_check_range
  implicit none
  !------------------------------------------------------------------------
  ! debug switches      |-------- 31 characters ------|
  !------------------------------------------------------------------------
  logical, parameter :: l_debug_mc_fluid_setup          = .FALSE.
  logical, parameter :: l_debug_mc_gyr_tens             = .FALSE.
  logical, parameter :: l_debug_grow_recoil             = .FALSE.
  logical, parameter :: l_debug_open_dir                = .FALSE.
  logical, parameter :: l_debug_put_chain_in_bins       = .FALSE.
  logical, parameter :: l_debug_e_monomer               = .FALSE.
  logical, parameter :: l_debug_ran_uniform             = .FALSE.
  logical, parameter :: l_debug_draw_random_pos         = .FALSE.
  logical, parameter :: l_debug_insert_chain            = .FALSE.
  logical, parameter :: l_debug_bin_fluid_particles     = .FALSE.
  logical, parameter :: l_debug_pot_intra_chain         = .FALSE.
  logical, parameter :: l_debug_pot_walls               = .FALSE.
  logical, parameter :: l_debug_pot_fluid               = .FALSE.
  logical, parameter :: l_debug_check_xybx_all          = .FALSE.
  logical, parameter :: l_debug_bonds_cross             = .FALSE.
  logical, parameter :: l_debug_resolve_bond_crossings  = .FALSE.
  !------------------------------------------------------------------------
  !------------------------------------------------------------------------
  ! switches for making the subroutines more talkative
  logical, parameter :: l_check_xy_bond_xings_self_verb = .FALSE.
  logical, parameter :: l_insert_chain_verb             = .FALSE.
  logical, parameter :: l_mc_moves_verbose              = .TRUE.
  !------------------------------------------------------------------------
  !------------------------------------------------------------------------
  ! check for periodic boundary condition (pbc) folding
  logical, parameter :: l_pbc_fold_check                = .FALSE.
  !------------------------------------------------------------------------
  !------------------------------------------------------------------------
  ! if a initial configuration is to be created
  logical ::            l_create_initial_mc_conf        = .TRUE.
  !------------------------------------------------------------------------

  !=========================!
  ! Recoil Growth algorithm !
  !=========================!
  ! array for holding the parent node of a tree node
  integer, dimension(:), allocatable :: Parent
  ! number of childs of a node
  ! the number of childs must not exceed huge(1_kind_number)
  integer, dimension(:), allocatable :: Nchild
  ! maximal tree search depth
  ! 2^31-1 = 2147483647 = huge(1) on 32 bit systems
  integer :: Maxtree = 2**24 ! 1 Mega
  ! depth of recoiling
  integer :: Nrecoil = 12
  ! Nchoi(i): number of choices in tree generation i (Nchoi(1) = 1)
  integer, dimension(n_mon) :: Nchoi
  ! number of choices at first and last monomer with Nchoi(i_mon) .gt. 1
  real(kind=dp) :: n_choice_first = 2.0_dp
  real(kind=dp) :: n_choice_last = 2.0_dp
  ! maximal bond angle when drawing bonds
  ! 120 degrees (typical bead-spring polymer bond-angle)
  ! real(kind=dp) :: open_angle = twopi / 3.0_dp
  ! for getting the real equilibrium distribution, set open_angle = pi
  real(kind=dp) :: open_angle = pi
  ! switch for deciding if bond before is to be taken as direction
  logical :: l_open_angle_eq_pi = .TRUE.
  ! minimal initial wall distace in units of sigma for z-direction
  real(kind=dp) :: min_initial_wall_distance = 0.0_dp
  ! prefactor for scaling the bond-length.  Makes bond-crossings less likely
  ! and the chains more compact.
  real(kind=dp) :: scale_bond_length = 1.0_dp
  ! bond-length, natural bond-length
  real(kind=dp) :: bond_len = 0.0_dp, bond_length_natural = 0.0_dp
  ! initial sigma for the fluid (which is ramped during the relaxation)
  real(kind=dp) :: sigma_fluid_init = 0.0_dp
  ! original value for the rest of the simulation
  real(kind=dp) :: sigma_fluid_natural = 0.0_dp
  ! number of tries to build a complete chain
  integer :: n_max_chain_grow_tries = 10000
  ! i/o units
  integer, parameter :: mc_sim_log_file_unit = 80
  integer, parameter :: mc_energy_log_unit   = 81
  integer, parameter :: mc_gyr_tens_log_unit = 82
  integer, parameter :: mc_end_pos_unit      = 83
  integer, parameter :: mc_bond_angle_unit   = 84
  ! switches for writing
  logical :: l_write_positions_mc = .FALSE., &
       l_write_mc_energy  = .TRUE., l_write_mc_gyr_tens   = .TRUE., &
       l_write_mc_end_pos = .TRUE., l_write_mc_bond_angle = .TRUE.
  ! MC loop variables (like for MD)
  integer :: i_time_mc = 0, s_time_mc = 0, n_relax_mc = 0, n_obser_mc = 0, &
       n_save_mc = huge(1)
  ! number of MC moves between configuration storage
  integer :: n_linear_out_mc = huge(1)
  ! sample times
  logical :: l_read_sample_list_mc = .FALSE.
  integer :: nr_samples_total_mc = huge(1)
  integer, save :: next_sample_time_mc = huge(1), &
       next_sample_index_mc = huge(1)
  integer, dimension(:), pointer, save :: sample_times_mc
  character(len=400) :: file_sample_list_mc = 'file_name_of_sample_list_mc'
  ! Configurational Bias Monte Carlo algorithm parameters
  !=========================================================
  ! For 2-d and highly constraint geometries, the Recoil Growth algorithm
  ! works much better.
  ! number of trial directions for MC step
  ! In a highly constrained situation and 3-d trial vectors it is advantageous
  ! to sample more points (say 128 -- 1024)
  ! If n_trial_vectors is made smaller, then  n_max_trial_vector_draws should
  ! be set to a bigger value.
  integer, parameter :: n_trial_vectors = 256
  ! number of trial vectors to be assumed to be "free" for calculation of the
  ! old rosenbluth factor.  The fewer free vectors you assume for a "good"
  ! chain, the higher the acceptance rate, but also the worse the quality of
  ! the accepted chains.  Try to match the fraction to
  ! 0.5 (z_space_wall - 2*range)/range, i.e. the fraction of the surface
  ! with no repulsive interaction with the walls on and the total surface of
  ! a ball lying in the middle between the walls.  1/8 corresponds to about
  ! z_space_wall=2.6 and a purely repulsive system.
  integer, parameter :: n_free_trial_vectors = n_trial_vectors/8
  ! switch for trying to resolve bond-crossings in the x-y plane by swapping
  ! positions
  logical, parameter :: l_resolve_bond_crossings        = .FALSE.
  ! the creation of thin layers with trial vectors drawn  from a sphere is
  ! difficult, as a lot of the weights will be 0 due to wall interaction.
  ! Use a large n_trial_vectors and a small n_free_trial_vectors (1, e.g.) to
  ! get a reasonable acceptance rate.
  ! number of allowed redraws of trial vectors for a step from one monomer to
  ! the next.  Trial vectors are redrawn when all weights are identical to 0,
  ! that means that the chain is in a hole and can't grow any further.
  integer, parameter :: n_max_trial_vector_draws = 10
  ! number of allowed redraws of the first position of the chain.
  ! It is redrawn when the weight is (numerically) zero.  For walls which are
  ! quite close to each other, this can happen quite frequently.
  integer, parameter :: n_max_first_position_draws = 1000
  ! number of tries to build a complete chain
  integer, parameter :: n_max_chain_build_tries = 10000

contains
  !------------------------------------------------------------------------
  ! initializes the MC fluid setup routines
  subroutine init_mc_fluid(mc_fluid_params_file)
    character(len=*), intent(in) :: mc_fluid_params_file
    integer, parameter :: fileunit = 10
    integer            :: io_status, counter, i_mon
    character(len=65)  :: string
    real(kind=dp) :: slope, rest
    counter = 0
    ! read parameters from file
    open(unit=fileunit, file=mc_fluid_params_file, &
         iostat=io_status, action="read", status="old")
    if(io_status /= 0) then
```

```
  write(unit=*, fmt='(3a)') "ERROR (SR init_mc_fluid): &
       &Could not read MC parameter file >>", mc_fluid_params_file, "<<"
  stop
else
  write(unit=*, fmt='(3a)') "MESSAGE (SR init_mc_fluid): &
       &Reading MC parameter file >>", mc_fluid_params_file, "<<"
end if
! read line by line
!------------------
read(unit=fileunit, iostat=io_status, fmt='(i14, a)') s_time_mc, string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, i14, a)') "MESSAGE: Read ", s_time_mc, string
end if
read(unit=fileunit, iostat=io_status, fmt='(i14, a)') n_relax_mc, string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, i14, a)') "MESSAGE: Read ", n_relax_mc, string
end if
read(unit=fileunit, iostat=io_status, fmt='(i14, a)') n_obser_mc, string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, i14, a)') "MESSAGE: Read ", n_obser_mc, string
end if
read(unit=fileunit, iostat=io_status, fmt='(i14, a)') n_save_mc, string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, i14, a)') "MESSAGE: Read ", n_save_mc, string
end if
read(unit=fileunit, iostat=io_status, fmt='(i14, a)') &
     n_linear_out_mc, string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, i14, a)') "MESSAGE: Read ", n_linear_out_mc, string
end if
read(unit=fileunit, iostat=io_status, fmt='(l14, a)') &
     l_write_positions_mc, string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, l14, a)') "MESSAGE: Read ", l_write_positions_mc, string
end if
read(unit=fileunit, iostat=io_status, fmt='(l14, a)') &
     l_read_sample_list_mc, string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, l14, a)') "MESSAGE: Read ", l_read_sample_list_mc, string
end if
read(unit=fileunit, iostat=io_status, fmt='(a)') file_sample_list_mc
counter = counter + 1
if(io_status /= 0) then
  print *, "ERROR: Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, a)') "MESSAGE: Read ", trim(file_sample_list_mc)
end if
read(unit=fileunit, iostat=io_status, fmt='(a)') string
counter = counter + 1
if(io_status /= 0) then
  print *, "ERROR: Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, a)') "MESSAGE: Read ", string
end if
read(unit=fileunit, iostat=io_status, fmt='(i14, a)') &
     n_max_chain_grow_tries, string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, i14, a)') "MESSAGE: Read ", n_max_chain_grow_tries, string
end if
read(unit=fileunit, iostat=io_status, fmt='(i14, a)') Nrecoil, string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, i14, a)') "MESSAGE: Read ", Nrecoil, string
end if
read(unit=fileunit, iostat=io_status, fmt='(i14, a)') Maxtree, string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, i14, a)') "MESSAGE: Read ", Maxtree, string
end if
read(unit=fileunit, iostat=io_status, fmt='(e14.6, a)') n_choice_first, &
     string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, e14.6, a)') "MESSAGE: Read ", n_choice_first, string
end if
read(unit=fileunit, iostat=io_status, fmt='(e14.6, a)') n_choice_last, &
     string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, e14.6, a)') "MESSAGE: Read ", n_choice_last, string
end if
read(unit=fileunit, iostat=io_status, fmt='(e14.6, a)') open_angle, &
     string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, e14.6, a)') "MESSAGE: Read ", open_angle, string
end if
read(unit=fileunit, iostat=io_status, fmt='(e14.6, a)') &
     min_initial_wall_distance, string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, e14.6, a)') "MESSAGE: Read ", &
       min_initial_wall_distance, string
end if
if(n_dim .eq. n_dim_pbc+1) then
  ! define minimal initial distance to the walls in z-direction.
  ! The RG-algo will automatically exclude high wall-fluid potential
  ! configurations, but the acceptance rate is higher with
  ! min_initial_wall_distance > 0 at the price of not truly being in
  ! equilibrium.  However, for fluctuating walls in MD constant-pressure
  ! simulations this is not important
  min_initial_wall_distance = min_initial_wall_distance &
       * minval(sigma(:, :))
  ! check if there's room for a particle
  if(.not. l_2d_flat_setup) then
    if(boundary(n_dim) .le. 2.0_dp * min_initial_wall_distance) then
      write(unit=*, fmt='(a, g13.6, a, g13.6)') &
           "ERROR (SR init_mc_fluid): boundary(n_dim)=", &
           boundary(n_dim), " .le. 2*min_initial_wall_distance=", &
           2.0_dp * min_initial_wall_distance
      stop
    end if
  end if
end if
read(unit=fileunit, iostat=io_status, fmt='(e14.6, a)') &
     scale_bond_length, string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, e14.6, a)') "MESSAGE: Read ", scale_bond_length, string
end if
read(unit=fileunit, iostat=io_status, fmt='(e14.6, a)') &
     sigma_fluid_init, string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, e14.6, a)') "MESSAGE: Read ", sigma_fluid_init, string
end if
read(unit=fileunit, iostat=io_status, fmt='(a)') string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, a)') "MESSAGE: Read ", string
end if
read(unit=fileunit, iostat=io_status, fmt='(l14, a)') &
     l_write_mc_energy, string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, l14, a)') "MESSAGE: Read ", l_write_mc_energy, string
end if
read(unit=fileunit, iostat=io_status, fmt='(l14, a)') &
     l_write_mc_gyr_tens, string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, l14, a)') "MESSAGE: Read ", l_write_mc_gyr_tens, string
end if
read(unit=fileunit, iostat=io_status, fmt='(l14, a)') &
     l_write_mc_end_pos, string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, l14, a)') "MESSAGE: Read ", l_write_mc_end_pos, string
end if
read(unit=fileunit, iostat=io_status, fmt='(l14, a)') &
     l_write_mc_bond_angle, string
counter = counter + 1
if(io_status /= 0) then
  write(unit=*, fmt='(a,i3,a)') "ERROR (SR init_mc_fluid): &
       &Wrong format when reading ", counter, "th format."
  stop
else
  write(*, '(a, l14, a)') "MESSAGE: Read ", l_write_mc_bond_angle, string
end if
close(fileunit)
! check Nrecoil
if(Nrecoil .le. 0) then
  write(unit=*, fmt='(a)') "ERROR (SR init_mc_fluid): &
       &Recoil length must be greater 0"
  stop
end if
! convert open_angle from degrees to radians
open_angle = twopi * (open_angle / 360.0_dp)
if(abs(open_angle - pi) .lt. (10.0_dp)**(-precision(pi)+1)) then
  l_open_angle_eq_pi = .TRUE.
  write(unit=*, fmt='(a)') "MESSAGE (SR init_mc_fluid): &
       &l_open_angle_eq_pi set to .TRUE. (all directions possible)"
else
  l_open_angle_eq_pi = .FALSE.
end if
! check angle
if(open_angle .lt. 0.0_dp .or. open_angle .gt. pi) then
  write(unit=*, fmt='(a,g13.6,a)') "ERROR (SR init_mc_fluid): &
       &open_angle=", open_angle, " not in [0, pi]"
  stop
end if
! check radius of gyration writing for simple liquid
if(l_write_mc_gyr_tens .and. n_mon .eq. 1) then
  write(unit=*, fmt='(a)') "CAUTION (SR init_mc_fluid): &
       &Calculation of gyration tensor for simple liquid makes no sense"
  l_write_mc_gyr_tens = .FALSE.
```

```
  end if
  if(l_write_mc_bond_angle .and. n_mon .le. 2) then
    write(unit=*, fmt='(a, i6, a)') "CAUTION (SR init_mc_fluid): &
         &Calculation of bond-angles ", n_mon, "-mers makes no sense"
    l_write_mc_bond_angle = .FALSE.
  end if
  ! read file containing the ist of times at which configurations are written
  if(l_read_sample_list_mc) call read_sample_list(file_sample_list_mc, &
       sample_times_mc, s_time_mc, n_relax_mc, n_obser_mc)
  ! allocate tree search bookkeeping arrays
  allocate(Parent(Maxtree))
  allocate(Nchild(Maxtree))
  !ccccccccccccccccccccccccccccccccccccccccc
  ! Generate The Number Of Trial Directions    C
  !ccccccccccccccccccccccccccccccccccccccccc
  if(n_choice_first .lt. 1.0_dp) then
    write(unit=*, fmt='(a,g13.6,a)') "ERROR (SR init_mc_fluid): &
         &n_choice_first=", n_choice_first, " must not be < 1"
    stop
  end if
  if(n_choice_last .lt. 1.0_dp) then
    write(unit=*, fmt='(a,g13.6,a)') "ERROR (SR init_mc_fluid): &
         &n_choice_last=", n_choice_last, " must not be < 1"
    stop
  end if
  ! number of choices a each generation i_mon
  ! we can use different numbers for each i_mon
  Nchoi(1) = 1
  slope = real(n_choice_last - n_choice_first) / real(n_mon -1)
  rest = 0.0_dp
  do i_mon = 2, n_mon
      Nchoi(i_mon) = anint(real(i_mon -1, kind=dp) * slope &
           + n_choice_first + rest)
      ! memorize the rest, which we try to incorporate in the next number
      rest = real(i_mon -1, kind=dp) * slope + n_choice_first + rest &
           - real(Nchoi(i_mon), kind=dp)
  end do
  ! bond-length in the polymer
  !---------------------------
  ! We assume a homopolymer with fixed bond-length of type (1,1) for all
  ! monomers.
  ! This assumption enters implicitely in the whole algorithm:
  ! 1) the first bead is treated particularly, for heterogene polymers
  !    one would also have to decide randomly at which end to start
  ! 2) in the interaction in a chain we assume that the energy between next
  !    neighbors is always the same
  ! save original values
  bond_length_natural = bond_length(1, 1)
  sigma_fluid_natural = sigma(1, 1)
  ! bond-length and fluid-sigma can be rescaled independently.  However,
  ! this can lead to very stretched bonds (imagine a very small sigma and
  ! a large bond-length) where the FENE-potential diverges.
  ! The relevant argument for the FENE-potential is
  ! bond-length**2 / (r_chain**2 * sigma**2)
  ! The natural rescaling is to rescale bonds and sigma identically.
  if((scale_bond_length*bond_length_natural)/sigma_fluid_init &
       .ge. r_chain) then
    write(unit=*, fmt='(a,g13.6,a,g13.6,a)') "ERROR (SR init_mc_fluid): &
         &(scale_bond_length*bond_length_natural)/sigma_fluid_init=", &
         (scale_bond_length*bond_length_natural)/sigma_fluid_init, &
         " .ge. r_chain=", r_chain, ", FENE potential will diverge"
    stop
  end if
  ! reset the global fluid sigma and rescale the bond-length
  ! if we do relaxation
  if(n_relax_mc .gt. 0) then
    sigma(1, 1) =  sigma_fluid_init
    ! change consistently all other values depending on sigma(1,1)
    call reinit_lj_params
    bond_len = scale_bond_length * bond_length(1, 1)
  else
    bond_len = bond_length(1, 1)
  end if
  write(unit=*, fmt='(a,f8.6,a,f8.6,a)') "MESSAGE (SR init_mc_fluid): &
       &bond-length=", bond_len, ", (natural: ", bond_length_natural, ")"
  write(unit=*, fmt='(a,f8.6,a,f8.6,a)') "MESSAGE (SR init_mc_fluid): &
       &sigma(1,1)=", sigma(1, 1), ", (natural: ", sigma_fluid_natural, ")"
  ! open file-units for writing
  call open_mc_fluid_io_units
  ! Read old configuration if it exists, if not set flag to create one
  ! from scratch
  call init_mc_config("mc_conf_old")
end subroutine init_mc_fluid
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
subroutine init_mc_config(old_mc_configuration_file)
  character(len=*), intent(in) :: old_mc_configuration_file
  integer, parameter :: in_file = 10
  integer            :: io_status
  ! try to open file containing configuration data
  open(unit=in_file, file=old_mc_configuration_file, &
       iostat=io_status, action="read", status="old");
  ! close file
  close(in_file)
  if(io_status .ne. 0) then
    write(unit=*, fmt='(3a)') "MESSAGE (SR init_mc_config): &
         &Could not read configuration file for MC >>", &
         old_mc_configuration_file, "<<"
    l_create_initial_mc_conf = .TRUE.
  else
    if(l_use_config_rng) then
      call read_configuration_rng(old_mc_configuration_file)
    else
      call read_configuration(old_mc_configuration_file)
    end if
    ! check if chains were ripped by folding
    call chain_sens_fold
    ! if we start a new simulation with an old configuration, we shift the
    ! wall atomes back in the simulation box and also the chains centers of
    ! mass by changing the pbc counters
    if(s_time_mc .eq. 0) call shift_chains_in_box
    l_create_initial_mc_conf = .FALSE.
  end if
end subroutine init_mc_config
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! frees the memory taken by the large arrays for tree search bookkeeping
subroutine free_rg_memory
  if(allocated(Parent)) deallocate(Parent)
  if(allocated(Nchild)) deallocate(Nchild)
end subroutine free_rg_memory
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! initialize fluid by means of MC routines
subroutine mc_fluid_setup
  logical :: loverlap, l_accept_new_chain, l_store
  integer :: i_chain, i_mon, i_part
  real(kind=dp) :: weight_new, weight_old, en_new, en_old, en_total, &
       accept_prob
  integer :: node_count, node_count_min, node_count_max
  real(kind=dp) :: node_count_avg, node_count_sigma
  integer :: i_chain_grow_tries, i_mc_moves
  real(kind=dp), dimension(n_dim, n_mon) :: chain_new, chain_old
  real(kind=dp) :: ln_max_real
  ! variables for putting together filenames
  integer :: length
  character(len=240) :: filename
  character(len=21)  :: string
  logical :: l_calc_bond_en = .FALSE.
  integer :: i_loop, chain_moved_counter
  integer, dimension(n_chain) :: chain_moved
  if(l_debug_mc_fluid_setup) print *, "DEBUG: Entering SR mc_fluid_setup"
  ! sanity checks
  if(n_mon_tot .eq. 0 .or. .not. allocated(bin_fluid)) then
    write(unit=*, fmt='(a)') &
         "ERROR (SR mc_fluid_setup): There's no fluid."
    stop
  end if
  call init_mc_fluid("params_mc_fluid")
  ! set position writing switch to .false.
  l_store = .FALSE.
  ! initialize fluid bins
  bin_fluid(:, :, :, 0)  = 0
  bin_fluid(:, :, :, 1:) = huge(1)
  ! total energy
  en_total = 0.0_dp
  ! for catching numerical infinities in the calculation of accceptance
  ! probabilities
  ln_max_real = log(huge(1.0_dp))
  if(l_create_initial_mc_conf) then
    write(unit=*, fmt=('(a)'))&
         "MESSAGE (SR mc_fluid_setup): Setting up initial configuration"
    ! loop over all chains
    do i_chain = 1, n_chain
      i_chain_grow_tries = 1
      node_count_min = huge(1)
      node_count_max = 0
      node_count_avg   = 0.0_dp
      node_count_sigma = 0.0_dp
      ! loop over tries to set up a complete chain
      do
        ! try to insert a chain without overlap
        call Grow_Recoil(.false., loverlap , weight_new, node_count, &
             chain_new, i_chain)
        node_count_min = min(node_count, node_count_min)
        node_count_max = max(node_count, node_count_max)
        node_count_avg   = node_count_avg   + real(node_count, kind=dp)
        node_count_sigma = node_count_sigma + real(node_count, kind=dp)**2
        if(l_debug_mc_fluid_setup) &
             write(unit=*, fmt='(a,l1,a,g13.6,a,i12)')&
             "DEBUG (SR mc_fluid_setup): loverlap=", loverlap, &
             ", weight=", weight_new, ", node_count=", node_count
        ! accept any chain
        if(.not. loverlap) then
          call put_chain_in_bins(i_chain, chain_new, .FALSE.)
          ! a chain should never have crossings in 2-d.  If this is the case
          ! then the potential parameters have to be changed.
          if(l_forbid_xy_bond_crossings .and. n_dim_pbc .eq. 2) then
            if(.not. check_xy_bond_crossings_all(i_chain)) then
              write(unit=*, fmt='(a)') "ERROR (SR mc_fluid_setup): &
                   &xy-plane bond crossings detected."
              call fluid_positions_out("mc_xy-bonds-cross_t", i_time_mc)
              stop
            end if
          end if
          node_count_avg = node_count_avg / real(i_chain_grow_tries, kind=dp)
          node_count_sigma =  node_count_sigma &
               / real(i_chain_grow_tries, kind=dp)
          node_count_sigma = sqrt(node_count_sigma - node_count_avg**2)
          write(unit=*, fmt='(a, i8, a, i8, a)') &
               "MESSAGE: chain", i_chain, ":", i_chain_grow_tries, &
               " calls to SR Grow_Recoil"
          write(unit=*, fmt='(4(a,g11.4))') &
               "MESSAGE: node count: min=", node_count_min, ", max=", &
               node_count_max, ", avg=", node_count_avg, ", sigma=", &
               node_count_sigma
          ! get total energy of this chain (SR e_chain must be used, not
          ! SR e_chain_old, as chain_new(:) is not folded)
          call e_chain(i_chain, chain_new, en_new)
          en_total = en_total + en_new
          if(.FALSE.) then
            ! integer to string conversion
            write(string, '(i12)') i_chain
            string = adjustl(string)
            ! put together the filename
            filename = "chain_"//trim(string)
            length = len(trim(adjustl(filename)))
            filename = adjustl(filename(:length))
            call fluid_positions_out(filename, 0, &
                 (i_chain-1) * n_mon +1, i_chain * n_mon)
          end if
          ! start to work on next chain
          exit
        else if(i_chain_grow_tries .eq. n_max_chain_grow_tries) then
          write(unit=*, fmt='(a)') &
               "ERROR (SR mc_fluid_setup): maximal number of chain &
               &insertion tries exceeded."
          stop
        else
          i_chain_grow_tries = i_chain_grow_tries +1
        end if
      end do
    end do
  else
    ! bin all fluid particles
    call bin_fluid_particles(1, n_mon_tot)
  end if
  !-------------------------!
  ! equilibrate by MC moves !
  !-------------------------!
  if(n_obser .gt. s_time) then
    write(unit=*, fmt='(a)') "MESSAGE (SR mc_fluid_setup): &
         &Starting MC equilibration"
  end if
  if(l_write_positions_mc) then
    if(n_dim .eq. n_dim_pbc) then
      call fluid_positions_out("mc_fl_pos_t", 0)
    else
      call particle_positions_out("mc_pos_t", 0)
    end if
  end if
  if(l_write_mc_energy) &
       write(unit=mc_energy_log_unit, fmt='(i12, g13.5)') 0, en_total
  if(l_write_mc_gyr_tens)   call mc_gyr_tens(0)
  if(l_write_mc_end_pos)    call mc_end_pos_out(0)
  if(l_write_mc_bond_angle) call mc_bond_angle(0)
  !=====================!
  ! main Monte Carlo loop !
```

```
!======================!
! count MC moves
i_mc_moves = 0
! keep track of chain moves for each chain
chain_moved(:) = 0
node_count_min = huge(1)
node_count_max = 0
node_count_avg   = 0.0_dp
node_count_sigma = 0.0_dp
! if the LJ-parameters are ramped, the bond-energy changes so it
! has to be included in the total energy of a chain
if(n_relax_mc .gt. 0) l_calc_bond_en = .TRUE.
! s_time is the number of steps already completed, so the loop starts at
! the next step, thus '+1'
do i_time_mc = s_time_mc+1, n_relax_mc + n_obser_mc
  ! ramp during n_relax_mc steps
  if(i_time_mc .le. n_relax_mc .and. n_relax_mc .gt. 0) then
    bond_len = scale_bond_length * bond_length(1, 1) &
         + ((bond_length_natural - scale_bond_length * bond_length(1, 1)) &
         / real(n_relax_mc, kind=dp)) * real(i_time_mc, kind=dp)
    sigma(1, 1) =  sigma_fluid_init &
         + ((sigma_fluid_natural - sigma_fluid_init) &
         / real(n_relax_mc, kind=dp)) * real(i_time_mc, kind=dp)
    call reinit_lj_params
    if(i_time_mc .eq. n_relax_mc) then
      write(unit=*, fmt='(a)') "MESSAGE (SR mc_fluid_setup): &
           &Relaxation done"
      ! we still have to calculate the bond-energies until all chains
      ! have the same bonds
    end if
  end if
  ! select a chain at random
  i_chain = int(ran_uniform() * real(n_chain, kind=dp)) +1
  if(l_debug_mc_fluid_setup) &
       write(unit=*, fmt='(a,i12,a,i6,a,i12)') &
       "DEBUG (SR mc_fluid_setup): &
       &i_time_mc=", i_time_mc,", i_chain=", i_chain, &
       ", i_mc_moves=", i_mc_moves
  ! write chain positions on helper array
  do i_mon = 1, n_mon
    i_part = (i_chain -1) * n_mon + i_mon
    chain_old(:, i_mon) = r0(i_part, :)
  end do
  ! get weight for old chain
  call Grow_Recoil(.true., loverlap, weight_old, node_count, &
       chain_old, i_chain)
  if(loverlap) then
    write(unit=*, fmt='(a, i12, a)') &
         "ERROR (SR mc_fluid_setup): Old chain ", i_chain, " has overlap"
    stop
  end if
  ! try to grow new chain
  call Grow_Recoil(.false., loverlap, weight_new, node_count, &
       chain_new, i_chain)
  node_count_min = min(node_count, node_count_min)
  node_count_max = max(node_count, node_count_max)
  node_count_avg   = node_count_avg   + real(node_count, kind=dp)
  node_count_sigma = node_count_sigma + real(node_count, kind=dp)**2
  ! if this chain has an overlap we know that no MC move will result
  if(.not. loverlap) then
    ! get energies
    call e_chain_old(i_chain, chain_old, en_old, l_calc_bond_en)
    call e_chain(i_chain, chain_new, en_new, l_calc_bond_en)
    ! catch cases where the energy term is too big (we only test the case
    ! en_old > en_new, the other one gives an underflow = 0.0)
    if(temp_inv * (en_old - en_new) .ge. ln_max_real) then
      write(unit=*, fmt='(2(a,g13.6))') "MESSAGE (SR mc_fluid_setup): &
           &+Infinity for energy factor in Pacc calculation"
      accept_prob = 1.0_dp
    else
      ! while the energy computed for a chain is always the same for the
      ! same chain, the weight is *not*, because we draw different random
      ! trial directions each time (but the average weight reaches a limit
      ! for an infinite number of weight calculations)
      accept_prob = (weight_new / weight_old) &
         * exp(temp_inv * (en_old - en_new))
    end if
    if(l_debug_mc_fluid_setup) then
      write(unit=*, fmt='(2(a,g13.6))') "DEBUG (SR mc_fluid_setup): &
           &weight_new=", weight_new, ", weight_old=", weight_old
      write(unit=*, fmt='(3(a,g13.6))') "      en_new=", en_new, &
           ", en_old=", en_old, ", accept_prob=", accept_prob
    end if
    if(ran_uniform() .lt. accept_prob) then
      l_accept_new_chain = .TRUE.
    else
      l_accept_new_chain = .FALSE.
    end if
    ! decide if new chain is accepted
    if(l_accept_new_chain) then
      if(l_debug_mc_fluid_setup) print *, "DEBUG: Accepted MC-move"
      call put_chain_in_bins(i_chain, chain_new, .TRUE.)
      if(l_forbid_xy_bond_crossings .and. n_dim_pbc .eq. 2) then
        if(.not. check_xy_bond_crossings_all()) then
          write(unit=*, fmt='(a)') "ERROR (SR mc_fluid_setup): &
               &xy-plane bond crossings detected."
          call fluid_positions_out("mc_xy-bonds-cross_t", i_time_mc)
          stop
        end if
      end if
      if(l_mc_moves_verbose) &
           write(unit=*, fmt='(5(a,g10.4))') "MESSAGE: &
           &Move: Wn=", weight_new, ", Wo=", weight_old, &
           ", En=", en_new, ", Eo=", en_old, ", Pacc=", accept_prob
      i_mc_moves = i_mc_moves +1
      if(i_time_mc .gt. n_relax_mc) &
           chain_moved(i_chain) = chain_moved(i_chain) + 1
      en_total = en_total - en_old + en_new
      if(l_write_mc_energy) write(unit=mc_energy_log_unit, &
           fmt='(i12, g13.5)') i_time_mc, en_total
      if(l_write_mc_gyr_tens)   call mc_gyr_tens(i_time_mc)
      if(l_write_mc_end_pos)    call mc_end_pos_out(i_time_mc)
      if(l_write_mc_bond_angle) call mc_bond_angle(i_time_mc)
    end if ! move accepted
  end if ! (.not. loverlap) for new chain
  ! MC step completed
  !==================
  ! write out safety copy every n_save_mc MCS
  if((i_time_mc.gt.s_time_mc) .and. (mod(i_time_mc, n_save_mc).eq.0)) then
    if(l_use_config_rng) then
      call store_configuration_rng("mc_last_save_file", .TRUE.)
    else
      call store_configuration("mc_last_save_file")
    end if
  end if
  if(n_relax_mc + n_obser_mc - s_time_mc .gt. 10 .and. &
       mod(i_time_mc, (n_relax_mc + n_obser_mc - s_time_mc) / 10) .eq. 0) &
       write(unit=*, fmt='(a, i12)') "MESSAGE: Finished MCS", i_time_mc
  if(l_write_positions_mc) then
    if(mod(i_time_mc, n_linear_out_mc) .eq. 0) l_store = .TRUE.
    if(l_read_sample_list_mc .and. &
         (i_time_mc .eq. next_sample_time_mc)) then
      call get_next_sample_time(sample_times_mc, next_sample_time_mc, &
           next_sample_time_mc)
      ! if we don't want to write so far, change that
      if(.not. l_store) l_store = .TRUE.
    end if
  end if
  if(l_store) then
    if(n_dim .eq. n_dim_pbc) then
      call fluid_positions_out("mc_fl_pos_t", i_time_mc)
    else
      call particle_positions_out("mc_pos_t", i_time_mc)
    end if
    ! reset switch
    l_store = .FALSE.
  end if
end do ! loop over MC steps
if(l_write_positions_mc) then
  ! the loop index is incremented by one after the loop and we reset it
  i_time_mc = n_relax_mc + n_obser_mc
  if(n_dim .eq. n_dim_pbc) then
    call fluid_positions_out("mc_fl_pos_final_t", i_time_mc)
  else
    call particle_positions_out("mc_pos_final_t", i_time_mc)
  end if
end if
! if no simulation steps have been done, set bond-lemgth and sigma
! to natural values
if(s_time_mc .eq. n_relax_mc + n_obser_mc) then
  write(unit=*, fmt='(a)') "MESSAGE (SR mc_fluid_setup): &
       &No MC-steps done, set sigma(1,1) and bond_len to natural value"
  bond_len = bond_length_natural
  sigma(1,1) = sigma_fluid_natural
  call reinit_lj_params
end if
! check if the ramped values are at the natural values
if(sigma_fluid_natural .ne. sigma(1,1)) then
  write(unit=*, fmt='(2(a, f8.6))') "ERROR (SR mc_fluid_setup): &
     &sigma_fluid_natural=", sigma_fluid_natural, &
     " .ne. sigma(1,1)=", sigma(1,1)
  stop
end if
if(bond_length_natural .ne. bond_len) then
  write(unit=*, fmt='(2(a, f8.6))') "ERROR (SR mc_fluid_setup): &
     &bond_length_natural=", bond_length_natural, &
     " .ne. bond_len=", bond_len
  stop
end if
if(n_obser_mc .gt. s_time_mc) then
  ! store new configuration
  call store_configuration_rng("mc_conf_new", .TRUE.)
  node_count_avg = node_count_avg / real(n_obser_mc - s_time_mc, kind=dp)
  node_count_sigma =  node_count_sigma &
       / real(n_obser_mc - s_time_mc, kind=dp)
  node_count_sigma = sqrt(node_count_sigma - node_count_avg**2)
  write(unit=*, fmt='(4(a,g11.4))') &
       "MESSAGE: node count: min=", node_count_min, ", max=", &
       node_count_max, ", avg=", node_count_avg, ", sigma=", &
       node_count_sigma
  write(unit=*, fmt='(a,g13.6,a,f8.4,a)') &
       "MESSAGE (SR mc_fluid_setup): acceptance rate=", &
       real(i_mc_moves, kind=dp) / real(n_obser_mc - s_time_mc, kind=dp), &
       " (", 100.0_dp*real(i_mc_moves,kind=dp) &
       / real(n_obser_mc - s_time_mc,kind=dp), "%)"
  write(unit=mc_sim_log_file_unit, fmt='(3(a, i8))') &
       "n_mon=", n_mon, ", n_chain=", n_chain, ", n_mon_tot=", n_mon_tot
  if(n_dim .eq. n_dim_pbc+1) then
    write(unit=mc_sim_log_file_unit, fmt='(a, l2, a)') &
         "l_2d_flat_setup=", l_2d_flat_setup, " for system with walls"
  else
    write(unit=mc_sim_log_file_unit, fmt='(a, l2, a)') &
         "l_2d_flat_setup ignored for system without walls"
  end if
  if(n_dim_pbc .eq. 2) then
    write(unit=mc_sim_log_file_unit, fmt='(a, l2)') &
         "l_forbid_xy_bond_crossings=", l_forbid_xy_bond_crossings
  else
    write(unit=mc_sim_log_file_unit, fmt='(a)') &
         "l_forbid_xy_bond_crossings ignored as n_dim_pbc .ne. 2"
  end if
  write(unit=mc_sim_log_file_unit, fmt='(a)') &
       "Data observed during MC simulation:"
  write(unit=mc_sim_log_file_unit, fmt='(4(a,g11.4))') &
       "node count: min=", node_count_min, ", max=", &
       node_count_max, ", avg=", node_count_avg, ", sigma=", &
       node_count_sigma
  write(unit=mc_sim_log_file_unit, fmt='(a,g13.6,a,f8.4,a)') &
       "acceptance rate=", &
       real(i_mc_moves, kind=dp) / real(n_obser_mc - s_time_mc, kind=dp), &
       " (", 100.0_dp*real(i_mc_moves,kind=dp) &
       / real(n_obser_mc - s_time_mc,kind=dp), "%)"
  ! find out how many chains were moved and print statistics
  chain_moved_counter = 0
  do i_loop = 1, n_chain
    if(chain_moved(i_loop) .gt. 0) &
         chain_moved_counter = chain_moved_counter +1
  end do
  write(unit=*, fmt='(a, i8, a)') "MESSAGE (SR mc_fluid_setup): &
       &Moved", chain_moved_counter, " chains after relaxation"
  write(unit=mc_sim_log_file_unit, fmt='(a, i8, a)') &
       "Moved", chain_moved_counter, " chains after relaxation"
  write(unit=mc_sim_log_file_unit, fmt='(a)') &
       "Move statistics:"
  write(unit=mc_sim_log_file_unit, fmt='(a)') &
       "Chain index | number of moves after relaxation"
  do i_loop = 1, n_chain
!   write(unit=*, fmt='(a, i8, a, i8, a)') "MESSAGE: Chain", i_loop, &
!        " was moved", chain_moved(i_loop), " times"
    write(unit=mc_sim_log_file_unit, fmt='(2i8)') &
         i_loop, chain_moved(i_loop)
  end do
else
  write(unit=*, fmt='(a)') "MESSAGE (SR mc_fluid_setup): n_obser_mc .le. &
       &s_time_mc, so log-file not written."
end if
if(l_write_mc_energy)     close(mc_energy_log_unit)
if(l_write_mc_gyr_tens)   close(mc_gyr_tens_log_unit)
if(l_write_mc_end_pos)    close(mc_end_pos_unit)
if(l_write_mc_bond_angle) close(mc_bond_angle_unit)
close(mc_sim_log_file_unit)
call free_rg_memory
! if we already are in equilibrium, i.e. we used the natural sigma(1,1)
! and the natural bond length we don't have to ramp the potential in the
```

```
  ! MD-simulation, else we have to (as set in SR init_config)
  if(sigma_fluid_init .eq. sigma_fluid_natural .and. &
       scale_bond_length .eq. 1.0_dp) then
    r_2_min_init = 0.0_dp
    r_2_min      = 0.0_dp
    r_2_min_time = 0.0_dp
    l_r_2_min_finite = .FALSE.
    write(unit=*, fmt='(a,l2)') &
         "MESSAGE (SR mc_fluid_setup): MD-potential ramp switch &
         &l_r_2_min_finite =", l_r_2_min_finite
    write(unit=*, fmt='(a)') &
         "         as MC-simulation was done with correct potentials"
  end if
  if(l_debug_mc_fluid_setup) print *, "DEBUG: Leaving SR mc_fluid_setup"
end subroutine mc_fluid_setup
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! opens outout units
subroutine open_mc_fluid_io_units
  integer :: io_status
   open(unit=mc_sim_log_file_unit, file="mc_simulation_log.dat", &
        iostat=io_status, action="write", status="unknown")
   if(io_status .ne. 0) then
     write(unit=*, fmt='(a)') "ERROR (SR  open_mc_fluid_io_units): &
          &Could not open output file for MC simulation log"
     stop
   end if
  if(l_write_mc_energy) then
    open(unit=mc_energy_log_unit, file="mc_energy_log.dat", &
         iostat=io_status, action="write", status="unknown")
    if(io_status .ne. 0) then
      write(unit=*, fmt='(a)') "ERROR (SR  open_mc_fluid_io_units): &
           &Could not open output file for energy"
      stop
    end if
  end if
  if(l_write_mc_gyr_tens) then
    open(unit=mc_gyr_tens_log_unit, file="mc_gyr_tens_log.dat", &
         iostat=io_status, action="write", status="unknown")
    if(io_status .ne. 0) then
      write(unit=*, fmt='(a)') "ERROR (SR open_mc_fluid_io_units): &
           &Could not open output file for gyration tensor and &
           &end if-to-end distance"
      stop
    end if
  end if
  if(l_write_mc_end_pos) then
    open(unit=mc_end_pos_unit, file="mc_end_pos.dat", &
         iostat=io_status, action="write", status="unknown")
    if(io_status .ne. 0) then
      write(unit=*, fmt='(a)') "ERROR (SR open_mc_fluid_io_units): &
           &Could not open output file for end positions of chains"
      stop
    end if
  end if
  if(l_write_mc_bond_angle) then
    open(unit=mc_bond_angle_unit, file="mc_bond_angle.dat", &
         iostat=io_status, action="write", status="unknown")
    if(io_status .ne. 0) then
      write(unit=*, fmt='(a)') "ERROR (SR open_mc_fluid_io_units): &
           &Could not open output file for bond angles"
      stop
    end if
  end if
end subroutine open_mc_fluid_io_units
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! write the chain on the global folded coordinates r0 and re-initializes
! fluid bins
subroutine put_chain_in_bins(i_chain, chain_pos, l_full_config)
  integer, intent(in) :: i_chain
  real(kind=dp), dimension(n_dim, n_mon), intent(in) :: chain_pos
  logical, intent(in) :: l_full_config
  integer :: i_dim, i_mon, i_part
  if(l_debug_put_chain_in_bins) then
    print *, "DEBUG: Entering SR put_chain_in_bins"
    print *, "       i_chain=", i_chain, ", l_full_config=", l_full_config
  end if
  ! write chain coordinates on r0
  ! fold coordinates and memorize foldings
  do i_mon = 1, n_mon
    i_part = (i_chain -1) * n_mon + i_mon
    do i_dim = 1, n_dim_pbc
      r0(i_part, i_dim) = chain_pos(i_dim, i_mon)
      if(r0(i_part,i_dim).gt.boundary(i_dim)) then
        r0(i_part,i_dim) = r0(i_part,i_dim) - boundary(i_dim)
        pbc_count(i_part,i_dim) = 1
      else if(r0(i_part,i_dim).lt.0.0_dp) then
        r0(i_part,i_dim) = r0(i_part,i_dim) + boundary(i_dim)
        pbc_count(i_part,i_dim) = -1
      else ! leave position unchanged and set pbc_counter to 0
        pbc_count(i_part,i_dim) = 0
      end if
    end do
    r0(i_part, n_dim_pbc+1:n_dim) = chain_pos(n_dim_pbc+1:n_dim, i_mon)
  end do
  ! put new chain in bins
  if(l_full_config) then
    call bin_fluid_particles(1, n_mon_tot)
  else
    call bin_fluid_particles(1, i_chain * n_mon)
  end if
  ! define unfolded coordinates of this chain
  do i_dim = 1, n_dim_pbc
    r0_unfolded((i_chain -1) * n_mon +1:i_chain * n_mon, i_dim) &
         = r0((i_chain -1) * n_mon +1:i_chain * n_mon, i_dim) &
         + real(pbc_count((i_chain -1) * n_mon +1:i_chain * n_mon, i_dim), &
         kind=dp) * boundary(i_dim)
  end do
  r0_unfolded((i_chain -1) * n_mon +1:i_chain * n_mon, n_dim_pbc+1:n_dim) &
       = r0((i_chain -1) * n_mon +1:i_chain * n_mon, n_dim_pbc+1:n_dim)
  if(l_debug_put_chain_in_bins) &
       print *, "DEBUG: Leaving SR put_chain_in_bins"
  return
end subroutine put_chain_in_bins
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! writes out the gyration tensor, the end-to-end-distance and the mean
! distance of the monomers to the endpoints at MC step i_time_mc
subroutine mc_gyr_tens(i_time_mc)
  integer, intent(in) :: i_time_mc
  integer :: i_dim, j_dim
  real(kind=dp), dimension(n_dim, n_dim) :: gyr_ten
  if(l_debug_mc_gyr_tens) &
       print *, "DEBUG: Entering SR mc_gyr_tens"
  call calc_gyration_tensor(gyr_ten)
    write(unit=mc_gyr_tens_log_unit, advance="no", fmt='(i12)') i_time_mc
    do i_dim=1, n_dim
      do j_dim=i_dim, n_dim
        write (unit=mc_gyr_tens_log_unit, advance="no", fmt='(g13.5)') &
             gyr_ten(i_dim, j_dim) / real(n_mon_tot, kind=dp)
      end do
    end do
    write (unit=mc_gyr_tens_log_unit, advance="no", fmt='(g13.5)') &
         calc_end_to_end_dist() / real(n_chain, kind=dp)
    write (unit=mc_gyr_tens_log_unit, advance="yes", fmt='(g13.5)') &
         calc_mean_mon_end_dist() / real(2*n_mon_tot, kind=dp)
  if(l_debug_mc_gyr_tens) &
       print *, "DEBUG: Leaving SR mc_gyr_tens"
end subroutine mc_gyr_tens
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! write the positions of the chain ends at MCS i_time_mc
subroutine mc_end_pos_out(i_time_mc)
  integer, intent(in) :: i_time_mc
  character(len=40) :: format_string
  integer :: i_chain
  ! helper array
  real(kind=dp), dimension(2*n_dim, n_chain) :: end_pos
  ! on the fly generation of the format string
  write(format_string, fmt='(a,i6,a)') "(i12,", 2*n_chain*n_dim, &
       "(g14.6))"
  ! Fortran stores columns continuously (although this is not a strict
  ! requirement of the standard AFAIK)
  do i_chain = 1, n_chain
    end_pos(1:n_dim, i_chain)         = r0_unfolded((i_chain -1)*n_mon +1, :)
    end_pos(n_dim+1:2*n_dim, i_chain) = r0_unfolded(i_chain * n_mon, :)
  end do
  write(unit=mc_end_pos_unit, fmt=format_string) i_time_mc, end_pos
end subroutine mc_end_pos_out
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
subroutine mc_bond_angle(i_time_mc)
  integer, intent(in) :: i_time_mc
  real(kind=dp) :: min_angle, mean_angle, sigma_angle
  ! as long as the bond-length is fixed, we don't need to normalize the bond
  ! vectors to length 1
  call bond_angle_stats(.FALSE., min_angle, mean_angle, sigma_angle)
  write(unit=mc_bond_angle_unit, fmt='(i12, 3(g12.4))') i_time_mc, &
       min_angle, mean_angle, sqrt(sigma_angle)
end subroutine mc_bond_angle
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
subroutine bond_angle_stats(l_norm, min_cos_angle, mean_cos_angle, &
     var_cos_angle)
  ! if we have to normalize by a bond-length /= 1
  logical, intent(in) :: l_norm
  real(kind=dp), intent(out) :: min_cos_angle, mean_cos_angle, var_cos_angle
  integer :: i_mon, i_chain, i_part
  real(kind=dp) :: cos_angle
  min_cos_angle =  huge(1.0_dp)
  mean_cos_angle = 0.0_dp
  var_cos_angle = 0.0_dp
  do i_chain = 1, n_chain
    do i_mon = 2, n_mon -1
      i_part = (i_chain -1) * n_mon + i_mon
      cos_angle = dot_product( &
           r0_unfolded(i_part +1, :) - r0_unfolded(i_part, :), &
           r0_unfolded(i_part, :) - r0_unfolded(i_part -1, :))
      if(l_norm) cos_angle = cos_angle / (sqrt(dot_product( &
           r0_unfolded(i_part +1, :) - r0_unfolded(i_part, :), &
           r0_unfolded(i_part +1, :) - r0_unfolded(i_part, :))) &
           * sqrt(dot_product( &
           r0_unfolded(i_part, :) - r0_unfolded(i_part-1, :),&
           r0_unfolded(i_part, :) - r0_unfolded(i_part-1, :))))
      min_cos_angle = min(cos_angle, min_cos_angle)
      mean_cos_angle = mean_cos_angle + cos_angle
      var_cos_angle = var_cos_angle + cos_angle**2
    end do
  end do
  mean_cos_angle = mean_cos_angle / real(n_chain * (n_mon -2), kind=dp)
  var_cos_angle = var_cos_angle / real(n_chain * (n_mon -2), kind=dp)
  var_cos_angle = var_cos_angle - mean_cos_angle**2
  return
end subroutine bond_angle_stats
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! Recoil growth for one chain
subroutine Grow_Recoil(Lold, Lovrlap, Weight, node_count, chain_pos, Ichain)
  ! Lold: If an old chain is retraced or a new one is build
  logical, intent(in) :: Lold
  ! Lovrlap: Chain hits another particle and is stopped
  logical, intent(out) :: Lovrlap
  ! Weight: Weight of the chain (for deciding on acceptance)
  real(kind=dp), intent(out) :: Weight
  ! count how many tree nodes were visited
  integer, intent(out) :: node_count
  ! Ichain: Chain index
  integer, intent(in) :: Ichain
  ! chain_pos: Coordinates of the "new" chain, was: Xn,Yn,Zn
  real(kind=dp), dimension(n_dim, n_mon) :: chain_pos
  ! Lfixed:  Bead has been fixed
  logical, dimension(n_mon) :: Lfixed
  ! Lopen:   Direction is open
  logical :: Lopen
  ! flag for signalling infinite interaction energy (particle in a wall)
  logical :: l_infinity
  integer :: Ifix, i_mon, i_part, Ip, Ipoint
  ! Iparent: index of parent of a node
  ! Parent(i): array holding the index of the parent of a node
  ! Nchild(i): number of childs of a node
  ! remember: the number of childs must not exceed huge(1_kind_number)
  integer :: Iparent
  ! Ndir: number of free directions in tree generation i
  ! Whmain: index of node that constitutes monomer i_mon in the main chain
  integer, dimension(n_mon) :: Ndir, Whmain
  ! variables for finding free feelers from each fixed bead
  logical :: Lready1, Lready2
  integer :: mybead, maxbead
  ! Bond was: Bx,By,Bz; Trial was: Xt,Yt,Zt
  real(kind=dp), dimension(n_dim) :: Bond, Trial
  ! chain_pos_copy was: Xnn, Ynn, Znn
  real(kind=dp), dimension(n_dim, n_mon) :: chain_pos_copy
  real(kind=dp) :: En
  ! probability for a position to be open
  real(kind=dp) :: P_Open
  ! array for storing the probabilities for being open for the main chain
  real(kind=dp), dimension(n_mon) :: Pmain
```

```
if(l_debug_grow_recoil) print *, "DEBUG: Entering SR grow_recoil"
! initialize
temp_inv = 1.0_dp / temp
chain_pos(:, :) = 0.0_dp
! reset probability
Pmain(:)  = 0.0_dp
! so far we have not fixed anything
Lfixed(:) = .false.
Lovrlap   = .false.
! in an existing chain we always have a direction
Ndir(:) = 1
! if the chain already exists, then copy it on chain_pos which we use
! in the branch we we retrace an old chain.  This is done to emphasize
! the similarity of the branches
if(Lold) then
  do i_mon = 1, n_mon
    i_part = (Ichain -1) * n_mon + i_mon
    chain_pos(:, i_mon) = r0(i_part, :)
  end do
end if
!cccccccccccccccccccc
! Grow The Chains   C
!cccccccccccccccccccc
if(.not.Lold) then
  !cccccccccccccccc
  ! First Bead    C
  !cccccccccccccccc
  ! draw random position
  call draw_random_pos(Trial)
  chain_pos(:, 1) = Trial(:)
  ! get interaction energy of this choice
  call e_monomer(Ichain, 1, chain_pos, En, l_infinity)
  ! decide if this trial direction is open
  call Open_Dir(En, l_infinity, Lopen, P_Open)
  if(.not.Lopen) then
    Weight  = 0.0_dp
    Lovrlap = .true.
    node_count = 0
    return
  end if
  ! first monomer is always fixed
  Lfixed(1) = .true.
  ! first node has no parent
  Parent(1) = 0
  ! note probability
  Pmain(1)  = P_Open
  if(l_debug_grow_recoil) &
       write(unit=*, fmt='(a,l1,a,i6,a,i6,a,g10.3,a,l1)') &
       "DEBUG (SR grow_recoil): Lold=", Lold, ", Ichain=", Ichain, &
       ", i_mon=", 1, ", Pmain=", Pmain(1), ", Lopen=", Lopen
  ! we have not tried any new trial directions (childs)
  Nchild(1) = 0
  ! the first point (node) is the parent of the rest of the tree
  Ipoint    = 1
  Iparent   = 1
  ! go to next monomer
  i_mon     = 2
  do while(i_mon.le.n_mon)
    !ccccccccccccccccccccccccccccccccccccccccccc
    ! Check If We Have Used The Max. Number   C
    ! Of Trial Directions                     C
    !ccccccccccccccccccccccccccccccccccccccccccc
    if(Nchild(Iparent).eq.Nchoi(i_mon)) then
      ! recoil back by one, get the parent
      i_mon   = i_mon - 1
      Iparent = Parent(Iparent)
      ! if this monomer is already fixed, return
      if(Lfixed(i_mon)) then
        Weight  = 0.0_dp
        Lovrlap = .true.
        node_count = Ipoint
        return
      end if
    else ! if not all trial directions are exhausted
      !ccccccccccccccccccccccccccccc
      ! Generate A New Configuration   C
      !ccccccccccccccccccccccccccccc
      if(l_open_angle_eq_pi .or. (i_mon .eq. 2)) then
        call draw_2d_3d_bond(bond_len, Bond)
      else
        ! if we already have a bond we can use it to draw a vector which
        ! does not backfold too far
        call draw_2d_3d_bond(bond_len, Bond, &
             chain_pos(:, i_mon-2) - chain_pos(:, i_mon-1))
      end if
      ! define new trial position and decide if it is open
      chain_pos(:, i_mon) = chain_pos(:, i_mon-1) + Bond(:)
      call e_monomer(Ichain, i_mon, chain_pos, En, l_infinity)
      call Open_Dir(En, l_infinity, Lopen, P_Open)
      Ipoint = Ipoint + 1
      if(Ipoint.gt.Maxtree) then
        write(unit=*, fmt='(a)') &
             "ERROR (SR Grow_Recoil): Tree node count overflow"
        stop
      end if
      ! parent (Iparent) has another child (Ipoint), which in turn has
      ! no childs so far
      Nchild(Iparent) = Nchild(Iparent) + 1
      Parent(Ipoint)  = Iparent
      Nchild(Ipoint)  = 0
      if(Lopen) then
        ! we only assign P_Open to Pmain when we have found an
        ! open direction (otherwise the value would be overwritten later,
        ! because we use the last P_Open with Lopen == .TRUE. at i_mon)
        Pmain(i_mon) = P_Open
        if(l_debug_grow_recoil) &
             write(unit=*, fmt='(a,l1,a,i6,a,i6,a,g10.3,a,l1)') &
             "DEBUG (SR grow_recoil): Lold=", Lold, ", Ichain=", Ichain, &
             ", i_mon=", i_mon, ", Pmain=", Pmain(i_mon), ", Lopen=", Lopen
        ! go to next monomer, the present monomer will become parent
        i_mon   = i_mon + 1
        Iparent = Ipoint
        !ccccccccccccccccccccc
        ! Try To Fix A Bead    C
        !ccccccccccccccccccccc
        if(i_mon.le.n_mon) then
          ! if we are still in the chain,
          ! fix Ifix = i_mon - Nrecoil in the presently growing chain
          Ifix = i_mon - Nrecoil
          if(Ifix.gt.0) then
            Lfixed(Ifix) = .true.
          end if
        end if
      end if !(Lopen)
    end if !(Nchild(Iparent).Eq.Nchoi(i_mon))
  end do ! loop over all monomers
else ! Lold == .TRUE.
  !ccccccccccccccccccccc
  ! Old Configuration    C
  !ccccccccccccccccccccc
  ! initialize tree
  Parent(1) = 0
  Nchild(1) = 0
  Ipoint    = 1
  !cccccccccccccccc
  ! First Bead    C
  !cccccccccccccccc
  i_mon = 1
  call e_monomer(Ichain, i_mon, chain_pos, En, l_infinity)
  call Open_Dir(En, l_infinity, Lopen, P_Open)
  if(l_debug_grow_recoil) then
    ! this must not happen, as a chain with P_Open == 0 must not have
    ! been created in the first place.
    if(P_Open .eq. 0.0_dp) then
      write(unit=*, fmt='(a,l1,a,i6,a,i6,a,g10.3,a,l1)') &
           "ERROR (SR grow_recoil): Lold=", Lold, ", Ichain=", Ichain, &
           ", i_mon=", i_mon, ", Pmain=", Pmain(i_mon)
      call fluid_positions_out("grow_recoil_old_dump", i_time_mc)
      stop
    end if
  end if
  Pmain(i_mon)  = P_Open
  if(l_debug_grow_recoil) &
       write(unit=*, fmt='(a,l1,a,i6,a,i6,a,g10.3,a,l1)') &
       "DEBUG (SR grow_recoil): Lold=", Lold, ", Ichain=", Ichain, &
       ", i_mon=", i_mon, ", Pmain=", Pmain(i_mon)
  ! just run along the chain, record the P_Open values and the parent
  ! indices
  do i_mon = 2, n_mon
    call e_monomer(Ichain, i_mon, chain_pos, En, l_infinity)
    call Open_Dir(En, l_infinity, Lopen, P_Open)
    if(l_debug_grow_recoil) then
      if(P_Open .eq. 0.0_dp) then
        write(unit=*, fmt='(a,l1,a,i6,a,i6,a,g10.3,a,l1)') &
             "ERROR (SR grow_recoil): Lold=", Lold, ", Ichain=", Ichain, &
             ", i_mon=", i_mon, ", Pmain=", Pmain(i_mon)
        call fluid_positions_out("grow_recoil_old_dump", i_time_mc)
        stop
      end if
    end if
    Ipoint = Ipoint + 1
    if(Ipoint.gt.Maxtree) then
      write(unit=*, fmt='(a)') &
           "ERROR (SR Grow_Recoil): Tree node count overflow"
      stop
    end if
    Nchild(Ipoint)   = 0
    Parent(Ipoint)   = Ipoint - 1
    Nchild(Ipoint-1) = 1
    Pmain(i_mon)     = P_Open
    if(l_debug_grow_recoil) &
         write(unit=*, fmt='(a,l1,a,i6,a,i6,a,g10.3,a,l1)') &
         "DEBUG (SR grow_recoil): Lold=", Lold, ", Ichain=", Ichain, &
         ", i_mon=", i_mon, ", Pmain=", Pmain(i_mon)
  end do
end if!(Lold)
!ccccccccccccccccccccccccccccc
! What Is The Main Chain  ???   C
! Write Answer On Whmain        C
!ccccccccccccccccccccccccccccc
Ip    = Ipoint
i_mon = n_mon
! retrace the chain
do while(Ip.ne.0)
  Whmain(i_mon) = Ip
  Ip            = Parent(Ip)
  i_mon         = i_mon - 1
end do
! loop over the monomers in the chain
do Mybead = 2, n_mon
  Lready1      = .false.
  Lready2      = .false.
  ! we already found an open direction at Mybead, namely the one
  ! where the chain continues
  Ndir(Mybead) = 1
  ! make a copy of the coordinates
  chain_pos_copy(:, :) = chain_pos(:, :)
  !ccccccccccccccccccccccccccccc
  ! Search The Open Directions   C
  !ccccccccccccccccccccccccccccc
  ! This is the tricky part of the algorithm:
  ! Given a complete chain, look at every monomer and find out
  ! how many feelers of length Nrecoil can be grown there
  ! There are to exit flags:
  ! 1) Lready1: when we are back to where we started the search for
  !    open directions (thus we are done with the forward part of the
  !    tree)
  ! 2) Lready2: When we found a (just one!) feeler that has length
  !    Nrecoil
  do while(.not.Lready1)
    ! start at the current bead
    i_mon   = Mybead
    ! we maximally need to grow to Maxbead
    Maxbead = min(n_mon, i_mon -1 + Nrecoil)
    Iparent = Parent(Whmain(i_mon))
    Lready2 = .false.
    do while(.not.Lready2)
      if(Nchild(Iparent).eq.Nchoi(i_mon)) then
        ! recoil
        i_mon   = i_mon - 1
        Iparent = Parent(Iparent)
        ! if we went back behing the starting point Mybead, we are done
        if(i_mon.lt.Mybead) then
          Lready1 = .true.
          Lready2 = .true.
        end if
      else
        ! generate new trial direction
        if(l_open_angle_eq_pi .or. (i_mon .eq. 2)) then
          call draw_2d_3d_bond(bond_len, Bond)
        else
          call draw_2d_3d_bond(bond_len, Bond, &
               chain_pos_copy(:, i_mon-2) - chain_pos_copy(:, i_mon-1))
        end if
        chain_pos_copy(:, i_mon) = chain_pos_copy(:, i_mon-1) + Bond(:)
        call e_monomer(Ichain, i_mon, chain_pos_copy, En, l_infinity)
        call Open_Dir(En, l_infinity, Lopen, P_Open)
        Ipoint = Ipoint + 1
        if(Ipoint.gt.Maxtree) then
          write(unit=*, fmt='(a)') &
               "ERROR (SR Grow_Recoil): Tree node count overflow"
          stop
        end if
        Nchild(Iparent) = Nchild(Iparent) + 1
        Parent(Ipoint)  = Iparent
        Nchild(Ipoint)  = 0
        if(Lopen) then
          i_mon   = i_mon + 1
```

```
            Iparent = Ipoint
            !ccccccccccccccccccccc
            ! Try To Fix A Bead    C
            !ccccccccccccccccccccc
            if(i_mon.gt.Maxbead) then
              ! we have grown past Maxbead, so we found a free feeler.
              ! Set signal flag and increment counter for free feelers
              Lready2      = .true.
              Ndir(Mybead) = Ndir(Mybead) + 1
            end if
          end if
        end if
      end do ! loop over Lready2
    end do ! loop over Lready1
  end do ! loop over beads
  !ccccccccccccccccccc
  ! Weight Calculation C
  !ccccccccccccccccccc
  if(l_debug_grow_recoil) then
    do i_mon = 1, n_mon
      if(Pmain(i_mon) .eq. 0.0_dp) then
        write(unit=*, fmt='(a,l1,a,i6,a,i6,a)') &
             "ERROR (SR grow_recoil): Lold=", Lold, &
             ", Ichain=", Ichain, ", Pmain(", i_mon, ")= 0"
        stop
      end if
    end do
  end if
  Weight = 1.0_dp / Pmain(1)
  do i_mon = 2, n_mon
    Weight = Weight * real(Ndir(i_mon), kind=dp) / Pmain(i_mon)
  end do
  node_count = Ipoint
  if(l_debug_grow_recoil) print *, "DEBUG: Leaving SR grow_recoil"
  return
end subroutine Grow_Recoil
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! draws a random first position of a chain
subroutine draw_random_pos(pos)
  real(kind=dp), dimension(n_dim), intent(out) :: pos
  integer :: i_dim
  if(l_debug_draw_random_pos) print *, "DEBUG: Entering SR draw_random_pos"
  do i_dim = 1, n_dim_pbc
    pos(i_dim) = ran_uniform() * boundary(i_dim)
  end do
  ! if there are walls define postion perpendicular to walls
  if(n_dim .eq. n_dim_pbc+1) then
    if(l_2d_flat_setup) then
      ! for flat setup in the x-y plane set particle in the middle
      ! between the walls
      pos(n_dim) = 0.5_dp*boundary(n_dim)
    else
      ! boundary(n_dim) is initialized with the initial wall separation
      ! if this monomer is too close to the wall, the move is likely to be
      ! rejected.
      pos(n_dim) = (boundary(n_dim)-min_initial_wall_distance) &
           * ran_uniform() + min_initial_wall_distance
    end if
  end if
  if(l_debug_draw_random_pos) print *, "DEBUG: Leaving SR draw_random_pos"
end subroutine draw_random_pos
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! impose periodic boundary conditions in xy on particle with index i_part
subroutine impose_pbc(i_part, l_count_pbc)
  integer, intent(in) :: i_part
  logical, intent(in), optional :: l_count_pbc
  integer :: i_dim
  ! this is the periodic boundary folding used in SR predictor
  do i_dim = 1, n_dim_pbc
    if(r0(i_part,i_dim).gt.boundary(i_dim)) then
      r0(i_part,i_dim) = r0(i_part,i_dim) - boundary(i_dim)
      if(present(l_count_pbc) .and. l_count_pbc) &
           pbc_count(i_part,i_dim) = 1
    else if(r0(i_part,i_dim).lt.0.0_dp) then
      r0(i_part,i_dim) = r0(i_part,i_dim) + boundary(i_dim)
      if(present(l_count_pbc) .and. l_count_pbc) &
           pbc_count(i_part,i_dim) = -1
    end if
  end do
end subroutine impose_pbc
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! impose periodic boundary conditions in xy on position point(:)
subroutine pbc_fold(point)
  real(kind=dp), dimension(n_dim), intent(inout) :: point
  integer :: i_dim
  do i_dim = 1, n_dim_pbc
    if(point(i_dim).gt.boundary(i_dim)) then
      point(i_dim) = point(i_dim) - boundary(i_dim)
    else if(point(i_dim).lt.0.0_dp) then
      point(i_dim) = point(i_dim) + boundary(i_dim)
    end if
  end do
  if(l_pbc_fold_check) then
    do i_dim = 1, n_dim_pbc
      if(point(i_dim).gt.boundary(i_dim)) then
        write(unit=*, fmt='(a, i1, a, g13.6, a, g13.6, a)') &
             "ERROR (SR pbc_fold): point(", i_dim, ")=", point(i_dim), &
             " > boundary=",boundary(i_dim), " after folding. Box too small."
        stop
      else if(point(i_dim).lt.0.0_dp) then
        write(unit=*, fmt='(a, i1, a, g13.6, a)') &
             "ERROR (SR pbc_fold): point(", i_dim, ")=", point(i_dim), &
             " < 0.0 after folding.  Box too small."
        stop
      end if
    end do
  end if
end subroutine pbc_fold
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! checks if really all positions are folded back
subroutine check_pbc_fluid(l_part_index, u_part_index)
  integer, intent(in) :: l_part_index, u_part_index
  integer :: i_dim, i_part
  do i_dim = 1, n_dim_pbc
    do i_part = l_part_index, u_part_index
      if(r0(i_part,i_dim).gt.boundary(i_dim)) then
        write(unit=*, fmt='(a,i5,a,i2,a,e14.6,a,i1,a,g14.6)') &
             "ERROR (SR check_pbc_fluid): Fluid position r0(",i_part,",", &
             i_dim, ")=", r0(i_part,i_dim), " > boundary(", i_dim, ")=", &
             boundary(i_dim)
        stop
      else if(r0(i_part,i_dim).lt.0.0_dp) then
        write(unit=*, fmt='(a, i5, a, i2, a, g14.6, a)') &
             "ERROR (SR check_pbc_fluid): Fluid position r0(",i_part,",", &
             i_dim, ")=", r0(i_part,i_dim), " < 0.0"
        stop
      end if
    end do
  end do
end subroutine check_pbc_fluid
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! energy of chain i_chain
subroutine e_chain(i_chain, chain_positions, en_chain, l_calc_bond_en)
  integer, intent(in) :: i_chain
  real(kind=dp), dimension(n_dim, n_mon), intent(in) :: chain_positions
  real(kind=dp), intent(out) :: en_chain
  logical, intent(in), optional :: l_calc_bond_en
  logical :: l_infinity
  integer :: i_mon
  real(kind=dp) :: en, bond_en
  real(kind=dp), dimension(n_dim) :: bond_vec
  en_chain = 0.0_dp
  do i_mon = 1, n_mon
    call e_monomer(i_chain, i_mon, chain_positions, en, l_infinity)
    if(l_infinity) then
      write(unit=*, fmt='(a, i7, a, i7, a)') &
           "ERROR (SR e_chain): l_infinity for i_chain=", i_chain, &
           ", i_mon=", i_mon, " encountered"
      call fluid_positions_out("e_chain_infinite.dat", i_time_mc, &
           1, i_chain * n_mon)
      stop
    end if
    en_chain = en_chain + en
  end do
  if(present(l_calc_bond_en) .and. l_calc_bond_en .and. n_mon .gt. 1) then
    ! we assume that all bonds are identical in the chain
    !-----------------------------------------------
    bond_vec(:) = chain_positions(:, 1) - chain_positions(:, 2)
    ! coordinates are not folded, so we don't need to apply the
    ! minimum image convention
    call e_bond(bond_vec, bond_en)
    en_chain = en_chain + real(n_mon -1, kind=dp) * bond_en
    ! print *, "DEBUG (e_chain): bond_en=", bond_en
  end if
end subroutine e_chain
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! energy of old chain i_chain which is already binned
subroutine e_chain_old(i_chain, chain_positions, en_chain, l_calc_bond_en)
  integer, intent(in) :: i_chain
  real(kind=dp), dimension(n_dim, n_mon), intent(in) :: chain_positions
  real(kind=dp), intent(out) :: en_chain
  logical, intent(in), optional :: l_calc_bond_en
  logical :: l_infinity
  integer :: i_mon, i_part, l_part_excl_index
  real(kind=dp) :: potential_wall, potential_fluid, bond_en
  real(kind=dp), dimension(n_dim) :: bond_vec
  en_chain = 0.0_dp
  ! it is assumed that the coordinates chain_positions are folded
  !============================================================
  do i_mon = 1, n_mon
    i_part = (i_chain -1) * n_mon + i_mon
    if(l_fluid_wall_interaction) then
      ! interaction with the walls
      ! if a particle is "behind" the wall, l_infinity is set to .TRUE.
      ! meaning that the potential is infinite
      call pot_walls(chain_positions(:, i_mon), type(i_part), &
           potential_wall, l_infinity)
      if(l_debug_e_monomer) &
           print *, "DEBUG (SR e_chain_old): E_wall=", potential_wall, &
           ", l_infinity=", l_infinity
      if(l_infinity) then
        write(unit=*, fmt='(a, i7, a, i7, a)') &
             "ERROR (SR e_chain_old): l_infinity for i_chain=", i_chain, &
             ", i_mon=", i_mon, " encountered for fluid-wall interaction"
        call fluid_positions_out("e_chain_old_infinite.dat", i_time_mc, &
             1, i_chain * n_mon)
        stop
      end if
      en_chain = en_chain + potential_wall
    end if
    ! interaction with all chains which already exist (and are put in the
    ! fluid bins), in i_chain go only up to i_mon -2
    l_part_excl_index = max(i_part - 1, (i_chain - 1) * n_mon + 1)
    call pot_fluid(chain_positions(:, i_mon), type(i_part), &
         potential_fluid, l_infinity, l_part_excl_index, i_chain * n_mon)
    if(l_debug_e_monomer) &
         print *, "DEBUG (SR e_chain_old): E_fluid=", potential_fluid, &
         ", l_infinity=", l_infinity
    if(l_infinity) then
      write(unit=*, fmt='(a, i7, a, i7, a)') &
           "ERROR (SR e_chain_old): l_infinity for i_chain=", i_chain, &
           ", i_mon=", i_mon, " encountered for fluid-fluid interaction"
      call fluid_positions_out("e_chain_old_infinite.dat", i_time_mc, &
           1, i_chain * n_mon)
      stop
    end if
    en_chain = en_chain + potential_fluid
  end do ! loop over monomers
  if(present(l_calc_bond_en) .and. l_calc_bond_en .and. n_mon .gt. 1) then
    ! we assume that all bonds are identical in the chain
    !-----------------------------------------------
    bond_vec(:) = chain_positions(:, 1) - chain_positions(:, 2)
    ! coordinates are folded
    call apply_mic(bond_vec)
    call e_bond(bond_vec, bond_en)
    en_chain = en_chain + real(n_mon -1, kind=dp) * bond_en
    ! print *, "DEBUG (e_chain_old): bond_en=", bond_en
  end if
end subroutine e_chain_old
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! calculates the bond energy of a given bond-vector (LJ and FENE)
subroutine e_bond(bond_vec, bond_en)
  real(kind=dp), dimension(n_dim), intent(in) :: bond_vec
  real(kind=dp), intent(out) :: bond_en
  real(kind=dp) :: r_dummy, r_2
  ! the types of the fluid particles
  ! we assume homopolymers of type 1 (see SR conf_default)
  integer, parameter :: i_type = 1, j_type = 1
  r_2 = dot_product(bond_vec(:), bond_vec(:))
  ! calculate LJ potential
  bond_en = lj_pot(i_type, j_type, r_2)
  ! calculate FENE potential
  r_dummy = r_2/(r_chain_2 * sigma_2(i_type, j_type))
  if(r_dummy .ge. 1.0_dp) then
    ! the FENE-potential diverges at r_chain_2 * sigma_2(i_type, j_type).
```

```
    write(unit=*, fmt='(a, i2, a, i2, a, g13.6, a)') &
         "ERROR (SR e_bond): r_2/(r_chain_2*sigma_2(", i_type, &
         ",", j_type, ")=", r_dummy, ".ge. 1.0"
    write(unit=*, fmt='(a, g13.6, a, f7.4, a, f7.4, a)') &
         "        distance=", sqrt(r_2), ", bond-length=", &
         bond_len, " (FENE pot. diverges at", &
         sqrt(r_chain_2 * sigma_2(i_type, j_type)), ")"
    stop
  end if
  bond_en = bond_en - 0.5_dp * k_chain * r_chain_2 &
       * epsil(i_type, j_type) * log(1.0_dp - r_dummy)
end subroutine e_bond
!-----------------------------------------------------------------------------

!-----------------------------------------------------------------------------
! energy of monomer i_mon in chain Ichain
subroutine e_monomer(Ichain, i_mon, chain_positions, En, l_infinity)
  integer, intent(in) :: Ichain
  integer, intent(in) :: i_mon
  real(kind=dp), dimension(n_dim, n_mon), intent(in) :: chain_positions
  real(kind=dp), intent(out) :: En
  logical, intent(out) :: l_infinity
  ! the types of the fluid particles
  ! we assume homopolymers of type 1 (see SR conf_default)
  ! (for different types use i_type = type((i_chain -1) * n_mon + i_mon) )
  integer, parameter :: i_type = 1
  real(kind=dp) :: potential_wall, potential_fluid, &
       potential_intra_chain
  ! we make a copy of the position we are examining so that we can apply
  ! periodic boundary conditions on it.
  real(kind=dp), dimension(n_dim) :: point
  if(l_debug_e_monomer) print *, "DEBUG: Entering SR e_monomer, Ichain=", &
       Ichain, ", i_mon=", i_mon
  En = 0.0_dp
  potential_wall = 0.0_dp
  potential_fluid = 0.0_dp
  potential_intra_chain = 0.0_dp
  l_infinity = .FALSE.
  point(:) = chain_positions(:, i_mon)
  ! fold coordinates back
  call pbc_fold(point)
  ! interaction energy is calculated in several steps
  if(l_fluid_wall_interaction) then
    ! interaction with the walls
    ! if a particle is "behind" the wall, l_infinity is set to .TRUE.
    ! meaning that the potential is infinite
    call pot_walls(point, i_type, potential_wall, l_infinity)
    if(l_debug_e_monomer) &
         print *, "DEBUG (SR e_monomer): E_wall=", potential_wall, &
         ", l_infinity=", l_infinity
    if(l_infinity) return
  end if
  ! interaction with all chains which already exist (and are put in the
  ! fluid bins), except Ichain.  This means that for a single chain in the
  ! system we will get 0 (and we can skip this call to gain speed)
  if(n_chain .gt. 1) then
    call pot_fluid(point, i_type, potential_fluid, l_infinity, &
         (Ichain - 1) * n_mon +1, Ichain * n_mon)
  else
    if(l_debug_e_monomer) print *, "DEBUG (SR e_monomer): &
         &Skipped SR pot_fluid for single chain"
  end if
  if(l_debug_e_monomer) &
       print *, "DEBUG (SR e_monomer): E_fluid=", potential_fluid, &
       ", l_infinity=", l_infinity
  if(l_infinity) return
  ! interaction with all monomers in the chain Ichain, up to monomer index
  ! i_mon -2.  This is because the interaction with the neighboring monomer
  ! is always the same at fixed bond length and would cancel in the ratio
  ! of the weights anyways.  We use an extra subroutine here, because the
  ! new chain is not binned and dynamically bin the trial particles is too
  ! CPU-expensive (yes, I've tried that).
  if(i_mon .gt. 2) then
    call pot_intra_chain(i_mon, chain_positions, potential_intra_chain, &
         l_infinity)
    if(l_debug_e_monomer) &
         print *, "DEBUG (SR e_monomer): E_intra_chain=", &
         potential_intra_chain, ", l_infinity=", l_infinity
    if(l_infinity) return
  end if ! compute intra-chain interaction
  En = potential_wall + potential_fluid + potential_intra_chain
  if(l_debug_e_monomer) print *, "DEBUG: Leaving SR e_monomer"
  return
end subroutine e_monomer
!-----------------------------------------------------------------------------

!-----------------------------------------------------------------------------
! intra chain potential
subroutine pot_intra_chain(i_mon, chain_positions, potential, l_infinity)
  integer, intent(in) :: i_mon
  real(kind=dp), dimension(n_dim, n_mon), intent(in) :: chain_positions
  real(kind=dp), intent(out) :: potential
  logical, intent(out) :: l_infinity
  ! helpers
  integer :: i_mon_loop
  real(kind=dp) :: r_2, potential_check, &
       one_bond_away, two_bonds_away, three_bonds_away
  real(kind=dp), dimension(n_dim) :: delta_r
  ! we assume homopolymers of type 1 (see SR conf_default)
  integer, parameter :: i_type = 1, j_type = 1
  if(l_debug_pot_intra_chain) print *, "DEBUG: Entering SR pot_intra_chain"
  ! constants for deciding where the next interaction site can be
  one_bond_away    = (range_1(i_type, j_type) +          bond_len)**2
  two_bonds_away   = (range_1(i_type, j_type) + 2.0_dp * bond_len)**2
  three_bonds_away = (range_1(i_type, j_type) + 3.0_dp * bond_len)**2
  potential = 0.0_dp
  i_mon_loop = 1
  do while(i_mon_loop .le. i_mon -2)
    delta_r(:) = chain_positions(:, i_mon_loop) - chain_positions(:, i_mon)
    ! for a very stretched chain in a small box the chain ends could see
    ! each other (this is undesirable, though)
    call apply_mic(delta_r)
    r_2 = dot_product(delta_r(:), delta_r(:))
    ! check whether interaction takes place
    if(r_2.lt.range_2(i_type,j_type)) then
      ! to avoid a (numerical) division by zero problem, we
      ! test if r_2 is (numerically) zero (yes, this can happen!)
      if(r_2 .le. 0.0_dp) then
        l_infinity = .TRUE.
        potential = 0.0_dp
        if(l_debug_pot_intra_chain) &
             print *, "DEBUG (SR pot_intra_chain): &
             &Got l_infinity in intra-chain interaction between monomer", &
             i_mon, " and", i_mon_loop
        return
      end if
      if(l_debug_pot_intra_chain) &
           write(unit=*, fmt='(a,i6,a,i6,a,g10.3)') &
           "DEBUG (SR pot_intra_chain): interaction between monomer", &
           i_mon, " and", i_mon_loop, " lj_pot=", lj_pot(i_type, j_type, r_2)
      potential = potential + lj_pot(i_type, j_type, r_2)
      ! go to next monomer
      i_mon_loop = i_mon_loop +1
    else if(r_2 .lt. one_bond_away) then
      ! for the small values the if statements are probably faster than
      ! the calculation involving square roots
      i_mon_loop = i_mon_loop +1
    else if(r_2 .lt. two_bonds_away) then
      i_mon_loop = i_mon_loop +2
    else if(r_2 .lt. three_bonds_away) then
      i_mon_loop = i_mon_loop +3
    else
      ! here we compute how far away from i_mon_loop the next possible
      ! interaction site can be
      i_mon_loop = i_mon_loop &
           + int((sqrt(r_2) - range_1(i_type, j_type)) / bond_len) +1
    end if
  end do !loop over all monomers before i_mon -1
  ! we can check if the above method gives the same results as the simple
  ! loop over all i_mon_loop = 1 ... i_mon - 2.
  if(l_debug_pot_intra_chain) then
    potential_check = 0.0_dp
    do i_mon_loop = 1, i_mon - 2
      delta_r(:) = chain_positions(:, i_mon_loop) - chain_positions(:, i_mon)
      ! for a very stretched chain in a small box the chain ends could see
      ! each other (this is undesirable, though)
      call apply_mic(delta_r)
      r_2 = dot_product(delta_r(:), delta_r(:))
      ! check whether interaction takes place
      if(r_2.lt.range_2(i_type,j_type)) then
        ! to avoid a (numerical) division by zero problem, we
        ! test if r_2 is (numerically) zero (yes, this can happen!)
        if(r_2 .le. 0.0_dp) then
          l_infinity = .TRUE.
          potential_check = 0.0_dp
          if(l_debug_pot_intra_chain) &
               print *, "DEBUG (SR pot_intra_chain): &
               &Got l_infinity in intra-chain interaction between monomer", &
               i_mon, " and", i_mon_loop
          return
        end if
        if(l_debug_pot_intra_chain) write(unit=*, fmt='(a,i6,a,i6,a,g10.3)')&
             "DEBUG (SR pot_intra_chain): In intra-chain&
             & interaction between monomer", i_mon, " and", i_mon_loop, &
             " lj_pot=", lj_pot(i_type, j_type, r_2)
        potential_check = potential_check + lj_pot(i_type, j_type, r_2)
      end if
    end do
    if(abs(potential - potential_check) &
         .gt. (10.0_dp)**(-precision(potential)+1) * potential) then
      write(unit=*, fmt='(2(a, g13.6))') "ERROR (SR pot_intra_chain): &
           &potential=", potential, ", potential_check=", potential_check
      stop
    end if
  end if
  if(l_debug_pot_intra_chain) print *, "DEBUG: Leaving SR pot_intra_chain"
end subroutine pot_intra_chain
!-----------------------------------------------------------------------------

!-----------------------------------------------------------------------------
! decides is an direction is open
subroutine Open_Dir(etot, l_infinity, l_open, p_open)
  !ccccccccccccccccccccccccccccccccccccccccccccccccc
  !     decide if a bead is either open or closed   c
  !     this is quite general; in principle you     c
  !     are allowed to design your own criteria !   c
  !ccccccccccccccccccccccccccccccccccccccccccccccccc
  real(kind=dp), intent(in)  :: etot
  logical, intent(in)        :: l_infinity
  logical, intent(out)       :: l_open
  real(kind=dp), intent(out) :: p_open
  ! we have to initialize l_open
  l_open = .false.
  ! if we encountered an infinite energy, this direction is surely closed
  if(l_infinity) then
    p_open = 0.0_dp
    return
  end if
  ! for high energies we can get a numerical 0 for p_open.  But the random
  ! numbers are striclty greater than 0, so we have to initialize
  ! l_open = .false. to get l_open = .false. in this case.
  p_open = min(1.0_dp, exp(-temp_inv * etot))
  if(ran_uniform() .lt. p_open) l_open = .true.
  if(l_debug_open_dir) then
    if(p_open .eq. 0.0_dp .and. l_open) then
      print *, "ERROR (SR Open_Dir): p_open .eq. 0.0_dp .and. l_open"
      stop
    end if
  end if
  return
end subroutine Open_Dir
!-----------------------------------------------------------------------------

!-----------------------------------------------------------------------------
real(kind=dp) function ran_uniform()
  ! hold some random numbers available, so that function calls can be avoided
  integer, parameter :: rn_buff_size = 256
  real(kind=rngRealKind), dimension(rn_buff_size), save :: rn_buff
  integer, save :: used = rn_buff_size
  if(l_debug_ran_uniform) print *, "DEBUG: Entering FCT ran_uniform"
  if(used .lt. rn_buff_size) then
    used = used +1
    ran_uniform = real(rn_buff(used), kind=dp)
    return
  else
    ! fill the buffer again
    call ranlux(rn_buff, rn_buff_size)
  if(l_debug_ran_uniform) &
       print *, "DEBUG (FCT ran_uniform): Filled random number buffer"
    used = 1
    ran_uniform = real(rn_buff(used), kind=dp)
    return
  end if
end function ran_uniform
!-----------------------------------------------------------------------------

!-----------------------------------------------------------------------------
! draw a bond in the approbriate dimensions.  The case open_angle .eq. pi
! is dealt with before.
subroutine draw_2d_3d_bond(bond_len, bond, direction)
  real(kind=dp), intent(in) :: bond_len
  real(kind=dp), dimension(n_dim), intent(out) :: bond
  real(kind=dp), dimension(n_dim), intent(in), optional :: direction
  select case(n_dim)
  case(3)
    if(n_dim_pbc .eq. 3) then
      if(present(direction)) then
```

```
        call draw_unit_sphere_vec(bond, direction)
      else
        call draw_unit_sphere_vec(bond)
      end if
    else if(n_dim_pbc .eq. 2) then
      if(.not.l_2d_flat_setup) then
        if(present(direction)) then
          call draw_unit_sphere_vec(bond, direction)
        else
          call draw_unit_sphere_vec(bond)
        end if
      else
        if(present(direction)) then
          call draw_unit_circle_vec(bond, direction)
        else
          call draw_unit_circle_vec(bond)
        end if
      end if
    else
      write(unit=*, fmt='(a,i1,a,i1,a)') "ERROR (SR draw_2d_3d_bond): &
           &n_dim=", n_dim, " and n_dim_pbc=", n_dim_pbc, " not implemented"
      stop
    end if
  case(2)
    if(n_dim_pbc .eq. 2) then
      if(present(direction)) then
        call draw_unit_circle_vec(bond, direction)
      else
        call draw_unit_circle_vec(bond)
      end if
    else if(n_dim_pbc .eq. 1) then
      if(.not.l_2d_flat_setup) then
        if(present(direction)) then
          call draw_unit_circle_vec(bond, direction)
        else
          call draw_unit_circle_vec(bond)
        end if
      else
        ! this is a rather simple bond
        if(ran_uniform() .lt. 0.5_dp) then
          bond(1) = 1.0_dp
        else
          bond(1) = -1.0_dp
        end if
      end if
    else
      write(unit=*, fmt='(a,i1,a,i1,a)') "ERROR (SR draw_2d_3d_bond): &
           &n_dim=", n_dim, " and n_dim_pbc=", n_dim_pbc, " not implemented"
      stop
    end if
  case(1)
    if(n_dim_pbc .eq. 1) then
      if(ran_uniform() .lt. 0.5_dp) then
        bond(1) = 1.0_dp
      else
        bond(1) = -1.0_dp
      end if
    else
      write(unit=*, fmt='(a,i1,a,i1,a)') "ERROR (SR draw_2d_3d_bond): &
           &n_dim=", n_dim, " and n_dim_pbc=", n_dim_pbc, " not implemented"
      stop
    end if
  end select
  ! rescale with bond-length
  bond(:) = bond_len * bond(:)
  return
end subroutine draw_2d_3d_bond
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! draws a vector on the unit circle with opening angle open_angle if an
! additional direction vector is specified.
subroutine draw_unit_circle_vec(vec, direction)
  real(kind=dp), dimension(n_dim), intent(out)          :: vec
  real(kind=dp), dimension(n_dim), intent(in), optional :: direction
  real(kind=dp) :: phi
  if(present(direction)) then
    ! we draw the angle phi and compute the components
    ! then the angle from the direction vector is added
    phi = 2.0_dp * open_angle * (ran_uniform() - 0.5_dp)
  else
    ! if there's no direction, we draw any angle
    phi = twopi * (ran_uniform() - 0.5_dp)
  end if
  ! we have at least two dimensions, otherwise we couldn't draw a circle
  if(present(direction)) phi = phi + atan(direction(2)/direction(1))
  vec(1)     = cos(phi)
  vec(2)     = sin(phi)
  ! if we do a flat setup in a 3d system, the 3rd coordinate is set to 0
  if(n_dim .eq. 3 .and. n_dim_pbc .eq. 2) vec(n_dim) = 0.0_dp
  return
end subroutine draw_unit_circle_vec
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! draws a vector on the unit sphere with opening angle open_angle if an
! additional direction vector (3-d) is specified.
subroutine draw_unit_sphere_vec(vec, direction)
  real(kind=dp), dimension(n_dim), intent(out)          :: vec
  real(kind=dp), dimension(n_dim), intent(in), optional :: direction
  real(kind=dp) :: cos_phi, sin_phi, z_component, angle
  integer, parameter :: n_max_iter = 25
  integer :: iter
  do iter = 1, n_max_iter
    ! we draw the sines and cosines instead of drawing the angles and
    ! computing them, because trigonometric operations are relatively
    ! expensive
    cos_phi = 2.0_dp * ran_uniform() - 1.0_dp
    sin_phi = 2.0_dp * ran_uniform() - 1.0_dp
    z_component = cos_phi**2 + sin_phi**2
    ! z_component must be in (0, 1) (open interval), that is, must be inside
    ! the unit circle (without 0).  Just rescaling cos_phi and sin_phi
    ! would overly prefer the corners of the circle.
    do while (z_component .ge. 1.0_dp .or. z_component .eq. 0.0_dp)
      cos_phi = 2.0_dp * ran_uniform() - 1.0_dp
      sin_phi = 2.0_dp * ran_uniform() - 1.0_dp
      z_component = cos_phi**2 + sin_phi**2
    end do
    ! z_component is distributed between 0 and 1, so &
    ! 1.0_dp - 2.0_dp * z_component
    ! is distributed between -1 and 1.  Because cos_phi and sin_phi were
    ! evenly distributed and z_component was defined as the sums of the
    ! squares,  z_component will have the distribution of the z-component
    ! of points on the unit sphere
    vec(n_dim)= 1.0_dp - 2.0_dp * z_component
    ! we have to rescale cos_phi and sin_phi to have a length of 1
    ! just write down the sum of the squares of the components to verify
    z_component = 2.0_dp * sqrt(1.0_dp - z_component)
    vec(1) = z_component * cos_phi
    vec(2) = z_component * sin_phi
    if(present(direction)) then
      angle = acos(dot_product(direction(:), vec(:)) &
           / sqrt(dot_product(direction(:), direction(:))))
      ! if the angle between direction and vec is in range we are done
      if(angle .le. open_angle) return
    else
      return
    end if
  end do
  ! after n_max_iter iterations we just take the last vector we found
  write(unit=*, fmt='(a)') "WARNING (SR draw_unit_sphere_vec): &
       &Took last vector, as n_max_iter exceeded."
end subroutine draw_unit_sphere_vec
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! sets up an initial configuration using conventional configurational bias
! monte carlo.  Does not equilibrate the configuration.
subroutine mc_fluid_setup_cbmc
  integer :: i_chain, i_part, i_dim
  ! accounting of chains
  logical :: l_new_chain_inserted
  integer :: call_chain_insert_counter
  ! maximal number of calls to CBMC insertion of chain before giving up
  integer, parameter :: n_max_insert_calls = 10000
  do i_chain = 1, n_chain
    call_chain_insert_counter = 0
    ! use configurational bias monte carlo for chain setup
    ! (works also for simple liquid)
    do
      call_chain_insert_counter = call_chain_insert_counter +1
      ! try to insert a chain until we successfully inserted one
      call insert_chain(type(1 + (i_chain-1)*n_mon : i_chain*n_mon), &
           (i_chain-1) * n_mon, l_new_chain_inserted)
      if(l_new_chain_inserted) then
        write(unit=*, fmt='(a, i6, a, i6)') &
             "MESSAGE (SR mc_fluid_setup_cbmc): needed", &
             call_chain_insert_counter, " calls for inserting chain", &
             i_chain
        exit
      end if
      if(call_chain_insert_counter .gt. n_max_insert_calls) then
        write(unit=*, fmt='(a, i6)') &
             "ERROR (SR mc_fluid_setup_cbmc): call_chain_insert_counter &
             &.gt. n_max_insert_calls=", n_max_insert_calls
        stop
      end if
    end do
    ! apply periodic boundary conditions if necessary for the whole chain.
    ! If we applied the pbc during creation of the chain, and particle i
    ! gets folded in the simulation box, particle i+1 would not be folded.
    ! This seems a bit artificial, although the simulation runs identically
    ! with either method, as we are working with equivalence classes of
    ! particles anyways.
    do i_part = 1 + (i_chain-1)*n_mon, i_chain*n_mon
      do i_dim = 1, n_dim_pbc
        if(r0(i_part,i_dim).gt.boundary(i_dim)) then
          r0(i_part,i_dim) = r0(i_part,i_dim) - boundary(i_dim)
          pbc_count(i_part,i_dim) = pbc_count(i_part,i_dim) + 1
        else if(r0(i_part,i_dim).lt.0.0_dp) then
          r0(i_part,i_dim) = r0(i_part,i_dim) + boundary(i_dim)
          pbc_count(i_part,i_dim) = pbc_count(i_part,i_dim) - 1
        end if
      end do
    end do
  end do ! i_chain = 1, n_chain
  ! now that we have a configuration it is not necessary anymore to ramp
  ! the effective potential
  r_2_min_init = 0.0_dp
  r_2_min      = 0.0_dp
  r_2_min_time = 0.0_dp
  l_r_2_min_finite = .FALSE.
end subroutine mc_fluid_setup_cbmc
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! create a chain using cbmc and tries to insert it
subroutine insert_chain(monomer_types, n_particles_before, &
     l_new_chain_inserted)
  integer, dimension(n_mon), intent(in) :: monomer_types
  integer, intent(in) :: n_particles_before
  logical, intent(out) :: l_new_chain_inserted
  real(kind=dp), dimension(n_dim, n_mon)   :: chain_positions
  real(kind=dp), dimension(n_dim, n_mon)   :: chain_positions_full
  ! arrays holding random numbers
  ! if one knows the number of random numbers needed, it is more efficient
  ! to draw them at once thus saving calls.
  real(kind=rngRealKind), dimension(n_mon) :: random_mon
  ! Boltzman weights of trial vectors for fixed monomer in chain
  real(kind=dp), dimension(n_trial_vectors) :: weight
  ! sum of the weigths for each monomer
  real(kind=dp), dimension(n_mon) :: cumulated_weights
  ! trial vectors
  real(kind=dp), dimension(n_dim, n_trial_vectors) :: trial_vectors
  real(kind=dp), dimension(n_dim) :: point
  ! which trial vectors were taken
  integer, dimension(n_mon) :: chosen_trial_vectors
  ! counter
  integer :: counter_first_position_draws, counter_trial_vector_draws, &
       counter_chain_build_tries
  ! length of the trial vectors
  real(kind=dp) :: step_length
  ! potential energies
  real(kind=dp) :: potential_wall, potential_fluid
  ! flag if potential is "infinite".  This can happen for the wall and also
  ! for the fluid interaction in case of backfolding.
  logical :: l_infinity
  ! monomer positions in chain
  integer :: i_mon_pos, i_mon_pos_2
  ! maximal length of chainreached so far
  integer :: max_reached_chain_length
  ! sum of reached chain-length for getting average
  real(kind=dp) :: sum_reached_chain_lengths
  ! trial vector index
  integer :: i_trial
  ! counter for chain bond crossings in x-y plane
  integer :: n_xy_bond_crossings
  ! weights
  real(kind=dp) :: select_cum_weight, cumulated
  ! Rosenbluth factors of chains
  real(kind=dp) :: rosenbluth_factor_new
  real(kind=dp) :: rosenbluth_factor_old
  real(kind=dp) :: rosenbluth_rescale
  ! energy of a bond
  real(kind=dp) :: bond_energy
  ! total potential of the chains.
  real(kind=dp), save :: total_potential_chains = 0.0_dp
  real(kind=dp), dimension(n_trial_vectors, n_mon) :: potential_mon
  integer :: n_chains_before
  !============================================================================
  if(l_debug_insert_chain) print *, "DEBUG: Entering SR insert_chain"
  ! initialize
  temp_inv = 1.0_dp / temp
  n_chains_before = n_particles_before / n_mon
  if(l_debug_insert_chain) write(unit=*, fmt=*) &
```

```
      "DEBUG: n_chains_before = ", n_chains_before
! reset counter
counter_chain_build_tries = 0
max_reached_chain_length = 0
sum_reached_chain_lengths = 0.0_dp
! We only need the bond-energy contribution from the LJ potential.  The
! FENE part will cancel in the fraction
! rosenbluth_factor_new / rosenbluth_factor_old, because the bond length is
! fixed.  If the bond-length is not fixed, then the mean bond-length will
! be very close to the preferred bond length and the deviations will be
! small.  We assume that all fluid particles are of type 1.
! (For a copolymer the algorithm has to be changed anyways, e.g. the
! treatment of the first position has to be revised).
bond_energy = lj_pot(1, 1, (scale_bond_length * bond_length(1, 1))**2)
if((n_mon .gt. 1) .and. (scale_bond_length .ne. 1.0_dp)) then
  write(unit=*, fmt='(a, f7.3)') &
       "CAUTION (SR insert_chain): scale_bond_length=", &
       scale_bond_length, " not unity"
end if
! the Rosenbluth factor of an ideal chain, which does not interact with
! a wall and where non-bonded neighbor interactions don't happen
! Think: linear chain.
! This is the most energetically favourable configuration possible.
!rosenbluth_factor_old = real(n_trial_vectors, kind=dp) * &
!     (real(n_trial_vectors,kind=dp)*exp(-temp_inv*bond_energy))**(n_mon-1)
! with this rosenbluth_factor_old we exert that there is at least one
! "free" bond.
! The problem is that if n_trial_vectors is large, even configurations with
! a very high energy are likely to be accepted
! rosenbluth_factor_old = real(n_trial_vectors, kind=dp) * &
!      exp(-temp_inv * bond_energy)**(n_mon-1)
! here we assume a "good" chain to have only a fraction of "free" bonds
! For example only an oktant of a sphere.
if(n_free_trial_vectors .le. 0) then
  write(unit=*, fmt='(a, i4, a)') &
       "ERROR (SR insert_chain): n_free_trial_vectors=", &
       n_free_trial_vectors, " .le. 0"
  stop
end if
! rosenbluth_factor_old = real(n_trial_vectors, kind=dp) * &
!       (real(n_free_trial_vectors, kind=dp) &
!       * exp(-temp_inv * bond_energy))**(n_mon-1)
! rosenbluth_rescale is used to rescale the entries in cumulated_weights(:)
! below to get closer to unity to avoid numerical underflow.  It is a
! typical value for the cumulated_weights (hopefully ;))
rosenbluth_rescale = &
     exp(temp_inv * bond_energy)/real(n_free_trial_vectors, kind=dp)
! rosenbluth_factor_old above is rescaled by rosenbluth_rescale**N
rosenbluth_factor_old = real(n_trial_vectors, kind=dp) * rosenbluth_rescale
if(n_mon .gt. 1 .and. l_insert_chain_verb) &
     write(unit=*, fmt='(a, g13.6, a, g13.6)') &
     "MESSAGE (SR insert_chain): rosenbluth_rescale=", rosenbluth_rescale,& 
     ", rosenbluth_factor_old=", rosenbluth_factor_old
! rosenbluth_factor_old = real(n_trial_vectors, kind=dp) * &
!     * exp(log(real(n_free_trial_vectors, kind=dp) &
!     - temp_inv * bond_energy))**(n_mon-1)
! due to finite range of the reals in the computer the Rosenbluth-factor
! can get 0
if(rosenbluth_factor_old .le. 0.0_dp) then
  write(unit=*, fmt=*) &
       "ERROR (SR insert_chain): rosenbluth_factor_old=", &
       rosenbluth_factor_old
  write(unit=*, fmt=*) &
       "      real(n_free_trial_vectors, kind=dp)=", &
       real(n_free_trial_vectors, kind=dp)
  write(unit=*, fmt=*) &
       "      exp(-temp_inv * bond_energy))=", &
       exp(-temp_inv * bond_energy)
  stop
end if
! this is a loop over tries to build a complete chain
do
  counter_chain_build_tries = counter_chain_build_tries +1
  ! initialize
  weight(:) = 0.0_dp
  potential_mon(:, :) = 0.0_dp
  chosen_trial_vectors(:) = 0
  ! specify position of first monomer in chain
  i_mon_pos = 1
  counter_first_position_draws = 0
  ! bin the particles we have so far
  call bin_fluid_particles(1, n_particles_before)
  ! create first monomer at random position.
  ! one step is always self avoiding.
  !=========================================
  ! it can happen that the position of the first monomer is such that its
  ! potential is very high, leading to a numerical 0 in the weight.  In
  ! this case we try another position for the first monomer.
  do
    call draw_random_pos(chain_positions(:, i_mon_pos))
    ! write new particle position to r0(:, :)
    r0(n_particles_before + i_mon_pos, :) = chain_positions(:, i_mon_pos)
    ! if a particle is "behind" the wall, l_infinity is set to .TRUE.
    ! meaning that the potential is infinite
    call pot_walls(chain_positions(:, i_mon_pos), &
         monomer_types(i_mon_pos), potential_wall, l_infinity)
    if(.not. l_infinity) then
      call pot_fluid(chain_positions(:,i_mon_pos), &
           monomer_types(i_mon_pos), potential_fluid, l_infinity)
      if(.not.l_infinity) then
        weight(1) = exp(-temp_inv * (potential_wall + potential_fluid))
      else
        weight(1) = 0.0_dp
      end if
    else
      weight(1) = 0.0_dp
    end if
    if(weight(1) .gt. 0.0_dp) then
      ! we have found a position with finite energy
      exit
    else
      if(l_debug_insert_chain) then
        print *, "DEBUG: weight(1) = 0.0_dp, new first position"
      end if
      counter_first_position_draws = counter_first_position_draws +1
      if(counter_first_position_draws .gt. n_max_first_position_draws) then
        write(unit=*, fmt='(a, i7, a)') &
             "ERROR (SR insert_chain): weight(1) = 0.0_dp for the last", &
             n_max_first_position_draws, " first positions."
        write(unit=*, fmt='(a, i8, a)') &
             "      (number of fluid particles already in system &
             &n_particles_before=", n_particles_before, ")"
        stop
      end if
    end if ! (weight(1). gt. 0.0_dp)
  end do ! end loop over first position draws
  ! redefine weight (follow the notation in Frenkel/Smit with the prefactor
  ! n_trial_vectors)
  chosen_trial_vectors(i_mon_pos) = 1
  potential_mon(chosen_trial_vectors(i_mon_pos), i_mon_pos) &
       = potential_wall + potential_fluid
  weight(1) = real(n_trial_vectors, kind=dp) * weight(1)
  cumulated_weights(i_mon_pos) = weight(1)
  ! update fluid bins with the new particle.
  ! periodic boundaries could not be crossed (because the first position is
  ! always chosen within the boundaries)
  call bin_fluid_particles(n_particles_before + i_mon_pos)
  ! generate random walk for next monomers on chain
  call ranlux(random_mon, n_mon)
  do i_mon_pos = 2, n_mon
    counter_trial_vector_draws = 0
    if(l_debug_insert_chain) then
      if(sum(bin_fluid(:, :, :, 0)) &
           .ne. n_particles_before + i_mon_pos -1) then
        print *, "DEBUG: sum particles in bins:", &
             sum(bin_fluid(:, :, :, 0)), ", should be:", &
             n_particles_before + i_mon_pos -1
        stop
      end if
    end if
    ! it can happen that all the weights for this step become 0,
    ! in this case we retry
    do
      ! get step length.  Either the preferred bond length or a length
      ! drawn according to its Boltzmann weight
      step_length = choose_length(type(monomer_types(i_mon_pos-1)), &
           type(monomer_types(i_mon_pos)))
      ! calculate potentials and weights for each trial vector
      do i_trial = 1, n_trial_vectors
        ! get step length.  Either the preferred bond length or a length
        ! drawn according to its Boltzmann weight
        step_length = choose_length(type(monomer_types(i_mon_pos-1)), &
             type(monomer_types(i_mon_pos)))
        call draw_2d_3d_bond(step_length, trial_vectors(:, i_trial))
        ! get interaction potential
        !--------------------------
        ! add position of previous monomer to get the position of the
        ! trial monomer
        point(:) = r0(n_particles_before + i_mon_pos -1, :) &
             + trial_vectors(:, i_trial)
        ! unless the length of the trial vectors gets bigger than the
        ! binning box width, there should be no difference, because of
        ! the int() cast in finding the bin-indices.  However, when
        ! bonds can be long, this can happen and we have to fold the
        ! coordinates.
        call pbc_fold(point)
        call pot_walls(point, monomer_types(i_mon_pos), &
             potential_wall, l_infinity)
        ! if a particle is "behind" the wall, l_infinity is set to .TRUE.
        ! meaning that the potential is infinite
        if(.not. l_infinity) then
          call pot_fluid(point, monomer_types(i_mon_pos), &
               potential_fluid, l_infinity)
          ! if two fluid particles sit too close then l_infinity == .TRUE.
          if(.not. l_infinity) then
            weight(i_trial) = exp(-temp_inv*(potential_wall &
                 + potential_fluid))
            potential_mon(i_trial,i_mon_pos) = &
                 potential_wall + potential_fluid
            if(l_debug_insert_chain) then
              ! check if there is no interaction
              ! (must not happen, as particles are bonded!)
              if(potential_fluid .eq. 0.0_dp) then
                print *, "DEBUG: i_mon_pos=", i_mon_pos, ", i_trial=", &
                     i_trial, ", potential_fluid=",  potential_fluid, &
                     " (bond_energy=", bond_energy, ")"
                print *, "       part index", &
                     n_particles_before+i_mon_pos-1, " is in bin:"
                call get_bin_indices(r0(n_particles_before+i_mon_pos -1,:))
                print *, "r0 center=", r0(n_particles_before+i_mon_pos-1,:)
                print *, "trial vector=", trial_vectors(:, i_trial)
                call pbc_fold(point)
                print *, "point=", point, " is in bin"
                call get_bin_indices(r0(n_particles_before+i_mon_pos -1,:))
                stop
              end if
            end if
          else
            weight(i_trial) = 0.0_dp
          end if
        else
          weight(i_trial) = 0.0_dp
        end if
      end do
      cumulated_weights(i_mon_pos) = sum(weight(:))
      if(cumulated_weights(i_mon_pos) .gt. 0.0_dp) then
        exit
      else
        if(l_debug_insert_chain) then
          write(unit=*, fmt='(a, i7, a)') &
               "DEBUG: cumulated_weights(", i_mon_pos, ") = 0.0_dp"
        end if
        counter_trial_vector_draws = counter_trial_vector_draws +1
        if(counter_trial_vector_draws .gt. n_max_trial_vector_draws) then
          max_reached_chain_length = &
               max(max_reached_chain_length, i_mon_pos-1)
          sum_reached_chain_lengths = sum_reached_chain_lengths &
               + real(i_mon_pos-1, kind=dp)
          write(unit=*, fmt='(a, i7, a, i4, a)') &
               "MESSAGE (SR insert_chain): cumulated_weights(", &
               i_mon_pos, ") = 0.0 for last", &
               n_max_trial_vector_draws, " trial sets"
          !write(unit=*, fmt='(a, i8, a, 3g13.6)') &
          !     "        center at: r0(", n_particles_before+i_mon_pos-1, &
          !     ", :)=", r0(n_particles_before + i_mon_pos -1, :)
          write(unit=*, fmt='(a)') &
               "        Retrying with new startpoint..."
          goto 100
        end if
      end if
    end do ! draws of trial vectors
    ! choose a trial vector according to its Boltzmann weight
    !========================================================
    ! draw a random number, scale it with the cumulated_weights
    select_cum_weight = cumulated_weights(i_mon_pos) &
         * real(random_mon(i_mon_pos -1), kind=dp)
    ! find the index where select_cum_weight reaches the cumulated_weight
    !--------------------------------------------------------------------
    ! there is the possibility that cumulated_weights(i_mon_pos) is just
    ! the minimal number that can be described by a real.  Multiplying this
    ! number with random in (0, 1) can leave the result unchanged (!), so
    ! select_cum_weight = cumulated_weights(i_mon_pos).
    ! So, the selection has to be done with .LE. to catch the case
    ! select_cum_weight = cumulated_weights(i_mon_pos).
    ! Besides, cumulated_weights(i_mon_pos) is always .GT. 0, so every
    ! selection uses a half open interval.
    cumulated = 0.0_dp
    do i_trial = 1, n_trial_vectors
      cumulated = cumulated + weight(i_trial)
      if(select_cum_weight .le. cumulated) then
```

```
          chosen_trial_vectors(i_mon_pos) = i_trial
          exit
        end if
      end do
      if(chosen_trial_vectors(i_mon_pos) .eq. 0) then
        write(unit=*, fmt='(a)') &
             "ERROR (SR insert_chain): Fallthrough when searching for trial &
             &vector"
        write(unit=*, fmt=*) &
             "      weight(:)=", weight(:)
        write(unit=*, fmt=*) &
             "      cumulated_weights=", cumulated_weights(i_mon_pos)
        write(unit=*, fmt=*) &
             "      random number=    ", &
             real(random_mon(i_mon_pos -1), kind=dp)
        write(unit=*, fmt=*) &
             "      select_cum_weight=", select_cum_weight
        write(unit=*, fmt=*) &
             "      cumulated=        ", cumulated
        stop
      end if
      ! DEBUG
      if(l_debug_insert_chain) then
        write(unit=*, fmt=*) &
             "DEBUG: for i_mon_pos=", i_mon_pos, " found weights:"
        write(unit=*, fmt=*) weight(:)
        write(unit=*, fmt=*) &
             "      cumulated_weights=", cumulated_weights(i_mon_pos)
        write(unit=*, fmt=*) &
             "      select_cum_weight=", select_cum_weight
        write(unit=*, fmt=*) &
             "      selected i_trial=", chosen_trial_vectors(i_mon_pos)
        write(unit=*, fmt=*) &
             "      sum(weight(1:i_trial-1)=", &
             sum(weight(1:chosen_trial_vectors(i_mon_pos)-1))
        write(unit=*, fmt=*) &
             "      sum(weight(1:i_trial)=", &
             sum(weight(1:chosen_trial_vectors(i_mon_pos)))
      end if
      ! define new position
      chain_positions(:, i_mon_pos) = &
           trial_vectors(:, chosen_trial_vectors(i_mon_pos))
      ! write new particle position to r0(:, :)
      r0(n_particles_before + i_mon_pos, :) = &
           r0(n_particles_before+i_mon_pos-1,:)+chain_positions(:,i_mon_pos)
      ! apply periodic boundary conditions in x, y
      call impose_pbc(n_particles_before + i_mon_pos)
      ! update fluid bins with the new particle
      call bin_fluid_particles(n_particles_before + i_mon_pos)
    end do ! loop over all monomers
    ! build up full chain positions (not folded)
    i_mon_pos_2 = 1
    chain_positions_full(:, i_mon_pos_2) &
         = chain_positions(:, i_mon_pos_2)
    do i_mon_pos_2 = 2, n_mon
      chain_positions_full(:, i_mon_pos_2)= &
           chain_positions_full(:, i_mon_pos_2 -1) &
           + chain_positions(:, i_mon_pos_2)
    end do
    ! check if chain is self crossing in x-y plane
    if(l_forbid_xy_bond_crossings .and. (n_mon .ge. 2)) then
      if(l_debug_resolve_bond_crossings) &
           call write_chain_in_xy_plane(chain_positions_full, &
           "chain_pos_start.dat")
      if(l_resolve_bond_crossings) then
        ! resolve bond crossings
        call check_xy_bond_crossings_self(chain_positions_full, &
             n_xy_bond_crossings, l_resolve_bond_crossings)
        write(unit=*, fmt='(a, i7, a)') &
             "MESSAGE (SR insert_chain): Detected and resolved", &
             n_xy_bond_crossings, " self xy-bond crossings in chain"
        if(l_debug_resolve_bond_crossings) &
             call write_chain_in_xy_plane(chain_positions_full, &
             "chain_pos_swapped.dat")
      end if
      ! check if there are no bond crossigns anymore
      call check_xy_bond_crossings_self(chain_positions_full, &
           n_xy_bond_crossings)
      if(n_xy_bond_crossings .gt. 0) then
        write(unit=*, fmt='(a, i7, a)') &
             "MESSAGE (SR insert_chain): Detected", n_xy_bond_crossings, &
             " self xy-bond crossings in chain"
        write(unit=*, fmt='(a)') &
             "        Retrying with new startpoint..."
        if(l_resolve_bond_crossings) then
          write(unit=*, fmt='(a, i7, a)') &
               "ERROR (SR insert_chain): Detected", n_xy_bond_crossings, &
               " self xy-bond crossings in chain *after* resolving them"
          ! dump chain projection on xy-plane
          call write_chain_in_xy_plane(chain_positions_full, &
               "chain_pos_dump.dat")
          stop
        end if
        ! retry with new startpoint
        goto 100
      end if
    end if
    ! if we are here, then we have successfully build a chain (which has not
    ! been accepted yet)
    max_reached_chain_length = &
         max(max_reached_chain_length, n_mon)
    sum_reached_chain_lengths = sum_reached_chain_lengths &
         + real(n_mon, kind=dp)
    write(unit=*, fmt='(a, f11.3)') &
         "MESSAGE (SR insert_chain): Completed chain, average length of &
         &builds=", sum_reached_chain_lengths &
         / real(counter_chain_build_tries, kind=dp)
    ! get out of loop
    exit
    ! we retry
100 continue
    if(counter_chain_build_tries .eq. n_max_chain_build_tries) then
      write(unit=*, fmt='(a, i7)') &
           "ERROR (SR insert_chain): counter_chain_build_tries .eq. &
           &n_max_chain_build_tries=", n_max_chain_build_tries
      write(unit=*, fmt='(a, i8, a, f11.3)') &
           "MESSAGE (SR insert_chain): max_reached_chain_length=", &
           max_reached_chain_length, " average length of builds=", &
           sum_reached_chain_lengths &
           / real(counter_chain_build_tries, kind=dp)
      write(unit=*, fmt='(a)') "        Giving up.  System to small?"
      stop
    end if
  end do ! chain build tries
  !============================================================================
  ! DEBUG
  if(l_debug_insert_chain) then
    do i_mon_pos = 1, n_mon
      write(unit=*, fmt =*) &
           "DEBUG: cumulated_weights(", i_mon_pos, ")=", &
           cumulated_weights(i_mon_pos)
    end do
  end if
  ! rescale weights to avoid numerical underflow
  cumulated_weights(:) = rosenbluth_rescale * cumulated_weights(:)
  ! calculate Rosenbluth factor of newly grown chain
  rosenbluth_factor_new = product(cumulated_weights(:))
  ! DEBUG
  if(l_debug_insert_chain) then
    write(unit=*, fmt=*) &
         "DEBUG:  rosenbluth_factor_new=", rosenbluth_factor_new
    write(unit=*, fmt=*) &
         "DEBUG:  rosenbluth_factor_old=", rosenbluth_factor_old
  end if
  ! insert the chain according to its Rosenbluth factor
  if(random_mon(n_mon) * rosenbluth_factor_old &
       .lt. rosenbluth_factor_new) then
    l_new_chain_inserted = .TRUE.
    if(l_debug_insert_chain) then
      write(unit=*, fmt='(a)') &
           "DEBUG:  accepted chain for insertion"
    end if
    ! add up energy of newly chosen chain
    do i_mon_pos = 1, n_mon
      total_potential_chains = total_potential_chains &
           + potential_mon(chosen_trial_vectors(i_mon_pos), i_mon_pos)
    end do
    ! If we applied the pbc during creation of the chain, and particle i
    ! gets folded in the simulation box, particle i+1 would not be folded.
    ! This seems a bit artificial, although the simulation runs identically
    ! with either method, as we are working with equivalence classes of
    ! particles anyways.
    ! apply periodic boundary conditions if neccessary for the whole chain in
    ! the calling program unit.
    do i_mon_pos = 1, n_mon
      r0(n_particles_before + i_mon_pos, :) = &
           chain_positions_full(:, i_mon_pos)
    end do
  else
    l_new_chain_inserted = .FALSE.
    if(l_debug_insert_chain) then
      write(unit=*, fmt='(a)') &
           "DEBUG:  did not accept chain for insertion"
    end if
  end if
  write(unit=*, fmt='(a, g10.3, a, l2, a)') &
       "MESSAGE (SR insert_chain): R_new/R_old =", &
       rosenbluth_factor_new / rosenbluth_factor_old, &
       " (chain accepted:", l_new_chain_inserted, ")"
  if(l_debug_insert_chain) print *, "DEBUG: Leaving SR insert_chain"
end subroutine insert_chain
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! finds bin indices
subroutine get_bin_indices(point)
  real(kind=dp), dimension(n_dim), intent(in) :: point
  integer, dimension(n_dim_max) :: i_bin
  integer :: i_dim
  i_bin(n_dim+1:n_dim_max) = 0
  ! find indices of bins
  do i_dim = 1, n_dim_pbc
    i_bin(i_dim)= int(point(i_dim)/r_bin(i_dim))
    if(i_bin(i_dim) .eq. n_bin(i_dim)) i_bin(i_dim) = n_bin(i_dim) - 1
  end do
  if(n_dim .eq. n_dim_pbc+1) then
    i_bin(n_dim)= int((point(n_dim) - lowest_fluid_z)/r_bin(n_dim))
    if(i_bin(n_dim) .eq. n_bin(n_dim)) i_bin(n_dim) = n_bin(n_dim) - 1
  end if
  write(unit=*, fmt='(a, 3(i3, a))') &
       "Is in bin(", i_bin(1), ",", i_bin(2), ",", i_bin(n_dim_max), "):"
  write(unit=*, fmt=*) &
       bin_fluid(i_bin(1), i_bin(2), i_bin(n_dim_max), :)
end subroutine get_bin_indices
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
subroutine apply_mic(delta_r)
  real(kind=dp), dimension(n_dim), intent(inout) :: delta_r
  integer :: i_dim
  ! minimum image convention in xy plane only
  ! here one can save a lot of cpu time if the right method is chosen
  if(l_mic_use_int_cast) then
    do i_dim = 1, n_dim_pbc
      delta_r(i_dim) = delta_r(i_dim) - boundary(i_dim) &
           * aint(delta_r(i_dim)/half_bound(i_dim))
    end do
  else ! faster on Pentiums
    do i_dim = 1, n_dim_pbc
      if(delta_r(i_dim) > half_bound(i_dim)) then
        delta_r(i_dim) = delta_r(i_dim) - boundary(i_dim)
      else if(delta_r(i_dim) < -half_bound(i_dim)) then
        delta_r(i_dim) = delta_r(i_dim) + boundary(i_dim)
      end if
    end do
  end if
end subroutine apply_mic
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! gets the bond length for to monomers.
! If the integrated probability is present, then we draw a bond length
! according to Boltzmann weights.  If not, just take the energetically
! preferred bond-length.
function choose_length(type_one, type_two, int_probability)
  real(kind=dp) :: choose_length
  integer, intent(in) :: type_one, type_two
  ! integrated probability for drawing bond-length
  real(kind=dp), optional :: int_probability
  if(present(int_probability)) then
    write(unit=*, fmt='(a)') "ERROR (FCT choose_length): Selecting &
         &bond-lengths with Boltzmann weights not implemented yet..."
    stop
  end if
  choose_length = scale_bond_length * bond_length(type_one, type_two)
end function choose_length
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! puts fluid particles in index range l_part_index, u_part_index into bins.
! If only l_part_index is present, the we update the bins with this index.
subroutine bin_fluid_particles(l_part_index, u_part_index)
  integer, intent(in) :: l_part_index
  integer, intent(in), optional :: u_part_index
  integer :: i_dim, i_part, part_range
  ! bin looping variables
  integer, dimension(n_dim_max) :: i_bin
  ! the bin geometry was defined in SR init_binning
  ! reset bins if we bin a range of particles, one single particle is just
```

```
  ! added
  if(present(u_part_index)) then
    bin_fluid(:, :, :, 0)  = 0
    bin_fluid(:, :, :, 1:) = huge(1)
    part_range = u_part_index - l_part_index
  else
    part_range = 0
  end if
  !==================!
  ! bin the particles !
  !==================!
  ! As r_bin(1) = boundary(1)/n_bin(1) one could write
  ! i_bin_x = int(r0(i_part,1)/r_bin(1)) if 0 <= position < L if we had
  ! infinite precision.
  ! But because of finite precision int(r0(i_part,1)/r_bin(1)) could become
  ! n_bin(1) (if r_bin(1) was rounded towards zero) outside the valid range.
  ! To prevent this we use the if() statements after the int cast.
  ! bins in dimensions which are not in use get i_bin = 0
  i_bin(n_dim+1:n_dim_max) = 0
  do i_part = l_part_index, l_part_index + part_range
    ! check range of coordinates
    ! if(l_binning3d_check_range) call binning3d_check_range(i_part)
    ! find indices of bins
    do i_dim = 1, n_dim_pbc
      i_bin(i_dim)= int(r0(i_part, i_dim)/r_bin(i_dim))
      if(i_bin(i_dim) .eq. n_bin(i_dim)) i_bin(i_dim) = n_bin(i_dim) - 1
    end do
    ! lowest_fluid_z is defined in SR binning3d
    if(n_dim .eq. n_dim_pbc+1) then
      i_dim = n_dim
      i_bin(i_dim)= int((r0(i_part, i_dim) - lowest_fluid_z) /r_bin(i_dim))
      if(i_bin(i_dim) .eq. n_bin(i_dim)) i_bin(i_dim) = n_bin(i_dim) - 1
    end if
    ! increment counter
    bin_fluid(i_bin(1), i_bin(2), i_bin(n_dim_max), 0) = &
         bin_fluid(i_bin(1), i_bin(2), i_bin(n_dim_max), 0) + 1
    ! check overrun
    if(l_binning3d_check_bin_overrun) then
      if(bin_fluid(i_bin(1), i_bin(2), i_bin(n_dim_max), 0).gt.n_bin_fl) then
        write(unit=*, fmt='(a,i4,a,i4,a,i4,a,i3,a,i3)') &
             "ERROR (SR bin_fluid_particles): bin_fluid(", &
             i_bin(1),",",i_bin(2),",",i_bin(n_dim_max),",0)=", &
             bin_fluid(i_bin(1), i_bin(2), i_bin(n_dim_max), 0), &
             " > n_bin_fl=", n_bin_fl
        stop
      end if
    end if !(l_binning3d_check_bin_overrun)
    ! put particle into bin
    bin_fluid(i_bin(1), i_bin(2), i_bin(n_dim_max), &
         bin_fluid(i_bin(1), i_bin(2), i_bin(n_dim_max), 0)) = i_part
    if(l_debug_bin_fluid_particles) then
      write(unit=*, fmt='(a, i7, a, 3g13.6)') &
           "DEBUG (SR bin_fluid_particles): r0(", i_part, ",:)=", &
           r0(i_part, :)
      write(unit=*, fmt='(a, 3(i4, a))') &
           "     was put in bin (", i_bin(1), ",", i_bin(2), &
           ",", i_bin(n_dim_max), ")"
    end if
  end do ! loop over all particles to bin
  if(l_binning3d_check_number) then
    if(n_mon_tot .gt. 0) then
      if(sum(bin_fluid(:, :, :, 0)) .ne. part_range+1) then
        write(unit=*, fmt='(a)') "ERROR (SR bin_fluid_particles): &
             &Incorrect number of particles in bin_fluid:"
        write(unit=*, fmt='(2(a, i11))') &
             "                          Found",  sum(bin_fluid(:, :, :, 0)), &
             ", but part_range+1=", part_range+1
        stop
      end if
    end if
  end if !(l_binning3d_check_number)
end subroutine bin_fluid_particles
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! bin wall particles
subroutine bin_wall_particles
  ! bin looping variables
  integer, dimension(n_dim_max-1) :: i_bin
  integer :: i_bin_x, i_bin_y
  integer :: i_dim, i_part
  ! sanity check
  if(n_dim .ne. n_dim_pbc+1) then
    write(unit=*, fmt='(a)') "ERROR (SR bin_wall_particles): &
         &n_dim .ne. n_dim_pbc+1 so there are no walls."
    stop
  end if
  bin_twall(:, :, 0)  = 0
  bin_twall(:, :, 1:) = huge(1)
  bin_bwall(:, :, 0)  = 0
  bin_bwall(:, :, 1:) = huge(1)
  ! top wall
  do i_part = 1+n_mon_tot,n_mon_tot+n_top_wall
      ! check range of coordinates
      ! if(l_binning3d_check_range) call binning3d_check_range(i_part)
      ! find indices of bins
      do i_dim = 1, n_dim_pbc
        i_bin(i_dim)= int(r0(i_part, i_dim)/r_bin(i_dim))
        if(i_bin(i_dim) .eq. n_bin(i_dim)) i_bin(i_dim) = n_bin(i_dim) - 1
      end do
      ! increment counter
      bin_twall(i_bin(1), i_bin(n_dim_max-1), 0) = &
         bin_twall(i_bin(1), i_bin(n_dim_max-1), 0) + 1
      ! check overrun
      if(l_binning3d_check_bin_overrun) then
        if(bin_twall(i_bin(1), i_bin(2), 0) .gt. n_bin_wa) then
          write(unit=*, fmt='(a,i1,a,i1,a,i3,a,i3)') &
               "ERROR (SR bin_wall_particles): bin_twall(", &
               i_bin(1),",",i_bin(2),",0)=", &
               bin_twall(i_bin(1), i_bin(2), 0), &
              " > n_bin_wa=", n_bin_wa
          stop
        end if
      end if !(l_binning3d_check_bin_overrun)
      ! put particle into bin
      bin_twall(i_bin(1), i_bin(2), bin_twall(i_bin(1), i_bin(2), 0)) &
          = i_part
  end do ! loop over top wall
  ! bottom wall
  do i_part = 1+n_mon_tot+n_top_wall, n_part
      ! check range of coordinates
      ! if(l_binning3d_check_range) call binning3d_check_range(i_part)

      ! find indices of bins
      do i_dim = 1, n_dim_pbc
        i_bin(i_dim)= int(r0(i_part, i_dim)/r_bin(i_dim))
        if(i_bin(i_dim) .eq. n_bin(i_dim)) i_bin(i_dim) = n_bin(i_dim) - 1
      end do
      ! increment counter
      bin_bwall(i_bin(1), i_bin(n_dim_max-1), 0) = &
           bin_bwall(i_bin(1), i_bin(n_dim_max-1), 0) + 1
      ! check overrun
      if(l_binning3d_check_bin_overrun) then
        if(bin_bwall(i_bin(1), i_bin(2), 0) .gt. n_bin_wa) then
          write(unit=*, fmt='(a,i1,a,i1,a,i3,a,i3)') &
               "ERROR (SR bin_wall_particles): bin_bwall(", &
               i_bin(1),",",i_bin(2),",0)=", &
               bin_bwall(i_bin(1), i_bin(2), 0), &
              " > n_bin_wa=", n_bin_wa
          stop
        end if
      end if !(l_binning3d_check_bin_overrun)
      ! put particle into bin
      bin_bwall(i_bin(1), i_bin(2), bin_bwall(i_bin(1), i_bin(2), 0)) &
          = i_part
  end do !loop over bottom wall
  ! check if we have all particles
  if(l_binning3d_check_number) then
    if(n_dim .eq. n_dim_pbc+1) then
      if(sum(bin_twall(:, :, 0)) .ne. n_top_wall) then
        write(unit=*, fmt='(a)') "ERROR (SR bin_wall_particles): &
             &Incorrect number of particles in bin_twall:"
        write(unit=*, fmt='(2(a, i11))') &
             "                          Found",  sum(bin_twall(:, :, 0)), &
             ", but n_top_wall= ", n_top_wall
        stop
      end if
      if(sum(bin_bwall(:, :, 0)) .ne. n_bottom_wall) then
        write(unit=*, fmt='(a)') "ERROR (SR bin_wall_particles): &
             &Incorrect number of particles in bin_bwall:"
        write(unit=*, fmt='(2(a, i11))') &
             "                          Found",  sum(bin_bwall(:, :, 0)), &
             ", but n_bottom_wall= ", n_bottom_wall
        stop
      end if
    end if
  end if !(l_binning3d_check_number)
  if(.FALSE.) then
    do i_bin_x = 0, n_bin(1)-1
      do i_bin_y = 0, n_bin(2)-1
        write(unit=*, fmt='(a, i3, a, i3, a, i3)') &
           "bin_twall(", i_bin_x, ",", i_bin_y, ", 0)=", &
           bin_twall(i_bin_x, i_bin_y, 0)
        write(unit=*, fmt=*) bin_twall(i_bin_x, i_bin_y, 1:)
      end do
    end do
    do i_bin_x = 0, n_bin(1)-1
      do i_bin_y = 0, n_bin(2)-1
        write(unit=*, fmt='(a, i3, a, i3, a, i3)') &
             "bin_bwall(", i_bin_x, ",", i_bin_y, ", 0)=", &
             bin_bwall(i_bin_x, i_bin_y, 0)
        write(unit=*, fmt=*) bin_bwall(i_bin_x, i_bin_y, 1:)
      end do
    end do
  end if
end subroutine bin_wall_particles
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! LJ potential for two particles of type i_type and j_type at squared
! distance r_2
real(kind=dp) function lj_pot(i_type, j_type, r_2)
  integer, intent(in) :: i_type, j_type
  real(kind=dp), intent(in) :: r_2
  real(kind=dp) :: r_6, r_12
  r_6 = (sigma_2(i_type,j_type)/r_2)**3
  r_12 = r_6**2
  lj_pot = four_epsil(i_type,j_type) * (r_12 - r_6 - e_shift(i_type,j_type))
end function lj_pot
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! potential of a fluid particle at point exercised from the walls
subroutine pot_walls(point, j_type, potential, l_infinity)
  real(kind=dp), dimension(n_dim), intent(in) :: point
  integer :: i_dim
  ! the type of the fluid particle
  integer, intent(in) :: j_type
  real(kind=dp), intent(out) :: potential
  logical, intent(out) :: l_infinity
  ! particle type (wall has type n_type)
  integer, parameter :: i_type = n_type
  ! bin and particle indices
  integer, dimension(n_dim_max-1) :: i_bin
  integer :: i_bin_x, i_bin_y, j_bin_x, j_bin_y, k_bin_x, k_bin_y
  integer :: i_bin_part, i_part
  ! helpers
  real(kind=dp) :: r_2
  real(kind=dp), dimension(n_dim) :: delta_r
  if(l_debug_pot_walls) print *, "DEBUG: Entering SR pot_walls"
  if(l_debug_pot_walls) then
    if(j_type .lt. 0 .or. j_type .gt. n_type) then
         write(unit=*, fmt=*) &
         "ERROR (SR pot_walls): j_type=", j_type, " out of range"
         stop
       end if
  end if
  ! initialize
  potential = 0.0_dp
  if(n_dim .eq. n_dim_pbc+1) then
    ! check range of point.  If "z"-coordinate is outside walls, give this
    ! point "infinite" potential
    if(point(n_dim) .ge. r0_twall(n_dim) &
         .or. point(n_dim) .le. r0_bwall(n_dim)) then
      l_infinity = .TRUE.
      return
    end if
  end if
  l_infinity = .FALSE.
  ! point must lie inside the box !
  i_bin(n_dim_pbc+1:n_dim_max-1) = 0
  ! find bin indices
  do i_dim =1, n_dim_pbc
    i_bin(i_dim) = int(point(i_dim) / r_bin(i_dim))
    if(i_bin(i_dim) .eq. n_bin(i_dim)) i_bin(i_dim) = n_bin(i_dim) - 1
  end do
  if(l_debug_pot_walls) then
    write(unit=*, fmt=*) &
         "DEBUG: point=", point(:), " is in bin (" , i_bin(1), &
         ",", i_bin(2), ")"
  end if
  i_bin_x = i_bin(1)
  i_bin_y = i_bin(n_dim_max-1)
  ! loop over approbriate neighbor bins as set by delta_bin(X)
  do j_bin_x = i_bin_x-1, i_bin_x + delta_bin(1)
    if(n_dim_pbc .ge. 1) then
      ! do not forget periodic boundary conditions
      if(j_bin_x .lt. 0) then
        k_bin_x = n_bin(1) - 1
      else if(j_bin_x .eq. n_bin(1)) then
        k_bin_x = 0
      else
```

```
        k_bin_x = j_bin_x
      end if
    else ! note: the wall has only directions with pbc
      ! direction not used
      k_bin_x = 0
    end if
    do j_bin_y = i_bin_y-1, i_bin_y + delta_bin(2)
      if(n_dim_pbc .ge. 2) then
        if(j_bin_y .lt. 0) then
          k_bin_y = n_bin(2) - 1
        else if(j_bin_y .eq. n_bin(2)) then
          k_bin_y = 0
        else
          k_bin_y = j_bin_y
        end if
      else
        k_bin_y = 0
      end if
      ! loop over all particles in this top wall bin
      do i_bin_part = 1, bin_twall(k_bin_x, k_bin_y, 0)
        ! get the particle index
        i_part = bin_twall(k_bin_x, k_bin_y, i_bin_part)
        ! the vector from test particle to wall particle
        delta_r(:) = r0(i_part, :) - point(:)
        call apply_mic(delta_r)
        r_2 = dot_product(delta_r(:), delta_r(:))
        ! check whether interaction takes place
        if(r_2.lt.range_2(i_type,j_type)) then
          ! to avoid a (numerical) division by zero problem, we
          ! test if r_2 is (numerically) zero
          if(r_2 .le. 0.0_dp) then
            if(l_debug_pot_walls) &
                 print *, "encountered r_2 = 0.0_dp, returning."
            l_infinity = .TRUE.
            potential = 0.0_dp
            return
          end if
          potential = potential + lj_pot(i_type, j_type, r_2)
        end if !(r_2.lt.range_2(i_type,j_type))
      end do
      ! loop over all particles in this bottom wall bin
      do i_bin_part = 1, bin_bwall(k_bin_x, k_bin_y, 0)
        ! get the particle index
        i_part = bin_bwall(k_bin_x, k_bin_y, i_bin_part)
        ! the vector from test particle to wall particle
        delta_r(:) = r0(i_part, :) - point(:)
        call apply_mic(delta_r)
        r_2 = dot_product(delta_r(:), delta_r(:))
        ! check whether interaction takes place
        if(r_2.lt.range_2(i_type,j_type)) then
          ! to avoid a (numerical) division by zero problem, we
          ! test if r_2 is (numerically) zero
          if(r_2 .le. 0.0_dp) then
            if(l_debug_pot_walls) &
                 print *, "encountered r_2 = 0.0_dp, returning."
            l_infinity = .TRUE.
            potential = 0.0_dp
            return
          end if
          potential = potential + lj_pot(i_type, j_type, r_2)
        end if !(r_2.lt.range_2(i_type,j_type))
      end do
    end do
  end do
  if(l_debug_pot_walls) &
       print *, "DEBUG: Leaving SR pot_walls with potential=", potential
end subroutine pot_walls
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! Potential energy of a fluid particle at point, in the potentials of the
! other fluid particles not in range
! l_part_excl_index ... u_part_excl_index.
! if point is too close to another fluid particle (backfolding !) then
! l_infinity is set to .TRUE.
subroutine pot_fluid(point, j_type, potential, l_infinity, &
     l_part_excl_index, u_part_excl_index)
  real(kind=dp), dimension(n_dim), intent(in) :: point
  integer, intent(in) :: j_type
  real(kind=dp), intent(out) :: potential
  logical, intent(out) :: l_infinity
  integer, intent(in), optional :: l_part_excl_index, u_part_excl_index
  integer :: i_dim
  ! particle type
  integer :: i_type
  ! bin and particle indices
  integer, dimension(n_dim_max) :: i_bin
  integer ::  i_bin_x, i_bin_y, i_bin_z, j_bin_x, j_bin_y, j_bin_z, &
       k_bin_x, k_bin_y, k_bin_z, i_bin_part, n_bin_part, i_part
  ! helpers
  real(kind=dp) :: r_2, potential_check
  real(kind=dp), dimension(n_dim) :: delta_r
  if(l_debug_pot_fluid) print *, "DEBUG: Entering SR pot_fluid"
  if(l_debug_pot_fluid) then
    if(j_type .lt. 0 .or. j_type .gt. n_type) then
      write(unit=*, fmt=*) &
           "ERROR (SR pot_fluid): j_type=", j_type, " out of range"
      stop
    end if
    write(unit=*, fmt='(a,3(f9.3,a))') &
         "DEBUG (SR pot_fluid): point=(", point(1), ",", point(2), &
         ",", point(n_dim), ")"
    write(unit=*, fmt='(a,i12,a,i12)') &
         "DEBUG (SR pot_fluid): Excluding i_part=", l_part_excl_index, &
         "...", u_part_excl_index
  end if
  ! initialize
  potential = 0.0_dp
  l_infinity = .FALSE.
  i_bin(n_dim_pbc+1:n_dim_max) = 0
  ! find bin indices
  do i_dim =1, n_dim_pbc
    i_bin(i_dim) = int(point(i_dim) / r_bin(i_dim))
    if(i_bin(i_dim) .eq. n_bin(i_dim)) i_bin(i_dim) = n_bin(i_dim) - 1
  end do
  if(n_dim .eq. n_dim_pbc+1) then
    i_dim = n_dim
    ! lowest_fluid_z is defined in SR init_binning
    i_bin(i_dim) = int((point(i_dim) - lowest_fluid_z) / r_bin(i_dim))
    if(i_bin(i_dim) .eq. n_bin(i_dim)) i_bin(i_dim) = n_bin(i_dim) - 1
  end if
  i_bin_x = i_bin(1)
  i_bin_y = i_bin(2)
  i_bin_z = i_bin(n_dim)
  ! we use n_bin_used here, in order to have the same code as in SR binning3d
  ! they are initialized with n_bin_used(:) = n_bin(:) in SR init_binning
  ! loop over approbriate neighbor bins as set by delta_bin(X)
  do j_bin_x = i_bin_x-1, i_bin_x + delta_bin(1)
    if(n_dim_pbc .ge. 1) then
      ! do not forget periodic boundary conditions
      if(j_bin_x .lt. 0) then
        k_bin_x = n_bin(1) - 1
      else if(j_bin_x .eq. n_bin(1)) then
        k_bin_x = 0
      else
        k_bin_x = j_bin_x
      end if
    else if(n_dim .ge. 1) then
      ! no pbc
      if(j_bin_x .lt. 0 .or. j_bin_x .gt. n_bin_used(1)-1) cycle
      k_bin_x = j_bin_x
    else
      ! direction not used
      k_bin_x = 0
    end if
    do j_bin_y = i_bin_y-1, i_bin_y + delta_bin(2)
      if(n_dim_pbc .ge. 2) then
        if(j_bin_y .lt. 0) then
          k_bin_y = n_bin(2) - 1
        else if(j_bin_y .eq. n_bin(2)) then
          k_bin_y = 0
        else
          k_bin_y = j_bin_y
        end if
      else if(n_dim .ge. 2) then
        if(j_bin_y .lt. 0 .or. j_bin_y .gt. n_bin_used(2)-1) cycle
        k_bin_y = j_bin_y
      else
        k_bin_y = 0
      end if
      do j_bin_z = i_bin_z-1, i_bin_z + delta_bin(n_dim_max)
        if(n_dim_pbc .eq. n_dim_max) then
          if(j_bin_z .lt. 0) then
            k_bin_z = n_bin(n_dim_max) - 1
          else if(j_bin_z .eq. n_bin(n_dim_max)) then
            k_bin_z = 0
          else
            k_bin_z = j_bin_z
          end if
        else if(n_dim .eq. n_dim_max) then
          if(j_bin_z.lt.0 .or. j_bin_z.gt.n_bin_used(n_dim_max)-1) cycle
          k_bin_z = j_bin_z
        else
          k_bin_z = 0
        end if
        ! # of particles in bin
        n_bin_part = bin_fluid(k_bin_x, k_bin_y, k_bin_z, 0)
        ! loop over all possible neighbors
        do i_bin_part = 1, n_bin_part
          ! get the particle index
          i_part = bin_fluid(k_bin_x, k_bin_y, k_bin_z, i_bin_part)
          ! if the particle index is in the excluded range,
          ! cycle this do loop
          if(present(l_part_excl_index) &
               .and. present(u_part_excl_index)) then
            if(i_part .ge. l_part_excl_index &
                 .and. i_part .le. u_part_excl_index) then
              if(l_debug_pot_fluid) &
                   print *, "DEBUG (SR pot_fluid): Skipped i_part=", i_part
              cycle
            end if
          end if
          ! get the type
          i_type = type(i_part)
          ! the vector from test particle to fluid particle
          delta_r(:) = r0(i_part, :) - point(:)
          call apply_mic(delta_r)
          r_2 = dot_product(delta_r(:), delta_r(:))
          if(l_debug_pot_fluid) write(unit=*, fmt='(a,i6,a,g13.6)') &
               "DEBUG (SR pot_fluid): i_part=", i_part, ", r=", sqrt(r_2)
          ! check whether interaction takes place
          if(r_2.lt.range_2(i_type,j_type)) then
            ! to avoid a (numerical) division by zero problem, we
            ! test if r_2 is (numerically) zero
            if(r_2 .le. 0.0_dp) then
              if(l_debug_pot_fluid) &
                   print *, "encountered r_2 = 0.0_dp, returning."
              l_infinity = .TRUE.
              potential = 0.0_dp
              return
            end if
            if(l_debug_pot_fluid) write(unit=*, fmt='(a,i6,a,g13.6)') &
                 "DEBUG (SR pot_fluid): i_part=", i_part, ", lj_pot=", &
                 lj_pot(i_type, j_type, r_2)
            potential = potential + lj_pot(i_type, j_type, r_2)
          end if !(r_2.lt.range_2(i_type,j_type))
        end do
      end do
    end do
  end do

  if(l_debug_pot_fluid) then
    ! check result against a calculation without binning
    potential_check = 0.0_dp
    do i_part = 1, n_mon_tot
      ! if the particle index is in the excluded range,
      ! cycle this do loop
      if(present(l_part_excl_index) &
           .and. present(u_part_excl_index)) then
        if(i_part .ge. l_part_excl_index &
             .and. i_part .le. u_part_excl_index) then
          if(l_debug_pot_fluid) &
               print *, "DEBUG (SR pot_fluid): Skipped i_part=", i_part
          cycle
        end if
      end if
      ! get the type
      i_type = type(i_part)
      ! the vector from test particle to fluid particle
      delta_r(:) = r0(i_part, :) - point(:)
      call apply_mic(delta_r)
      r_2 = dot_product(delta_r(:), delta_r(:))
      ! check whether interaction takes place
      if(r_2.lt.range_2(i_type,j_type)) then
        ! to avoid a (numerical) division by zero problem, we
        ! test if r_2 is (numerically) zero
        if(r_2 .le. 0.0_dp) then
          if(l_debug_pot_fluid) &
               print *, "encountered r_2 = 0.0_dp, returning."
          l_infinity = .TRUE.
          potential = 0.0_dp
          return
        end if
        if(l_debug_pot_fluid) write(unit=*, fmt='(a,i6,a,g13.6)') &
             "DEBUG (SR pot_fluid): i_part=", i_part, ", lj_pot=", &
             lj_pot(i_type, j_type, r_2)
        potential_check = potential_check + lj_pot(i_type, j_type, r_2)
      end if
    end do ! loop over all particles
```

```
    if(abs(potential - potential_check) &
         .gt. (10.0_dp)**(-precision(potential)+1)) then
      write(unit=*, fmt='(2(a, g13.6))') "ERROR (SR pot_fluid): &
           &potential=", potential, ", potential_check=", potential_check
      stop
    end if
  end if
  if(l_debug_pot_fluid) &
       print *, "DEBUG Leaving pot_fluid with", potential
end subroutine pot_fluid
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! checks if a chain crosses itself
subroutine check_xy_bond_crossings_self(coordinates, n_xy_bond_crossings, &
     l_swap_positions)
  real(kind=dp), dimension(n_dim, n_mon), intent(inout) :: coordinates
  integer, intent(out) :: n_xy_bond_crossings
  logical, intent(in), optional :: l_swap_positions
  integer :: i_bond_1, i_bond_2, i_bond_1_start
  integer :: reverse_start, reverse_stop
  ! initialize
  n_xy_bond_crossings = 0
  i_bond_1_start = 1
200 continue
  do i_bond_1 = i_bond_1_start, n_mon -1
    do i_bond_2 = 1, n_mon -1
      ! bonds can't cross with neighboring bonds
      if(i_bond_1 .eq. i_bond_2-1 &
           .or. i_bond_1 .eq. i_bond_2 &
           .or. i_bond_1 .eq. i_bond_2+1) cycle
      if(bonds_cross(coordinates(:, i_bond_1), &
                     coordinates(:, i_bond_1+1), &
                     coordinates(:, i_bond_2), &
                     coordinates(:, i_bond_2+1))) then
        n_xy_bond_crossings = n_xy_bond_crossings +1
        if(l_check_xy_bond_xings_self_verb) then
          write(unit=*, fmt='(a, i7, a, i7)') &
               "MESSAGE (check_xy_bond_crossings_self): Bond crossing &
               &between bonds", i_bond_1, " and", i_bond_2
        end if
        ! swap the positions of the chain segment which lead to these
        ! crossed bonds
        if(present(l_swap_positions)) then
          if(l_swap_positions) then
            ! find right range of positions to invert
            ! (the innermost positions)
            if(i_bond_1 .gt. i_bond_2) then
              reverse_start = i_bond_2 +1
              reverse_stop  = i_bond_1
            else
              reverse_start = i_bond_1 +1
              reverse_stop  = i_bond_2
            end if
            call reverse_positions(coordinates(:,reverse_start:reverse_stop))
            ! new bond-crossings could have been created, so start anew
            ! at the smallest bond involved.
            i_bond_1_start = min(i_bond_1, i_bond_2)
            goto 200
          end if
        end if
      end if ! bonds cross
    end do
  end do
end subroutine check_xy_bond_crossings_self
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! reverses the positions in array coordinates
subroutine reverse_positions(coordinates)
  real(kind=dp), dimension(:, :), intent(inout) :: coordinates
  ! copy of the coordinates
  real(kind=dp), dimension(size(coordinates, 1), &
                           size(coordinates, 2)) :: copy_coordinates
  integer :: i_loop
  copy_coordinates = coordinates
  do i_loop = 0, size(coordinates, 2) -1
    coordinates(:, lbound(coordinates, 2) + i_loop) = &
         copy_coordinates(:, ubound(copy_coordinates, 2) - i_loop)
  end do
end subroutine reverse_positions
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! checks if there are chain crossings in the xy-plane in a configuration
! Unfolded coordinates are used.
! Note: This routines scales with n_mon_tot**2, so it is slow for large
! systems.  Don't call it too often.
logical function check_xy_bond_crossings_all(u_chain_index)
  integer, intent(in), optional :: u_chain_index
  integer :: i_bond_1, i_bond_2, max_bond_index, first_bond_in_next_chain, &
       i_dim
  real(kind=dp), dimension(1:n_dim_pbc) :: cm_1_f, cm_2_f, cm_1_uf, cm_2_uf,&
         pbc_shift_1,  pbc_shift_2, delta_cm, &
         pos11_f, pos12_f, pos21_f, pos22_f
  real(kind=dp) :: r_2, max_bond, max_bond_2
  integer :: n_xy_bond_crossings = 0
  if(l_debug_check_xybx_all) &
       print *, "DEBUG: Entering FCT check_xy_bond_crossings_all"
  if(n_dim_pbc .ne. 2) then
    write(unit=*, fmt='(a)') "ERROR (FCT check_xy_bond_crossings_all): &
         &n_dim_pbc .ne. 2"
    stop
  end if
  if(present(u_chain_index)) then
    max_bond_index = u_chain_index * n_mon -1
  else
    max_bond_index = n_chain * n_mon -1
  end if
  ! maximal bond_length, squared
  max_bond_2 = r_chain_2 * sigma_2(1, 1)
  max_bond = sqrt(max_bond_2)
  i_bond_1 = 1
  do while(i_bond_1 .le. max_bond_index)
    ! cycle for chain ends
    if(mod(i_bond_1, n_mon) .eq. 0) then
      i_bond_1 = i_bond_1 +1
      cycle
    end if
    ! get center of mass
    cm_1_uf(:) = 0.5_dp * (r0_unfolded(i_bond_1, 1:n_dim_pbc) &
         + r0_unfolded(i_bond_1+1, 1:n_dim_pbc))
    ! fold cm back in box
    do i_dim = 1, n_dim_pbc
      pbc_shift_1(i_dim) = real(floor(cm_1_uf(i_dim) / boundary(i_dim)), &
           kind=dp)
      cm_1_f(i_dim) = cm_1_uf(i_dim) - pbc_shift_1(i_dim) * boundary(i_dim)
    end do
    pos11_f(:)=r0_unfolded(i_bond_1,1:n_dim_pbc)-pbc_shift_1(:)*boundary(:)
    pos12_f(:)=r0_unfolded(i_bond_1+1,1:n_dim_pbc)-pbc_shift_1(:)*boundary(:)
    if(l_debug_check_xybx_all) then
        print *, "DEBUG: i_bond_1=", i_bond_1
        print *, "DEBUG: cm_1_uf(:)=", cm_1_uf(:)
        print *, "DEBUG: cm_1_f(:)=", cm_1_f(:)
        print *, "DEBUG: pos11_f(:)=", pos11_f(:)
        print *, "DEBUG: pos12_f(:)=", pos12_f(:)
    end if
    ! a bond cannot cross itself nor the next bond (as defined by the
    ! indices), so start at i_bond_1 +2
    i_bond_2 = i_bond_1 +2
    do while(i_bond_2 .le. max_bond_index)
      ! cycle for chain ends
      if(mod(i_bond_2, n_mon) .eq. 0) then
        i_bond_2 = i_bond_2 +1
        cycle
      end if
      ! get center of mass
      cm_2_uf(:) = 0.5_dp * (r0_unfolded(i_bond_2, 1:n_dim_pbc) &
           + r0_unfolded(i_bond_2+1, 1:n_dim_pbc))
      ! fold cm back in box
      do i_dim = 1, n_dim_pbc
        pbc_shift_2(i_dim) = real(floor(cm_2_uf(i_dim) / boundary(i_dim)), &
             kind=dp)
        cm_2_f(i_dim) = cm_2_uf(i_dim) - pbc_shift_2(i_dim) * boundary(i_dim)
      end do
      ! calculate distance of the centers taking the into account
      ! the minimum image convention
      delta_cm(:) = cm_1_f(:) - cm_2_f(:)
      if(l_debug_check_xybx_all) then
        print *, "DEBUG: i_bond_2=", i_bond_2
        print *, "DEBUG: cm_2_uf(:)=", cm_2_uf(:)
        print *, "DEBUG: cm_2_f(:)=", cm_2_f(:)
        print *, "DEBUG: delta_cm(:)=", delta_cm(:)
      end if
      ! count shifts negative for cm_2_f
      do i_dim = 1, n_dim_pbc
        if(delta_cm(i_dim) > half_bound(i_dim)) then
          delta_cm(i_dim) = delta_cm(i_dim) - boundary(i_dim)
          pbc_shift_2(i_dim) = pbc_shift_2(i_dim) - 1.0_dp
        else if(delta_cm(i_dim) < -half_bound(i_dim)) then
          delta_cm(i_dim) = delta_cm(i_dim) + boundary(i_dim)
          pbc_shift_2(i_dim) = pbc_shift_2(i_dim) + 1.0_dp
        end if
      end do
      ! get distance squared
      r_2 = dot_product(delta_cm(:), delta_cm(:))
      if(l_debug_check_xybx_all) then
        print *, "DEBUG: After MIC: delta_cm(:)=", delta_cm(:)
        print *, "DEBUG: r_2=", r_2
      end if
      if(r_2 .lt. max_bond_2) then
        pos21_f(:) = r0_unfolded(i_bond_2,   1:n_dim_pbc) &
             - pbc_shift_2(:) * boundary(:)
        pos22_f(:) = r0_unfolded(i_bond_2+1, 1:n_dim_pbc) &
         - pbc_shift_2(:) * boundary(:)
        if(l_debug_check_xybx_all) then
          print *, "DEBUG: pos21_f(:)=", pos11_f(:)
          print *, "DEBUG: pos22_f(:)=", pos12_f(:)
        end if
        if(bonds_cross(pos11_f(:), pos12_f(:), pos21_f(:), pos22_f(:), &
             "xy-bond-crossings.dat")) then
          n_xy_bond_crossings = n_xy_bond_crossings +1
          write(unit=*, fmt='(a, i7, a, i7)') &
               "MESSAGE (check_xy_bond_crossings_all): Bond crossing &
               &detected between bonds", i_bond_1, " and", i_bond_2
        end if
        ! move on to next bond
        i_bond_2 = i_bond_2 +1
      else if(r_2 .lt. 4.0_dp * max_bond_2) then
        ! i_bond_1 and i_bond_2 can't cross, move on to next bond
        i_bond_2 = i_bond_2 +1
      else
        ! here we compute how far away from i_bond_1 the next possible
        ! crossing bond must be *along* the chain.  If the chain ends, take
        ! the first bond of the next chain.
        first_bond_in_next_chain = ((i_bond_2 / n_mon) +1) * n_mon
        i_bond_2 = i_bond_2 + int(sqrt(r_2) / max_bond)
        i_bond_2 = min(first_bond_in_next_chain, &
             i_bond_2 + int(sqrt(r_2) / max_bond))
      end if
    end do ! loop over i_bond_2
    ! move on to next bond
    i_bond_1 = i_bond_1 +1
  end do ! loop over i_bond_1
  if(n_xy_bond_crossings .gt. 0) then
    check_xy_bond_crossings_all = .FALSE.
  else
    check_xy_bond_crossings_all = .TRUE.
  end if
  if(l_debug_check_xybx_all) &
       print *, "DEBUG: Leaving FCT check_xy_bond_crossings_all"
end function check_xy_bond_crossings_all
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
logical function bonds_cross(vec_1, vec_2, vec_3, vec_4, filename)
  ! bonds 1-2 and 3-4 are checked
  real(kind=dp), dimension(n_dim_pbc), intent(in) :: &
       vec_1, vec_2, vec_3, vec_4
  character(len=*), intent(in), optional :: filename
  integer, parameter :: fileunit = 20
  integer :: io_status
  ! helpers for the linear algebra
  real(kind=dp) :: a, b, c, d, e, f, s, t, det
  if(n_dim_pbc .ne. 2) then
    write(unit=*, fmt='(a)') "ERROR (FCT bonds_cross): n_dim_pbc .ne. 2"
    stop
  end if
  a = vec_2(1) - vec_1(1)
  b = vec_2(2) - vec_1(2)
  c = vec_3(1) - vec_4(1)
  d = vec_3(2) - vec_4(2)
  e = vec_3(1) - vec_1(1)
  f = vec_3(2) - vec_1(2)
  ! determinant of matrix((a, c), (b, d))
  det = a*d - b*c
  if(l_debug_bonds_cross) then
    print *, "det=", det
  end if
  ! det == 0 means bonds are parallel (and do not cross)
  ! we include also almost parallel.
  ! I don't now exactly when to say "det == 0", but spacing(x) gives the
  ! spacing of number in the vicinity of x and hence indicates total loss
  ! of precision in the computation of det.
  if(abs(det) .lt. spacing(a*d)) then
    bonds_cross = .FALSE.
    return
  end if
  ! where the crossing lies on the bonds
  t = (d*e - c*f)/det
```

```
    s = (a*f - b*e)/det
    if(l_debug_bonds_cross) then
      print *, "s=", s, ", t=", t
    end if
    ! due to finite precision, we exclude the borders by epsilon() if we want
    ! to exclude the knee of neighboring bonds to be counted as crossings
    ! Note: epsilon(x) = spacing(1.0_dp)
    ! if((t .le. epsilon(1.0_dp)) .or. (t .ge. 1.0_dp - epsilon(1.0_dp))) then
    if((t .lt. 0.0_dp) .or. (t .gt. 1.0_dp)) then
      ! crossing point along bond 1-2 not between 1 and 2
      bonds_cross = .FALSE.
      return
    end if
    ! if((s .le. epsilon(1.0_dp)) .or. (s .ge. 1.0_dp - epsilon(1.0_dp))) then
    if((s .lt. 0.0_dp) .or. (s .gt. 1.0_dp)) then
      ! crossing point along bond 3-4 not between 3 and 4
      bonds_cross = .FALSE.
      return
    end if
    bonds_cross = .TRUE.
    if(l_debug_bonds_cross) then
      write(unit=*, fmt='(2(a, g13.6))') &
           "DEBUG: Crossing in bond 1-2 at", t, ", bond 3-4 at", s
      write(unit=*, fmt='(a)') "DEBUG: Positions in x-y plane:"
      write(unit=*, fmt='(2f10.3)') vec_1(1), vec_1(2)
      write(unit=*, fmt='(2f10.3)') vec_2(1), vec_2(2)
      write(unit=*, fmt='(2f10.3)') vec_3(1), vec_3(2)
      write(unit=*, fmt='(2f10.3)') vec_4(1), vec_4(2)
    end if
    if(present(filename)) then
      ! append crossing point
      open(unit=fileunit, file=filename, status="unknown", &
           action="write", position="append", iostat=io_status)
      if(io_status .ne. 0) then
        write (unit=*, fmt='(3a)') &
             "ERROR: Couldn't open file >>", filename, &
             "<< for writing."
        stop
      else
        write(unit=fileunit, fmt='(5g13.5)') &
             vec_1(1) + t*a, vec_1(2) + t*b, det, t, s
        close(fileunit)
      end if
    end if ! present(filename)
    return
  end function bonds_cross
  !------------------------------------------------------------------------------

  !------------------------------------------------------------------------------
  subroutine write_chain_in_xy_plane(coordinates, filename)
    real(kind=dp), dimension(n_dim, n_mon), intent(in) :: coordinates
    character(len=*), intent(in) :: filename
    integer, parameter :: fileunit = 20
    integer :: io_status, i_mon
    if(n_dim_pbc .lt. 2) then
      write(unit=*, fmt='(a)') "ERROR (SR write_chain_in_xy_plane): &
           &n_dim_pbc .lt. 2"
      stop
    end if
    ! open file for replacement
    open(unit=fileunit, file=filename, status="replace", &
         action="write", iostat=io_status)
    if(io_status .ne. 0) then
      write (unit=*, fmt='(3a)') &
           "ERROR: Couldn't open file >>", filename, &
           "<< for writing."
      stop
    else
      write(unit=*, fmt='(3a)') &
           "MESSAGE: Writing chain projected onto xy-plane to >>", &
           filename, "<<"
      do i_mon = 1, n_mon
        write(unit=fileunit, fmt='(2f10.3)') &
             coordinates(1, i_mon), coordinates(2, i_mon)
      end do
    end if
    close(fileunit)
  end subroutine write_chain_in_xy_plane
  !------------------------------------------------------------------------------

  !------------------------------------------------------------------------------
  ! if the sigma of the fluid changes, we have to change several other
  ! variables, too.
  ! Neighbor-list radii and bond-lengths are not changed
  subroutine reinit_lj_params
    integer :: i_type, j_type
    real(kind=dp) :: r_dummy
    ! apply standard sum rules to Lennard Jones parameters
    do i_type = 1, n_type-1
      do j_type = i_type+1,n_type
        epsil(i_type,j_type) &
             = sqrt(epsil(i_type,i_type)*epsil(j_type,j_type))
        epsil(j_type,i_type) = epsil(i_type,j_type)
        sigma(i_type,j_type) &
             = (sigma(i_type,i_type)+sigma(j_type,j_type))/2.0_dp
        sigma(j_type,i_type) = sigma(i_type,j_type)
      end do
    end do
    ! initialize sigma_2
    ! initialize prefactors for bond-potentials
    ! absorb prefactor 4 in LJ-epsilon
    do i_type = 1,n_type
      do j_type = 1,n_type
        sigma_2(i_type,j_type) = sigma(i_type,j_type)**2
        epsil_k_chain_over_sigma_2(i_type, j_type) = &
             (epsil(i_type,j_type) * k_chain)/sigma_2(i_type,j_type)
        four_epsil(i_type,j_type) = 4.0_dp * epsil(i_type,j_type)
      end do
    end do
    ! define cutoffs and shifts in energy, find maximal interaction range
    do i_type = 1, n_type
      do j_type = 1, n_type
        select case(f_cut_off)
        case(0)
          ! purely repulsive potential
          ! 2**(1/6) = 1.122462
          range_2(j_type,i_type) = (2.0_dp**(1.0_dp/6.0_dp) &
               * sigma(j_type,i_type))**2
        case(1)
          ! frequently used by M. H. Mueser and M. O. Robbins
          range_2(j_type,i_type) = (2.2_dp*sigma(j_type,i_type))**2
        case(2)
          ! long range interaction as used by C. Bennemann, F. Varnik
          ! the difference to 2.2 is not very important.
          ! 2 * 2**(1/6) = 2.244924
          range_2(j_type,i_type) = (2.0_dp * 2.0_dp**(1.0_dp/6.0_dp) &
               * sigma(j_type,i_type))**2
        case(3)
          ! purely repulsive walls and fluid particles with long range
          ! LJ-interaction like in case 2
          range_2(j_type,i_type) = (2.0_dp**(1.0_dp/6.0_dp) &
               * sigma(j_type,i_type))**2
          ! fluid particles are always defined as type 1 (see SR conf_default)
          if((j_type .eq. 1) .and. (i_type .eq. 1)) then
            range_2(j_type,i_type) = (2.0_dp * 2.0_dp**(1.0_dp/6.0_dp) &
                 * sigma(j_type,i_type))**2
          end if
        case default ! fall-through
          write(unit=*, fmt='(a, i2, a)') &
               "ERROR (SR init_parameters): case(f_cut_off=",&
               f_cut_off, ") not recognized."
          stop
        end select
        r_dummy = (sigma(j_type,i_type)**2/range_2(j_type,i_type))**3
        ! Epsilon does not need to be attached at this point.  Epsilon enters
        ! in the interaction routines.
        e_shift(j_type, i_type) = r_dummy*(r_dummy-1.0_dp)
      end do
    end do
    ! LJ-ranges
    do i_type = 1, n_type
      do j_type = 1, n_type
        range_1(i_type, j_type) = sqrt(range_2(j_type, i_type))
      end do
    end do
  end subroutine reinit_lj_params
  !------------------------------------------------------------------------------
end module mcfluid
```

luxury.f90

```
!  The random number module, written by Alan Miller.
!
!  Notes for use with the "rheosimVx.y" MD-simulation program:
!------------------------------------------------------------------------------
!  On DEC Alpha the default integers are also 32 bit.
!  There's a very useful program called "kindfind.f90" to display the
!  different KINDS of REALS and INTEGERS available.
!  It seems that the numbers generated do not depend on the REAL KIND used, and
!  they are only approximately equal on different platforms.
!  Martin Aichele, 2001-02-09
!------------------------------------------------------------------------------
MODULE luxury
!  Subtract-and-borrow random number generator proposed by Marsaglia and
!  Zaman, implemented by F. James with the name RCARRY in 1991, and later
!  improved by Martin Luescher in 1993 to produce "Luxury Pseudorandom
!  Numbers".  Fortran 77 coded by F. James, 1993
!  References:
!  M. Luscher, Computer Physics Communications  79 (1994) 100
!  F. James, Computer Physics Communications 79 (1994) 111
!  This Fortran 90 version is by Alan Miller (alan @ mel.dms.csiro.au)
!  Latest revision - 11 September 1995
!  LUXURY LEVELS.
!  ------ ------      The available luxury levels are:
!  level 0  (p=24): equivalent to the original RCARRY of Marsaglia
!           and Zaman, very long period, but fails many tests.
!  level 1  (p=48): considerable improvement in quality over level 0,
!           now passes the gap test, but still fails spectral test.
!  level 2  (p=97): passes all known tests, but theoretically still
!           defective.
!  level 3  (p=223): DEFAULT VALUE.  Any theoretically possible
!           correlations have very small chance of being observed.
!  level 4  (p=389): highest possible luxury, all 24 bits chaotic.
!! +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ ++
!!  Calling sequences for RANLUX:                                       ++
!!      CALL RANLUX (RVEC, LEN)   returns a vector RVEC of LEN      ++
!!                   32-bit random floating point numbers between   ++
!!                   zero (not included) and one (also not incl.).  ++
!!      CALL RLUXGO(LUX,INT,K1,K2) initializes the generator from   ++
!!                one 32-bit integer INT and sets Luxury Level LUX   ++
!!                which is integer between zero and MAXLEV, or if    ++
!!                LUX .GT. 24, it sets p=LUX directly.  K1 and K2    ++
!!                should be set to zero unless restarting at a break ++
!!                point given by output of RLUXAT (see RLUXAT).      ++
!!      CALL RLUXAT(LUX,INT,K1,K2) gets the values of four integers ++
!!                which can be used to restart the RANLUX generator  ++
!!                at the current point by calling RLUXGO.  K1 and K2 ++
!!                specify how many numbers were generated since the  ++
!!                initialization with LUX and INT.  The restarting   ++
!!                skips over  K1+K2*E9   numbers, so it can be long. ++
!!   A more efficient but less convenient way of restarting is by:  ++
!!      CALL RLUXIN(ISVEC)    restarts the generator from vector    ++
!!                   ISVEC of 25 32-bit integers (see RLUXUT)       ++
!!      CALL RLUXUT(ISVEC)    outputs the current values of the 25  ++
!!                32-bit integer seeds, to be used for restarting  ++
!!      ISVEC must be dimensioned 25 in the calling program          ++
!! +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ ++
IMPLICIT NONE
! the REAL KIND
INTEGER, PARAMETER  :: rk = 4
INTEGER             :: isdext(25)
INTEGER, PARAMETER  :: maxlev = 4, lxdflt = 3, jsdflt = 314159265
INTEGER             :: ndskip(0:maxlev) = (/ 0, 24, 73, 199, 365 /)
INTEGER             :: igiga = 1000000000, i24 = 24, j24 = 10
REAL(KIND=rk), PARAMETER  :: twop12 = 4096.0_rk
INTEGER, PARAMETER  :: itwo24 = 2**24, icons = 2147483563
INTEGER, SAVE       :: next(24), luxlev = lxdflt, nskip, inseed, jseed
LOGICAL, SAVE       :: notyet = .true.
INTEGER             :: in24 = 0, kount = 0, mkount = 0
REAL(KIND=rk), SAVE :: seeds(24), carry = 0.0_rk, twom24, twom12
!                           default
!  Luxury Level     0   1   2  *3*    4
!    ndskip        /0, 24, 73, 199, 365/
! Corresponds to p=24  48  97  223  389
!     time factor   1  2   3    6   10   on slow workstation
!                   1 1.5  2    3    5   on fast mainframe
!                   1 1.5 2.5   5  8.5   on PC using LF90
PUBLIC notyet, i24, j24, carry, seeds, twom24, twom12, luxlev
PUBLIC nskip, ndskip, in24, next, kount, mkount, inseed

CONTAINS

SUBROUTINE ranlux(rvec, lenv)
IMPLICIT NONE
INTEGER, INTENT(IN)        :: lenv
REAL(KIND=rk), INTENT(OUT) :: rvec(lenv)
!     Local variables
```

```
INTEGER             :: i, k, lp, ivec, iseeds(24), isk
REAL(KIND=rk)       :: uni
! DEBUG
! print *, "DEBUG (SR ranlux):", lenv, " numbers drawn"
!  NOTYET is .TRUE. if no initialization has been performed yet.
!              Default Initialization by Multiplicative Congruential
IF (notyet) THEN
  notyet = .false.
  jseed = jsdflt
  inseed = jseed
  WRITE (6,'(A,I12)') ' RANLUX DEFAULT INITIALIZATION: ', jseed
  luxlev = lxdflt
  nskip = ndskip(luxlev)
  lp = nskip + 24
  in24 = 0
  kount = 0
  mkount = 0
  WRITE (6,'(A,I2,A,I4)') ' RANLUX DEFAULT LUXURY LEVEL = ', luxlev, ' p =', lp
  twom24 = 1.0_rk
  DO i = 1, 24
    twom24 = twom24 * 0.5_rk
    k = jseed / 53668
    jseed = 40014 * (jseed-k*53668) - k * 12211
    IF (jseed.LT.0) jseed = jseed + icons
    iseeds(i) = MOD(jseed,itwo24)
  END DO
  twom12 = twom24 * 4096.0_rk
  DO i = 1, 24
    seeds(i) = REAL(iseeds(i), KIND=rk) * twom24
    next(i) = i - 1
  END DO
  next(1) = 24
  i24 = 24
  j24 = 10
  carry = 0.0_rk
  IF (seeds(24).EQ.0.0_rk) carry = twom24
END IF
!          The Generator proper: "Subtract-with-borrow",
!          as proposed by Marsaglia and Zaman,
!          Florida State University, March, 1989
DO ivec = 1, lenv
  uni = seeds(j24) - seeds(i24) - carry
  IF (uni.LT.0.0_rk) THEN
    uni = uni + 1.0_rk
    carry = twom24
  ELSE
    carry = 0.0_rk
  END IF
  seeds(i24) = uni
  i24 = next(i24)
  j24 = next(j24)
  rvec(ivec) = uni
!  small numbers (with less than 12 "significant" bits) are "padded".
  IF (uni.LT.twom12) THEN
    rvec(ivec) = rvec(ivec) + twom24 * seeds(j24)
!        and zero is forbidden in case someone takes a logarithm
    IF (rvec(ivec).EQ.0.0_rk) rvec(ivec) = twom24 * twom24
  END IF
!         Skipping to luxury.  As proposed by Martin Luscher.
  in24 = in24 + 1
  IF (in24.EQ.24) THEN
    in24 = 0
    kount = kount + nskip
    DO isk = 1, nskip
      uni = seeds(j24) - seeds(i24) - carry
      IF (uni.LT.0.0_rk) THEN
        uni = uni + 1.0_rk
        carry = twom24
      ELSE
        carry = 0.0_rk
      END IF
      seeds(i24) = uni
      i24 = next(i24)
      j24 = next(j24)
    END DO
  END IF
END DO
kount = kount + lenv
IF (kount.GE.igiga) THEN
  mkount = mkount + 1
  kount = kount - igiga
END IF
RETURN
END SUBROUTINE ranlux

!           Subroutine to input and float integer seeds from previous run
SUBROUTINE rluxin
!     the following IF BLOCK added by Phillip Helbig, based on conversation
!     with Fred James; an equivalent correction has been published by James.
IMPLICIT NONE
!     Local variables
INTEGER             :: i, isd
IF (notyet) THEN
  WRITE (6,'(A)') ' Proper results ONLY with initialisation from 25 ',  &
                  'integers obtained with RLUXUT'
  notyet = .false.
END IF
twom24 = 1.0_rk
DO i = 1, 24
  next(i) = i - 1
  twom24 = twom24 * 0.5_rk
END DO
next(1) = 24
twom12 = twom24 * 4096.0_rk
WRITE (6,'(A)') ' FULL INITIALIZATION OF RANLUX WITH 25 INTEGERS:'
WRITE (6,'(5X,5I12)') isdext
DO i = 1, 24
  seeds(i) = REAL(isdext(i), KIND=rk) * twom24
END DO
carry = 0.0_rk
IF (isdext(25).LT.0) carry = twom24
isd = ABS(isdext(25))
i24 = MOD(isd,100)
isd = isd / 100
j24 = MOD(isd,100)
isd = isd / 100
in24 = MOD(isd,100)
isd = isd / 100
luxlev = isd
IF (luxlev.LE.maxlev) THEN
  nskip = ndskip(luxlev)
  WRITE (6,'(A,I2)') ' RANLUX LUXURY LEVEL SET BY RLUXIN TO: ', luxlev
ELSE IF (luxlev.GE.24) THEN
  nskip = luxlev - 24
  WRITE (6,'(A,I5)') ' RANLUX P-VALUE SET BY RLUXIN TO:', luxlev
ELSE
  nskip = ndskip(maxlev)
  WRITE (6,'(A,I5)') ' RANLUX ILLEGAL LUXURY RLUXIN: ', luxlev
  luxlev = maxlev
END IF
inseed = -1
RETURN
END SUBROUTINE rluxin

!                    Subroutine to ouput seeds as integers
SUBROUTINE rluxut
IMPLICIT NONE
!     Local variables
INTEGER             :: i
DO i = 1, 24
  isdext(i) = INT(seeds(i)*twop12*twop12)
END DO
isdext(25) = i24 + 100 * j24 + 10000 * in24 + 1000000 * luxlev
IF (carry.GT.0.0_rk) isdext(25) = -isdext(25)
RETURN
END SUBROUTINE rluxut

!                    Subroutine to output the "convenient" restart point
SUBROUTINE rluxat(lout, inout, k1, k2)
IMPLICIT NONE
INTEGER, INTENT(OUT) :: lout, inout, k1, k2
lout = luxlev
inout = inseed
k1 = kount
k2 = mkount
RETURN
END SUBROUTINE rluxat

!                    Subroutine to initialize from one or three integers
SUBROUTINE rluxgo(lux, ins, k1, k2)
IMPLICIT NONE
INTEGER, INTENT(IN) :: lux, ins, k1, k2
!     Local variables
INTEGER             :: ilx, i, iouter, iseeds(24), isk, k, inner, izip, izip2
REAL(KIND=rk)       :: uni
IF (lux.LT.0) THEN
  luxlev = lxdflt
ELSE IF (lux.LE.maxlev) THEN
  luxlev = lux
ELSE IF (lux.LT.24.OR.lux.GT.2000) THEN
  luxlev = maxlev
  WRITE (6,'(A,I7)') ' RANLUX ILLEGAL LUXURY RLUXGO: ', lux
ELSE
  luxlev = lux
  DO ilx = 0, maxlev
    IF (lux.EQ.ndskip(ilx)+24) luxlev = ilx
  END DO
END IF
IF (luxlev.LE.maxlev) THEN
  nskip = ndskip(luxlev)
  WRITE (6,'(A,I2,A,I4)') ' RANLUX LUXURY LEVEL SET BY RLUXGO :', luxlev,  &
                          '     P=', nskip + 24
ELSE
  nskip = luxlev - 24
  WRITE (6,'(A,I5)') ' RANLUX P-VALUE SET BY RLUXGO TO:', luxlev
END IF
in24 = 0
IF (ins.LT.0) WRITE (6,'(A)') &
              ' Illegal initialization by RLUXGO, negative input seed'
IF (ins.GT.0) THEN
  jseed = ins
  WRITE (6,'(A,3I12)') ' RANLUX INITIALIZED BY RLUXGO FROM SEEDS', jseed, k1, k2
ELSE
  jseed = jsdflt
  WRITE (6,'(A)') ' RANLUX INITIALIZED BY RLUXGO FROM DEFAULT SEED'
END IF
inseed = jseed
notyet = .false.
twom24 = 1.0_rk
DO i = 1, 24
  twom24 = twom24 * 0.5_rk
  k = jseed / 53668
  jseed = 40014 * (jseed-k*53668) - k * 12211
  IF (jseed.LT.0) jseed = jseed + icons
  iseeds(i) = MOD(jseed,itwo24)
END DO
twom12 = twom24 * 4096.0_rk
DO i = 1, 24
  seeds(i) = REAL(iseeds(i), KIND=rk) * twom24
  next(i) = i - 1
END DO
next(1) = 24
i24 = 24
j24 = 10
carry = 0.0_rk
IF (seeds(24).EQ.0.0_rk) carry = twom24
!        If restarting at a break point, skip K1 + IGIGA*K2
!        Note that this is the number of numbers delivered to
!        the user PLUS the number skipped (if luxury .GT. 0).
kount = k1
mkount = k2
IF (k1+k2.NE.0) THEN
  DO iouter = 1, k2 + 1
    inner = igiga
    IF (iouter.EQ.k2+1) inner = k1
    DO isk = 1, inner
      uni = seeds(j24) - seeds(i24) - carry
      IF (uni.LT.0.) THEN
        uni = uni + 1.0_rk
        carry = twom24
      ELSE
        carry = 0.0_rk
      END IF
      seeds(i24) = uni
      i24 = next(i24)
      j24 = next(j24)
    END DO
  END DO
!          Get the right value of IN24 by direct calculation
  in24 = MOD(kount,nskip+24)
  IF (mkount.GT.0) THEN
    izip = MOD(igiga, nskip+24)
    izip2 = mkount * izip + in24
    in24 = MOD(izip2, nskip+24)
  END IF
!          Now IN24 had better be between zero and 23 inclusive
  IF (in24.GT.23) THEN
    WRITE (6,'(A/A,3I11,A,I5)') &
                '  Error in RESTARTING with RLUXGO:', '  The values', ins, &
              k1, k2, ' cannot occur at luxury level', luxlev
    in24 = 0
  END IF
END IF
RETURN
END SUBROUTINE rluxgo
END MODULE luxury
```

utilitiesV1.9.f90

```
! module with some helper functions:
!-------------------------------------
! * reading and writing of configurations
! * state of the simulation
! * inversion of force ramp and low pass filtering of variables
! * everything else...
! Martin Aichele, 2001-12-12
! last modified 2003-02-24
! V1.9 with real dimension switch
! Martin Aichele, 2003-02-24
! last modified 2003-02-28
module utilities
  use globals
  use luxury, only: rluxut, rluxin, isdext
  implicit none
  !-----------------------------------------------------------------------------
  ! debug switches      |-------- 31 characters ------|
  !-----------------------------------------------------------------------------
  logical, parameter :: l_debug_control                = .FALSE.
  logical, parameter :: l_debug_chain_sens_fold        = .FALSE.
  logical, parameter :: l_debug_fluid_positions_out    = .FALSE.
  logical, parameter :: l_debug_particle_positions_out  = .FALSE.
  !-----------------------------------------------------------------------------
  !-----------------------------------------------------------------------------
  ! output units
  integer, parameter :: pos_vel_out_unit               = 50
  integer, parameter :: forces_out_unit                = 51
  integer, parameter :: energy_out_unit                = 52
  !-----------------------------------------------------------------------------

  ! control verbosity of SR chain_sens_fold
  logical, parameter :: l_chain_sens_fold_verbose      = .FALSE.

  ! sample times
  integer, save :: nr_samples_total = huge(1)
  integer, save :: next_sample_time = huge(1), next_sample_index = huge(1)
  integer, dimension(:), pointer, save :: sample_times

  real(kind=dp), dimension(n_dim), save :: cm_fluid_1
contains
  !-----------------------------------------------------------------------------
  ! write out the state of the simulation and some information
  subroutine simulationstate
    write(unit=*, fmt='(a, i10)') &
         "DEBUG: Now doing MD step", i_time
    write(unit=*, fmt='(a, g13.6, a, g13.6)') &
         "       temp=", temp, " r_2_min=", r_2_min
    if(n_dim .eq. n_dim_pbc+1) then
      write(unit=*, fmt='(a, g13.6)') &
           "       r0_twall(n_dim) - r0_bwall(n_dim)=", &
           r0_twall(n_dim) - r0_bwall(n_dim)
    end if
    write(unit=*, fmt='(a)') "-----------------------------------------&
        &-------------------------------"
  end subroutine simulationstate
  !-----------------------------------------------------------------------------

  !-----------------------------------------------------------------------------
  ! records low pass filtered variables, controls invertion of the force ramp.
  subroutine control
    ! logical value for deciding if a warning is the first after some MD steps
    ! without warning about a high sliding velocity
    logical, save :: l_first_sliding_vel_warning = .TRUE.
    integer :: i_dim
    character(len=80) :: format_string
    if(l_debug_control) then
      print *, "DEBUG: Entering SR control"
    end if
    ! observe position, velocity and force of top wall with "low pass" filter
    ! over the last n_time_ave MDS
    do i_dim = 1, n_dim
      r0_twall_1(i_dim) = &
           (1.0_dp-1.0_dp/real(n_time_ave, kind=dp))*r0_twall_1(i_dim) &
           + (1.0_dp / real(n_time_ave, kind=dp))*r0_twall(i_dim)
      velocity_twall_1(i_dim) = &
           (1.0_dp-1.0_dp/real(n_time_ave, kind=dp))*velocity_twall_1(i_dim) &
           + (1.0_dp / real(n_time_ave, kind=dp))*rx_twall(i_dim,1)
      if(l_compute_cm_fluid) then
        cm_fluid_1(i_dim) = &
             (1.0_dp-1.0_dp/real(n_time_ave, kind=dp)) * cm_fluid_1(i_dim) &
             + (1.0_dp / real(n_time_ave, kind=dp)) &
             * sum(r0_unfolded(1:n_mon_tot,i_dim))
      end if
      select case(f_twall(i_dim))
      case(force_mode)
        ext_force_twall_1(i_dim) = &
             (1.0_dp-1.0_dp/real(n_time_ave,kind=dp))*ext_force_twall_1(i_dim)&
             + (1.0_dp / real(n_time_ave, kind=dp))*ext_force_twall(i_dim)
      case(spring_mode)
        ext_force_twall_1(i_dim) = &
             (1.0_dp-1.0_dp/real(n_time_ave,kind=dp))*ext_force_twall_1(i_dim)&
             + (1.0_dp / real(n_time_ave, kind=dp))*ext_force_twall(i_dim)
      end select
      total_force_twall_1(i_dim) = &
          (1.0_dp-1.0_dp/real(n_time_ave,kind=dp))*total_force_twall_1(i_dim) &
           + (1.0_dp / real(n_time_ave, kind=dp))*total_force_twall(i_dim)
    end do
    ! average positions
    ! this bit of code does not really belong here...
!    r0_ave(:, :) = (1.0_dp-1.0_dp/real(n_time_ave,kind=dp))*r0_ave(:, :) &
!         + (1.0_dp / real(n_time_ave, kind=dp))*r0(:, :)
    ! write out every n_time_ave time steps
    ! For the displacements we want a fixed absolute precision
    if(mod(i_time, n_time_ave).eq.0) then
      ! top wall position and force are only written when the two walls are
      ! actually simulated
      if(n_dim .eq. n_dim_pbc+1) then
        ! position and velocity of top wall
        if(.not. l_compute_cm_fluid) then
          write(format_string, fmt='(a,i1,a,i1,a)') &
               "(es17.11, ", n_dim, "f12.4, ", n_dim, "g12.4)"
          write(pos_vel_out_unit, fmt=format_string) &
               r_time, (r0_twall_1(i_dim), i_dim=1, n_dim), &
               (velocity_twall_1(i_dim)/dt, i_dim=1, n_dim)
        else
          ! write out center of mass of fluid in last n_dim columns
          write(format_string, fmt='(a,i1,a,i1,a,i1,a)') &
               "(es17.11, ",n_dim,"f12.4, ",n_dim,"g12.4, ",n_dim,"f12.4)"
          write(pos_vel_out_unit, fmt=format_string) &
               r_time, (r0_twall_1(i_dim), i_dim=1, n_dim), &
               (velocity_twall_1(i_dim)/dt, i_dim=1, n_dim), &
               (cm_fluid_1(i_dim)/real(n_mon_tot, kind=dp), i_dim=1, n_dim)
        end if
        ! write out total_force_twall normalized to the number of top wall
        ! particles
        write(format_string, fmt='(a,i1,a)') "(es17.11, ", 2*n_dim, "g12.4)"
        write(forces_out_unit, fmt=format_string) r_time, &
             (ext_force_twall_1(i_dim), i_dim=1, n_dim), &
             (total_force_twall_1(i_dim)/real(n_top_wall,kind=dp), &
             i_dim=1, n_dim)
      end if ! walls
    end if ! time to write out

    ! If we are in the mode with a ramped force, we need to keep
    ! track of the time when the force last went through zero.
    ! This point of time is used to define the position where the
    ! system was pinned last. (Of course, this is only an approximation.)
    ! If the system has moved more than 10 lattice spacings since
    ! this last pinning, one can usually safely assume that no repinning
    ! will occur and the ramp is inverted. The force is ramped down four
    ! times as fast as the ramp is moved up. (Good empirical value.)
    ! Inversion of the force ramp is done only in x,y, not in z.
    do i_dim = 1, n_dim_pbc
      if((f_twall(i_dim).eq.force_mode) &
           .and.(ramp_force_twall(i_dim).ne.0.0_dp)) then
        ! keep track of last pinned top wall coordinate
        ! define position as pinned when force goes through zero.
        ! (when it is less or equal the value after 1 dt)
        if(abs(ext_force_twall(i_dim)) .le. abs(ramp_force_twall(i_dim))) then
          r0_last_pinned(i_dim) = r0_twall(i_dim)
        end if
        ! If top wall has moved more than 10 lattice spacings invert ramp
        ! The max() is there for having always 10 lattice spacings, regardless
        ! of wall orientation and dimension
        if( (r0_twall(i_dim)-r0_last_pinned(i_dim))*s_force_grad(i_dim) &
             .gt. 10.0_dp * max(x_space, y_space)) then
          s_force_grad(i_dim) = -4.0_dp * s_force_grad(i_dim)
        end if
        ! Here we limit the velocity to max_sliding_vel Lennard Jones units
        ! per time unit.
        ! Otherwise, we will easily reach sliding speeds in the order of
        ! the speed of sound.
        if ( abs(rx_twall(i_dim, 1))/dt .gt. max_sliding_vel ) then
          rx_twall(i_dim, 1) = max_sliding_vel &
               * dt * rx_twall(i_dim, 1)/abs(rx_twall(i_dim, 1))
          if(l_first_sliding_vel_warning) then
            l_first_sliding_vel_warning = .FALSE.
            write(unit=*, fmt='(a, i1, a, g13.6, a, es17.11)') &
                 "CAUTION (SR control): rx_twall(", i_dim, &
                 ", 1)/dt > max_sliding_vel=", max_sliding_vel, &
                 " at r_time=", r_time
          end if
        else
          ! if the sliding velocity was smaller than the threshold, then
          ! the next warning will belong to a new sliding process
          l_first_sliding_vel_warning = .TRUE.
        end if
      end if ! if in ramped force mode
    end do
    if(l_debug_control) then
      print *, "DEBUG: Leaving SR control"
    end if
  end subroutine control
  !-----------------------------------------------------------------------------

  !-----------------------------------------------------------------------------
  ! subroutine for the computation and output of wall-wall potential
  subroutine wall_wall_quantities
    ! routine is absorbed in SR energy
    write(unit=*, fmt='(a)') &
         "ERROR (SR wall_wall_quantities): This SR is absorbed in SR energy."
    stop
    if(i_time .eq. s_time +1) then
      simulated_wall_wall_potential_1 = v_wall_wall
    else
      simulated_wall_wall_potential_1 = &
           (1.0_dp-1.0_dp/real(n_time_ave,kind=dp)) &
           * simulated_wall_wall_potential_1 &
           + (1.0_dp / real(n_time_ave, kind=dp))*v_wall_wall
    end if
    if(mod(i_time, n_time_ave) .eq. 0) write(53, fmt='(es17.11,g12.5)') &
         r_time, simulated_wall_wall_potential_1
  end subroutine wall_wall_quantities
  !-----------------------------------------------------------------------------

  !-----------------------------------------------------------------------------
  ! stores a configuration (new version, higher precision and saves state of
  ! the random number generator)
  subroutine store_configuration_rng(new_configuration_file, l_write_unfolded)
    character(len=*), intent(in) :: new_configuration_file
    ! if folded or unfolded particle positions are written
    logical, intent(in) :: l_write_unfolded
    integer, parameter :: fileunit = 10
    integer :: iostatus
    integer :: i_dim
    real(kind=dp) :: r_dummy
    character(len=80) :: format_string
    integer :: i_wall  ! loop over wall particles
    integer :: i_part  ! loop over particles
    integer :: i_order ! loop over predictor-corrector coefficients
    ! note on output field width and precision:
    ! the maximal kind=4 integer is 2147483647 (at least on IA32 systems)
    ! so i12 (for a space and an minus sign is sufficient)
    ! We demand 15 digits precision for our real kind, so this is also the
    ! precision we have to write out.  Using a higher precision is better for
    ! smooth continuation runs.
    ! open file for replacement
    open(unit=fileunit, file=new_configuration_file, status="unknown", &
         action="write", iostat=iostatus)
    if(iostatus .ne. 0) then
      write (unit=*, fmt=*) "WARNING: Could not open output file >>", &
           new_configuration_file, "<<"
    else
      write (unit=*, fmt='(3a, i12)') &
           "MESSAGE: Writing configuration to >>", new_configuration_file, &
           "<< at MDS", i_time
      write(unit=fileunit, fmt='(2i12)') n_mon, n_chain
      write(unit=fileunit, fmt='(2i12)') n_cell_w_x, n_cell_w_y
      write(unit=fileunit, fmt='(i3)')   n_order
      write(format_string, fmt='(a,i1,a)') "(i3, ", n_dim, "g23.15)"
      ! particle positions
      if(l_write_unfolded) then
        do i_part = 1, n_part
          write(unit=fileunit, fmt=format_string) type(i_part), &
               (r0(i_part, i_dim) + real(pbc_count(i_part,i_dim),kind=dp)&
               * boundary(i_dim), i_dim=1, n_dim)
        end do
      else
        do i_part = 1, n_part
          write(unit=fileunit, fmt=format_string) &
               type(i_part), (r0(i_part, i_dim), i_dim=1, n_dim)
        end do
      end if
      ! derivatives
      write(format_string, fmt='(a,i1,a)') "(", n_dim, "g23.15)"
      do i_order = 1, n_order
        ! write out coefficients without potencies of dt
        r_dummy = dt**i_order
        do i_part = 1, n_part
          write(unit=fileunit, fmt=format_string) &
               (rx(i_part,i_dim,i_order)/r_dummy, i_dim=1, n_dim)
        end do
      end do
      ! random forces from previous MD step
```

```
    do i_part = 1, n_part
      write(unit=fileunit, fmt=format_string) &
           (force_random(i_part, i_dim), i_dim=1, n_dim)
    end do
    ! equilibrium positions
    if(l_write_unfolded) then
      do i_wall = 1, n_wall
        i_part = i_wall + n_mon_tot
                 write(unit=fileunit, fmt=format_string) &
              (r_wall_equi(i_wall, i_dim) &
              + real(pbc_count(i_part,i_dim),kind=dp)*boundary(i_dim), &
              i_dim=1, n_dim)
      end do
    else
      do i_wall = 1, n_wall
        write(unit=fileunit, fmt=format_string) &
              (r_wall_equi(i_wall,i_dim),i_dim=1,n_dim)
      end do
    end if
    ! position of top wall
    write(unit=fileunit, fmt=format_string) (r0_twall(i_dim), i_dim=1, n_dim)
    ! divide by dt**i_order, so that we can continue the simulation with
    ! another timestep
    do i_order = 1,n_order
      r_dummy = dt**i_order
      write(unit=fileunit, fmt=format_string) &
           (rx_twall(i_dim,i_order)/r_dummy,i_dim=1,n_dim)
    end do
    write(format_string, fmt='(a,i1,a)') "(", n_dim, "i12)"
    write(unit=fileunit, fmt=format_string) (pbc_twall(i_dim),i_dim=1,n_dim)
    write(format_string, fmt='(a,i1,a)') "(", n_dim, "g23.15)"
    write(unit=fileunit, fmt=format_string) (r0_bwall(i_dim),i_dim=1,n_dim)
    write(format_string, fmt='(a,i1,a)') "(", n_dim, "i12)"
    write(unit=fileunit, fmt=format_string) (pbc_bwall(i_dim),i_dim=1,n_dim)
    write(format_string, fmt='(a,i1,a)') "(", n_dim, "g23.15)"
    write(unit=fileunit, fmt=format_string) &
         (r0_spring_twall(i_dim),i_dim=1,n_dim)
    write(format_string, fmt='(a,i1,a)') "(", n_dim, "i12)"
    write(unit=fileunit, fmt=format_string) &
         (s_force_grad(i_dim),i_dim=1,n_dim)
    ! write out state of the random number generator
    ! isdext is declared in module luxury
    call rluxut
    write(unit=fileunit, fmt='(6i12)') isdext(1:6)
    write(unit=fileunit, fmt='(6i12)') isdext(7:12)
    write(unit=fileunit, fmt='(6i12)') isdext(13:18)
    write(unit=fileunit, fmt='(6i12)') isdext(19:24)
    write(unit=fileunit, fmt='(i12)')  isdext(25)
    ! write out MD steps counter
    write(unit=fileunit, fmt='(a, i12, a, i12)') &
         "s_time=", s_time, " i_time=", i_time
  end if ! iostatus .ne. 0
  close(fileunit)
end subroutine store_configuration_rng
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! reads a configuration and performs some checks for consistency
subroutine read_configuration_rng(old_configuration_file)
  character(len=*), intent(in) :: old_configuration_file
  integer, parameter :: fileunit = 10
  integer :: iostatus
  ! loop variables
  integer :: i_dim
  integer :: i_mon, i_chain, i_cell_w_x, i_cell_w_y, i_wall, i_part
  ! loops over predictor-corrector coefficients
  integer :: i_order, j_order
  ! how many times the positions are shifted by a boundary
  integer :: pbc_shift
  character(len=80) :: format_string
  ! helper variable
  real(kind=dp) :: r_dummy
  ! open file for replacement
  open(unit=fileunit, file=old_configuration_file, status="old", &
       action="read", iostat=iostatus)
  if(iostatus .ne. 0) then
    write (unit=*, fmt='(3a)') &
         "ERROR (SR read_configuration_rng) : Could not read old &
         &configuration >>", old_configuration_file, "<<"
    stop
  else
    write(unit=*, fmt='(3a)') &
         "MESSAGE: Reading stored configuration in >>", &
         old_configuration_file, "<<"
    read(unit=fileunit, fmt='(2i12)') i_mon, i_chain
    read(unit=fileunit, fmt='(2i12)') i_cell_w_x, i_cell_w_y
    read(unit=fileunit, fmt='(i3)')   j_order
    ! check whether number of monomers right
    if((i_mon.ne.n_mon).or.(i_chain.ne.n_chain)) then
      write(unit=*, fmt='(a)') "WARNING (SR read_configuration): &
           &Polymer setups might not be compatible:"
      write(unit=*, fmt='(4(a, i6))') &
           "     i_mon=", i_mon, ", n_mon=", n_mon, &
           ", i_chain=", i_chain, ", n_chain=", n_chain
      if((i_mon*i_chain).ne.(n_mon*n_chain)) then
        write(unit=*, fmt='(a)') &
             "ERROR (SR read_configuration): Polymer setups not compatible:"
        write(unit=*, fmt='(2(a, i8))') &
             "     i_mon*i_chain=", i_mon*i_chain, &
             ", n_mon*n_chain=", n_mon*n_chain
        stop
      end if
    end if
    ! check whether number of wall unit cells right
    if((i_cell_w_x.ne.n_cell_w_x).or.(i_cell_w_y.ne.n_cell_w_y)) then
      write(unit=*, fmt='(a)') &
           "WARNING (SR read_configuration_rng): &
           &Set-up of wall might not be compatible"
      if((i_cell_w_x*i_cell_w_y).ne.(n_cell_w_x*n_cell_w_y)) then
        write(unit=*, fmt='(a)') &
             "ERROR (SR read_configuration_rng): &
             &Set-up of wall is not compatible:"
        write(unit=*, fmt='(a,i3,a,i3)') &
             "i_cell_w_x =", i_cell_w_x, " i_cell_w_y=", i_cell_w_y
        write(unit=*, fmt='(a,i3,a,i3)') &
             "n_cell_w_x =", n_cell_w_x, " n_cell_w_y=", n_cell_w_y
        stop
      end if
    end if
    ! check order of algorithm
    if(j_order.ne.n_order) then
      write(unit=*, fmt='(a, i2)') &
           "WARNING (SR read_configuration_rng): Order of old run", j_order
      write(unit=*, fmt='(a, i2)') &
           "WARNING (SR read_configuration_rng): Order of new run", n_order
      if(n_order.lt.j_order) then
        write(unit=*, fmt='(a)') "ERROR (SR read_configuration_rng): &
             &Order of new run < order of old run."
        write(unit=*, fmt='(a)') &
             "It is only possible to propagate a configuration with equal &
             &or higher order."
        stop
      end if
    end if
    ! read positions
    ! since version V1.7 we store the position in the canonical way: x, y, z
    write(format_string, fmt='(a,i1,a)') "(i3,", n_dim, "g23.15)"
    do i_part = 1, n_part
      read(unit=fileunit, fmt=format_string) &
           type(i_part), (r0(i_part, i_dim), i_dim=1, n_dim)
    end do
    ! read derivatives
    write(format_string, fmt='(a,i1,a)') "(", n_dim, "g23.15)"
    do i_order = 1, j_order
      r_dummy = dt**i_order ! scale coefficients with potencies of dt
      do i_part = 1, n_part
        read(unit=fileunit, fmt=format_string) &
             (rx(i_part,i_dim,i_order), i_dim=1, n_dim)
        do i_dim = 1,n_dim
          rx(i_part,i_dim,i_order) = rx(i_part,i_dim,i_order)*r_dummy
        end do
      end do
    end do
    ! zero out derivatives of higher order
    do i_order = j_order, n_order
      rx(:, :, i_order) = 0.0_dp
    end do
    ! random forces from previous MD step
    do i_part = 1, n_part
      read(unit=fileunit, fmt=format_string) &
           (force_random(i_part, i_dim), i_dim=1, n_dim)
    end do
    ! equilibrium sites of wall atoms
    do i_wall = 1, n_wall
      read(unit=fileunit, fmt=format_string) &
           (r_wall_equi(i_wall,i_dim), i_dim=1, n_dim)
    end do
    ! displacement of top wall and its derivatives
    read(unit=fileunit, fmt=format_string) &
         (r0_twall(i_dim),i_dim=1,n_dim)
    do i_order = 1,j_order
      r_dummy = dt**i_order
      read(unit=fileunit,fmt=format_string) &
           (rx_twall(i_dim,i_order),i_dim=1,n_dim)
      do i_dim = 1,n_dim
        rx_twall(i_dim,i_order) = rx_twall(i_dim,i_order)*r_dummy
      end do
    end do
    ! zero out derivatives of higher order
    do i_order = j_order, n_order
      rx_twall(:, i_order) = 0.0_dp
    end do
    ! periodic boundary conditions tracking for top wall
    write(format_string, fmt='(a,i1,a)') "(", n_dim, "i12)"
    read(unit=fileunit, fmt=format_string) (pbc_twall(i_dim),i_dim=1,n_dim)
    ! displacement of bottom wall (at present always (0, 0, 0))
    write(format_string, fmt='(a,i1,a)') "(", n_dim, "g23.15)"
    read(unit=fileunit, fmt=format_string) &
         (r0_bwall(i_dim),i_dim=1,n_dim)
    ! periodic boundary conditions tracking for bottom wall
    write(format_string, fmt='(a,i1,a)') "(", n_dim, "i12)"
    read(unit=fileunit, fmt=format_string) (pbc_bwall(i_dim),i_dim=1,n_dim)
    ! the position of the external spring
    write(format_string, fmt='(a,i1,a)') "(", n_dim, "g23.15)"
    read(unit=fileunit,  fmt=format_string) &
         (r0_spring_twall(i_dim),i_dim=1,n_dim)
    ! the status of the force ramp
    write(format_string, fmt='(a,i1,a)') "(", n_dim, "i12)"
    read(unit=fileunit, fmt=format_string)(s_force_grad(i_dim),i_dim=1,n_dim)
    ! here we have the coice if we restart the random number generator
    ! from the configuration or not (seed in the parameter file is then used)
    if(l_read_conf_rng_reinit) then
      ! read state of random number generator
      read(unit=fileunit, fmt='(6i12)') isdext(1:6)
      read(unit=fileunit, fmt='(6i12)') isdext(7:12)
      read(unit=fileunit, fmt='(6i12)') isdext(13:18)
      read(unit=fileunit, fmt='(6i12)') isdext(19:24)
      read(unit=fileunit, fmt='(i12)')  isdext(25)
      call rluxin
    end if
    close(fileunit)
  end if ! configuration file could be read

  ! fold coordinates back in simulation box in case the saved coordinates
  ! were world (unfolded) coordinates
  !-----------------------------------------------------------------------
  ! impose periodic boundary conditions for fluid particles in xy
  do i_dim = 1, n_dim_pbc
    do i_part = 1, n_mon_tot
      pbc_shift = int(r0(i_part,i_dim) / boundary(i_dim))
      pbc_count(i_part,i_dim) = pbc_shift
      r0(i_part,i_dim) = r0(i_part,i_dim) &
           - boundary(i_dim) * real(pbc_shift, kind=dp)
      if(r0(i_part, i_dim).lt.0.0_dp) then
        r0(i_part,i_dim) = r0(i_part,i_dim) + boundary(i_dim)
        pbc_count(i_part,i_dim) = pbc_count(i_part,i_dim) - 1
      end if
    end do
  end do
  ! impose periodic boundary conditions for wall particles
  do i_dim = 1, n_dim_pbc
    do i_wall = 1, n_wall
      i_part = i_wall + n_mon_tot
      pbc_shift = int(r0(i_part,i_dim) / boundary(i_dim))
      pbc_count(i_part,i_dim) = pbc_shift
      r0(i_part,i_dim) = r0(i_part,i_dim) &
           - boundary(i_dim) * real(pbc_shift, kind=dp)
      ! do the same for the equilibrium positions
      r_wall_equi(i_wall,i_dim) = r_wall_equi(i_wall,i_dim) &
           - boundary(i_dim) * real(pbc_shift, kind=dp)
      if(r0(i_part, i_dim).lt.0.0_dp) then
        r0(i_part,i_dim) = r0(i_part,i_dim) + boundary(i_dim)
        r_wall_equi(i_wall,i_dim) = &
             r_wall_equi(i_wall,i_dim) + boundary(i_dim)
        pbc_count(i_part,i_dim) = pbc_count(i_part,i_dim) - 1
      end if
    end do
  end do
  ! copy values on pbc_count_old(:, :)
  pbc_count_old(:, :) = pbc_count(:, :)
  ! define unfolded coordinates
  do i_dim = 1, n_dim
    r0_unfolded(:, i_dim) = r0(:, i_dim) &
         + real(pbc_count(:, i_dim), kind=dp)*boundary(i_dim)
  end do
end subroutine read_configuration_rng
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! reads a configuration and performs some checks for consistency
! compatibility routine for versions before V1.9
subroutine read_configuration_rng_3d(old_configuration_file)
```

```
  character(len=*), intent(in) :: old_configuration_file
  integer, parameter :: fileunit = 10
  integer :: iostatus
  ! loop variables
  integer :: i_dim
  integer :: i_mon, i_chain, i_cell_w_x, i_cell_w_y, i_wall, i_part
  ! loops over predictor-corrector coefficients
  integer :: i_order, j_order
  ! how many times the positions are shifted by a boundary
  integer :: pbc_shift
  ! helper variable
  real(kind=dp) :: r_dummy
  ! only in this case we can use this configuration read routine
  if(.not. (n_dim .eq. 3 .and. n_dim_pbc .eq. 2)) then
    write(unit=*, fmt='(a)') "ERROR (SR read_configuration_rng_3d): &
         &.not. (n_dim .eq. 3 .and. n_dim_pbc .eq. 2)"
    stop
  end if
  ! open file for replacement
  open(unit=fileunit, file=old_configuration_file, status="old", &
       action="read", iostat=iostatus)
  if(iostatus .ne. 0) then
    write (unit=*, fmt='(3a)') &
         "ERROR (SR read_configuration_rng_3d) : Could not read old &
         &configuration >>", old_configuration_file, "<<"
    stop
  else
    write(unit=*, fmt='(3a)') &
         "MESSAGE: Reading stored configuration in >>", &
         old_configuration_file, "<<"
    read(unit=fileunit, fmt='(2i12)') i_mon, i_chain
    read(unit=fileunit, fmt='(2i12)') i_cell_w_x, i_cell_w_y
    read(unit=fileunit, fmt='(i3)')   j_order
    ! check whether number of monomers right
    if((i_mon.ne.n_mon).or.(i_chain.ne.n_chain)) then
      write(unit=*, fmt='(a)') "WARNING (SR read_configuration): &
           &Polymer setups might not be compatible:"
      write(unit=*, fmt='(4(a, i6))') &
           "          i_mon=", i_mon, ", n_mon=", n_mon, &
           ", i_chain=", i_chain, ", n_chain=", n_chain
      if((i_mon*i_chain).ne.(n_mon*n_chain)) then
        write(unit=*, fmt='(a)') &
             "ERROR (SR read_configuration): Polymer setups not compatible:"
        write(unit=*, fmt='2(a, i8))') &
             "      i_mon*i_chain=", i_mon*i_chain, &
             ", n_mon*n_chain=", n_mon*n_chain
        stop
      end if
    end if
    ! check whether number of wall unit cells right
    if((i_cell_w_x.ne.n_cell_w_x).or.(i_cell_w_y.ne.n_cell_w_y)) then
      write(unit=*, fmt='(a)') &
           "WARNING (SR read_configuration_rng_3d): &
           &Set-up of wall might not be compatible"
      if((i_cell_w_x*i_cell_w_y).ne.(n_cell_w_x*n_cell_w_y)) then
        write(unit=*, fmt='(a)') &
             "ERROR (SR read_configuration_rng_3d): &
             &Set-up of wall is not compatible:"
        write(unit=*, fmt='(a,i3,a,i3)') &
             "i_cell_w_x =", i_cell_w_x, " i_cell_w_y=", i_cell_w_y
        write(unit=*, fmt='(a,i3,a,i3)') &
             "n_cell_w_x =", n_cell_w_x, " n_cell_w_y=", n_cell_w_y
        stop
      end if
    end if
    ! check order of algorithm
    if(j_order.ne.n_order) then
      write(unit=*, fmt='(a, i2)') &
           "WARNING (SR read_configuration_rng_3d): Order of old run",j_order
      write(unit=*, fmt='(a, i2)') &
           "WARNING (SR read_configuration_rng_3d): Order of new run",n_order
      if(n_order.lt.j_order) then
        write(unit=*, fmt='(a)') "ERROR (SR read_configuration_rng_3d): &
             &Order of new run < order of old run."
        write(unit=*, fmt='(a)') &
             "It is only possible to propagate a configuration with equal &
             &or higher order."
        stop
      end if
    end if
    ! read positions
    ! since version V1.7 we store the position in the canonical way: x, y, z
    do i_part = 1, n_part
      read(unit=fileunit, fmt='(i3, 3g26.18)') &
           type(i_part), (r0(i_part, i_dim), i_dim=1, n_dim)
    end do
    ! read derivatives
    do i_order = 1, j_order
      r_dummy = dt**i_order ! scale coefficients with potencies of dt
      do i_part = 1, n_part
        read(unit=fileunit, fmt='(3g26.18)') &
             (rx(i_part,i_dim,i_order), i_dim=1, n_dim)
        do i_dim = 1,n_dim
          rx(i_part,i_dim,i_order) = rx(i_part,i_dim,i_order)*r_dummy
        end do
      end do
    end do
    ! zero out derivatives of higher order
    do i_order = j_order, n_order
      rx(:, :, i_order) = 0.0_dp
    end do
    ! random forces from previous MD step
    do i_part = 1, n_part
      read(unit=fileunit, fmt='(3g26.18)') &
           (force_random(i_part, i_dim), i_dim=1, n_dim)
    end do
    ! equilibrium sites of wall atoms
    do i_wall = 1, n_wall
      read(unit=fileunit, fmt='(3g26.18)') &
           (r_wall_equi(i_wall,i_dim), i_dim=1, n_dim)
    end do
    ! displacement of top wall and its derivatives
    read(unit=fileunit, fmt='(3g26.18)') &
         (r0_twall(i_dim),i_dim=1,n_dim)
    do i_order = 1,j_order
      r_dummy = dt**i_order
      read(unit=fileunit,fmt='(3g26.18)') &
           (rx_twall(i_dim,i_order),i_dim=1,n_dim)
      do i_dim = 1,n_dim
        rx_twall(i_dim,i_order) = rx_twall(i_dim,i_order)*r_dummy
      end do
    end do
    ! zero out derivatives of higher order
    do i_order = j_order, n_order
      rx_twall(:, i_order) = 0.0_dp
    end do
    ! periodic boundary conditions tracking for top wall
    read(unit=fileunit, fmt='(3i12)') (pbc_twall(i_dim),i_dim=1,n_dim)
    ! displacement of bottom wall (at present always (0, 0, 0))
    read(unit=fileunit, fmt='(3g26.18)') &
         (r0_bwall(i_dim),i_dim=1,n_dim)
    ! periodic boundary conditions tracking for bottom wall
    read(unit=fileunit, fmt='(3i12)') (pbc_bwall(i_dim),i_dim=1,n_dim)
    ! the position of the external spring
    read(unit=fileunit,  fmt='(3g26.18)') &
         (r0_spring_twall(i_dim),i_dim=1,n_dim)
    ! the status of the force ramp
    read(unit=fileunit, fmt='(3i12)') (s_force_grad(i_dim),i_dim=1,n_dim)
    ! here we have the coice if we restart the random number generator
    ! from the configuration or not (seed in the parameter file is then used)
    if(l_read_conf_rng_reinit) then
      ! read state of random number generator
      read(unit=fileunit, fmt='(6i12)') isdext(1:6)
      read(unit=fileunit, fmt='(6i12)') isdext(7:12)
      read(unit=fileunit, fmt='(6i12)') isdext(13:18)
      read(unit=fileunit, fmt='(6i12)') isdext(19:24)
      read(unit=fileunit, fmt='(i12)')  isdext(25)
      call rluxin
    end if
    close(fileunit)
  end if ! configuration file could be read

  ! fold coordinates back in simulation box in case the saved coordinates
  ! were world (unfolded) coordinates
  !-----------------------------------------------------------------------
  ! impose periodic boundary conditions for fluid particles in xy
  do i_dim = 1, n_dim-1
    do i_part = 1, n_mon_tot
      pbc_shift = int(r0(i_part,i_dim) / boundary(i_dim))
      pbc_count(i_part,i_dim) = pbc_shift
      r0(i_part,i_dim) = r0(i_part,i_dim) &
           - boundary(i_dim) * real(pbc_shift, kind=dp)
      if(r0(i_part, i_dim).lt.0.0_dp) then
        r0(i_part,i_dim) = r0(i_part,i_dim) + boundary(i_dim)
        pbc_count(i_part,i_dim) = pbc_count(i_part,i_dim) - 1
      end if
    end do
  end do
  ! impose periodic boundary conditions for wall particles
  do i_dim = 1, n_dim-1
    do i_wall = 1, n_wall
      i_part = i_wall + n_mon_tot
      pbc_shift = int(r0(i_part,i_dim) / boundary(i_dim))
      pbc_count(i_part,i_dim) = pbc_shift
      r0(i_part,i_dim) = r0(i_part,i_dim) &
           - boundary(i_dim) * real(pbc_shift, kind=dp)
      ! do the same for the equilibrium positions
      r_wall_equi(i_wall,i_dim) = r_wall_equi(i_wall,i_dim) &
           - boundary(i_dim) * real(pbc_shift, kind=dp)
      if(r0(i_part, i_dim).lt.0.0_dp) then
        r0(i_part,i_dim) = r0(i_part,i_dim) + boundary(i_dim)
        r_wall_equi(i_wall,i_dim) = &
             r_wall_equi(i_wall,i_dim) + boundary(i_dim)
        pbc_count(i_part,i_dim) = pbc_count(i_part,i_dim) - 1
      end if
    end do
  end do
  ! define unfolded coordinates
  r0_unfolded(:, 1) = r0(:, 1) + real(pbc_count(:, 1), kind=dp)*boundary(1)
  r0_unfolded(:, 2) = r0(:, 2) + real(pbc_count(:, 2), kind=dp)*boundary(2)
  r0_unfolded(:, n_dim) = r0(:, n_dim)
  ! copy values on pbc_count_old(:, :)
  pbc_count_old(:, :) = pbc_count(:, :)
end subroutine read_configuration_rng_3d
!-----------------------------------------------------------------------------

!-----------------------------------------------------------------------
! reads a configuration and performs some checks for consistency
! compatibility routine for versions before V1.9
subroutine read_configuration_rng_2d(old_configuration_file)
  character(len=*), intent(in) :: old_configuration_file
  integer, parameter :: fileunit = 10
  integer :: iostatus
  integer, parameter :: two = 2
  ! loop variables
  integer :: i_dim
  integer :: i_mon, i_chain, i_cell_w_x, i_cell_w_y, i_wall, i_part
  ! loops over predictor-corrector coefficients
  integer :: i_order, j_order
  ! how many times the positions are shifted by a boundary
  integer :: pbc_shift
  ! helper variable
  real(kind=dp) :: r_dummy
  ! only in this case we can use this configuration read routine
  if(.not. (n_dim .eq. 2 .and. n_dim_pbc .eq. 2)) then
    write(unit=*, fmt='(a)') "ERROR (SR read_configuration_rng_2d): &
         &.not. (n_dim .eq. 2 .and. n_dim_pbc .eq. 2)"
    stop
  end if
  ! open file for replacement
  open(unit=fileunit, file=old_configuration_file, status="old", &
       action="read", iostat=iostatus)
  if(iostatus .ne. 0) then
    write (unit=*, fmt='(3a)') &
         "ERROR (SR read_configuration_rng_2d) : Could not read old &
         &configuration >>", old_configuration_file, "<<"
    stop
  else
    write(unit=*, fmt='(3a)') &
         "MESSAGE: Reading stored configuration in >>", &
         old_configuration_file, "<<"
    read(unit=fileunit, fmt='(2i12)') i_mon, i_chain
    read(unit=fileunit, fmt='(2i12)') i_cell_w_x, i_cell_w_y
    read(unit=fileunit, fmt='(i3)')   j_order

    ! check whether number of monomers right
    if((i_mon.ne.n_mon).or.(i_chain.ne.n_chain)) then
      write(unit=*, fmt='(a)') "WARNING (SR read_configuration): &
           &Polymer setups might not be compatible:"
      write(unit=*, fmt='(4(a, i6))') &
           "          i_mon=", i_mon, ", n_mon=", n_mon, &
           ", i_chain=", i_chain, ", n_chain=", n_chain
      if((i_mon*i_chain).ne.(n_mon*n_chain)) then
        write(unit=*, fmt='(a)') &
             "ERROR (SR read_configuration): Polymer setups not compatible:"
        write(unit=*, fmt='2(a, i8))') &
             "      i_mon*i_chain=", i_mon*i_chain, &
             ", n_mon*n_chain=", n_mon*n_chain
        stop
      end if
    end if
    ! check whether number of wall unit cells right
    if((i_cell_w_x.ne.n_cell_w_x).or.(i_cell_w_y.ne.n_cell_w_y)) then
      write(unit=*, fmt='(a)') &
           "WARNING (SR read_configuration_rng_2d): &
           &Set-up of wall might not be compatible"
      if((i_cell_w_x*i_cell_w_y).ne.(n_cell_w_x*n_cell_w_y)) then
        write(unit=*, fmt='(a)') &
             "ERROR (SR read_configuration_rng_2d): &
             &Set-up of wall is not compatible:"
        write(unit=*, fmt='(a,i3,a,i3)') &
             "i_cell_w_x =", i_cell_w_x, " i_cell_w_y=", i_cell_w_y
        write(unit=*, fmt='(a,i3,a,i3)') &
```

```
          "n_cell_w_x =", n_cell_w_x, " n_cell_w_y=", n_cell_w_y
      stop
    end if
  end if
  ! check order of algorithm
  if(j_order.ne.n_order) then
    write(unit=*, fmt='(a, i2)') &
         "WARNING (SR read_configuration_rng_2d): Order of old run",j_order
    write(unit=*, fmt='(a, i2)') &
         "WARNING (SR read_configuration_rng_2d): Order of new run",n_order
    if(n_order.lt.j_order) then
      write(unit=*, fmt='(a)') "ERROR (SR read_configuration_rng_2d): &
           &Order of new run < order of old run."
      write(unit=*, fmt='(a)') &
           "It is only possible to propagate a configuration with equal &
           &or higher order."
      stop
    end if
  end if
  ! read positions
  ! since version V1.7 we store the position in the canonical way: x, y, z
  do i_part = 1, n_part
    read(unit=fileunit, fmt='(i3, 3g26.18)') &
         type(i_part), (r0(i_part, i_dim), i_dim=1, two)
  end do
  ! read derivatives
  do i_order = 1, j_order
    r_dummy = dt**i_order ! scale coefficients with potencies of dt
    do i_part = 1, n_part
      read(unit=fileunit, fmt='(3g26.18)') &
           (rx(i_part,i_dim,i_order), i_dim=1, two)
      do i_dim = 1,two
        rx(i_part,i_dim,i_order) = rx(i_part,i_dim,i_order)*r_dummy
      end do
    end do
  end do
  ! zero out derivatives of higher order
  do i_order = j_order, n_order
    rx(:, :, i_order) = 0.0_dp
  end do
  ! random forces from previous MD step
  do i_part = 1, n_part
    read(unit=fileunit, fmt='(3g26.18)') &
         (force_random(i_part, i_dim), i_dim=1, two)
  end do
  ! equilibrium sites of wall atoms
  do i_wall = 1, n_wall
    read(unit=fileunit, fmt='(3g26.18)') &
         (r_wall_equi(i_wall,i_dim), i_dim=1, two)
  end do
  ! displacement of top wall and its derivatives
  read(unit=fileunit, fmt='(3g26.18)') &
       (r0_twall(i_dim),i_dim=1,two)
  do i_order = 1,j_order
    r_dummy = dt**i_order
    read(unit=fileunit,fmt='(3g26.18)') &
         (rx_twall(i_dim,i_order),i_dim=1,two)
    do i_dim = 1,two
      rx_twall(i_dim,i_order) = rx_twall(i_dim,i_order)*r_dummy
    end do
  end do
  ! zero out derivatives of higher order
  do i_order = j_order, n_order
    rx_twall(:, i_order) = 0.0_dp
  end do
  ! periodic boundary conditions tracking for top wall
  read(unit=fileunit, fmt='(3i12)') (pbc_twall(i_dim),i_dim=1,two)
  ! displacement of bottom wall (at present always (0, 0, 0))
  read(unit=fileunit, fmt='(3g26.18)') &
       (r0_bwall(i_dim),i_dim=1,two)
  ! periodic boundary conditions tracking for bottom wall
  read(unit=fileunit, fmt='(3i12)') (pbc_bwall(i_dim),i_dim=1,two)
  ! the position of the external spring
  read(unit=fileunit,  fmt='(3g26.18)') &
       (r0_spring_twall(i_dim),i_dim=1,two)
  ! the status of the force ramp
  read(unit=fileunit, fmt='(3i12)') (s_force_grad(i_dim),i_dim=1,two)
  ! here we have the coice if we restart the random number generator
  ! from the configuration or not (seed in the parameter file is then used)
  if(l_read_conf_rng_reinit) then
    ! read state of random number generator
    read(unit=fileunit, fmt='(6i12)') isdext(1:6)
    read(unit=fileunit, fmt='(6i12)') isdext(7:12)
    read(unit=fileunit, fmt='(6i12)') isdext(13:18)
    read(unit=fileunit, fmt='(6i12)') isdext(19:24)
    read(unit=fileunit, fmt='(i12)')  isdext(25)
    call rluxin
  end if
  close(fileunit)
end if ! configuration file could be read

! fold coordinates back in simulation box in case the saved coordinates
! were world (unfolded) coordinates
!-----------------------------------------------------------------------
! impose periodic boundary conditions for fluid particles in xy
do i_dim = 1, two
  do i_part = 1, n_mon_tot
    pbc_shift = int(r0(i_part,i_dim) / boundary(i_dim))
    pbc_count(i_part,i_dim) = pbc_shift
    r0(i_part,i_dim) = r0(i_part,i_dim) &
         - boundary(i_dim) * real(pbc_shift, kind=dp)
    if(r0(i_part, i_dim).lt.0.0_dp) then
      r0(i_part,i_dim) = r0(i_part,i_dim) + boundary(i_dim)
      pbc_count(i_part,i_dim) = pbc_count(i_part,i_dim) - 1
    end if
  end do
end do
! impose periodic boundary conditions for wall particles
do i_dim = 1, two
  do i_wall = 1, n_wall
    i_part = i_wall + n_mon_tot
    pbc_shift = int(r0(i_part,i_dim) / boundary(i_dim))
    pbc_count(i_part,i_dim) = pbc_shift
    r0(i_part,i_dim) = r0(i_part,i_dim) &
         - boundary(i_dim) * real(pbc_shift, kind=dp)
    ! do the same for the equilibrium positions
    r_wall_equi(i_wall,i_dim) = r_wall_equi(i_wall,i_dim) &
         - boundary(i_dim) * real(pbc_shift, kind=dp)
    if(r0(i_part, i_dim).lt.0.0_dp) then
      r0(i_part,i_dim) = r0(i_part,i_dim) + boundary(i_dim)
      r_wall_equi(i_wall,i_dim) = &
           r_wall_equi(i_wall,i_dim) + boundary(i_dim)
      pbc_count(i_part,i_dim) = pbc_count(i_part,i_dim) - 1
    end if
  end do
end do
! define unfolded coordinates
r0_unfolded(:, 1) = r0(:, 1) + real(pbc_count(:, 1), kind=dp)*boundary(1)
r0_unfolded(:, two)=r0(:, two)+real(pbc_count(:, 2), kind=dp)*boundary(two)
! copy values on pbc_count_old(:, :)
pbc_count_old(:, :) = pbc_count(:, :)
end subroutine read_configuration_rng_2d
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
! checks if chains bonds are ripped by the folding in the simulation box and
! changes periodic counts accordingly to avoid this.
! This is "chain sensitive folding" giving the name to this SR.
subroutine chain_sens_fold
  integer :: i_chain, i_part, i_dim
  integer :: pbc_shift
  integer :: shift_counter
  real(kind=dp), dimension(n_dim) :: Rcm_chain
  if(l_debug_chain_sens_fold) &
       print *, "DEBUG: Entering SR chain_sens_fold"
  ! there's nothing to do for a simple fluid
  if(n_mon .eq. 1) then
    if(l_chain_sens_fold_verbose) then
      write(unit=*, fmt='(a)') "MESSAGE (SR chain_sens_fold):&
           & Nothing to do for simple fluid"
    end if
    return
  end if
  ! counter for the number of position shifts performed
  shift_counter = 0
  ! unfolded coordinates must already be defined
  do i_chain = 1, n_chain
    if(l_debug_chain_sens_fold) then
      print *, "i_chain=", i_chain
      do i_part = (i_chain -1) * n_mon +1, i_chain * n_mon
        call writevec(r0_unfolded(i_part, :))
      end do
    end if
    do i_dim = 1, n_dim_pbc
      do i_part = (i_chain -1) * n_mon +2, i_chain * n_mon
        if(l_debug_chain_sens_fold) then
          print *, "diff(", i_part, ",", i_dim, ")=", &
             r0_unfolded(i_part, i_dim) - r0_unfolded(i_part -1, i_dim)
        end if
        if(r0_unfolded(i_part, i_dim) - r0_unfolded(i_part -1, i_dim) &
             .lt. -half_bound(i_dim)) then
          r0_unfolded(i_part, i_dim) = &
               r0_unfolded(i_part, i_dim) + boundary(i_dim)
          shift_counter = shift_counter +1
          if(l_chain_sens_fold_verbose) then
            write(unit=*, fmt='(a, i8, a, i1, a)') &
                 "MESSAGE (SR chain_sens_fold): Shifted particle", i_part, &
                 " by +boundary(", i_dim, ")"
          end if
        else if(r0_unfolded(i_part, i_dim) - r0_unfolded(i_part -1, i_dim) &
             .gt. half_bound(i_dim)) then
          r0_unfolded(i_part, i_dim) &
               = r0_unfolded(i_part, i_dim) - boundary(i_dim)
          shift_counter = shift_counter +1
          if(l_chain_sens_fold_verbose) then
            write(unit=*, fmt='(a, i8, a, i1, a)') &
                 "MESSAGE (SR chain_sens_fold): Shifted particle", i_part, &
                 " by -boundary(", i_dim, ")"
          end if
        end if
      end do
    end do
  end do
  ! check if everything is okay now.
  do i_chain = 1, n_chain
    do i_dim = 1, n_dim_pbc
      do i_part = (i_chain -1) * n_mon +2, i_chain * n_mon
        if(r0_unfolded(i_part, i_dim) - r0_unfolded(i_part -1, i_dim) &
             .lt. -half_bound(i_dim)) then
          write(unit=*, fmt='(a, i8, a, i1, a)') &
               "ERROR (SR chain_sens_fold): & Shifted again particle", &
               i_part, " by +boundary(", i_dim, ")"
          stop
        else if(r0_unfolded(i_part, i_dim) - r0_unfolded(i_part -1, i_dim) &
             .gt. half_bound(i_dim)) then
          write(unit=*, fmt='(a, i8, a, i1, a)') &
               "ERROR (SR chain_sens_fold): & Shifted again particle", &
               i_part, " by -boundary(", i_dim, ")"
          stop
        end if
      end do
    end do
  end do
  if(shift_counter .gt. 0) then
    write(unit=*, fmt='(a, i8, a)') "MESSAGE (SR chain_sens_fold): &
         &Shifted", shift_counter, " particle posistions."
  else
    write(unit=*, fmt='(a)') "MESSAGE (SR chain_sens_fold): &
         &Left positions unchanged"
  end if
  ! get the folded coordinates, this defines the new
  ! pbc_count(1:n_mon_tot, :)
  do i_dim = 1, n_dim-1
    do i_part = 1, n_mon_tot
      pbc_shift = int(r0_unfolded(i_part,i_dim) / boundary(i_dim))
      pbc_count(i_part,i_dim) = pbc_shift
      r0(i_part,i_dim) = r0_unfolded(i_part,i_dim) &
           - boundary(i_dim) * real(pbc_shift, kind=dp)
      if(r0(i_part, i_dim).lt.0.0_dp) then
        r0(i_part,i_dim) = r0(i_part,i_dim) + boundary(i_dim)
        pbc_count(i_part,i_dim) = pbc_count(i_part,i_dim) - 1
      end if
    end do
  end do
  ! copy values on pbc_count_old(:, :)
  pbc_count_old(:, :) = pbc_count(:, :)
  if(l_debug_chain_sens_fold) &
       print *, "DEBUG: Leaving SR chain_sens_fold"
end subroutine chain_sens_fold
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
! when a new simulation is started, it is esthetically more appealing to
! have all chains as much in the simulation box as possible.  To this end,
! we shift the chain centers of mass back in the simulation box.
! We also shift the wall atoms back in the box.
subroutine shift_chains_in_box
  integer :: i_dim, i_chain
  logical :: l_chains_shifted
  ! best integer approximation to the center-of-mass pbc shift of a chain
  ! in one dimension
  integer :: cm_pbc_count
  l_chains_shifted = .FALSE.
  do i_chain=1, n_chain
    do i_dim=1, n_dim_pbc
      cm_pbc_count = nint( &
           real(sum(pbc_count((i_chain-1)*n_mon+1:i_chain*n_mon, i_dim)), &
           kind=dp) / real(n_mon, kind=dp))
      pbc_count((i_chain-1)*n_mon+1:i_chain*n_mon, i_dim) = &
           pbc_count((i_chain-1)*n_mon+1:i_chain*n_mon, i_dim) - cm_pbc_count
      if(cm_pbc_count .ne. 0) l_chains_shifted = .TRUE.
```

```
      end do
    end do
    if(l_chains_shifted) then
      write(unit=*, fmt='(a)') "MESSAGE (SR shift_chains_in_box): &
           & Shifted chains back in simulation box"
    else
      write(unit=*, fmt='(a)') "MESSAGE (SR shift_chains_in_box): &
           & Shifted no chains back in simulation box"
    end if
    if(maxval(abs(pbc_count(n_mon_tot+1:, :))) .gt. 0) then
      ! simply forget the periodic boundary counts of the walls
      pbc_count(n_mon_tot+1:, :) = 0
      write(unit=*, fmt='(a)') "MESSAGE (SR shift_chains_in_box): &
           & Shifted wall atoms back in simulation box by resetting &
           &pbc_count(n_mon_tot+1:, :)"
    else
      write(unit=*, fmt='(a)') "MESSAGE (SR shift_chains_in_box): &
           & Wall atoms not shifted, because pbc_count(n_mon_tot+1:, :)=0"
    end if
    ! copy values on pbc_count_old(:, :)
    pbc_count_old(:, :) = pbc_count(:, :)
  end subroutine shift_chains_in_box
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! prints a vector to the screen
  subroutine writevec(vec)
    real(kind=dp), intent(in), dimension(:) :: vec
    integer :: i_dim
    write(unit=*, advance="no", fmt='(a)') "("
    do i_dim=1, ubound(vec,1) - lbound(vec,1)
      write(unit=*, advance="no", fmt='(f8.2, a)') vec(i_dim), ","
    end do
    write(unit=*, advance="yes", fmt='(f8.2, a)') vec(ubound(vec,1)), ")"
  end subroutine writevec
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! subroutine for monitoring the total energy of the system as a test of the
  ! algorithm and implementation.
  subroutine energy(energy_log_file)
    character(len=*), intent(in) :: energy_log_file
    integer :: iostatus
    integer :: i_dim
    integer :: i_part ! loop over particles
    ! centers of mass kinetic energies
    real(kind=dp) :: t_cms
    ! reference frame for fluid particle kinetic energies
    real(kind=dp), dimension(n_dim) :: vel_fl_cm_frame
    ! reference frame for wall particle kinetic energies
    real(kind=dp), dimension(n_dim) :: vel_tw_cm_frame, vel_bw_cm_frame
    ! reference frame for kinetic energies
    real(kind=dp), dimension(n_dim) :: vel_ref_frame
    ! external static energy of top wall (think "gravitation")
    real(kind=dp) :: static_energy_top_wall
    ! the sum of all energies
    real(kind=dp)       :: sum_all_energies
    real(kind=dp), save :: sum_all_energies_1
    ! low pass filtered kinetic energies
    real(kind=dp), save :: t_fluid_lpf, t_wall_lpf
    ! initialize
    static_energy_top_wall = 0.0_dp
    ! coupling to external spring or force gives static energy
    ! if we use a force ramp, move the top wall in the plane or we are
    ! pulling on the external spring we can't expect energy conservation
    ! and non static contributions are important.
    do i_dim = 1, n_dim
      select case(f_twall(i_dim))
      ! velocity_mode does not give an extra (static) energy
      case(force_mode)
        ! "gravitational energy" in z-direction
        if(i_dim .eq. 3) then
          ! if external force presses _down_ on the system, there is more
          !  static energy in the system when it moves _up_, therefore the "-"
          static_energy_top_wall = static_energy_top_wall &
               - real(n_top_wall, kind=dp) * ext_force_twall(i_dim) &
               * (r0_twall(i_dim) - r0_bwall(i_dim))
        end if
      case(spring_mode)
        ! energy in spring
        static_energy_top_wall = static_energy_top_wall &
             + real(n_top_wall, kind=dp)*(0.5_dp * k_spring_twall(i_dim)) &
             * (r0_spring_twall(i_dim)-r0_twall(i_dim))**2
      end select
    end do
    ! initialize kinetic energies
    t_fluid = 0.0_dp
    t_wall = 0.0_dp
    ! chose reference frame for energy calculation.
    ! We can separate the center of mass motion and chose every reference
    ! system we please, but the absolute value changes.
    ! Changing the reference frame from the laboratory system will introduce
    ! fluctuations which are by O(n_part) bigger than the energy fluctuations
    ! due to the propagation and will thus obscure the energy conservation
    ! check with the velocity-Verlet algorithm with no thermostat.
    ! So for checking energy conservation,
    ! l_use_lab_system_for_energies .eq. .TRUE. is required.
    if(l_use_lab_system_for_energies) then
      vel_ref_frame(:)   = 0.0_dp
      vel_fl_cm_frame(:) = 0.0_dp
      vel_tw_cm_frame(:) = 0.0_dp
      vel_bw_cm_frame(:) = 0.0_dp
    else
      ! this is for getting the kinetic temperature in the fluid and the walls
      ! (assuming that the fluid moves on average with half the top wall
      ! velocity, which must be true as long time average for symmetry between
      ! the walls)
      ! We also assume that the wall particles move with the equilibrium
      ! positions
      vel_ref_frame(:)   = 0.5_dp * rx_twall(:, 1)
      vel_fl_cm_frame(:) = 0.5_dp * rx_twall(:, 1)
      vel_tw_cm_frame(:) = rx_twall(:, 1)
      vel_bw_cm_frame(:) = 0.0_dp
!!$      vel_ref_frame(:) = 0.0_dp
!!$      vel_fl_cm_frame(:) = 0.0_dp
!!$      vel_tw_cm_frame(:) = 0.0_dp
!!$      vel_bw_cm_frame(:) = 0.0_dp
!!$
      ! taking the actual centers of mass costs a lot of cpu-time and
      ! introduces more fluctuations, as we are summing up a lot of numbers
      ! of similar magnitude with both signs.
!!$      if(n_mon_tot .gt. 0) then
!!$        vel_fl_cm_frame(:) = sum(rx(1:n_mon_tot, :, 1)) &
!!$             / real(n_mon_tot, kind=dp)
!!$      else
!!$        vel_fl_cm_frame(:) = 0.0_dp
!!$      end if
!!$
!!$      vel_tw_cm_frame(:) = sum(rx(n_mon_tot+1:n_mon_tot+n_top_wall, :, 1)) &
!!$           / real(n_top_wall, kind=dp)
!!$      vel_bw_cm_frame(:) = sum(rx(n_mon_tot+n_top_wall+1:n_part, :, 1)) &
!!$           / real(n_bottom_wall, kind=dp)
!!$
!!$      vel_ref_frame(:) = vel_fl_cm_frame(:)
    end if
    ! kinetic energy of walls equilibrium positions thought as rigid lattice
    t_cms = 0.0_dp
    t_cms = t_cms &
         + 0.5_dp * mass_twall &
         * (dot_product(rx_twall(:, 1) - vel_ref_frame(:), &
         rx_twall(:, 1) - vel_ref_frame(:)) / dt_2) &
         + 0.5_dp * mass_bwall &
         * (dot_product(vel_ref_frame(:), &
         vel_ref_frame(:)) / dt_2)
    ! the center of mass motion of all particles against the reference frame
    t_cms = t_cms &
         + 0.5_dp * mass_fluid_part &
         * dot_product(vel_fl_cm_frame(:) - vel_ref_frame(:), &
                       vel_fl_cm_frame(:) - vel_ref_frame(:)) / dt_2 &
         + 0.5_dp * mass_twall_part &
         * dot_product(vel_tw_cm_frame(:) - vel_ref_frame(:), &
                       vel_tw_cm_frame(:) - vel_ref_frame(:)) / dt_2 &
         + 0.5_dp * mass_bwall_part &
         * dot_product(vel_bw_cm_frame(:) - vel_ref_frame(:), &
                       vel_bw_cm_frame(:) - vel_ref_frame(:)) / dt_2
    ! calculate kinetic energies of particles
    ! fluid
    do i_part = 1, n_mon_tot
      t_fluid = t_fluid + mass(i_part) &
           * dot_product(rx(i_part, :, 1) - vel_fl_cm_frame(:), &
           rx(i_part, :, 1) - vel_fl_cm_frame(:))
    end do
    if(f_wall_fix .ne. 1) then
      ! kinetic energy of fluid particles
      ! the kinetic energy of the whole walls was taken into account
      ! separately in t_cms
      do i_part = n_mon_tot+1, n_mon_tot + n_top_wall
        t_wall = t_wall + mass(i_part) &
             * dot_product(rx(i_part, :, 1) - vel_tw_cm_frame(:), &
             rx(i_part, :, 1) - vel_tw_cm_frame(:))
      end do
      do i_part = n_mon_tot + n_top_wall +1, n_part
        t_wall = t_wall + mass(i_part) &
             * dot_product(rx(i_part, :, 1) - vel_bw_cm_frame(:), &
             rx(i_part, :, 1) - vel_bw_cm_frame(:))
      end do
    end if
    ! divide by common factor (the dt_2 comes from the storage in rx)
    t_wall = t_wall/(2.0_dp * dt_2)
    t_fluid = t_fluid/(2.0_dp * dt_2)
    ! potential energies have been computed in interaction routines
    v_total = v_wall_wall + v_wall_harm_tom + v_wall_harm_fre_kon_linear + &
         v_wall_harm_fre_kon_vector + v_fluid_wall + v_fluid_fluid + &
         v_intra_molec
    t_total = t_wall + t_fluid  ! kinetic energies of particles
    e_total = v_total + t_total ! total energy of particles
    sum_all_energies = e_total + static_energy_top_wall + t_cms
    ! write out time and energy contributions of the present MD step
    if(l_write_energy_every_mds) then
      open(unit=energy_log_file_unit, file=energy_log_file, status="old", &
           action="write", position="append", iostat=iostatus)
      if(iostatus .ne. 0) then
        write (unit=*,fmt='(a)') "WARNING: Could not append to energy log file"
      else
        if(.FALSE.) then ! normalized to the number of particles
          ! distinguish if there is fluid or not
          if(n_mon_tot .gt. 0) then
            write(unit=energy_log_file_unit, fmt='(e14.6, e21.13, 11e13.5)') &
                 r_time, &
                 sum_all_energies / real(n_moving, kind=dp), &
                 static_energy_top_wall / real(n_top_wall, kind=dp), &
                 t_cms / real(n_top_wall, kind=dp), &
                 t_wall / real(n_wall, kind=dp), &
                 t_fluid / real(n_mon_tot, kind=dp), &
                 v_wall_wall / real(n_wall, kind=dp), &
                 v_wall_harm_tom / real(n_wall, kind=dp), &
                 v_wall_harm_fre_kon_linear / real(n_wall, kind=dp), &
                 v_wall_harm_fre_kon_vector / real(n_wall, kind=dp), &
                 v_fluid_wall / real(n_mon_tot, kind=dp), &
                 v_fluid_fluid / real(n_mon_tot, kind=dp), &
                 v_intra_molec / real(n_mon_tot, kind=dp)
          else
            write(unit=energy_log_file_unit, fmt='(e14.6, e21.13, 7e13.5)') &
                 r_time, &
                 sum_all_energies / real(n_moving, kind=dp), &
                 static_energy_top_wall / real(n_top_wall, kind=dp), &
                 t_cms / real(n_top_wall, kind=dp), &
                 t_wall / real(n_wall, kind=dp), &
                 v_wall_wall / real(n_wall, kind=dp), &
                 v_wall_harm_tom / real(n_wall, kind=dp), &
                 v_wall_harm_fre_kon_linear / real(n_wall, kind=dp), &
                 v_wall_harm_fre_kon_vector / real(n_wall, kind=dp)
          end if
        else ! do not normalize to number of particles
          ! distinguish if there is fluid or not
          if(n_mon_tot .gt. 0) then
            write(unit=energy_log_file_unit, fmt='(e14.6, e21.13, 11e13.5)') &
                 r_time, &
                 sum_all_energies, &
                 static_energy_top_wall, &
                 t_cms, &
                 t_wall, &
                 t_fluid, &
                 v_wall_wall, &
                 v_wall_harm_tom, &
                 v_wall_harm_fre_kon_linear, &
                 v_wall_harm_fre_kon_vector, &
                 v_fluid_wall, &
                 v_fluid_fluid, &
                 v_intra_molec
          else
            write(unit=energy_log_file_unit, fmt='(e14.6, e21.13, 7e13.5)') &
                 r_time, &
                 sum_all_energies, &
                 static_energy_top_wall, &
                 t_cms, &
                 t_wall, &
                 v_wall_wall, &
                 v_wall_harm_tom, &
                 v_wall_harm_fre_kon_linear, &
                 v_wall_harm_fre_kon_vector
          end if
        end if
        close(energy_log_file_unit)
      end if ! if(iostatus .ne. 0)
    end if ! write_energy_every_mds
    ! computing first and second moment of the energy distributions makes only
    ! sense if we are not ramping the effective minimum distance.
    if(i_time .gt. n_relax) then
      ! sum up for the first and second moment
      v_wall_wall_1 = v_wall_wall_1 + v_wall_wall
      t_wall_1      = t_wall_1      + t_wall
      e_wall_wall_1 = e_wall_wall_1 + (v_wall_wall+t_wall)

      v_wall_wall_2 = v_wall_wall_2 + v_wall_wall**2
      t_wall_2      = t_wall_2      + t_wall**2
```

```
    e_wall_wall_2 = e_wall_wall_2 + (v_wall_wall+t_wall)**2
    v_fluid_fluid_1 = v_fluid_fluid_1 + v_fluid_fluid
    t_fluid_1       = t_fluid_1       + t_fluid
    e_fluid_fluid_1 = e_fluid_fluid_1 + (v_fluid_fluid+t_fluid)
    v_fluid_fluid_2 = v_fluid_fluid_2 + v_fluid_fluid**2
    t_fluid_2       = t_fluid_2       + t_fluid**2
    e_fluid_fluid_2 = e_fluid_fluid_2 + (v_fluid_fluid+t_fluid)**2
    v_total_1 = v_total_1 + v_total
    t_total_1 = t_total_1 + t_total
    e_total_1 = e_total_1 + e_total
    v_total_2 = v_total_2 + v_total**2
    t_total_2 = t_total_2 + t_total**2
    e_total_2 = e_total_2 + e_total**2
  end if
  ! total energy
  if(i_time .eq. s_time +1) then
    ! assign the first value computed
    sum_all_energies_1 = sum_all_energies
  else
    sum_all_energies_1 = &
         (1.0_dp-1.0_dp/real(n_time_ave, kind=dp))*sum_all_energies_1 &
        + (1.0_dp / real(n_time_ave, kind=dp))*sum_all_energies
  end if
  ! wall-wall potential
  if(i_time .eq. s_time +1) then
    simulated_wall_wall_potential_1 = v_wall_wall
  else
    simulated_wall_wall_potential_1 = &
         (1.0_dp-1.0_dp/real(n_time_ave,kind=dp)) &
        * simulated_wall_wall_potential_1 &
        + (1.0_dp / real(n_time_ave, kind=dp))*v_wall_wall
  end if
  ! kinetic energy of fluid
  ! Note: t_fluid_1 is the average over all observation steps
  if(n_mon_tot .gt. 0) then
    if(i_time .eq. s_time +1) then
      t_fluid_lpf = t_fluid
    else
      t_fluid_lpf = &
           (1.0_dp-1.0_dp/real(n_time_ave, kind=dp)) * t_fluid_lpf &
           + (1.0_dp / real(n_time_ave, kind=dp)) * t_fluid
    end if
  end if
  ! kinetic energy of wall particles in rest frame of wall centers of mass
  if(f_wall_fix .eq. 0) then
    if(i_time .eq. s_time +1) then
      t_wall_lpf = t_wall
    else
      t_wall_lpf = &
           (1.0_dp-1.0_dp/real(n_time_ave, kind=dp)) * t_wall_lpf &
           + (1.0_dp / real(n_time_ave, kind=dp)) * t_wall
    end if
  end if
  ! write out every n_time_ave time steps
  if(mod(i_time, n_time_ave).eq.0) then
    ! normalize to particle numbers when writing out
    if((n_mon_tot .gt. 0) .and. (f_wall_fix .eq. 0)) then
      write(energy_out_unit, fmt='(es17.11, 4g14.6)') r_time, &
           sum_all_energies_1 / real(n_part, kind=dp), &
           simulated_wall_wall_potential_1 / real(n_wall, kind=dp), &
           t_fluid_lpf / real(n_mon_tot, kind=dp), &
           t_wall_lpf / real(n_wall, kind=dp)
    else if(n_mon_tot .gt. 0) then ! f_wall_fix .ne. 0
      write(energy_out_unit, fmt='(es17.11, 3g14.6)') r_time, &
           sum_all_energies_1 / real(n_part, kind=dp), &
           simulated_wall_wall_potential_1 / real(n_wall, kind=dp), &
           t_fluid_lpf / real(n_mon_tot, kind=dp)
    else
      write(energy_out_unit, fmt='(es17.11, 3g14.6)') r_time, &
           sum_all_energies_1 / real(n_part, kind=dp), &
           simulated_wall_wall_potential_1 / real(n_wall, kind=dp), &
           t_wall_lpf / real(n_wall, kind=dp)
    end if
  end if
end subroutine energy
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! subroutine for writing out useful information and observables
subroutine simulation_log_out(simulation_log_file)
  character(len=*), intent(in) :: simulation_log_file
  integer, parameter :: fileunit = 20
  integer :: io_status
  ! check if the file we want to write already exists
  open(unit=fileunit, file=simulation_log_file, status="old", &
       iostat=io_status)
  if(io_status == 0) then
    write(unit=*, fmt='(3a)') "CAUTION: File >>", simulation_log_file, "<<"
    write(unit=*, fmt='(a)') &
         "          already exists and will be overwritten."
    close(unit=fileunit)
  end if
  ! open file for replacement
  open(unit=fileunit, file=simulation_log_file, status="replace", &
       action="write", iostat=io_status)
  if(io_status /= 0) then
     write (*, *) "WARNING: Could not open output file for simulation log &
          &data >>", simulation_log_file, "<<"
  end if
  write(unit=fileunit, fmt='(a)') &
       "This file contains information about the system parameters of the "
  write(unit=fileunit, fmt='(a)') &
       "simulation and data collected during the simulation"
  write(unit=fileunit, fmt='(2a)') &
       "Program was ", program_name
  write(unit=fileunit, fmt='(a)') &
       "=============================================================================&
       &============"
  write(unit=fileunit, fmt='(a)') ""
  write(unit=fileunit, fmt='(a)') &
       "Compiled in program parameters (as set in module globals):"
  write(unit=fileunit, fmt='(a)') &
       "-----------------------------------------------------------------------------&
       &------------"
  write(unit=fileunit, fmt='(3(a,i1))') "n_dim_max=", n_dim_max, &
       ", n_dim=", n_dim, ", n_dim_pbc=", n_dim_pbc
  write(unit=fileunit, fmt='(a, i2)') &
       "Order of integration algorithm: n_order =", n_order
  write(unit=fileunit, fmt='(a)') ""
  write(unit=fileunit, fmt='(2(a, i4))') &
       "Wall size parameters: n_cell_w_x = ", n_cell_w_x, &
       ", n_cell_w_y = ", n_cell_w_y
  write(unit=fileunit, fmt='(2(a, i8))') &
       "Wall numbers: n_top_wall=", n_top_wall, &
       ", n_bottom_wall=", n_bottom_wall
  write(unit=fileunit, fmt='(3(a, i8))') &
       "Fluid numbers: n_mon=", n_mon, ", n_chain=", n_chain, &
       ", n_mon_tot=", n_mon_tot
  write(unit=fileunit, fmt='(a, g13.6, a)') &
       "Covering:", real(n_mon_tot)/real(n_wall/2), " mono-layers"
  write(unit=fileunit, fmt='(a, g13.6, a)') &
       "Fluid density in periodic volume:", &
       real(n_mon_tot, kind=dp) / pbc_volume
  write(unit=fileunit, fmt='(a, i8)') &
       "Total number of particles in the simulation: n_part=", n_part
  write(unit=fileunit, fmt='(a, i8)') &
       "Total number of moving particles: n_moving=", n_moving
  write(unit=fileunit, fmt='(a)') ""
  write(unit=fileunit, fmt='(a)') &
       "FENE polymer backbone potential parameters:"
  write(unit=fileunit, fmt='(2(a, f13.6))') &
       "k_chain=", k_chain, ", r_chain=", r_chain
  write(unit=fileunit, fmt='(a)') ""
  write(unit=fileunit, fmt='(a, l2)') &
       "l_use_mc_fluid_setup           ", l_use_mc_fluid_setup
  if(n_dim_pbc .eq. 2) then
    write(unit=fileunit, fmt='(a, l2)') &
         "l_forbid_xy_bond_crossings     ", l_forbid_xy_bond_crossings
  else
    write(unit=fileunit, fmt='(a)') &
         "l_forbid_xy_bond_crossings     ignored as n_dim_pbc .ne. 2"
  end if
  if(n_dim .eq. n_dim_pbc+1) then
    write(unit=fileunit, fmt='(a, l2, a)') &
         "l_2d_flat_setup                ", l_2d_flat_setup, &
         " for system with walls"
  else
    write(unit=fileunit, fmt='(a)') &
         "l_2d_flat_setup                ignored for system without walls"
  end if
  write(unit=fileunit, fmt='(a, l2)') &
       "l_use_lab_system_for_energies  ", l_use_lab_system_for_energies
  write(unit=fileunit, fmt='(a, l2)') &
       "l_fluid_fluid_interaction      ", l_fluid_fluid_interaction
  write(unit=fileunit, fmt='(a, l2)') &
       "l_fluid_wall_interaction       ", l_fluid_wall_interaction
  write(unit=fileunit, fmt='(a, l2)') &
       "l_allow_wall_wall_interaction  ", l_allow_wall_wall_interaction
  write(unit=fileunit, fmt='(a, l2)') &
       "l_intra_wall_interaction       ", l_intra_wall_interaction
  write(unit=fileunit, fmt='(a, l2)') &
       "l_intra_molec_interaction      ", l_intra_molec_interaction
  write(unit=fileunit, fmt='(a)') ""
  write(unit=fileunit, fmt='(a)') "Data observed during the simulation"
  write(unit=fileunit, fmt='(a)') &
       "-----------------------------------------------------------------------"
  write(unit=fileunit, fmt='(a)') ""
  write(unit=fileunit, fmt='(a, f16.8, a, f16.8)') &
       "x_space=", x_space,", y_space=", y_space
  write(unit=fileunit, fmt='(a, f16.8)') &
       "load (per wall atom) -- pressure conversion factor=", &
       0.5_dp * x_space * y_space
  write(unit=fileunit, fmt='(a)') "Boundaries of fluid (x, y, z):"
  write(unit=fileunit, fmt='(3f16.10)') boundary(:)
  write(unit=fileunit, fmt='(a)') ""
  write(unit=fileunit, fmt='(a)') "Memory usage of bins:"
  ! note:  there's room for n_bin_XX particles, the first entry is
  ! the number of entries, and the index runs from 0 to n_bin
  write(unit=fileunit, fmt='(2(i8, a))') n_bin_fl_max, " of", n_bin_fl, &
       " fluid bin entries maximally used"
  write(unit=fileunit, fmt='(2(i8, a))') n_bin_wa_max, " of", n_bin_wa, &
       " wall  bin entries maximally used"
  write(unit=fileunit, fmt='(a)') ""
  write(unit=fileunit, fmt='(a)') "Memory usage of neighbor lists:"
  ! note:  there's room for n_neigh_XX particles, the first entry is
  ! the number of entries, and the index runs from 0 to n_neigh_XX
  write(unit=fileunit, fmt='(2(i8, a))') n_neigh_ff_max, " of", &
       n_neigh_ff, " fluid-fluid neighbor list entries maximally used"
  write(unit=fileunit, fmt='(2(i8, a))') n_neigh_fw_max, " of", &
       n_neigh_fw, " fluid-wall  neighbor list entries maximally used"
  write(unit=fileunit, fmt='(2(i8, a))') n_neigh_ww_max, " of", &
       n_neigh_ww, " wall-wall   neighbor list entries maximally used"
  write(unit=fileunit, fmt='(a)') ""
  write(unit=fileunit, fmt='(a, i12, a)') &
       "Neighbor list updates divided by n_mds_after_relax=", &
       n_mds_after_relax, " :"
  write(unit=fileunit, fmt='(f10.6, a)') &
       100.0_dp*real(counter_list_updates,kind=dp) &
       / real(n_mds_after_relax, kind=dp), "%"
  write(unit=fileunit, fmt='(a)') ""
  write(unit=fileunit, fmt='(a, es13.6)') &
       "Maximal encountered (absolute) coordinate correction:", &
       max_encountered_correction
  ! normalize data by the number of observations and number of wall atoms
  ! here was a very nasty little bug:
  ! real(n_wall * n_mds_after_relax, kind=dp) will give you
  ! (on 32 bit systems) a negative result, due to integer overflow.
  v_wall_wall_1 = v_wall_wall_1 &
       / (real(n_wall, kind=dp) * real(n_mds_after_relax, kind=dp))
  t_wall_1      = t_wall_1 &
       / (real(n_wall, kind=dp) * real(n_mds_after_relax, kind=dp))
  e_wall_wall_1 = e_wall_wall_1 &
       / (real(n_wall, kind=dp) * real(n_mds_after_relax, kind=dp))
  ! variance scales with N**2, so divide by it
  v_wall_wall_2 = v_wall_wall_2 &
       / (real(n_wall**2, kind=dp) * real(n_mds_after_relax, kind=dp))
  t_wall_2      = t_wall_2      &
       / (real(n_wall**2, kind=dp) * real(n_mds_after_relax, kind=dp))
  e_wall_wall_2 = e_wall_wall_2 &
       / (real(n_wall**2, kind=dp) * real(n_mds_after_relax, kind=dp))
  ! compute second moments
  v_wall_wall_2 = v_wall_wall_2 - v_wall_wall_1**2
  t_wall_2      = t_wall_2      - t_wall_1**2
  e_wall_wall_2 = e_wall_wall_2 - e_wall_wall_1**2
  ! with this scaling we get the compressibility
  v_wall_wall_2 = real(n_wall, kind=dp) * v_wall_wall_2/temp**2
  t_wall_2      = real(n_wall, kind=dp) * t_wall_2     /temp**2
  e_wall_wall_2 = real(n_wall, kind=dp) * e_wall_wall_2/temp**2
  write(unit=fileunit, fmt='(a)') ""
  write(unit=fileunit, fmt='(a)') &
       "Mean and variance*N/T^2 of interaction energies:"
  write(unit=fileunit, fmt='(a)') &
       "-----------------------------------------------"
  write(unit=fileunit, fmt='(a)') "Wall-Wall energies:"
  write(unit=fileunit, fmt='(2e15.6, a)') v_wall_wall_1, v_wall_wall_2, &
       " wall-wall potential energy"
  write(unit=fileunit, fmt='(2e15.6, a)') t_wall_1     , t_wall_2     , &
       " wall-atoms kinetic energy"
  write(unit=fileunit, fmt='(2e15.6, a)') e_wall_wall_1, e_wall_wall_2, &
       " wall-wall total energy"
  if(n_mon_tot .gt. 0) then
    v_fluid_fluid_1 = v_fluid_fluid_1 &
         / (real(n_mon_tot, kind=dp) * real(n_mds_after_relax, kind=dp))
    t_fluid_1       = t_fluid_1       &
         / (real(n_mon_tot, kind=dp) * real(n_mds_after_relax, kind=dp))
    e_fluid_fluid_1 = e_fluid_fluid_1 &
         / (real(n_mon_tot, kind=dp) * real(n_mds_after_relax, kind=dp))
    v_fluid_fluid_2 = v_fluid_fluid_2 &
         / (real(n_mon_tot**2, kind=dp) * real(n_mds_after_relax, kind=dp))
    t_fluid_2       = t_fluid_2       &
         / (real(n_mon_tot**2, kind=dp) * real(n_mds_after_relax, kind=dp))
    e_fluid_fluid_2 = e_fluid_fluid_2 &
         / (real(n_mon_tot**2, kind=dp) * real(n_mds_after_relax, kind=dp))
    v_fluid_fluid_2 = v_fluid_fluid_2 - v_fluid_fluid_1**2
    t_fluid_2       = t_fluid_2       - t_fluid_1**2
    e_fluid_fluid_2 = e_fluid_fluid_2 - e_fluid_fluid_1**2
    v_fluid_fluid_2 = real(n_mon_tot, kind=dp) * v_fluid_fluid_2/temp**2
```

```
    t_fluid_2       = real(n_mon_tot, kind=dp) * t_fluid_2      /temp**2
    e_fluid_fluid_2 = real(n_mon_tot, kind=dp) * e_fluid_fluid_2/temp**2
    write(unit=fileunit, fmt='(a)') ""
    write(unit=fileunit, fmt='(a)') "Fluid-fluid energies:"
    write(unit=fileunit, fmt='(2e15.6, a)') v_fluid_fluid_1,v_fluid_fluid_2,&
         " fluid-fluid potential energy"
    write(unit=fileunit, fmt='(2e15.6, a)') t_fluid_1      ,t_fluid_2      ,&
         " fluid-atoms kinetic energy"
    write(unit=fileunit, fmt='(2e15.6, a)') e_fluid_fluid_1,e_fluid_fluid_2,&
         " fluid-fluid total energy"
  end if !(n_mon_tot .gt. 0)
  ! normalize by number of observations and number of particles in fluid and
  ! walls.  For the kinetic energy, we have to use the number of moving
  ! particles, n_moving
  v_total_1 = v_total_1 &
       / (real(n_part, kind=dp) * real(n_mds_after_relax, kind=dp))
  t_total_1 = t_total_1 &
       / (real(n_moving, kind=dp) * real(n_mds_after_relax, kind=dp))
  e_total_1 = e_total_1 &
       / (real(n_part, kind=dp) * real(n_mds_after_relax, kind=dp))
  v_total_2 = v_total_2 &
       / (real(n_part**2, kind=dp) * real(n_mds_after_relax, kind=dp))
  t_total_2 = t_total_2 &
       / (real(n_moving**2, kind=dp) * real(n_mds_after_relax, kind=dp))
  e_total_2 = e_total_2 &
       / (real(n_part**2, kind=dp) * real(n_mds_after_relax, kind=dp))
  v_total_2 = v_total_2 - v_total_1**2
  t_total_2 = t_total_2 - t_total_1**2
  e_total_2 = e_total_2 - e_total_1**2
  v_total_2 = real(n_part, kind=dp) * v_total_2 / temp**2
  t_total_2 = real(n_moving, kind=dp) * t_total_2 / temp**2
  e_total_2 = real(n_part, kind=dp) * e_total_2 / temp**2
  write(unit=fileunit, fmt='(a)') ""
  write(unit=fileunit, fmt='(a)') "Sums of fluid and wall contributions:"
  write(unit=fileunit, fmt='(2e15.6, a)') v_total_1, v_total_2, &
       " total potential energy"
  write(unit=fileunit, fmt='(2e15.6, a)') t_total_1, t_total_2, &
       " total kinetic energy"
  write(unit=fileunit, fmt='(2e15.6, a)') e_total_1, e_total_2, &
       " total potential and kinetic energy"
  write(unit=fileunit, fmt='(a)') ""
  write(unit=fileunit, fmt='(a)') &
       "-------------------------------------------------------------------"
  write(unit=fileunit, fmt='(a)') ""
  write(unit=fileunit, fmt='(a)') "fort.?? contents"
  write(unit=fileunit, fmt='(a)') &
       "-------------------------------------------------------------------"
  ! write out what the fort.XX files contain
  write(unit=fileunit, fmt='(a)') "fort.41: radius of gyration and squared &
       &end-to-end distance low pass filtered"
  if(l_compute_press_tens) then
    write(unit=fileunit, fmt='(a)') &
         "fort.42: pressure tensor low pass filtered"
  end if
  write(unit=fileunit, fmt='(a, i2, a)') "fort.", pos_vel_out_unit, &
       ": Position and velocity of top wall low pass filtered"
  write(unit=fileunit, fmt='(a, i2, a)') "fort.", forces_out_unit, &
       ": external and total force on top wall low pass filtered"
  write(unit=fileunit, fmt='(a, i2, a)') "fort.", energy_out_unit, &
       ": total, wall-wall, fluid and wall kinetic energy low pass filtered"
  ! close output file
  close(fileunit)
end subroutine simulation_log_out
!-------------------------------------------------------------------------

!-------------------------------------------------------------------------
! wrapper for writing positions and velocities on linear scale and at the
! times in the sample list (if read)
subroutine pos_and_vel_out
  if(l_write_particle_positions .or. l_write_particle_velocities) then
    if(mod(i_time, n_linear_out) .eq. 0) then
      if(l_write_particle_positions) then
        if(n_dim .eq. n_dim_pbc) then
          call fluid_positions_out("fl_pos_t", i_time)
        else
          call particle_positions_out("pos_t", i_time)
        end if
      end if
      if(l_write_particle_velocities) call particle_velocities_out("vel_")
    end if
    if(l_read_sample_list .and. (i_time .eq. next_sample_time)) then
      call get_next_sample_time(sample_times, next_sample_index, &
           next_sample_time)
      ! we only write out if the sample time is different from the linear
      ! saving time
      if(mod(i_time, n_linear_out) .ne. 0) then
        if(l_write_particle_positions) then
          if(n_dim .eq. n_dim_pbc) then
            call fluid_positions_out("fl_pos_t", i_time)
          else
            call particle_positions_out("pos_t", i_time)
          end if
        end if
        if(l_write_particle_velocities) call particle_velocities_out("vel_")
      end if
    end if
  end if
end subroutine pos_and_vel_out
!-------------------------------------------------------------------------

!-------------------------------------------------------------------------
subroutine fluid_positions_out(fluid_pos_basename, i_time_var, &
     part_index_low, part_index_high)
  character(len=*), intent(in) :: fluid_pos_basename
  ! time variable: MC-step or MD-step
  integer, intent(in) :: i_time_var
  integer, intent(in), optional :: part_index_low, part_index_high
  integer, parameter :: fileunit = 61
  integer :: io_status
  ! variables for putting together the filename
  integer :: length
  character(len=200)  :: filename
  character(len=80) :: format_string
  ! loop variables
  integer :: i_dim, i_part, i_part_low, i_part_high
  if(l_debug_fluid_positions_out) &
       print *, "DEBUG: Entering SR fluid_positions_out"
  ! fill in default arguments
  if(present(part_index_low)) then
    i_part_low = part_index_low
  else
    i_part_low = 1
  end if
  if(present(part_index_high)) then
    i_part_high = min(n_mon_tot, part_index_high)
  else
    i_part_high = n_mon_tot
  end if
  ! put together the output-filename
  write(filename, '(i12)') i_time_var
  filename = adjustl(filename)
  filename = fluid_pos_basename//trim(filename)//".dat"
  length = len(trim(adjustl(filename)))
  filename = adjustl(filename(:length))
  ! check if the file we want to write already exists
  open(unit=fileunit, file=trim(filename), status="old", &
       iostat=io_status)
  if(io_status == 0) then
    write(unit=*, fmt='(3a)') "CAUTION: File >>", trim(filename), "<<"
    write(unit=*, fmt='(a)') &
         "          already exists and will be overwritten."
    close(unit=fileunit)
  end if
  ! open file for replacement
  open(unit=fileunit, file=trim(filename), status="replace", &
       action="write", iostat=io_status)
  if(io_status .ne. 0) then
    write (unit=*, fmt='(3a)') "WARNING: Could not open output file >>", &
         trim(filename), "<<"
    write (unit=*, fmt='(a, i8)') "          iostat=", io_status
  else
    ! write out data
    write(unit=*, fmt='(3a)') "MESSAGE: Writing fluid positions to >>", &
         trim(filename), "<<"
    ! The format for the configurations is extremely simple:
    ! (x, y, z) of each particle.  For creating specific input files
    ! (e.g. for VMD or RasMol) we call approbriate scripts.
    ! We write out the unfolded coordinates of the particles, that is they
    ! are not folded back in the simulation box.
    ! Thus, dynamic trajectories and polymer bonds are not destroyed.
    ! Besides, this is how C. Bennemann stored his configurations.
    ! 6 digits precicion is the precision of 32-bit reals (floats in C).
    ! We need the absolute precision of the positions in case we compute
    ! differences of particle positions.
    write(format_string, fmt='(a,i1,a)') "(", n_dim, "f13.5)"
    do i_part = i_part_low, i_part_high
      write(unit=fileunit, fmt=format_string) (r0(i_part, i_dim) &
           + real(pbc_count(i_part, i_dim), kind=dp)*boundary(i_dim), &
           i_dim=1, n_dim)
    end do
  end if ! file successfully opened
  close(fileunit)
  if(l_debug_fluid_positions_out) &
       print *, "DEBUG: Leaving SR fluid_positions_out"
end subroutine fluid_positions_out
!-------------------------------------------------------------------------

!-------------------------------------------------------------------------
! writes out (x,y,z) positions of all particles, if particle indices
! part_index_low, part_index_high are given as optional arguments,
! write only these particle positions.
subroutine particle_positions_out(particle_pos_basename, i_time_var, &
     part_index_low, part_index_high, l_write_indices_tbwall)
  character(len=*), intent(in) :: particle_pos_basename
  ! time variable: MC-step or MD-step
  integer, intent(in) :: i_time_var
  integer, intent(in), optional :: part_index_low, part_index_high
  logical, intent(in), optional :: l_write_indices_tbwall
  integer, parameter :: fileunit = 61
  integer :: io_status
  ! variables for putting together the filename
  integer :: length
  character(len=240) :: filename
  character(len=21)  :: string
  character(len=80) :: format_string
  ! loop variables
  integer :: i_dim, i_part, i_part_low, i_part_high
  if(l_debug_particle_positions_out) &
       print *, "DEBUG: Entering SR particle_positions_out"
  ! fill in default arguments
  if(present(part_index_low)) then
    i_part_low = part_index_low
  else
    i_part_low = 1
  end if
  if(present(part_index_high)) then
    i_part_high = part_index_high
  else
    i_part_high = n_part
  end if
  ! integer to string conversion
  write(string, '(i12)') i_time_var
  string = adjustl(string)
  ! put together the filename
  filename = particle_pos_basename//trim(string)//".dat"
  ! we need the length of the filename to get rid of a newline
  length = len(trim(adjustl(filename)))
  ! trim() applied here does not do the job, we have do do it every time
  filename = adjustl(filename(:length))
  ! check if the file we want to write already exists
  open(unit=fileunit, file=trim(filename), status="old", &
       iostat=io_status)
  if(io_status == 0) then
    write(unit=*, fmt='(3a)') "CAUTION: File >>", trim(filename), "<<"
    write(unit=*, fmt='(a)') &
         "          already exists and will be overwritten."
    close(unit=fileunit)
  end if
  ! open file for replacement
  open(unit=fileunit, file=trim(filename), status="replace", &
       action="write", iostat=io_status)
  if(io_status .ne. 0) then
    write (unit=*, fmt='(3a)') "WARNING: Could not open output file >>", &
         trim(filename), "<<"
    write (unit=*, fmt='(a, i8)') "          iostat=", io_status
  else
    ! write out data
    write(unit=*, fmt='(3a)') "MESSAGE: Writing particle positions to >>", &
         trim(filename), "<<"
    ! The format for the configurations is extremely simple:
    ! (x, y, z) of each particle.  For creating specific input files
    ! (e.g. for VMD or RasMol) we call approbriate scripts.
    ! We write out the unfolded coordinates of the particles, that is they
    ! are not folded back in the simulation box.
    ! Thus, dynamic trajectories and polymer bonds are not destroyed.
    ! Besides, this is how C. Bennemann stored his configurations.
    ! 6 digits precicion is the precision of 32-bit reals (floats in C).
    ! We need the absolute precision of the positions in case we compute
    ! differences of particle positions.
    write(format_string, fmt='(a,i1,a)') "(", n_dim, "f13.5)"
    do i_part = i_part_low, i_part_high
      write(unit=fileunit, fmt=format_string) (r0(i_part, i_dim) &
           + real(pbc_count(i_part, i_dim), kind=dp)*boundary(i_dim), &
           i_dim=1, n_dim)
    end do
    if(.not. present(l_write_indices_tbwall) .or. &
         present(l_write_indices_tbwall) .and. l_write_indices_tbwall) then
      ! write out particle indices so that we know what positions we have
      write(unit=fileunit, fmt='(5i10)') &
           n_mon_tot, n_top_wall, n_bottom_wall, i_part_low, i_part_high
      ! position of top and bottom wall lattice
```

```
      write(unit=fileunit, fmt=format_string) r0_twall(:)
      write(unit=fileunit, fmt=format_string) r0_bwall(:)
    end if
    close(fileunit)
  end if ! file successfully opened
  if(l_debug_particle_positions_out) &
       print *, "DEBUG: Leaving SR particle_positions_out"
end subroutine particle_positions_out
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! writes out (x,y,z) velocities of all particles, if particle indices
! part_index_low, part_index_high are given as optional arguments,
! write only these particle positions.
subroutine particle_velocities_out(particle_vel_basename, &
     part_index_low, part_index_high)
  character(len=*), intent(in) :: particle_vel_basename
  integer, intent(in), optional :: part_index_low, part_index_high
  integer, parameter :: fileunit = 61
  integer :: io_status
  ! variables for putting together the filename
  integer :: length
  character(len=240) :: filename
  character(len=21)  :: string
  character(len=80) :: format_string
  ! loop variables
  integer :: i_part, i_part_low, i_part_high
  ! fill in default arguments
  if(present(part_index_low)) then
    i_part_low = part_index_low
  else
    i_part_low = 1
  end if
  if(present(part_index_high)) then
    i_part_high = part_index_high
  else
    i_part_high = n_part
  end if
  ! integer to string conversion
  write(string, '(i20)') i_time
  string = adjustl(string)
  ! put together the filename
  filename = particle_vel_basename//"t"//trim(string)//".dat"
  ! we need the length of the filename to get rid of a newline
  length = len(trim(adjustl(filename)))
  ! trim() applied here does not do the job, we have do do it every time
  filename = adjustl(filename(:length))
  ! check if the file we want to write already exists
  open(unit=fileunit, file=trim(filename), status="old", &
       iostat=io_status)
  if(io_status == 0) then
    write(unit=*, fmt='(3a)') "CAUTION: File >>", trim(filename), "<<"
    write(unit=*, fmt='(a)') &
         "         already exists and will be overwritten."
    close(unit=fileunit)
  end if
  ! open file for replacement
  open(unit=fileunit, file=trim(filename), status="replace", &
       action="write", iostat=io_status)
  if(io_status .ne. 0) then
    write (unit=*, fmt='(3a)') "WARNING: Could not open output file >>", &
         trim(filename), "<<"
    write (unit=*, fmt='(a, i12)') "          iostat=", io_status
  else
    ! write out data
    write(unit=*, fmt='(3a)') "MESSAGE: Writing particle velocities to >>", &
         trim(filename), "<<"
    ! write out the velocities with the same precision.  The velocities have
    ! average 0, so the relative precision is important.
    write(format_string, fmt='(a,i1,a)') "(", n_dim, "g14.6)"
    do i_part = i_part_low, i_part_high
      write(unit=fileunit, fmt=format_string) rx(i_part, :, 1)/dt
    end do
    ! write out indices
    ! write(unit=fileunit, fmt='(a, i8)') "i_part_low=", i_part_low
    ! write(unit=fileunit, fmt='(a, i8)') "i_part_high=", i_part_high
    ! write out particle indices so that we know what positions we have
    write(unit=fileunit, fmt='(5i10)') &
         n_mon_tot, n_top_wall, n_bottom_wall, i_part_low, i_part_high
    ! velocities of top wall lattice.  The bottom wall has fixed position,
    ! thus velocity 0.
    write(unit=fileunit, fmt=format_string) rx_twall(:, 1)/dt
    close(fileunit)
  end if ! file successfully opened
end subroutine particle_velocities_out
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! writes out information describing a configuration:
! particle numbers, periodic boundary conditions
subroutine write_config_infos(config_info_filename)
  character(len=*), intent(in) :: config_info_filename
  integer :: i_dim
  integer, parameter :: fileunit = 61
  integer :: io_status
  ! check if the file we want to write already exists
  open(unit=fileunit, file=config_info_filename, status="old", &
       iostat=io_status)
  if(io_status == 0) then
    write(unit=*, fmt='(3a)') "CAUTION: File >>", config_info_filename, "<<"
    write(unit=*, fmt='(a)') &
         "         already exists and will be overwritten."
    close(unit=fileunit)
  end if
  ! open file for replacement
  open(unit=fileunit, file=config_info_filename, status="replace", &
       action="write", iostat=io_status)
  if(io_status .ne. 0) then
    write (unit=*, fmt='(3a)') "WARNING: Could not open output file >>", &
         config_info_filename, "<<"
    write (unit=*, fmt='(a, i8)') "          iostat=", io_status
  else
    write (unit=*, fmt='(3a)') "MESSAGE: Writing configuration information &
         &to file >>", config_info_filename, "<<"
    ! number of fluid particles
    write(unit=fileunit, fmt='(a, i8)') &
         "n_mon_tot= ", n_mon_tot
    ! number of top wall particles
    write(unit=fileunit, fmt='(a, i8)') &
         "n_top_wall= ", n_top_wall
    ! number of bottom wall particles
    write(unit=fileunit, fmt='(a, i8, a)') &
         "n_bottom_wall= ", n_bottom_wall
    ! chain length
    write(unit=fileunit, fmt='(a, i8)') &
         "n_mon= ", n_mon
    ! dimensions of the system where periodic boundaries are applied
    do i_dim = 1, n_dim-1
      write(unit=fileunit, fmt='(a, i1, a, f17.10)') &
      "boundary(", i_dim, ")= ", boundary(i_dim)
    end do
  end if
  close(fileunit)
end subroutine write_config_infos
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! read the list with sample times
subroutine read_sample_list(file_sample_list, sample_times, &
     s_time, n_relax, n_obser)
  character(len=*), intent(in) :: file_sample_list
  ! the intent attribute is not allowed for pointers (M/R ch. 5.7.1)
  integer, dimension(:), pointer :: sample_times
  integer, intent(in) :: s_time, n_relax, n_obser
  integer, parameter :: fileunit = 10
  integer :: iostatus
  character(len=40) :: string
  integer :: i_sample_time
  ! open file for replacement
  open(unit=fileunit, file=file_sample_list, status="old", &
       action="read", iostat=iostatus)
  if(iostatus .ne. 0) then
    write (unit=*, fmt='(3a)') &
         "ERROR (SR read_sample_list): Could not read file &
         & with sample times >>", file_sample_list, "<<"
    stop
  else
    write(unit=*, fmt='(2a)') &
         "MESSAGE (SR read_sample_list): Reading ", trim(file_sample_list)
    ! read number of sample time in list, then allocate memory
    read(unit=fileunit, fmt='(a17, i12)') string, nr_samples_total
    if(nr_samples_total .le. 0) then
      write(unit=*, fmt='(a, i12, a)') &
           "ERROR (SR read_sample_list): nr_samples_total=", &
           nr_samples_total, " .le. 0"
      stop
    end if
    allocate(sample_times(1:nr_samples_total))
    do i_sample_time = 1, nr_samples_total
      read(unit=fileunit, fmt='(i12)') sample_times(i_sample_time)
    end do
    close(fileunit)
  end if ! could read sample times
  ! write out sample list
  ! write(unit=*, fmt='(a17, i12)') trim(string), nr_samples_total
  ! do i_sample_time = 1, nr_samples_total
  !   write(unit=*, fmt='(i12)') sample_times(i_sample_time)
  ! end do
  ! check matching with n_obser and n_relax
  if( (n_obser + n_relax + s_time) .lt. sample_times(nr_samples_total)) then
    write(unit=*, fmt='(2(a, i12))') &
         "ERROR (SR read_sample_list): n_obser + n_relax +s_time= ", &
          n_obser + n_relax + s_time, &
          " .lt. sample_times(nr_samples_total)=", &
          sample_times(nr_samples_total)
    stop
  end if
  ! define first time and index
  do next_sample_index = 1, nr_samples_total
    if(sample_times(next_sample_index) .ge. s_time) then
      next_sample_time = sample_times(next_sample_index)
      exit
    end if
  end do
  ! fallthrough, take last sample time for definitiveness
  if(next_sample_index  .eq. nr_samples_total +1) then
    write(unit=*, fmt='(2(a, i12))') &
         "CAUTION (SR read_sample_list): Start time of simulation=", &
         s_time, " is bigger then latest time in sample_list=", &
         sample_times(nr_samples_total)
    next_sample_index = nr_samples_total
    next_sample_time = sample_times(next_sample_index)
  end if
end subroutine read_sample_list
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
subroutine get_next_sample_time(sample_times, next_sample_index, &
     next_sample_time)
  integer, dimension(:), intent(in) :: sample_times
  integer, intent(inout) :: next_sample_index, next_sample_time
 next_sample_index =  next_sample_index +1
 if(next_sample_index .gt. nr_samples_total) then
   write(unit=*, fmt='(a, i12, a)') "MESSAGE (SR get_next_sample_time):", &
        sample_times(next_sample_index-1), " was the last sample time"
 else
   next_sample_time = sample_times(next_sample_index)
 end if
end subroutine get_next_sample_time
!------------------------------------------------------------------------------
end module utilities
```

poppingV1.9.f90

```
! poppingVx.y.f90
! module for finding potential minima between walls or in wall potential
! Also calculates probability distributions of particle motions.
! Martin Aichele, 2002-01-10
! last modified 2002-07-22
module popping

use globals
implicit none
!------------------------------------------------------------------------------
! debug switches      |-------- 31 characters ------|
!------------------------------------------------------------------------------
logical, parameter :: l_debug_p_dist_calc_and_write   = .FALSE.
logical, parameter :: l_debug_p_dist_write_out        = .FALSE.
logical, parameter :: l_debug_fluid_vel_p_dist        = .FALSE.
logical, parameter :: l_debug_fluid_vel_xy_pl_pdist   = .FALSE.
```

```
  logical, parameter :: l_debug_bond_ang_vel_p_dist     = .FALSE.
  logical, parameter :: l_debug_fluid_vel_xyz_p_dist    = .FALSE.
  logical, parameter :: l_debug_fluid_accel_p_dist      = .FALSE.
  logical, parameter :: l_debug_bin_data_array          = .FALSE.
  !---------------------------------------------------------------------------
  ! switch for deciding if we are calculating the potential of a fluid particle
  ! between the walls or a wall particle above a wall
  logical, parameter :: l_fluid_between_walls = .TRUE.
  !---------------------------------------------------------------------------
  ! Probability distributions of fluid velocities
  !---------------------------------------------------------------------------
  ! for commensurate surfaces at high shear rates, high velocities occur
  ! (v > 7).  The out of histogram range situation is repaired in the binning
  ! routine by putting the entries which are out of range into the outermost
  ! bins.
  ! switch for deciding if for bi-mers of center of mass is investigated
  logical :: l_take_cm_of_bimer = .FALSE.
  ! (X,Y,Z)-average
  ! number of bins for the probability distribution of fluid velocities
  integer, parameter :: n_fluid_vel_p_dist_bins = 1400
  ! bin start
  real(kind=dp), parameter :: fluid_vel_bin_start = 0.0_dp
  ! bin width
  real(kind=dp), parameter :: fluid_vel_bin_width = 0.008_dp
  ! bin end
  real(kind=dp), parameter :: fluid_vel_bin_end = fluid_vel_bin_start &
       + n_fluid_vel_p_dist_bins * fluid_vel_bin_width
  ! time averaged probability distribution of velocities
  real(kind=dp), dimension(0:n_fluid_vel_p_dist_bins-1) :: &
       fluid_vel_p_dist_ave = 0.0_dp
  ! (X,Y)-plane average
  ! number of bins for the probability distribution of fluid velocities in
  ! xy-plane in thermostatting system
  integer, parameter :: n_fluid_vel_xy_pl_pdist_bins = 1400
  ! bin width
  real(kind=dp), parameter :: fluid_vel_xy_pl_bin_width = 0.008_dp
  ! bin start
  real(kind=dp), parameter :: fluid_vel_xy_pl_bin_start = 0.0_dp
  ! bin end
  real(kind=dp), parameter :: fluid_vel_xy_pl_bin_end = &
       fluid_vel_xy_pl_bin_start &
       + n_fluid_vel_xy_pl_pdist_bins * fluid_vel_xy_pl_bin_width
  ! time averaged probability distribution of velocities
  real(kind=dp), dimension(0:n_fluid_vel_xy_pl_pdist_bins-1) :: &
       fluid_vel_xy_pl_pdist_ave = 0.0_dp
  ! X
  ! number of bins for the probability distribution of fluid velocities in x
  integer, parameter :: n_fluid_vel_x_p_dist_bins = 2000
  ! bin width
  real(kind=dp), parameter :: fluid_vel_x_bin_width = 0.008_dp
  ! bin start
  real(kind=dp), parameter :: fluid_vel_x_bin_start = &
       - n_fluid_vel_x_p_dist_bins/2 * fluid_vel_x_bin_width
  ! bin end
  real(kind=dp), parameter :: fluid_vel_x_bin_end = &
       + n_fluid_vel_x_p_dist_bins/2 * fluid_vel_x_bin_width
  ! time averaged probability distribution of velocities
  real(kind=dp), dimension(0:n_fluid_vel_x_p_dist_bins-1) :: &
       fluid_vel_x_p_dist_ave = 0.0_dp
  ! Y
  ! number of bins for the probability distribution of fluid velocities in y
  integer, parameter :: n_fluid_vel_y_p_dist_bins = 2000
  ! bin width
  real(kind=dp), parameter :: fluid_vel_y_bin_width = 0.008_dp
  ! bin start
  real(kind=dp), parameter :: fluid_vel_y_bin_start = &
       - n_fluid_vel_y_p_dist_bins/2 * fluid_vel_y_bin_width
  ! bin end
  real(kind=dp), parameter :: fluid_vel_y_bin_end =  &
       + n_fluid_vel_y_p_dist_bins/2 * fluid_vel_y_bin_width
  ! time averaged probability distribution of velocities
  real(kind=dp), dimension(0:n_fluid_vel_y_p_dist_bins-1) :: &
       fluid_vel_y_p_dist_ave = 0.0_dp
  ! Z
  ! number of bins for the probability distribution of fluid velocities in z
  integer, parameter :: n_fluid_vel_z_p_dist_bins = 2000
  ! bin width
  real(kind=dp), parameter :: fluid_vel_z_bin_width = 0.008_dp
  ! bin start
  real(kind=dp), parameter :: fluid_vel_z_bin_start = &
       - n_fluid_vel_z_p_dist_bins/2 * fluid_vel_z_bin_width
  ! bin end
  real(kind=dp), parameter :: fluid_vel_z_bin_end =  &
       + n_fluid_vel_z_p_dist_bins/2 * fluid_vel_z_bin_width
  ! time averaged probability distribution of velocities
  real(kind=dp), dimension(0:n_fluid_vel_z_p_dist_bins-1) :: &
       fluid_vel_z_p_dist_ave = 0.0_dp
  ! angular velocity
  ! number of bins for the probability distribution of angular velocities
  integer, parameter :: n_bond_ang_vel_p_dist_bins = 8000
  ! bin width
  real(kind=dp), parameter :: bond_ang_vel_bin_width = 2.0_dp
  ! bin start
  real(kind=dp), parameter :: bond_ang_vel_bin_start = &
       - n_bond_ang_vel_p_dist_bins/2 * bond_ang_vel_bin_width
  ! bin end
  real(kind=dp), parameter :: bond_ang_vel_bin_end =  &
       + n_bond_ang_vel_p_dist_bins/2 * bond_ang_vel_bin_width
  ! time averaged probability distribution of velocities
  real(kind=dp), dimension(0:n_bond_ang_vel_p_dist_bins-1) :: &
       bond_ang_vel_p_dist_ave = 0.0_dp

contains
  !---------------------------------------------------------------------------
  ! get an array and parameters for a histogram and create histogram from
  ! the data in the input array, values are added up on out_histogram(:)
  ! using real histograms is faster if int to real conversion can be avoided
  ! by using them.
  subroutine bin_data_array(in_array, out_histogram, histogram_start, &
       histogram_bin_width, l_graceful_overflow)
    real(kind=dp), dimension(:), intent(in)    :: in_array
    real(kind=dp), dimension(:), intent(inout) :: out_histogram
    real(kind=dp), intent(in)                  :: histogram_start
    real(kind=dp), intent(in)                  :: histogram_bin_width
    logical, intent(in), optional              :: l_graceful_overflow
    integer :: bin_index, data_loop
    real(kind=dp) :: histogram_end
    ! from the dimension of the histogram we know the maximal value in the
    ! histogram
    histogram_end = real(ubound(out_histogram,1)-lbound(out_histogram,1) +1, &
         kind=dp) * histogram_bin_width + histogram_start
    if(l_debug_bin_data_array) then
      print *, "DEBUG: Entering SR bin_data_array"
      print *, "histogram_start=", histogram_start
      print *, "histogram_bin_width=", histogram_bin_width
      print *, "histogram_end=", histogram_end
      print *, "lbound=", lbound(out_histogram,1), &
           ", ubound=", ubound(out_histogram,1)
    end if
    ! the dimension has to be specified in the calls to lbound, so that the
    ! compiler knows that the result is a scalar
    ! loop over input data
    do data_loop = lbound(in_array, 1), ubound(in_array, 1)
      ! get index of histogram
      bin_index = int((in_array(data_loop) - histogram_start) &
           / histogram_bin_width) + lbound(out_histogram, 1)
      ! if data is in valid range
      if(lbound(out_histogram,1) .le. bin_index &
           .and. bin_index .le. ubound(out_histogram,1)) then
        ! increment histogram
        out_histogram(bin_index) = out_histogram(bin_index) + 1.0_dp
        if(l_debug_bin_data_array) then
          print *, "put", in_array(data_loop), " to bin_index", bin_index, &
               ", borders:", &
               histogram_start + histogram_bin_width &
               * real(bin_index - lbound(in_array, 1)), &
               ", ", histogram_start + histogram_bin_width &
               * real(bin_index +1 -lbound(in_array, 1))
        end if
      else
        if(present(l_graceful_overflow)) then
          ! if histogram overflows are rare we can put an equivalent number of
          ! particles in the outermost bins as last resort.  This way, the
          ! computation of the moments if only slightly affected.
          ! If one boundary is 0 then artefacts will result.
          ! (Like negative entries)
          if(l_graceful_overflow) then
            if(bin_index .gt. ubound(out_histogram,1)) then
              out_histogram(ubound(out_histogram, 1)) &
                   = out_histogram(ubound(out_histogram, 1))&
                   + anint(in_array(data_loop) &
                   / (histogram_end - 0.5_dp*histogram_bin_width))
            else
              out_histogram(lbound(out_histogram, 1)) &
                   = out_histogram(lbound(out_histogram, 1)) &
                   + anint(in_array(data_loop) &
                   / (histogram_start + 0.5_dp*histogram_bin_width))
            end if
          end if ! l_graceful_overflow
        end if ! present(l_graceful_overflow)
      end if ! input value in valid range
    end do ! loop over each input value
    if(l_debug_bin_data_array) then
      print *, "DEBUG: Leaving SR bin_data_array"
    end if
  end subroutine bin_data_array
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! calculate the probability distribution of the fluid particle velocities
  subroutine fluid_vel_p_dist(out_file, time_constant, style)
    character(len=*), intent(in) :: out_file
    integer, intent(in)          :: time_constant
    character(len=*), intent(in) :: style
    ! helper array holding the moduli of the velocities
    real(kind=dp), dimension(1:n_mon_tot) :: mod_velocity
    ! start and end indices to be considered in mod_velocity
    integer :: vel_begin_index, vel_end_index
    integer :: i_part, i_bond, i_chain, i_mon
    real(kind=dp), dimension(n_dim) :: vel_cm
    ! time in MDS when this routine was first called
    integer, save :: first_call_time = -huge(1)
    ! averaged moments
    real(kind=dp), save :: fl_vel_pd_average_ave, &
         fl_vel_pd_second_moment_ave, &
         fl_vel_pd_third_moment_ave, &
         fl_vel_pd_forth_moment_ave
    if(l_debug_fluid_vel_p_dist) &
         print *, "DEBUG: Entering SR fluid_vel_p_dist "
    ! sanity check
    if(n_mon_tot .eq. 0) then
      write(unit=*, fmt=*) &
           "ERROR (SR  fluid_vel_p_dist): n_mon_tot .eq. 0, nothing to do."
      stop
    end if
    if(time_constant .le. 0) then
      write(unit=*, fmt='(a)') &
           "ERROR (SR  fluid_vel_p_dist): time_constant must be positive"
      stop
    end if
    if(.not. (style .eq. "moments_only" .or. style .eq. "prob_dist" &
         .or. style .eq. "ave_prob_dist")) then
      write(unit=*, fmt='(3a)') &
           "ERROR (SR fluid_vel_p_dist): style=", style, &
           " of data output not recognized"
      stop
    end if

    ! set switch fo going to center of mass for 2-mers
    if(n_mon .eq. 2) then
      l_take_cm_of_bimer = .TRUE.
    else
      l_take_cm_of_bimer = .FALSE.
    end if
    if(l_take_cm_of_bimer) then
      vel_begin_index = 1
      vel_end_index   = n_chain * (n_mon -1)
      ! get center of mass velocities, one for each bond
      i_bond = 0
      do i_chain = 1, n_chain
        do i_mon = 1, n_mon-1
          i_bond = i_bond +1
          vel_cm(:) = rx((i_chain-1)*n_mon + i_mon, :, 1) &
               + rx((i_chain-1)*n_mon + i_mon +1, :, 1)
          ! times 0.5 for the average of the two particles
          mod_velocity(i_bond) = &
               0.5_dp * dt_inv * sqrt(dot_product(vel_cm(:), vel_cm(:)))
        end do
      end do
    else
      vel_begin_index = 1
      vel_end_index   = n_mon_tot
      ! rx(:, :, 1) stores velocity * dt of particles
      do i_part = vel_begin_index, vel_end_index
        mod_velocity(i_part) = &
             dt_inv * sqrt(dot_product(rx(i_part, :, 1), rx(i_part, :, 1)))
      end do
    end if
    if(style .ne. "moments_only") then
      if(maxval(mod_velocity(vel_begin_index:vel_end_index)) &
           .gt. fluid_vel_bin_end) then
        write(unit=*, fmt='(a, g13.6, a, g13.6)') &
             "WARNING (SR fluid_vel_p_dist): max. |v|=", &
             maxval(mod_velocity(vel_begin_index:vel_end_index)), &
             " gt. fluid_vel_bin_end=", fluid_vel_bin_end
      end if
    end if
    if(style .ne. "ave_prob_dist") then
```

```
    ! get probability distribution, average data and write out
    call p_dist_calc_and_write(out_file, time_constant, first_call_time, &
         style, "fluid velocity probability distribution", &
         "time | average | sigma | 3rd moment | 4th moment", &
         "time | average | sigma | 3rd moment | 4th moment &
         &| P(0) | P(1) | ... | P(n) |...", &
         mod_velocity(vel_begin_index:vel_end_index), fluid_vel_p_dist_ave, &
         fluid_vel_bin_start, fluid_vel_bin_end, fluid_vel_bin_width, &
         fl_vel_pd_average_ave, fl_vel_pd_second_moment_ave, &
         fl_vel_pd_third_moment_ave, fl_vel_pd_forth_moment_ave)
  else
    call bin_data_array(mod_velocity(vel_begin_index:vel_end_index), &
         fluid_vel_p_dist_ave, fluid_vel_bin_start, fluid_vel_bin_width, &
         .TRUE.)
  end if
  if(l_debug_fluid_vel_p_dist) &
       print *, "DEBUG: Leaving SR fluid_vel_p_dist "
end subroutine fluid_vel_p_dist
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! calculate the probability distribution of the fluid particle velocities
! in the xy-plane (radially averaged)
subroutine fluid_vel_xy_pl_pdist(out_file, time_constant, style)
  character(len=*), intent(in) :: out_file
  integer, intent(in)          :: time_constant
  character(len=*), intent(in) :: style
  ! helper array holding the moduli of the velocities
  real(kind=dp), dimension(1:n_mon_tot) :: mod_velocity
  ! start and end indices to be considered in mod_velocity
  integer :: vel_begin_index, vel_end_index
  integer :: i_part, i_bond, i_chain, i_mon
  real(kind=dp), dimension(n_dim) :: vel_cm
  ! time in MDS when this routine was first called
  integer, save :: first_call_time = -huge(1)
  ! averaged moments
  real(kind=dp), save :: fl_vel_xy_pl_pd_avg_ave, &
       fl_vel_xy_pl_pd_second_mom_ave, &
       fl_vel_xy_pl_pd_third_mom_ave, &
       fl_vel_xy_pl_pd_forth_mom_ave
  if(l_debug_fluid_vel_xy_pl_pdist) &
       print *, "DEBUG: Entering SR fluid_vel_xy_pl_pdist "
  ! sanity check
  if(n_mon_tot .eq. 0) then
    write(unit=*, fmt=*) &
         "ERROR (SR  fluid_vel_xy_pl_pdist): n_mon_tot.eq.0, nothing to do."
    stop
  end if
  if(time_constant .le. 0) then
    write(unit=*, fmt='(a)') &
         "ERROR (SR  fluid_vel_xy_pl_pdist): time_constant must be positive"
    stop
  end if
  if(.not. (style .eq. "moments_only" .or. style .eq. "prob_dist" &
       .or. style .eq. "ave_prob_dist")) then
    write(unit=*, fmt='(3a)') &
         "ERROR (SR fluid_vel_xy_pl_pdist): style=", style, &
         " of data output not recognized"
    stop
  end if
  ! set switch fo going to center of mass for 2-mers
  if(n_mon .eq. 2) then
    l_take_cm_of_bimer = .TRUE.
  else
    l_take_cm_of_bimer = .FALSE.
  end if
  if(l_take_cm_of_bimer) then
    vel_begin_index = 1
    vel_end_index   = n_chain * (n_mon -1)
    ! get center of mass velocities in xy-plane, one for each bond
    i_bond = 0
    do i_chain = 1, n_chain
      do i_mon = 1, n_mon-1
        i_bond = i_bond +1
        vel_cm(1:2) = rx((i_chain-1)*n_mon + i_mon, 1:2, 1) &
             + rx((i_chain-1)*n_mon + i_mon +1, 1:2, 1)
        ! substract the average motion of the fluid
        vel_cm(1:2) = vel_cm(1:2) - 0.5_dp*rx_twall(1:2, 1)
        ! times 0.5 for the average of the two particles
        mod_velocity(i_bond) = &
             0.5_dp*dt_inv*sqrt(dot_product(vel_cm(1:2), vel_cm(1:2)))
      end do
    end do
  else
    vel_begin_index = 1
    vel_end_index   = n_mon_tot
    ! rx(:, :, 1) stores velocity * dt of particles
    do i_part = vel_begin_index, vel_end_index
      ! substract the average motion of the fluid
      mod_velocity(i_part) = dt_inv &
           * sqrt(dot_product(rx(i_part, 1:2, 1) - 0.5_dp*rx_twall(1:2, 1), &
           rx(i_part, 1:2, 1) - 0.5_dp*rx_twall(1:2, 1)))
    end do
  end if
  if(style .ne. "moments_only") then
    if(maxval(mod_velocity(vel_begin_index:vel_end_index)) &
         .gt. fluid_vel_xy_pl_bin_end) then
      write(unit=*, fmt='(a, g13.6, a, g13.6)') &
           "WARNING (SR fluid_vel_xy_pl_pdist): max. |v|=", &
           maxval(mod_velocity(vel_begin_index:vel_end_index)), &
           " gt. fluid_vel_xy_pl_bin_end=", fluid_vel_xy_pl_bin_end
    end if
  end if
  if(style .ne. "ave_prob_dist") then
    ! get probability distribution, average data and write out
    call p_dist_calc_and_write(out_file, time_constant, first_call_time, &
         style, "fluid velocity in xy-plane probability distribution", &
         "time | average | sigma | 3rd moment | 4th moment", &
         "time | average | sigma | 3rd moment | 4th moment &
         &| P(0) | P(1) | ... | P(n) |...", &
         mod_velocity(vel_begin_index:vel_end_index), &
         fluid_vel_xy_pl_pdist_ave, &
         fluid_vel_xy_pl_bin_start, fluid_vel_xy_pl_bin_end, &
         fluid_vel_xy_pl_bin_width, &
         fl_vel_xy_pl_pd_avg_ave, fl_vel_xy_pl_pd_second_mom_ave, &
         fl_vel_xy_pl_pd_third_mom_ave, fl_vel_xy_pl_pd_forth_mom_ave)
  else
    call bin_data_array(mod_velocity(vel_begin_index:vel_end_index), &
         fluid_vel_xy_pl_pdist_ave, fluid_vel_xy_pl_bin_start, &
         fluid_vel_xy_pl_bin_width, &
         .TRUE.)
  end if
  if(l_debug_fluid_vel_xy_pl_pdist) &
       print *, "DEBUG: Leaving SR fluid_vel_xy_pl_pdist "
end subroutine fluid_vel_xy_pl_pdist
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! calculate the probability distribution of the fluid particle angular
! velocities in the center of mass of the bond
subroutine bond_ang_vel_p_dist(out_file, time_constant, style)
  character(len=*), intent(in) :: out_file
  integer, intent(in)          :: time_constant
  character(len=*), intent(in) :: style
  ! helper array holding the velocities
  real(kind=dp), dimension(1:n_chain*(n_mon-1)) :: velocity
  integer :: i_mon, i_chain, i_bond
  ! time in MDS when this routine was first called
  integer, save :: first_call_time = -huge(1)
  ! averaged moments
  real(kind=dp), save :: bond_av_pd_average_ave, &
       bond_av_pd_second_moment_ave, &
       bond_av_pd_third_moment_ave, &
       bond_av_pd_forth_moment_ave
  if(l_debug_bond_ang_vel_p_dist) &
       print *, "DEBUG: Entering SR bond_ang_vel_p_dist "
  ! sanity check
  if(n_mon .lt. 2) then
    write(unit=*, fmt=*) &
         "ERROR (SR  bond_ang_vel_p_dist): n_mon .lt. 2, nothing to do."
    stop
  end if
  if(n_mon_tot .eq. 0) then
    write(unit=*, fmt=*) &
         "ERROR (SR  bond_ang_vel_p_dist): n_mon_tot .eq. 0, nothing to do."
    stop
  end if
  if(time_constant .le. 0) then
    write(unit=*, fmt='(a)') &
         "ERROR (SR  bond_ang_vel_p_dist): time_constant must be positive"
    stop
  end if
  if(.not. (style .eq. "moments_only" .or. style .eq. "prob_dist" &
       .or. style .eq. "ave_prob_dist")) then
    write(unit=*, fmt='(3a)') &
         "ERROR (SR bond_ang_vel_p_dist): style=", style, &
         " of data output not recognized"
    stop
  end if
  ! get angular velocities, one for each bond
  i_bond = 0
  do i_chain = 1, n_chain
    do i_mon = 1, n_mon-1
      i_bond = i_bond +1
      velocity(i_bond) = angul_vel((i_chain-1)*n_mon + i_mon, &
           (i_chain-1)*n_mon + i_mon +1)
    end do
  end do
  if(style .ne. "moments_only") then
    if(maxval(velocity(:)) .gt. bond_ang_vel_bin_end) then
      write(unit=*, fmt='(a, g13.6, a, g13.6)') &
           "WARNING (SR bond_ang_vel_p_dist): max. v=", &
           maxval(velocity(:)), " .gt. bond_ang_vel_bin_end=", &
           bond_ang_vel_bin_end
    end if
    if(minval(velocity(:)) .lt. bond_ang_vel_bin_start) then
      write(unit=*, fmt='(a, g13.6, a, g13.6)') &
           "WARNING (SR bond_ang_vel_p_dist): min. v=", &
           minval(velocity(:)), " .lt. bond_ang_vel_bin_start=", &
           bond_ang_vel_bin_start
    end if
  end if
  if(style .ne. "ave_prob_dist") then
    ! get probability distribution, average data and write out
    call p_dist_calc_and_write(out_file, time_constant, first_call_time, &
         style, "fluid velocity probability distribution", &
         "time | average | sigma | 3rd moment | 4th moment", &
         "time | average | sigma | 3rd moment | 4th moment &
         &| P(0) | P(1) | ... | P(n) |...", &
         velocity, bond_ang_vel_p_dist_ave, &
         bond_ang_vel_bin_start, bond_ang_vel_bin_end, &
         bond_ang_vel_bin_width, &
         bond_av_pd_average_ave, bond_av_pd_second_moment_ave, &
         bond_av_pd_third_moment_ave, bond_av_pd_forth_moment_ave)
  else
    call bin_data_array(velocity, bond_ang_vel_p_dist_ave, &
        bond_ang_vel_bin_start, bond_ang_vel_bin_width, .TRUE.)
  end if
  if(l_debug_bond_ang_vel_p_dist) &
       print *, "DEBUG: Leaving SR bond_ang_vel_p_dist "
end subroutine bond_ang_vel_p_dist
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! calculate the probability distribution of the fluid particle velocities
! in x and y
subroutine fluid_vel_xyz_p_dist(out_file_x, out_file_y, out_file_z, &
     time_constant, style)
  character(len=*), intent(in) :: out_file_x, out_file_y, out_file_z
  integer, intent(in)          :: time_constant
  character(len=*), intent(in) :: style
  ! helper array holding the velocities
  real(kind=dp), dimension(1:n_mon_tot) :: velocity
  ! start and end indices to be considered in mod_velocity
  integer :: vel_begin_index, vel_end_index
  integer :: i_part, i_bond, i_chain, i_mon
  ! time in MDS when this routine was first called
  integer, save :: first_call_time = -huge(1)
  ! averaged moments
  real(kind=dp), save :: fl_vel_x_pd_average_ave, &
       fl_vel_x_pd_second_moment_ave, &
       fl_vel_x_pd_third_moment_ave, &
       fl_vel_x_pd_forth_moment_ave
  real(kind=dp), save :: fl_vel_y_pd_average_ave, &
       fl_vel_y_pd_second_moment_ave, &
       fl_vel_y_pd_third_moment_ave, &
       fl_vel_y_pd_forth_moment_ave
  real(kind=dp), save :: fl_vel_z_pd_average_ave, &
       fl_vel_z_pd_second_moment_ave, &
       fl_vel_z_pd_third_moment_ave, &
       fl_vel_z_pd_forth_moment_ave
  if(l_debug_fluid_vel_xyz_p_dist) &
       print *, "DEBUG: Entering SR fluid_vel_xyz_p_dist "
  ! sanity check
  if(n_mon_tot .eq. 0) then
    write(unit=*, fmt=*) &
         "ERROR (SR  fluid_vel_xyz_p_dist): n_mon_tot .eq. 0, nothing to do."
    stop
  end if
  if(time_constant .le. 0) then
    write(unit=*, fmt='(a)') &
         "ERROR (SR  fluid_vel_xyz_p_dist): time_constant must be positive"
    stop
  end if
  if(.not. (style .eq. "moments_only" .or. style .eq. "prob_dist" &
       .or. style .eq. "ave_prob_dist")) then
    write(unit=*, fmt='(3a)') &
         "ERROR (SR fluid_vel_xyz_p_dist): style=", style, &
         " of data output not recognized"
    stop
  end if
```

```
! set switch fo going to center of mass for 2-mers
if(n_mon .eq. 2) then
  l_take_cm_of_bimer = .TRUE.
else
  l_take_cm_of_bimer = .FALSE.
end if

if(l_take_cm_of_bimer) then
  vel_begin_index = 1
  vel_end_index   = n_chain * (n_mon -1)
else
  vel_begin_index = 1
  vel_end_index   = n_mon_tot
end if
! X
if(l_take_cm_of_bimer) then
  ! get center of mass velocities, one for each bond
  i_bond = 0
  do i_chain = 1, n_chain
    do i_mon = 1, n_mon-1
      i_bond = i_bond +1
      velocity(i_bond) = 0.5_dp * dt_inv &
           * (rx((i_chain-1)*n_mon + i_mon, 1, 1) &
           + rx((i_chain-1)*n_mon + i_mon +1, 1, 1))
    end do
  end do
else
  ! rescale with dt
  velocity(vel_begin_index:vel_end_index) &
       = dt_inv * rx(vel_begin_index:vel_end_index, 1, 1)
end if
if(style .ne. "moments_only") then
  if(maxval(velocity(vel_begin_index:vel_end_index)) &
       .gt. fluid_vel_x_bin_end) then
    write(unit=*, fmt='(a, g13.6, a, g13.6)') &
         "WARNING (SR fluid_vel_xyz_p_dist): max. v=", &
         maxval(velocity(vel_begin_index:vel_end_index)), &
         " .gt. fluid_vel_x_bin_end=", fluid_vel_x_bin_end
  end if
  if(minval(velocity(vel_begin_index:vel_end_index)) &
       .lt. fluid_vel_x_bin_start) then
    write(unit=*, fmt='(a, g13.6, a, g13.6)') &
         "WARNING (SR fluid_vel_xyz_p_dist): min. v=", &
         minval(velocity(vel_begin_index:vel_end_index)), &
         " .lt. fluid_vel_x_bin_start=", fluid_vel_x_bin_start
  end if
end if
if(style .ne. "ave_prob_dist") then
  ! get probability distribution, average data and write out
  call p_dist_calc_and_write(out_file_x, time_constant, first_call_time, &
       style, "fluid velocity probability distribution", &
       "time | average | sigma | 3rd moment | 4th moment", &
       "time | average | sigma | 3rd moment | 4th moment &
       &| P(0) | P(1) | ... | P(n) |...", &
       velocity(vel_begin_index:vel_end_index), fluid_vel_x_p_dist_ave, &
       fluid_vel_x_bin_start, fluid_vel_x_bin_end, fluid_vel_x_bin_width, &
       fl_vel_x_pd_average_ave, fl_vel_x_pd_second_moment_ave, &
       fl_vel_x_pd_third_moment_ave, fl_vel_x_pd_forth_moment_ave)
else
  call bin_data_array(velocity(vel_begin_index:vel_end_index), &
       fluid_vel_x_p_dist_ave, fluid_vel_x_bin_start, &
       fluid_vel_x_bin_width, .TRUE.)
end if

! Y
if(l_take_cm_of_bimer) then
  ! get center of mass velocities, one for each bond
  i_bond = 0
  do i_chain = 1, n_chain
     do i_mon = 1, n_mon-1
       i_bond = i_bond +1
       velocity(i_bond) = (rx((i_chain-1)*n_mon + i_mon, 2, 1) &
            + rx((i_chain-1)*n_mon + i_mon +1, 2, 1))/(2.0_dp*dt)
     end do
  end do
else
  ! rescale with dt
  velocity(vel_begin_index:vel_end_index) &
       = dt_inv * rx(vel_begin_index:vel_end_index, 2, 1)
end if
if(style .ne. "moments_only") then
  if(maxval(velocity(vel_begin_index:vel_end_index)) &
       .gt. fluid_vel_y_bin_end) then
    write(unit=*, fmt='(a, g13.6, a, g13.6)') &
         "WARNING (SR fluid_vel_xyz_p_dist): max. v=", &
         maxval(velocity(vel_begin_index:vel_end_index)), &
         " .gt. fluid_vel_y_bin_end=", fluid_vel_y_bin_end
  end if
  if(minval(velocity(vel_begin_index:vel_end_index)) &
       .lt. fluid_vel_y_bin_start) then
    write(unit=*, fmt='(a, g13.6, a, g13.6)') &
         "WARNING (SR fluid_vel_xyz_p_dist): min. v=", &
         minval(velocity(vel_begin_index:vel_end_index)), &
         " .lt. fluid_vel_y_bin_start=", fluid_vel_y_bin_start
  end if
end if
if(style .ne. "ave_prob_dist") then
  ! get probability distribution, average data and write out
  call p_dist_calc_and_write(out_file_y, time_constant, first_call_time, &
       style, "fluid velocity probability distribution", &
       "time | average | sigma | 3rd moment | 4th moment", &
       "time | average | sigma | 3rd moment | 4th moment &
       &| P(0) | P(1) | ... | P(n) |...", &
       velocity(vel_begin_index:vel_end_index), fluid_vel_y_p_dist_ave, &
       fluid_vel_y_bin_start, fluid_vel_y_bin_end, fluid_vel_y_bin_width, &
       fl_vel_y_pd_average_ave, fl_vel_y_pd_second_moment_ave, &
       fl_vel_y_pd_third_moment_ave, fl_vel_y_pd_forth_moment_ave)
else
  call bin_data_array(velocity(vel_begin_index:vel_end_index), &
       fluid_vel_y_p_dist_ave, fluid_vel_y_bin_start, &
       fluid_vel_y_bin_width, .TRUE.)
end if

! Z
if(l_take_cm_of_bimer) then
  ! get center of mass velocities, one for each bond
  i_bond = 0
  do i_chain = 1, n_chain
    do i_mon = 1, n_mon-1
      i_bond = i_bond +1
      velocity(i_bond) = (rx((i_chain-1)*n_mon + i_mon, n_dim, 1) &
           + rx((i_chain-1)*n_mon + i_mon +1, n_dim, 1))/(2.0_dp*dt)
    end do
  end do
else
  ! rescale with dt
  velocity(vel_begin_index:vel_end_index) &
       = dt_inv * rx(vel_begin_index:vel_end_index, n_dim, 1)
end if
if(style .ne. "moments_only") then
  if(maxval(velocity(vel_begin_index:vel_end_index)) &
       .gt. fluid_vel_z_bin_end) then
    write(unit=*, fmt='(a, g13.6, a, g13.6)') &
         "WARNING (SR fluid_vel_xyz_p_dist): max. v=", &
         maxval(velocity(vel_begin_index:vel_end_index)), &
         " .gt. fluid_vel_z_bin_end=", fluid_vel_z_bin_end
  end if
  if(minval(velocity(vel_begin_index:vel_end_index)) &
       .lt. fluid_vel_z_bin_start) then
    write(unit=*, fmt='(a, g13.6, a, g13.6)') &
         "WARNING (SR fluid_vel_xyz_p_dist): min. v=", &
         minval(velocity(vel_begin_index:vel_end_index)), &
         " .lt. fluid_vel_z_bin_start=", fluid_vel_z_bin_start
  end if
end if
if(style .ne. "ave_prob_dist") then
  ! get probability distribution, average data and write out
  call p_dist_calc_and_write(out_file_z, time_constant, first_call_time, &
       style, "fluid velocity probability distribution", &
       "time | average | sigma | 3rd moment | 4th moment", &
       "time | average | sigma | 3rd moment | 4th moment &
       &| P(0) | P(1) | ... | P(n) |...", &
       velocity(vel_begin_index:vel_end_index), fluid_vel_z_p_dist_ave, &
       fluid_vel_z_bin_start, fluid_vel_z_bin_end, fluid_vel_z_bin_width, &
       fl_vel_z_pd_average_ave, fl_vel_z_pd_second_moment_ave, &
       fl_vel_z_pd_third_moment_ave, fl_vel_z_pd_forth_moment_ave)
else
  call bin_data_array(velocity(vel_begin_index:vel_end_index), &
       fluid_vel_z_p_dist_ave, fluid_vel_z_bin_start, &
       fluid_vel_z_bin_width, .TRUE.)
end if

if(l_debug_fluid_vel_xyz_p_dist) &
     print *, "DEBUG: Leaving SR fluid_vel_xyz_p_dist "
end subroutine fluid_vel_xyz_p_dist
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! calculate the probability distribution of the fluid particle accelerations
subroutine fluid_accel_p_dist(out_file, time_constant, style)
  character(len=*), intent(in) :: out_file
  integer, intent(in)          :: time_constant
  character(len=*), intent(in) :: style
  ! helper array holding the moduli of the accelerations
  real(kind=dp), dimension(1:n_mon_tot) :: mod_accel
  integer :: i_part
  ! time in MDS when this routine was first called
  integer, save :: first_call_time = -huge(1)
  ! averaged moments
  real(kind=dp), save :: fl_accel_pd_average_ave, &
       fl_accel_pd_second_moment_ave, &
       fl_accel_pd_third_moment_ave, &
       fl_accel_pd_forth_moment_ave
  ! number of bins for the probability distribution of fluid velocities
  ! if the number of binning boxes is very large, it makes no sense to write
  ! them all out
  integer, parameter :: n_fluid_accel_p_dist_bins = 3000
  ! bin start
  real(kind=dp), parameter :: fluid_accel_bin_start = 0.0_dp
  ! bin width
  real(kind=dp), parameter :: fluid_accel_bin_width = 0.25_dp
  ! bin end
  real(kind=dp), parameter :: fluid_accel_bin_end = fluid_accel_bin_start &
       + n_fluid_accel_p_dist_bins * fluid_accel_bin_width
  ! time averaged probability distribution of velocities
  real(kind=dp), dimension(0:n_fluid_accel_p_dist_bins-1), save :: &
       fluid_accel_p_dist_ave
  ! if overflows of the histograms should be handled gracefully.  That is if
  ! a particle is replaced by entries in the outermost bins.  If this is
  ! only a rare event we can save a lot of bins.
  logical, parameter :: l_graceful_overflow = .TRUE.
  if(l_debug_fluid_accel_p_dist) &
       print *, "DEBUG: Entering SR fluid_accel_p_dist "
  ! sanity check
  if(n_mon_tot .eq. 0) then
    write(unit=*, fmt=*) &
         "ERROR (SR  fluid_accel_p_dist): n_mon_tot .eq. 0, nothing to do."
    stop
  end if
  if(time_constant .le. 0) then
    write(unit=*, fmt='(a)') &
         "ERROR (SR  fluid_accel_p_dist): time_constant must be positive"
    stop
  end if
  if(.not. (style .eq. "moments_only" .or. style .eq. "prob_dist")) then
    write(unit=*, fmt='(3a)') &
         "ERROR (SR  fluid_accel_p_dist): style=", style, &
         " of data output not recognized"
    stop
  end if
  ! rx(:, :, 2) stores acceleration * dt^2/2 of particles
  do i_part = 1, n_mon_tot
     mod_accel(i_part) = &
          sqrt(dot_product(rx(i_part, :, 2), rx(i_part, :, 2)))
   end do
   ! rescale
   mod_accel(:) = 2.0_dp * mod_accel(:)/dt_2
   if(style .eq. "prob_dist") then
     if(maxval(mod_accel(:)) .gt. fluid_accel_bin_end) then
       write(unit=*, fmt='(a, g13.6, a, g13.6)') &
            "WARNING (SR fluid_accel_p_dist ): max. |a|=", &
            maxval(mod_accel(:)), " gt. fluid_accel_bin_end=", &
            fluid_accel_bin_end
     end if
   end if
   ! get probabilty distribution, average data and write out
   call p_dist_calc_and_write(out_file, time_constant, first_call_time, &
     style, "fluid acceleration probability distribution",  &
     "time | average | sigma | 3rd moment | 4th moment", &
     "time | average | sigma | 3rd moment | 4th moment &
     &| P(0) | P(1) | ... | P(n) |...", &
     mod_accel, fluid_accel_p_dist_ave, &
     fluid_accel_bin_start, fluid_accel_bin_end, fluid_accel_bin_width, &
     fl_accel_pd_average_ave, fl_accel_pd_second_moment_ave, &
     fl_accel_pd_third_moment_ave, fl_accel_pd_forth_moment_ave, &
     l_graceful_overflow)
  if(l_debug_fluid_accel_p_dist) &
       print *, "DEBUG: Leaving SR fluid_accel_p_dist "
end subroutine fluid_accel_p_dist
!------------------------------------------------------------------------------


!------------------------------------------------------------------------------
! calculate the probability distribution of a given histogram, as well as
! centralized moments and writes them out
subroutine p_dist_calc_and_write(out_file, time_constant, first_call_time, &
     style, header, string_style_one, string_style_two, &
     in_value_array, histogram_ave, histogram_ave_start, histogram_ave_end, &
     histogram_ave_bin_width, average_ave, second_moment_ave, &
     third_moment_ave, forth_moment_ave, l_graceful_overflow)
  character(len=*), intent(in) :: out_file
  integer, intent(in)          :: time_constant
  character(len=*), intent(in) :: style
  ! time in MDS when this routine was first called
  integer, intent(inout) :: first_call_time
  character(len=*), intent(in) :: header, string_style_one, &
       string_style_two
  real(kind=dp), dimension(:), intent(in) :: in_value_array
  real(kind=dp), dimension(:), intent(inout) :: histogram_ave
```

```
real(kind=dp), dimension(size(histogram_ave)) :: single_histogram
real(kind=dp), intent(in) :: histogram_ave_start, histogram_ave_end, &
     histogram_ave_bin_width
! averaged moments
real(kind=dp), intent(inout) :: average_ave, second_moment_ave, &
     third_moment_ave, forth_moment_ave
logical, intent(in), optional :: l_graceful_overflow
! single time moments
real(kind=dp) :: normalization, average, variance, second_moment, &
     third_moment, forth_moment
! I/O stuff
integer, parameter :: fileunit=61
integer :: iostatus
character(len=20000) :: line
character(len=40) :: word
real(kind=dp) :: r_dummy
integer :: i_histogram_bin
! sanity check
if(time_constant .le. 0) then
  write(unit=*, fmt='(a)') &
       "ERROR (SR  p_dist_calc_and_write): time_constant must be positive"
  stop
end if
if(.not. (style .eq. "moments_only" .or. style .eq. "prob_dist")) then
  write(unit=*, fmt='(3a)') &
       "ERROR (SR p_dist_calc_and_write): style=", style, &
       " of data output not recognized"
  stop
end if
if(style .eq. "prob_dist") then
  if(abs(real(size(histogram_ave, 1),kind=dp) * histogram_ave_bin_width) &
       - (histogram_ave_end - histogram_ave_start) &
       .gt. 10.0_dp**(-precision(1.0_dp)+1)) then
    write(unit=*, fmt='(a, g13.6, a, g13.6)') &
         "ERROR  (SR p_dist_calc_and_write): Size of histogram problem:", &
         "size(histogram_ave, 1)) * histogram_ave_bin_width)=", &
         real(size(histogram_ave, 1), kind=dp) * histogram_ave_bin_width, &
         ", but histogram_ave_end - histogram_ave_start=", &
         histogram_ave_end - histogram_ave_start
    stop
  end if
  ! bin data
  single_histogram(:) = 0.0_dp
  if(present(l_graceful_overflow)) then
    call bin_data_array(in_value_array, single_histogram, &
         histogram_ave_start, histogram_ave_bin_width, l_graceful_overflow)
  else
    call bin_data_array(in_value_array, single_histogram, &
         histogram_ave_start, histogram_ave_bin_width)
  end if
  ! initialize averaged probability_distribution if necessary and write out
  if(first_call_time .eq. -huge(1)) then
    first_call_time = i_time
    histogram_ave(:) = single_histogram(:)
    ! open file for replacement
    open(unit=fileunit, file=out_file, status="replace", &
         action="write", iostat=iostatus)
    if(iostatus .ne. 0) then
      write (unit=*, fmt=*) "WARNING: Could not open output file >>", &
           out_file, "<<"
    else
      ! write out preamble
      write(unit=*, fmt='(3a)') &
           "MESSAGE: Writing probability distribution to >>", out_file,"<<"
      write(unit=fileunit, fmt='(a)') "# "//header
      write(unit=fileunit, fmt='(a, g13.6)') &
           "# bin start = ", histogram_ave_start
      write(unit=fileunit, fmt='(a, g13.6)') &
           "# bin end = ",  histogram_ave_end
      write(unit=fileunit, fmt='(a, g13.6)') &
           "# bin width = ", histogram_ave_bin_width
      write(unit=fileunit, fmt='(a, i7)') "# number of bins:", &
           size(histogram_ave, 1)
      write(unit=fileunit, fmt='(a)') "# "//string_style_two
      close(fileunit)
    end if ! (iostatus .ne. 0)
  else ! average
    if(time_constant .gt. 1) then
      histogram_ave(:) = &
           (1.0_dp-1.0_dp/real(time_constant,kind=dp)) * histogram_ave(:)& 
           + (1.0_dp/real(time_constant,kind=dp)) * single_histogram(:)
    else
      histogram_ave(:) = single_histogram(:)
    end if
  end if ! first call
else
  ! compute moments only.  In this case we don't need to bin the values
  ! (thus saving CPU time)
  ! initialize averaged moments
  if(first_call_time .eq. -huge(1)) then
    first_call_time = i_time
    call moments(in_value_array, average_ave, &
         second_moment_ave, third_moment_ave, forth_moment_ave)
    ! open file for replacement
    open(unit=fileunit, file=out_file, status="replace", &
         action="write", iostat=iostatus)
    if(iostatus .ne. 0) then
      write (unit=*, fmt=*) "WARNING: Could not open output file >>", &
           out_file, "<<"
    else
      ! write out preamble
      write(unit=*, fmt='(3a)') &
           "MESSAGE: Writing moments of prob. dist. to >>", out_file, "<<"
      write(unit=fileunit, fmt='(a)') "# "//header
      write(unit=fileunit, fmt='(a)') "# "//string_style_one
      close(fileunit)
    end if ! (iostatus .ne. 0)
  else ! average
    call moments(in_value_array, average, &
         second_moment, third_moment, forth_moment)
    if(time_constant .gt. 1) then
      average_ave = &
           (1.0_dp-1.0_dp/real(time_constant,kind=dp)) * average_ave &
           + (1.0_dp/real(time_constant,kind=dp)) * average
      second_moment_ave = &
           (1.0_dp-1.0_dp/real(time_constant,kind=dp)) *second_moment_ave &
           + (1.0_dp/real(time_constant,kind=dp)) * second_moment
      third_moment_ave = &
           (1.0_dp-1.0_dp/real(time_constant,kind=dp)) * third_moment_ave &
           + (1.0_dp/real(time_constant,kind=dp)) * third_moment
      forth_moment_ave = &
           (1.0_dp-1.0_dp/real(time_constant,kind=dp)) * forth_moment_ave &
           + (1.0_dp/real(time_constant,kind=dp)) * forth_moment
    else
      average_ave = average
      second_moment_ave = second_moment
      third_moment_ave = third_moment
      forth_moment_ave = forth_moment
    end if
  end if ! first call
end if !(style .eq. "prob_dist")

! when the time has come to write out...
if(mod((i_time - first_call_time), time_constant) .eq. 0) then
  if(style .eq. "prob_dist") then
    ! compute normalization and average
    normalization = 0.0_dp
    average_ave = 0.0_dp
    do i_histogram_bin = lbound(histogram_ave, 1), ubound(histogram_ave, 1)
      ! this normalization does not include the bin width.  For the moments
      ! we don't need that, but later for the probabilities
      normalization = normalization &
           + histogram_ave(i_histogram_bin)
      average_ave = average_ave + histogram_ave(i_histogram_bin) &
           * (histogram_ave_start + ((real(i_histogram_bin &
           - lbound(histogram_ave, 1),kind=dp)+0.5_dp) &
           * histogram_ave_bin_width))
    end do
    ! can happen if histogram is empty, then we don't want strange
    ! 0/0 conditions
    if(normalization .eq. 0.0_dp) then
      write(unit=*, fmt='(a)') "WARNING (SR p_dist_calc_and_write): &
           &normalization = 0.0, set to 1.0"
      normalization = 1.0_dp
    end if

    ! normalize
    average_ave = average_ave/normalization
    ! compute variance
    variance = 0.0_dp
    do i_histogram_bin = lbound(histogram_ave, 1), ubound(histogram_ave, 1)
      variance = variance &
           + histogram_ave(i_histogram_bin) &
           * (histogram_ave_start + ((real(i_histogram_bin &
           - lbound(histogram_ave, 1), kind=dp) + 0.5_dp) &
           * histogram_ave_bin_width) - average_ave)**2
    end do
    ! normalize
    variance = variance/normalization
    third_moment_ave = 0.0_dp
    forth_moment_ave = 0.0_dp
    do i_histogram_bin = lbound(histogram_ave, 1), ubound(histogram_ave, 1)
      third_moment_ave = third_moment_ave &
           + histogram_ave(i_histogram_bin) &
           * (histogram_ave_start + ((real(i_histogram_bin &
           - lbound(histogram_ave, 1), kind=dp) + 0.5_dp) &
           * histogram_ave_bin_width) - average_ave)**3
      forth_moment_ave = forth_moment_ave &
           + histogram_ave(i_histogram_bin) &
           * (histogram_ave_start + ((real(i_histogram_bin &
           - lbound(histogram_ave, 1), kind=dp) + 0.5_dp) &
           * histogram_ave_bin_width) - average_ave)**4
    end do
    ! normalize
    third_moment_ave = third_moment_ave/normalization
    forth_moment_ave = forth_moment_ave/normalization
    second_moment_ave = sqrt(variance)
    third_moment_ave = third_moment_ave/second_moment_ave**3
    forth_moment_ave = forth_moment_ave/variance**2 - 3.0_dp
  end if !(style .eq "prob_dist")
  ! open file for appending
  open(unit=fileunit, file=out_file, status="old", &
       action="write", position="append", iostat=iostatus)
  if(iostatus .ne. 0) then
    write (unit=*, fmt=*) "WARNING: Could not append to output file >>", &
         out_file, "<<"
  else
    ! append data
    ! because F90 is line oriented, we first create a string with the line
    ! we want to write (I WANT C++ !!!)
    write(word, fmt='(es17.11)') r_time
    word = adjustl(word)
    line = word
    write(word, fmt='(es11.4)') average_ave
    word = adjustl(word)
    line = trim(adjustl(line))//" "//word
    write(word, fmt='(es11.4)') second_moment_ave
    word = adjustl(word)
    line = trim(adjustl(line))//" "//word
    write(word, fmt='(es11.4)') third_moment_ave
    word = adjustl(word)
    line = trim(adjustl(line))//" "//word
    write(word, fmt='(es11.4)') forth_moment_ave
    word = adjustl(word)
    line = trim(adjustl(line))//" "//word
    if(style .eq. "prob_dist") then
      do i_histogram_bin = lbound(histogram_ave,1), ubound(histogram_ave,1)
        ! normalize to probability distribution
        r_dummy = histogram_ave(i_histogram_bin) &
             / (normalization * histogram_ave_bin_width)
        ! if r_dummy is smaller than 0.1E-99 we replace it by 0.0 in order
        ! to get well formatted output
        if(r_dummy .le. 0.100E-99_dp) r_dummy = 0.0_dp
        write(word, fmt='(es10.3)') r_dummy
        word = adjustl(word)
        line = trim(adjustl(line))//" "//word
      end do
    end if ! writing probability distribution
    if(l_debug_p_dist_calc_and_write) then
      write(unit=*,fmt='(a, i12, a)') &
           "DEBUG (SR p_dist_calc_and_write): at i_time=", i_time, &
           " writing: "
      write(unit=*,fmt='(2a)') "      ", trim(adjustl(line))
    end if
    ! finally write to file
    write(unit=fileunit, fmt='(a)') trim(adjustl(line))
    close(fileunit)
  end if
end if ! time to write
end subroutine p_dist_calc_and_write
!-----------------------------------------------------------------------------

!-----------------------------------------------------------------------------
! calculate moments of probability distribution and write out probablity
! distribution and moments
subroutine p_dist_write_out(out_file, header, string, &
     histogram_ave, histogram_ave_start, histogram_ave_end, &
     histogram_ave_bin_width)
  character(len=*), intent(in) :: out_file, header, string
  real(kind=dp), dimension(:), intent(in) :: histogram_ave
  real(kind=dp), intent(in) :: histogram_ave_start, histogram_ave_end, &
       histogram_ave_bin_width
  ! moments
  real(kind=dp) :: normalization, variance, average_ave, second_moment_ave, &
       third_moment_ave, forth_moment_ave
  ! I/O stuff
  integer, parameter :: fileunit=61
  integer :: iostatus
  integer :: i_histogram_bin
  real(kind=dp) :: r_dummy, bin_center
  if(l_debug_p_dist_write_out) then
```

```
    print *, "DEBUG: Entering SR p_dist_write_out"
    print *, "histogram_ave_start=", histogram_ave_start
    print *, "histogram_ave_end=", histogram_ave_end
    print *, "histogram_ave_bin_width=", histogram_ave_bin_width
    print *, "Writing histogram:"
    print *, histogram_ave
  end if
  ! open file for replacement
  open(unit=fileunit, file=out_file, status="replace", &
       action="write", iostat=iostatus)
  if(iostatus .ne. 0) then
    write (unit=*, fmt=*) "WARNING: Could not open output file >>", &
         out_file, "<<"
    return
  end if
  ! write out preamble
  write(unit=*, fmt='(3a)') &
       "MESSAGE: Writing probability distribution to >>", out_file,"<<"
  write(unit=fileunit, fmt='(a)') "# "//header
  write(unit=fileunit, fmt='(a, g13.6)') &
       "# bin start = ", histogram_ave_start
  write(unit=fileunit, fmt='(a, g13.6)') &
       "# bin end = ",  histogram_ave_end
  write(unit=fileunit, fmt='(a, g13.6)') &
       "# bin width = ", histogram_ave_bin_width
  write(unit=fileunit, fmt='(a, i7)') "# number of bins:", &
       size(histogram_ave, 1)
  ! compute normalization and average
  normalization = 0.0_dp
  average_ave = 0.0_dp
  do i_histogram_bin = lbound(histogram_ave, 1), ubound(histogram_ave, 1)
    ! this normalization does not include the bin width.  For the moments
    ! we don't need that, but later for the probabilities
    normalization = normalization &
         + histogram_ave(i_histogram_bin)
    average_ave = average_ave + histogram_ave(i_histogram_bin) &
         * (histogram_ave_start + ((real(i_histogram_bin &
         - lbound(histogram_ave, 1), kind=dp) + 0.5_dp) &
         * histogram_ave_bin_width))
  end do
  ! can happen if histogram is empty, then we don't want strange
  ! 0/0 conditions
  if(normalization .eq. 0.0_dp) then
    write(unit=*, fmt='(a)') "WARNING (SR p_dist_write_out): &
         &normalization = 0.0, set to 1.0"
    normalization = 1.0_dp
  end if
  ! normalize
  average_ave = average_ave/normalization
  ! compute variance
  variance = 0.0_dp
  do i_histogram_bin = lbound(histogram_ave, 1), ubound(histogram_ave, 1)
    variance = variance &
         + histogram_ave(i_histogram_bin) &
         * (histogram_ave_start + ((real(i_histogram_bin &
         - lbound(histogram_ave, 1), kind=dp) + 0.5_dp) &
         * histogram_ave_bin_width) - average_ave)**2
  end do
  ! normalize
  variance = variance/normalization
  third_moment_ave = 0.0_dp
  forth_moment_ave = 0.0_dp
  do i_histogram_bin = lbound(histogram_ave, 1), ubound(histogram_ave, 1)
    third_moment_ave = third_moment_ave &
         + histogram_ave(i_histogram_bin) &
         * (histogram_ave_start + ((real(i_histogram_bin &
         - lbound(histogram_ave, 1), kind=dp) + 0.5_dp) &
         * histogram_ave_bin_width) - average_ave)**3
    forth_moment_ave = forth_moment_ave &
         + histogram_ave(i_histogram_bin) &
         * (histogram_ave_start + ((real(i_histogram_bin &
         - lbound(histogram_ave, 1), kind=dp) + 0.5_dp) &
         * histogram_ave_bin_width) - average_ave)**4
  end do
  ! normalize
  third_moment_ave = third_moment_ave/normalization
  forth_moment_ave = forth_moment_ave/normalization
  second_moment_ave = sqrt(variance)
  third_moment_ave = third_moment_ave/second_moment_ave**3
  forth_moment_ave = forth_moment_ave/variance**2 - 3.0_dp
  write(unit=fileunit, fmt='(a, es11.4)') "# average       ", average_ave
  write(unit=fileunit, fmt='(a, es11.4)') "# second_moment ", &
       second_moment_ave
  write(unit=fileunit, fmt='(a, es11.4)') "# third_moment  ", &
       third_moment_ave
  write(unit=fileunit, fmt='(a, es11.4)') "# forth_moment  ", &
       forth_moment_ave
  write(unit=fileunit, fmt='(a)') "# "//string
  do i_histogram_bin = lbound(histogram_ave, 1), ubound(histogram_ave, 1)
    bin_center = histogram_ave_start &
         + (real(i_histogram_bin-lbound(histogram_ave, 1), kind=dp)+0.5_dp)&
         * histogram_ave_bin_width
    ! normalize to probability distribution
    r_dummy = histogram_ave(i_histogram_bin) &
         / (normalization * histogram_ave_bin_width)
    ! if r_dummy is smaller than 0.1E-99 we replace it by 0.0 in order
    ! to get well formatted output
    if(r_dummy .le. 0.100E-99_dp) r_dummy = 0.0_dp
    write(unit=fileunit, fmt='(es11.4, es13.6)') bin_center, r_dummy
  end do
  close(fileunit)
  if(l_debug_p_dist_write_out) &
       print *, "DEBUG: Leaving SR p_dist_write_out"
end subroutine p_dist_write_out
!------------------------------------------------------------------------
!------------------------------------------------------------------------
! calculates the first, second, third, and forth centralized moments from
! a given array
subroutine moments(in_array, first, second, third, forth)
  real(kind=dp), intent(in), dimension(:) :: in_array
  real(kind=dp), intent(out) :: first, second, third, forth
  real(kind=dp) :: number, sum_values, sum_squares, sum_cubes, sum_hypercubes
  real(kind=dp), dimension(size(in_array)) :: help_array
  ! initialize
  help_array = in_array**2
  sum_values = sum(in_array)
  sum_squares = sum(help_array)
  sum_cubes = sum(in_array * help_array)
  sum_hypercubes = sum(help_array * help_array)
  number = real(size(in_array), kind=dp)
  first = sum_values / number
  second = sqrt(sum_squares / number - first**2)
  third = (sum_cubes - 3.0_dp * sum_squares * first) / number &
       + 2.0_dp * first**3
  third = third / second**3
  forth = (sum_hypercubes - 4.0_dp * sum_cubes * first &
       + 6.0_dp * sum_squares * first**2) / number - 3.0_dp * first**4
  forth = forth / second**4 - 3.0_dp
end subroutine moments
!------------------------------------------------------------------------
!------------------------------------------------------------------------
function angul_vel(i_part, j_part)
  integer, intent(in) :: i_part, j_part
  real(kind=dp) :: angul_vel
  real(kind=dp), dimension(2) :: vec_cm_to_j, vel_j_in_cm
  ! a prefactor 0.5 in the two vectors cancels out
  vec_cm_to_j(:) = r0(j_part, 1:2) - r0(i_part, 1:2)
  vel_j_in_cm(:) = (rx(j_part, 1:2, 1) - rx(i_part, 1:2, 1)) * dt_inv
  angul_vel = vec_cm_to_j(1)*vel_j_in_cm(2) - vec_cm_to_j(2)*vel_j_in_cm(1)
  angul_vel = angul_vel / dot_product(vec_cm_to_j, vec_cm_to_j)
  return
end function angul_vel
!------------------------------------------------------------------------
end module popping
```

polymerV1.9.f90

```
! polymerVx.y.f90
! module for fluid and polymer analysis
! Gyration tensor
! Pressure tensor
! Martin Aichele, 2002-06-25
! last changed 2002-07-22
module polymer
  use globals
  implicit none
  !-----------------------------------------------------------------------
  ! debug switches      |-------- 31 characters ------|
  !-----------------------------------------------------------------------
  logical, parameter :: l_debug_calc_gyration_tensor    = .FALSE.
  logical, parameter :: l_debug_calc_end_to_end_dist    = .FALSE.
  logical, parameter :: l_debug_calc_mean_mon_end_dist  = .FALSE.
  !-----------------------------------------------------------------------
  ! output units
  integer, parameter :: gyr_ten_out_unit                = 41
  integer, parameter :: press_ten_out_unit              = 42
  !-----------------------------------------------------------------------
contains
  !-----------------------------------------------------------------------
  ! gyration tensor, end-to-end distance squared and mean monomer-end
  ! distance squared
  subroutine gyration_tensor(time_constant)
    integer, intent(in) :: time_constant
    ! time in MDS when this routine was first called
    integer, save :: first_call_time = -huge(1)
    integer :: i_dim, j_dim
    real(kind=dp), dimension(n_dim, n_dim), save :: gyr_ten_ave = 0.0_dp
    real(kind=dp), save :: re_sq_ave = 0.0_dp
    real(kind=dp), save :: rm_sq_ave = 0.0_dp
    real(kind=dp), dimension(n_dim, n_dim) :: gyr_ten
    real(kind=dp) :: re_sq, rm_sq
    ! for a simple fluid we get 0
    if(n_mon .eq. 1) then
      write(unit=*, fmt='(a)') "ERROR (SR gyration_tensor): n_mon .eq. 1"
      write(unit=*, fmt='(a)') "      For a simple fluid all entries are 0.0"
      stop
    end if
    ! get values for this MD step
    call calc_gyration_tensor(gyr_ten)
    re_sq = calc_end_to_end_dist()
    rm_sq = calc_mean_mon_end_dist()
    ! initialize or average
    if(first_call_time .eq. -huge(1)) then
      first_call_time = i_time
      gyr_ten_ave(:, :) = gyr_ten(:, :)
      re_sq_ave = re_sq
      rm_sq_ave = rm_sq
    else ! average
      if(time_constant .gt. 1) then
        gyr_ten_ave(:, :) = &
             (1.0_dp-1.0_dp/real(time_constant,kind=dp)) * gyr_ten_ave(:, :)&
             + (1.0_dp/real(time_constant,kind=dp)) * gyr_ten(:, :)
        re_sq_ave = &
             (1.0_dp-1.0_dp/real(time_constant,kind=dp)) * re_sq_ave &
             + (1.0_dp/real(time_constant,kind=dp)) * re_sq
        rm_sq_ave = &
             (1.0_dp-1.0_dp/real(time_constant,kind=dp)) * rm_sq_ave &
             + (1.0_dp/real(time_constant,kind=dp)) * rm_sq
      else
        gyr_ten_ave(:, :) = gyr_ten(:, :)
        re_sq_ave = re_sq
        rm_sq_ave = rm_sq
      end if
    end if ! first call
    ! when the time has come to write out...
    if(mod(i_time, time_constant) .eq. 0) then
      write(unit=gyr_ten_out_unit, advance="no", fmt='(es17.11)') r_time
      do i_dim=1, n_dim
        do j_dim=i_dim, n_dim
          write (unit=gyr_ten_out_unit, advance="no", fmt='(g13.5)') &
               gyr_ten_ave(i_dim, j_dim) / real(n_mon_tot, kind=dp)
        end do
      end do
      write (unit=gyr_ten_out_unit, advance="no", fmt='(g13.5)') &
           re_sq_ave / real(n_chain, kind=dp)
      write (unit=gyr_ten_out_unit, advance="yes", fmt='(g13.5)') &
           rm_sq_ave / real(2*n_mon_tot, kind=dp)
    end if
  end subroutine gyration_tensor
  !-----------------------------------------------------------------------
  !-----------------------------------------------------------------------
  ! calculates the gyration tensor
  ! normalization is done elsewhere
  subroutine calc_gyration_tensor(gyr_ten)
    real(kind=dp), dimension(n_dim, n_dim), intent(out) :: gyr_ten
    integer :: index_a, index_b, i_part, i_chain, i_dim, j_dim
    real(kind=dp), dimension(n_dim) :: Rcm_chain
```

```
    if(l_debug_calc_gyration_tensor) &
         print *, "DEBUG: Entering SR calc_gyration_tensor"
    ! initialize
    gyr_ten(:, :) = 0.0_dp
    ! loop over all chains
    do i_chain = 1, n_chain
      ! get center of mass of chain
      Rcm_chain(:) = 0.0_dp
      do i_part = (i_chain -1) * n_mon +1, i_chain * n_mon
        Rcm_chain(:) =  Rcm_chain(:) + r0_unfolded(i_part, :)
      end do
      Rcm_chain(:) =  Rcm_chain(:)/real(n_mon, kind=dp)
      do index_a = 1, n_dim
        ! use symmetry
        do index_b = index_a, n_dim
          do i_part = (i_chain -1) * n_mon +1, i_chain * n_mon
            gyr_ten(index_a, index_b) = gyr_ten(index_a, index_b) &
                 + (r0_unfolded(i_part, index_a) - Rcm_chain(index_a)) &
                 * (r0_unfolded(i_part, index_b) - Rcm_chain(index_b))
          end do
        end do
      end do
!!$      print *, "half bond:", &
!!$           2.0_dp*sqrt((r0_unfolded(i_part, 1) - Rcm_chain(1))**2 &
!!$           + (r0_unfolded(i_part, 2) - Rcm_chain(2))**2 &
!!$           + (r0_unfolded(i_part, 3) - Rcm_chain(3))**2)
    end do ! end loop over all chains
    if(l_debug_calc_gyration_tensor) then
      do i_dim=1, n_dim
        do j_dim=i_dim, n_dim
          write (unit=*, advance="no", fmt='(g13.5)') &
               gyr_ten(i_dim, j_dim) / real(n_mon_tot, kind=dp)
        end do
      end do
      write (unit=*, fmt=*) ""
    end if
    if(l_debug_calc_gyration_tensor) &
         print *, "DEBUG: Leaving SR calc_gyration_tensor"
  end subroutine calc_gyration_tensor
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! calculates the squared end-to-end distance
  ! normalization is done elsewhere
  real(kind=dp) function calc_end_to_end_dist()
    integer :: i_chain
    if(l_debug_calc_end_to_end_dist) &
         print *, "DEBUG: Entering FCT calc_end_to_end_dist"
    ! initialize
    calc_end_to_end_dist = 0.0_dp
    do i_chain = 1, n_chain
      calc_end_to_end_dist = calc_end_to_end_dist + dot_product( &
           r0_unfolded((i_chain - 1) * n_mon + 1, :) &
           - r0_unfolded(i_chain * n_mon, :), &
           r0_unfolded((i_chain - 1) * n_mon + 1, :) &
           - r0_unfolded(i_chain * n_mon, :))
    end do
    if(l_debug_calc_end_to_end_dist) then
      write(unit=*, fmt='(g11.3)') &
           calc_end_to_end_dist / real(n_chain, kind=dp)
    end if
    if(l_debug_calc_end_to_end_dist) &
         print *, "DEBUG: Leaving FCT calc_end_to_end_dist"
  end function calc_end_to_end_dist
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! calculates the mean squared distance of a monomer from the endpoints
  ! normalization is done elsewhere
  real(kind=dp) function calc_mean_mon_end_dist()
    integer :: i_chain, i_mon, i_part
    if(l_debug_calc_mean_mon_end_dist) &
         print *, "DEBUG: Entering FCT calc_mean_mon_end_dist"
    ! initialize
    calc_mean_mon_end_dist = 0.0_dp
    do i_chain = 1, n_chain
      do i_mon = 1, n_mon
        i_part = (i_chain - 1) * n_mon + i_mon
        calc_mean_mon_end_dist = calc_mean_mon_end_dist &
             + dot_product( &
             r0_unfolded(i_part, :) &
             - r0_unfolded((i_chain -1) * n_mon +1, :), &
             r0_unfolded(i_part, :) &
             - r0_unfolded((i_chain -1) * n_mon +1, :)) &
             + dot_product( &
             r0_unfolded(i_part, :) &
             - r0_unfolded(i_chain * n_mon, :), &
             r0_unfolded(i_part, :) &
             - r0_unfolded(i_chain * n_mon, :))
      end do
    end do
    if(l_debug_calc_end_to_end_dist) then
      write(unit=*, fmt='(g11.3)') &
           calc_mean_mon_end_dist / real(2*n_mon*n_chain, kind=dp)
    end if
    if(l_debug_calc_end_to_end_dist) &
         print *, "DEBUG: Leaving FCT calc_mean_mon_end_dist"
  end function calc_mean_mon_end_dist
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! pressure tensor
  subroutine pressure_tensor(time_constant)
    integer, intent(in) :: time_constant
    ! time in MDS when this routine was first called
    integer, save :: first_call_time = -huge(1)
    real(kind=dp), dimension(n_dim, n_dim), save :: press_ten_ave = 0.0_dp
    integer :: i_dim, j_dim
    real(kind=dp), dimension(n_dim, n_dim) :: press_ten
    real(kind=dp) :: normalization
    ! upper and lower bound of the particles in z-direction at this time
    real(kind=dp) :: twall_z_t, bwall_z_t
    ! get value for this MD step
    call calc_pressure_tensor(press_ten)
    ! define normalization as volume occupied by particles
    if(n_dim .eq. n_dim_pbc+1) then
      if(f_wall_fix .ne. 1) then
        twall_z_t = sum(r0(n_mon_tot+1:n_mon_tot+n_top_wall, n_dim)) &
             / real(n_top_wall, kind=dp)
        bwall_z_t = sum(r0(n_mon_tot+n_top_wall+1:n_part, n_dim)) &
             / real(n_bottom_wall, kind=dp)
      else
        twall_z_t = r0_twall(n_dim)
        bwall_z_t = r0_bwall(n_dim)
      end if
      press_ten(:, :) =  press_ten(:, :) / (twall_z_t - bwall_z_t)
    end if
    ! initialize or average
    if(first_call_time .eq. -huge(1)) then
      first_call_time = i_time
      press_ten_ave(:, :) = press_ten(:, :)
    else ! average
      if(time_constant .gt. 1) then
        press_ten_ave(:, :) = &
             (1.0_dp-1.0_dp/real(time_constant,kind=dp)) * press_ten_ave(:, :)&
             + (1.0_dp/real(time_constant,kind=dp)) * press_ten(:, :)
      else
        press_ten_ave(:, :) = press_ten(:, :)
      end if
    end if ! first call
    ! reset potential contribution.  We reset here, because we need
    ! press_tens_pot(:, :) if n_relax .gt. 0
    press_tens_pot(:, :) = 0.0_dp
    press_tens_pot_walls(:, :) = 0.0_dp
    ! when the time has come to write out...
    if(mod(i_time, time_constant) .eq. 0) then
      ! we have already divided by the extension in "z"
      normalization = pbc_volume
      write(unit=press_ten_out_unit, advance="no", fmt='(es17.11)') r_time
      do i_dim=1, n_dim
        do j_dim=i_dim, n_dim
          write (unit=press_ten_out_unit, advance="no", fmt='(g12.4)') &
               press_ten_ave(i_dim, j_dim) / normalization
        end do
      end do
      write(unit=press_ten_out_unit, fmt=*) ""
    end if
  end subroutine pressure_tensor
  !---------------------------------------------------------------------------

  !---------------------------------------------------------------------------
  ! calculate the pressure tensor.  It is advantageous to do this in the
  ! simulation, as all forces are known.
  subroutine calc_pressure_tensor(press_ten)
    real(kind=dp), dimension(n_dim, n_dim), intent(out) :: press_ten
    real(kind=dp), dimension(n_dim, n_dim) :: press_tens_kin
    real(kind=dp), dimension(n_dim) :: stream_vel
    integer :: index_a, index_b, i_part
    if(f_thermostat_mode .eq. 4 .and. (n_dim .ne. n_dim_pbc)) then
      write(unit=*, fmt='(a)') "ERROR (SR calc_pressure_tensor): &
           &f_thermostat_mode .eq. 4 could create problems with moving walls"
      stop
    end if
    ! reset kinetic contribution
    press_tens_kin(:, :) = 0.0_dp
    ! define the streaming velocity of the fluid
    ! (unit here is velocity * dt)
    ! 0.5_dp * rx_twall(:, 1) should be very close to
    ! sum(rx(1:n_mon_tot, :, 1)) / real(n_mon_tot, kind=dp)
    ! (assuming that the fluid moves on average with half the top wall
    ! velocity, which must be true as long time average for symmetry between
    ! the walls)
    stream_vel(:) = 0.5_dp * rx_twall(:, 1)
    ! take kinetic contribution
    if(f_wall_fix .eq. 1) then
      do index_a = 1, n_dim
        do index_b = index_a, n_dim
          press_tens_kin(index_a, index_b) = sum(mass(type(1:n_mon_tot)) &
               * (rx(1:n_mon_tot, index_a, 1) - stream_vel(index_a)) &
               * (rx(1:n_mon_tot, index_b, 1) - stream_vel(index_b)))
          press_tens_kin(index_a, index_b) = press_tens_kin(index_a, index_b) &
               + mass_twall * (rx_twall(index_a, 1)- stream_vel(index_a)) &
               * (rx_twall(index_b, 1)- stream_vel(index_b))
          ! bottom wall does not move
          press_tens_kin(index_a, index_b) = press_tens_kin(index_a, index_b) &
                + mass_bwall * stream_vel(index_a) * stream_vel(index_b)
        end do
      end do
    else
      do index_a = 1, n_dim
        do index_b = index_a, n_dim
          press_tens_kin(index_a, index_b) = sum(mass(type(1:n_part)) &
               * (rx(1:n_part, index_a, 1) - stream_vel(index_a)) &
               * (rx(1:n_part, index_b, 1) - stream_vel(index_b)))
        end do
      end do
    end if
    ! rx(:, :, 1) stores velocity / dt
    press_tens_kin(:, :) =  press_tens_kin(:, :) / dt_2
    ! add contributions
    ! the - sign comes from the sign convention chosen in the interaction
    ! routines.
    !                                        |
    !                                       \|/
    !                                        V
    press_ten(:, :) = press_tens_kin(:, :) - press_tens_pot(:, :)
    if(l_compute_press_tens_wall_contr) then
      press_ten(:, :) = press_ten(:, :) - press_tens_pot_walls(:, :)
    end if
!!$    print *, "Pressure tensor at mds", i_time, ":"
!!$    print *,  press_tens_kin(:, :)
!!$    print *, -press_tens_pot(:, :)
  end subroutine calc_pressure_tensor
  !---------------------------------------------------------------------------
end module polymer
```

Appendix D

Analysis Code

In this appendix we list the code for the analysis routines for the calculation of the static structure factors of our polymer melt. The Makefile explains the dependencies of the different program modules.

Makefile

```
# Makefile for various analysis routines
# Martin Aichele, 2002-04-16
# last changed 2003-01-29
# what compiler we use
CC=gcc
#CC=icc
CPP=g++
#CPP=icc
# debugging options
#DEBUGOPS=-g -O0
# warning options
WARNINGS=-W -Wall -Wpointer-arith -Wshadow
#WARNINGS=-w1
# machine target options
# specifying '-march=CPU TYPE' implies '-mcpu=CPU TYPE'
MACHOPS=-march=pentiumpro
# gcc
################################
# optimization options (try first)
# -ffast-math
# -fno-exceptions
# -funroll-loops
# -finline-functions
# -frerun-loop-opt
OPTOPS=
# gcc
# use special optimizations for the static structure factors
# -O3 does not help
#OPTFSSF=-O2 -fomit-frame-pointer -malign-double
OPTFSSF=-O2 -fomit-frame-pointer -malign-double -frerun-loop-opt -funroll-loops
# icc
#OPTFSSF=-O3 -xK -ip -prefetch
# gcc
# special optimizations for qveclib
OPTQVEC=-O2 -malign-double -march=i686 -funroll-loops -fmove-all-movables \
-fomit-frame-pointer
# standard optimization
STDOPT=-O1
# library options
LIBOPS=-lm
# LAPACK libraries (archives)
LPDIR=/usr/local/lib/CLAPACK/
TMGLIB=$(LPDIR)tmglib_LINUX.a
LAPACKLIB=$(LPDIR)lapack_LINUX.a
BLASLIB=$(LPDIR)blas_LINUX.a
F77LIB=$(LPDIR)F2CLIBS/libF77.a
I77LIB=$(LPDIR)F2CLIBS/libI77.a
LPLIBS=$(TMGLIB) $(LAPACKLIB) $(BLASLIB) $(F77LIB) $(I77LIB)
STDOPS=$(WARNINGS) $(MACHOPS) $(OPTOPS) $(DEBUGOPS)
# Version numbering
VER_GEN_LIB=V1.1
VER_QVEC_LIB=V1.7
VER_SF_LIB=V0.5
VER_STAT_LIB=V0.5
# displacements library
VER_DISPL_LIB=V1.1
# mean square displacements
VER_ALLG=V1.3
# dynamical scattering functions
VER_DYNSF=V1.1
# R_e and R_g
VER_RERG=V1.0
# cosine of the bond angle
VER_CBA=V1.0
# (Static) Rouse mode correlation matrix
VER_RMCM=V1.2
# Rouse mode autocorrelation function
VER_RMACF=V1.2
# structure factor library
VER_STRUFA_LIB=V1.2
# fast static structure factor
VER_FSSF=V1.9
# three particle static structure factor (S_3)
VER_S3=V1.6
# direct correlation function
VER_DCF=V1.0
.PHONY : clean
all : allg dynsf rerg cba rmcm rmacf fssf s3 dcf
# libraries
yasptrj.o : yasptrj.h yasptrj.c
$(CC) $(STDOPS) $(STDOPT) -c -o yasptrj.o yasptrj.c
general.o : yasptrj.o \
general_lib$(VER_GEN_LIB).h general_lib$(VER_GEN_LIB).c
$(CC) $(STDOPS) $(STDOPT) -c -o general.o \
general_lib$(VER_GEN_LIB).c
displacement_lib.o : displacement_lib_$(VER_DISPL_LIB).h \
   displacement_lib_$(VER_DISPL_LIB).c
$(CC) $(STDOPS) $(STDOPT) -c -o displacement_lib.o\
  displacement_lib_$(VER_DISPL_LIB).c
random.o   : mt19937-1.h mt19937-1_ma.c
$(CC) $(STDOPS) $(STDOPT) -c -o random.o mt19937-1_ma.c
qveclib.o : general_lib$(VER_GEN_LIB).h \
qvec_lib$(VER_QVEC_LIB).h \
mt19937-1.h qvec_lib$(VER_QVEC_LIB).cpp
$(CPP) $(STDOPS) $(QVECOPT) -c -o qveclib.o \
qvec_lib$(VER_QVEC_LIB).cpp
sflib.o : general_lib$(VER_GEN_LIB).h \
sf_lib$(VER_SF_LIB).h sf_lib$(VER_SF_LIB).c
$(CC) $(STDOPS) $(STDOPT) -c -o sflib.o \
sf_lib$(VER_SF_LIB).c
statlib.o : general_lib$(VER_GEN_LIB).h \
static_lib$(VER_STAT_LIB).h static_lib$(VER_STAT_LIB).c
$(CC) $(STDOPS) $(STDOPT) -c -o statlib.o \
static_lib$(VER_STAT_LIB).c
# displacement functions g_0 ... g_5
allg.o : displacement_lib_$(VER_DISPL_LIB).h g0-g5$(VER_ALLG).c
$(CC) $(STDOPS) $(STDOPT) -c -o allg.o g0-g5$(VER_ALLG).c
allg : displacement_lib.o allg.o
$(CC) $(STDOPS) $(LIBOPS) $(STDOPT) -o allg$(VER_ALLG).exe\
displacement_lib.o allg.o
# dynamical scattering functions
dynsf.o : general.o \
qvec_lib$(VER_QVEC_LIB).h qvec_lib$(VER_QVEC_LIB).cpp\
sf_lib$(VER_SF_LIB).h scat_fctns$(VER_DYNSF).cpp
$(CPP) $(STDOPS) $(STDOPT) -c -o dynsf.o \
scat_fctns$(VER_DYNSF).cpp
dynsf : general.o random.o qveclib.o sflib.o yasptrj.o dynsf.o
$(CPP) $(STDOPS) $(LIBOPS) $(STDOPT) -o dynsf$(VER_DYNSF).exe \
general.o random.o qveclib.o sflib.o yasptrj.o dynsf.o
# radius of gyration and end-to end distance
rerg.o : general_lib$(VER_GEN_LIB).h static_lib$(VER_STAT_LIB).h \
radius_gyration_end2end$(VER_RERG).c
$(CC) $(STDOPS) $(STDOPT) -c -o rerg.o \
radius_gyration_end2end$(VER_RERG).c
rerg : general.o yasptrj.o statlib.o rerg.o
$(CC) $(STDOPS) $(LIBOPS) $(STDOPT) -o rerg$(VER_RERG).exe \
general.o yasptrj.o statlib.o rerg.o
# cosine of bond angle
cba.o : general.o static_lib$(VER_STAT_LIB).h \
cos_bond_angle$(VER_CBA).c
$(CC) $(STDOPS) $(STDOPT) -c -o cba.o \
cos_bond_angle$(VER_CBA).c
cba : general.o yasptrj.o statlib.o cba.o
$(CC) $(STDOPS) $(LIBOPS) $(STDOPT) -o cba$(VER_CBA).exe \
general.o yasptrj.o statlib.o cba.o
# static Rouse mode correlation matrix
rmcm.o : general.o static_lib$(VER_STAT_LIB).h \
rouse_mode_correl_matrix$(VER_RMCM).c
$(CC) $(STDOPS) $(STDOPT) -c -o rmcm.o \
rouse_mode_correl_matrix$(VER_RMCM).c
rmcm : general.o statlib.o yasptrj.o rmcm.o
$(CC) $(STDOPS) $(LIBOPS) $(STDOPT) -o rmcm$(VER_RMCM).exe \
general.o yasptrj.o statlib.o rmcm.o
# Rouse mode autocorrelation functions
rmacf.o : general.o rouse_mode_autocorr$(VER_RMACF).c
$(CC) $(STDOPS) $(STDOPT) -c -o rmacf.o \
rouse_mode_autocorr$(VER_RMACF).c
rmacf : general.o statlib.o yasptrj.o rmacf.o
$(CC) $(STDOPS) $(LIBOPS) $(STDOPT) -o rmacf$(VER_RMACF).exe \
general.o statlib.o yasptrj.o rmacf.o
# structure factors
strufalib.o : general_lib$(VER_GEN_LIB).h \
structure_factor_lib$(VER_STRUFA_LIB).h \
structure_factor_lib$(VER_STRUFA_LIB).c
$(CC) $(STDOPS) $(OPTFSSF) -c -o strufalib.o \
structure_factor_lib$(VER_STRUFA_LIB).c
fssf.o : general_lib$(VER_GEN_LIB).h \
 structure_factor_lib$(VER_STRUFA_LIB).h \
fssf$(VER_FSSF).c
$(CC) $(STDOPS) $(STDOPT) -c -o fssf.o \
fssf$(VER_FSSF).c
s3.o : general_lib$(VER_GEN_LIB).h qvec_lib$(VER_QVEC_LIB).h \
s3$(VER_S3).cpp
$(CPP) $(STDOPS) -c s3$(VER_S3).cpp -o s3.o
fssf : general.o strufalib.o yasptrj.o fssf.o
$(CC) $(STDOPS) $(LIBOPS) $(STDOPT) -o fssf$(VER_FSSF).exe \
general.o strufalib.o yasptrj.o fssf.o
s3 : general.o yasptrj.o random.o qveclib.o s3.o
```

```
$(CPP) $(STDOPS) $(LIBOPS) general.o yasptrj.o random.o \
qveclib.o s3.o -o s3$(VER_S3).exe
# direct correlation functions (note: order of the libraries is important)
dcf : general.o dir_corr_fct$(VER_DCF).c
$(CC) $(MACHOPS) $(STDOPT) $(DEBUGOPS) -W -Wall \
-Wpointer-arith dir_corr_fct$(VER_DCF).c $(LPLIBS) yasptrj.o general.o -lm \
-o dcf$(VER_DCF).exe
clean :
rm -f *.o core
```

D.1 Code for the Static Structure Factors

Main fssfV1.9.c

```
/* fssfVx.y.c
 * "Fast" Static Structure Factor calculation:
 *-----------------------------------------
 * Static structure factors of monomers on different and the same chain,
 * structure factor of a chain, structure factor of the centers of masses, and
 * structure factor of a monomer relative to the center of mass of a chain.
 * "Fast" because calculation of O(q^3) sines and cosines is avoided and
 * replaced by simple arithmetic operations.
 * Martin Aichele, 2000-05-16
 * last modified: 2002-10-31
 * V1.4 for NPT-ensemble and YASP trajectories.
 * Martin Aichele, 2002-11-01
 * last modified: 2002-11-06
 * V1.6 more memory efficient, correct bin-centers
 * Martin Aichele, 2002-11-13
 * last modified: 2002-11-15
 * V1.7 some refinements
 * V1.8 more general (reads Sidi's configurations)
 * Martin Aichele, 2002-12-11
 * V1.9 adaption for pure 2-d films
 * Martin Aichele, 2003-03-28
 */
#include "general_libV1.1.h"
#include "structure_factor_libV1.2.h"
/* identify program */
#ifndef CALC_ONLY_COHERENT_SF_YES
  #define WHAT "fssfV1.9"
#else
  #define WHAT "coh_sf_fssfV1.9"
#endif
int
main(int argc, char **argv){
  FILE *parameterfile_p;
  /* #chains, #monomers per chain, total # of monomers */
  unsigned int nr_chains, mon_per_chain, nr_monomers;
  /* timestep of the simulation */
  double timestep;
  /* monomers per chain, used as a step to generalization */
  unsigned int *chainlengths = NULL;
  /* parameter reading checking variables */
  unsigned int read_success, read_attempt;
  /* loop variables */
  unsigned int iLoop, jLoop, kLoop, lLoop, mLoop;
  /* how many simulation and sub simulation runs */
  unsigned int  nr_sim_runs=0, nr_timeseries, *nr_sub_sim_runs = NULL;
  /* how many directory levels */
  unsigned int nr_dir_levels = 1;
  /* path to data (make sure it is not too long) */
  char data_path[620], data_path_temp[623];
  /* ok, I know that this is dangerous...
   * so don't use this code for flight control over heavily populated areas ;)
   */
  unsigned int  dummy_length = 2000;
  /* dummy string */
  char *dummy;
  /* a string for messages */
  char string[400];
  /* filename of the sample list or matrix */
  char file_sample_times[160];
  /* string for determining if we use a sample matrix or list */
  char saving_scheme[80];
  /* string for the prefix of the configuration files */
  char pos_file_prefix[80];
  /* string for the postfix of the configuration files */
  char pos_file_postfix[80];
  /* information about the saved times is stored here */
  unsigned long int **sample_matrix = NULL;
  /* number of samples times and number of timescales woven together */
  unsigned int nr_samples_per_startpoint, nr_startpoints;
  /* number of configurations */
  unsigned int nr_configurations, nr_conf_read = 0;
  /* number of wall atoms */
  unsigned int nr_wall_atoms;
  /* wall configurations */
  r3vector * wall_conf_t = NULL;
  /* positions of top and bottom wall at a time */
  r3vector twall_pos_t, bwall_pos_t;
  /* fluid configuration */
  r3vector * conf_t = NULL;
  /* center of masses of the individual chains */
  r3vector * Rcm_chains_t = NULL;
  /* YASP frames ********************************************************/
  YaspFrame frame_t;
  /* when all boundaries in the parameter file are 0 then we have a fluctuating
   * volume and the system size is read for every configuration
   */
  boolean l_const_volume = TRUE;
  /**********************************************************************/
  /* return value of configuration reading routines */
  int conf_read_status = -1;
  /* length of the string which describes the path to the data */
  int stringlength;
  /* for printing a header in the output file */
  unsigned int nrDataDescriptionLines;
  char ** dataDescription = NULL;
  unsigned int line = 0;
  /* if coordinates should be transformed into the center of mass systems of
   * the two walls in the film simulations */
  boolean l_transform_to_cm_walls_system = FALSE;
  /* periodic boundaries */
  double boundary[DIM];
  int max_qindex[DIM];
  int total_max_qindex = 0;
  /* number of bins, number of bins in each direction for writing out the
   * q-plane */
  unsigned int nrBins = 0, nrBins_xy = 0;
  double binWidth = -1.0, binWidth_xy = -1.0;
  double maxQvalue = 0.0;
  /* if for 2-d systems the whole q-plane should be written */
  boolean l_write_xy_plane = FALSE;
  unsigned int * nr_qvecs_in_bins = NULL;
  unsigned int * nr_confs_in_bins = NULL;
  double * qvalues_in_bins = NULL;
  unsigned int nrResultBins_xy = -1;
  unsigned int * nr_qvecs_in_xy_bins = NULL;
  unsigned int * nr_confs_in_xy_bins = NULL;
  double ** qvalues_in_xy_bins = NULL;
  /* lattice units */
  double *latticeUnits;
  /* factor for spreading the lattice units by an integer n, i.e. we take only
   * every n-th point in each direction, that makes the program n^dim faster
   * (but resolution goes down, too).
   * This feature is useful for large system dimensions, where the lattice
   * units are very small.
   */
  unsigned int spreadLatticeFactor = 1;
  /* sines and cosines for the monomers */
  double *** sines = NULL;
  double *** cosines = NULL;
  /* sines and cosines for the chain centers of mass */
  double *** sines_cm = NULL;
  double *** cosines_cm = NULL;
  /* memory for the results:
   * _self denotes the single chain contribution, \sum_{i=1}^{nr_chains}
   * _all denotes all chain contribution:
   * \sum_{i=1}^{nr_chains}\sum_{j=1}^{nr_chains}, so it is the sum of the
   * self and the purely distinct contributions.  For efficiency reasons, we
   * compute \sum_{i=1}^{nr_chains}\sum_{j=1}^{nr_chains} instead of
   * \sum_{i=1}^{nr_chains}\sum_{j=1, j \ne i}^{nr_chains}.
   * (Because the double sum of products can be written as product of simple
   * sums, so we gain O(nr_chains)).
   */
  STRUF_RES_TYPE **results = NULL, **results_cm = NULL;
  STRUF_RES_TYPE **results_xy = NULL, **results_cm_xy = NULL;
  int nr_result_columns = -1, nr_result_cm_columns = -1,
     nr_result_xy_columns = -1, nr_result_cm_xy_columns = -1;
  int nr_result_lines = -1, nr_result_cm_lines = -1,
     nr_result_xy_lines = -1, nr_result_cm_xy_lines = -1;
  int sf_data_start_index = -1, sf_data_cm_start_index = -1, \
    sf_data_xy_start_index = -1, sf_data_cm_xy_start_index = -1;
  /* helpers */
  int nr_mon_pairs = -1;
#ifndef CALC_ONLY_COHERENT_SF_YES
  int column;
  unsigned int monIndex1, monIndex2;
  double coh_self_sf, coh_all_sf;
#endif
  /************************************************************************
   *        D O N E    D E C L A R I N G   V A R I A B L E S              *
   ************************************************************************/
  /* say hello */
  printf("MESSAGE: this is %s\n", argv[0]);
  /* check dimension */
#ifdef DIM
  if(DIM == 2 || DIM == 3){
    printf("MESSAGE: System is %u-dimensional.\n", DIM);
  }else{
    fprintf(stderr, "ERROR: DIM not 2 or 3\n");
    exit(1);
  }
#else
  fprintf(stderr, "ERROR: DIM not defined\n");
  exit(1);
#endif
#ifdef THREE_D
  if(DIM != 3){
    fprintf(stderr, "ERROR: THREE_D defined, but DIM != 3\n");
    exit(1);
  }
#ifdef TWO_D
    fprintf(stderr, "ERROR: THREE_D and TWO_D defined\n");
    exit(1);
#endif
#endif
#ifdef TWO_D
  if(DIM != 2){
    fprintf(stderr, "ERROR: TWO_D defined, but DIM != 2\n");
    exit(1);
  }
#ifdef THREE_D
    fprintf(stderr, "ERROR: THREE_D and TWO_D defined\n");
    exit(1);
#endif
#endif
  /* Read parameter file */
  if(TRUE){
    if (argc < 2 || argc > 3) {
      fprintf(stderr,"ERROR: Usage is %s parameter_file [output_decriptor]\n",
      argv[0]);
      exit(1);
    }else if(argc == 3){
      printf("MESSAGE: Output files will have postfix '%s'\n", argv[2]);
    }
  }else{
    printf("WARNING: Reading fixed parameter file\n");
    argv[1] = "params_statics_yasp";
  }
  if ((parameterfile_p = fopen(argv[1],"r")) == NULL){
    sprintf(string, "Couldn't open %s", argv[1]);
    error(string, HERE);
  }
  /* with read_success we keep track of the number of successfully
     read variables */
  read_success=0;
  read_attempt=0;
  /* timestep of simulation */
  ++read_attempt;
  read_success += fscanf(parameterfile_p,"timestep=%lf\n", &timestep);
  /*  number of chains in the melt*/
  ++read_attempt;
```

```
  read_success += fscanf(parameterfile_p,"nr_chains=%u\n",&nr_chains);
  /* monomers per chain */
  ++read_attempt;
  read_success += fscanf(parameterfile_p,"mon_per_chain=%u\n",&mon_per_chain);
  /* number of wall atoms */
  ++read_attempt;
  read_success += fscanf(parameterfile_p,"nr_wall_atoms=%u\n",&nr_wall_atoms);
  /* how many directory levels */
  ++read_attempt;
  read_success += fscanf(parameterfile_p,"nr_dir_levels=%u\n", &nr_dir_levels);
  /* how many different simulation runs */
  ++read_attempt;
  read_success += fscanf(parameterfile_p,"nr_sim_runs=%u\n", &nr_sim_runs);
  if(read_success != read_attempt){
    sprintf(string,"Read only %u variables, but expected %u\n", read_success,
    read_attempt);
    error(string, HERE);
  }
  /* this is a check needed because of the way the configurations are
   * accessed
   */
  if(nr_sim_runs > 99){
    fprintf(stderr, "ERROR: This program can handle only up to 99 different \
simulation runs");
    exit(2);
  }
  /* number of sub simulation runs */
  nr_sub_sim_runs = (unsigned int *)calloc(nr_sim_runs,sizeof(unsigned int));
  if(nr_dir_levels == 2){
    for(iLoop=0; iLoop < nr_sim_runs; ++iLoop){
      ++read_attempt;
      read_success += fscanf(parameterfile_p,"%u\n", nr_sub_sim_runs+iLoop);
      if(nr_sub_sim_runs[iLoop] > 99){
fprintf(stderr, "ERROR: This program can handle only up to 99 \
different sub simulation runs");
exit(2);
      }
    }
  }else if(nr_dir_levels == 1){
    for(iLoop=0; iLoop < nr_sim_runs; ++iLoop)
      nr_sub_sim_runs[iLoop] = 1;
  }else{
    fprintf(stderr, "ERROR: nr_dir_levels=%u not valid.\n", nr_dir_levels);
    exit(2);
  }
  /* string describing the saving scheme */
  ++read_attempt;
  read_success += fscanf(parameterfile_p,"saving_scheme=%s\n", saving_scheme);
  /* string with the prefix of the configuration file */
  ++read_attempt;
  read_success += fscanf(parameterfile_p,"pos_file_prefix=%s\n",
 pos_file_prefix);
  /* string with the postfix of the configuration file */
  ++read_attempt;
  read_success += fscanf(parameterfile_p,"pos_file_postfix=%s\n",
 pos_file_postfix);
  /* we can't successfully read nothing, so we mark blank */
  if(strcmp(pos_file_postfix, "blank") == 0){
    printf("MESSAGE: pos_file_postfix defined as blank.\n");
    pos_file_postfix[0] = '\0';
  }
  /* filename of the file that contains all times of all timeseries */
  ++read_attempt;
  read_success += fscanf(parameterfile_p,"file_sample_times=%s\n",
 file_sample_times);
  /* path to the configurations to be read, data is expected in the
   * directories
   * data_path[sim_run]/
   * the shell is not involved, so path expansion like for '~' is not done.
   */
  ++read_attempt;
  read_success += fscanf(parameterfile_p,"data_path=%s\n", data_path);
  ++read_attempt;
  read_success += fscanf(parameterfile_p, \
 "l_transform_to_cm_walls_system=%d\n", \
 &l_transform_to_cm_walls_system);
  /* check if we read as many variables as expected */
  if(read_success != read_attempt){
    fprintf(stderr, "ERROR: Read only %u variables, but expected %u\n",
    read_success, read_attempt);
    exit(2);
  }
  /* periodic boundaries */
  for(iLoop = 0; iLoop < DIM; iLoop++){
    ++read_attempt;
    read_success += fscanf(parameterfile_p, "boundary=%lf\n", boundary+iLoop);
  }
  /* sanity check */
  for(iLoop = 0; iLoop < DIM; iLoop++)
    if(boundary[iLoop] < 0.0){
      fprintf(stderr, "ERROR: boundary[%u]=%f < 0.0\n", iLoop, \
      boundary[iLoop]);
      exit(1);
    }
#ifdef TWO_D
  if(boundary[0] == 0.0 && boundary[1] == 0.0){
    l_const_volume = FALSE;
  }
#endif
#ifdef THREE_D
  if(boundary[0] == 0.0 && boundary[1] == 0.0 && boundary[2] == 0.0){
    l_const_volume = FALSE;
  }
#endif
  ++read_attempt;
  read_success += fscanf(parameterfile_p, \
 "l_write_xy_plane=%d\n", &l_write_xy_plane);
  if(DIM == 3 && l_write_xy_plane == TRUE){
    fprintf(stderr, "ERROR: DIM == 3 && l_write_xy_plane == TRUE\n");
    exit(2);
  }
  ++read_attempt;
  read_success += fscanf(parameterfile_p,"spreadLatticeFactor=%u\n", \
 &spreadLatticeFactor);
  ++read_attempt;
  read_success += fscanf(parameterfile_p,"binWidth=%lf\n", &binWidth);
  ++read_attempt;
  read_success += fscanf(parameterfile_p,"nrBins=%u\n", &nrBins);
  if(l_write_xy_plane){
    ++read_attempt;
    read_success += fscanf(parameterfile_p,"binWidth_xy=%lf\n", &binWidth_xy);
    ++read_attempt;
    read_success += fscanf(parameterfile_p,"nrBins_xy=%u\n", &nrBins_xy);
  }
  /* close parameter file */
  fclose(parameterfile_p);
  if(read_success != read_attempt){
    sprintf(string,"Read only %u variables, but expected %u\n", read_success,
    read_attempt);
    error(string, HERE);
  }
  printf("MESSAGE: Parameters used:\n");
  printf("============================================================\
==========\n");
  printf("timestep=%f\n", timestep);
  printf("nr_chains=%u\nmon_per_chain=%u\n", nr_chains, mon_per_chain);
  printf("nr_wall_atoms=%u\n", nr_wall_atoms);
  printf("nr_dir_levels=%u\n", nr_dir_levels);
  printf("nr_sim_runs=%u\n", nr_sim_runs);
  if(nr_dir_levels > 1){
    for(iLoop=0; iLoop<nr_sim_runs; ++iLoop)
      printf("simulation run %u has %u sub runs\n", iLoop,
     nr_sub_sim_runs[iLoop]);
  }
  printf("saving_scheme=%s\n", saving_scheme);
  printf("pos_file_prefix=%s\n", pos_file_prefix);
  printf("pos_file_postfix=%s\n", pos_file_postfix);
  printf("file_sample_times=`%s'\n", file_sample_times);
  printf("data_path=`%s'\n", data_path);
  printf("l_transform_to_cm_walls_system=%d\n",l_transform_to_cm_walls_system);
  printf("------------------------------------------------------------\
----------\n");
  if(l_const_volume == TRUE){
    for(iLoop = 0; iLoop < DIM; iLoop++)
      printf("boundary(%u)=%f\n", iLoop, boundary[iLoop]);
  }else{
    printf("Reading boundaries anew for each configuration.\n");
  }
  printf("l_write_xy_plane=%d\n", l_write_xy_plane);
  printf("spreadLatticeFactor=%u\n", spreadLatticeFactor);
  printf("binWidth=%f\n", binWidth);
  printf("nrBins=%u (therefore 0 <= q <= %f)\n", \
 nrBins, binWidth*(double)nrBins);
  if(l_write_xy_plane){
    printf("binWidth_xy=%f\n", binWidth_xy);
    printf("nrBins_xy=%u (therefore 0 <= |q| <= %f)\n", \
   nrBins_xy, binWidth_xy*(double)nrBins_xy);
  }
  printf("============================================================\
==========\n\n");
  /*************************************************************************
   *           D O N E   R E A D I N G  P A R A M E T E R S                 *
   *************************************************************************/
  /* check if saving scheme is recognized */
  if( !(strcmp(saving_scheme, "list") == 0)
|| (strcmp(saving_scheme, "matrix") == 0)
|| (strcmp(saving_scheme, "list_yasp") == 0)
|| (strcmp(saving_scheme, "matrix_yasp") == 0)
|| (strcmp(saving_scheme, "list_2d") == 0)
|| (strcmp(saving_scheme, "matrix_2d") == 0)))
    error("ERROR: Unrecognised saving scheme", HERE);
  /* now read the times */
  /* for list and list_yasp we just have a list of times.  For dynamic
   * quantities we sometimes use a saving scheme with a matrix
   */
  if (strcmp(saving_scheme, "list")         == 0
      || strcmp(saving_scheme, "list_yasp") == 0
      || strcmp(saving_scheme, "list_2d")   == 0 ){
    if ((parameterfile_p = fopen(file_sample_times, "r")) == NULL){
      sprintf(string, "ERROR: Couldn't open %s", file_sample_times);
      error(string, HERE);
    }
    /* how many samples there are */
    if(fscanf(parameterfile_p, "NR_SAMPLES_TOTAL=%u\n",
      &nr_samples_per_startpoint) == 1)
      printf("MESSAGE: nr_samples_per_startpoint = %u\n",
     nr_samples_per_startpoint);
    else
      error("ERROR: Couldn't read nr_samples_per_startpoint in sample_list",
    HERE);
    if(nr_samples_per_startpoint == 0)
      error("nr_samples_per_startpoint == 0\n", HERE);
    /* treat the list like a matrix */
    nr_startpoints = 1;
  }else{ /* we have a sample_matrix */
    if ((parameterfile_p = fopen(file_sample_times, "r")) == NULL) {
      sprintf(string, "ERROR: Couldn't open %s \n", file_sample_times);
      error(string, HERE);
    }
    if(fscanf(parameterfile_p, "NR_of_columns=NR_STARTPOINTS=%u\n",
      &nr_startpoints)!= 1)
      error("ERROR: NR_STARTPOINTS not read correctly.\n", HERE);
    printf("\nMESSAGE: nr_startpoints=%u\n", nr_startpoints);
    if(fscanf(parameterfile_p, "NR_of_rows=NR_SAMPLES_PER_STARTPOINT=%u\n",
      &nr_samples_per_startpoint) != 1)
      error("ERROR: NR_SAMPLES_PER_STARTPOINT not read correctly.\n", HERE);
    printf("MESSAGE: nr_samples_per_startpoint=%u\n\n",
   nr_samples_per_startpoint);
  }/* end if */
  /* from now on we can hopefully treat the list like a matrix */
  sample_matrix = ulimatrix(0, nr_startpoints-1,
    0, nr_samples_per_startpoint-1);
  /* The fgets(dummy, DUMMY_LENGTH, file_p) stuff is here to allow to read
   * just the first nr_startpoints columns of a possibly bigger matrix.
   * Note that there's a difference between "%lu" and "%lu " in fscanf:
   * In the former case the file pointer points at the position after the last
   * read number, in the latter case to the position of the next non-white-
   * space character
   */
  dummy = (char*)malloc(dummy_length * sizeof(char));
  /* read the sample matrix entries */
  printf("MESSAGE: Sample matrix:\n");
  for(iLoop=0; iLoop < nr_samples_per_startpoint; ++iLoop){
    for(jLoop=0; jLoop < nr_startpoints; ++jLoop){
      fscanf(parameterfile_p, "%lu", sample_matrix[jLoop] + iLoop);
      printf("%lu ", sample_matrix[jLoop][iLoop]);
    }
    fgets(dummy, dummy_length, parameterfile_p);
    printf("\n");
  }
  fclose(parameterfile_p);
  free(dummy);
  /* how many timeseries there are */
  nr_timeseries = 0;
  for(iLoop=0; iLoop<nr_sim_runs; ++iLoop)
    nr_timeseries += nr_sub_sim_runs[iLoop];
  nr_timeseries *= nr_startpoints;
  /* how many configurations were read */
  nr_configurations = nr_timeseries * nr_samples_per_startpoint;
  /* sanity check */
  if(nr_timeseries == 0)
    error("ERROR: There are no timeseries", HERE);
```

```
  else{
    printf("MESSAGE: There are %u timeseries.\n", nr_timeseries);
    printf("MESSAGE: There are %u configurations.\n", nr_configurations);
  }
  /* compute number of monomers in the melt */
  nr_monomers = nr_chains * mon_per_chain;
  /*************************************************************************
   *           E N D   O F   C O M M O N   C O D E                         *
   *************************************************************************/
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Now in main(), file %s, line %u\n", HERE);
#endif
  if(strcmp(saving_scheme, "list_yasp") == 0
     || strcmp(saving_scheme, "matrix_yasp") == 0){
    /* tell user that we swap y and z for 2-d systems */
    if(DIM == 2)
      printf("MESSAGE: Swapping y <-> z in YASP frames for 2-d films\n");
    /* allocate frame */
    AllocYaspFrame(&frame_t, nr_monomers);
  }/* if yasp */
  /* fluid configuration */
  conf_t = (r3vector*)calloc(nr_monomers, sizeof(r3vector));
  /* chain centers of mass */
  Rcm_chains_t = (r3vector*)calloc(nr_chains, sizeof(r3vector));
  /* wall configuration */
  wall_conf_t  = (r3vector*)calloc(nr_wall_atoms, sizeof(r3vector));
  /* here we generate a list containing the chain lengths.
   * This is not needed at present, as we have a fixed length equal for all
   * chains.  But maybe we want to investigate models with different chain
   * lengths one fine day.
   */
  chainlengths = (unsigned int *)calloc(nr_chains, sizeof (unsigned int));
  /* initialize according to our model */
  for(iLoop=0; iLoop < nr_chains; ++iLoop)
    chainlengths[iLoop] = mon_per_chain;
  /* allocate memory for the results */
  /***********************************/
  /* number of monomer pairs (with index b >= index a) */
  nr_mon_pairs = SQUARE(mon_per_chain)/2 + (mon_per_chain +1)/2;
  /* in which column-index the data starts
   * layout:
   * q | q in bin | weight | data ....
   */
  sf_data_start_index = 3;
  sf_data_cm_start_index = 3;
  /* instead of |q| we write q_x, q_y for the bin and the weighed center of
   * the bin, thus we need 2 columns more */
  sf_data_xy_start_index = 5;
  sf_data_cm_xy_start_index = 5;
  /* number of lines = number of bins */
  nr_result_lines = nrBins;
  nr_result_cm_lines = nrBins;
  /* note: nrBins_xy is the number of bins in *one direction*.
   * we support only quadratic lattices although extension to non-quadratic
   * lattices is prepared */
  nrResultBins_xy = SQUARE(nrBins_xy);
  nr_result_xy_lines = nrResultBins_xy;
  nr_result_cm_xy_lines = nrResultBins_xy;
#ifndef CALC_ONLY_COHERENT_SF_YES
  printf("MESSAGE: Calculating monomer separated structure factors.\n");
  nr_result_columns = sf_data_start_index + 2 + 2 * nr_mon_pairs;
  nr_result_cm_columns = sf_data_cm_start_index + 1 + 2 * mon_per_chain;
  nr_result_xy_columns = sf_data_xy_start_index + 2 + 2 * nr_mon_pairs;
  nr_result_cm_xy_columns = sf_data_cm_xy_start_index + 1 + 2 * mon_per_chain;
#else
  /* if only the structure factor of the melt and the chains centers of mass
   * should be computed */
  printf("MESSAGE: Calculating only coherent structure factors.\n");
  nr_result_columns = sf_data_start_index + 2;
  nr_result_cm_columns = sf_data_cm_start_index + 1;
  nr_result_xy_columns = sf_data_xy_start_index + 2;
  nr_result_cm_xy_columns = sf_data_cm_xy_start_index + 1;
#endif
  results = srtype_matrix(0, nr_result_lines -1, 0, nr_result_columns -1);
  for(iLoop=0; iLoop < (unsigned)nr_result_lines; ++iLoop)
    for(jLoop=0; jLoop < (unsigned)nr_result_columns; ++jLoop)
      results[iLoop][jLoop] = 0.0;
  results_cm = srtype_matrix(0, nr_result_cm_lines -1, \
     0, nr_result_cm_columns -1);
  for(iLoop=0; iLoop < (unsigned)nr_result_cm_lines; ++iLoop)
    for(jLoop=0; jLoop < (unsigned)nr_result_cm_columns; ++jLoop)
      results_cm[iLoop][jLoop] = 0.0;
  if(l_write_xy_plane){
    results_xy = srtype_matrix(0, nr_result_xy_lines -1, \
 0, nr_result_xy_columns -1);
    for(iLoop=0; iLoop < (unsigned)nr_result_xy_lines; ++iLoop)
      for(jLoop=0; jLoop < (unsigned)nr_result_xy_columns; ++jLoop)
results_xy[iLoop][jLoop] = 0.0;
    results_cm_xy = srtype_matrix(0, nr_result_cm_xy_lines -1, \
    0, nr_result_cm_xy_columns -1);
    for(iLoop=0; iLoop < (unsigned)nr_result_cm_xy_lines; ++iLoop)
      for(jLoop=0; jLoop < (unsigned)nr_result_cm_xy_columns; ++jLoop)
results_cm_xy[iLoop][jLoop] = 0.0;
  }

  /* print results to screen */
  /*
  for(iLoop=0; iLoop < (unsigned)nr_result_cm_lines; ++iLoop){
    for(jLoop=0; jLoop < (unsigned)nr_result_cm_columns; ++jLoop)
      fprintf(stdout,"%.4E ", results_cm[iLoop][jLoop]);
    fprintf(stdout,"\n");
  }
  */
  /* get maximal q-value we consider, allocate q-vector related arrays */
  maxQvalue = MAX(((double)nrBins*binWidth),((double)nrBins_xy*binWidth_xy));
  printf("MESSAGE: Biggest q-value to consider = %f\n", maxQvalue);
  latticeUnits = (double*)calloc(DIM, sizeof(double));
  nr_qvecs_in_bins = (unsigned int *)calloc(nrBins, \
    sizeof(unsigned int));
  nr_confs_in_bins = (unsigned int *)calloc(nrBins, \
    sizeof(unsigned int));
  qvalues_in_bins = (double*)calloc(nrBins, sizeof(double));
  if(l_write_xy_plane){
    nr_qvecs_in_xy_bins = (unsigned int *)calloc(nrResultBins_xy, \
 sizeof(unsigned int));
    nr_confs_in_xy_bins = (unsigned int *)calloc(nrResultBins_xy, \
 sizeof(unsigned int));
    /* qvalues_in_xy_bins will be initialized in get_qvecs_in_xy_bins() */
    qvalues_in_xy_bins = dmatrix(0, nrResultBins_xy -1, 0, 1);
  }/* if(l_write_xy_plane) */
  if(l_const_volume == TRUE){
    printf("MESSAGE: Assuming constant volume.\n");
    for(iLoop = 0; iLoop < DIM; ++iLoop){
      latticeUnits[iLoop] = (2.0 * PI * (double)spreadLatticeFactor) \
/ boundary[iLoop];
```

```
      /* maximal indices needed for generating lattice vectors in the range of
       * q-vectors we want to consider.
       * we want to make sure that the last bin is completely filled, thus +1:
       */
      max_qindex[iLoop] = \
(int)(MAX(((double)nrBins*binWidth), \
  ((double)nrBins_xy*binWidth_xy))/latticeUnits[iLoop]) +1;
      printf("MESSAGE: latticeUnits[%u]=%f, max_qindex[%u] = %u\n", \
     iLoop, latticeUnits[iLoop], iLoop, max_qindex[iLoop]);
    }
    /* get overall maximal index for allocation */
    for(iLoop = 0; iLoop < DIM; ++iLoop)
      if(max_qindex[iLoop] >= max_qindex[(iLoop+1)%DIM] \
&& max_qindex[iLoop] >= max_qindex[(iLoop+DIM-1)%DIM])
total_max_qindex = max_qindex[iLoop];
    printf("MESSAGE: total_max_qindex = %u\n", total_max_qindex);
    /* allocate memory for storage of intermediate results */
    allocate_all_sin_cos(nr_monomers, nr_chains, total_max_qindex, \
 &sines, &cosines, &sines_cm, &cosines_cm);
    /* make the mapping onto the bins */
    if(l_write_xy_plane)
      get_qvecs_in_xy_bins(max_qindex, latticeUnits, \
   binWidth_xy, binWidth_xy, nrBins_xy, nrBins_xy, \
   nr_qvecs_in_xy_bins, qvalues_in_xy_bins);
    get_qvecs_in_bins(max_qindex, latticeUnits, binWidth, nrBins, \
      nr_qvecs_in_bins, qvalues_in_bins);
  }else{
    /* just allocate arrays whose size does not depend on boxsize */
    alloc_all_sin_cos_arr(nr_monomers, nr_chains);
  } /* if(l_const_volume == TRUE) */
  /*************************************************************************
   *                          M A I N   L O O P                            *
   *************************************************************************/
  /* loop over all simulation runs */
  for(iLoop=0; iLoop < nr_sim_runs; ++iLoop){
    /* loop over sub runs in a simulation run. */
    for(jLoop=0; jLoop < nr_sub_sim_runs[iLoop]; ++jLoop){
      /* complete the paths with the number of the simulation run and
 sub run */
      stringlength = strlen(data_path);
      strncpy(data_path_temp, data_path, (size_t)stringlength);
      /* integer to ASCII conversion needed */
      if(iLoop >= 10)
data_path_temp[stringlength++] = (char)(iLoop / 10) + '0';
      data_path_temp[stringlength++] = (char)(iLoop % 10) + '0';
      if(nr_dir_levels > 1){
data_path_temp[stringlength++] = '.';
if(jLoop >= 10)
  data_path_temp[stringlength++] = (char)(jLoop / 10) + '0';
data_path_temp[stringlength++] = (char)(jLoop % 10) + '0';
/* that's for example /data/bla_bla_bla/conf4.5 */
      }
      /*  terminate string */
      data_path_temp[stringlength] = '\0';
      /* loop over all starting points */
      for(kLoop=0; kLoop < nr_startpoints; ++kLoop){
       /*********************************************************************
 *                    LOOP OVER ONE TIMESERIES                           *
 *********************************************************************/
for(lLoop=0; lLoop < nr_samples_per_startpoint; ++lLoop){
  printf("MESSAGE: Calculating for run %u, sub run %u, startpoint %u, \
sample %u ...\n", iLoop, jLoop, kLoop, lLoop);
  fflush(stdout);
  if(strcmp(saving_scheme, "list_yasp") == 0
     || strcmp(saving_scheme, "matrix_yasp") == 0){
    conf_read_status=readconf_yasp(data_path, &frame_t, nr_monomers,
   sample_matrix[kLoop][lLoop],conf_t);
  }else if(DIM == 3){
    conf_read_status = \
      readconf_rs(data_path_temp, pos_file_prefix, pos_file_postfix, \
  nr_monomers, sample_matrix[kLoop][lLoop], conf_t);
  }else if(DIM == 2 && \
   ((strcmp(saving_scheme, "list_2d") == 0) || \
    (strcmp(saving_scheme, "matrix_2d") == 0))){
    conf_read_status = \
      readconf_rs_2d(data_path_temp, pos_file_prefix,pos_file_postfix,\
     nr_monomers, sample_matrix[kLoop][lLoop], conf_t);
  }else if(DIM == 2){
    conf_read_status = \
      read_pos_rs_all(data_path_temp, pos_file_prefix, nr_monomers,
      nr_wall_atoms, sample_matrix[kLoop][lLoop],
      conf_t, wall_conf_t, &twall_pos_t, &bwall_pos_t);
    if(l_transform_to_cm_walls_system == TRUE)
      /* go to cm-system of the two walls */
      transform_to_walls_cm(conf_t, nr_monomers, \
    twall_pos_t, bwall_pos_t);
  }else{
    fprintf(stderr, "ERROR: Fallthrough on file reading.\n");
    exit(1);
  }
  /* check if configuration was read correctly */
  if(conf_read_status){
    printf("WARNING: Configuration %lu not correctly read\n",
   sample_matrix[kLoop][lLoop]);
    printf("          error flag value = %d\n", conf_read_status);
    if(nr_conf_read > 0){
      printf("          Exiting, setting nr_configurations=%u to %u\n",\
     nr_configurations, nr_conf_read);
      nr_configurations = nr_conf_read;
      goto premature_exit;
    }else{
      fprintf(stderr, "ERROR: No configuration was read.\n");
      exit(2);
    }
  }else{
    /* increment counter for successful reads of frames */
    nr_conf_read++;
  }
  if(l_const_volume == FALSE){
    /* get the boundaries of this frame,
     * get lattice constants, maximal q-index and
     * reallocate help arrays */
    /* right now this is only implemented for YASP trajectories */
    for(mLoop = 0; mLoop < DIM; ++mLoop)
      boundary[mLoop] = frame_t.box[mLoop][mLoop];
    if(DIM == 2 && (strcmp(saving_scheme, "list") == 0 \
    || strcmp(saving_scheme, "list_yasp") == 0)){
      /* y = z */
      boundary[1] = frame_t.box[2][2];
      /* mark boundary[2] (z) invalid */
      boundary[2] = -1.0;
    }
    for(mLoop = 0; mLoop < DIM; ++mLoop){
      latticeUnits[mLoop] = (2.0 * PI * (double)spreadLatticeFactor) \
```

```
/ boundary[mLoop];
      max_qindex[mLoop] = \
(int)(MAX(((double)nrBins*binWidth), \
  ((double)nrBins_xy*binWidth_xy))\
      / latticeUnits[mLoop]) +1;
    }
    /* get overall maximal index for allocation */
    for(mLoop = 0; mLoop < DIM; ++mLoop)
      if(max_qindex[mLoop] >= max_qindex[(mLoop+1)%DIM] \
 && max_qindex[mLoop] >= max_qindex[(mLoop+DIM-1)%DIM])
total_max_qindex = max_qindex[mLoop];
    if(DIM == 3){
      printf("MESSAGE: boundaries: %lf, %lf, %lf; total_max_qindex=\
%u\n", boundary[0], boundary[1], boundary[2], total_max_qindex);
    }else{
      printf("MESSAGE: boundaries: %lf, %lf; total_max_qindex=%u\n", \
boundary[0], boundary[1], total_max_qindex);
    }
    /* check if total_max_qindex has increased and reallocate memory
     *  in this case */
    realloc_all_sin_cos(nr_monomers, nr_chains, total_max_qindex, \
&sines, &cosines, &sines_cm, &cosines_cm);
    /* make the mapping onto the bins */
    /* This is redundant, as we have to map each q-vector into bins
     * when calculating the structure factors anyways for fluctuating
     * simulation boxes.  But this is not causing significant CPU
     * overhead and allows to use the constant volume routines.
     */
    if(l_write_xy_plane){
      get_qvecs_in_xy_bins(max_qindex, latticeUnits, \
   binWidth_xy, binWidth_xy, nrBins_xy, \
   nrBins_xy, \
   nr_qvecs_in_xy_bins, qvalues_in_xy_bins);
    }
    get_qvecs_in_bins(max_qindex, latticeUnits, binWidth, nrBins, \
      nr_qvecs_in_bins, qvalues_in_bins);
  } /* if(l_const_volume == FALSE) */
  if(l_write_xy_plane){
    /* look in which bins we have entries from this conf (and count
     * number of configurations) */
    get_confs_in_bins(nrResultBins_xy, nr_qvecs_in_xy_bins, \
      nr_confs_in_xy_bins);
    for(mLoop=0; mLoop < (unsigned)nr_result_xy_lines; ++mLoop){
      results_xy[mLoop][2] += qvalues_in_xy_bins[mLoop][0];
      results_xy[mLoop][3] += qvalues_in_xy_bins[mLoop][1];
      results_xy[mLoop][4] += (double)nr_qvecs_in_xy_bins[mLoop];
    }
    for(mLoop=0; mLoop < (unsigned)nr_result_cm_xy_lines; ++mLoop){
      results_cm_xy[mLoop][2] += qvalues_in_xy_bins[mLoop][0];
      results_cm_xy[mLoop][3] += qvalues_in_xy_bins[mLoop][1];
      results_cm_xy[mLoop][4] += (double)nr_qvecs_in_xy_bins[mLoop];
    }
  }/* if(l_write_xy_plane) */
  get_confs_in_bins(nrBins, nr_qvecs_in_bins, nr_confs_in_bins);
  for(mLoop=0; mLoop < (unsigned)nr_result_lines; ++mLoop){
    results[mLoop][1] += qvalues_in_bins[mLoop];
    results[mLoop][2] += (double)nr_qvecs_in_bins[mLoop];
  }
  for(mLoop=0; mLoop < (unsigned)nr_result_cm_lines; ++mLoop){
    results_cm[mLoop][1] += qvalues_in_bins[mLoop];
    results_cm[mLoop][2] += (double)nr_qvecs_in_bins[mLoop];
  }
  /* compute the centers of mass */
  calc_Rcm_chains(conf_t, nr_chains, chainlengths, Rcm_chains_t);
  /* we don't go to periodized coordinates as this would destroy the
   * *self* motion of the particles.  Besides, the reciprocal lattice
   * vectors take care of the periodic boundary conditions.
   */
  /* calculate for this configuration */
    calc_all_sin_cos(latticeUnits, nr_monomers, conf_t, max_qindex,
   sines, cosines);
    calc_all_sin_cos(latticeUnits, nr_chains, Rcm_chains_t, max_qindex,
   sines_cm, cosines_cm);
#ifdef THREE_D
    calc_self_and_distinct_sf_3d(sines, cosines, sines_cm, cosines_cm,
 mon_per_chain, nr_monomers, nr_chains,
 max_qindex, latticeUnits,
 nrBins, binWidth,
 sf_data_start_index, results,
 sf_data_cm_start_index, results_cm);
#else
    calc_self_and_distinct_sf_2d(sines, cosines, sines_cm, cosines_cm,
 mon_per_chain, nr_monomers, nr_chains,
 max_qindex, latticeUnits,
 l_write_xy_plane,
 binWidth, nrBins,
 sf_data_start_index, results,
 sf_data_cm_start_index, results_cm,
 binWidth_xy,  binWidth_xy,
 nrBins_xy, nrBins_xy,
 sf_data_xy_start_index, results_xy,
 sf_data_cm_xy_start_index,
 results_cm_xy);
#endif
}/* end loop over one timeseries */
      }/* end loop over all startpoints */
    }/* end loop over all sub runs */
  }/* end loop over all simulation runs */
  /*****************************************************************************
   *                    D O N E   W I T H   L O O P S                          *
   *****************************************************************************/
  /* goto label when a frame could not be read correctly */
  premature_exit :
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Now in main(), file %s, line %u\n", HERE);
#endif
  /* close yasp trajectory */
  if(strcmp(saving_scheme, "list_yasp") == 0
     || strcmp(saving_scheme, "matrix_yasp") == 0){
    CloseYaspTrj();
  }
  /*
  printf("DEBUG: nr_qvecs_in_xy_bins:\n");
  for(iLoop = 0; iLoop < (unsigned)nrResultBins_xy; iLoop++)
    printf("%u ", nr_qvecs_in_xy_bins[iLoop]);
  printf("\n");
  printf("DEBUG: nr_confs_in_xy_bins:\n");
  for(iLoop = 0; iLoop < (unsigned)nrResultBins_xy; iLoop++)
    printf("%u ", nr_confs_in_xy_bins[iLoop]);
  printf("\n");
  */
  if(l_write_xy_plane){
    /* calculate summed quantities, normalize */
    for(iLoop = 0; iLoop < (unsigned)nr_result_xy_lines; iLoop++){
#ifndef CALC_ONLY_COHERENT_SF_YES
      column = sf_data_xy_start_index +2;
      coh_self_sf = 0.0;
      coh_all_sf = 0.0;
      for(monIndex1 = 0; monIndex1 < mon_per_chain; ++monIndex1){
for(monIndex2 = monIndex1; monIndex2 < mon_per_chain; ++monIndex2){
  results_xy[iLoop][column] /= (double)nr_chains;
  results_xy[iLoop][column + nr_mon_pairs] /= (double)nr_chains;
  if(monIndex1 == monIndex2){
    /* we didn't compute the self contribution, because it must be 1.0
     * anyways.  However, this identity makes a good test */
    results_xy[iLoop][column] = results_xy[iLoop][4];
    coh_self_sf += results_xy[iLoop][column];
    coh_all_sf += results_xy[iLoop][column + nr_mon_pairs];
  }else{ /* use symmetry relation */
    coh_self_sf += 2.0 * results_xy[iLoop][column];
    coh_all_sf += 2.0 * results_xy[iLoop][column + nr_mon_pairs];
  }
  /* go to next column */
  column++;
}
      }
      /* debug check */
      if(column + nr_mon_pairs != nr_result_xy_columns){
printf("WARNING: column+nr_mon_pairs=%u != nr_result_xy_columns=%u\n",\
       column + nr_mon_pairs, nr_result_xy_columns);
      }
      results_xy[iLoop][sf_data_xy_start_index] \
= coh_self_sf / (double)mon_per_chain;
      results_xy[iLoop][sf_data_xy_start_index +1] \
= coh_all_sf / (double)mon_per_chain;
#else
      results_xy[iLoop][sf_data_xy_start_index]    /= (double)nr_monomers;
      results_xy[iLoop][sf_data_xy_start_index +1] /= (double)nr_monomers;
#endif
    }
    /* write out q information,
     * normalize by number of q-vectors in each bin with function data */
    for(iLoop = 0; iLoop < (unsigned)nr_result_xy_lines; ++iLoop){
      if(results_xy[iLoop][4] > 0)
for(jLoop = sf_data_xy_start_index; \
    jLoop < (unsigned)nr_result_xy_columns; ++jLoop)
  results_xy[iLoop][jLoop] /= results_xy[iLoop][4];
      results_xy[iLoop][0] = ((double)(iLoop/nrBins_xy)+0.5)*binWidth_xy;
      results_xy[iLoop][1] = ((double)(iLoop%nrBins_xy)+0.5)*binWidth_xy;
      if(nr_confs_in_xy_bins[iLoop] > 0){
results_xy[iLoop][2] /= (double)nr_confs_in_xy_bins[iLoop];
results_xy[iLoop][3] /= (double)nr_confs_in_xy_bins[iLoop];
      }else if(results_xy[iLoop][2] != 0.0 || results_xy[iLoop][3] != 0.0){
printf("WARNING: q-vectors in bins which are supposed to be empty\n");
      }
      /* for the statistical weight we divide by the total number of
       * configurations */
      results_xy[iLoop][4] /= (double)nr_configurations;
    }
    /* create a descriptive header for the output */
    nrDataDescriptionLines = 6;
    dataDescription = charmatrix(0, nrDataDescriptionLines-1, 0, 160);
    line = 0;
    sprintf(dataDescription[line], \
    "# Static structure factors of monomers\n");
    ++line;
    sprintf(dataDescription[line], \
    "# used %u configurations\n", nr_configurations);
    ++line;
    sprintf(dataDescription[line], \
    "# data generated by %s (compiled %s)\n", WHAT, __DATE__);
    ++line;
    sprintf(dataDescription[line], \
    "# bin-width = %lf, number bins = %u, q_max = %lf\n", \
    binWidth_xy, nrBins_xy, maxQvalue);
    ++line;
#ifndef CALC_ONLY_COHERENT_SF_YES
    sprintf(dataDescription[line], \
    "#  q_x | q_y | q_x in bin | q_y in bin|   weight   | S_self(q) | \
 S_all(q) | S^{a,b}_self(q), b>=a ... | S^{a,b}_all(q), b>=a ...\n");
#else
    sprintf(dataDescription[line], \
    "#  q_x | q_y | q_x in bin | q_y in bin|   weight   | S_self(q) | \
S_all(q)\n");
#endif
    ++line;
    sprintf(dataDescription[line], "#\n");
    ++line;
    /* line count check */
    if(line != nrDataDescriptionLines){
      fprintf(stderr, "ERROR: line=%u != nrDataDescriptionLines=%u\n", \
      line, nrDataDescriptionLines);
      exit(2);
    }
    /* put together the name of the file used for saving */
    /* sprintf needs an char array which is big enough for its result ! */
    if(argc == 3){
      sprintf(string, "%s.%s", \
      "Stat_struct_facts_xy", argv[2]);
    }else{
      sprintf(string, "%s_q_le_%.1f_bw%.2f.dat", \
    "Static_structure_factors_xy", \
    maxQvalue, binWidth_xy);
    }
#ifdef DIAGNOSTICS_ON
    printf("DIAGNOSTICS: Now in main(), file %s, line %u\n", HERE);
#endif
    /* write out results */
    write_results_sf(results_xy, nr_result_xy_lines, \
     nr_result_xy_columns,
     sf_data_xy_start_index -1, l_write_xy_plane,
     string, dataDescription,
     nrDataDescriptionLines, argv[1]);
    /*****************************************************************************/
    for(iLoop = 0; iLoop < (unsigned)nr_result_cm_xy_lines; iLoop++)
      for(jLoop = sf_data_cm_xy_start_index; \
  jLoop < (unsigned)nr_result_cm_xy_columns; jLoop++)
results_cm_xy[iLoop][jLoop] /= (double)nr_chains;
    /* write out q information,
     * normalize by number of q-vectors in each bin with function data */
    for(iLoop=0; iLoop < (unsigned)nr_result_cm_xy_lines; ++iLoop){
      if(results_cm_xy[iLoop][4] > 0)
for(jLoop=sf_data_cm_xy_start_index; \
    jLoop < (unsigned)nr_result_cm_xy_columns; ++jLoop)
  results_cm_xy[iLoop][jLoop] /= results_cm_xy[iLoop][4];
      results_cm_xy[iLoop][0] = ((double)(iLoop/nrBins_xy)+0.5)*binWidth_xy;
      results_cm_xy[iLoop][1] = ((double)(iLoop%nrBins_xy)+0.5)*binWidth_xy;
      if(nr_confs_in_xy_bins[iLoop] > 0){
results_cm_xy[iLoop][2] /= (double)nr_confs_in_xy_bins[iLoop];
results_cm_xy[iLoop][3] /= (double)nr_confs_in_xy_bins[iLoop];
      }else if(results_cm_xy[iLoop][2]!=0.0 || results_cm_xy[iLoop][3]!=0.0){
```

```
printf("WARNING: q-vectors in xy-bins which should be empty\n");
      }
      /* for the statistical weight we divide by the total number of
       * configurations */
      results_cm_xy[iLoop][4] /= (double)nr_configurations;
    }
    /* create header */
    line = 0;
#ifndef CALC_ONLY_COHERENT_SF_YES
    sprintf(dataDescription[line], \
    "# Static structure factors of centers of mass and of monomers in \
chain relative to centers of mass\n");
#else
    sprintf(dataDescription[line], \
    "# Static structure factor of chain centers of mass\n");
#endif
    ++line;
    sprintf(dataDescription[line], \
    "# used %u configurations\n", nr_configurations);
    ++line;
    sprintf(dataDescription[line], \
    "# data generated by %s (compiled %s)\n", WHAT, __DATE__);
    ++line;
    sprintf(dataDescription[line], \
    "# bin-width = %lf, number bins = %u, q_max = %lf\n", \
    binWidth_xy, nrBins_xy, maxQvalue);
    ++line;
#ifndef CALC_ONLY_COHERENT_SF_YES
    sprintf(dataDescription[line], \
    "#  q_x | q_y | q_x in bin | q_y in bin |   weight   | S^C(q) | \
S^{a}_self(q), a=1,...,N | S^{a}_all(q) a=1,...,N\n");
#else
    sprintf(dataDescription[line], \
    "#  q_x | q_y | q_x in bin | q_y in bin |   weight   | S^C(q)\n");
#endif
    ++line;
    sprintf(dataDescription[line], "#\n");
    ++line;
    /* line count check */
    if(line != nrDataDescriptionLines){
      fprintf(stderr, "ERROR: line=%u != nrDataDescriptionLines=%u\n", \
      line, nrDataDescriptionLines);
      exit(2);
    }
    /* put together the name of the file used for saving */
    /* sprintf needs an char array which is big enough for its result ! */
    if(argc == 3){
      sprintf(string, "%s.%s", \
      "Stat_struct_facts_chains_cm_xy", argv[2]);
    }else{
      sprintf(string, "%s_q_le_%.1f_bw%.2f.dat", \
      "Structure_factors_chain_cm_xy", \
      maxQvalue, binWidth_xy);
    }
#ifdef DIAGNOSTICS_ON
    printf("DIAGNOSTICS: Now in main(), file %s, line %u\n", HERE);
#endif
    /* write out results */
    write_results_sf(results_cm_xy, nr_result_cm_xy_lines, \
     nr_result_cm_xy_columns,
     sf_data_cm_xy_start_index -1, l_write_xy_plane,
     string, dataDescription,
     nrDataDescriptionLines, argv[1]);
    /* free memory */
    free(nr_qvecs_in_xy_bins);
    free_dmatrix(qvalues_in_xy_bins, 0, nrResultBins_xy -1, 0, 1);
    free_srtype_matrix(results_xy, 0, nr_result_xy_lines -1, \
 0, nr_result_xy_columns -1);
    free_srtype_matrix(results_cm_xy, 0, nr_result_cm_xy_lines -1, \
 0, nr_result_cm_xy_columns -1);
    free_charmatrix(dataDescription, 0, nrDataDescriptionLines-1, 0, 160);
  }
  /*****************************************************************************/
  /* D O N E   W I T H   X Y - P L A N E   D A T A                             */
  /*****************************************************************************/
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Now in main(), file %s, line %u\n", HERE);
#endif
  for(iLoop = 0; iLoop < (unsigned)nr_result_lines; iLoop++){
#ifndef CALC_ONLY_COHERENT_SF_YES
    column = sf_data_start_index +2;
    coh_self_sf = 0.0;
    coh_all_sf = 0.0;
    for(monIndex1 = 0; monIndex1 < mon_per_chain; ++monIndex1){
      for(monIndex2 = monIndex1; monIndex2 < mon_per_chain; ++monIndex2){
results[iLoop][column] /= (double)nr_chains;
results[iLoop][column + nr_mon_pairs] /= (double)nr_chains;
if(monIndex1 == monIndex2){
  /* we didn't compute the self contribution, because it must be 1.0
   * anyways.  However, this identity makes a good test */
  results[iLoop][column] = results[iLoop][2];
  coh_self_sf += results[iLoop][column];
  coh_all_sf += results[iLoop][column + nr_mon_pairs];
}else{ /* use symmetry relation */
  coh_self_sf += 2.0 * results[iLoop][column];
  coh_all_sf += 2.0 * results[iLoop][column + nr_mon_pairs];
}
column++;
      }
    }
    if(column + nr_mon_pairs != nr_result_columns){
      printf("WARNING: column + nr_mon_pairs =%u != nr_result_columns=%u\n",\
     column + nr_mon_pairs, nr_result_columns);
    }
    results[iLoop][sf_data_start_index] \
      = coh_self_sf / (double)mon_per_chain;
    results[iLoop][sf_data_start_index+1] \
      = coh_all_sf / (double)mon_per_chain;
#else
    results[iLoop][sf_data_start_index]   /= (double)nr_monomers;
    results[iLoop][sf_data_start_index+1] /= (double)nr_monomers;
#endif
  }/* end loop over bins */
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Now in main(), file %s, line %u\n", HERE);
#endif
  /* write out q information,
   * normalize by number of q-vectors in each bin with function data */
  for(iLoop=0; iLoop < (unsigned)nr_result_lines; ++iLoop){
    if(results[iLoop][2] > 0)
      for(jLoop=sf_data_start_index; \
  jLoop < (unsigned)nr_result_columns; ++jLoop)
results[iLoop][jLoop] /= results[iLoop][2];
    results[iLoop][0] = ((double)iLoop + 0.5)*binWidth;
    if(nr_confs_in_bins[iLoop] > 0)
      results[iLoop][1] /= (double)nr_confs_in_bins[iLoop];
    results[iLoop][2] /= (double)nr_configurations;
  }
  /* create a descriptive header for the output */
  nrDataDescriptionLines = 6;
  dataDescription = charmatrix(0, nrDataDescriptionLines-1, 0, 160);
  line = 0;
  sprintf(dataDescription[line], \
  "# Static structure factors of monomers\n");
  ++line;
  sprintf(dataDescription[line], \
  "# used %u configurations\n", nr_configurations);
  ++line;
  sprintf(dataDescription[line], \
  "# data generated by %s (compiled %s)\n", WHAT, __DATE__);
  ++line;
  sprintf(dataDescription[line], \
    "# bin-width = %lf, number bins = %u, q_max = %lf\n", \
  binWidth, nrBins, maxQvalue);
  ++line;
#ifndef CALC_ONLY_COHERENT_SF_YES
  sprintf(dataDescription[line], \
  "#  q | q in bin |   weight   | S_self(q) |  S_all(q) | S^{a,b}_\
self(q), b>=a ...  S^{a,b}_all(q), b>=a\n");
#else
  sprintf(dataDescription[line], \
  "#  q | q in bin |   weight   | S_self(q) |  S_all(q)\n");
#endif
  ++line;
  sprintf(dataDescription[line], "#\n");
  ++line;
  /* line count check */
  if(line != nrDataDescriptionLines){
    fprintf(stderr, "ERROR: line=%u != nrDataDescriptionLines=%u\n", \
    line, nrDataDescriptionLines);
    exit(2);
  }
  /* put together the name of the file used for saving */
  /* sprintf needs an char array which is big enough for its result ! */
  if(argc == 3){
    sprintf(string, "%s.%s", \
    "Stat_struct_facts", argv[2]);
  }else{
    sprintf(string, "%s_q_le_%.1f_bw%.2f.dat", \
    "Static_structure_factors", maxQvalue, binWidth);
  }
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Now in main(), file %s, line %u\n", HERE);
#endif
  /* write out results */
  write_results_sf(results, nr_result_lines, \
   nr_result_columns,
   sf_data_start_index -1, FALSE,
   string, dataDescription,
   nrDataDescriptionLines, argv[1]);
  /*****************************************************************************/
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Now in main(), file %s, line %u\n", HERE);
#endif
  for(iLoop = 0; iLoop < (unsigned)nr_result_cm_lines; iLoop++)
    for(jLoop = sf_data_cm_start_index; \
jLoop < (unsigned)nr_result_cm_columns; jLoop++){
      results_cm[iLoop][jLoop] /= (double)nr_chains;
    }
  for(iLoop=0; iLoop < (unsigned)nr_result_cm_lines; ++iLoop){
    if(results_cm[iLoop][2] > 0)
      for(jLoop = sf_data_cm_start_index; \
  jLoop < (unsigned)nr_result_cm_columns; ++jLoop)
results_cm[iLoop][jLoop] /= results_cm[iLoop][2];
    results_cm[iLoop][0] = ((double)iLoop + 0.5)*binWidth;
    if(nr_confs_in_bins[iLoop] > 0)
      results_cm[iLoop][1] /= (double)nr_confs_in_bins[iLoop];
    results_cm[iLoop][2] /= (double)nr_configurations;
  }
  /* create header */
  line = 0;
#ifndef CALC_ONLY_COHERENT_SF_YES
  sprintf(dataDescription[line], \
  "# Static structure factors of centers of mass and of monomers in \
chain relative to centers of mass\n");
#else
  sprintf(dataDescription[line], \
  "# Static structure factor of chain centers of mass\n");
#endif
  ++line;
  sprintf(dataDescription[line], \
  "# used %u configurations\n", nr_configurations);
  ++line;
  sprintf(dataDescription[line], \
  "# data generated by %s (compiled %s)\n", WHAT, __DATE__);
  ++line;
  sprintf(dataDescription[line], \
  "# bin-width = %lf, number bins = %u, q_max = %lf\n", \
  binWidth, nrBins, maxQvalue);
  ++line;
#ifndef CALC_ONLY_COHERENT_SF_YES
  sprintf(dataDescription[line], \
  "#  q  | q in bin |   weight   | S^C(q) | S^{a}_self(q) | S^{a}_\
all(q)\n");
#else
  sprintf(dataDescription[line], \
  "#  q  | q in bin |   weight   | S^C(q)\n");
#endif
  ++line;
  sprintf(dataDescription[line], "#\n");
  ++line;
  /* line count check */
  if(line != nrDataDescriptionLines){
    fprintf(stderr, "ERROR: line=%u != nrDataDescriptionLines=%u\n", \
    line, nrDataDescriptionLines);
    exit(2);
  }
  /* put together the name of the file used for saving */
  /* sprintf needs an char array which is big enough for its result ! */
  if(argc == 3){
    sprintf(string, "%s.%s", \
    "Stat_struct_facts_chain_cm", argv[2]);
  }else{
    sprintf(string, "%s_q_le_%.1f_bw%.2f.dat", \
    "Static_structure_factors_chain_cm", \
    maxQvalue, binWidth);
  }
  /* write out results */
  write_results_sf(results_cm, nr_result_cm_lines, \
   nr_result_cm_columns,
   sf_data_cm_start_index -1, FALSE,
   string, dataDescription,
   nrDataDescriptionLines, argv[1]);
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Now in main(), file %s, line %u\n", HERE);
```

```
#endif
  /* free memory */
  free(nr_qvecs_in_bins);
  free(qvalues_in_bins);
  free_srtype_matrix(results, 0, nr_result_lines -1, \
     0, nr_result_columns -1);
  free_srtype_matrix(results_cm, 0, nr_result_cm_lines-1, \
     0, nr_result_cm_columns-1);
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Now in main(), file %s, line %u\n", HERE);
#endif
  /*****************************************************************************/
  /* free memory */
  free(conf_t);
  free(wall_conf_t);
  free(Rcm_chains_t);
  free(chainlengths);
  free_charmatrix(dataDescription, 0, nrDataDescriptionLines-1, 0, 160);
  free_all_sin_cos(nr_monomers, nr_chains, total_max_qindex, sines, cosines, \
   sines_cm, cosines_cm);
  printf("MESSAGE: %s says goodbye\n", argv[0]);
  return 0;
}/* end main() */
```

Structure factor library header

```
/* structure_factor_libVx.y.h
 * headers for the strcture factor library
 * Martin Aichele, 2001-09-05
 * last changed 2002-11-14
 */
#ifdef __cplusplus
extern "C" {
#endif
#ifndef _STRUCTURE_FACTOR_LIB_H
#define _STRUCTURE_FACTOR_LIB_H
#include "general_libV1.1.h"
/* type for storing the results of the structure factor calculation
 * if memory is tight use floats (but check if the results are insensitive)
 */
typedef double STRUF_RES_TYPE;
/* if only the structure factor of the melt and the chains centers of mass
 * should be computed */
#define CALC_ONLY_COHERENT_SF_YES
#include<limits.h>
/* dynamically allocates memory for a matrix(nrl..nrh)(ncl..nch)
   with STRUF_RES_TYPE entries; taken from "Numerical Recipes" */
STRUF_RES_TYPE **srtype_matrix(const int, const int, const int, const int);
/* frees the memory taken by the STRUF_RES_TYPE matrix */
void free_srtype_matrix(STRUF_RES_TYPE **m, const int, const int, \
const int, const int);
/* re-allocate memory for computations if necessary */
void realloc_all_sin_cos(const unsigned int,
 const unsigned int,
 const unsigned int,
 double ****, double ****,
 double ****, double ****);
/* allocate helper arrays in case we allocate the tensors and arrays
 * separately */
void alloc_all_sin_cos_arr(const unsigned int,
   const unsigned int);
/* allocate memory for computations */
void allocate_all_sin_cos(const unsigned int,
  const unsigned int,
  const unsigned int,
  double ****, double ****,
  double ****, double ****);
void free_all_sin_cos(const unsigned int,
      const unsigned int,
      const unsigned int,
      double ***, double ***,
      double ***, double ***);
/* calculate all sines and cosines of all positions and all q-vectors
 * in the lattice.
 * In principle one could do the same by calculating potencies:
 * (cos(x) + i sin(x))^n = (cos(nx) + i sin(nx)),
 * which is a bit faster but might cause numerical problems due to accumulated
 * round off errors. */
void calc_all_sin_cos(const double * const,
      const unsigned int,
      const r3vector * const,
      const int[],
      double *** const,
      double *** const);
/* given a triplet of  q-vector indices (integer) and a particle index,
 * return cos(\vec{q}\cdot\vec{r}_i).
 * It is more efficient to loop over the q-lattice and treat the signs in the
 * loop, but this routine allows to look at a specific set of q-vectors in any
 * order. */
double cos_qr(const int * const,
      const unsigned int,
      double *** const,
      double *** const);
/* the same as cos_qr for the sine */
double sin_qr(const int * const,
      const unsigned int,
      double *** const,
      double *** const);
/* get the number and average values of q-vectors in each bin */
void
get_qvecs_in_bins(const int[],
  const double * const,
  double,
  const unsigned int,
  unsigned int * const,
  double * const);
/* get the number and average values of q-vectors in each bin.
 * This is a version for a 2-d plane */
void
get_qvecs_in_xy_bins(const int[],
     const double * const,
     const double,
     const double,
     const unsigned int,
     const unsigned int,
     unsigned int * const,
     double ** const);
/* finds out to which bins a configuration contributes */
void get_confs_in_bins(const unsigned int,
       const unsigned int * const,
       unsigned int * const);
#ifdef THREE_D
/* take the single dimension sines and cosines.  Calculate the full 3-d
 * sin(\vec{q} \in \vec{r}) and cos(\vec{q} \in \vec{r}) values and sum these
 * up to the self and distinct structure factors (done by other functions).
 */
void
calc_self_and_distinct_sf_3d(double *** const,
     double *** const,
     double *** const,
     const unsigned int,
     const unsigned int,
     const int[],
     const double * const,
     const unsigned int,
     const double,
     const unsigned int,
     STRUF_RES_TYPE ** const,
     const unsigned int,
     STRUF_RES_TYPE ** const);
/* This function has flags to decide in which dimensions the q-vectors are 0.
 * If q-indices are zero fewer equivalent images of the positive octant
 * exist.
 */
void
low_d_calc_all_sf_3d(double *** const,
     double *** const,
     double *** const,
     double *** const,
     const unsigned int,
     const unsigned int,
     const unsigned int,
     const int[],
     const double * const,
     const unsigned int,
     const double,
     const unsigned int,
     STRUF_RES_TYPE ** const,
     const unsigned int,
     STRUF_RES_TYPE ** const,
     const unsigned int,
     const unsigned int,
     const unsigned int);
/* given sin(\vec{q} \in \vec{r_i}) and cos(\vec{q} \in \vec{r_i}) for all
 * particle and center of mass positions
 * calculate the self and distinct structure factors.
 * This is were the CPU time goes!
 */
void
assign_self_dist_sf_3d(const unsigned int,
       const unsigned int,
       const unsigned int,
       STRUF_RES_TYPE * const,
       const unsigned int,
       STRUF_RES_TYPE * const);
#else /* THREE_D */
/* take the single dimension sines and cosines.  Calculate the full 3-d
 * sin(\vec{q} \in \vec{r}) and cos(\vec{q} \in \vec{r}) values and sum these
 * up to the self and distinct structure factors (done by other functions).
 */
void
calc_self_and_distinct_sf_2d(double *** const,
     double *** const,
     double *** const,
     double *** const,
     const unsigned int,
     const unsigned int,
     const unsigned int,
     const int[],
     const double * const,
     const boolean,
     const double,
     const unsigned int,
     const unsigned int,
     STRUF_RES_TYPE ** const,
     const unsigned int,
     STRUF_RES_TYPE ** const,
     const double,
     const double,
     const unsigned int,
     const unsigned int,
     const unsigned int,
     STRUF_RES_TYPE ** const,
     const unsigned int,
     STRUF_RES_TYPE ** const);
/* This function has flags to decide in which dimensions the q-vectors are 0.
 * If q-indices are zero fewer equivalent images of the positive quadrant
 * exist.
 */
void
low_d_calc_all_sf_2d(double *** const,
     double *** const,
     double *** const,
     double *** const,
     const unsigned int,
     const unsigned int,
     const unsigned int,
     const int[],
     const double * const,
     const boolean,
     const double,
     const unsigned int,
     const unsigned int,
     STRUF_RES_TYPE ** const,
     const unsigned int,
     STRUF_RES_TYPE ** const,
     const double,
     const double,
     const unsigned int,
     const unsigned int,
     const unsigned int,
     STRUF_RES_TYPE ** const,
     const unsigned int,
     STRUF_RES_TYPE ** const,
     const unsigned int,
     const unsigned int);

/* given sin(\vec{q} \in \vec{r_i}) and cos(\vec{q} \in \vec{r_i}) for all
 * particle and center of mass positions
 * calculate the self and distinct structure factors.
 */
void
assign_self_dist_sf_2d(const unsigned int,
       const unsigned int,
       const boolean,
       const unsigned int,
       STRUF_RES_TYPE * const,
```

```
      const unsigned int,
      STRUF_RES_TYPE * const,
      const boolean,
      const unsigned int,
      STRUF_RES_TYPE * const,
      const unsigned int,
      STRUF_RES_TYPE * const);
#endif /* THREE_D */
/* This function takes a matrix with the results with its dimensions along
 * with information about what kind of data we are writing here.
 * This function is tuned for the structure factor data.
 */
void
write_results_sf(STRUF_RES_TYPE ** const,
 const unsigned int,
 const unsigned int,
 const unsigned int,
 const boolean,
 const char *,
 char **,
 const unsigned int,
 const char *);
#endif
#ifdef __cplusplus
}
#endif
```

Structure factor library code

```
/* structure_factor_libVx.y.c
 * routines for the calculation of static structure factors
 ************************************************************
 * Thanks to Hendrik Meyer for pointing out the method with saving the single
 * sines and cosines.
 * Martin Aichele, 2001-09-04
 * last modified 2002-06-03
 * small changes for fssfV1.4, 2002-11-06
 * memory layout change in V1.2, 2002-11-13
 */
#include "structure_factor_libV1.2.h"
/* debug switches */
#define DEBUG_CHAIN_AND_MELT_SF_ALL_Q_OFF
#define DEBUG_CALC_SELF_AND_DISTINCT_SF_3D_OFF
#define DEBUG_ASSIGN_SELF_DIST_SF_3D_OFF
#define DEBUG_CALC_SELF_AND_DISTINCT_SF_2D_OFF
#define DEBUG_ASSIGN_SELF_DIST_SF_2D_OFF
/* the NULL initialization allows for calling free() on these pointers before
 * allocation.  In this case, free() does nothing.
 */
static double *sin_q=NULL, *cos_q=NULL, *sin_q_cm=NULL, *cos_q_cm=NULL;
/* intermediate result arrays */
static double *sinqx_sinqy=NULL, *sinqx_cosqy=NULL, \
  *cosqx_sinqy=NULL, *cosqx_cosqy=NULL;
/* intermediate result arrays for chain centers of mass */
static double *sinqx_sinqy_cm=NULL, *sinqx_cosqy_cm=NULL, \
  *cosqx_sinqy_cm=NULL, *cosqx_cosqy_cm=NULL;
/* helper arrays */
/* static double *sin_q_helper=NULL, *cos_q_helper=NULL; */
#ifdef THREE_D
static double \
*sinqx_sinqy_sinqz=NULL, *sinqx_sinqy_cosqz, *sinqx_cosqy_sinqz, \
  *sinqx_cosqy_cosqz=NULL, *cosqx_sinqy_sinqz, *cosqx_sinqy_cosqz, \
  *cosqx_cosqy_sinqz=NULL, *cosqx_cosqy_cosqz; \
static double \
  *sinqx_sinqy_sinqz_cm=NULL, *sinqx_sinqy_cosqz_cm=NULL, \
    *sinqx_cosqy_sinqz_cm=NULL, *sinqx_cosqy_cosqz_cm=NULL, \
    *cosqx_sinqy_sinqz_cm=NULL, *cosqx_sinqy_cosqz_cm=NULL, \
    *cosqx_cosqy_sinqz_cm=NULL, *cosqx_cosqy_cosqz_cm=NULL;
#endif

/* dynamically allocates memory for a matrix(nrl..nrh)(ncl..nch)
   with STRUF_RES_TYPE entries; taken from "Numerical Recipes" */
STRUF_RES_TYPE **srtype_matrix(const int nrl, const int nrh, \
      const int ncl, const int nch){
  int i;
  STRUF_RES_TYPE **m;
  if( nrh < nrl || nch < ncl){
    fprintf(stderr, "ERROR: Wrong index structure in srtype_matrix()\n");
    exit(1);
  }
  m=(STRUF_RES_TYPE **)malloc((unsigned) (nrh-nrl+1) \
      * sizeof(STRUF_RES_TYPE*));
  if (!m) fprintf(stderr,"ERROR: Allocation failure 1 in srtype_matrix()\n");
  m -= nrl;
  for(i=nrl;i<=nrh;i++) {
    m[i]=(STRUF_RES_TYPE *)malloc((unsigned)(nch-ncl+1) \
  * sizeof(STRUF_RES_TYPE));
    if (!m[i]) fprintf(stderr, \
        "ERROR: Allocation failure 2 in srtype_matrix()\n");
    m[i] -= ncl;
  }
  return m;
}/* end srtype_matrix(....) */

/* frees the memory taken by the STRUF_RES_TYPE matrix */
void free_srtype_matrix(STRUF_RES_TYPE **m, \
const int nrl, const int nrh, \
const int ncl, const int nch){
  int i;
  if( nrh < nrl || nch < ncl){
    fprintf(stderr, "ERROR: Wrong index structure in free_srtype_matrix()\n");
    exit(1);
  }
  /* imitate the behavior of free() */
  if(m == NULL) return;
  for(i=nrh;i>=nrl;i--) free((void*) (m[i]+ncl));
  free((void*) (m+nrl));
}/* end free_srtype_matrix(.....) */

/* re-allocate memory for computations if necessary */
void realloc_all_sin_cos(const unsigned int nr_monomers,
 const unsigned int nr_chains,
 const unsigned int max_qindex,
 double **** sines_p, double **** cosines_p,
 double **** sines_cm_p, double **** cosines_cm_p){
  static unsigned int max_qindex_old = 0;
  if(max_qindex == 0){
    printf("ERROR (realloc_all_sin_cos): Called with max_qindex=0.\n");
    exit(1);
  }
  if(max_qindex > max_qindex_old){
    printf("MESSAGE (realloc_all_sin_cos): Reallocating sines and cosines.\n");
    printf("MESSAGE                         max_qindex=%u,max_qindex_old=%u\n",\
 max_qindex, max_qindex_old);
    fflush(stdout);
    /* if a pointer is NULL, free() does nothing, so this function also works
     * if the tensors were not allocated previously.
     * due to the memory layout of the tensors, a simple realloc() won't do */
    free_d_rank3tensor(*sines_p, 0, max_qindex_old, 0, \
       nr_monomers-1, 0, DIM -1);
    free_d_rank3tensor(*cosines_p, 0, max_qindex_old, 0, \
       nr_monomers-1, 0, DIM -1);
    free_d_rank3tensor(*sines_cm_p, 0, max_qindex_old, 0, \
       nr_chains-1, 0, DIM -1);
    free_d_rank3tensor(*cosines_cm_p, 0, max_qindex_old, 0, \
       nr_chains-1, 0, DIM -1);
    *sines_p = d_rank3tensor(0, max_qindex, 0, nr_monomers -1, 0, DIM -1);
    *cosines_p = d_rank3tensor(0, max_qindex, 0, nr_monomers -1, 0, DIM -1);
    *sines_cm_p = d_rank3tensor(0, max_qindex, 0, nr_chains-1, 0, DIM -1);
    *cosines_cm_p = d_rank3tensor(0, max_qindex, 0, nr_chains-1, 0, DIM -1);
    max_qindex_old = max_qindex;
  }
}/* realloc_all_sin_cos() */

/* allocate helper arrays in case we allocate the tensors and arrays separately
 */
void alloc_all_sin_cos_arr(const unsigned int nr_monomers,
   const unsigned int nr_chains){
  sin_q = (double*)calloc(nr_monomers, sizeof(double));
  cos_q = (double*)calloc(nr_monomers, sizeof(double));
  sin_q_cm = (double*)calloc(nr_chains, sizeof(double));
  cos_q_cm = (double*)calloc(nr_chains, sizeof(double));
  sinqx_sinqy = (double*)calloc(nr_monomers, sizeof(double));
  sinqx_cosqy = (double*)calloc(nr_monomers, sizeof(double));
  cosqx_sinqy = (double*)calloc(nr_monomers, sizeof(double));
  cosqx_cosqy = (double*)calloc(nr_monomers, sizeof(double));
  sinqx_sinqy_cm = (double*)calloc(nr_chains, sizeof(double));
  sinqx_cosqy_cm = (double*)calloc(nr_chains, sizeof(double));
  cosqx_sinqy_cm = (double*)calloc(nr_chains, sizeof(double));
  cosqx_cosqy_cm = (double*)calloc(nr_chains, sizeof(double));
  /*
    sin_q_helper = (double*)calloc(nr_chains, sizeof(double));
    cos_q_helper = (double*)calloc(nr_chains, sizeof(double));
  */
#ifdef THREE_D
  sinqx_sinqy_sinqz = (double*)calloc(nr_monomers, sizeof(double));
  sinqx_sinqy_cosqz = (double*)calloc(nr_monomers, sizeof(double));
  sinqx_cosqy_sinqz = (double*)calloc(nr_monomers, sizeof(double));
  sinqx_cosqy_cosqz = (double*)calloc(nr_monomers, sizeof(double));
  cosqx_sinqy_sinqz = (double*)calloc(nr_monomers, sizeof(double));
  cosqx_sinqy_cosqz = (double*)calloc(nr_monomers, sizeof(double));
  cosqx_cosqy_sinqz = (double*)calloc(nr_monomers, sizeof(double));
  cosqx_cosqy_cosqz = (double*)calloc(nr_monomers, sizeof(double));
  sinqx_sinqy_sinqz_cm = (double*)calloc(nr_chains, sizeof(double));
  sinqx_sinqy_cosqz_cm = (double*)calloc(nr_chains, sizeof(double));
  sinqx_cosqy_sinqz_cm = (double*)calloc(nr_chains, sizeof(double));
  sinqx_cosqy_cosqz_cm = (double*)calloc(nr_chains, sizeof(double));
  cosqx_sinqy_sinqz_cm = (double*)calloc(nr_chains, sizeof(double));
  cosqx_sinqy_cosqz_cm = (double*)calloc(nr_chains, sizeof(double));
  cosqx_cosqy_sinqz_cm = (double*)calloc(nr_chains, sizeof(double));
  cosqx_cosqy_cosqz_cm = (double*)calloc(nr_chains, sizeof(double));
#endif
} /*  alloc_all_sin_cos_arr() */

/* allocate memory for computations */
void allocate_all_sin_cos(const unsigned int nr_monomers,
  const unsigned int nr_chains,
  const unsigned int max_qindex,
  double **** sines_p, double **** cosines_p,
  double **** sines_cm_p, double **** cosines_cm_p){
  if(max_qindex == 0){
    printf("ERROR (allocate_all_sin_cos): Called with max_qindex=0.\n");
    exit(1);
  }
  *sines_p = d_rank3tensor(0, max_qindex, 0, nr_monomers -1, 0, DIM -1);
  *cosines_p = d_rank3tensor(0, max_qindex, 0, nr_monomers -1, 0, DIM -1);
  *sines_cm_p = d_rank3tensor(0, max_qindex, 0, nr_chains-1, 0, DIM -1);
  *cosines_cm_p = d_rank3tensor(0, max_qindex, 0, nr_chains-1, 0, DIM -1);
  sin_q = (double*)calloc(nr_monomers, sizeof(double));
  cos_q = (double*)calloc(nr_monomers, sizeof(double));
  sin_q_cm = (double*)calloc(nr_chains, sizeof(double));
  cos_q_cm = (double*)calloc(nr_chains, sizeof(double));
  sinqx_sinqy = (double*)calloc(nr_monomers, sizeof(double));
  sinqx_cosqy = (double*)calloc(nr_monomers, sizeof(double));
  cosqx_sinqy = (double*)calloc(nr_monomers, sizeof(double));
  cosqx_cosqy = (double*)calloc(nr_monomers, sizeof(double));
  sinqx_sinqy_cm = (double*)calloc(nr_chains, sizeof(double));
  sinqx_cosqy_cm = (double*)calloc(nr_chains, sizeof(double));
  cosqx_sinqy_cm = (double*)calloc(nr_chains, sizeof(double));
  cosqx_cosqy_cm = (double*)calloc(nr_chains, sizeof(double));
  /*
    sin_q_helper = (double*)calloc(nr_chains, sizeof(double));
    cos_q_helper = (double*)calloc(nr_chains, sizeof(double));
  */
#ifdef THREE_D
  sinqx_sinqy_sinqz = (double*)calloc(nr_monomers, sizeof(double));
  sinqx_sinqy_cosqz = (double*)calloc(nr_monomers, sizeof(double));
  sinqx_cosqy_sinqz = (double*)calloc(nr_monomers, sizeof(double));
  sinqx_cosqy_cosqz = (double*)calloc(nr_monomers, sizeof(double));
  cosqx_sinqy_sinqz = (double*)calloc(nr_monomers, sizeof(double));
  cosqx_sinqy_cosqz = (double*)calloc(nr_monomers, sizeof(double));
  cosqx_cosqy_sinqz = (double*)calloc(nr_monomers, sizeof(double));
  cosqx_cosqy_cosqz = (double*)calloc(nr_monomers, sizeof(double));
  sinqx_sinqy_sinqz_cm = (double*)calloc(nr_chains, sizeof(double));
  sinqx_sinqy_cosqz_cm = (double*)calloc(nr_chains, sizeof(double));
  sinqx_cosqy_sinqz_cm = (double*)calloc(nr_chains, sizeof(double));
  sinqx_cosqy_cosqz_cm = (double*)calloc(nr_chains, sizeof(double));
  cosqx_sinqy_sinqz_cm = (double*)calloc(nr_chains, sizeof(double));
  cosqx_sinqy_cosqz_cm = (double*)calloc(nr_chains, sizeof(double));
  cosqx_cosqy_sinqz_cm = (double*)calloc(nr_chains, sizeof(double));
  cosqx_cosqy_cosqz_cm = (double*)calloc(nr_chains, sizeof(double));
#endif
} /*  allocate_all_sin_cos() */

void free_all_sin_cos(const unsigned int nr_monomers,
      const unsigned int nr_chains,
      const unsigned int max_qindex,
```

```
      double *** sines, double *** cosines,
      double *** sines_cm, double *** cosines_cm){
  /* if a pointer is NULL, free() does nothing */
  free_d_rank3tensor(sines, 0, max_qindex, 0, nr_monomers-1, 0, DIM -1);
  sines = NULL;
  free_d_rank3tensor(cosines, 0, max_qindex, 0, nr_monomers-1, 0, DIM -1);
  cosines = NULL;
  free_d_rank3tensor(sines_cm, 0, max_qindex, 0, nr_chains-1, 0, DIM -1);
  sines_cm = NULL;
  free_d_rank3tensor(cosines_cm, 0, max_qindex, 0, nr_chains-1, 0, DIM -1);
  cosines_cm = NULL;
  free(sin_q);
  sin_q = NULL;
  free(cos_q);
  cos_q = NULL;
  free(sin_q_cm);
  sin_q_cm = NULL;
  free(cos_q_cm);
  cos_q_cm = NULL;
  free(sinqx_sinqy);
  sinqx_sinqy = NULL;
  free(sinqx_cosqy);
  sinqx_cosqy = NULL;
  free(cosqx_sinqy);
  cosqx_sinqy = NULL;
  free(cosqx_cosqy);
  cosqx_cosqy = NULL;
  free(sinqx_sinqy_cm);
  sinqx_sinqy_cm = NULL;
  free(sinqx_cosqy_cm);
  sinqx_cosqy_cm = NULL;
  free(cosqx_sinqy_cm);
  cosqx_sinqy_cm = NULL;
  free(cosqx_cosqy_cm);
  cosqx_cosqy_cm = NULL;
  /*
    free(sin_q_helper);
    sin_q_helper = NULL;
    free(cos_q_helper);
    cos_q_helper = NULL;
  */
#ifdef THREE_D
  free(sinqx_sinqy_sinqz);
  sinqx_sinqy_sinqz = NULL;
  free(sinqx_sinqy_cosqz);
  sinqx_sinqy_cosqz = NULL;
  free(sinqx_cosqy_sinqz);
  sinqx_cosqy_sinqz = NULL;
  free(sinqx_cosqy_cosqz);
  sinqx_cosqy_cosqz = NULL;
  free(cosqx_sinqy_sinqz);
  cosqx_sinqy_sinqz = NULL;
  free(cosqx_sinqy_cosqz);
  cosqx_sinqy_cosqz = NULL;
  free(cosqx_cosqy_sinqz);
  cosqx_cosqy_sinqz = NULL;
  free(cosqx_cosqy_cosqz);
  cosqx_cosqy_cosqz = NULL;
  free(sinqx_sinqy_sinqz_cm);
  sinqx_sinqy_sinqz_cm = NULL;
  free(sinqx_sinqy_cosqz_cm);
  sinqx_sinqy_cosqz_cm = NULL;
  free(sinqx_cosqy_sinqz_cm);
  sinqx_cosqy_sinqz_cm = NULL;
  free(sinqx_cosqy_cosqz_cm);
  sinqx_cosqy_cosqz_cm = NULL;
  free(cosqx_sinqy_sinqz_cm);
  cosqx_sinqy_sinqz_cm = NULL;
  free(cosqx_sinqy_cosqz_cm);
  cosqx_sinqy_cosqz_cm = NULL;
  free(cosqx_cosqy_sinqz_cm);
  cosqx_cosqy_sinqz_cm = NULL;
  free(cosqx_cosqy_cosqz_cm);
  cosqx_cosqy_cosqz_cm = NULL;
#endif
} /* void free_all_sin_cos() */

/* calculate all sines and cosines of all positions and all q-vectors
 * in the lattice.
 * In principle one could do the same by calculating potencies:
 * (cos(x) + i sin(x))^n = (cos(nx) + i sin(nx)),
 * which is a bit faster but might cause numerical problems due to accumulated
 * round off errors. */
void calc_all_sin_cos(const double * const latticeUnits,
      const unsigned int nr_positions,
      const r3vector * const configuration,
      const int max_qindex[],
      double *** const sines,
      double *** const cosines){
  unsigned int particle;
  int qindex;
  double wavenumber[DIM];
  int iDim;
  /* for qindex=0, we know the values */
  qindex=0;
  /* loop over all particles */
  for(particle=0; particle < nr_positions; particle++){
    for(iDim = 0; iDim < DIM; ++iDim){
      cosines[qindex][particle][iDim] = 1.0;
      sines[qindex][particle][iDim]   = 0.0;
    }
  }
  /* loop over all positive q-indices */
  for(qindex=1; qindex <= max_qindex[0]; qindex++){
    wavenumber[0] = latticeUnits[0] * (double)qindex;
    /* loop over all particles */
    for(particle=0; particle < nr_positions; particle++){
      /* compute sines and cosines */
      cosines[qindex][particle][0] \
= cos(configuration[particle].x * wavenumber[0]);
      sines[qindex][particle][0] \
= sin(configuration[particle].x * wavenumber[0]);
    }
  }
  for(qindex=1; qindex <= max_qindex[1]; qindex++){
    wavenumber[1] = latticeUnits[1] * (double)qindex;
    /* loop over all particles */
    for(particle=0; particle < nr_positions; particle++){
      /* compute sines and cosines */
      cosines[qindex][particle][1] \
= cos(configuration[particle].y * wavenumber[1]);
      sines[qindex][particle][1] \
= sin(configuration[particle].y * wavenumber[1]);
    }
  }
#ifdef THREE_D
  for(qindex=1; qindex <= max_qindex[2]; qindex++){
    wavenumber[2] = latticeUnits[2] * (double)qindex;
    /* loop over all particles */
    for(particle=0; particle < nr_positions; particle++){
      /* compute sines and cosines */
      cosines[qindex][particle][2] \
= cos(configuration[particle].z * wavenumber[2]);
      sines[qindex][particle][2] \
= sin(configuration[particle].z * wavenumber[2]);
    }
  }
#endif
}/* calc_all_sin_cos() */

/* given a triplet of  q-vector indices (integer) and a particle index,
 * return cos(\vec{q}\cdot\vec{r}_i).
 * It is more efficient to loop over the q-lattice and treat the signs in the
 * loop, but this routine allows to look at a specific set of q-vectors in any
 * order. */
double cos_qr(const int * const qvi,
      const unsigned int part,
      double *** const sines,
      double *** const cosines){
  /* the signs of the q-indices */
  double sigx, sigy, sigz;
  /* the unsigned indices of the q-vector */
  unsigned int qx, qy, qz;
  /* not much is saved when we treat cases where at least one q-index is 0
   * differently, because these vectors have measure 0 in q-space. */
  /* figure out the signs of the indices for using antisymmetry of the sine
   * function, assign absolute indices */
  if (qvi[0] < 0){
    sigx = -1.0;
    qx   = -qvi[0];
  }else{
    sigx = 1.0;
    qx   = qvi[0];
  }
  if (qvi[1] < 0){
    sigy = -1.0;
    qy   = -qvi[1];
  }else{
    sigy = 1.0;
    qy   = qvi[1];
  }
  if (qvi[2] < 0){
    sigz = -1.0;
    qz   = -qvi[2];
  }else{
    sigz = 1.0;
    qz   = qvi[2];
  }
  return \
    cosines[qx][part][0]    * cosines[qy][part][1]   * cosines[qz][part][2]
    -sigx*sines[qx][part][0] * sigy*sines[qy][part][1]*cosines[qz][part][2]
    -sigx*sines[qx][part][0] * cosines[qy][part][1]   *sigz*sines[qz][part][2]
    -cosines[qx][part][0]    * sigy*sines[qy][part][1]*sigz*sines[qz][part][2];
} /* cos_qr() */

/* the same as cos_qr for the sine */
double sin_qr(const int * const qvi,
      const unsigned int part,
      double *** const sines,
      double *** const cosines){
  /* the signs of the q-indices */
  double sigx, sigy, sigz;
  /* the unsigned indices of the q-vector */
  unsigned int qx, qy, qz;
  if (qvi[0] < 0){
    sigx = -1.0;
    qx   = -qvi[0];
  }else{
    sigx = 1.0;
    qx   = qvi[0];
  }
  if (qvi[1] < 0){
    sigy = -1.0;
    qy   = -qvi[1];
  }else{
    sigy = 1.0;
    qy   = qvi[1];
  }
  if (qvi[2] < 0){
    sigz = -1.0;
    qz   = -qvi[2];
  }else{
    sigz = 1.0;
    qz   = qvi[2];
  }
  return \
    -sigx*sines[qx][part][0] * sigy*sines[qy][part][1]*sigz*sines[qz][part][2]
    +sigx*sines[qx][part][0] * cosines[qy][part][1]   * cosines[qz][part][2]
    +cosines[qx][part][0]    * sigy*sines[qy][part][1]* cosines[qz][part][2]
    +cosines[qx][part][0]    * cosines[qy][part][1]   *sigz*sines[qz][part][2];
} /* sin_qr() */

/* get the number and average values of q-vectors in each bin */
void
get_qvecs_in_bins(const int maxIndex[],
  const double * const latticeUnits,
  const double binWidth,
  const unsigned int nrBins,
  unsigned int * const nr_qvecs_in_bin,
  double * const qvalues_in_bin){
  /* q-lattice indices */
  int xIndex, yIndex, zIndex;
  int zIndexLow, zIndexHigh;
  double rec_length_sq;
  unsigned int binIndex;
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Entering get_qvecs_in_bins(), file %s, line %u\n", \
 HERE);
#endif
  /* initialize */
  for(binIndex = 0; binIndex < nrBins; binIndex++){
    nr_qvecs_in_bin[binIndex] = 0;
    qvalues_in_bin[binIndex] = 0.0;
  }
  /* loop over half the lattice */
  for(xIndex=0; xIndex <= maxIndex[0]; xIndex++){
    for(yIndex = (xIndex == 0) ? 0 : -maxIndex[1]; \
yIndex <= maxIndex[1]; yIndex++){
#ifdef THREE_D
      if(yIndex == 0 && xIndex == 0){
zIndexLow = 0;
      }else{
zIndexLow = -maxIndex[2];
      }
      zIndexHigh = maxIndex[2];
#else
      zIndexLow = zIndexHigh = 0;
```

```
#endif
      /* this loop is at zIndex = 0 for DIM==2. */
      for(zIndex = zIndexLow; zIndex <= zIndexHigh; zIndex++){
/* reciprocal length of this vector */
#ifdef THREE_D
rec_length_sq = SQUARE(latticeUnits[0]) \
  * (double)(xIndex*xIndex) \
  + SQUARE(latticeUnits[1]) \
  * (double)(yIndex*yIndex) \
  + SQUARE(latticeUnits[2]) \
  * (double)(zIndex*zIndex);
#else
rec_length_sq = SQUARE(latticeUnits[0]) \
  * (double)(xIndex*xIndex) \
  + SQUARE(latticeUnits[1]) \
  * (double)(yIndex*yIndex);
#endif
binIndex = (unsigned int)(sqrt(rec_length_sq)/binWidth);
if(binIndex < nrBins){
  nr_qvecs_in_bin[binIndex] += 1;
  qvalues_in_bin[binIndex] += sqrt(rec_length_sq);
}
      } /* loop over zIndex */
    } /* loop over yIndex */
  } /* loop over xIndex */
  /* get average value for q-vector length in each bin */
  for(binIndex = 0; binIndex < nrBins; ++binIndex)
    if(nr_qvecs_in_bin[binIndex] > 0)
      qvalues_in_bin[binIndex] /= (double)nr_qvecs_in_bin[binIndex];
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Leaving get_qvecs_in_bins(), file %s, line %u\n", \
 HERE);
#endif
} /* get_qvecs_in_bins() */

/* get the number and average values of q-vectors in each bin.
 * This is a versionfor a 2-d plane */
void
get_qvecs_in_xy_bins(const int maxIndex[],
     const double * const latticeUnits,
     const double binWidth_x,
     const double binWidth_y,
     const unsigned int nrBins_x,
     const unsigned int nrBins_y,
     unsigned int * const nr_qvecs_in_xy_bin,
     double ** const qvalues_in_xy_bin){
  /* q-lattice indices */
  int xIndex, yIndex, zIndex;
  int zIndexLow, zIndexHigh;
  unsigned int nrBins, binIndex, binIndexXY[2];
  double mod_q[2];
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Entering get_qvecs_in_xy_bins(), file %s, line %u\n", \
HERE);
#endif
  /* here, nrBins is the total number of bins, not just the number of bins in
   * one direction */
  nrBins = nrBins_x * nrBins_y;
  /* initialize */
  for(binIndex = 0; binIndex < nrBins; ++binIndex){
    qvalues_in_xy_bin[binIndex][0] = 0.0;
    qvalues_in_xy_bin[binIndex][1] = 0.0;
    nr_qvecs_in_xy_bin[binIndex] = 0;
  }
  /* loop over half the lattice */
  for(xIndex=0; xIndex <= maxIndex[0]; xIndex++){
    for(yIndex = (xIndex == 0) ? 0 : -maxIndex[1]; \
yIndex <= maxIndex[1]; yIndex++){
#ifdef THREE_D
      if(yIndex == 0 && xIndex == 0){
zIndexLow = 0;
      }else{
zIndexLow = -maxIndex[2];
      }
      zIndexHigh = maxIndex[2];
#else
      zIndexLow = zIndexHigh = 0;
#endif
      /* this loop is at zIndex = 0 for DIM==2. */
      for(zIndex = zIndexLow; zIndex <= zIndexHigh; zIndex++){
/* get bin-indices in x and y */
/* xIndex is always >= 0 */
mod_q[0] = latticeUnits[0] * (double)xIndex;
mod_q[1] = latticeUnits[1] * abs((double)yIndex);
binIndexXY[0] = (unsigned int)(mod_q[0]/binWidth_x);
binIndexXY[1] = (unsigned int)(mod_q[1]/binWidth_y);
if(binIndexXY[0] < nrBins_x && binIndexXY[1] < nrBins_y ){
  binIndex = binIndexXY[0] * nrBins_x + binIndexXY[1];
  nr_qvecs_in_xy_bin[binIndex] += 1;
  qvalues_in_xy_bin[binIndex][0] += mod_q[0];
  qvalues_in_xy_bin[binIndex][1] += mod_q[1];
}
      } /* loop over zIndex */
    } /* loop over yIndex */
  } /* loop over xIndex */
  /* get average value for q-vector length in each bin */
  for(binIndex = 0; binIndex < nrBins; ++binIndex){
    if(nr_qvecs_in_xy_bin[binIndex] > 0){
      qvalues_in_xy_bin[binIndex][0] /= (double)nr_qvecs_in_xy_bin[binIndex];
      qvalues_in_xy_bin[binIndex][1] /= (double)nr_qvecs_in_xy_bin[binIndex];
    }
    /*
  printf("nr=%u, qx=%f, qy=%f\n", nr_qvecs_in_xy_bin[binIndex],\
  qvalues_in_xy_bin[binIndex][0], qvalues_in_xy_bin[binIndex][1]);
    */
    /* check if values are ok */
    if(nr_qvecs_in_xy_bin[binIndex] > 0){
      if(qvalues_in_xy_bin[binIndex][0] \
 < (double)(binIndex/nrBins_x) * binWidth_x \
 || qvalues_in_xy_bin[binIndex][0] \
 > (double)(binIndex/nrBins_x +1)*binWidth_x){
fprintf(stderr, "ERROR get_qvecs_in_xy_bins(): qvalues_in_xy_bin[%u]\
[0] not in range\n", binIndex);
fprintf(stderr, "      Tested: %f <= %f <= %f\n", \
(double)(binIndex/nrBins_x) * binWidth_x, \
qvalues_in_xy_bin[binIndex][0], \
(double)(binIndex/nrBins_x +1)*binWidth_x);
exit(3);
      }
      if(qvalues_in_xy_bin[binIndex][1] \
 <(double)(binIndex%nrBins_y)*binWidth_y \
 || qvalues_in_xy_bin[binIndex][1] \
 > (double)(binIndex%nrBins_y +1)*binWidth_y){
fprintf(stderr, "ERROR get_qvecs_in_xy_bins(): qvalues_in_xy_bin[%u]\
[1] not in range\n", binIndex);
fprintf(stderr, "      Tested: %f <= %f <= %f\n", \
(double)(binIndex%nrBins_y) * binWidth_y, \
qvalues_in_xy_bin[binIndex][1], \
(double)(binIndex%nrBins_y +1)*binWidth_y);
exit(3);
      }
    }/* (nr_qvecs_in_xy_bin[binIndex] > 0) */
  }
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Leaving get_qvecs_in_xy_bins(), file %s, line %u\n",\
 HERE);
#endif
} /* get_qvecs_in_xy_bins() */

/* finds out to which bins a configuration contributes */
void get_confs_in_bins(const unsigned int nrBins,
       const unsigned int * const nr_qvecs_in_bins,
       unsigned int * const nr_confs_in_bins){
  unsigned int iLoop;
  /*  nr_confs_in_bins is used as a counter which is incremented for each
   * configuration falling into it
   */
  for(iLoop = 0; iLoop < nrBins; iLoop++)
    if(nr_qvecs_in_bins[iLoop] > 0)
      nr_confs_in_bins[iLoop]++;
}/* get_confs_in_bins() */

#ifdef THREE_D
/* take the single dimension sines and cosines.  Calculate the full 3-d
 * sin(\vec{q} \in \vec{r}) and cos(\vec{q} \in \vec{r}) values and sum these
 * up to the self and distinct structure factors (done by other functions).
 */
void
calc_self_and_distinct_sf_3d(double *** const sines,
     double *** const cosines,
     double *** const sines_cm,
     double *** const cosines_cm,
     const unsigned int mon_per_chain,
     const unsigned int nr_positions,
     const unsigned int nr_positions_cm,
     const int max_qindex[],
     const double * const latticeUnits,
     const unsigned int nrBins,
     const double binWidth,
     const unsigned int sf_data_start_index,
     STRUF_RES_TYPE ** const results,
     const unsigned int sf_data_cm_start_index,
     STRUF_RES_TYPE ** const results_cm){
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Entering calc_self_and_distinct_sf_3d(), file %s, \
line %u\n", HERE);
#endif
  /* compute separately for the cases where q=0 in one or more dimensions */
  low_d_calc_all_sf_3d(sines, cosines, sines_cm, cosines_cm,
       mon_per_chain, nr_positions,
       nr_positions_cm,
       max_qindex, latticeUnits,
       nrBins, binWidth,
       sf_data_start_index,
       results,
       sf_data_cm_start_index,
       results_cm,
       0, 0, 0);
  low_d_calc_all_sf_3d(sines, cosines, sines_cm, cosines_cm,
       mon_per_chain, nr_positions,
       nr_positions_cm,
       max_qindex, latticeUnits,
       nrBins, binWidth,
       sf_data_start_index,
       results,
       sf_data_cm_start_index,
       results_cm,
       0, 0, 1);
  low_d_calc_all_sf_3d(sines, cosines, sines_cm, cosines_cm,
       mon_per_chain, nr_positions,
       nr_positions_cm,
       max_qindex, latticeUnits,
       nrBins, binWidth,
         sf_data_start_index,
       results,
       sf_data_cm_start_index,
       results_cm,
       0, 1, 0);
  low_d_calc_all_sf_3d(sines, cosines, sines_cm, cosines_cm,
       mon_per_chain, nr_positions,
       nr_positions_cm,
       max_qindex, latticeUnits,
       nrBins, binWidth,
       sf_data_start_index,
       results,
       sf_data_cm_start_index,
       results_cm,
       0, 1, 1);
  low_d_calc_all_sf_3d(sines, cosines, sines_cm, cosines_cm,
       mon_per_chain, nr_positions,
       nr_positions_cm,
       max_qindex, latticeUnits,
       nrBins, binWidth,
       sf_data_start_index,
       results,
       sf_data_cm_start_index,
       results_cm,
       1, 0, 0);
  low_d_calc_all_sf_3d(sines, cosines, sines_cm, cosines_cm,
       mon_per_chain, nr_positions,
       nr_positions_cm,
       max_qindex, latticeUnits,
       nrBins, binWidth,
       sf_data_start_index,
       results,
       sf_data_cm_start_index,
       results_cm,
       1, 0, 1);
  low_d_calc_all_sf_3d(sines, cosines, sines_cm, cosines_cm,
       mon_per_chain, nr_positions,
       nr_positions_cm,
       max_qindex, latticeUnits,
       nrBins, binWidth,
       sf_data_start_index,
       results,
       sf_data_cm_start_index,
       results_cm,
       1, 1, 0);
  low_d_calc_all_sf_3d(sines, cosines, sines_cm, cosines_cm,
       mon_per_chain, nr_positions,
       nr_positions_cm,
       max_qindex, latticeUnits,
       nrBins, binWidth,
       sf_data_start_index,
       results,
       sf_data_cm_start_index,
       results_cm,
       1, 1, 1);
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Leaving calc_self_and_distinct_sf_3d(), file %s, line \
%u\n", HERE);
#endif
} /* calc_self_and_distinct_sf_3d() */

/* This function has flags to decide in which dimensions the q-vectors are 0.
 * If q-indices are zero fewer equivalent images of the positive octant
```

```
 * exist.
 */
void
low_d_calc_all_sf_3d(double *** const sines,
     double *** const cosines,
     double *** const sines_cm,
     double *** const cosines_cm,
     const unsigned int mon_per_chain,
     const unsigned int nr_positions,
     const unsigned int nr_positions_cm,
     const int max_qindex[],
     const double * const latticeUnits,
     const unsigned int nrBins,
     const double binWidth,
     const unsigned int sf_data_start_index,
     STRUF_RES_TYPE ** const results,
     const unsigned int sf_data_cm_start_index,
     STRUF_RES_TYPE ** const results_cm,
     const unsigned int qx_flag,
     const unsigned int qy_flag,
     const unsigned int qz_flag){
  /* q-lattice indices */
  unsigned int qi_x, qi_y, qi_z;
  /* particle */
  unsigned int part, part_cm;
  /* squared reciprocal vector lengths */
  double q_x_sq, q_y_sq, q_x_sq_plus_q_y_sq, mod_q_sq;
  unsigned int binIndex;
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Entering low_d_calc_all_sf_3d(%u,%u,%u), file %s, line \
%u\n", qx_flag, qy_flag, qz_flag, HERE);
#endif
#ifdef CHECK_DIMENSION_FLAGS_ON
  /* check validity of flags */
  if(!(qx_flag==0 || qx_flag==1)){
    fprintf(stderr, "ERROR: low_d_calc_all_sf_3d(), file %s, \
line %i\n", HERE);
    fprintf(stderr, "       qx_flag=%u not valid\n", qx_flag);
    exit(2);
  }
  if(!(qy_flag==0 || qy_flag==1)){
    fprintf(stderr, "ERROR: low_d_calc_all_sf_3d(), file %s, \
line %i\n", HERE);
    fprintf(stderr, "       qy_flag=%u not valid\n", qy_flag);
    exit(2);
  }
  if(!(qz_flag==0 || qz_flag==1)){
    fprintf(stderr, "ERROR: low_d_calc_all_sf_3d(), file %s, \
line %i\n", HERE);
    fprintf(stderr, "       qz_flag=%u not valid\n", qz_flag);
    exit(2);
  }
#endif
  /* loop over the selected range */
  for(qi_x = qx_flag; qi_x <= max_qindex[0] * qx_flag; qi_x++){
    q_x_sq = SQUARE((double)qi_x * latticeUnits[0]);
    for(qi_y = qy_flag; qi_y <= max_qindex[1] * qy_flag; qi_y++){
      q_y_sq = SQUARE((double)qi_y * latticeUnits[1]);
      q_x_sq_plus_q_y_sq = q_x_sq + q_y_sq;
      /* calculate and save sin and cosine products of x and y contributions
       * for all particles */
      for(part = 0; part < nr_positions; part++){
sinqx_sinqy[part] \
  = sines[qi_x][part][0]   * sines[qi_y][part][1];
sinqx_cosqy[part] \
  = sines[qi_x][part][0]   * cosines[qi_y][part][1];
cosqx_sinqy[part] \
  = cosines[qi_x][part][0] * sines[qi_y][part][1];
cosqx_cosqy[part] \
  = cosines[qi_x][part][0] * cosines[qi_y][part][1];
      }
      for(part_cm = 0; part_cm < nr_positions_cm; part_cm++){
sinqx_sinqy_cm[part_cm] \
  = sines_cm[qi_x][part_cm][0]   * sines_cm[qi_y][part_cm][1];
sinqx_cosqy_cm[part_cm] \
  = sines_cm[qi_x][part_cm][0]   * cosines_cm[qi_y][part_cm][1];
cosqx_sinqy_cm[part_cm] \
  = cosines_cm[qi_x][part_cm][0] * sines_cm[qi_y][part_cm][1];
cosqx_cosqy_cm[part_cm] \
  = cosines_cm[qi_x][part_cm][0] * cosines_cm[qi_y][part_cm][1];
      }
      for(qi_z = qz_flag; qi_z <= max_qindex[2] * qz_flag; qi_z++){
mod_q_sq = q_x_sq_plus_q_y_sq + SQUARE((double)qi_z * latticeUnits[2]);
binIndex = (unsigned int)(sqrt(mod_q_sq)/binWidth);
/* consider only q-vectors within binning range */
if(binIndex < nrBins){
#ifdef DEBUG_CALC_SELF_AND_DISTINCT_SF_3D_ON
  printf("DEBUG: At q-vector (%i, %i, %i), q^2=%f, binIndex=%u\n", \
 qi_x, qi_y, qi_z, mod_q_sq, binIndex);
#endif
  /* calculate all triple sine and cosine products */
  for(part=0; part < nr_positions; part++){
    sinqx_sinqy_sinqz[part]=sinqx_sinqy[part]*sines[qi_z][part][2];
    sinqx_sinqy_cosqz[part]=sinqx_sinqy[part]*cosines[qi_z][part][2];
    sinqx_cosqy_sinqz[part]=sinqx_cosqy[part]*sines[qi_z][part][2];
    sinqx_cosqy_cosqz[part]=sinqx_cosqy[part]*cosines[qi_z][part][2];
    cosqx_sinqy_sinqz[part]=cosqx_sinqy[part]*sines[qi_z][part][2];
    cosqx_sinqy_cosqz[part]=cosqx_sinqy[part]*cosines[qi_z][part][2];
    cosqx_cosqy_sinqz[part]=cosqx_cosqy[part]*sines[qi_z][part][2];
    cosqx_cosqy_cosqz[part]=cosqx_cosqy[part]*cosines[qi_z][part][2];
  }
  for(part_cm = 0; part_cm < nr_positions_cm; part_cm++){
    sinqx_sinqy_sinqz_cm[part_cm] = \
      sinqx_sinqy_cm[part_cm]*sines_cm[qi_z][part_cm][2];
    sinqx_sinqy_cosqz_cm[part_cm] = \
      sinqx_sinqy_cm[part_cm]*cosines_cm[qi_z][part_cm][2];
    sinqx_cosqy_sinqz_cm[part_cm] = \
      sinqx_cosqy_cm[part_cm]*sines_cm[qi_z][part_cm][2];
    sinqx_cosqy_cosqz_cm[part_cm] = \
      sinqx_cosqy_cm[part_cm]*cosines_cm[qi_z][part_cm][2];
    cosqx_sinqy_sinqz_cm[part_cm] = \
      cosqx_sinqy_cm[part_cm]*sines_cm[qi_z][part_cm][2];
    cosqx_sinqy_cosqz_cm[part_cm] = \
      cosqx_sinqy_cm[part_cm]*cosines_cm[qi_z][part_cm][2];
    cosqx_cosqy_sinqz_cm[part_cm] = \
      cosqx_cosqy_cm[part_cm]*sines_cm[qi_z][part_cm][2];
    cosqx_cosqy_cosqz_cm[part_cm] =
      cosqx_cosqy_cm[part_cm]*cosines_cm[qi_z][part_cm][2];
  }
  /* the positive (borders included) (y,z) quadrant must always be
   * computed */
  /* qy >= 0 , qz >= 0 */
  for(part=0; part < nr_positions; part++){
    sin_q[part] = \
      - sinqx_sinqy_sinqz[part] \
      + sinqx_cosqy_cosqz[part] \
      + cosqx_sinqy_cosqz[part] \
      + cosqx_cosqy_sinqz[part];
    cos_q[part] = \
      + cosqx_cosqy_cosqz[part] \
      - sinqx_sinqy_cosqz[part] \
      - sinqx_cosqy_sinqz[part] \
      - cosqx_sinqy_sinqz[part];
  }
  for(part_cm=0; part_cm < nr_positions_cm; part_cm++){
    sin_q_cm[part_cm] = \
      - sinqx_sinqy_sinqz_cm[part_cm] \
      + sinqx_cosqy_cosqz_cm[part_cm] \
      + cosqx_sinqy_cosqz_cm[part_cm] \
      + cosqx_cosqy_sinqz_cm[part_cm];
    cos_q_cm[part_cm] = \
      + cosqx_cosqy_cosqz_cm[part_cm] \
      - sinqx_sinqy_cosqz_cm[part_cm] \
      - sinqx_cosqy_sinqz_cm[part_cm] \
      - cosqx_sinqy_sinqz_cm[part_cm];
  }
  assign_self_dist_sf_3d(mon_per_chain, nr_positions_cm,
 sf_data_start_index,
 results[binIndex],
 sf_data_cm_start_index,
 results_cm[binIndex]);
  if(!(qx_flag == 0 && qy_flag == 0) && (qz_flag == 1)){
    /* qy >= 0 , qz < 0 */
    for(part=0; part < nr_positions; part++){
      sin_q[part] = \
+ sinqx_sinqy_sinqz[part] \
+ sinqx_cosqy_cosqz[part] \
+ cosqx_sinqy_cosqz[part] \
- cosqx_cosqy_sinqz[part];
      cos_q[part] = \
+ cosqx_cosqy_cosqz[part] \
- sinqx_sinqy_cosqz[part] \
+ sinqx_cosqy_sinqz[part] \
+ cosqx_sinqy_sinqz[part];
    }
    for(part_cm=0; part_cm < nr_positions_cm; part_cm++){
      sin_q_cm[part_cm] = \
+ sinqx_sinqy_sinqz_cm[part_cm] \
+ sinqx_cosqy_cosqz_cm[part_cm] \
+ cosqx_sinqy_cosqz_cm[part_cm] \
- cosqx_cosqy_sinqz_cm[part_cm];
      cos_q_cm[part_cm] = \
+ cosqx_cosqy_cosqz_cm[part_cm] \
- sinqx_sinqy_cosqz_cm[part_cm] \
+ sinqx_cosqy_sinqz_cm[part_cm] \
+ cosqx_sinqy_sinqz_cm[part_cm];
    }
    assign_self_dist_sf_3d(mon_per_chain, nr_positions_cm,
   sf_data_start_index,
   results[binIndex],
   sf_data_cm_start_index,
   results_cm[binIndex]);
  }/* if (!(qx_flag == 0 && qy_flag == 0) && (qz_flaq == 1)) */
  if(qx_flag == 1 && qy_flag == 1){
    /* qy < 0 , qz >= 0 */
    for(part=0; part < nr_positions; part++){
      sin_q[part] = \
+ sinqx_sinqy_sinqz[part] \
+ sinqx_cosqy_cosqz[part] \
- cosqx_sinqy_cosqz[part] \
+ cosqx_cosqy_sinqz[part];
      cos_q[part] = \
+ cosqx_cosqy_cosqz[part] \
+ sinqx_sinqy_cosqz[part] \
- sinqx_cosqy_sinqz[part] \
+ cosqx_sinqy_sinqz[part];
    }
    for(part_cm=0; part_cm < nr_positions_cm; part_cm++){
      sin_q_cm[part_cm] = \
+ sinqx_sinqy_sinqz_cm[part_cm] \
+ sinqx_cosqy_cosqz_cm[part_cm] \
- cosqx_sinqy_cosqz_cm[part_cm] \
+ cosqx_cosqy_sinqz_cm[part_cm];
      cos_q_cm[part_cm] = \
+ cosqx_cosqy_cosqz_cm[part_cm] \
+ sinqx_sinqy_cosqz_cm[part_cm] \
- sinqx_cosqy_sinqz_cm[part_cm] \
+ cosqx_sinqy_sinqz_cm[part_cm];
    }
    assign_self_dist_sf_3d(mon_per_chain, nr_positions_cm,
   sf_data_start_index,
   results[binIndex],
   sf_data_cm_start_index,
   results_cm[binIndex]);
  }/* if(qx_flag == 1 && qy_flag == 1) */
  if(qx_flag == 1 && qy_flag == 1 && qz_flag == 1){
    /* qy < 0 , qz < 0 */
    for(part=0; part < nr_positions; part++){
      sin_q[part] = \
- sinqx_sinqy_sinqz[part] \
+ sinqx_cosqy_cosqz[part] \
- cosqx_sinqy_cosqz[part] \
- cosqx_cosqy_sinqz[part];
      cos_q[part] = \
+ cosqx_cosqy_cosqz[part] \
+ sinqx_sinqy_cosqz[part] \
+ sinqx_cosqy_sinqz[part] \
- cosqx_sinqy_sinqz[part];
    }
    for(part_cm=0; part_cm < nr_positions_cm; part_cm++){
      sin_q_cm[part_cm] = \
- sinqx_sinqy_sinqz_cm[part_cm] \
+ sinqx_cosqy_cosqz_cm[part_cm] \
- cosqx_sinqy_cosqz_cm[part_cm] \
- cosqx_cosqy_sinqz_cm[part_cm];
      cos_q_cm[part_cm] = \
+ cosqx_cosqy_cosqz_cm[part_cm] \
+ sinqx_sinqy_cosqz_cm[part_cm] \
+ sinqx_cosqy_sinqz_cm[part_cm] \
- cosqx_sinqy_sinqz_cm[part_cm];
    }
    assign_self_dist_sf_3d(mon_per_chain, nr_positions_cm,
   sf_data_start_index,
   results[binIndex],
   sf_data_cm_start_index,
   results_cm[binIndex]);
  }/* if(qx_flag == 1 && qy_flag == 1 && qz_flag == 1) */
}/* if |q| in allowed range */
      }/* loop over qi_z */
    }/* loop over qi_y */
  }/* loop over qi_x */
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Leaving low_d_calc_all_sf_3d(), file %s, line %u\n",
```

```
 HERE);
#endif
} /* low_d_calc_all_sf_3d() */

/* given sin(\vec{q} \in \vec{r_i}) and cos(\vec{q} \in \vec{r_i}) for all
 * particle and center of mass positions
 * calculate the self and distinct structure factors.
 * This is were the CPU time goes!
 */
void
assign_self_dist_sf_3d(const unsigned int mon_per_chain,
       const unsigned int nr_chains,
       const unsigned int sf_data_start_index,
       STRUF_RES_TYPE * const bin_results,
       const unsigned int sf_data_cm_start_index,
       STRUF_RES_TYPE * const bin_results_cm){
  /* loop indices */
  unsigned int chain, first_mon_in_chain;
  /* sum variables */
  double sum_cos, sum_sin, sum_sin_cm, sum_cos_cm;
#ifdef CALC_ONLY_COHERENT_SF_YES
  unsigned int part_index;
#else
  unsigned int second_mon_in_chain, first_help_index, second_help_index;
  unsigned int result_column;
  double sum_cos_first_mon, sum_sin_first_mon, \
    sum_cos_second_mon, sum_sin_second_mon;
#endif
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Entering assign_self_dist_sf_3d(), file %s, line %u\n",
 HERE);
#endif
  /* it is important for efficiency to avoid two-fold sums! */
  /* structure factor of the chain centers of mass */
  sum_sin_cm = 0.0;
  sum_cos_cm = 0.0;
  for(chain = 0; chain < nr_chains; ++chain){
    sum_sin_cm += sin_q_cm[chain];
    sum_cos_cm += cos_q_cm[chain];
  }
  bin_results_cm[sf_data_cm_start_index] \
    += SQUARE(sum_sin_cm) + SQUARE(sum_cos_cm);
#ifndef CALC_ONLY_COHERENT_SF_YES
  /* structure factor of the monomers in the chain, self contribution */
  for(first_mon_in_chain = 0; \
      first_mon_in_chain < mon_per_chain; first_mon_in_chain++){
    for(chain = 0, first_help_index = first_mon_in_chain; \
chain < nr_chains; chain++, first_help_index += mon_per_chain){
      bin_results_cm[sf_data_cm_start_index + 1 + first_mon_in_chain] \
+= sin_q[first_help_index] * sin_q_cm[chain] \
+  cos_q[first_help_index] * cos_q_cm[chain];
    }/* end loop over chains */
  }/* end loop over monomers in chain */
  /* structure factor of the monomers in the chain, distinct contribution */
  for(first_mon_in_chain = 0; \
      first_mon_in_chain < mon_per_chain; first_mon_in_chain++){
    sum_sin = 0.0;
    sum_cos = 0.0;
    for(chain = 0, first_help_index = first_mon_in_chain; \
chain < nr_chains; \
chain++, first_help_index += mon_per_chain){
      sum_sin += sin_q[first_help_index];
      sum_cos += cos_q[first_help_index];
    }/* end loop over first chain index */
    bin_results_cm[sf_data_cm_start_index + 1 + mon_per_chain \
 + first_mon_in_chain] \
      += sum_sin * sum_sin_cm + sum_cos * sum_cos_cm;
  }/* end loop over monomers in chain */
  /* in which column-index the results are written */
  result_column = sf_data_start_index + 2;
  /* structure factor of the monomer pairs in a chain */
  for(first_mon_in_chain = 0; \
      first_mon_in_chain < mon_per_chain; \
      first_mon_in_chain++){
    /* because we skip the second_mon_in_chain == first_mon_in_chain
     * calculation we advance over this column */
    result_column++;
    /* we know the answer for
     * second_mon_in_chain == first_mon_in_chain: 1
     * (but this makes a good test)
     */
    for(second_mon_in_chain = first_mon_in_chain +1; \
second_mon_in_chain < mon_per_chain; \
second_mon_in_chain++){
      for(chain = 0, \
    first_help_index = first_mon_in_chain, \
    second_help_index = second_mon_in_chain; \
  chain < nr_chains; \
  chain++, \
    first_help_index += mon_per_chain, \
    second_help_index += mon_per_chain){
bin_results[result_column]\
  += sin_q[first_help_index] * sin_q[second_help_index] \
  +  cos_q[first_help_index] * cos_q[second_help_index];
      }/* end loop over chains */
      /* go to next column */
      result_column++;
    }/* end loop over second_mon_in_chain */
  }/* end loop over first_mon_in_chain */

  /* result_column has the right value */
  /* structure factor of the monomer pairs in different chains */
  for(first_mon_in_chain = 0; \
      first_mon_in_chain < mon_per_chain; \
      first_mon_in_chain++){
    sum_sin_first_mon = 0.0;
    sum_cos_first_mon = 0.0;
    for(chain = 0, first_help_index = first_mon_in_chain; \
chain < nr_chains; \
chain++, first_help_index += mon_per_chain){
      sum_sin_first_mon += sin_q[first_help_index];
      sum_cos_first_mon += cos_q[first_help_index];
    }
    /* use symmetry in the monomer indices */
    for(second_mon_in_chain = first_mon_in_chain; \
second_mon_in_chain < mon_per_chain; \
second_mon_in_chain++){
      sum_sin_second_mon = 0.0;
      sum_cos_second_mon = 0.0;
      for(chain = 0, second_help_index = second_mon_in_chain; \
  chain < nr_chains; \
  chain++, second_help_index += mon_per_chain){
sum_sin_second_mon += sin_q[second_help_index];
sum_cos_second_mon += cos_q[second_help_index];
      }
      bin_results[result_column] \
+= sum_sin_first_mon * sum_sin_second_mon \
+  sum_cos_first_mon * sum_cos_second_mon;
      /* go to next column */
      result_column++;
    }/* end loop over second_mon_in_chain */
  }/* end loop over first_mon_in_chain */
  /* substract self contribution later */
#else
  /* coherent structure factor of the melt, self contribution */
  for(chain = 0, part_index = 0; chain < nr_chains; chain++){
    sum_sin = 0.0;
    sum_cos = 0.0;
    for(first_mon_in_chain = 0; first_mon_in_chain < mon_per_chain; \
first_mon_in_chain++, part_index++){
      sum_sin += sin_q[part_index];
      sum_cos += cos_q[part_index];
    }
    bin_results[sf_data_start_index] += SQUARE(sum_sin) + SQUARE(sum_cos);
  }
  /* coherent structure factor of the melt (distinct) */
  sum_sin = 0.0;
  sum_cos = 0.0;
  for(part_index = 0; part_index < nr_chains * mon_per_chain; ++part_index){
    sum_sin += sin_q[part_index];
    sum_cos += cos_q[part_index];
  }
  bin_results[sf_data_start_index + 1] += SQUARE(sum_sin) + SQUARE(sum_cos);
  /* substract self contribution later */
#endif
#ifdef DEBUG_ASSIGN_SELF_DIST_SF_3D_ON
#ifndef CALC_ONLY_COHERENT_SF_YES
  printf("DEBUG: values for this bin:\n");
  for(result_column = 0; result_column <= sf_data_start_index + 1 \
    + SQUARE(mon_per_chain) + (mon_per_chain +1); ++result_column)
    printf("%f ", bin_results[result_column]);
  printf("\n");
  printf("DEBUG: values for this cm-bin:\n");
  for(result_column = 0; \
      result_column <= sf_data_start_index + 2  * mon_per_chain; \
      ++result_column)
    printf("%f ", bin_results_cm[result_column]);
  printf("\n");
#else
  printf("DEBUG: values for this bin:\n");
  for(part_index = 0; part_index <= sf_data_start_index+1; ++part_index)
    printf("%f ", bin_results[part_index]);
  printf("\n");
  printf("DEBUG: values for this cm-bin:\n");
  for(part_index = 0; part_index <= sf_data_cm_start_index; ++part_index)
    printf("%f ", bin_results[part_index]);
  printf("\n");
#endif
#endif
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Leaving assign_self_dist_sf_3d(), file %s, line %u\n",
 HERE);
#endif
}/* assign_self_dist_sf_3d() */
#else /* THREE_D */
/* take the single dimension sines and cosines.  Calculate the full 3-d
 * sin(\vec{q} \in \vec{r}) and cos(\vec{q} \in \vec{r}) values and sum these
 * up to the self and distinct structure factors (done by other functions).
 */
void
calc_self_and_distinct_sf_2d(double *** const sines,
     double *** const cosines,
     double *** const sines_cm,
     double *** const cosines_cm,
     const unsigned int mon_per_chain,
     const unsigned int nr_positions,
     const unsigned int nr_positions_cm,
     const int max_qindex[],
     const double * const latticeUnits,
     const boolean l_write_xy_plane,
     const double binWidth,
     const unsigned int nrBins,
     const unsigned int sf_data_start_index,
     STRUF_RES_TYPE ** const results,
     const unsigned int sf_data_cm_start_index,
     STRUF_RES_TYPE ** const results_cm,
     const double binWidth_x,
     const double binWidth_y,
     const unsigned int nrBins_x,
     const unsigned int nrBins_y,
     const unsigned int sf_data_xy_start_index,
     STRUF_RES_TYPE ** const results_xy,
     const unsigned int sf_data_cm_xy_start_index,
     STRUF_RES_TYPE ** const results_cm_xy){

#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Entering calc_self_and_distinct_sf_2d(), file %s, \
line %u\n", HERE);
#endif
  /* compute separately for the cases where q=0 in one or more dimensions */
  low_d_calc_all_sf_2d(sines, cosines, sines_cm, cosines_cm,
       mon_per_chain, nr_positions,
       nr_positions_cm,
       max_qindex, latticeUnits,
       l_write_xy_plane,
       binWidth,
       nrBins,
       sf_data_start_index, results,
       sf_data_cm_start_index, results_cm,
       binWidth_x,
       binWidth_y,
       nrBins_x,
       nrBins_y,
       sf_data_xy_start_index, results_xy,
       sf_data_cm_xy_start_index, results_cm_xy,
       0, 0);
  low_d_calc_all_sf_2d(sines, cosines, sines_cm, cosines_cm,
       mon_per_chain, nr_positions,
       nr_positions_cm,
       max_qindex, latticeUnits,
       l_write_xy_plane,
       binWidth,
       nrBins,
       sf_data_start_index, results,
       sf_data_cm_start_index, results_cm,
       binWidth_x,
       binWidth_y,
       nrBins_x,
       nrBins_y,
       sf_data_xy_start_index, results_xy,
       sf_data_cm_xy_start_index, results_cm_xy,
       0, 1);
  low_d_calc_all_sf_2d(sines, cosines, sines_cm, cosines_cm,
       mon_per_chain, nr_positions,
       nr_positions_cm,
       max_qindex, latticeUnits,
       l_write_xy_plane,
       binWidth,
       nrBins,
       sf_data_start_index, results,
       sf_data_cm_start_index, results_cm,
       binWidth_x,
```

```
       binWidth_y,
       nrBins_x,
       nrBins_y,
       sf_data_xy_start_index, results_xy,
       sf_data_cm_xy_start_index, results_cm_xy,
       1, 0);
  low_d_calc_all_sf_2d(sines, cosines, sines_cm, cosines_cm,
       mon_per_chain, nr_positions,
       nr_positions_cm,
       max_qindex, latticeUnits,
       l_write_xy_plane,
       binWidth,
       nrBins,
       sf_data_start_index, results,
       sf_data_cm_start_index, results_cm,
       binWidth_x,
       binWidth_y,
       nrBins_x,
       nrBins_y,
       sf_data_xy_start_index, results_xy,
       sf_data_cm_xy_start_index, results_cm_xy,
       1, 1);
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Leaving calc_self_and_distinct_sf_2d(), file %s, \
line %u\n", HERE);
#endif
} /* calc_self_and_distinct_sf_2d() */

/* This function has flags to decide in which dimensions the q-vectors are 0.
 * If q-indices are zero fewer equivalent images of the positive quadrant
 * exist.
 */
void
low_d_calc_all_sf_2d(double *** const sines,
     double *** const cosines,
     double *** const sines_cm,
     double *** const cosines_cm,
     const unsigned int mon_per_chain,
     const unsigned int nr_positions,
     const unsigned int nr_positions_cm,
     const int max_qindex[],
     const double * const latticeUnits,
     const boolean l_write_xy_plane,
     const double binWidth,
     const unsigned int nrBins,
     const unsigned int sf_data_start_index,
     STRUF_RES_TYPE ** const results,
     const unsigned int sf_data_cm_start_index,
     STRUF_RES_TYPE ** const results_cm,
     const double binWidth_x,
     const double binWidth_y,
     const unsigned int nrBins_x,
     const unsigned int nrBins_y,
     const unsigned int sf_data_xy_start_index,
     STRUF_RES_TYPE ** const results_xy,
     const unsigned int sf_data_cm_xy_start_index,
     STRUF_RES_TYPE ** const results_cm_xy,
     const unsigned int qx_flag,
     const unsigned int qy_flag){
  /* q-lattice indices */
  unsigned int qi_x = UINT_MAX, qi_y = UINT_MAX;
  /* particle */
  unsigned int part = UINT_MAX, part_cm = UINT_MAX;
  /* squared reciprocal vector lengths */
  double q_x_sq = -1.0, q_y_sq = -1.0, mod_q_sq = -1.0;
  unsigned int binIndex = UINT_MAX;
  unsigned int binIndex_xy = UINT_MAX, binIndexXY[2];
  /* binIndexValid is always computed, binIndexValid_xy not.
   * binIndexValid_xy = FALSE is necessary if l_write_xy_plane == FALSE,
   * because for efficiency, binIndexValid_xy is not touched in this case.
   */
   boolean binIndexValid = TRUE, binIndexValid_xy = FALSE;
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Entering low_d_calc_all_sf_2d(), file %s, line %u\n", \
HERE);
#endif
#ifdef CHECK_DIMENSION_FLAGS_ON
  /* check validity of flags */
  if(!(qx_flag==0 || qx_flag==1)){
    fprintf(stderr, "ERROR: low_d_calc_all_sf_2d(), file %s, \
line %i\n", HERE);
    fprintf(stderr, "       qx_flag=%u not valid\n", qx_flag);
    exit(2);
  }
  if(!(qy_flag==0 || qy_flag==1)){
    fprintf(stderr, "ERROR: low_d_calc_all_sf_2d(), file %s, \
line %i\n", HERE);
    fprintf(stderr, "       qy_flag=%u not valid\n", qy_flag);
    exit(2);
  }
#endif
  /* loop over the selected range */
  for(qi_x = qx_flag; qi_x <= max_qindex[0] * qx_flag; qi_x++){
    if(l_write_xy_plane)
      binIndexXY[0] = ((double)qi_x * latticeUnits[0])/binWidth_x;
    q_x_sq = SQUARE((double)qi_x * latticeUnits[0]);
    for(qi_y = qy_flag; qi_y <= max_qindex[1] * qy_flag; qi_y++){
      if(l_write_xy_plane){
binIndexXY[1] = ((double)qi_y * latticeUnits[1])/binWidth_y;
if(binIndexXY[0] < nrBins_x && binIndexXY[1] < nrBins_y){
  binIndexValid_xy = TRUE;
  binIndex_xy = binIndexXY[0]*nrBins_x + binIndexXY[1];
}else
  binIndexValid_xy = FALSE;
      }
      q_y_sq = SQUARE((double)qi_y * latticeUnits[1]);
      mod_q_sq = q_x_sq + q_y_sq;
      binIndex = (unsigned int)(sqrt(mod_q_sq)/binWidth);
      if(binIndex < nrBins)
binIndexValid = TRUE;
      else
binIndexValid = FALSE;
      /* consider only q-vectors within binning range */
      if(binIndexValid || binIndexValid_xy){
#ifdef DEBUG_CALC_SELF_AND_DISTINCT_SF_2D_ON
printf("DEBUG: At q-vector (%i, %i), q^2=%f, q=%f\n", \
       qi_x, qi_y, mod_q_sq, sqrt(mod_q_sq));
printf("DEBUG: binIndex=%u, binIndex_xy=%u\n", binIndex, binIndex_xy);
printf("DEBUG: binIndexValid = %d, binIndexValid_xy = %d\n", \
       binIndexValid, binIndexValid_xy);
#endif
/* calculate and save sin and cosine products of x and y contributions
 * for all particles */
for(part = 0; part < nr_positions; part++){
  sinqx_sinqy[part] \
    = sines[qi_x][part][0]   * sines[qi_y][part][1];
  sinqx_cosqy[part] \
    = sines[qi_x][part][0]   * cosines[qi_y][part][1];
  cosqx_sinqy[part] \
    = cosines[qi_x][part][0] * sines[qi_y][part][1];
  cosqx_cosqy[part] \
    = cosines[qi_x][part][0] * cosines[qi_y][part][1];
}
for(part_cm = 0; part_cm < nr_positions_cm; part_cm++){
  sinqx_sinqy_cm[part_cm] \
    = sines_cm[qi_x][part_cm][0]   * sines_cm[qi_y][part_cm][1];
  sinqx_cosqy_cm[part_cm] \
    = sines_cm[qi_x][part_cm][0]   * cosines_cm[qi_y][part_cm][1];
  cosqx_sinqy_cm[part_cm] \
    = cosines_cm[qi_x][part_cm][0] * sines_cm[qi_y][part_cm][1];
  cosqx_cosqy_cm[part_cm] \
    = cosines_cm[qi_x][part_cm][0] * cosines_cm[qi_y][part_cm][1];
}
/* calculate sines and cosines */
for(part = 0; part < nr_positions; part++){
  sin_q[part] = cosqx_cosqy[part] - sinqx_sinqy[part];
  cos_q[part] = sinqx_cosqy[part] + cosqx_sinqy[part];
}
for(part_cm = 0; part_cm < nr_positions_cm; part_cm++){
  sin_q_cm[part_cm] = \
    cosqx_cosqy_cm[part_cm] - sinqx_sinqy_cm[part_cm];
  cos_q_cm[part_cm] = \
    sinqx_cosqy_cm[part_cm] + cosqx_sinqy_cm[part_cm];
}
if(l_write_xy_plane)
  assign_self_dist_sf_2d(mon_per_chain, nr_positions_cm,
 binIndexValid,
 sf_data_start_index, results[binIndex],
 sf_data_cm_start_index, results_cm[binIndex],
 binIndexValid_xy,
 sf_data_xy_start_index,
 results_xy[binIndex_xy],
 sf_data_cm_xy_start_index,
 results_cm_xy[binIndex_xy]);
else
  assign_self_dist_sf_2d(mon_per_chain, nr_positions_cm,
 binIndexValid,
 sf_data_start_index, results[binIndex],
 sf_data_cm_start_index, results_cm[binIndex],
 binIndexValid_xy,
 0, NULL, 0, NULL);
/* positive quadrant, take also the negative quadrant in y:
 * qy < 0 */
if(qx_flag != 0 && qy_flag != 0){
  for(part = 0; part < nr_positions; part++){
    sin_q[part] = cosqx_cosqy[part] + sinqx_sinqy[part];
    cos_q[part] = sinqx_cosqy[part] - cosqx_sinqy[part];
  }
  for(part_cm = 0; part_cm < nr_positions_cm; part_cm++){
    sin_q_cm[part_cm] = \
      cosqx_cosqy_cm[part_cm] + sinqx_sinqy_cm[part_cm];
    cos_q_cm[part_cm] = \
      sinqx_cosqy_cm[part_cm] - cosqx_sinqy_cm[part_cm];
  }
  if(l_write_xy_plane)
    assign_self_dist_sf_2d(mon_per_chain, nr_positions_cm,
   binIndexValid,
   sf_data_start_index, results[binIndex],
   sf_data_cm_start_index,results_cm[binIndex],
   binIndexValid_xy,
   sf_data_xy_start_index,
   results_xy[binIndex_xy],
   sf_data_cm_xy_start_index,
   results_cm_xy[binIndex_xy]);
  else
    assign_self_dist_sf_2d(mon_per_chain, nr_positions_cm,
   binIndexValid,
   sf_data_start_index, results[binIndex],
   sf_data_cm_start_index,results_cm[binIndex],
   binIndexValid_xy,
   0, NULL, 0, NULL);
}/* if(qx_flag != 0 && qy_flag != 0) */
      }/* if |q| in allowed range */
    }/* loop over qi_y */
  }/* loop over qi_x */
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Leaving low_d_calc_all_sf_2d(), file %s, line %u\n",
 HERE);
#endif
} /* low_d_calc_all_sf_2d() */

/* given sin(\vec{q} \in \vec{r_i}) and cos(\vec{q} \in \vec{r_i}) for all
 * particle and center of mass positions
 * calculate the self and distinct structure factors.
 */
void
assign_self_dist_sf_2d(const unsigned int mon_per_chain,
       const unsigned int nr_chains,
       const boolean binIndexValid,
       const unsigned int sf_data_start_index,
       STRUF_RES_TYPE * const bin_results,
       const unsigned int sf_data_cm_start_index,
       STRUF_RES_TYPE * const bin_results_cm,
       const boolean binIndexValid_xy,
       const unsigned int sf_data_xy_start_index,
       STRUF_RES_TYPE * const bin_results_xy,
       const unsigned int sf_data_cm_xy_start_index,
       STRUF_RES_TYPE * const bin_results_cm_xy){
  double helper;
  /* loop indices */
  unsigned int chain, first_mon_in_chain;
  /* sum variables */
  double sum_cos, sum_sin, sum_sin_cm, sum_cos_cm;
#ifdef CALC_ONLY_COHERENT_SF_YES
  unsigned int part_index;
#else
  unsigned int second_mon_in_chain, first_help_index, second_help_index;
  unsigned int result_column,result_xy_column;
  double sum_cos_first_mon, sum_sin_first_mon, \
    sum_cos_second_mon, sum_sin_second_mon;
#endif

#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Entering assign_self_dist_sf_2d(), file %s, line %u\n",
 HERE);
#endif
#ifdef DEBUG_ASSIGN_SELF_DIST_SF_2D_ON
  printf("DEBUG assign_self_dist_sf_2d(): binIndex=%u, binIndexValid=%d\n", \
 binIndex, binIndexValid);
  printf("DEBUG assign_self_dist_sf_2d(): binIndex_xy=%u, binIndexValid_xy=\
%d\n", binIndex_xy, binIndexValid_xy);
#endif

  /* it is important for efficiency to avoid two-fold sums! */
  /* structure factor of the chain centers of mass */
  sum_sin_cm = 0.0;
  sum_cos_cm = 0.0;
  for(chain = 0; chain < nr_chains; ++chain){
    sum_sin_cm += sin_q_cm[chain];
    sum_cos_cm += cos_q_cm[chain];
  }
  helper = SQUARE(sum_sin_cm) + SQUARE(sum_cos_cm);
  if(binIndexValid)
```

```
    bin_results_cm[sf_data_cm_start_index] += helper;
  if(binIndexValid_xy)
    bin_results_cm_xy[sf_data_cm_xy_start_index] += helper;
#ifndef CALC_ONLY_COHERENT_SF_YES
  /* structure factor of the monomers in the chain, self contribution */
  for(first_mon_in_chain = 0; \
      first_mon_in_chain < mon_per_chain; first_mon_in_chain++){
    for(chain = 0, first_help_index = first_mon_in_chain; \
chain < nr_chains; chain++, first_help_index += mon_per_chain){
helper = sin_q[first_help_index] * sin_q_cm[chain] \
+ cos_q[first_help_index] * cos_q_cm[chain];
      if(binIndexValid)
bin_results_cm[sf_data_cm_start_index + 1 + first_mon_in_chain] \
  += helper;
      if(binIndexValid_xy)
bin_results_cm_xy[sf_data_cm_xy_start_index+1+first_mon_in_chain] \
  += helper;
    }/* end loop over chains */
  }/* end loop over monomers in chain */
  /* structure factor of the monomers in the chain, distinct contribution */
  for(first_mon_in_chain = 0; \
      first_mon_in_chain < mon_per_chain; first_mon_in_chain++){
    sum_sin = 0.0;
    sum_cos = 0.0;
    for(chain = 0, first_help_index = first_mon_in_chain; \
chain < nr_chains; \
chain++, first_help_index += mon_per_chain){
      sum_sin += sin_q[first_help_index];
      sum_cos += cos_q[first_help_index];
    }/* end loop over first chain index */
    helper = sum_sin * sum_sin_cm + sum_cos * sum_cos_cm;
    if(binIndexValid)
      bin_results_cm[sf_data_cm_start_index + 1 + mon_per_chain \
   + first_mon_in_chain] += helper;
    if(binIndexValid_xy)
      bin_results_cm_xy[sf_data_cm_xy_start_index + 1 + mon_per_chain \
   + first_mon_in_chain] += helper;
  }/* end loop over monomers in chain */
  /* in which column-index the results are written */
  result_column = sf_data_start_index + 2;
  result_xy_column = sf_data_xy_start_index + 2;
  /* structure factor of the monomer pairs in a chain */
  for(first_mon_in_chain = 0; \
      first_mon_in_chain < mon_per_chain; \
      first_mon_in_chain++){
    /* because we skip the second_mon_in_chain == first_mon_in_chain
     * calculation we advance over this column */
    result_column++;
    result_xy_column++;
    /* we know the answer for
     * second_mon_in_chain == first_mon_in_chain: 1
     * (but this makes a good test)
     */
    for(second_mon_in_chain = first_mon_in_chain +1; \
second_mon_in_chain < mon_per_chain; \
second_mon_in_chain++){
      for(chain = 0, \
    first_help_index = first_mon_in_chain, \
    second_help_index = second_mon_in_chain; \
  chain < nr_chains; \
  chain++, \
    first_help_index += mon_per_chain, \
    second_help_index += mon_per_chain){
helper = sin_q[first_help_index] * sin_q[second_help_index] \
  + cos_q[first_help_index] * cos_q[second_help_index];
if(binIndexValid)
  bin_results[result_column] += helper;
if(binIndexValid_xy)
  bin_results_xy[result_xy_column] += helper;
      }/* end loop over chains */
      /* go to next column */
      result_column++;
      result_xy_column++;
    }/* end loop over second_mon_in_chain */
  }/* end loop over first_mon_in_chain */
  /* result_column and result_xy_column have the right value */
  /* structure factor of the monomer pairs in different chains */
  for(first_mon_in_chain = 0; \
      first_mon_in_chain < mon_per_chain; \
      first_mon_in_chain++){
    sum_sin_first_mon = 0.0;
    sum_cos_first_mon = 0.0;
    for(chain = 0, first_help_index = first_mon_in_chain; \
chain < nr_chains; \
chain++, first_help_index += mon_per_chain){
      sum_sin_first_mon += sin_q[first_help_index];
      sum_cos_first_mon += cos_q[first_help_index];
    }
    /* use symmetry in the monomer indices */
    for(second_mon_in_chain = first_mon_in_chain; \
second_mon_in_chain < mon_per_chain; \
second_mon_in_chain++){
      sum_sin_second_mon = 0.0;
      sum_cos_second_mon = 0.0;
      for(chain = 0, second_help_index = second_mon_in_chain; \
  chain < nr_chains; \
  chain++, second_help_index += mon_per_chain){
sum_sin_second_mon += sin_q[second_help_index];
sum_cos_second_mon += cos_q[second_help_index];
      }
      helper = sum_sin_first_mon * sum_sin_second_mon \
+ sum_cos_first_mon * sum_cos_second_mon;
      if(binIndexValid)
bin_results[result_column] += helper;
      if(binIndexValid_xy)
bin_results_xy[result_xy_column] += helper;
      /* go to next column */
      result_column++;
      result_xy_column++;
    }/* end loop over second_mon_in_chain */
  }/* end loop over first_mon_in_chain */
  /* substract self contribution later */
#else
  /* coherent structure factor of the melt, self contribution */
  for(chain = 0, part_index = 0; chain < nr_chains; chain++){
    sum_sin = 0.0;
    sum_cos = 0.0;
    for(first_mon_in_chain = 0; first_mon_in_chain < mon_per_chain; \
first_mon_in_chain++, part_index++){
      sum_sin += sin_q[part_index];
      sum_cos += cos_q[part_index];
    }
      helper = SQUARE(sum_sin) + SQUARE(sum_cos);
      /* we use only the first entry in the result matrices */
      if(binIndexValid)
bin_results[sf_data_start_index] += helper;
      if(binIndexValid_xy)
bin_results_xy[sf_data_xy_start_index] += helper;
  }
  /* coherent structure factor of the melt (distinct) */
  sum_sin = 0.0;
  sum_cos = 0.0;
  for(part_index = 0; part_index < nr_chains * mon_per_chain; ++part_index){
    sum_sin += sin_q[part_index];
    sum_cos += cos_q[part_index];
  }
  helper = SQUARE(sum_sin) + SQUARE(sum_cos);
  if(binIndexValid)
    bin_results[sf_data_start_index +1] += helper;
  if(binIndexValid_xy)
    bin_results_xy[sf_data_xy_start_index +1] += helper;
  /* substract self contribution later (if pure distinct contribution is
   * desired)*/
#endif
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Leaving assign_self_dist_sf_2d(), file %s, line %u\n",
 HERE);
#endif
}/* assign_self_dist_sf_2d() */
#endif /* THREE_D */
/* This function takes a matrix with the results with its dimensions along
 * with information about what kind of data we are writing here.
 * This function is tuned for the structure factor data.
 */
void
write_results_sf(STRUF_RES_TYPE ** const results,
 const unsigned int nr_values,
 const unsigned int nr_kind_of_values,
 const unsigned int column_with_counter,
 const boolean l_write_xy_plane,
 const char * outfile_name,
 char ** description,
 const unsigned int nr_desc_lines,
 const char * parameterfile){
  FILE *savefile_p, *parameterfile_p;
  char read_line[79];
  char write_line[81];
  unsigned int iLoop, saved, plane_size_x = UINT_MAX;
  if(l_write_xy_plane){
    plane_size_x = (unsigned int)ceil(sqrt((double)nr_values));
    /* test */
    if(SQUARE(plane_size_x) != nr_values){
      fprintf(stderr, "ERROR write_results_sf(): (plane_size_x=%u)^2!=\
nr_values=%u\n", plane_size_x, nr_values);
      exit(3);
    }
  }
  if ((savefile_p=fopen(outfile_name, "w"))==NULL) {
    fprintf(stderr,"ERROR: Couldn't open outputfile %s \n", outfile_name);
    exit(21);
  }
  fprintf(stdout,"MESSAGE: Writing data to '%s' ...\n", outfile_name );
  /* write some hopefully descriptive lines */
  for(iLoop = 0; iLoop < nr_desc_lines; ++iLoop)
    fprintf(savefile_p, "%s", description[iLoop]);
  /* now write every value to a file */
  if(l_write_xy_plane){
    for(saved=0; saved < nr_values; ++saved){
      for(iLoop=0; iLoop < nr_kind_of_values; ++iLoop)
fprintf(savefile_p,"%#f ",results[saved][iLoop]);
      fprintf(savefile_p,"\n");
      if(saved % plane_size_x == plane_size_x -1)
fprintf(savefile_p, "\n");
    }
  }else{
    /* first line: comment out, because it is at q=0 */
    fprintf(savefile_p, "# ");
    for(iLoop=0; iLoop < nr_kind_of_values; ++iLoop)
      fprintf(savefile_p,"%#f ",results[0][iLoop]);
    fprintf(savefile_p,"\n");
    /* for the lines with q > 0 check if results are > 0 */
    for(saved=1; saved < nr_values; ++saved){
      if(results[saved][column_with_counter] > 0){
for(iLoop=0; iLoop < nr_kind_of_values; ++iLoop)
  fprintf(savefile_p,"%#f ",results[saved][iLoop]);
fprintf(savefile_p,"\n");
      }
    }
  }
  fprintf(savefile_p, "\n\n#Parameters read from file '%s':\n#\n", \
  parameterfile);
  if ((parameterfile_p = fopen(parameterfile,"r")) == NULL)
    fprintf(stdout,"WARNING: Couldn't open %s \n", parameterfile);
  else{
    /* just copy the parameterfile to the end of the datafile, so that we know
       how we got these numbers... */
    /* Not very good: lines are truncated after 78 characters */
    while(fgets(read_line, 79, parameterfile_p) != NULL){
      /* the #'s signal gnuplot and xmgr that these lines are comments */
      sprintf(write_line,"#%s", read_line);
      fputs(write_line, savefile_p);
      /* if there's no '\n' at the end of the line, put one there */
      if(strchr(read_line, (int)'\n') == NULL) write_line[79]='\n';
    }
    fclose(parameterfile_p);
  }
  fprintf(savefile_p,"\n");
  fclose(savefile_p);
  fprintf(stdout,"MESSAGE: ...done writing '%s'.\n", outfile_name);
}/* end write_results_sf() */
```

General library header

```
/* general_libVx.y.h
 * header file for frequently used routines for analysis code.
 * Goes back to ancient times when I started my diploma thesis (1999)
 * Martin Aichele, 2002-04-23
 * last modified: 2002-06-03
 * V1.1 reads YASP format configurations by H. Meyer
 * Martin Aichele, 2002-10-31
 * last modified: 2002-11-04
 */
#ifdef __cplusplus
extern "C" {
#endif
#ifndef _GENERAL_LIB_H
#define _GENERAL_LIB_H
#include<stdio.h>
#include<string.h>
#include<stdlib.h> /* needed for 'malloc' and the like */
#include<math.h>
/* header for the YASP trajectory i/o routines */
#include"yasptrj.h"
/* for preprocessing time dimension switching */
#define TWO_D
/**#define THREE_D */
/* define dimension */
#ifdef TWO_D
  #define DIM 2
#endif
#ifdef THREE_D
  #define DIM 3
#endif
/* switch for run time diagnostic output */
#define DIAGNOSTICS_OFF
/* structure for 3-dimensional vectors.  We don't call it "vector", because
 * there is the class of the same name in the C++ STL
 */
typedef struct {
   double x;
   double y;
   double z;
} r3vector;
/* structure for 2-dimensional vectors */
typedef struct {
   double x;
   double y;
} r2vector;
typedef int boolean;
#define TRUE 1
#define FALSE 0
/* define macros */
#ifndef M_PI
#define PI M_PI /* M_PI should be in math.h */
#else
#define PI 3.14159265358979323846
#endif
/* type safe MIN and MAX by Morten Welinder <terra@diku.dk>
 * as included in the Linux Kernel > V2.4.10
 */
#define MIN(x,y)                                \
  ({ const typeof(x) _x = x;                    \
     const typeof(y) _y = y;                    \
                                                \
     (void) (&_x == &_y);                       \
                                                \
     _x < _y ? _x : _y;                         \
  })

#define MAX(x,y)                                \
  ({ const typeof(x) _x = x;                    \
     const typeof(y) _y = y;                    \
                                                \
     (void) (&_x == &_y);                       \
                                                \
     _x > _y ? _x : _y;                         \
  })

#define SQUARE(x) ( (x)*(x) )
/* squared distance of two r3vectors */
#define DIST_SQ(a,b) (SQUARE((a).x-(b).x)+SQUARE((a).y-(b).y) \
+SQUARE((a).z-(b).z))
#define DIST_SQ2D(a,b) (SQUARE((a).x-(b).x)+SQUARE((a).y-(b).y))
/* modulus of the difference of two r3vectors */
#define DISTANCE(a,b) (sqrt(SQUARE((a).x-(b).x)+SQUARE((a).y-(b).y)\
+SQUARE((a).z-(b).z)))
/* modulus of a r3vector */
#define MODULUS(a) (sqrt(SQUARE((a).x)+SQUARE((a).y)+SQUARE((a).z)))
#define MODULUS2D(a) (sqrt(SQUARE((a).x)+SQUARE((a).y)))
/* modulus squared of a r3vector */
#define MODULUS_SQ(a) (SQUARE((a).x)+SQUARE((a).y)+SQUARE((a).z))
#define MODULUS_SQ2D(a) (SQUARE((a).x)+SQUARE((a).y))
/* a and b are r3vectors */
#define SCALARPRODUCT(a,b) (((a).x)*((b).x)+((a).y)*((b).y)+((a).z)*((b).z))
#define SCALARPRODUCT2D(a,b) (((a).x)*((b).x)+((a).y)*((b).y))
/* this is for identifying annoying lines of code
 * used when calling error()
 */
#define HERE __FILE__, __LINE__
/* prints an error message, filename, and line where it was called */
void
error(const char *, const char *, const int);
/* the difference of two r3vectors */
r3vector diffvec(const r3vector, const r3vector);
/* the sum of two r3vectors */
r3vector sumvec(const r3vector, const r3vector);
/* prints a r3vector to the screen */
void print_r3vector(const r3vector);
/* dynamically allocates memory for a matrix(nrl..nrh)(ncl..nch)
   with unsigned long integer entries; taken from "Numerical Recipes " */
unsigned long int **ulimatrix(int, int, int, int);
/* frees the memory taken by the unsigned long integer matrix */
void free_ulimatrix(unsigned long int**, int, int, int, int);
/* dynamically allocates memory for a matrix(nrl..nrh)(ncl..nch)
   with unsigned integer entries; taken from "Numerical Recipes " */
unsigned int **umatrix(int, int, int, int);
/* frees the memory taken by the unsigned long integer matrix */
void free_umatrix(unsigned int**, int, int, int, int);
/* dynamically allocates memory for a matrix(nrl..nrh)(ncl..nch) with double
entries; taken from "Numerical Recipes" */
double **dmatrix(int, int, int, int);
/* frees the memory taken by the integer matrix */
void free_dmatrix(double**, int, int, int, int);

/* dynamically allocates memory for a rank 3 tensor
 * (nhl..nhh)(nrl..nrh)(ncl..nch) with double entries
 */
double ***d_rank3tensor(const int, const int, \
const int, const int, \
const int, const int);
/* frees the memory taken by the double rank 3 tensor */
void free_d_rank3tensor(double ***, \
const int, const int, \
const int, const int, \
const int, const int);
/* dynamically allocates memory for a matrix(nrl..nrh)(ncl..nch) with double
entries; taken from "Numerical Recipes" */
char **charmatrix(int, int, int, int);
/* frees the memory taken by the integer matrix */
void free_charmatrix(char**, int, int, int, int);
/* dynamically allocates memory for a matrix(nrl..nrh)(ncl..nch) with r3vector
entries; taken from "Numerical Recipes" */
r3vector **vecmatrix(int, int, int, int);
/* frees the memory taken by the r3vector matrix */
void free_vecmatrix(r3vector**, int, int, int, int);
/* reads a configuration */
void readconf(const char*, const int, const unsigned long int, r3vector*);
/* reads a configuration */
int readconf_rs(const char*, const char*, const char*, const unsigned int, \
const unsigned long int, r3vector * const);
/* reads a configuration in 2-d */
int readconf_rs_2d(const char*, const char*, const char*, const unsigned int, \
   const unsigned long int, r3vector * const);
/* Reads fluid positions from YASP trajectory */
int readconf_yasp(char[], YaspFrame * const,
  const unsigned int,
  const unsigned int,
  r3vector * const);
/* Reads fluid and wall positions in configuration file of the film simulations
 */
int read_pos_rs_all(const char *, const char *, const int, const int,
    const unsigned long int,
    r3vector * const, r3vector * const,
    r3vector * const, r3vector * const);
/* transforms the fluid positions to the center of mass positions of the two
 * walls */
void transform_to_walls_cm(r3vector * const,
   const unsigned int,
   const r3vector,
   const r3vector);
/* calculates the center of mass for every chain in the melt at a fixed time
 */
void calc_Rcm_chains(const r3vector * const,
     const unsigned int,
     unsigned int const * const,
     r3vector * const);
/* takes a matrix with the results along with some information and writes
 * them to a file
 */
void save_results(double **,
                  const unsigned int,
  const unsigned int,
  const char *, const char *, const char *,
  const char *);
/* This function takes a matrix with the results with its dimensions along
 * with information about what kind of data we are writing here.
 */
void
write_results(double **,
      const unsigned int,
      const unsigned int,
      const char *,
      char **,
      const unsigned int,
      const char *);

#endif
#ifdef __cplusplus
}
#endif
```

General library code

```
/* general_libVx.y.c
 * library for frequently used routines for analysis code.
 * Goes back to ancient times when I started my diploma thesis (1999)
 * Uses some code by C. Bennemann (1997-1998)
 * Martin Aichele, 2002-04-23
 * last modified: 2002-05-29
 * V1.1 reads YASP format configurations by H. Meyer
 * Martin Aichele, 2002-10-31
 * last modified: 2002-11-04
 */
#include "general_libV1.1.h"
#define DEBUG_CALC_RCM_CHAINS_OFF
/* the difference of two r3vectors */
r3vector diffvec(const r3vector a, const r3vector b){
  r3vector help_vec;
  help_vec.x = a.x - b.x;
  help_vec.y = a.y - b.y;
  help_vec.z = a.z - b.z;
  return help_vec;
}/* end diffvec() */

/* the sum of two r3vectors */
r3vector sumvec(const r3vector a, const r3vector b){
  r3vector help_vec;
  help_vec.x = a.x + b.x;
  help_vec.y = a.y + b.y;
  help_vec.z = a.z + b.z;
  return help_vec;
}/* end sumvec() */

/* prints a r3vector to the screen */
void print_r3vector(const r3vector vec){
  printf("(%f, %f, %f)\n", vec.x, vec.y, vec.z);
}/* end print_r3vector() */

/* prints an error message, filename, and line where it was called */
void
error(const char * error_message, const char * file, const int line){
  fprintf(stderr,"ERROR: in file %s, line %d :\n%s\n", file, line,
  error_message);
  fprintf(stderr,"Exiting gracefully.\n");
```

```
  exit(1);
} /* end error() */

/* dynamically allocates memory for a matrix(nrl..nrh)(ncl..nch)
   with unsigned long integer entries; taken from "Numerical Recipes" */
unsigned long int **ulimatrix(int nrl, int nrh, int ncl,int nch){
  int i;
  unsigned long int**m;
  if( nrh < nrl || nch < ncl){
    fprintf(stderr, "ERROR: Wrong index structure in ulimatrix()\n");
    exit(1);
  }
  m=(unsigned long int **)malloc((unsigned) (nrh-nrl+1)*sizeof(unsigned long \
int*));
  if (!m) fprintf(stderr,"ERROR: Allocation failure 1 in ulimatrix()\n");
  m -= nrl;
  for(i=nrl;i<=nrh;i++) {
    m[i]=(unsigned long int *)malloc((unsigned) (nch-ncl+1)*sizeof(unsigned \
long int));
    if (!m[i]) fprintf(stderr,"ERROR: Allocation failure 2 in ulimatrix()\n");
    m[i] -= ncl;
  }
  return m;
}/* end ulimatrix(....) */

/* frees the memory taken by the unsigned long integer matrix */
void free_ulimatrix(unsigned long int **m, int nrl, int nrh, int ncl, int nch){
  int i;
  if( nrh < nrl || nch < ncl){
    fprintf(stderr, "ERROR: Wrong index structure in free_ulimatrix()\n");
    exit(1);
  }
  for(i=nrh;i>=nrl;i--) free((void*) (m[i]+ncl));
  free((void*) (m+nrl));
}/* end free_ulimatrix() */

/* dynamically allocates memory for a matrix(nrl..nrh)(ncl..nch)
   with unsigned integer entries; taken from "Numerical Recipes" */
unsigned int **umatrix(int nrl, int nrh, int ncl,int nch){
  int i;
  unsigned int**m;
  if( nrh < nrl || nch < ncl){
    fprintf(stderr, "ERROR: Wrong index structure in umatrix()\n");
    exit(1);
  }
  m=(unsigned int **)malloc((unsigned) (nrh-nrl+1)*sizeof(unsigned int*));
  if (!m) fprintf(stderr,"ERROR: Allocation failure 1 in umatrix()\n");
  m -= nrl;
  for(i=nrl;i<=nrh;i++) {
    m[i]=(unsigned int *)malloc((unsigned) (nch-ncl+1)*sizeof(unsigned int));
    if (!m[i]) fprintf(stderr,"ERROR: Allocation failure 2 in umatrix()\n");
    m[i] -= ncl;
  }
  return m;
}/* end umatrix(....) */

/* frees the memory taken by the unsigned integer matrix */
void free_umatrix(unsigned int **m, int nrl, int nrh, int ncl, int nch){
  int i;
  if( nrh < nrl || nch < ncl){
    fprintf(stderr, "ERROR: Wrong index structure in free_umatrix()\n");
    exit(1);
  }
  for(i=nrh;i>=nrl;i--) free((void*) (m[i]+ncl));
  free((void*) (m+nrl));
}/* end free_umatrix() */

/* dynamically allocates memory for a matrix(nrl..nrh)(ncl..nch)
   with double entries; taken from "Numerical Recipes" */
double **dmatrix(int nrl, int nrh, int ncl,int nch){
  int i;
  double **m;
  if( nrh < nrl || nch < ncl){
    fprintf(stderr, "ERROR: Wrong index structure in dmatrix()\n");
    exit(1);
  }
  m=(double **)malloc((unsigned) (nrh-nrl+1)*sizeof(double*));
  if (!m) fprintf(stderr,"ERROR: Allocation failure 1 in dmatrix()\n");
  m -= nrl;
  for(i=nrl;i<=nrh;i++) {
    m[i]=(double *)malloc((unsigned) (nch-ncl+1)*sizeof(double));
    if (!m[i]) fprintf(stderr,"ERROR: Allocation failure 2 in dmatrix()\n");
    m[i] -= ncl;
  }
  return m;
}/* end dmatrix(....) */

/* frees the memory taken by the double matrix */
void free_dmatrix(double **m, int nrl, int nrh, int ncl, int nch){
  int i;
  if( nrh < nrl || nch < ncl){
    fprintf(stderr, "ERROR: Wrong index structure in free_dmatrix()\n");
    exit(1);
  }
  for(i=nrh;i>=nrl;i--) free((void*) (m[i]+ncl));
  free((void*) (m+nrl));
}/* end free_dmatrix(.....) */

/* dynamically allocates memory for a rank 3 tensor
 * (nhl..nhh)(nrl..nrh)(ncl..nch) with double entries
 */
double ***d_rank3tensor(const int nhl, const int nhh, \
const int nrl, const int nrh, \
const int ncl, const int nch){
  int iLoop, jLoop;
  double ***tensor;
  if( nhh < nhl || nrh < nrl || nch < ncl){
    fprintf(stderr, "ERROR: Wrong index structure in d_rank3tensor()\n");
    exit(1);
  }
  tensor = (double ***)malloc((unsigned)(nhh-nhl+1)*sizeof(double**));
  if (!tensor)
    fprintf(stderr, "ERROR: Allocation failure 1 in d_rank3tensor()\n");
  tensor -= nhl;
  for(jLoop = nhl; jLoop <= nhh; jLoop++){
    tensor[jLoop] = (double **)malloc((unsigned)(nrh-nrl+1)*sizeof(double*));
    if (!tensor[jLoop])
      fprintf(stderr, "ERROR: Allocation failure 2 in d_rank3tensor()\n");
    tensor[jLoop] -= nrl;
    for(iLoop = nrl; iLoop <= nrh; iLoop++) {
      tensor[jLoop][iLoop] = \
(double *)malloc((unsigned)(nch-ncl+1)*sizeof(double));
      if (!tensor[jLoop][iLoop])
fprintf(stderr,"ERROR: Allocation failure 3 in d_rank3tensor()\n");
      tensor[jLoop][iLoop] -= ncl;
    }
  }
  return tensor;
}/* end d_rank3tensor() */

/* frees the memory taken by the double rank 3 tensor */
void free_d_rank3tensor(double ***tensor, \
const int nhl, const int nhh, \
const int nrl, const int nrh, \
const int ncl, const int nch){
  int iLoop, jLoop;
  if( nhh < nhl || nrh < nrl || nch < ncl){
    fprintf(stderr, "ERROR: Wrong index structure in free_d_rank3tensor()\n");
    exit(1);
  }
  /* imitate the free() behavior */
  if(tensor == NULL) return;
  for(jLoop = nhh; jLoop >= nhl; jLoop--){
    for(iLoop = nrh; iLoop >= nrl; iLoop--){
      free((void*) (tensor[jLoop][iLoop]+ncl));
    }
    free((void*) (tensor[jLoop]+nrl));
  }
  free((void*)(tensor + nhl));
}/* end free_d_rank3tensor() */

/* dynamically allocates memory for a matrix(nrl..nrh)(ncl..nch)
   with char entries; taken from "Numerical Recipes" */
char **charmatrix(int nrl, int nrh, int ncl,int nch){
  int i;
  char **m;
  if( nrh < nrl || nch < ncl){
    fprintf(stderr, "ERROR: Wrong index structure in charmatrix()\n");
    exit(1);
  }
  m=(char **)malloc((unsigned) (nrh-nrl+1)*sizeof(char*));
  if (!m) fprintf(stderr,"ERROR: Allocation failure 1 in charmatrix()\n");
  m -= nrl;
  for(i=nrl;i<=nrh;i++) {
    m[i]=(char *)malloc((unsigned) (nch-ncl+1)*sizeof(char));
    if (!m[i]) fprintf(stderr,"ERROR: Allocation failure 2 in charmatrix()\n");
    m[i] -= ncl;
  }
  return m;
}/* end charmatrix(....) */

/* frees the memory taken by the char matrix */
void free_charmatrix(char **m, int nrl, int nrh, int ncl, int nch){
  int i;
  if( nrh < nrl || nch < ncl){
    fprintf(stderr, "ERROR: Wrong index structure in free_charmatrix()\n");
    exit(1);
  }
  for(i=nrh;i>=nrl;i--) free((void*) (m[i]+ncl));
  free((void*) (m+nrl));
}/* end free_charmatrix(.....) */

/* dynamically allocates memory for a matrix(nrl..nrh)(ncl..nch)
   with r3vector entries, so actually it's a tensor */
r3vector **vecmatrix(int nrl, int nrh, int ncl,int nch){
  int i;
  r3vector **m;

  if( nrh < nrl || nch < ncl){
    fprintf(stderr, "ERROR: Wrong index structure in vecmatrix()\n");
    exit(1);
  }
  /* allocate the memory */
  m=(r3vector **)malloc((unsigned) (nrh-nrl+1)*sizeof(r3vector*));
  if (!m) fprintf(stderr,"ERROR: Allocation failure 1 in vecmatrix()\n");
/* C-arrays have indices 0,..,N so we define an offset here if we want to
     start with nrl */
  m -= nrl;
  for(i=nrl;i<=nrh;i++) {
    m[i]=(r3vector *)malloc((unsigned) (nch-ncl+1)*sizeof(r3vector));
    if (!m[i]) fprintf(stderr,"ERROR: Allocation failure 2 in vecmatrix()\n");
    m[i] -= ncl;
  }
  return m;
}/* end imatrix(....) */

/* frees the memory taken by the r3vector matrix */
void free_vecmatrix(r3vector **m, int nrl, int nrh, int ncl, int nch){
  int i;
  if( nrh < nrl || nch < ncl){
    fprintf(stderr, "ERROR: Wrong index structure in free_vecmatrix()\n");
    exit(1);
  }
  /* void* is the generic type of a memory address */
  for(i=nrh;i>=nrl;i--) free((void*) (m[i]+ncl));
  free((void*) (m+nrl));
}/* end free_vecmatrix(.....) */

/* Reads a configuration of C. Bennemann's bulk simulations */
void readconf(const char *path, const int nr_monomers,
      const unsigned long int time,
      r3vector *conf){
  int i;
  char fname[160];
  FILE *file_p;
  r3vector *monomer;
  sprintf(fname,"%s/xyz_t=%lu.dat", path, time);
  if ((file_p=fopen(fname,"r"))==NULL) {
    fprintf(stderr,"ERROR: Couldn't open %s\n",fname);
    exit(2);
  }
  /* fprintf(stdout,"MESSAGE: Reading %s\n",fname); */
  monomer = conf;
  for(i=0; i<nr_monomers; i++,monomer++)
    fscanf(file_p,"%lf %lf %lf\n",&monomer->x,&monomer->y,&monomer->z);
  if (i != nr_monomers){
    fprintf(stderr,"ERROR: In file %s only %d monomer positions!\n",fname,i);
    exit(2);
  }
  fclose(file_p);
} /* end readconf() */

/* Reads fluid positions in configuration file.
 * This is a more general routine (rs stands for rheosim)
 * Reads data from (as the data format is the same, but the file
 * naming scheme differs):
 * Cristoph Bennemann's 10mer simulations
 * Martin Aichele's 10mer simulations
 * Martin Aichele's simulations with rheosimVx.y in 2d
 * Fathollah Varniks simulations (position files)
 * Sidi Khefifs simulations (based on F. Varniks code)
 */
int readconf_rs(const char *path, const char *prefix, const char * postfix,
const unsigned int nr_monomers,
const unsigned long int time,
r3vector * const conf){
  unsigned int partLoop;
  char fname[400];
  FILE *file_p;
  r3vector *monomer;
  sprintf(fname,"%s/%s%lu%s", path, prefix, time, postfix);
  if ((file_p=fopen(fname,"r"))==NULL) {
```

```
    fprintf(stderr,"ERROR (readconf_rs): Couldn't open %s\n",fname);
    return 1;
  }
#ifdef DIAGNOSTICS_ON
  fprintf(stdout, "DIAGNOSTICS (readconf_rs): Reading %s\n",fname);
#endif
  monomer = conf;
  for(partLoop = 0; partLoop < nr_monomers; partLoop++, monomer++)
    fscanf(file_p,"%lf %lf %lf\n",&monomer->x,&monomer->y,&monomer->z);
  if (partLoop != nr_monomers){
    fprintf(stderr,"ERROR (readconf_rs): In file %s only %d monomer \
positions!\n",fname,partLoop);
    return 2;
  }
  fclose(file_p);
  return 0;
} /* end readconf_rs() */
/* Reads fluid positions in configuration file.  This version is for real
 * 2-d data.
 */
int readconf_rs_2d(const char *path, const char *prefix, const char * postfix,
    const unsigned int nr_monomers,
    const unsigned long int time,
    r3vector * const conf){
  unsigned int partLoop;
  char fname[400];
  FILE *file_p;
  r3vector *monomer;
  sprintf(fname,"%s/%s%lu%s", path, prefix, time, postfix);
  if ((file_p=fopen(fname,"r"))==NULL) {
    fprintf(stderr,"ERROR (readconf_rs_2d): Couldn't open %s\n",fname);
    return 1;
  }
#ifdef DIAGNOSTICS_ON
  fprintf(stdout, "DIAGNOSTICS (readconf_rs_2d): Reading %s\n",fname);
#endif
  monomer = conf;
  for(partLoop = 0; partLoop < nr_monomers; partLoop++, monomer++){
    fscanf(file_p,"%lf %lf\n",&monomer->x,&monomer->y);
    /* zero out z-coordinate */
    monomer->z = 0.0;
  }
  if (partLoop != nr_monomers){
    fprintf(stderr,"ERROR (readconf_rs_2d): In file %s only %d monomer \
positions!\n",fname,partLoop);
    return 2;
  }
  fclose(file_p);
#ifdef DIAGNOSTICS_ON
  fprintf(stdout, "DIAGNOSTICS (readconf_rs_2d): Read %s\n",fname);
#endif
  return 0;
} /* end readconf_rs_2d() */

/* Reads fluid positions from YASP trajectory */
/* returns 0 on success, 1 if a frame could not be read because the
 * trajectory ended before, 2 if a frame could not be read otherwise.
 * exits if the problem with the trajectory is fundamental */
int readconf_yasp(char filename[], YaspFrame * const frame_p,
  const unsigned int nr_monomers,
  const unsigned int n_frame,
  r3vector * const conf){
  unsigned int iLoop;
  int atoms;
  char YaspTitle[80]; int records = 0;
  static boolean l_trj_is_open = FALSE;
  static unsigned int framecounter = 0;
#ifdef DIAGNOSTICS_ON
  fprintf(stdout, "DIAGNOSTICS Entering readconf_yasp() with n_frame=%u, \
framecounter=%u\n", n_frame, framecounter);
#endif
  /* initialize or re-initialize trajectory */
  if(n_frame <= framecounter || !l_trj_is_open){
    if(l_trj_is_open){
      printf("MESSAGE (readconf_yasp): Re-initializing trajectory\n");
      framecounter = 0;
      /* if the trajectory was open before, we have to close it first */
      CloseYaspTrj();
    }else{
      printf("MESSAGE (readconf_yasp): Initializing trajectory\n");
      l_trj_is_open = TRUE;
    }
    if (OpenYaspTrj(filename)){
      fprintf(stderr, "ERROR (readconf_yasp): Could not open YASP \
trajectory '%s'\n", filename);
      exit(1);
    }
    atoms = GetYaspAtoms();
    if((unsigned)atoms != nr_monomers){
      fprintf(stderr, "ERROR (readconf_yasp): atoms in YASP trajectory = %d\
 != nr_monomers=%u\n", atoms, nr_monomers);
      exit(2);
    }
    /*     GetYaspHeader() obtains the descriptor string title and the
     *                     number of records per frame, skips
     *                     to the first frame. Must be called
     *                     before any call to GetYaspFrame(). Returns 0
     *                     on success, nonzero otherwise.
     */
    if (GetYaspHeader(YaspTitle, &records)){
      fprintf(stderr, "ERROR (readconf_yasp): Could not open trajectory\
 header\n");
      exit(3);
    }
  }
  /* read frames until we reach n_frame */
  do{
    if(YaspEOF()){
      fprintf(stdout,"WARNING (readconf_yasp): Reached end of trajectory.\n");
      fprintf(stdout,"WARNING                  n_frame=%u, framecounter=%u\n",
      n_frame, framecounter);
      return 1;
    }
    if (GetYaspFrame(nr_monomers, frame_p)){
      fprintf(stdout,"WARNING (readconf_yasp): Error when reading frame.\n");
      fprintf(stdout,"WARNING                  n_frame=%u, framecounter=%u\n",
      n_frame, framecounter);
      return 2;
    }
    framecounter++;
  } while(framecounter < n_frame);
  /* this is now the right frame with n_frame == framecounter */
  if(DIM == 3){
    for (iLoop=0; iLoop < nr_monomers; iLoop++) {
      conf[iLoop].x = frame_p->x[iLoop];
      conf[iLoop].y = frame_p->y[iLoop];
      conf[iLoop].z = frame_p->z[iLoop];
    }
  }else{
    for (iLoop=0; iLoop < nr_monomers; iLoop++) {
      conf[iLoop].x = frame_p->x[iLoop];
      /* swap y and z */
      conf[iLoop].y = frame_p->z[iLoop];
      conf[iLoop].z = frame_p->y[iLoop];
    }
  }
#ifdef DIAGNOSTICS_ON
  fprintf(stdout, "DIAGNOSTICS Leaving readconf_yasp()\n");
#endif
  return 0;
} /* end readconf_yasp() */

/* Reads fluid and wall positions in configuration file of the film simulations
 */
int read_pos_rs_all(const char *path, const char *prefix,
    const int nr_fluid_atoms,
    const int nr_wall_atoms,
    const unsigned long int time,
    r3vector * const conf_fluid,
    r3vector * const conf_wall,
    r3vector * const pos_twall,
    r3vector * const pos_bwall){
  int i_fluid, counter_fluid;
  int i_wall, counter_wall, counter_tbwall;
  int n_mon_tot, n_top_wall, n_bottom_wall, i_part_low, i_part_high;
  char fname[400];
  FILE *file_p;
  r3vector *pos_p;
  /* initialize */
  counter_fluid = 0;
  counter_wall = 0;
  counter_tbwall = 0;
  sprintf(fname,"%s/%s%lu.dat", path, prefix, time);
#ifdef DIAGNOSTICS_ON
  fprintf(stdout, "DIAGNOSTICS (read_pos_rs_all): Reading %s\n",fname);
#endif
  if ((file_p=fopen(fname,"r"))==NULL) {
    fprintf(stderr,"ERROR (read_pos_rs_all): Couldn't open %s\n",fname);
    return 1;
  }
  /* fprintf(stdout,"MESSAGE: Reading %s\n",fname); */
  pos_p = conf_fluid;
  for(i_fluid = 0; i_fluid < nr_fluid_atoms; i_fluid++, pos_p++)
    counter_fluid += \
      fscanf(file_p,"%lf %lf %lf\n", &pos_p->x, &pos_p->y, &pos_p->z);
  if (counter_fluid != 3*nr_fluid_atoms){
    fprintf(stderr, "ERROR (read_pos_rs_all): In file %s only %d fluid \
positions!\n", fname, counter_fluid/3);
    return 2;
  }
  pos_p = conf_wall;
  for(i_wall = 0; i_wall < nr_wall_atoms; i_wall++, pos_p++)
    counter_wall += \
      fscanf(file_p,"%lf %lf %lf\n", &pos_p->x, &pos_p->y, &pos_p->z);
  if (counter_wall != 3*nr_wall_atoms){
    fprintf(stderr, "ERROR (read_pos_rs_all): In file %s only %d wall \
positions!\n", fname, counter_wall/3);
    return 3;
  }
  /* read control numbers in file */
  fscanf(file_p, "%u %u %u %u %u\n", &n_mon_tot, &n_top_wall, &n_bottom_wall,\
 &i_part_low, &i_part_high);
  if(i_part_high - i_part_low +1 != nr_fluid_atoms + nr_wall_atoms){
    fprintf(stderr, \
    "ERROR (read_pos_rs_all): Particle number check failed:\n");
    fprintf(stderr, "       i_part_low=%u\n", i_part_low);
    fprintf(stderr, "       i_part_high=%u\n", i_part_high);
    fprintf(stderr, "       nr_fluid_atoms + nr_wall_atoms=%u\n", \
    nr_fluid_atoms + nr_wall_atoms);
    return 4;
  }
  /* read top and bottom wall position */
  counter_tbwall += \
    fscanf(file_p, "%lf %lf %lf\n", &pos_twall->x,&pos_twall->y,&pos_twall->z);
  counter_tbwall += \
    fscanf(file_p, "%lf %lf %lf\n", &pos_bwall->x,&pos_bwall->y,&pos_bwall->z);
  if(counter_tbwall != 6){
    fprintf(stderr,"ERROR (read_pos_rs_all): Failed to read wall positions\n");
    return 5;
  }
  fclose(file_p);
  return 0;
} /* end read_pos_rs_all() */

/* transforms the fluid positions to the center of mass positions of the two
 * walls */
void transform_to_walls_cm(r3vector * const conf,
   const unsigned int nr_monomers,
   const r3vector twall_pos,
   const r3vector bwall_pos){
  unsigned int iLoop;
  r3vector walls_cm_pos;
  walls_cm_pos.x = (twall_pos.x + bwall_pos.x)/2.0;
  walls_cm_pos.y = (twall_pos.y + bwall_pos.y)/2.0;
  walls_cm_pos.z = (twall_pos.z + bwall_pos.z)/2.0;
  for(iLoop = 0; iLoop < nr_monomers; iLoop++)
    conf[iLoop] = sumvec(conf[iLoop], walls_cm_pos);
}/* end transform_to_walls_cm() */

/* calculates the center of mass for every chain in the melt at a fixed time
 */
void calc_Rcm_chains(r3vector const * const configuration,
     const unsigned int nr_chains,
     unsigned int const * const chain_lengths,
     r3vector * const Rcm_chains){
  unsigned int chain, mon_1, index_mon_1, chainlength;
  unsigned int sum_monomers = 0;
  r3vector help_vec;
  /* loop over all chains */
  for(chain = 0, index_mon_1 = 0; chain < nr_chains; chain++){
    chainlength = chain_lengths[chain];
    /* test if the chain has at least one monomer
     * if there's a chain with length 0 something is seriously wrong
     */
    if(chainlength == 0){
      fprintf(stderr, "ERROR (calc_Rcm_chains): Chain %u has length 0 !\n",
      chain);
      exit (1);
    }
    /* keep track of the number of monomers in the melt */
    sum_monomers += chainlength;
    /* initialize help_vec */
      help_vec.x = 0.0;
      help_vec.y = 0.0;
      help_vec.z = 0.0;
```

```
    /* now loop over all monomers in a chain */
    for(mon_1 = 0; mon_1 < chainlength; mon_1++, index_mon_1++){
      /* add up the positions */
      help_vec.x += configuration[index_mon_1].x;
      help_vec.y += configuration[index_mon_1].y;
      help_vec.z += configuration[index_mon_1].z;
    }/* end loop over one chain */
    /* divide by the number of monomers in the chain */
    help_vec.x /= (double)chainlength;
    help_vec.y /= (double)chainlength;
    help_vec.z /= (double)chainlength;
    /* assign value to array containing the results */
    Rcm_chains[chain] = help_vec;
  }/* end loop over all chains */
  if(index_mon_1 != sum_monomers){
    fprintf(stderr, "ERROR: Wrong index in calc_Rcm_chains() :\nindex_mon_1 = \
%u, but sum_monomers = %u\n", index_mon_1, sum_monomers);
    exit(3);
  }
#ifdef DEBUG_CALC_RCM_CHAINS_ON
  printf("DEBUG calc_Rcm_chains(): Printing monomer positions:\n");
  for(index_mon_1 = 0; index_mon_1 < sum_monomers; ++index_mon_1)
    print_r3vector(configuration[index_mon_1]);
  printf("\n");
  printf("DEBUG calc_Rcm_chains():Printing chain center of mass positions:\n");
  for(chain = 0; chain < nr_chains; ++chain)
    print_r3vector(configuration[chain]);
  printf("\n");
#endif
} /* end calc_Rcm_chains() */

/* takes a matrix with the results along with some information and writes
 * them to a file
 */
void save_results(double **results,
  const unsigned int nr_values,
  const unsigned int nr_kind_of_values,
  const char *what, const char *origin,
  const char *header,
  const char *parameterfile){
  FILE *savefile_p, *parameterfile_p;
  char outfile[160];
  char read_line[79];
  char write_line[81];
  unsigned int i, saved;

  /* put together the name of the file used for saving */
  /* sprintf needs an char array which is big enough for its result ! */
  sprintf(outfile, "%s_%s.dat", what, origin);
  if ((savefile_p=fopen(outfile,"w"))==NULL) {
    fprintf(stderr,"ERROR: Couldn't open outputfile %s \n", outfile);
    exit(1);
  }
  fprintf(stdout,"MESSAGE: Writing data to '%s' ...\n", outfile );
  /* if theres a header string describing the contents of the file we write
   * it at the beginning of the file
   */
  if(strlen(header) > 0)
    fprintf(savefile_p, "%s", header);
  /* now write every value to a file */
  for (saved = 0; saved < nr_values; ++saved){
    for(i=0; i < nr_kind_of_values; ++i)
      fprintf(savefile_p,"%.6E ", results[saved][i]);
    fprintf(savefile_p,"\n");
  }
  fprintf(savefile_p, "\n#Data generated by %s\n#Parameters read from file \
'%s':\n#\n", origin, parameterfile);
 if ((parameterfile_p = fopen(parameterfile,"r")) == NULL)
    fprintf(stdout,"WARNING: Couldn't open %s \n", parameterfile);
 else{
   /* just copy the parameterfile to the end of the datafile, so that we know
      how we got these numbers... */
   /* Not very good: lines are truncated after 78 characters */
   while(fgets(read_line, 79, parameterfile_p) != NULL){
     /* the #'s signal gnuplot and xmgr that these lines are comments */
     sprintf(write_line,"#%s", read_line);
     fputs(write_line, savefile_p);
     /* if there's no '\n' at the end of the line, put one there */
     if(strchr(read_line, (int)'\n') == NULL) write_line[79]='\n';
   }
   fclose(parameterfile_p);
 }
 fprintf(savefile_p,"\n");
 fclose(savefile_p);
 fprintf(stdout,"MESSAGE: ...done writing '%s'.\n", outfile);
}/* end save_results() */

/* This function takes a matrix with the results with its dimensions along
 * with information about what kind of data we are writing here.
 */
void
write_results(double **results,
      const unsigned int nr_values,
      const unsigned int nr_kind_of_values,
      const char * outfile_name,
      char ** description,
      const unsigned int nr_desc_lines,
      const char * parameterfile){
  FILE *savefile_p, *parameterfile_p;
  char read_line[79];
  char write_line[81];
  unsigned int iLoop, saved;
  if ((savefile_p=fopen(outfile_name, "w"))==NULL) {
    fprintf(stderr,"ERROR: Couldn't open outputfile %s \n", outfile_name);
    exit(21);
  }
  fprintf(stdout,"MESSAGE: Writing data to '%s' ...\n", outfile_name );
  /* write some hopefully descriptive lines */
  for(iLoop = 0; iLoop < nr_desc_lines; ++iLoop)
    fprintf(savefile_p, "%s", description[iLoop]);
  /* now write every value to a file */
  /* the first line is special, because the time is 0.0 and this creates
   * problems on a logarithmic plot
   */
  if(results[0][0] == 0.0)
    fprintf(savefile_p, "# ");
  for(saved=0; saved < nr_values; ++saved){
    for(iLoop=0; iLoop < nr_kind_of_values; ++iLoop)
fprintf(savefile_p,"%#f ",results[saved][iLoop]);
    fprintf(savefile_p,"\n");
  }
  fprintf(savefile_p, "\n\n#Parameters read from file '%s':\n#\n", \
  parameterfile);
  if ((parameterfile_p = fopen(parameterfile,"r")) == NULL)
    fprintf(stdout,"WARNING: Couldn't open %s \n", parameterfile);
  else{
    /* just copy the parameterfile to the end of the datafile, so that we know
       how we got these numbers... */
    /* Not very good: lines are truncated after 78 characters */
    while(fgets(read_line, 79, parameterfile_p) != NULL){
      /* the #'s signal gnuplot and xmgr that these lines are comments */
      sprintf(write_line,"#%s", read_line);
      fputs(write_line, savefile_p);
      /* if there's no '\n' at the end of the line, put one there */
      if(strchr(read_line, (int)'\n') == NULL) write_line[79]='\n';
    }
    fclose(parameterfile_p);
  }
 fprintf(savefile_p,"\n");
 fclose(savefile_p);
 fprintf(stdout,"MESSAGE: ...done writing '%s'.\n", outfile_name);
}/* end write_results() */
```

D.2 Code for the Three Particle Static Structure Factors

Main s3V1.6.cpp

```
// calculate the triple correlators S_3
// Martin Aichele, 2001-11-12
// last modified 2001-12-05
/*include "static_libV0.5.h"*/
#include "general_libV1.1.h"
#include "qvec_libV1.7.h"
#include "mt19937-1.h"
/* program version to be appended to the data written */
#define VERSION "V1.6"
/* identify program */
#define WHAT "S3"
/* length of a dummy string used to swallow unwanted input */
#define DUMMY_LENGTH 1000
// set to DRY_RUN_ONLY to bypass S_3 calculation, only the reciprocal lattice
// vector searching is done
#define DRY_RUN_ONLYXXX
int
main(int argc, char *argv[]){
  FILE *parameterfile_p;
  /* #chains, #monomers per chain, total # of monomers */
  unsigned int nr_chains, mon_per_chain, nr_monomers;
  /* simulation box size */
  double box_size;
  /* parameter reading checking variables */
  unsigned int read_success, read_attempt;
  /* loop variables */
  unsigned int iLoop, jLoop;
  /* information about the simulation data */
  unsigned int nr_sim_runs, *nr_sub_sim_runs;
  unsigned int nr_startpoints, nr_samples_per_startpoint, nr_timeseries;
  unsigned int nr_configurations, nr_configurations_counter=0;
  /* path to data */
  char data_path[160], data_path_temp[163];
  /* filename containing the times at which configurations were saved */
  char file_sample_times[160];
  /* string for determining if we use a sample matrix or list */
  char saving_scheme[80];
  /* times at which configurations have been saved */
  /* if we have just a list, the dimensions are chosen approbriately */
  unsigned long int **sample_matrix;
  /* loop variables */
  unsigned int confLoop, simRunLoop, subSimRunLoop, startPointLoop;
  /* configuration */
  r3vector *configuration_t0;
  /* needed for dealing with filenames */
  int stringlength;
  /* a string for messages */
  char string[400];
  /* dummy string to swallow unneeded input */
  char * dummy;
  ////////////////////////////////////////////////////////////////////////////
  // declarations specific for S_3
  ////////////////////////////////////////////////////////////////////////////
  unsigned int nrBins; // number of MCT-calculation bins
  double binWidth;     // width of these bins in reciprocal space
  double qMax;         // maximal modulus of reciprocal vectors
  // bin indices
  int qBin, kBin, pBin;
  int pBinMin, pBinMax;
  // number of tupels (\vec{q}, \vec{k}) we want in each (q, k, p) bin
  unsigned int wishedEntries;
  // factor for equalizing the number of vector tupels in each (q, k, p) bin
  double equalizationFactor;
  // start-up value for the random number generator
  unsigned int rngSeed;
```

```
// read in configuration loop states
unsigned int confLoopStart, simRunLoopStart, subSimRunLoopStart, \
  startPointLoopStart;
// how many configurations are to be calculated before program exits
unsigned int nr_confs_to_calculate = 0;
// name of file for new parameters
char newParameterFileName[800];
/*************************************************************************
 *                  D O N E   W I T H   D E C L A R A T I O N S          *
 *************************************************************************/
/* say hello */
printf("MESSAGE: this is %s\n", argv[0]);
/* Read parameter file */
if (argc != 2) {
  fprintf(stderr,"ERROR: Usage is %s parameter_file\n",
  argv[0]);
  exit(1);
}
if ((parameterfile_p = fopen(argv[1],"r")) == NULL){
  sprintf(string, "Couldn't open %s", argv[1]);
  error(string, HERE);
}
// DEBUG
parameterfile_p = fopen("s3params","r");
/* with read_success we keep track of the number of successfully
   read variables */
read_success=0;
read_attempt=0;
/* dimension of box */
++read_attempt;
read_success += fscanf(parameterfile_p,"box_size=%lf\n",&box_size);
/*  number of chains in the melt*/
++read_attempt;
read_success += fscanf(parameterfile_p,"nr_chains=%u\n",&nr_chains);
/* monomers per chain */
++read_attempt;
read_success += fscanf(parameterfile_p,"mon_per_chain=%u\n",&mon_per_chain);
/* how many different simulation runs */
++read_attempt;
read_success += fscanf(parameterfile_p,"nr_sim_runs=%u\n", &nr_sim_runs);
if(read_success != read_attempt){
  sprintf(string,"Read only %u variables, but expected %u\n", read_success,
  read_attempt);
  error(string, HERE);
}
/* number of sub simulation runs */
nr_sub_sim_runs = (unsigned int *)calloc(nr_sim_runs, sizeof(unsigned int));
for(iLoop=0; iLoop < nr_sim_runs; ++iLoop){
  ++read_attempt;
  read_success += fscanf(parameterfile_p,"%u\n", nr_sub_sim_runs+iLoop);
}
/* string describing the saving scheme */
++read_attempt;
read_success += fscanf(parameterfile_p,"saving_scheme=%s\n", saving_scheme);
/* filename of the file that contains the sample times */
++read_attempt;
read_success += fscanf(parameterfile_p,"file_sample_times=%s\n",
file_sample_times);
/* path to the configurations to be read, data is expected in the
 * directories
 * "data_path[sim_run].[sub_sim_run]/"
 */
++read_attempt;
read_success += fscanf(parameterfile_p,"data_path=%s\n", data_path);
////////////////////////////////////////////////////////////////////////////
// input variables for S_3
++read_attempt;
read_success += fscanf(parameterfile_p,"nrBins=%u\n", &nrBins);
++read_attempt;
read_success += fscanf(parameterfile_p,"binWidth=%lf\n", &binWidth);
++read_attempt;
read_success += fscanf(parameterfile_p,"wishedEntries=%u\n", &wishedEntries);
++read_attempt;
read_success += fscanf(parameterfile_p,"equalizationFactor=%lf\n",
&equalizationFactor);
++read_attempt;
read_success += fscanf(parameterfile_p,"rngSeed=%u\n", &rngSeed);
// define the maximal q-value
qMax = static_cast<double>(nrBins)*binWidth;
// read states of configuration loops
++read_attempt;
read_success += fscanf(parameterfile_p,"nr_confs_to_calculate=%u\n",
&nr_confs_to_calculate);
++read_attempt;
read_success += fscanf(parameterfile_p,"confLoopStart=%u\n",
&confLoopStart);
++read_attempt;
read_success += fscanf(parameterfile_p,"simRunLoopStart=%u\n",
&simRunLoopStart);
++read_attempt;
read_success += fscanf(parameterfile_p,"subSimRunLoopStart=%u\n",
&subSimRunLoopStart);
++read_attempt;
read_success += fscanf(parameterfile_p,"startPointLoopStart=%u\n",
&startPointLoopStart);
////////////////////////////////////////////////////////////////////////////
/* check if we read as many variables as expected */
if(read_success != read_attempt){
  sprintf(string, "Read only %u variables, but expected %u\n", read_success,
  read_attempt);
  error(string, HERE);
}
fclose(parameterfile_p);
printf("MESSAGE: Parameters used:\n");
printf("=============================================================\n");
printf("box_size=%f\n", box_size);
printf("nr_chains=%d\nmon_per_chain=%d\n", nr_chains, mon_per_chain);
printf("nr_sim_runs=%u\n", nr_sim_runs);
for(iLoop=0; iLoop<nr_sim_runs; ++iLoop)
  printf("simulation run %u has %u sub runs\n", iLoop,
 nr_sub_sim_runs[iLoop]);
printf("-------------------------------------------------------------\n");
printf("%s", "Additional parameters for S_3:\n");
printf("nrBins=%u\n", nrBins);
printf("binWidth=%f", binWidth);
printf(" (therefore: qMax=%f)\n", qMax);
printf("wishedEntries=%u\n", wishedEntries);
printf("equalizationFactor=%f\n", equalizationFactor);
printf("rngSeed=%u\n\n", rngSeed);
printf("=============================================================\n");
printf("nr_confs_to_calculate=%u\n", nr_confs_to_calculate);
printf("start values of configuration loops:\n");
printf("confLoopStart=%u\n", confLoopStart);
printf("simRunLoopStart=%u\n", simRunLoopStart);
printf("subSimRunLoopStart=%u\n", subSimRunLoopStart);
printf("startPointLoopStart=%u\n\n", startPointLoopStart);
printf("=============================================================\n");
/*************************************************************************
 *           D O N E   R E A D I N G   P A R A M E T E R S               *
 *************************************************************************/
/* check if saving scheme is recognized */
if( !(strcmp(saving_scheme, "list") == 0)
|| (strcmp(saving_scheme, "matrix") == 0))
  error("ERROR: Unrecognised saving scheme", HERE);
/* now read the times */
if (strcmp(saving_scheme, "list") == 0){
  if ((parameterfile_p = fopen(file_sample_times, "r")) == NULL){
    sprintf(string, "ERROR: Couldn't open %s", file_sample_times);
    error(string, HERE);
  }
  /* how many samples there are */
  if(fscanf(parameterfile_p, "NR_SAMPLES_TOTAL=%u\n",
    &nr_samples_per_startpoint) == 1)
    printf("MESSAGE: nr_samples_per_startpoint = %u\n",
   nr_samples_per_startpoint);
  else
    error("ERROR: Couldn't read nr_samples_per_startpoint in sample_list",
  HERE);
  if(nr_samples_per_startpoint == 0)
    error("nr_samples_per_startpoint == 0\n", HERE);
  /* treat the list like a matrix */
  nr_startpoints = 1;
}else{ /* we have a sample_matrix */
  if ((parameterfile_p = fopen(file_sample_times, "r")) == NULL) {
    sprintf(string, "ERROR: Couldn't open %s \n", file_sample_times);
    error(string, HERE);
  }
  if(fscanf(parameterfile_p, "NR_of_columns=NR_STARTPOINTS=%u\n",
    &nr_startpoints)!= 1)
    error("ERROR: NR_STARTPOINTS not read correctly.\n", HERE);
  if(fscanf(parameterfile_p, "NR_of_rows=NR_SAMPLES_PER_STARTPOINT=%u\n",
    &nr_samples_per_startpoint) != 1)
    error("ERROR: NR_SAMPLES_PER_STARTPOINT not read correctly.\n", HERE);

  printf("\nMESSAGE: nr_startpoints=%u\n", nr_startpoints);
  printf("MESSAGE: nr_samples_per_startpoint=%u\n\n",
 nr_samples_per_startpoint);
}/* end if */
/* from now on we can hopefully treat the list like a matrix */
sample_matrix = ulimatrix(0, nr_startpoints-1,
  0, nr_samples_per_startpoint-1);
/* The fgets(dummy, DUMMY_LENGTH, file_p) stuff is here to allow to read
 * just the first nr_startpoints columns of a possibly bigger matrix.
 * Note that there's a difference between "%lu" and "%lu " in fscanf:
 * In the former case the file pointer points at the position after the last
 * read number, in the latter case to the position of the next non-white-
 * space character
 */
dummy = (char*)malloc(DUMMY_LENGTH * sizeof(char));
/* read the sample matrix entries */
printf("MESSAGE: Sample matrix:\n");
for(iLoop=0; iLoop < nr_samples_per_startpoint; ++iLoop){
  for(jLoop=0; jLoop < nr_startpoints; ++jLoop){
    fscanf(parameterfile_p, "%lu", sample_matrix[jLoop] + iLoop);
    printf("%lu ", sample_matrix[jLoop][iLoop]);
  }
  fgets(dummy, DUMMY_LENGTH, parameterfile_p);
  printf("\n");
}
fclose(parameterfile_p);
free(dummy);
/* how many timeseries there are */
nr_timeseries = 0;
for(iLoop=0; iLoop<nr_sim_runs; ++iLoop)
  nr_timeseries += nr_sub_sim_runs[iLoop];
nr_timeseries *= nr_startpoints;
/* how many configurations we have */
nr_configurations = nr_timeseries * nr_samples_per_startpoint;
/* sanity check */
if(nr_timeseries == 0)
  error("ERROR: There are no timeseries", HERE);
else
  printf("MESSAGE: There are %u timeseries.\n", nr_timeseries);
/* compute number of monomers in the melt */
nr_monomers = nr_chains * mon_per_chain;
/* put together filename of output file */
sprintf(string, "%s_%s.dat", WHAT, VERSION);
/* allocate memory for monomer positions */
configuration_t0 = (r3vector *)calloc(nr_monomers, sizeof(r3vector));
#ifdef DIAGNOSTICS_ON
  printf("DIAGNOSTICS: Now in main(), file %s, line %u\n", HERE);
#endif
  /* done with general tasks for static quantities */
  ////////////////////////////////////////////////////////////////////////////
  // more declarations for S3
  ////////////////////////////////////////////////////////////////////////////
  const double twoPiOverBoxlength = 2.0*PI/box_size;
  // we want to make sure that the last bin is completely filled, thus +1:
  const short int maxIndex = static_cast<short int>(qMax/twoPiOverBoxlength)+1;
  cout << "MESSAGE: maxIndex=" << maxIndex << "\n";
  // squared maximal lattice index
  const unsigned int maxIndexSq \
    = static_cast<unsigned int>(maxIndex * maxIndex);
  // at each length (in squared lattice units) we have a vector of reciprocal
  // lattice vectors.  0 is included.
  vector<vector<RLVecType> > qvecsAtSLUQ(maxIndexSq+1);
  vector<vector<RLVecType> > qvecsAtSLUK(maxIndexSq+1);
  // for each bin list all SLU belonging to this bin.
  vector<vector<int> > binToSLU(nrBins);
  // bin indices of squared lattice moduli.
  // we use ints to use -INT_MAX as a flag if this SLU is not in any bin.
  // 0 is included.  Because we are looking on *differences* of lattice
  // vectors, the highest length is DIM*(2*maxIndex)^2.
  // Vectors with squared modulus > maxIndexSq are not needed, but we have to
  // deal with them.
  vector<int> SLUtoBin(DIM*4*maxIndexSq+1);
  // here we can make a destinction between \vec{q} and \vec{k}:
  // if we are not interested in the imaginary part, we can take either
  // \vec{q} or \vec{k} from a half lattice.
#ifdef CALC_COMPLEX_NO
  assign_whole_qvectors_to_lengths(maxIndex, qvecsAtSLUQ);
  assign_half_qvectors_to_lengths(maxIndex, qvecsAtSLUK);
#endif
#ifdef CALC_COMPLEX_YES
  assign_whole_qvectors_to_lengths(maxIndex, qvecsAtSLUQ);
  assign_whole_qvectors_to_lengths(maxIndex, qvecsAtSLUK);
#endif
  // initialize conversion between reciprocal lattice vector (rl vector) bins
  // and squared lattice units
  mct_to_slu(binWidth, nrBins, twoPiOverBoxlength, maxIndexSq,
```

```
    binToSLU, SLUtoBin);
#ifdef TAKE_OUTER_BINS_YES
  cout << "MESSAGE: Taking outer bins.\n";
#else
  cout << "MESSAGE: Not taking outer bins.\n";
#endif
  // find out the maximal number of vectors for \vec{q} and \vec{K}
  // maxNumberAllK is either
  //               == maxNumberAllQ if we calculate the complex part
  //               == maxNumberAllQ/2 if we calculate only the real part of S_3
  // make sure that indexType is big enough (check how many vectors can be in
  // a bin with a given bin-index)
  unsigned int counterAllQ = 0;
  unsigned int maxNumberAllQ = 0;
  for(qBin = 0; qBin < static_cast<int>(nrBins); ++qBin){
    counterAllQ = 0;
    for(vector<int>::const_iterator iterSLUQ = binToSLU[qBin].begin(); \
iterSLUQ != binToSLU[qBin].end(); ++iterSLUQ)
      counterAllQ += qvecsAtSLUQ[*iterSLUQ].size();
    maxNumberAllQ = max(counterAllQ, maxNumberAllQ);
  }
  cout << "MESSAGE: maxNumberAllQ=" <<  maxNumberAllQ << "\n";
  unsigned int counterAllK = 0;
  unsigned int maxNumberAllK = 0;
  for(kBin = 0; kBin < static_cast<int>(nrBins); ++kBin){
    counterAllK = 0;
    for(vector<int>::const_iterator iterSLUK = binToSLU[kBin].begin(); \
iterSLUK != binToSLU[kBin].end(); ++iterSLUK)
      counterAllK += qvecsAtSLUK[*iterSLUK].size();
    maxNumberAllK = max(counterAllK, maxNumberAllK);
  }
  cout << "MESSAGE: maxNumberAllK=" <<  maxNumberAllK << "\n";
  // check if the index type for indexing rl vectors is large enough
  // INDEX_MAX itself is used as a flag for debugging!
  if(maxNumberAllQ >= static_cast<unsigned int>(INDEX_MAX)){
    cerr << "ERROR: maxNumberAllQ=" <<  maxNumberAllQ << " > INDEX_MAX="
 << INDEX_MAX << ", change indexType\n";
    exit (1);
  }
  // storage for the index tupels identifying tupels (\vec{q}, \vec{k}) for
  // given (q, k, p), \vec{p} = \vec{q} - \vec{k}
  // a lot of the entries will not be used.
  // For them qkpVecs[pBin].size() == 0
  vector<vector<indexTupelType> > qkpVecs(nrBins);
  // the rl vectors themselves are stored in these lists.
  // We separate the problem along q, ie. for each q we create a complete list
  // of \vec{q} at this q and then we loop over k.  Therefore it is more
  // efficient to take \vec{q} from the whole lattice when calculating only
  // the real part of S_3.
  vector<RLVecType> completeVecListQ, completeVecListK, uniqVecListK;
  // here we store the uniqVecLists scaled with the lattice constant
  vector<r3vector> completeVecListLatticeQ(maxNumberAllQ),
    uniqVecListLatticeK(maxNumberAllK);
  // results: #vector tupels, S_3^p, S_3.  If imaginary parts are also
  // calculated, reserve approbriate storage
#ifdef CALC_COMPLEX_NO
  const unsigned int nrResultEntries = 3;
#endif
#ifdef CALC_COMPLEX_YES
  const unsigned int nrResultEntries = 5;
#endif
  vector<vector<vector<vector<resultType> > > > \
    results(nrBins, vector<vector<vector<resultType> > > \
    (nrBins, vector<vector<resultType> > \
     (nrBins)));
  // only allocate what we need for the index tupels describing the vector
  // tupels
  for(qBin = 0; qBin < static_cast<int>(nrBins); ++qBin){
    for(kBin = 0; kBin < static_cast<int>(nrBins); ++kBin){
      // minimal and maximal index of p-bins which can be filled
#ifdef TAKE_OUTER_BINS_YES
      pBinMin = max(abs(qBin-kBin)-1, 0);
      pBinMax = min(qBin+kBin+1, static_cast<int>(nrBins-1));
#else
      // unless at 0 or nrBins-1, pBinMin and pBinMax are +1 resp. -1
      pBinMin = abs(qBin-kBin);
      pBinMax = qBin + kBin;
      if(pBinMax > static_cast<int>(nrBins) -1)
pBinMax = nrBins -1;
#endif
      for(pBin = pBinMin; pBin <=pBinMax; ++pBin){
// create entries where needed and set them to 0.0
// with the SGI STL this creates objects with capacity nrResultEntries
results[qBin][kBin][pBin].resize(nrResultEntries);
      }
    }
  }
  // the sines and cosines for all vectors in the vector lists
  vector<calculType*> allSinesQ, allCosinesQ, allSinesK, allCosinesK;
  allSinesQ.reserve(maxNumberAllQ);
  allCosinesQ.reserve(maxNumberAllQ);
  allSinesK.reserve(maxNumberAllK);
  allCosinesK.reserve(maxNumberAllK);
  // intermediate storage for S_3 and S_3^p for all p
  // g++ allows saying s3_re_chain_qk_at_all_p[nrBins], but icc complains
  sums3Type *s3_re_chain_qk_at_all_p, *s3_re_melt_qk_at_all_p;
  s3_re_chain_qk_at_all_p = (sums3Type*)calloc((size_t)nrBins, \
    (size_t)sizeof(sums3Type));
  s3_re_melt_qk_at_all_p = (sums3Type*)calloc((size_t)nrBins, \
   (size_t)sizeof(sums3Type));
#ifdef CALC_COMPLEX_YES
  sums3Type *s3_im_chain_qk_at_all_p, *s3_im_melt_qk_at_all_p;
  s3_im_chain_qk_at_all_p = (sums3Type*)calloc((size_t)nrBins, \
    (size_t)sizeof(sums3Type));
  s3_im_melt_qk_at_all_p = (sums3Type*)calloc((size_t)nrBins, \
   (size_t)sizeof(sums3Type));
#endif
  ////////////////////////////////////////////////////////////////////////////
  //  E N D   D E C L A R A T I O N S
  ////////////////////////////////////////////////////////////////////////////
  // initialize random number generator
  // when distributing the computation on several machines it is important to
  // use different seeds to improve statistics.
  sgenrand(rngSeed);
  // flush cout
  cout << flush;
  /*****************************************************************************
   *                          M A I N   L O O P S                              *
   *****************************************************************************/
  // loop over the configurations.
  /* loop over one timeseries */
  for(confLoop = confLoopStart; \
      confLoop < nr_samples_per_startpoint; ++confLoop){
    // confLoop
    // in case we use a lot of configurations per timeseries is is better to
    // draw new q-vectors for each configuration.  The confirurations of
    // different (sub-) simulation runs should be rather independent.
    /* loop over all simulation runs */
    for(simRunLoop = simRunLoopStart; \
simRunLoop < nr_sim_runs; ++simRunLoop){
      // simRunLoop
      /* Loop over sub runs in a simulation run. */
      for(subSimRunLoop = subSimRunLoopStart; \
  subSimRunLoop < nr_sub_sim_runs[simRunLoop]; ++subSimRunLoop){
// subSimRunLoop
/* complete the paths with the number of the simulation run and
   sub run */
stringlength = strlen(data_path);
strncpy(data_path_temp, data_path, (size_t)stringlength);
/* integer to ASCII conversion needed */
if(simRunLoop >= 10)
  data_path_temp[stringlength++] = (char)(simRunLoop / 10) + '0';
data_path_temp[stringlength++] = (char)(simRunLoop % 10) + '0';
data_path_temp[stringlength++] = '.';
if(subSimRunLoop >= 10)
  data_path_temp[stringlength++] = (char)(subSimRunLoop / 10)+'0';
data_path_temp[stringlength++] = (char)(subSimRunLoop % 10)+'0';
data_path_temp[stringlength] = '\0';
/* that's for example /data/bla_bla_bla/conf4.5 */
/* loop over all starting points */
for(startPointLoop = startPointLoopStart; \
    startPointLoop < nr_startpoints; ++startPointLoop){
  // startPointLoop
  cout << "MESSAGE: Now at configuration ..."
       << data_path_temp + strlen(data_path)
       << "/xyz_t=" << sample_matrix[startPointLoop][confLoop]
       << "\n" << flush;
  /*************************************************
   * DO CALCULATION FOR SINGLE CONFIGURATION HERE *
   *************************************************/
#ifndef DRY_RUN_ONLY
  readconf(data_path_temp, nr_monomers, \
   sample_matrix[startPointLoop][confLoop], \
   configuration_t0);
#endif
  /* count the number of read configurations for information */
  nr_configurations_counter++;
  for(qBin = static_cast<int>(nrBins)-1; qBin >= 0; --qBin){
    // qBin
    // create list of all \vec{q} in this q-bin
    completeVecListQ.clear();
    completeVecListQ.reserve(maxNumberAllQ);
    create_complete_list(qBin, binToSLU, qvecsAtSLUQ, \
 completeVecListQ);
    cout << "MESSAGE: Now at qBin=" << qBin
 << ", completeVecListQ.size()=" << completeVecListQ.size()
 << "\n" << flush;
#ifndef DRY_RUN_ONLY
    // scale lattice vectors with the lattice constant
    rescale_lattice_vectors(twoPiOverBoxlength, \
    completeVecListQ,
    completeVecListLatticeQ);
    // calc all sines and cosines for this configuration and
    // all \vec{q}
    calc_all_sines_cosines(nr_monomers, configuration_t0,
   completeVecListQ.size(),
   completeVecListLatticeQ,
   allSinesQ, allCosinesQ);
#endif
    // draw vector tupels for all k-bins and calculate
    for(kBin = static_cast<int>(nrBins)-1; kBin >= 0; --kBin){
      // kBin
      // clear the list of vectors
      uniqVecListK.clear();
      completeVecListK.clear();
      uniqVecListK.reserve(maxNumberAllK);
      completeVecListK.reserve(maxNumberAllK);
      // create list of all \vec{k} for this bin
      create_complete_list(kBin, binToSLU, qvecsAtSLUK, \
   completeVecListK);
      // minimal and maximal index of p-bins which can be filled
#ifdef TAKE_OUTER_BINS_YES
      pBinMin = max(abs(qBin-kBin)-1, 0);
      pBinMax = min(qBin+kBin+1, static_cast<int>(nrBins-1));
#else
      // unless at 0 or nrBins-1, pBinMin and pBinMax are +1 resp. -1
      pBinMin = abs(qBin-kBin);
      pBinMax = qBin + kBin;
      if(pBinMax > static_cast<int>(nrBins) -1)
pBinMax = nrBins -1;
#endif
      // wipe out previous vector tupels
      // we only want to clear() the vector<indexTupelType> for each
      // bin, not the object itself
      for(pBin = 0; pBin < static_cast<int>(nrBins); ++pBin){
qkpVecs[pBin].clear();
// if pBin is in the range where new tupels will be created,
// reserve memory.
if(pBin >= pBinMin && pBin <= pBinMax)
  qkpVecs[pBin].reserve(wishedEntries);
      }
      // get tupels for this k-bin and all p-bins
      qkp_realization(qBin, kBin, nrBins, wishedEntries,
      equalizationFactor,
      completeVecListQ, completeVecListK,
      maxNumberAllK, SLUtoBin,
      qkpVecs, uniqVecListK);
      // add up number of vector tupels found
      for(pBin = pBinMin; pBin <= pBinMax; ++pBin)
results[qBin][kBin][pBin][0] += \
  static_cast<resultType>(qkpVecs[pBin].size());
#ifndef DRY_RUN_ONLY
      // rescale lattice vectors and calculate
      rescale_lattice_vectors(twoPiOverBoxlength, \
      uniqVecListK, \
      uniqVecListLatticeK);
      qkp_calc_s3(nrBins, qkpVecs,
  uniqVecListK.size(), uniqVecListLatticeK,
  allSinesQ, allCosinesQ,
  allSinesK, allCosinesK,
  configuration_t0, mon_per_chain, nr_chains,
  s3_re_chain_qk_at_all_p,
  s3_re_melt_qk_at_all_p
#ifdef CALC_COMPLEX_YES
  , s3_im_chain_qk_at_all_p,
  s3_im_melt_qk_at_all_p
#endif
  );
      // assign to results
#ifdef TAKE_OUTER_BINS_YES
      pBinMin = max(abs(qBin-kBin)-1, 0);
      pBinMax = min(qBin+kBin+1, static_cast<int>(nrBins-1));
#else
```

```
      // unless at 0 or nrBins-1, pBinMin and pBinMax are +1 resp. -1
      pBinMin = abs(qBin-kBin);
      pBinMax = qBin + kBin;
      if(pBinMax > static_cast<int>(nrBins) -1)
pBinMax = nrBins -1;
#endif // TAKE_OUTER_BINS_YES
      for(pBin = pBinMin; pBin <=pBinMax; ++pBin){
#ifdef CALC_COMPLEX_NO
results[qBin][kBin][pBin][nrResultEntries-2] += \
  static_cast<resultType>(s3_re_chain_qk_at_all_p[pBin]);
results[qBin][kBin][pBin][nrResultEntries-1] += \
  static_cast<resultType>(s3_re_melt_qk_at_all_p[pBin]);
#endif // CALC_COMPLEX_NO
#ifdef CALC_COMPLEX_YES
results[qBin][kBin][pBin][nrResultEntries-4] += \
  static_cast<resultType>(s3_re_chain_qk_at_all_p[pBin]);
results[qBin][kBin][pBin][nrResultEntries-3] += \
  static_cast<resultType>(s3_im_chain_qk_at_all_p[pBin]);
results[qBin][kBin][pBin][nrResultEntries-2] += \
  static_cast<resultType>(s3_re_melt_qk_at_all_p[pBin]);
results[qBin][kBin][pBin][nrResultEntries-1] += \
  static_cast<resultType>(s3_im_melt_qk_at_all_p[pBin]);
#endif // CALC_COMPLEX_YES
      }// end loop over p-bins
#endif // #ifndef DRY_RUN_ONLY
    }// end loop over k-bin
  }// end loop over q-bin
#ifndef DRY_RUN_ONLY
  // write out results
  // confLoop+1 is the number of samples per startpoint we have
  // investigated so far.
  write_S3_results(argv[1], string,
confLoop, simRunLoop, subSimRunLoop, startPointLoop,
nr_configurations_counter, nr_monomers,
nrBins, binWidth, nrResultEntries, results);
#endif // #ifndef DRY_RUN_ONLY
  // write out parameters for continuation
  // name of new parameter file
  sprintf(newParameterFileName, "%s_new", argv[1]);
  // find out the next loop starting indices
  if(startPointLoop == nr_startpoints - 1){
    if(subSimRunLoop == nr_sub_sim_runs[simRunLoop] - 1){
      if(simRunLoop == nr_sim_runs - 1){
if(confLoop == nr_samples_per_startpoint - 1){
  cout << "MESSAGE:  Done with all remaining configurations."
       << "\n";
  printf("MESSAGE: %s says goodbye.\n", argv[0]);
  return 0;
}else{
  confLoopStart = confLoop + 1;
  simRunLoopStart = 0;
  subSimRunLoopStart = 0;
  startPointLoopStart = 0;
}
      }else{
simRunLoopStart = simRunLoop + 1;
subSimRunLoopStart = 0;
startPointLoopStart = 0;
      }
    }else{
      subSimRunLoopStart = subSimRunLoop + 1;
      startPointLoopStart = 0;
    }
  }else{
    startPointLoopStart = startPointLoop + 1;
  }
  // increment rngSeed for drawing different vectors the next time
  rngSeed += 1;
  write_new_params(box_size, nr_chains, mon_per_chain,
   nr_sim_runs, nr_sub_sim_runs, saving_scheme,
   file_sample_times, data_path,
   nrBins, binWidth, wishedEntries,
   equalizationFactor, rngSeed,
   nr_confs_to_calculate,
   confLoopStart, simRunLoopStart,
   subSimRunLoopStart, startPointLoopStart,
   newParameterFileName);
  // if we have calculated for enough configurations, exit
  if(nr_configurations_counter == nr_confs_to_calculate){
    cout << "MESSAGE: nr_configurations_counter == nr_confs_to_calcu\
late, exiting.\n";
    printf("MESSAGE: %s says goodbye.\n", argv[0]);
    return 0;
  }
}/* end loop over all startpoints */
    }/* end loop over all sub runs */
  }/* end loop over all simulation runs */
}/* end loop over one timeseries */
#ifdef DRY_RUN_ONLY
  cout << "MESSAGE: finishing dry run.\n";
  exit (0);
#endif
  /* free memory */
  free(configuration_t0);
  printf("MESSAGE: %s says goodbye.\n", argv[0]);
  return 0;
}/* end main() */
```

Q-vector library header

```
// header files for the lattice vectors library
// includes routines for S_3 calculations
// Martin Aichele, 2001-11-29
// Extensions for different lattice spacings and 2d, Martin Aichele, 2002-04-23
// last modified 2002-04-23
#ifndef _QVEC_LIB_H
#define _QVEC_LIB_H
#include "general_libV1.0.h"
#include "mt19937-1.h"
#include<iostream>
#include<cmath>
#include<climits>
// sometimes this style is needed
//#include<math.h>
//#include<limits.h>
#include<vector.h>
#include<algorithm>
// decide if the outermost bins are taken (_YES / _NO)
#define TAKE_OUTER_BINS_NO
// decide if (q, k, p) bins outside allowed range are written to the output
// file in order to get a complete file (_YES / _NO)
#define WRITE_EMPTY_POINTS_NO
// decide if complex part of S_3 is also calculated.  Might be a good test for
// statistics (_YES / _NO)
#define CALC_COMPLEX_NO
// decide, if the argument of the sines/cosines evaluations is declared as
// register variable.
// Speed improvement is not guaranteed.
#define USE_REGISTER_QINR_OFF
// decide of we save intermediate sums in the calculation of S_3.
// Should improve performance.
#define SAVE_S3_INTERMEDIATE_CHAIN_SUMS_ON
// probably no effect.
#define SAVE_S3_INTERMEDIATE_MELT_SUMS_OFF
// required for some compilers and good style.
using namespace std;
// the type of the reciprocal lattice vectors in lattice units.
// char would be big enough for our lattice, but there is no predefined
// arithmetic, so we have to use short ints.
// For performance, ints might be better.
typedef vector<short int> RLVecType;
// the type of the reciprocal lattice vectors in reciprocal space units.
typedef vector<double> RSVecType;
// the floating types here are chosen for adjusting precicion and memory
// consumption.  When changing the platform (32/64 bit) adjustments might be
// necessary.  Storage for intermediate sines and cosines uses half of the
// allocated memory.
// the type of the results
typedef double resultType;
// the type for intermediate sines and cosines
typedef float calculType;
// the type for summing up during the calculation of S_3
// when summing positive and negative numbers, a higher precision is
// advantageous.
typedef double sums3Type;
// the type of an uniq vector index
typedef short int indexType;
typedef short int unsignedIndexType;
// a tupel of indices describing a tupel of vectors
typedef struct indexTupelType{
  unsignedIndexType q;
  unsignedIndexType k;
}indexTupelType;
// the maximal index entry
#define INDEX_MAX SHRT_MAX
// type safe MIN and MAX by Morten Welinder <terra@diku.dk>
// as included in the Linux Kernel > V2.4.10
#define MIN(x,y)                                \
  ({ const typeof(x) _x = x;                    \
     const typeof(y) _y = y;                    \
                                                \
     (void) (&_x == &_y);                       \
                                                \
     _x < _y ? _x : _y;                         \
  })
#define MAX(x,y)                                \
  ({ const typeof(x) _x = x;                    \
     const typeof(y) _y = y;                    \
                                                \
     (void) (&_x == &_y);                       \
                                                \
     _x > _y ? _x : _y;                         \
  })
////////////////////////////////////////////////////////////////////////////////
//  F U N C T I O N   D E C L A R A T I O N S
////////////////////////////////////////////////////////////////////////////////
// NOTE: the keyword 'inline' forbids the use of the function declared inline
// outside the scope of the definition.
// That is: An inline function defined in qvec_libVx.y.cpp can only be called
// in this file.
// prints a list of RSVecs
void
printRSVecList(const vector<RSVecType>&);
// converts a RSVec to r3Vec.  Uses pointers to avoid copying.
void
RSVecType2r3vector(const RSVecType&, r3vector * const);
// the difference of two RLVecType variables
inline RLVecType
diffvec(const RLVecType&, const RLVecType&);
inline int
scalar_product(const RLVecType&, const RLVecType&);
void
assign_half_qvectors_to_lengths(const short int,
   vector<vector<RLVecType> >&);
void
assign_whole_qvectors_to_lengths(const short int,
   vector<vector<RLVecType> >&);
// create all vectors in the reciprocal lattice and take the one in a certain
// length range, measured in reciprocal space length (not lattice units).
// This version takes the positive half lattice
void
get_qvectors_in_range(const short int,
      const vector<double>&,
      const double,
      const double,
      vector<RSVecType>&,
      double&);
// given a complete list of vectors in a list of reciprocal vectors, select
// a random subset (or take all)
unsigned int get_random_subset(const unsigned int,
       const vector<RSVecType>&,
       vector<RSVecType>&);
void mct_to_slu(const double, const unsigned int,
const double,
const unsigned int,
vector<vector<int> >&,
vector<int>&);
// creates a list of all \vec{q} in a specified q-bin
void
create_complete_list(const int,
 const vector<vector<int> >&,
 const vector<vector<RLVecType> >&,
 vector<RLVecType>&);
// for a given tupel (q,k) of real numbers denoting the moduli of lattice
// vectors (i.e. the (q,k) index of a bin), search tupels of lattice vectors
// (\vec{q}, \vec{k}) sorted into bins according to |p|, where
```

```
// \vec{p} = \vec{q} - \vec{k}
void
qkp_realization(const int, const int,
const unsigned int,
const unsigned int,
const double,
const vector<RLVecType>&,
const vector<RLVecType>&,
const unsigned int,
const vector<int>&,
vector<vector<indexTupelType> >&,
vector<RLVecType>&);
// find overlaps between two uniquified vector lists and convert the unique
// vector indices
void
new_qk_index_conversion(const vector<RLVecType>&,
const vector<RLVecType>&,
vector<indexType>&);

// helper function which gets the number of successful index conversions
unsigned int
number_old_new_conversions(const vector<indexType>&);

// calculate sines and cosines for all vec{q} in this list
void
calc_all_sines_cosines(const unsigned int,
       const r3vector * const,
       const unsigned int,
       const vector<r3vector> &,
       vector<calculType* >&,
       vector<calculType* >&);

// given the tupels of vectors (\vec{q}, \vec{k}) and the particle positions,
// calculate S_3 for the melt and the chain.
// Here does the most CPU time go.
void
qkp_calc_s3(const unsigned int,
    const vector<vector<indexTupelType> >&,
    const unsigned int,
    const vector<r3vector>&,
    const vector<calculType* >&,
    const vector<calculType* >&,
    vector<calculType* >&,
    vector<calculType* >&,
    const r3vector * const,
    const unsigned int,
    const unsigned int,
    sums3Type * const,
    sums3Type * const
#ifdef CALC_COMPLEX_YES
    , sums3Type * const,
    sums3Type * const
#endif
    );

void
rescale_lattice_vectors(const double,
vector<RLVecType>&,
vector<r3vector>&);
// gets a uniquified list of RLVecs and returns the unique index of a RLVec.
// if RLVec does not exist, append it to the list.  Keep also track of indices
// which  occur more than once
inline unsignedIndexType
uniq_vec_index(vector<RLVecType>&,
       vector<unsignedIndexType>&,
       const RLVecType&);
// gets a uniquified list of RLVecs and returns the unique index of a RLVec.
// if RLVec does not exist, append it to the list.
// This function does not memorize vectors which occur more than once
inline unsignedIndexType
uniq_vec_index(vector<RLVecType>&,
       const RLVecType&);
// gets a uniquified list of RLVecs and returns the unique index of a RLVec.
// -INDEX_MAX is returned if RLVec was not found
inline indexType
UVLindex(const vector<RLVecType>&, const RLVecType&);
// writes out results
void
write_S3_results(char *,
 char *,
 const unsigned int,
 const unsigned int,
 const unsigned int,
 const unsigned int,
 const unsigned int,
 const unsigned int,
 const unsigned int,
 const double,
 const unsigned int,
 const vector<vector<vector<vector<resultType> > > >&);
void
write_new_params(const double,
 const unsigned int,
 const unsigned int,
 const unsigned int,
 const unsigned int * const,
 const char * const,
 const char * const,
 const char * const,
 const unsigned int,
 const double,
 const unsigned int,
 const double,
 unsigned int,
 const unsigned int,
 const unsigned int,
 const unsigned int,
 const unsigned int,
 const unsigned int,
 char * const);
#endif
```

Q-vector library code

```
// library for finding reciprocal lattice vectors and triplets of them
// includes routines for S_3 calculations
// Martin Aichele, 2001-11-06
// last modified 2001-12-06
// Extensions for different lattice spacings and 2d, Martin Aichele, 2002-04-23
// last modified 2002-04-30
#include"qvec_libV1.7.h"
#include"mt19937-1.h"
// DEBUG_???_ON turns on debugging output
#define DEBUG_ASSIGN_QVECS_TO_LENGTH_OFF
#define DEBUG_GET_QVECS_IN_RANGE_OFF
#define DEBUG_MCT_TO_SLU_OFF
#define DEBUG_QKP_REALIZATION_OFF
#define DEBUG_QKP_REALIZATION_VERBOSE_OFF
#define DEBUG_QKP_REALIZATION_COUNT_OFF
#define DEBUG_QKP_REALIZATION_RANDOM_OFF
#define DEBUG_QKP_CALC_S3_OFF
#define DEBUG_CALC_RE_S3_OFF
// some operators and functions
////////////////////////////////
// NOTE: the keyword 'inline' forbids the use of the function declared inline
// outside the scope of the definition.
// That is: An inline function defined in qvec_libVx.y.cpp can only be called
// in this file.
// prints a RSVec to the screen
void
printRSVec(const RSVecType& a){
  cout << "(";
  for(unsigned int loop = 0; loop < DIM-1; ++loop)
    cout << a[loop] << " ";
  cout << a[DIM-1];
  cout << ")\n";
}
void
printRSVecList(const vector<RSVecType>& rsveclist){
  for(vector<RSVecType>::const_iterator iter = rsveclist.begin();
      iter != rsveclist.end(); ++iter)
    printRSVec(*iter);
}
// converts a RSVec to r3Vec.  Uses pointers to avoid copying.
void
RSVecType2r3vector(const RSVecType& RSqVec, r3vector * const r3vec_p){
  r3vec_p->x = RSqVec[0];
  r3vec_p->y = RSqVec[1];
  if(DIM == 3){
    r3vec_p->z = RSqVec[2];
  }else{
    // C++ does not initialize _local_ objects to 0 of the
    // approbriate type, so we do this here.
    r3vec_p->z = 0.0;
  }
}// end convertRSVecType2r3vector()

// the difference of two RLVecType variables
inline RLVecType
diffvec(const RLVecType& a, const RLVecType& b){
  // no input check is done
  RLVecType c(DIM);
  for(unsigned int loop = 0; loop < DIM; loop++)
    c[loop] = a[loop] - b[loop];
  return c;
}
inline int
scalar_product(const RLVecType& a, const RLVecType& b){
  // no input check is done
  int AinB = 0;
  for(unsigned int loop = 0; loop < DIM; loop++)
    AinB += a[loop] * b[loop];
  return AinB;
}
inline int
mod_square_slu(const RLVecType& a){
  // no input check is done
  int aSq = 0;
  for(unsigned int loop = 0; loop < DIM; loop++)
    aSq += a[loop] * a[loop];
  return aSq;
}
// prints a RLVec to the screen
inline void
printRLVec(const RLVecType& a){
  cout << "(";
  for(unsigned int loop = 0; loop < DIM-1; ++loop)
    cout << a[loop] << " ";
  cout << a[DIM-1];
  cout << ")\n";
}

inline void
printIndexTupel(const indexTupelType& a){
  cout << "(" << a.q << "," << a.k << ")\n";
}

void
printRLVecList(const vector<RLVecType>& rlveclist){
  for(vector<RLVecType>::const_iterator iter = rlveclist.begin();
      iter != rlveclist.end(); ++iter)
    printRLVec(*iter);
}
void
printIndexTupelList(const vector<indexTupelType>& indextupellist){
  for(vector<indexTupelType>::const_iterator iter = indextupellist.begin();
      iter != indextupellist.end(); ++iter)
    printIndexTupel(*iter);
}

// create all vectors in the reciprocal lattice and store them by length
// measured in squared lattice units.
// this version takes the half lattice.
void
assign_half_qvectors_to_lengths(const short int maxIndex,
 vector<vector<RLVecType> >& qvecsAtSLU){
  const unsigned int maxIndexSq = static_cast<unsigned int>(maxIndex*maxIndex);
  // the length of a vector in squared lattice units
  unsigned int slu_length;
  // the vector in lattice units
  RLVecType rlVec(DIM);
#ifdef DEBUG_ASSIGN_QVECS_TO_LENGTH_ON
  cout << "DEBUG: Entering assign_half_qvectors_to_lengths()\n";
#endif
  // check if dimension is right.  This function with hardcoded dimension
  // is not very elegant, but it works
  if(DIM != 3){
    cerr << "ERROR (assign_half_qvectors_to_lengths): this code is meant for \
dimension 3.  Write another function for other dimensions.\n";
    exit (1);
  }
  // loop over half the lattice
  for(short int xIndex=0; xIndex <= maxIndex; xIndex++){
    for(short int yIndex = (xIndex == 0) ? 0 : -maxIndex; \
yIndex <= maxIndex; yIndex++){
      for(short int zIndex = (yIndex == 0 && xIndex == 0) ? 0 : -maxIndex; \
  zIndex <= maxIndex; zIndex++){
// this is not the fastest way, but this procedure is run only once,
```

```
// so efficiency is not prime.
slu_length = \
  static_cast<unsigned int>(xIndex*xIndex + yIndex*yIndex \
    + zIndex*zIndex);
#ifdef DEBUG_ASSIGN_QVECS_TO_LENGTH_ON
cout << "DEBUG: xIndex=" << xIndex << " yIndex=" << yIndex \
     << " zIndex=" << zIndex << ", slu_length=" << slu_length <<"\n";
#endif
if(slu_length <= maxIndexSq){
  rlVec[0] = xIndex;
  rlVec[1] = yIndex;
  rlVec[2] = zIndex;
#ifdef DEBUG_ASSIGN_QVECS_TO_LENGTH_ON
  cout << "appending ";
  printRLVec(rlVec);
  cout << "to qvecsAtSLU[" << slu_length << "]\n";
#endif
  qvecsAtSLU[slu_length].push_back(rlVec);
}// end if(slu_length <= maxIndexSq)
      }// ennd loop zIndex
    }// end loop yIndex
  }// end loop zIndex
#ifdef DEBUG_ASSIGN_QVECS_TO_LENGTH_ON
  for(unsigned int loop = 0; loop <= maxIndexSq; ++loop){
    cout << "At SLU=" << loop << " found " << qvecsAtSLU[loop].size() \
 << " \t rl vectors\n";
    cout << "these are:\n";
    for(vector<RLVecType>::const_iterator iter = qvecsAtSLU[loop].begin();
iter != qvecsAtSLU[loop].end(); ++iter){
      printRLVec(*iter);
    }
  }
  cout << "DEBUG: Leaving assign_half_qvectors_to_lengths()\n";
#endif
}// end assign_half_qvectors_to_lengths()

// create all vectors in the reciprocal lattice and store them by length
// measured in squared lattice units.
// this version takes the whole lattice.
void
assign_whole_qvectors_to_lengths(const short int maxIndex,
 vector<vector<RLVecType> >& qvecsAtSLU){
  const unsigned int maxIndexSq = static_cast<unsigned int>(maxIndex*maxIndex);
  // the length of a vector in squared lattice units
  unsigned int slu_length;
  // the vector in lattice units
  RLVecType rlVec(DIM);
#ifdef DEBUG_ASSIGN_QVECS_TO_LENGTH_ON
  cout << "DEBUG: Entering assign_whole_qvectors_to_lengths()\n";
#endif
  // check if dimension is right.  This function with hardcoded dimension
  // is not very elegant, but it works
  if(DIM != 3){
    cerr << "ERROR (assign_whole_qvectors_to_lengths): this code is meant for \
dimension 3.  Write another function for other dimensions.\n";
    exit (1);
  }
  // loop over full lattice
  for(short int xIndex = -maxIndex; xIndex <= maxIndex; ++xIndex)
    for(short int yIndex = -maxIndex; yIndex <= maxIndex; ++yIndex)
      for(short int zIndex = -maxIndex; zIndex <= maxIndex; ++zIndex){
// this is not the fastest way, but this procedure is run only once,
// so efficiency is not prime.
slu_length = \
  static_cast<unsigned int>(xIndex*xIndex + yIndex*yIndex \
    + zIndex*zIndex);
#ifdef DEBUG_ASSIGN_QVECS_TO_LENGTH_ON
cout << "DEBUG: xIndex=" << xIndex << " yIndex=" << yIndex \
     << " zIndex=" << zIndex << ", slu_length=" << slu_length <<"\n";
#endif
if(slu_length <= maxIndexSq){
  rlVec[0] = xIndex;
  rlVec[1] = yIndex;
  rlVec[2] = zIndex;
#ifdef DEBUG_ASSIGN_QVECS_TO_LENGTH_ON
  cout << "appending ";
  printRLVec(rlVec);
  cout << "to qvecsAtSLU[" << slu_length << "]\n";
#endif
  qvecsAtSLU[slu_length].push_back(rlVec);
}
      }
#ifdef DEBUG_ASSIGN_QVECS_TO_LENGTH_ON
  for(unsigned int loop = 0; loop <= maxIndexSq; ++loop){
    cout << "At SLU=" << loop << " found " << qvecsAtSLU[loop].size() \
 << " \t rl vectors\n";
    cout << "these are:\n";
    for(vector<RLVecType>::const_iterator iter = qvecsAtSLU[loop].begin();
iter != qvecsAtSLU[loop].end(); ++iter){
      printRLVec(*iter);
    }
  }
  cout << "DEBUG: Leaving assign_whole_qvectors_to_lengths()\n";
#endif
}// end assign_whole_qvectors_to_lengths()

// create all vectors in the reciprocal lattice and take the one in a certain
// length range, measured in reciprocal space length (not lattice units).
// This version takes the positive half lattice
void
get_qvectors_in_range(const short int maxIndex,
      const vector<double>& latticeUnits,
      const double rec_length_low,
      const double rec_length_high,
      vector<RSVecType>& qvecsInRSrange,
      double& averageQvalue){
#ifdef DEBUG_GET_QVECS_IN_RANGE_ON
  cout << "DEBUG: Entering get_qvectors_in_range()\n";
  cout << "DEBUG: rec_length_low=" << rec_length_low << ", rec_length_high="
       << rec_length_high << "\n";
#endif
  // check dimension
  if(DIM != 2 && DIM != 3){
    cerr << "ERROR (get_qvectors_in_range): DIM=" << DIM << "\n";
    exit (1);
  }
  // the vector in reciprocal space
  RSVecType rsVec(DIM);
  // the squared length of the reciprocal space vector
  double rec_length_sq, rec_length_low_sq, rec_length_high_sq;
  rec_length_low_sq  = SQUARE(rec_length_low);
  rec_length_high_sq = SQUARE(rec_length_high);
  short int zIndexLow, zIndexHigh;
  // initialize
  averageQvalue = 0.0;
  // loop over half the lattice
  for(short int xIndex=0; xIndex <= maxIndex; xIndex++){
    for(short int yIndex = (xIndex == 0) ? 0 : -maxIndex; \
yIndex <= maxIndex; yIndex++){
      if(DIM==3){
if(yIndex == 0 && xIndex == 0){
  zIndexLow = 0;
}else{
  zIndexLow = -maxIndex;
}
zIndexHigh = maxIndex;
      }else{
zIndexLow = zIndexHigh = 0;
      }
      // this loop is at zIndex = 0 for DIM==2.
      for(short int zIndex = zIndexLow; zIndex <= zIndexHigh; zIndex++){
// working with squares is faster
rec_length_sq = SQUARE(latticeUnits[0]) \
  * static_cast<double>(xIndex*xIndex) \
  + SQUARE(latticeUnits[1]) \
  * static_cast<double>(yIndex*yIndex)
  + SQUARE(latticeUnits[2]) \
  * static_cast<double>(zIndex*zIndex);
#ifdef DEBUG_GET_QVECS_IN_RANGE_ON
cout << "DEBUG: xIndex=" << xIndex << " yIndex=" << yIndex \
     << " zIndex=" << zIndex << ", rec_length=" \
     << sqrt(rec_length_sq) <<"\n";
#endif
// take this vector if in right range
if(rec_length_low_sq <= rec_length_sq \
   &&                   rec_length_sq <= rec_length_high_sq){
  // add up for the average length of the reciprocal vectors
  averageQvalue += sqrt(rec_length_sq);
  rsVec[0] = latticeUnits[0] * static_cast<double>(xIndex);
  rsVec[1] = latticeUnits[1] * static_cast<double>(yIndex);
  // in 2-d this entry does not exist
  if(DIM == 3){
    rsVec[2] = latticeUnits[2] * static_cast<double>(zIndex);
  }
  qvecsInRSrange.push_back(rsVec);
#ifdef DEBUG_GET_QVECS_IN_RANGE_ON
  cout << "DEBUG: Appended ";
  printRSVec(rsVec);
#endif
}
      }// end loop zIndex
    }// end loop yIndex
  }// end loop zIndex
  // what we found
  if(qvecsInRSrange.size() != 0){
    averageQvalue /= static_cast<double>(qvecsInRSrange.size());
    cout << "MESSAGE (get_qvectors_in_range): Found " \
 << qvecsInRSrange.size() << " q-vectors.\n";
    cout << "MESSAGE (get_qvectors_in_range): Average length of q-vectors is "\
 << averageQvalue << "\n";
  }else{
    cout << "WARNING (get_qvectors_in_range): Found no reciprocal vectors.\n";
  }
#ifdef DEBUG_GET_QVECS_IN_RANGE_ON
  cout << "DEBUG: Found the following RS vectors in range:\n";
  for(vector<RSVecType>::const_iterator iter = qvecsInRSrange.begin();
      iter != qvecsInRSrange.end(); ++iter){
      printRSVec(*iter);
  }
  cout << "DEBUG: Leaving get_qvectors_in_range()\n";
#endif
}// end get_qvectors_in_range()

// given a complete list of vectors in a list of reciprocal vectors, select
// a random subset (or take all)
unsigned int get_random_subset(const unsigned int number,
       const vector<RSVecType>& qvecs,
       vector<RSVecType>& selectedRSqvecs){
  unsigned int index;
  vector<RSVecType> qvecsCopy;
  vector<RSVecType>::pointer selectedPointer;
  // sanity check
  if(number <= 0){
    cerr << "ERROR (get_random_subset): Selection of " << number
 << " q-vectors requested\n";
    exit (2);
  }
  // if there are only a few vectors in the range, we take them all
  if(qvecs.size() <= number){
    selectedRSqvecs = qvecs;
    return selectedRSqvecs.size();
  }else{ // random selection
    // clear list
    selectedRSqvecs.clear();
    // make copy of list
    qvecsCopy = qvecs;
    for(unsigned int iLoop = 0; iLoop < number; ++iLoop){
      index = static_cast<unsigned int>( \
      static_cast<double>(qvecsCopy.size()) * genrand());
      // we need the pointer to erase the element
      selectedPointer = &qvecsCopy[index];
      selectedRSqvecs.push_back(*selectedPointer);
      // erase this vector, so that we don't select it again
      qvecsCopy.erase(selectedPointer);
    }
    if(selectedRSqvecs.size() != number){
      cerr << "ERROR (get_random_subset): selectedRSqvecs.size()="
  << selectedRSqvecs.size() << " != number=" << number << "\n";
      exit(2);
    }
    return number;
  }// if random selection
  // statement should never be reached, but compiler complaines
  // "control reaches end of non-void function" if there's no return here.
  return 0;
}// end get_random_subset()

// for MCT lattice indices qMod put together the list of slu values which
// belong to this range and vice versa.
void mct_to_slu(const double binWidth, const unsigned int nrBins, \
const double twoPiOverBoxlength,
const unsigned int maxIndexSq,
vector<vector<int> >& binToSLU,
vector<int>& SLUtoBin){
#ifdef DEBUG_MCT_TO_SLU_ON
  cout << "DEBUG: Entering mct_to_slu()\n";
#endif
  // upper and lower bound of reciprocal lattice vector (rl vector) bins
  double lower_bound, upper_bound;
  // we don't have to loop over loopSLU > maxIndexSq, as such vectors are too
  // long anyways
  int loopSLU = 0;
  for(unsigned int loopBin = 0; loopBin < nrBins \
&& loopSLU <= static_cast<int>(maxIndexSq); loopBin++){
```

```
    lower_bound = static_cast<double>(loopBin) * binWidth;
    upper_bound = static_cast<double>(loopBin + 1) * binWidth;
#ifdef DEBUG_MCT_TO_SLU_ON
    cout << "searching SLUs for bin index " << loopBin << " ("
 << lower_bound << " <= |q| < " << upper_bound
 << ")\n";
#endif
    // as long as sqrt(loopSLU) is in the range covered by this bin, add
    // this slu (length in squared lattice units) to the vector containing the
    // slu's for this bin.
    // here one could employ more sophisticated selection schemes.
    while(lower_bound <= twoPiOverBoxlength*sqrt(static_cast<double>(loopSLU))
  && twoPiOverBoxlength*sqrt(static_cast<double>(loopSLU))
  < upper_bound){
#ifdef DEBUG_MCT_TO_SLU_ON
      cout << "appending SLU=" << loopSLU << " to binToSLU[" << loopBin
   <<"] (|q|="
   << twoPiOverBoxlength*sqrt(static_cast<double>(loopSLU)) <<")\n";
#endif
      binToSLU[loopBin].push_back(loopSLU);
      loopSLU++;
    }
  }
#ifdef DEBUG_MCT_TO_SLU_ON
  for(unsigned int loopBin = 0; loopBin < nrBins; loopBin++){
    cout << "the following SLUs are stored in binToSLU[" << loopBin << "]:\n";
    for(vector<int>::const_iterator iter = binToSLU[loopBin].begin();
iter != binToSLU[loopBin].end(); ++iter)
      cout << *iter << " ";
    cout << "\n";
  }
#endif
  // the other way round
  // initialize to -INT_MAX
  // this means, that this SLU is not in any bin this is especially needed for
  // the lattice vectors of very big length.
  // This could also be needed for more complicated selection schemes (e.g. if
  // only a certain q-range is considered.
  for(vector<int>::iterator iter = SLUtoBin.begin(); \
      iter != SLUtoBin.end(); ++iter)
    *iter = -INT_MAX;
  for(int loopBin = 0; loopBin < static_cast<int>(nrBins); loopBin++)
    for(vector<int>::const_iterator iter = binToSLU[loopBin].begin();
iter != binToSLU[loopBin].end(); ++iter)
      SLUtoBin[*iter] = loopBin;
#ifdef DEBUG_MCT_TO_SLU_ON
  for(unsigned int loop = 0; loop <= DIM*maxIndexSq; ++loop)
    cout << "SLUtoBin["<< loop << "]=" << SLUtoBin[loop] << "\n";
#endif

#ifdef DEBUG_MCT_TO_SLU_ON
  cout << "DEBUG: Leaving mct_to_slu()\n";
#endif
}// end mct_to_slu()

// creates a list of all \vec{q} in a specified q-bin
void
create_complete_list(const int qBin,
     const vector<vector<int> >& binToSLU,
     const vector<vector<RLVecType> >& qvecsAtSLUQ,
     vector<RLVecType>& completeVecListQ){
  // sanity check
  if(qBin < 0){
    cerr << "ERROR (create_complete_list):  Encountered negative bin index"
 << "\n";
    exit (1);
  }
  // check if completeVecListQ is cleared
  if(completeVecListQ.size() != 0){
    cerr << "ERROR (create_complete_list):  completeVecListQ is not cleared"
 << "\n";
    exit (1);
  }
    // put together a list of all \vec{q} at this length
   for(vector<int>::const_iterator iterSLUQ = binToSLU[qBin].begin(); \
iterSLUQ != binToSLU[qBin].end(); ++iterSLUQ)
     completeVecListQ.insert(completeVecListQ.end(),
     qvecsAtSLUQ[*iterSLUQ].begin(),
     qvecsAtSLUQ[*iterSLUQ].end());
   // do a check
   // INDEX_MAX itself is used as a flag
   if(completeVecListQ.size() >= static_cast<unsigned int>(INDEX_MAX)){
     cerr << "ERROR: completeVecListQ.size()=" << completeVecListQ.size()
  << " >= INDEX_MAX=" << INDEX_MAX << "\n";
     cerr << "       change indexType!\n";
     exit (1);
   }
   // DEBUG
//    unsigned int nrAllQVecs = 0;
//    for(vector<int>::const_iterator iterSLU = binToSLU[qBin].begin();
//        iterSLU != binToSLU[qBin].end(); ++iterSLU)
//      nrAllQVecs += qvecsAtSLUQ[*iterSLU].size();
//    cout << "DEBUG (create_complete_list): nrAllQVecs = " <<  nrAllQVecs
//  << ", completeVecListQ.size()=" << completeVecListQ.size() << "\n";
}// end create_complete_list()

// for a given tupel (q,k) of real numbers denoting the moduli of lattice
// vectors (i.e. the (q,k) index of a bin), search tupels of lattice vectors
// (\vec{q}, \vec{k}) sorted into bins according to |p|, where
// \vec{p} = \vec{q} - \vec{k}
// This routine contributes significantly to the total CPU time needed when
// vector tupels are drawn anew for each configuration.  At present,
// the complexity scales like N^3 * log(N) which is much better than the N^4
// scaling in previous versions.
void
qkp_realization(const int qBin, const int kBin,
const unsigned int nrBins,
const unsigned int wishedEntries,
const double equalizationFactor,
const vector<RLVecType>& completeVecListQ,
const vector<RLVecType>& completeVecListK,
const unsigned int maxNumberAllK,
const vector<int>& SLUtoBin,
vector<vector<indexTupelType> >& qkpVecs,
vector<RLVecType>& uniqVecListK){
#ifdef DEBUG_QKP_REALIZATION_ON
  cout << "DEBUG: Entering qkp_realization() with qBin=" << qBin
       << ", kBin=" << kBin << "\n";
#endif
  // sanity check
  if(kBin < 0){
    cerr << "ERROR (qkp_realization): Encountered negative bin index" << "\n";
    exit (1);
  }
  if(uniqVecListK.size() != 0){
    cerr << "ERROR (qkp_realization): uniqVecListK was not cleared " << "\n";
    exit (1);
  }
  // numbers of all reciprocal lattice vectors belonging to a certain q or k
  unsignedIndexType nrAllQVecs, nrAllKVecs;
  // number of vector tupels we have to draw to fill all bins with
  // wishedEntries tupels
  unsigned int requiredTupels;
  // pVec = \vec{q} - \vec{k}
  RLVecType qVec(DIM), kVec(DIM), pVec(DIM);
  // p-bin index ranges
  int pBinRange, pBinRangeFull;
  int pBinMin, pBinMax;
  int binIndex;
  indexTupelType indexTupel;
  // for looping over \vec{q}s and \vec{k}s
  unsignedIndexType indexQ, indexK;
  // for converting indices in the complete vector list to unique vector
  // indices
  vector<unsignedIndexType> completeUniqConvertList(maxNumberAllK);
  // list of indices of completeVecListK occuring, will be uniquified
  vector<unsignedIndexType> uniqVecIndicesK(maxNumberAllK);
  uniqVecIndicesK.clear();
  // declaring these vectors<> static does not improve performance
  // get the number of available reciprocal lattice vectors at these lengths
  nrAllQVecs = static_cast<unsignedIndexType>(completeVecListQ.size());
  nrAllKVecs = static_cast<unsignedIndexType>(completeVecListK.size());
  // \vec{p} = \vec{q} - \vec{k}
  // due to the triangle inequality we have
  // |q-k| <= p <= q+k
  // we have thus expect a range of
  // q+k - |q-k| == q+k-q+2 = 2k || q+k+q-k = 2q
  // for p.
  // Because the bins are equidistant, the same relation can be written for the
  // bin indices, with the addition that the finite width of the bins is taken
  // into account.
  // Note, we are dealing with indices.
  // the number of bins in which we can find \vec{p}s not taking into account
  // that we cut off for bins with index >= nrBins
  pBinRangeFull = qBin+kBin +1 - max(abs(qBin-kBin)-1, 0) +1;
  // minimal and maximal index of p-bins which can be filled
#ifdef TAKE_OUTER_BINS_YES
  pBinMin = max(abs(qBin-kBin)-1, 0);
  pBinMax = min(qBin+kBin+1, static_cast<int>(nrBins-1));
#else
  // unless at 0 or nrBins-1, pBinMin and pBinMax are +1 resp. -1 of the
  // values above
  pBinMin = abs(qBin-kBin);
  pBinMax = qBin + kBin;
  if(pBinMax > static_cast<int>(nrBins) -1)
    pBinMax = nrBins -1;
#endif
  // number of p-bins which can contain results, which are really used
  pBinRange = pBinMax - pBinMin +1;
#ifdef DEBUG_QKP_REALIZATION_COUNT_ON
  cout << "DEBUG: qBin=" << qBin << ", kBin=" << kBin << " : nrAllQVecs= "
       << nrAllQVecs << ", nrAllKVecs= " <<  nrAllKVecs << ", #tupels= "
       << nrAllQVecs*nrAllKVecs << "\n";
  cout << "DEBUG: pBinMin=" << pBinMin << ", pBinMax=" << pBinMax
       << ", pBinRange=" << pBinRange << ", pBinRangeFull="
       << pBinRangeFull << "\n";
#endif
  // we want a certain number of vector tupels in each bin, so
  requiredTupels = wishedEntries * static_cast<unsigned int>(pBinRangeFull);
  // depending on the numbers, decide if we take all possible vector tupels
  // (\vec{q}, \vec{k}) or if a random selection scheme is applied
  if(nrAllQVecs * nrAllKVecs == 0){
#ifdef DEBUG_QKP_REALIZATION_ON
    // this can happen, we issue a message
    cout << "DEBUG: nrAllQVecs * nrAllKVecs == 0 for qBin=" << qBin
 << ", kBin=" << kBin << "\n";
#endif
  }else if(static_cast<unsigned int>(nrAllQVecs) \
   * static_cast<unsigned int>(nrAllKVecs) <= 2*requiredTupels){
    // it is not very efficient to draw randomly if nrAllQVecs * nrAllKVecs
    // is only slightly bigger than requiredTupels.
#ifdef DEBUG_QKP_REALIZATION_ON
    cout << "DEBUG: Taking all vector tupels\n";
#endif
    // first loop over all \vec{q}
    for(indexQ = 0; indexQ < nrAllQVecs; ++indexQ){
      qVec = completeVecListQ[indexQ];
      // then loop over all \vec{k}
      for(indexK = 0; indexK < nrAllKVecs; ++indexK){
kVec = completeVecListK[indexK];
    // calculate \vec{p} = \vec{q} - \vec{k}
pVec = diffvec(qVec, kVec);
#ifdef DEBUG_QKP_REALIZATION_VERBOSE_ON
cout << "DEBUG: printing (\\vec{q}, \\vec{k}, \\vec{p}):\n";
printRLVec(qVec);
printRLVec(kVec);
printRLVec(pVec);
#endif
binIndex = SLUtoBin[mod_square_slu(pVec)];
// if binIndex is in the valid range
// the condition binIndex <= pBinMax will be more often violated
// so we test it first for efficiency
if(binIndex <= pBinMax && binIndex >= pBinMin){
  // we have to take all vectors in order not to prefer a certain
  // region on the sphere, unlike random selection.
  // memorize only the indices of the vectors which make up the
  // tupel
  indexTupel.q = indexQ;
  indexTupel.k = indexK;
  qkpVecs[binIndex].push_back(indexTupel);
#ifdef DEBUG_QKP_REALIZATION_VERBOSE_ON
  cout << "DEBUG: appended (" << indexTupel.q << ","
       << indexTupel.k << ") to qkpVecs[" <<  binIndex << "]\n";
  cout << "qVec = ";
  printRLVec(completeVecListQ[indexTupel.q]);
  cout << "kVec = ";
  printRLVec(completeVecListK[indexTupel.k]);
#endif
}// if in right range
      }// end loop over \vec{k}
    }// end loop over \vec{q}
    // if we took all vectors, then the uniqVecListK is identical to
    // completeVecListK and we don't have to change indices
    uniqVecListK = completeVecListK;
  }else{
    // we select randomly
#ifdef DEBUG_QKP_REALIZATION_RANDOM_ON
    cout << "DEBUG: Random selection of vector tupels\n";
#endif
    // some values for |p| will be less often realized than others, so some
    // p-bins will be filled less than wishedEntries.  With drawing more
    // vector tupels by a factor of equalizationFactor we try to minimize this
    // unequality of entry numbers to achive a uniform filling of the bins.
    for(unsigned int loop = 0; \
loop < static_cast<unsigned int>\
```

```
  (static_cast<double>(requiredTupels)*equalizationFactor)+1; \
++loop){
      // select randomly a rl vector \vec{q}
      /////////////////////////////////////
      // indexQ is between 0 and nrAllQVecs-1 (C-indices !)
      // genrand() returns a number in [0, 1) (1 is exluded)
      indexQ = static_cast<int>(static_cast<double>(nrAllQVecs) * genrand());
      // this is our \vec{q}
      qVec = completeVecListQ[indexQ];
      // select randomly a rl vector \vec{k}
      /////////////////////////////////////
      indexK = static_cast<int>(static_cast<double>(nrAllKVecs) * genrand());
      // this is our \vec{k}
      kVec = completeVecListK[indexK];
      // calculate \vec{p} = \vec{q} - \vec{k}
      pVec = diffvec(qVec, kVec);
      binIndex = SLUtoBin[mod_square_slu(pVec)];
      // if binIndex is in the valid range
      if(binIndex <= pBinMax && binIndex >= pBinMin){
// if we don't already have enough vector tupels
// get index of \vec{q}, and append \vec{k} if we didn't have it before
// memorize only the indices of the vectors which make up the tupel
if(qkpVecs[binIndex].size() < wishedEntries){
  indexTupel.q = indexQ;
  indexTupel.k = indexK;
  uniqVecIndicesK.push_back(indexK);
  qkpVecs[binIndex].push_back(indexTupel);
}
      }// if in right range
    }// end loop over randomly selected tupels
    // create uniquified list of \vec{k} indices and convert the indices in
    // qkpVecs[] accordingly
    ////////////////////////////////////////////////////////////////////////////
    // the reason for first creating a list of all occuring indices and then
    // uniquifying it is performance.  sort() scales like N*log(N)
    // (if the STL uses a good algorithm, and if a lot of indices are
    // identical, the performance might e even better), if we created
    // a uniquified list each time we find a new tupel, we have N*N operations.
    // for N of the order of maxNumberAllK the difference is quite significant.
    // Besides, here we perform sorting and comparing of integers, not
    // of vectors composed of DIM integers.
#ifdef DEBUG_QKP_REALIZATION_ON
    cout << "DEBUG: before uniquifying: uniqVecIndicesK.size()="
 << uniqVecIndicesK.size() << "\n";
#endif
    sort(uniqVecIndicesK.begin(), uniqVecIndicesK.end());
    uniqVecIndicesK.erase(unique(uniqVecIndicesK.begin(),
        uniqVecIndicesK.end()),
  uniqVecIndicesK.end());
#ifdef DEBUG_QKP_REALIZATION_ON
    cout << "DEBUG: after uniquifying: uniqVecIndicesK.size()="
 << uniqVecIndicesK.size() << "\n";
    // initialize completeUniqConvertList.  Not necessary, but helps catching
    // bugs
    for(unsigned int loopCUCL = 0; loopCUCL < maxNumberAllK; ++loopCUCL)
      completeUniqConvertList[loopCUCL] = INDEX_MAX;
#endif
    for(unsignedIndexType uniqIndex = 0;
uniqIndex < static_cast<unsignedIndexType>(uniqVecIndicesK.size());
++uniqIndex){
      // build up uniquified list of \vec{k}
      uniqVecListK.push_back(completeVecListK[uniqIndex]);
      // map the complete indices on the unique indices
      completeUniqConvertList[uniqVecIndicesK[uniqIndex]] = uniqIndex;
    }
    // convert the indices for \vec{k} stored in qkpVecs[binIndex] to unique
    // indices
    for(binIndex = pBinMin; binIndex <= pBinMax; ++binIndex)
      for(vector<indexTupelType>::iterator \
    iterIndexTupelVector = qkpVecs[binIndex].begin();
  iterIndexTupelVector != qkpVecs[binIndex].end();
  ++iterIndexTupelVector){
#ifdef DEBUG_QKP_REALIZATION_ON
if(completeUniqConvertList[iterIndexTupelVector->k] == INDEX_MAX){
  cerr  << "ERROR: completeUniqConvertList["
<< iterIndexTupelVector->k << "] == INDEX_MAX" << "\n";
  exit(1);
}
#endif
iterIndexTupelVector->k \
  = completeUniqConvertList[iterIndexTupelVector->k];
      }// loop over all index tupels in a p-bin
  }// decide if random selection

#ifdef DEBUG_QKP_REALIZATION_ON
  cout << "DEBUG: number of reciprocal lattice vector tupels\
  in each p-Bin:\n";
  unsigned int nrFilledBins = 0;
  for(int loop = pBinMin; loop <= pBinMax; ++loop){
    if(qkpVecs[loop].size() > 0)
      ++nrFilledBins;
    cout << "qkpVecs[" << loop << "].size()=" << qkpVecs[loop].size() << "\n";
    printIndexTupelList(qkpVecs[loop]);
  }
  cout << "pBinRange=" << pBinRange << ", nrFilledBins=" << nrFilledBins
       <<"\n";
  cout << "pBinMin=" << pBinMin << ", pBinMax=" << pBinMax << "\n";
//   cout << "completeVecListQ.size()=" << completeVecListQ.size() <<"\n";
//   printRLVecList(completeVecListQ);
//   cout << "completeVecListK.size()=" << completeVecListK.size() <<"\n";
//   printRLVecList(completeVecListK);
//   cout << "\n";
#endif
#ifdef DEBUG_QKP_REALIZATION_ON
  cout << "DEBUG: Leaving qkp_realization()\n";
#endif
}// end qkp_realization()

// calculate the real part of S_3 for chain and melt.  Normalization is done at
// the end in the main program.
// The definition used is
// S_3(\vec{q}, \vec{k}) = \sum_i \sum_j \sum_l
//   e^i\vec{k}\cdot\vec{r}_i e^i\vec{p}\cdot\vec{r}_j
//    e^-i\vec{q}\cdot\vec{r}_i
//
// According to gprof this is the most CPU-intensive function using more than
// 90% of the CPU time, because it is called so often :
// O(nrBins^3 * wishedEntries) times for each configuration.
// The inline keyword might help the compiler.
void
calc_re_s3(const calculType * const sinesQ, \
   const calculType * const cosinesQ, \
   const calculType * const sinesK, \
   const calculType * const cosinesK,
   const unsigned int mon_per_chain, \
   const unsigned int nr_chains,\
   sums3Type * const s3chain, sums3Type * const s3melt){
#ifdef DEBUG_CALC_RE_S3_ON
  cout << "DEBUG: Entering calc_re_s3()\n";
#endif
  // it is probably better to use higher precision when adding up positive and
  // negative numbers.
  sums3Type sumSinQchain, sumCosQchain, sumSinQmelt, sumCosQmelt, \
    sumSinKchain, sumCosKchain, sumSinKmelt, sumCosKmelt;
  sums3Type \
    sumSinQSinKchain, sumSinQCosKchain, sumCosQSinKchain, sumCosQCosKchain, \
    sumSinQSinKmelt, sumSinQCosKmelt, sumCosQSinKmelt, sumCosQCosKmelt;
#ifdef SAVE_S3_INTERMEDIATE_CHAIN_SUMS_ON
  sums3Type \
    sumCosQchainSumCosKchain, sumCosQchainSumSinKchain, \
    sumSinQchainSumSinKchain, sumSinQchainSumCosKchain;
#endif
#ifdef SAVE_S3_INTERMEDIATE_MELT_SUMS_ON
  // introduction of intermediate sums for the melt does not lead to improved
  // performance, but that might depend on the CPU.
  sums3Type \
    sumCosQmeltSumCosKmelt, sumCosQmeltSumSinKmelt, \
    sumSinQmeltSumSinKmelt, sumSinQmeltSumCosKmelt;
#endif
  // initialize
  sumSinQmelt = 0.0;
  sumCosQmelt = 0.0;
  sumSinKmelt = 0.0;
  sumCosKmelt = 0.0;
  sumSinQSinKmelt = 0.0;
  sumSinQCosKmelt = 0.0;
  sumCosQSinKmelt = 0.0;
  sumCosQCosKmelt = 0.0;
  // loop over all chains
  unsigned int monomer = 0;
  unsigned int chainEnd = 0;
  for(unsigned int chain = 0; chain < nr_chains; ++chain){
    // reset chain sums
    sumSinQchain = 0.0;
    sumCosQchain = 0.0;
    sumSinKchain = 0.0;
    sumCosKchain = 0.0;
    sumSinQSinKchain = 0.0;
    sumSinQCosKchain = 0.0;
    sumCosQSinKchain = 0.0;
    sumCosQCosKchain = 0.0;
    // loop over all monomers in a chain
    for(chainEnd += mon_per_chain; monomer < chainEnd; ++monomer){
      sumSinQchain += sinesQ[monomer];
      sumCosQchain += cosinesQ[monomer];
      sumSinKchain += sinesK[monomer];
      sumCosKchain += cosinesK[monomer];
      sumSinQSinKchain += sinesQ[monomer]   * sinesK[monomer];
      sumSinQCosKchain += sinesQ[monomer]   * cosinesK[monomer];
      sumCosQSinKchain += cosinesQ[monomer] * sinesK[monomer];
      sumCosQCosKchain += cosinesQ[monomer] * cosinesK[monomer];
    }
#ifdef SAVE_S3_INTERMEDIATE_CHAIN_SUMS_ON
    // reduce number of multiplications by saving these products
    sumCosQchainSumCosKchain = sumCosQchain * sumCosKchain;
    sumCosQchainSumSinKchain = sumCosQchain * sumSinKchain;
    sumSinQchainSumSinKchain = sumSinQchain * sumSinKchain;
    sumSinQchainSumCosKchain = sumSinQchain * sumCosKchain;
    // calculate for this chain
    *s3chain += \
       sumCosQchainSumCosKchain * sumCosQCosKchain \
      + sumCosQchainSumSinKchain * sumCosQSinKchain \
      + sumSinQchainSumSinKchain * sumCosQCosKchain \
      - sumSinQchainSumCosKchain * sumCosQSinKchain \
      - sumCosQchainSumSinKchain * sumSinQCosKchain \
      + sumCosQchainSumCosKchain * sumSinQSinKchain \
      + sumSinQchainSumCosKchain * sumSinQCosKchain \
      + sumSinQchainSumSinKchain * sumSinQSinKchain;
#else
    *s3chain += \
       sumCosQchain * sumCosKchain * sumCosQCosKchain \
      + sumCosQchain * sumSinKchain * sumCosQSinKchain \
      + sumSinQchain * sumSinKchain * sumCosQCosKchain \
      - sumSinQchain * sumCosKchain * sumCosQSinKchain \
      - sumCosQchain * sumSinKchain * sumSinQCosKchain \
      + sumCosQchain * sumCosKchain * sumSinQSinKchain \
      + sumSinQchain * sumCosKchain * sumSinQCosKchain \
      + sumSinQchain * sumSinKchain * sumSinQSinKchain;
#endif
#ifdef DEBUG_CALC_RE_S3_ON
    cout << "DEBUG: s3chain = " << *s3chain << "\n";
#endif
    // sum up for the melt
    sumSinQmelt += sumSinQchain;
    sumCosQmelt += sumCosQchain;
    sumSinKmelt += sumSinKchain;
    sumCosKmelt += sumCosKchain;
    sumSinQSinKmelt += sumSinQSinKchain;
    sumSinQCosKmelt += sumSinQCosKchain;
    sumCosQSinKmelt += sumCosQSinKchain;
    sumCosQCosKmelt += sumCosQCosKchain;
  }// end loop over all chains

#ifdef SAVE_S3_INTERMEDIATE_MELT_SUMS_ON
    // reduce number of multiplications by saving these products
    sumCosQmeltSumCosKmelt = sumCosQmelt * sumCosKmelt;
    sumCosQmeltSumSinKmelt = sumCosQmelt * sumSinKmelt;
    sumSinQmeltSumSinKmelt = sumSinQmelt * sumSinKmelt;
    sumSinQmeltSumCosKmelt = sumSinQmelt * sumCosKmelt;
    // calculate for melt
    *s3melt += \
       sumCosQmeltSumCosKmelt * sumCosQCosKmelt \
      + sumCosQmeltSumSinKmelt * sumCosQSinKmelt \
      + sumSinQmeltSumSinKmelt * sumCosQCosKmelt \
      - sumSinQmeltSumCosKmelt * sumCosQSinKmelt \
      - sumCosQmeltSumSinKmelt * sumSinQCosKmelt \
      + sumCosQmeltSumCosKmelt * sumSinQSinKmelt \
      + sumSinQmeltSumCosKmelt * sumSinQCosKmelt \
      + sumSinQmeltSumSinKmelt * sumSinQSinKmelt;
#else
    *s3melt += \
       sumCosQmelt * sumCosKmelt * sumCosQCosKmelt \
      + sumCosQmelt * sumSinKmelt * sumCosQSinKmelt \
      + sumSinQmelt * sumSinKmelt * sumCosQCosKmelt \
      - sumSinQmelt * sumCosKmelt * sumCosQSinKmelt \
      - sumCosQmelt * sumSinKmelt * sumSinQCosKmelt \
      + sumCosQmelt * sumCosKmelt * sumSinQSinKmelt \
      + sumSinQmelt * sumCosKmelt * sumSinQCosKmelt \
      + sumSinQmelt * sumSinKmelt * sumSinQSinKmelt;
#endif

#ifdef DEBUG_CALC_RE_S3_ON
  cout << "DEBUG: s3melt = " << *s3melt << "\n";
#endif

#ifdef DEBUG_CALC_RE_S3_ON
  cout << "DEBUG: Leaving calc_re_s3()\n";
#endif
}// end calc_re_s3()

// calculate S_3 for chain and melt.
// this function evaluates real and imaginary part.  This is a good test for
// the statistics, as the imaginary part should vanish in the statistical
// mean.
```

```
// The definition used is
// S_3(\vec{q}, \vec{k}) = \sum_i \sum_j \sum_l
//   e^i\vec{k}\cdot\vec{r}_i e^i\vec{p}\cdot\vec{r}_j
//    e^-i\vec{q}\cdot\vec{r}_i
void
calc_cplx_s3(const calculType * const  sinesQ, \
     const calculType * const  cosinesQ, \
     const calculType * const  sinesK, \
     const calculType * const  cosinesK, \
     const unsigned int mon_per_chain, \
     const unsigned int nr_chains, \
     sums3Type * const s3chain_re, sums3Type * const s3melt_re, \
     sums3Type * const s3chain_im, sums3Type * const s3melt_im){
#ifdef DEBUG_CALC_CPLX_S3_ON
  cout << "DEBUG: Entering calc_cplx_s3()\n";
#endif
  sums3Type sumSinQchain, sumCosQchain, sumSinQmelt, sumCosQmelt, \
    sumSinKchain, sumCosKchain, sumSinKmelt, sumCosKmelt;
  sums3Type \
    sumSinQSinKchain, sumSinQCosKchain, sumCosQSinKchain, sumCosQCosKchain, \
    sumSinQSinKmelt, sumSinQCosKmelt, sumCosQSinKmelt, sumCosQCosKmelt;
  // initialize
  sumSinQmelt = 0.0;
  sumCosQmelt = 0.0;
  sumSinKmelt = 0.0;
  sumCosKmelt = 0.0;
  sumSinQSinKmelt = 0.0;
  sumSinQCosKmelt = 0.0;
  sumCosQSinKmelt = 0.0;
  sumCosQCosKmelt = 0.0;
  // loop over all chains
  unsigned int monomer = 0;
  unsigned int chainEnd = 0;
  for(unsigned int chain = 0; chain < nr_chains; ++chain){
    // reset chain sums
    sumSinQchain = 0.0;
    sumCosQchain = 0.0;
    sumSinKchain = 0.0;
    sumCosKchain = 0.0;
    sumSinQSinKchain = 0.0;
    sumSinQCosKchain = 0.0;
    sumCosQSinKchain = 0.0;
    sumCosQCosKchain = 0.0;
    // loop over all monomers in a chain
    for(chainEnd += mon_per_chain; monomer < chainEnd; ++monomer){
      sumSinQchain += sinesQ[monomer];
      sumCosQchain += cosinesQ[monomer];
      sumSinKchain += sinesK[monomer];
      sumCosKchain += cosinesK[monomer];
      sumSinQSinKchain += sinesQ[monomer]   * sinesK[monomer];
      sumSinQCosKchain += sinesQ[monomer]   * cosinesK[monomer];
      sumCosQSinKchain += cosinesQ[monomer] * sinesK[monomer];
      sumCosQCosKchain += cosinesQ[monomer] * cosinesK[monomer];
    }
    // calculate for this chain
    *s3chain_re += \
        sumCosQchain * sumCosKchain * sumCosQCosKchain \
      + sumCosQchain * sumSinKchain * sumCosQSinKchain \
      + sumSinQchain * sumSinKchain * sumCosQCosKchain \
      - sumSinQchain * sumCosKchain * sumCosQSinKchain \
      - sumCosQchain * sumSinKchain * sumSinQCosKchain \
      + sumCosQchain * sumCosKchain * sumSinQSinKchain \
      + sumSinQchain * sumCosKchain * sumSinQCosKchain \
      + sumSinQchain * sumSinKchain * sumSinQSinKchain;
    *s3chain_im += \
        sumCosQchain * sumSinKchain * sumCosQCosKchain \
      - sumCosQchain * sumCosKchain * sumCosQSinKchain \
      - sumSinQchain * sumCosKchain * sumCosQCosKchain \
      - sumSinQchain * sumSinKchain * sumCosQSinKchain \
      + sumCosQchain * sumCosKchain * sumSinQCosKchain \
      + sumCosQchain * sumSinKchain * sumSinQSinKchain \
      + sumSinQchain * sumSinKchain * sumSinQCosKchain \
      - sumSinQchain * sumCosKchain * sumSinQSinKchain;

#ifdef DEBUG_CALC_CPLX_S3_ON
    cout << "DEBUG: s3chain = (" << *s3chain_re << " +i " << *s3chain_im
 << ")\n";
#endif
    // sum up for the melt
    sumSinQmelt += sumSinQchain;
    sumCosQmelt += sumCosQchain;
    sumSinKmelt += sumSinKchain;
    sumCosKmelt += sumCosKchain;
    sumSinQSinKmelt += sumSinQSinKchain;
    sumSinQCosKmelt += sumSinQCosKchain;
    sumCosQSinKmelt += sumCosQSinKchain;
    sumCosQCosKmelt += sumCosQCosKchain;
  }// end loop over all chains
  // calculate for melt
  *s3melt_re += \
    sumCosQmelt * sumCosKmelt * sumCosQCosKmelt \
    + sumCosQmelt * sumSinKmelt * sumCosQSinKmelt \
    + sumSinQmelt * sumSinKmelt * sumCosQCosKmelt \
    - sumSinQmelt * sumCosKmelt * sumCosQSinKmelt \
    - sumCosQmelt * sumSinKmelt * sumSinQCosKmelt \
    + sumCosQmelt * sumCosKmelt * sumSinQSinKmelt \
    + sumSinQmelt * sumCosKmelt * sumSinQCosKmelt \
    + sumSinQmelt * sumSinKmelt * sumSinQSinKmelt;
  *s3melt_im += \
    sumCosQmelt * sumSinKmelt * sumCosQCosKmelt \
    - sumCosQmelt * sumCosKmelt * sumCosQSinKmelt \
    - sumSinQmelt * sumCosKmelt * sumCosQCosKmelt \
    - sumSinQmelt * sumSinKmelt * sumCosQSinKmelt \
    + sumCosQmelt * sumCosKmelt * sumSinQCosKmelt \
    + sumCosQmelt * sumSinKmelt * sumSinQSinKmelt \
    + sumSinQmelt * sumSinKmelt * sumSinQCosKmelt \
    - sumSinQmelt * sumCosKmelt * sumSinQSinKmelt;
#ifdef DEBUG_CALC_CPLX_S3_ON
  cout << "DEBUG: s3melt = (" << *s3melt_re << " +i " << *s3melt_im
    << ")\n";
#endif

#ifdef DEBUG_CALC_CPLX_S3_ON
  cout << "DEBUG: Leaving calc_cplx_s3()\n";
#endif
}// end calc_cplx_s3()

// calculate sines and cosines for all \vec{q} in completeVecListLattice
void
calc_all_sines_cosines(const unsigned int nr_monomers,
       const r3vector * const configuration,
       const unsigned int numberAllQ,
       const vector<r3vector> &completeVecListLattice,
       vector<calculType* >& allSinesQ,
       vector<calculType* >& allCosinesQ){
#ifdef USE_REGISTER_QINR_ON
  register calculType QinR;
#else
  calculType QinR;
#endif
  // check if more storage is needed
  if(allSinesQ.size() < numberAllQ){
  if(allSinesQ.size() != allCosinesQ.size()){
    cerr << "ERROR (calc_all_sines_cosines): allSinesQ.size() != \
allCosinesQ.size()\n";
    exit(1);
  }
#ifdef DEBUG_QKP_CALC_S3_ON
    cout << "DEBUG: Allocate new memory for allSinesQ and allCosinesQ\n";
    cout << "DEBUG: allSines.size()=" << allSinesQ.size() << ", "
 << "numberAllQ=" << numberAllQ << "\n";
#endif
    unsigned int requiredArraysNumber = numberAllQ - allSinesQ.size();
    for(unsigned int loop = 0; loop < requiredArraysNumber; ++loop){
      allSinesQ.push_back((calculType*) \
  malloc((size_t)(nr_monomers*sizeof(calculType))));
      allCosinesQ.push_back((calculType*) \
    malloc((size_t)(nr_monomers*sizeof(calculType))));
    }
#ifdef DEBUG_QKP_CALC_S3_ON
    cout << "DEBUG: Allocated new memory for allSinesQ and allCosinesQ\n";
    cout << "DEBUG: allSines.size()=" << allSinesQ.size() << ", "
 << "numberAllQ=" << numberAllQ << "\n";
#endif
    cout << "MESSAGE: New size of allSinesQ and allCosinesQ = "
 << allSinesQ.size()
 << "\n";
  }
  // now calculate the sines and cosines
  // first for \vec{q}
  for(unsigned IndexType completeVecIndex = 0;
      completeVecIndex < static_cast<unsignedIndexType>(numberAllQ);
      ++completeVecIndex){
      // loop over all particles
      for(unsigned int partLoop = 0; partLoop < nr_monomers; ++partLoop){
QinR = static_cast<calculType>\
  (SCALARPRODUCT(completeVecListLattice[completeVecIndex], \
 configuration[partLoop]));
allSinesQ[completeVecIndex][partLoop]   = sin(QinR);
allCosinesQ[completeVecIndex][partLoop] = cos(QinR);
      }// end loop over all particles
  }// end loop over completeVecIndex
}// end calc_all_sines_cosines()

// calculate S_3 for the melt and the chain after calculation of the sines and
// cosines for \vec{k}s.
void
qkp_calc_s3(const unsigned int nrBins,
    const vector<vector<indexTupelType> >& indexTupels_at_qkp,
    const unsigned int numberUVLK,
    const vector<r3vector>& uniqVecListLatticeK,
    const vector<calculType* >& allSinesQ,
    const vector<calculType* >& allCosinesQ,
    vector<calculType* >& allSinesK,
    vector<calculType* >& allCosinesK,
    const r3vector * const configuration,
    const unsigned int mon_per_chain,
    const unsigned int nr_chains,
    sums3Type * const s3_re_chain_qk_at_all_p,
    sums3Type * const s3_re_melt_qk_at_all_p
#ifdef CALC_COMPLEX_YES
    , sums3Type * const s3_im_chain_qk_at_all_p,
    sums3Type * const s3_im_melt_qk_at_all_p
#endif
    ){
#ifdef DEBUG_QKP_CALC_S3_ON
  cout << "DEBUG: Entering qkp_calc_s3()\n";
#endif
  const unsigned int nr_monomers = mon_per_chain * nr_chains;
  unsigned int partLoop;
  unsignedIndexType uniqVecIndexK;
  unsigned int requiredArraysNumber;
#ifdef USE_REGISTER_QINR_ON
  register calculType QinR;
#else
  calculType QinR;
#endif
  // final pointers which will be handed over
  calculType *finalPointerToSinQ, *finalPointerToCosQ, \
    *finalPointerToSinK, *finalPointerToCosK;
  if(allSinesK.size() != allCosinesK.size()){
    cerr << "ERROR (qkp_calc_s3): allSinesK.size() != allCosinesK.size() \n";
    exit(1);
  }
  // reset intermediate result arrays
  for(unsigned int loop = 0; loop < nrBins; ++loop){
    s3_re_chain_qk_at_all_p[loop] = 0.0;
    s3_re_melt_qk_at_all_p[loop] = 0.0;
#ifdef CALC_COMPLEX_YES
    s3_im_chain_qk_at_all_p[loop] = 0.0;
    s3_im_melt_qk_at_all_p[loop] = 0.0;
#endif
  }
  // if allocated memory for the sines and cosines is not sufficient,
  // allocate some more for the \vec{k}s
  if(allSinesK.size() < numberUVLK){
#ifdef DEBUG_QKP_CALC_S3_ON
    cout << "DEBUG: Allocate new memory for allSinesK and allCosinesK\n";
    cout << "DEBUG: allSines.size()=" << allSinesK.size() << ", "
 << "numberUVLK=" << numberUVLK << "\n";
#endif
    requiredArraysNumber = numberUVLK - allSinesK.size();
    for(unsigned int loop = 0; loop < requiredArraysNumber; ++loop){
      allSinesK.push_back((calculType*) \
  malloc((size_t)(nr_monomers*sizeof(calculType))));
      allCosinesK.push_back((calculType*) \
    malloc((size_t)(nr_monomers*sizeof(calculType))));
    }
#ifdef DEBUG_QKP_CALC_S3_ON
    cout << "DEBUG: Allocated new memory for allSinesK and allCosinesK\n";
    cout << "DEBUG: allSines.size()=" << allSinesK.size() << ", "
 << "numberUVLK=" << numberUVLK << "\n";
#endif
    cout << "MESSAGE: New size of allSinesK and allCosinesK = "
 << allSinesK.size()
 << "\n";
  }
#ifdef DEBUG_QKP_CALC_S3_ON
  cout << "DEBUG: Begin calculating sines and cosines\n";
#endif
  // sines and cosines for \vec{k}
  for(uniqVecIndexK = 0;
      uniqVecIndexK < static_cast<unsignedIndexType>(numberUVLK);
      ++uniqVecIndexK){
      // loop over all particles
      for(partLoop = 0; partLoop < nr_monomers; ++partLoop){
QinR = static_cast<calculType>\
  SCALARPRODUCT(uniqVecListLatticeK[uniqVecIndexK], \
configuration[partLoop]);
allSinesK[uniqVecIndexK][partLoop]   = sin(QinR);
allCosinesK[uniqVecIndexK][partLoop] = cos(QinR);
      }// end loop over all particles
```

```
  }// end loop over uniqVecIndexK
  // now put together the S_3(q, k, p_min,..., p_max)
  for(unsigned int pBin = 0; pBin < nrBins; ++pBin){
    for(vector<indexTupelType>::const_iterator \
  iterTupel = indexTupels_at_qkp[pBin].begin();
iterTupel != indexTupels_at_qkp[pBin].end(); ++iterTupel){
      // get the pointers to the sines and cosines
      finalPointerToSinQ = allSinesQ[iterTupel->q];
      finalPointerToCosQ = allCosinesQ[iterTupel->q];
      finalPointerToSinK = allSinesK[iterTupel->k];
      finalPointerToCosK = allCosinesK[iterTupel->k];
#ifdef DEBUG_QKP_CALC_S3_ON
      // check for \vec{q}
      if(finalPointerToSinQ == 0){
cerr << "ERROR: finalPointerToSinQ == 0\n";
exit (1);
      }
      // check for \vec{k}
      if(finalPointerToSinK == 0){
cerr << "ERROR: finalPointerToSinK == 0\n";
exit(1);
      }
#endif
      // now we have four pointers to the approbriate arrays containing
      // the sines and cosines
#ifdef CALC_COMPLEX_NO
      calc_re_s3(finalPointerToSinQ, finalPointerToCosQ, \
 finalPointerToSinK, finalPointerToCosK, \
 mon_per_chain, nr_chains, \
 s3_re_chain_qk_at_all_p+pBin, \
 s3_re_melt_qk_at_all_p+pBin);
#endif
#ifdef CALC_COMPLEX_YES
      calc_cplx_s3(finalPointerToSinQ, finalPointerToCosQ, \
   finalPointerToSinK, finalPointerToCosK, \
   mon_per_chain, nr_chains, \
   s3_re_chain_qk_at_all_p+pBin, \
   s3_re_melt_qk_at_all_p+pBin, \
   s3_im_chain_qk_at_all_p+pBin, \
   s3_im_melt_qk_at_all_p+pBin);
#endif
    }// loop over all tupels in one p-bin
  }// end loop over pBin
#ifdef DEBUG_QKP_CALC_S3_ON
  cout << "DEBUG: Leaving qkp_calc_s3()\n";
#endif
}// end qkp_calc_s3()

// rescales all lattice vectors found with the lattice constant
void
rescale_lattice_vectors(const double twoPiOverBoxlength,
vector<RLVecType>& uniqVecList,
vector<r3vector>& uniqVecListLattice){
  unsigned int loopUVL;
  for(loopUVL = 0; loopUVL < uniqVecList.size(); ++loopUVL){
uniqVecListLattice[loopUVL].x = twoPiOverBoxlength \
  * static_cast<double>(uniqVecList[loopUVL][0]);
uniqVecListLattice[loopUVL].y = twoPiOverBoxlength \
  * static_cast<double>(uniqVecList[loopUVL][1]);
uniqVecListLattice[loopUVL].z = twoPiOverBoxlength \
  * static_cast<double>(uniqVecList[loopUVL][2]);
  }
}// rescale_lattice_vectors()

// gets a uniquified list of RLVecs and returns the unique index of a RLVec.
// if RLVec does not exist, append it to the list.  We also keep track of
// indices which occur more than once.
inline indexType
uniq_vec_index(vector<RLVecType>& uniqVecList,
       vector<unsignedIndexType>& frequentIndexList,
       const RLVecType& rlVec){
  // the index of the vector
  indexType uniqVecIndex = 0;
  // iterator over the list of already existing vectors inthe list
  vector<RLVecType>::const_iterator uniqVecListIter = uniqVecList.begin();
  // while not at the end of the list and while we don't recognize this
  // vector increment the index which uniquely defines the rl vector.
  while(uniqVecListIter != uniqVecList.end() && *uniqVecListIter != rlVec){
    ++uniqVecListIter;
    ++uniqVecIndex;
  }
  if(uniqVecIndex == static_cast<indexType>(uniqVecList.size())){
    // then we did not find the vector in the uniquified list and we
    // append it
    uniqVecList.push_back(rlVec);
  }else{
    // we have seen this vector before, so we want to save the results for
    // this vector when calculating the sines and cosines
    frequentIndexList.push_back(uniqVecIndex);
  }
  return uniqVecIndex;
}// end uniq_vec_index()

// gets a uniquified list of RLVecs and returns the unique index of a RLVec.
// if RLVec does not exist, append it to the list.
// This function does not memorize vectors which occur more than once.
inline indexType
uniq_vec_index(vector<RLVecType>& uniqVecList,
       const RLVecType& rlVec){
  // the index of the vector
  unsignedIndexType uniqVecIndex = 0;
  // iterator over the list of already existing vectors in the list
  vector<RLVecType>::const_iterator uniqVecListIter = uniqVecList.begin();
  // while not at the end of the list and while we don't recognize this
  // vector increment the index which uniquely defines the rl vector.
  while(uniqVecListIter != uniqVecList.end() && *uniqVecListIter != rlVec){
    ++uniqVecListIter;
    ++uniqVecIndex;
  }
  if(uniqVecIndex == static_cast<indexType>(uniqVecList.size())){
    // then we did not find the vector in the uniquified list and we
    // append it
    uniqVecList.push_back(rlVec);
  }
  return uniqVecIndex;
}// end uniq_vec_index()

// find overlaps between two uniquified vector lists and convert the unique
// vector indices
void
new_qk_index_conversion(const vector<RLVecType>& oldUVL,
const vector<RLVecType>& newUVL,
vector<indexType>& newOldIndexList){
  for(vector<RLVecType>::const_iterator iterNew = newUVL.begin();
      iterNew != newUVL.end(); ++iterNew){
    newOldIndexList.push_back(UVLindex(oldUVL, *iterNew));
  }
}// end new_qk_index_conversion()

// helper function which gets the number of successful index conversions
unsigned int
number_old_new_conversions(const vector<indexType>& newOldIndexList){
  unsigned int counter = 0;
  for(vector<indexType>::const_iterator iter = newOldIndexList.begin();
      iter != newOldIndexList.end(); ++iter)
    if(*iter != -INDEX_MAX)
      ++counter;
  return counter;
}// end number_old_new_conversions()

// gets a uniquified list of RLVecs and returns the unique index of a RLVec.
// -INDEX_MAX is returned if RLVec was not found
inline indexType
UVLindex(const vector<RLVecType>& uniqVecList, const RLVecType& rlVec){
  // the index of the vector
  indexType uniqVecIndex = 0;
  // iterator over the list of already existing vectors inthe list
  vector<RLVecType>::const_iterator uniqVecListIter = uniqVecList.begin();
  // while not at the end of the list and while we don't recognize this
  // vector increment the index which uniquely defines the rl vector.
  while(uniqVecListIter != uniqVecList.end() && *uniqVecListIter != rlVec){
    ++uniqVecListIter;
    ++uniqVecIndex;
  }
  if(uniqVecIndex == static_cast<indexType>(uniqVecList.size()))
    uniqVecIndex = -INDEX_MAX;
  return uniqVecIndex;
}// end UVLindex()

// writes out results
void
write_S3_results(char * parameterfile,
 char * filename,
 const unsigned int confLoop,
 const unsigned int simRunLoop,
 const unsigned int subSimRunLoop,
 const unsigned int startPointLoop,
 const unsigned int nr_configurations_used,
 const unsigned int nr_monomers,
 const unsigned int nrBins,
 const double binWidth,
 const unsigned int nrResultEntries,
 const vector<vector<vector<vector<resultType> > > >& results){
  int qBin, kBin, pBin, pBinMin, pBinMax;
  FILE *outfile_p, *parameterfile_p;
  char read_line[400], write_line[401];
  if ((outfile_p = fopen(filename,"w")) == NULL){
      sprintf(filename, "Couldn't open %s", filename);
      error(filename, HERE);
  }
  fprintf(outfile_p, "# %s \n", "Triple correlation static structure factors \
S_3(q, k, p) and S_3^p(q, k, p)");
  fprintf(outfile_p, "# %s \n", "Results at:");
  fprintf(outfile_p, "# %s%u \n", "simulation run: ", simRunLoop);
  fprintf(outfile_p, "# %s%u \n", "sub simulation run: ", subSimRunLoop);
  fprintf(outfile_p, "# %s%u \n", "start point: ", startPointLoop);
  fprintf(outfile_p, "# %s%u \n", "configuration in timeseries: ", confLoop);
  fprintf(outfile_p, "# %s%u \n", "number of configurations used: ",
  nr_configurations_used);
  fprintf(outfile_p, "# \n");
#ifdef CALC_COMPLEX_NO
  fprintf(outfile_p, "# %s \n", "q | k | p | #tupels | S_3^p(q, k, p) | \
S_3(q, k, p)");
#endif
#ifdef CALC_COMPLEX_YES
  fprintf(outfile_p, "# %s \n", "q | k | p | #tupels | Re(S_3^p(q, k, p)) | \
 Im(S_3^p(q, k, p)) | Re(S_3(q, k, p)) |  Im(S_3(q, k, p))");
#endif
  // write out results
  for(qBin = 0; qBin < static_cast<int>(nrBins); ++qBin){
    for(kBin = 0; kBin < static_cast<int>(nrBins); ++kBin){
#ifdef TAKE_OUTER_BINS_YES
      pBinMin = max(abs(qBin-kBin)-1, 0);
      pBinMax = min(qBin+kBin+1, static_cast<int>(nrBins-1));
#else
      // unless at 0 or nrBins-1, pBinMin and pBinMax are +1 resp. -1
      pBinMin = abs(qBin-kBin);
      pBinMax = qBin + kBin;
      if(pBinMax > static_cast<int>(nrBins) -1)
pBinMax = nrBins -1;
#endif
      for(pBin = 0; pBin < static_cast<int>(nrBins); ++pBin){
if(pBin >= pBinMin && pBin <= pBinMax){
#ifdef CALC_COMPLEX_NO
  fprintf(outfile_p, "%.1f %.1f %.1f %.2f %.6e %.6e\n", \
  binWidth / 2.0 + binWidth * static_cast<double>(qBin), \
  binWidth / 2.0 + binWidth * static_cast<double>(kBin), \
  binWidth / 2.0 + binWidth * static_cast<double>(pBin), \
  results[qBin][kBin][pBin][0] \
  /static_cast<resultType>(nr_configurations_used), \
  results[qBin][kBin][pBin][nrResultEntries-2] \
  / (max(static_cast<resultType>(1.0), \
 results[qBin][kBin][pBin][0]) \
     * static_cast<resultType>(nr_monomers)),
  results[qBin][kBin][pBin][nrResultEntries-1] \
  / (max(static_cast<resultType>(1.0), \
 results[qBin][kBin][pBin][0]) \
     * static_cast<resultType>(nr_monomers)));
#endif
#ifdef CALC_COMPLEX_YES
  fprintf(outfile_p, "%.1f %.1f %.1f %.2f %.6e %.6e %.6e %.6e\n", \
  binWidth / 2.0 + binWidth * static_cast<double>(qBin), \
  binWidth / 2.0 + binWidth * static_cast<double>(kBin), \
  binWidth / 2.0 + binWidth * static_cast<double>(pBin), \
  results[qBin][kBin][pBin][0] \
  / static_cast<resultType>(nr_samples_per_startpoint),
  results[qBin][kBin][pBin][nrResultEntries-4] \
  / (max(static_cast<resultType>(1.0), \
 results[qBin][kBin][pBin][0]) \
     * static_cast<resultType>(nr_monomers)),
  results[qBin][kBin][pBin][nrResultEntries-3] \
  / (max(static_cast<resultType>(1.0), \
 results[qBin][kBin][pBin][0]) \
     * static_cast<resultType>(nr_monomers)),
  results[qBin][kBin][pBin][nrResultEntries-2] \
  / (max(static_cast<resultType>(1.0), \
 results[qBin][kBin][pBin][0]) \
     * static_cast<resultType>(nr_monomers)),
  results[qBin][kBin][pBin][nrResultEntries-1] \
  / (max(static_cast<resultType>(1.0), \
 results[qBin][kBin][pBin][0]) \
     * static_cast<resultType>(nr_monomers)));
#endif
#ifdef WRITE_EMPTY_POINTS_YES
}else{
  // nothing calculated
  fprintf(outfile_p, "%.1f %.1f %.1f %s\n", \
  binWidth / 2.0 + binWidth * static_cast<double>(qBin), \
  binWidth / 2.0 + binWidth * static_cast<double>(kBin), \
```

```
  binWidth / 2.0 + binWidth * static_cast<double>(pBin), "X");
#endif
}// if in right p-bin range
      }
    }
  }

  /* append parameter file, so that we know how we got the data */
  fprintf(outfile_p, "#\n#\n#Parameters read from file '%s':\n#\n",
  parameterfile);
  if ((parameterfile_p = fopen(parameterfile,"r")) == NULL)
    fprintf(stdout,"WARNING: Couldn't open %s \n", parameterfile);
  else{
    /* Not very good: lines are truncated after 399 characters */
    while(fgets(read_line, 399, parameterfile_p) != NULL){
      /* the #'s signal gnuplot and xmgr that these lines are comments */
      sprintf(write_line,"#%s", read_line);
      fputs(write_line, outfile_p);
    }
    fclose(parameterfile_p);
  }
  /* close file */
  fclose(outfile_p);
  printf("MESSAGE: Wrote data to %s\n", filename);
}// end write_results()

// function which writes out a parameter file for continuing the calculation
void
write_new_params(const double box_size,
 const unsigned int nr_chains,
 const unsigned int mon_per_chain,
 const unsigned int nr_sim_runs,
 const unsigned int * const nr_sub_sim_runs,
 const char * const saving_scheme,
 const char * const file_sample_times,
 const char * const data_path,
 const unsigned int nrBins,
 const double binWidth,
 const unsigned int  wishedEntries,
 const double equalizationFactor,
 unsigned int rngSeed,
 const unsigned int nr_confs_to_calculate,
 const unsigned int confLoopStart,
 const unsigned int simRunLoopStart,
 const unsigned int subSimRunLoopStart,
 const unsigned int startPointLoopStart,
 char * const newParamsFileName){
 FILE *outfile_p;
 unsigned int iLoop;
 if ((outfile_p = fopen(newParamsFileName,"w")) == NULL){
     sprintf(newParamsFileName, "Couldn't open %s", newParamsFileName);
     error(newParamsFileName, HERE);
 }
 // cout << "DEBUG: startPointLoopStart=" << startPointLoopStart
//         << ", subSimRunLoopStart=" << subSimRunLoopStart
//         << "\n       simRunLoopStart=" << simRunLoopStart
//         << ", confLoopStart=" << confLoopStart << "\n";
  fprintf(outfile_p, "box_size=%.15lf\n", box_size);
  fprintf(outfile_p, "nr_chains=%u\n", nr_chains);
  fprintf(outfile_p, "mon_per_chain=%u\n", mon_per_chain);
  fprintf(outfile_p, "nr_sim_runs=%u\n", nr_sim_runs);
  for(iLoop=0; iLoop<nr_sim_runs; ++iLoop)
    fprintf(outfile_p, "%u\n", nr_sub_sim_runs[iLoop]);
  fprintf(outfile_p, "saving_scheme=%s\n", saving_scheme);
  fprintf(outfile_p, "file_sample_times=%s\n", file_sample_times);
  fprintf(outfile_p, "data_path=%s\n", data_path);
  fprintf(outfile_p, "nrBins=%u\n", nrBins);
  fprintf(outfile_p, "binWidth=%lf\n", binWidth);
  fprintf(outfile_p, "wishedEntries=%u\n", wishedEntries);
  fprintf(outfile_p, "equalizationFactor=%lf\n", equalizationFactor);
  fprintf(outfile_p, "rngSeed=%u\n", rngSeed);
  fprintf(outfile_p, "nr_confs_to_calculate=%u\n", nr_confs_to_calculate);
  fprintf(outfile_p, "confLoopStart=%u\n", confLoopStart);
  fprintf(outfile_p, "simRunLoopStart=%u\n", simRunLoopStart);
  fprintf(outfile_p, "subSimRunLoopStart=%u\n", subSimRunLoopStart);
  fprintf(outfile_p, "startPointLoopStart=%u\n", startPointLoopStart);
  fclose(outfile_p);
  printf("MESSAGE: Wrote new parameters to %s\n", newParamsFileName);
}// end write_new_params()
```

Bibliography

[1] G. B. McKenna, in *Comprehensive Polymer Science*, edited by C. Booth and C. Price (Pergamon, New York, 1986), vol. 2, pp. 311–362.

[2] K. Binder, ed., *Monte Carlo and Molecular Dynamics Simulations in Polymer Science* (Oxford University Press, New York, 1995).

[3] E. Rössler and H. Sillescu, in *Material Science and Technology*, edited by J. Zarzycki (VCH, Weinheim, 1991), vol. IX, pp. 573–618.

[4] J. Jäckle, Rep. Prog. Phys. **49**, 171 (1986).

[5] P. Lunkenheimer, U. Schneider, R. Brand, and A. Loidl, Contempory Phys. **41**, 15 (2000).

[6] W. Götze, Condens. Matter Phys. **1**, 873 (1998).

[7] W. Götze, J. Phys.: Condens. Matter **11**, A1 (1999).

[8] W. Götze, in *Proceedings of the Les Houches Summer School of Theoretical Physics, Les Houches 1989, Session LI*, edited by J. P. Hansen, D. Levesque, and J. Zinn-Justin (North-Holland, Amsterdam, 1991), pp. 287–503.

[9] W. Paul and J. Baschnagel, in *Monte Carlo and Molecular Dynamics Simulations in Polymer Science*, edited by K. Binder (Oxford University Press, New York, 1995), chap. 6.

[10] W. Kob, J. Phys.: Condens. Matter **11**, R85 (1999).

[11] W. Kob, in *Les Houches Summer School for Theoretical Physics, LXXVII: Slow Relaxations and Nonequilibrium Dynamics in Condensed Matter* (2002), cond-mat/0212344.

[12] C. Bennemann, W. Paul, K. Binder, and B. Dünweg, Phys. Rev. E **57**, 843 (1998).

[13] C. Bennemann, Ph.D. thesis, Johannes Gutenberg-Universität Mainz (1998), unpublished.

[14] C. Bennemann, J. Baschnagel, and W. Paul, Eur. Phys. J. B **10**, 323 (1999).

[15] M. Aichele and J. Baschnagel, Eur. Phys. J. E **5**, 229 (2001).

[16] M. Aichele and J. Baschnagel, Eur. Phys. J. E **5**, 245 (2001).

[17] K. Binder, J. Baschnagel, and W. Paul, Progr. Poly. Sci. **28**, 115 (2003).

[18] S.-H. Chong and W. Götze, Phys. Rev. E **65**, 041503 (2002).

[19] S.-H. Chong and M. Fuchs, Phys. Rev. Lett. **88**, 185702 (2002).

[20] M. Doi and S. F. Edwards, *The Theory of Polymer Dynamics* (Oxford University Press, New York, 1986).

[21] H. Sillescu, J. Non-Cryst. Solids **243**, 81 (1999).

[22] M. D. Ediger, Annu. Rev. Phys. Chem. **51**, 99 (2000).

[23] S. C. Glotzer, J. Non-Cryst. Solids **274**, 342 (2000).

[24] R. Richert, J. Phys.: Condens. Matter **14**, R703 (2002).

[25] C. Bennemann, C. Donati, J. Baschnagel, and S. C. Glotzer, Nature **399**, 246 (1999).

[26] Y. Gebremichael, T. B. Schrøder, F. W. Starr, and S. C. Glotzer, Phys. Rev. E **64**, 051503 (2001).

[27] B. N. J. Persson, *Sliding Friction: Physical Principles and Applications* (Springer, Berlin, 2000).

[28] B. Bhushan, *Principles and Applications of Tribology* (Wiley, 1999).

[29] F. P. Bowden and D. Tabor, *The Friction and Lubrication of Solids*, Oxford Classic Texts in the Physical Sciences (Oxford University Press, New York, 2001), paperback reprint.

[30] G. He, M. H. Müser, and M. O. Robbins, Science **284**, 1650 (1999).

[31] G. He and M. O. Robbins, Phys. Rev. B **64**, 035413 (2001).

[32] M. H. Müser, L. Wenning, and M. O. Robbins, Phys. Rev. Lett. **86**, 1295 (2001).

[33] E. Kumacheva, Prog. Surf. Sci. **58**, 75 (1998).

[34] S. Consta, N. B. Wilding, D. Frenkel, and Z. Alexandrowicz, J. Chem. Phys. **110**, 3220 (1999).

[35] S. Consta, T. J. H. Vlugt, J. W. Hoeth, B. Smit, and D. Frenkel, Mol. Phys. **97**, 1243 (1999).

[36] D. Frenkel and B. Smit, *Understanding Molecular Simulation* (Academic Press, London, 2002), 2nd ed.

[37] J. M. Haile, *Molecular Dynamics Simulation: Elementary Methods* (John Wiley and Sons, Inc., New York, 1997).

[38] K. Kremer and G. S. Grest, J. Chem. Phys. **92**, 5057 (1990).

[39] M. H. Müser and M. O. Robbins, Phys. Rev. B **61**, 2335 (2000).

[40] G. He and M. O. Robbins, Trib. Lett. **10**, 7 (2001).

[41] M. H. Müser, M. Urbakh, and M. O. Robbins, Adv. Chem. Phys. **126**, 187 (2003).

[42] K. Shinjo and M. Hirano, Surf. Sci. **283**, 473 (1993).

[43] B. N. J. Persson, J. Chem. Phys. **113**, 5477 (2002).

[44] B. N. J. Persson and P. Ballone, J. Chem. Phys. **112**, 9524 (2000).

[45] C. Bennemann, W. Paul, K. Binder, and B. Dünweg, Phys. Rev. E **57**, 843 (1998).

[46] M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids* (Oxford Science Publications, Oxford, 1987).

[47] M. Lüscher, Comput. Phys. Commun. **79**, 100 (1994).

[48] F. James, Comput. Phys. Commun. **79**, 111 (1994).

[49] L. N. Shchur and P. Butera, Int. J. Mod. Phys **C9**, 607 (1998).

[50] I. Vattulainen, Phys. Rev. E **59**, 7200 (1999).

[51] C. W. Gear, *Numerical Initial Value Problems in ordinary Differential Equations* (Prentice Hall, Englewood Cliffs, NJ, 1971), chap. 9.

[52] B. D. Todd, D. Evans, and P. J. Daivis, Phys. Rev. E **52**, 1627 (1995).

[53] J. P. Hansen and I. R. McDonald, *Theory of Simple Liquids* (Academic Press, London, 1986).

[54] F. Varnik, J. Baschnagel, and K. Binder, J., Chem. Phys. **113**, 4444 (2000).

[55] M. Mondello and G. S. Grest, J. Chem. Phys. **106**, 9327 (1997).

[56] A. D. Sokal, in *Monte Carlo and Molecular Dynamics Simulations in Polymer Science*, edited by K. Binder (Oxford University Press, New York, 1995), chap. 2.

[57] M. N. Rosenbluth and A. W. Rosenbluth, J. Chem. Phys. **23**, 356 (1955).

[58] T. Stühn, private communication (2003).

[59] M. Fuchs, W. Götze, S. Hildebrand, and A. Latz, J. Phys.: Condens. Matter **4**, 7709 (1992).

[60] H. Z. Cummins, W. M. Du, M. Fuchs, W. Götze, S. Hildebrand, A. Latz, G. Li, and N. J. Tao, Phys. Rev. E **47**, 4223 (1993), with complement in Phys. Rev. E **59**, 5625 (1999).

[61] H. C. Barshilia, G. Li, G. Q. Shen, and H. Z. Cummins, Phys. Rev. E **59**, 5625 (1999), complement to Phys. Rev. E **47**, 4223 (1993).

[62] R. Schilling and T. Scheidsteger, Phys. Rev. E **56**, 2932 (1997).

[63] L. Fabbian, A. Latz, R. Schilling, F. Sciortino, P. Tartaglia, and C. Theis, Phys. Rev. E **62**, 2388 (2000).

[64] T. Franosch, M. Fuchs, W. Götze, M. R. Mayr, and A. P. Singh, Phys. Rev. E **55**, 7153 (1997).

[65] F. Sciortino and W. Kob, Phys. Rev. Lett. **86**, 648 (2001).

[66] A. Rinaldi, F. Sciortino, and P. Tartaglia, Phys. Rev. E **63**, 061210 (2001).

[67] J. Buchholz, W. Paul, F. Varnik, and K. Binder, J. Chem. Phys. **117**, 7364 (2002).

[68] H. Meyer and F. Müller-Plathe, J. Chem. Phys. **115**, 7807 (2001).

[69] H. Meyer and F. Müller-Plathe, Macromolecules **35**, 1241 (2002).

[70] C. Bennemann, J. Baschnagel, and W. Paul, Eur. Phys. J. B **10**, 323 (1999).

[71] D. Chandler and H. C. Andersen, J. Chem. Phys. **57**, 1930 (1972).

[72] *CLAPACK (C-translation of LAPACK)*, www.netlib.org/clapack.

[73] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, et al., *LAPACK Users' Guide* (Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999), 3rd ed., ISBN 0-89871-447-8 (paperback).

[74] H. Meyer, private communication (2001).

[75] J.-L. Barrat, J.-P. Hansen, and G. Pastore, Phys. Rev. Lett. **58**, 2075 (1987).

[76] H. W. Jackson and E. Feenberg, Rev. Mod. Phys. **34**, 686 (1962).

[77] M. Aichele, Diploma thesis, Johannes Gutenberg-Universität Mainz (2000), unpublished.

[78] V. Krakoviack, J. P. Hansen, and A. A. Louis, Europhys. Lett. **58**, 53 (2002).

[79] D. Chandler, J. Chem. Phys. **59**, 2742 (1973).

[80] K. S. Schweizer and J. G. Curro, Adv. Polymer Sci. **116**, 319 (1994).

[81] K. S. Schweizer and J. G. Curro, Adv. Chem. Phys. **98**, 1 (1997).

[82] W. Kob, in *Supercooled Liquids: Advances and Novel Applications*, edited by J. T. Fourkas, D. Kivelson, U. Mohanty, and K. A. Nelson (American Chemical Society, Washington DC, 1997), ACS Symposium Series 676, pp. 28–44.

[83] J.-L. Barrat, W. Götze, and A. Latz, J. Phys.: Condens. Matter **1**, 7163 (1989).

[84] S.-H. Chong (2002), private communication.

[85] M. Fuchs, J. Non-Cryst. Solids **172-174**, 241 (1994).

[86] S.-H. Chong and M. Aichele, in preparation.

[87] K. Schmidt-Rohr and H. W. Spiess, Phys. Rev. Lett. **66**, 1991 (1991).

[88] U. Tracht, M. Wilhelm, A. Heuer, H. Feng, K. Schmidt-Rohr, and H. W. Spiess, Phys. Rev. Lett. **81**, 2727 (1998).

[89] M. T. Cicerone and M. Ediger, J. Chem. Phys. **103**, 5684 (1995).

[90] B. Schiener, R. Böhmer, A. Loidl, and R. V. Chamberlin, Science **274**, 752 (1996).

[91] M. Yang and R. Richert, J. Chem. Phys. **115**, 2676 (2001).

[92] E. Weeks, J. C. Crocker, A. C. Levitt, A. Schofield, and D. A. Weitz, Science **287**, 627 (2000).

[93] W. K. Kegel and A. van Blaaderen, Science **287**, 290 (2000).

[94] E. Bartsch, Current Opinions in Coll. Int. Sci. **3**, 577 (1998).

[95] W. van Megen, Transport Theory Stat. Phys. **24**, 1017 (1995).

[96] W. van Megen and S. M. Underwood, Phys. Rev. E **49**, 4206 (1994).

[97] W. Götze and L. Sjögren, Phys. Rev. A **43**, 5442 (1991).

[98] W. van Megen, T. C. Mortensen, J. Müller, and S. R. Williams, Phys. Rev. E **58**, 6073 (1998).

[99] R. Yamamoto and A. Onuki, Phys. Rev. Lett. **81**, 4915 (1998).

[100] W. Kob, C. Donati, S. J. Plimpton, P. H. Poole, and S. C. Glotzer, Phys. Rev. Lett. **79**, 2827 (1997).

[101] C. Donati, J. F. Douglas, W. Kob, S. J. Plimpton, P. H. Poole, and S. C. Glotzer, Phys. Rev. Lett. **80**, 2338 (1998).

[102] C. Donati, S. C. Glotzer, P. H. Poole, W. Kob, and S. J. Plimpton, Phys. Rev. E **60**, 3107 (1999).

[103] N. Giovambattista, S. V. Buldyrev, F. W. Starr, and H. E. Stanley, Phys. Rev. Lett. **90**, 085506 (2003).

[104] B. Doliwa and A. Heuer, Phys. Rev. Lett. **80**, 4915 (1998).

[105] B. Doliwa and A. Heuer, Phys. Rev. E **61**, 6898 (2000).

[106] B. Doliwa and A. Heuer, J. Non-Cryst. Solids **307–310**, 32 (2002).

[107] K. Vollmayr-Lee, W. Kob, K. Binder, and A. Zippelius, J. Chem. Phys. **116**, 5158 (2002).

[108] C. Bennemann, W. Paul, J. Baschnagel, and K. Binder, J. Phys.: Condens. Matter **11**, 2179 (1999).

[109] C. Bennemann, J. Baschnagel, W. Paul, and K. Binder, Comp. Theo. Poly. Sci. **9**, 217 (1999).

[110] S. Kämmerer, W. Kob, and R. Schilling, Phys. Rev. E **58**, 2131 (1998).

[111] W. Kob and H. C. Andersen, Phys. Rev. E **51**, 4626 (1995).

[112] F. Sciortino, P. Gallo, P. Tartaglia, and S.-H. Chen, Phys. Rev. E **54**, 6331 (1996).

[113] S. Mossa, R. Di Leonardo, G. Ruocco, and M. Sampoli, Phys. Rev. E **62**, 612 (2000).

[114] J. Horbach, W. Kob, and K. Binder, Phil. Mag. B **77**, 297 (1998).

[115] J. Colmenero, F. Alvarez, and A. Arbe, Phys. Rev. E. **65**, 041804 (2002).

[116] R. Zorn, Phys. Rev. B **55**, 6249 (1997).

[117] M. Guenza, Phys. Rev. Lett. **88**, 025901 (2002).

[118] M. Fuchs, W. Götze, and M. R. Mayr, Phys. Rev. E **58**, 3384 (1998).

[119] K. Kremer and G. S. Grest, in *Monte Carlo and Molecular Dynamics Simulations in Polymer Science*, edited by K. Binder (Oxford University Press, New York, 1995), chap. 4.

[120] W. Kob and H. C. Andersen, Phys. Rev. E **52**, 4134 (1995).

[121] W. Götze and L. Sjögren, Z. Phys. B **65**, 415 (1987).

[122] J. P. Wittmer, A. Milchev, and M. E. Cates, J. Chem. Phys. **109**, 834 (1998).

[123] D. Dowson, *History of Tribology* (Longman, New York, 1979).

[124] M. Brillouin, *Notices sur les Travaux Scientifiques* (Gauthiers-Villars, Paris, 1909).

[125] C. Caroli and P. Nozières, Eur. Phys. J. B **4** (1998).

[126] H. Hertz, Journal für die reine und angewandte Mathematik **92**, 156 (1881).

[127] K. L. Johnson, K. Kendall, and A. D. Roberts, Proc. R. Soc. London A **324**, 301 (1971).

[128] A. Fodgen and L. R. White, J. Colloid Interface Sci. **138**, 414 (1990).

[129] R. S. Bradley, Philos. Mag. Suppl. **13**, 853 (1932).

[130] B. V. Derjaguin, V. M. Muller, and Y. P. Toporov, J. Colloid Interface Sci. **53**, 314 (1975).

[131] V. M. Muller, V. S. Yuschenko, and B. V. Derjaguin, J. Colloid Interface Sci. **92**, 92 (1983).

[132] D. Tabor, J. Colloid Interface Sci. **58**, 2 (1977).

[133] E. Gnecco, R. Bennewitz, and E. Meyer, Phys. Rev. Lett. (2002).

[134] M. G. Rozman, M. Urbakh, and J. Klafter, Phys. Rev. Lett. **77**, 683 (1996).

[135] M. G. Rozman, M. Urbakh, and J. Klafter, Phys. Rev. E **54**, 6485 (1996).

[136] P. A. Thompson and M. O. Robbins, Science **250**, 792 (1990).

[137] P. A. Thompson and M. O. Robbins, Phys. Rev. A **41**, 6830 (1990).

[138] L. Wenning and M. H. Müser, Europhys. Lett **54**, 693 (2001).

[139] B. N. J. Persson, V. N. Samoilov, S. Zilberman, and A. Nitzan, J. Chem. Phys. **117**, 3897 (2002).

[140] L. Prandtl, ZS. angew. Math. Mech. **8**, 85 (1928).

[141] G. A. Tomlinson, Phil. Mag. Series **7**, 905 (1929).

[142] Y. I. Frenkel and T. Kontorova, Zh. Eksp. Teor. Fiz. **8**, 1340 (1938).

[143] O. M. Braun and Y. S. Kivshar, Phys. Rep. **306**, 1 (1998).

[144] T. Strunz and F.-J. Elmer, Phys. Rev. E **58**, 1601 (1998).

[145] O. M. Braun and M. Peyrard, Phys. Rev. E **63**, 046110 (2001).

[146] P. Español and P. Warren, Europhys. Lett. **30**, 191 (1995).

[147] W. K. den Otter and J. H. R. Clarke, Europhys. Lett. **53**, 426 (2001).

[148] H. Mori, Prog. Theor. Phys. **33**, 7526 (1965).

[149] R. Zwanzig, Annu. Rev. Phys. Chem. **16**, 67 (1965).

[150] H. Risken, *The Fokker Planck equation* (Springer, 1984).

[151] C. Daly, J. Zhang, and J. B. Sokoloff, Phys. Rev. Lett. **90**, 24101 (2003).

[152] E. Jaynes, *Papers on Probability, Statistics and Statistical Physics* (Reidel, Dortrecht, 1982).

[153] J. Honerkamp, *Statistical Physics* (Springer, Berlin, 1998).

[154] M. H. Müser, Phys. Rev. Lett. **89**, 224301 (2002).

[155] W. Steele, Surf. Sci. **36**, 317 (1973).

[156] A. Patrykiejew, S. Sokołowski, and K. Binder, Surface Science Reports **37**, 207 (2000).

[157] D. S. Fisher, Phys. Rev. B **31**, 1396 (1985).

[158] E. D. Smith, M. O. Robbins, and M. Cieplak, Phys. Rev. B **54**, 8252 (1996).

[159] M. J. Stevens and M. O. Robbins, Phys. Rev. E **48**, 3778 (1993).

[160] P. Hänggi, P. Talkner, and M. Borkovec, Rev. Mod. Phys. **62**, 251 (1990).

[161] M. Ruths and S. Granick, Langmuir **16**, 8368 (2000).

[162] J. Krim and R. Chiarello, J. Vac. Sci. Technol. B **9**, 1343 (1991).

[163] J. Krim, D. H. Solina, and R. Chiarello, Phys. Rev. Lett. **66**, 181 (1991).

[164] A. Mukhopadhyay, Z. Jiang, C. B. Sung, and S. Granick, Phys. Rev. Lett. **89**, 136103 (2002).

[165] S. Caracciolo, A. Pelissetto, and A. D. Sokal, J. Phys. A: Math. Gen. **23**, L969 (1990).

[166] R. Everaers, I. S. Graham, and M. J. Zuckermann, J. Phys. A: Math. Gen. **28**, 1271 (1995), and references therein.

[167] C. Mischler, J. Baschnagel, and K. Binder, Adv. Coll. Interface Sci. **94**, 197 (2001).

[168] I. Carmesin and K. Kremer, J. Phys. France **51**, 915 (1990).

[169] F. Mugele, T. Becker, A. Klingner, and M. Salmeron, Coll. Surf. A **206**, 105 (2002), and references therein.