Oberflächengebundene flüssigkristalline Polymere in nematischer Lösung - eine Monte Carlo-Untersuchung

Dissertation

zur Erlangung des Grades
Doktor der Naturwissenschaften
am Fachbereich Physik
der Johannes Gutenberg-Universität
in Mainz

vorgelegt von

Harald Oliver Lange
geboren in Wiesbaden

Mainz, Januar 2001

Datum der mündlichen Prüfung : 8. Februar 2001

Oberflächengebundene flüssigkristalline Polymere in nematischer Lösung - eine Monte Carlo-Untersuchung

In der vorliegenden Arbeit wurde ein System von oberflächengebundenen flüssigkristallinen Polymeren in nematischer Lösung mittels Monte Carlo-Simulationen im NpT-Ensemble untersucht. Dabei wurden die Lösungsmittelteilchen und Polymerteilchen durch weiche abstoßende Ellipsoide des Durchmessers σ_s und der Länge σ_l modelliert. Darüber hinaus wirkten zwischen Teilchen innerhalb einer Kette zusätzliche Bondlängen- und Bondwinkelpotentiale. Die Kopfenden der Ketten waren mit ihren Mittelpunkten auf einen kubischen Gitter in der Oberfläche fixiert und wechselwirkten nicht mit den übrigen Teilchen.

Für die Simulation kam ein neuartiger „Configurational biased Monte Carlo"-Algorithmus zum Einsatz, mit dem es möglich war, die Simulation stark zu beschleunigen. Dies wurde erreicht, indem die Bindungen einer Polymerkette entfernt und die ehemaligen Kettenteilchen somit zu Lösungsmittelteilchen werden. Aus den Lösungsmittelteilchen wurde dann durch das Ziehen neuer Bindungen eine neue Kette generiert, über deren Annahme dann eine Metropolisabfrage entschied.

In den Simulationen sollte untersucht werden, inwieweit es möglich ist durch Variation der Pfropfdichte einer auf eine glatte Wand aufgebrachten Polymerbürste die Richtung der Orientierung des sie quellenden Lösungsmittels zu beeinflussen.

Neben Systemen ohne Ketten wurden Systeme mit Pfropfdichten bis zu $\sigma = 0.84/\sigma_s^2$ über mehrere Millionen Monte Carlo-Schritte simuliert.

Mit den durchgeführten Simulationen konnte gezeigt werden, daß ein Phasenübergang zwischen einer Phase mit flach entlang den Wänden ausgerichteten Teilchen und einer Phase mit Teilchen, deren Orientierung nicht parallel zur Wand ist, existiert. Dieser Phasenübergang tritt im hier betrachteten System bei einer Pfropfdichte von $\sigma = 0.13/\sigma_s^2$ auf.

Weiterhin konnte bei hohen Pfropfdichten ($\sigma \geq 0.8/\sigma_s^2$) eine Phase mit einer Ausrichtung der Lösungsmittelteilchen senkrecht zur Grundfläche gefunden werden, obwohl die Ketten auch bei diesen Pfropfdichten nicht senkrecht stehen und die Bürste und das Lösungsmittel entkoppeln.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Während unserer Schulzeit hat man uns gelehrt, daß Materie in drei Zuständen vorkommt: fest, flüssig und gasförmig. Dies ist jedoch nicht ganz richtig. [1]

Neben diesen drei Aggregatzuständen, die „gewöhnliche“ Stoffe annehmen können, existiert eine Klasse von Stoffen die sich in ihrem Verhalten davon unterscheiden: die Flüssigkristalle.

Als Flüssigkristalle bezeichnet man Materialien, die einen Mischzustand zwischen der festen und der flüssigen Phase annehmen können, einen flüssigkristallinen oder mesogenen Zustand. Dieser Zustand, den etwa fünf Prozent aller organischen Verbindungen aufweisen können, zeichnet sich dadurch aus, daß er sowohl Charakteristika einer Flüssigkeit als auch eines Festkörpers besitzt. Dabei ist es jedoch nicht so, daß es nur eine flüssigkristalline Phase gibt, sondern es können je nach betrachtetem Material mehrere, zum Teil sehr unterschiedliche flüssigkristalline Phasen auftreten.

Allen Stoffen, die eine oder mehrere flüssigkristalline Phase besitzen, ist jedoch gemein, daß sie aus entweder stäbchenförmigen oder scheibenförmigen Konstituenten bestehen, d.h. es existiert eine Achse, die gegenüber den anderen ausgezeichnet ist. Außer einer isotropen Phase (Abb. 1.1 links), in der es weder Positions- noch Orientierungsordnung gibt, und die somit einer Flüssigkeit entspricht, existieren eine ganze Reihe teilgeordneter Phasen, die sich durch ihre Ordnung und somit auch durch ihre Eigenschaften unterscheiden.

Die einfachste flüssigkristalline Phase für stäbchenförmige Teilchen ist die nematische Phase (Abb. 1.1 rechts). In ihr sind die Teilchen wie in einer Flüssigkeit positionsungeordnet, weisen jedoch eine Orientierungsordnung auf, die die elongierten Teilchen im Mittel mit ihrer ausgezeichneten Achse entlang eines Raumvektors $\hat{n}$ (dem Direktor) ausrichtet.

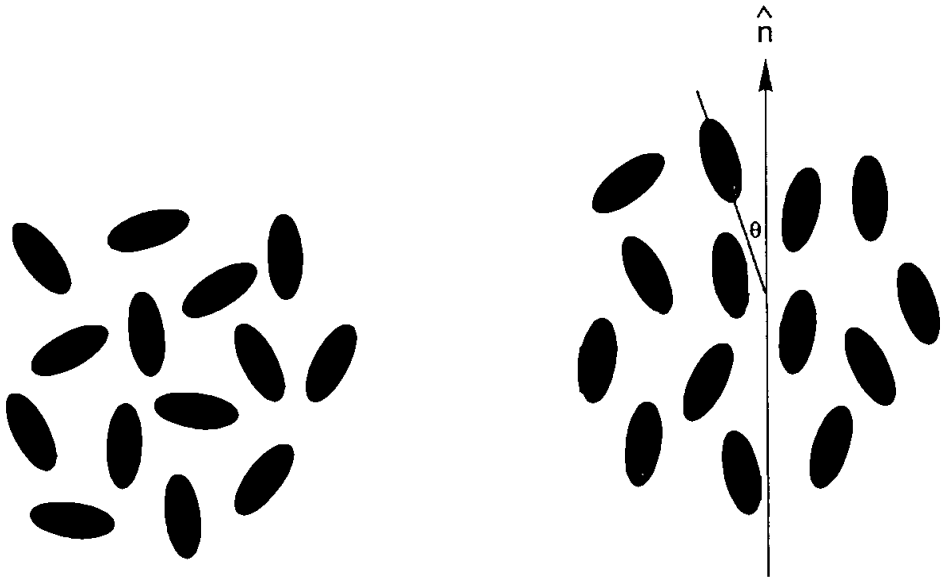

Abbildung 1.1:

Schematische Darstellung von isotroper Phase (links) und nematischer Phase (rechts)

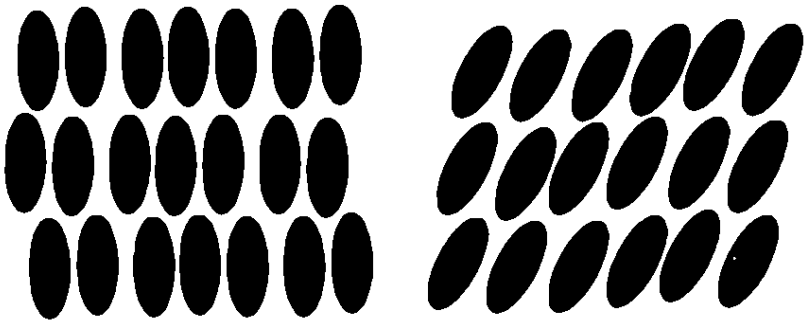

Abbildung 1.2:

Schematische Darstellung der Smektisch A-Phase (links) und der Smektisch C-Phase (rechts)

Den smektischen Phasen, die im Vergleich zur nematischen Phase bei niedrigeren Temperaturen auftreten, ist gemein, daß sie außer einer Orientierungsordnung auch eine beschränkte Positionsordnung besitzen.

Diese Positionsordnung sorgt dafür, daß sich im System in einer Raumrichtung geordnet einlagige Schichten ausbilden. Die Schichten haben einen festen Abstand zueinander, können jedoch einfach entlang ihrer Grenze gegeneinander verschoben werden. Innerhalb dieser Schichten sind die Teilchen positionsungeordnet, weisen jedoch eine Orientierungsordnung auf. So ist in der Smektisch A-Phase, welche die „Hochtemperaturphase“ der smektischen Phasen ist, der Direktor parallel zur Flächennormalen der Schicht und besitzt bei der Smektisch C-Phase einen Winkel zu ihr, der teilweise auch temperaturabhängig ist (Abb. 1.2).

Es existieren eine ganze Reihe weiterer smektischer Phasen, bei denen eine begrenzte Positionsordnung innerhalb der Schichten auftritt. Eine gute Übersicht liefert [2].

Weiterhin existieren cholesterische Phasen (der Name wurde abgeleitet vom englischen Wort für Cholesterin, cholesterol, bei dem zuerst eine solche Phase gefunden wurde), die sowohl nematischen Ursprungs (twisted nematics (Abb. 1.3)) als auch smektischen Ursprungs (twisted smectics (Abb. 1.4)) sein können, bei denen die Richtung des Direktors sich von Schicht zu Schicht ändert und somit eine Spiralstruktur des Direktors entsteht.

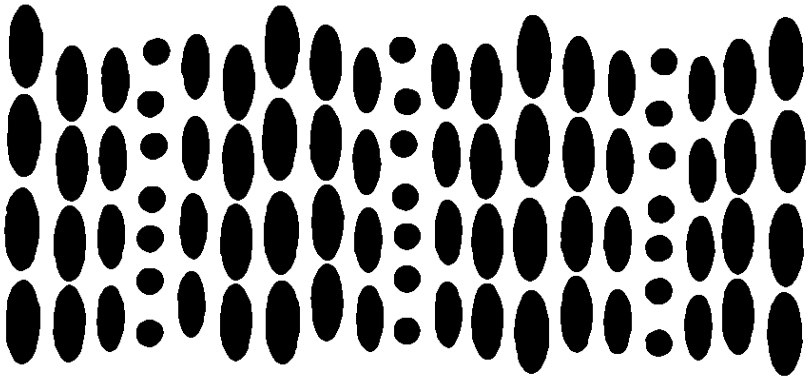

Abbildung 1.3: Schematische Darstellung der Twisted Nematic-Phase

Die Gründe für das Vorliegen flüssigkristalliner Phasen sind zum einen sterischer Art, d. h. das Verhalten wird durch die Geometrie der Teilchen und Packungseffekte verursacht, und zum anderen elektrostatischer Art.

Ein typisches Beispiel für ein flüssigkristallines Molekül ist Alkylcyanobiphenyl (n-CB) mit der Summenformel $NC-(C_6H_4)_2-(CH_2)_{n-1}-CH_3$.

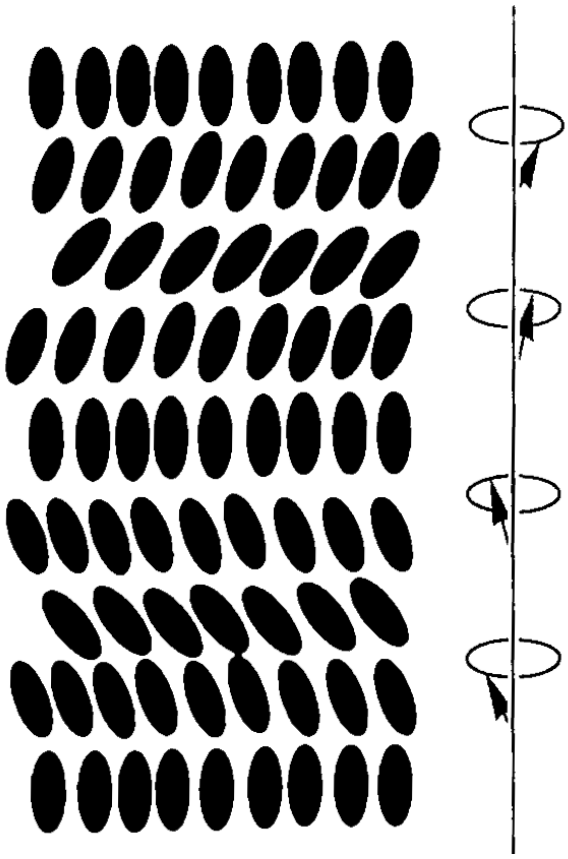

Abbildung 1.4: Schematische Darstellung der Twisted Smectic-Phase

Die Richtung des Direktors ist im Volumen nicht vorgegegeben. Oberflächen und Grenzflächen hingegen führen eine ausgezeichnete Richtung ein. Diesen Effekt, daß eine Oberfläche den mit ihr in Kontakt stehenden flüssigkristallinene Teilchen eine Orientierung vorgibt, nennt man Orientierungsverankerung [1, 3]. Wenn die orientierende Wirkung der Oberfläche in Konkurenz mit externen elektrischen oder magnetischen Felder tritt, so wird eine Verankerungsenergie bestimmbar [1, 4, 5].

Die Verankerungsenergie ist sinngemäß eine inverse Suszeptibilität und beschreibt, welchen Widerstand eine Orientierung an der Oberfläche einer Veränderung durch externe Drehmomente entgegensetzt. Makroskopisch kann man die Orientierungsverankerung als anisotrope Grenzflächenenergie auffassen, wobei die Grenzfläche dann die Oberfläche sowie eine Flüssigkristall-Grenzschicht mit einer Dicke von wenigen Moleküllagen umfaßt, innerhalb derer sich der Ordnungszustand vom Ordnungszustand im Inneren unterscheidet [6, 7, 8].

Zu Computersimulationen von Verankerung auf glatten und gewellten Oberflächen gibt es bereits einige Arbeiten [9, 10, 11, 12].

Es existieren auch Computersimulationen von Flüssigkristallen in dünnen Schichten, die sowohl auf Monte Carlo- [13] als auch auf Molekulardynamik-Methoden [14, 15] basieren. Meist werden dort Situationen untersucht, in denen die Oberflächen die Flüssigkristalle senkrecht verankern.

Das Ziel dieser Arbeit ist es, mittels Computersimulationen zu untersuchen, inwieweit sich die Neigung des Direktors in einem nematischen Flüssigkristall durch die Bindung von ebenfalls flüssigkristallinen Polymerbürsten auf glatten, den Nematen begrenzenden Wänden beeinflussen läßt.

Die zugrunde liegende Idee hierzu ist, die Streckung der Polymerbürste als orientierenden Faktor einzuführen. Da die Quellung von kovalent auf eine Oberfläche fixierten Polymeren mit einer Streckung des Polymerknäuels einhergeht [16, 17, 18, 19], wird eine vertikale Vorzugsrichtung der Polymerhauptketten bzw. der mesogenen Bauelementen, aus der sie besteht, induziert. Diese Streckung stellt einen Zustand minimaler freier Energie dar, der sowohl dem Bestreben des Polymers sich zu verdünnen als auch den elastischen Rückstellkräften Rechnung trägt.

Somit ist zur Erzeugung und Aufrechterhaltung dieses gestreckten Zustandes keinerlei weitere Beeinflussung durch äußere Parameter nötig. Solche oberflächengebundenen und gestreckten Polymermoleküle werden in der Literatur mit dem Begriff „Polymerbürsten“ (polymer brushes) beschrieben [18, 20, 21, 22, 23, 24].

Man erwartet in einem solchen Szenario, daß in einem System ohne Polymerbürsten, d.h. in einem System flüssigkristalliner Teilchen an einer glatten Wand, die Teilchen eine Ausrichtung parallel zur Wand einnehmen (planare Verankerung). Im Falle einer dichten Polymerbürste erwartet man eine homeotrope, d.h. senkrechte, Verankerung des Nematen auf der neuen, von der Bürste gebildeten Oberfläche (Abb. 1.5).

Unklar und somit untersuchenswert ist jedoch das Verhalten des Nematen bei einem System mit geringerer Pfropfdichte, also einer weniger dichten Polymerbürste. In diesem Fall könnte man sich zum einen vorstellen, daß es einen sprunghaften Übergang von einem Regime zum anderen gibt, oder daß das System zum anderen eine Neigung in eine Zwischenrichtung realisiert, die sich eventuell durch die Pfropfdichte steuern lassen könnte.

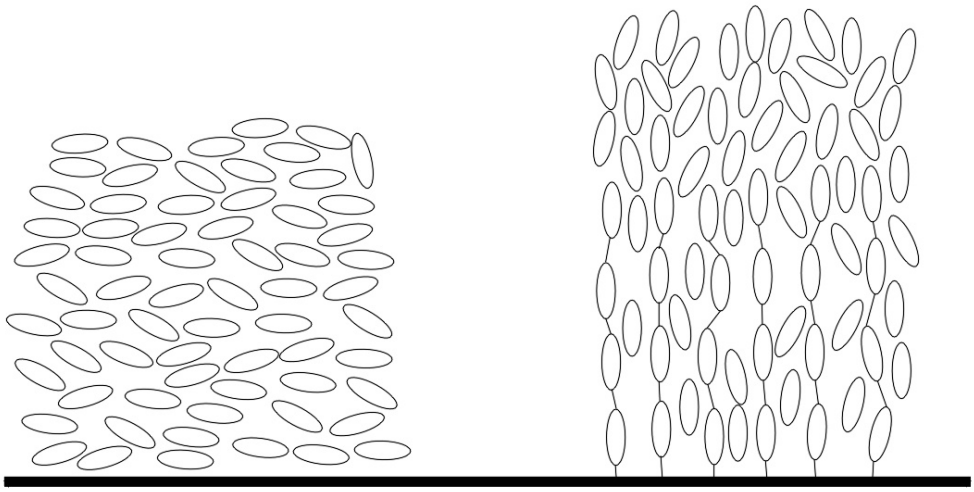

Abbildung 1.5:
Erwartetes Verankerungsverhalten eines nematischen Füssigkristalls bei Verankerung an einer glatten Wand (linke Seite) und Verankerung auf einer dichten Polymerbürste (rechte Seite).

Die zweite Variante wäre zum Beispiel für eine Twisted-Nematic (TN) Flüssigkristallzelle von Vorteil. Bei einem „twisted nematic liquid crystal display“ (TN-LCD) (Abbildung 1.6 zeigt eine schematische Darstellung) handelt es sich um eine Flüssigkristallzelle, bei der die Orientierung des Direktors an der oberen und der unteren Wand parallel zur Oberfläche ist, und die dabei einen Winkel von 90 Grad zueinander haben. Wegen der auf die Glasflächen aufgebrachten, zueinander senkrecht stehenden Polarisationsfilter und der Eigenschaft der Zelle, die Polarisationsrichtung um eben diese 90 Grad zu drehen, ist die Zelle im ausgeschalteten Zustand durchsichtig. Legt man jetzt eine Spannung an der Zelle an, so orientieren sich die Teilchen entlang der Stromrichtung und die Zelle verliert ihre Fähigkeit, die Polarisationsrichtung zu drehen. Somit wird die Zelle undurchsichtig (opaque).

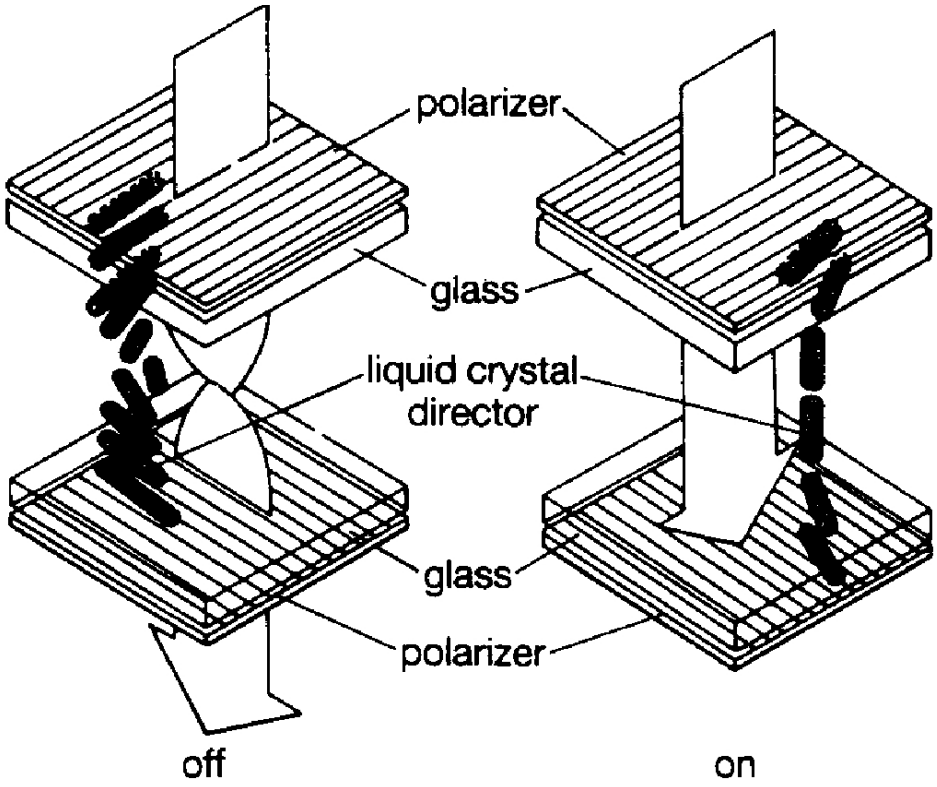

Abbildung 1.6: Darstellung einer Twisted-Nematic Flüssigkristallzelle

Dabei hängt die Schaltzeit der Zelle davon ab, wie die Teilchen an der Wand liegen, d.h., ob die Teilchen wirklich flach an der Wand liegen oder ob sie einen (kleinen) Winkel gegen die Wand aufweisen. Ein solcher Anstellwinkel (pre-tilt-angle), wie er experimentell durch mechanisches Reiben von Oberflächen realisiert wird [35, 36, 37], verringert die Schaltzeit einer solchen Zelle signifikant. Leider können bei diesem Verfahren nur Anstellwinkel von wenigen Grad realisiert werden. Weiterhin können wegen der immer bestehenden Gefahr des Abriebs keine großen defektfreien Displayflächen hergestellt werden.

Die Untersuchung des Verhaltens eines Systems flüssigkristalliner Polymerbürsten in nematischer Lösung könnte eine Alternative aufzeigen.

Kapitel 2

Modellbeschreibung und Simulationstechnik

2.1 Das Modell

Das in dieser Arbeit verwendete Modell geht auf ein Anfang der siebziger Jahre von Berne und Pechukas [25] entwickeltes und später von Gay und Berne [26] modifiziertes Modell zurück. Dieses Modell, das ursprünglich für die Simulation von allgemeinen asphärischen Molekülen entwickelt worden war, wurde dergestalt modifiziert, daß es für die Simulation von linearen Mehrzentren-Molekülen Anwendung finden konnte.

Bei linearen Mehrzentren-Molekülen, die mittels Lennard-Jones-artigen Potentialen miteinander wechselwirken, steigt die Rechenzeit, die zur Berechnung der Wechselwirkungsenergie zwischen zwei Moleküle benötigt wird, bedingt durch eine Doppelsumme über alle Wechselwirkungszentren der beiden an der Wechselwirkung beteiligten Moleküle, quadratisch mit der Anzahl der Wechselwirkungszentren an.

Um dem entgegenzuwirken, schlugen Gay und Berne ein Einzentren-Potential vor, welches die jeweilige relative Orientierung der beiden Moleküle zueinander berücksichtigt und somit durch Vermeidung dieser Doppelsumme die Wechselwirkungsenergie zweier elongierter Moleküle schneller berechnen kann.

Dieses Potential stellt sich wie folgt dar:

$$V(\hat{u}_1, \hat{u}_2, \vec{r}) = \epsilon(\hat{u}_1, \hat{u}_2, \hat{r}) \left[\left(\frac{1}{r - \sigma(\hat{u}_1, \hat{u}_2, \hat{r}) + 1} \right)^{12} - \left(\frac{1}{r - \sigma(\hat{u}_1, \hat{u}_2, \hat{r}) + 1} \right)^{6} \right] \quad (2.1)$$

Dabei sind die einzelnen Terme wie folgt definiert:

$$\epsilon(\hat{u}_1, \hat{u}_2, \hat{r}) = \epsilon^{\nu}(\hat{u}_1, \hat{u}_2)\, \epsilon'^{\mu}(\hat{u}_1, \hat{u}_2, \hat{r}) \quad (2.2)$$

$$\epsilon(\hat{u}_1, \hat{u}_2) = \epsilon_0 \left[1 - \chi^2 (\hat{u}_1 \cdot \hat{u}_2)^2 \right]^{-1/2} \quad (2.3)$$

$$\epsilon'(\hat{u}_1, \hat{u}_2, \hat{r}) = 1 - \frac{\chi'}{2} \left[\frac{(\hat{r} \cdot \hat{u}_1 + \hat{r} \cdot \hat{u}_2)^2}{1 + \chi'(\hat{u}_1 \cdot \hat{u}_2)} + \frac{(\hat{r} \cdot \hat{u}_1 - \hat{r} \cdot \hat{u}_2)^2}{1 - \chi'(\hat{u}_1 \cdot \hat{u}_2)} \right] \quad (2.4)$$

$$\sigma(\hat{u}_1, \hat{u}_2, \hat{r}) = \sigma_0 \left(1 - \frac{\chi}{2} \left[\frac{(\hat{r} \cdot \hat{u}_1 + \hat{r} \cdot \hat{u}_2)^2}{1 + \chi(\hat{u}_1 \cdot \hat{u}_2)} + \frac{(\hat{r} \cdot \hat{u}_1 - \hat{r} \cdot \hat{u}_2)^2}{1 - \chi(\hat{u}_1 \cdot \hat{u}_2)} \right] \right)^{-\frac{1}{2}} \quad (2.5)$$

$$\chi' = \frac{\epsilon_s^{1/\mu} - \epsilon_l^{1/\mu}}{\epsilon_s^{1/\mu} + \epsilon_l^{1/\mu}} \quad (2.6)$$

$$\chi = \frac{\sigma_{\parallel}^2 - \sigma_{\perp}^2}{\sigma_{\parallel}^2 + \sigma_{\perp}^2} \quad (2.7)$$

Die in den Gleichungen (2.2)–(2.7) auftretenden Größen haben dabei folgende Bedeutung:

ϵ_0 und σ_0 sind Konstanten, die die Energie- bzw. Längenskala festlegen, $\sigma_{\parallel}/\sigma_{\perp}$ ist das Längenverhältnis der betrachteten Elipsoide, ϵ_s und ϵ_l sind die angestrebten Potentialstärkeparameter für Seite-an-Seite- bzw. Ende-an-Ende-Konfigurationen. $\hat{u}_1$und $\hat{u}_1$ sind Einheitsvektoren, die die Richtung von $\sigma_{\parallel}$ (also die Orientierung) der Teilchen 1 bzw. 2 angeben, und $\hat{r}$ ist der normierte Verbindungsvektor der Mittelpunkte der beiden Teilchen.

Bei Simulationen dieses Modells [27, 28, 29, 30, 31] hat sich gezeigt, daß in diesem Modell neben der isotropen sowohl eine nematische als auch verschiedene smektische Phase auftreten können. Weiterhin können für $\sigma_{\parallel}/\sigma_{\perp} < 1$ auch verschiedene diskotische Phasen beobachtet werden [32, 33, 34].

Um diese smektische Phasen zu vermeiden, die durch den anziehenden Teil des Potentials und die Richtungsabhängigkeit von ϵ (Seite-an-Seite-Konfigurationen sind energetisch günstiger als Ende-an-Ende-Konfigurationen) verursacht werden, wurden die folgenden Änderungen des ursprünglichen Potentials eingeführt:

Das Potential wurde im Minimum abgeschnitten, nach oben zur x-Achse verschoben und von dort stetig mit Null fortgesetzt. Damit ist das Potential rein abstoßend. Weiterhin wurde (im Gegensatz zu beispielsweise [31]) ϵ richtungsunabhängig (also konstant) gesetzt. Physikalisch betrachtet entspricht dies einer Beschränkung auf die sterischen Eigenschaften des Systems.

Ein weiterer Vorteil der Beschränkung der Wechselwirkungsreichweite eines einzelnen Teilchens liegt in der Reduktion der Anzahl der Wechselwirkungspartner eines Teilchens, und somit in der Reduktion der für die Berechnung der Wechselwirkungsenergien eines Teilchens benötigten Rechenzeit.

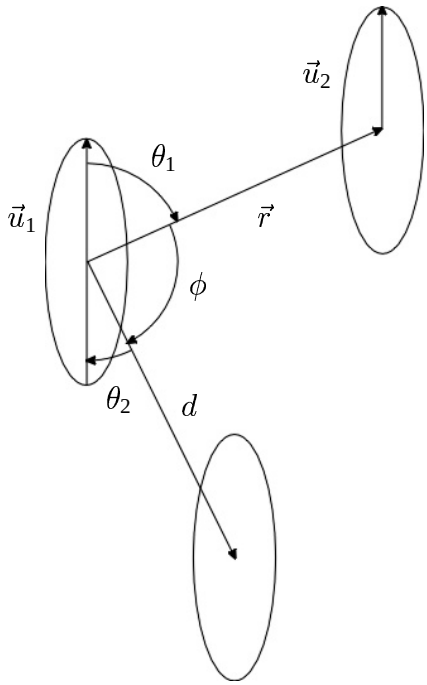

Abbildung 2.1: Skizze der verwendeten Winkel

Damit stellt sich das in dieser Arbeit verwendete Paarpotential für die Wechselwirkung zwischen zwei Lösungsmittelteilchen wie folgt dar:

$$V(\hat{u}_1, \hat{u}_2, \vec{r}) \;=\; \epsilon_0 \left[\left(\frac{1}{r - \sigma(\hat{u}_1, \hat{u}_2, \hat{r}) + 1} \right)^{12} - \left(\frac{1}{r - \sigma(\hat{u}_1, \hat{u}_2, \hat{r}) + 1} \right)^{6} + 1 \right] \quad (2.8)$$

mit

$$\sigma(\hat{u}_1, \hat{u}_2, \hat{r}) \;=\; \sigma_0 \left(1 - \frac{\chi}{2} \left[\frac{(\hat{r} \cdot \hat{u}_1 + \hat{r} \cdot \hat{u}_2)^2}{1 + \chi(\hat{u}_1 \cdot \hat{u}_2)} + \frac{(\hat{r} \cdot \hat{u}_1 - \hat{r} \cdot \hat{u}_2)^2}{1 - \chi(\hat{u}_1 \cdot \hat{u}_2)} \right] \right)^{-\frac{1}{2}} \quad (2.9)$$

$$\chi \;=\; \frac{\sigma_\parallel^2 - \sigma_\perp^2}{\sigma_\parallel^2 + \sigma_\perp^2} \quad (2.10)$$

Dies entspricht einem im Minimum abgeschnittenen und nach oben verschobenen Lennard-Jones-Potential zwischen zwei Ellipsoiden. Die Funktion σ nähert dabei den Kontaktabstand zweier harter Ellipsoide der Orientierungen $\hat{u}_1$ und $\hat{u}_2$ in Richtung $\hat{r}$ an.

Zunächst wurde das Bulk-Verhalten dieses Modells untersucht. Dieses wurde dann durch die Einführung von an den Wänden verankerten Ketten von Lösungsmittelteilchen und deren Wechselwirkungen über Bondlängen und Bondwinkelpotentiale zum Modell für Polymerbürsten in nematischem Lösungsmittel erweitert. Die Wechselwirkungen der Teilchen der Polymerbürsten untereinander und mit den Lösungsmittelteilchen genügen ebenfalls Gleichung (2.8), hinzu kommen jedoch Wechselwirkungen entlang einer Polymerkette, innerhalb derer ein Teilchen mit seinen durch Bonds verbundenen Nachbarn über zusätzlich eingeführte Bondlängen- und Bondwinkelpotentiale wechselwirkt (siehe Abb. 2.1).

Als Bondlängenpotential kam ein FENE-Potential mit einem Minimum bei d_0 zur Verwendung. Es hat die Form

$$V_{Bl}(d) \;=\; -\frac{c_{Bl}}{2}\, {d_{Bl}}^2 \ln\left(1 - \frac{(d - d_0)^2}{{d_{Bl}}^2} \right) \;:\; |d - d_0| \leq d_{Bl} \quad (2.11)$$

Dabei ist d der Abstand zweier entlang einer Polymerkette benachbarter Teilchen, c_{Bl} die Stärke der Wechselwirkung und d_{Bl} der Abstand, bei dem das Bondlängenpotential divergiert.

Ein Bondwinkelpotential der Form (Gleichung (2.12)) sorgt dafür, daß gestreckte Ketten begünstigt werden. Dabei sind θ_1, θ_2 die Winkel zwischen den Bindungen eines Teilchens und seiner Ausrichtung.

$$V_{Bw1}(\theta_1, \theta_2) \; = \; c_{Bw} \left((1 - \cos(\theta_1)) \; + \; (1 - \cos(\theta_2)) \right) \tag{2.12}$$

Zusätzlich wurde ein weiteres Bondwinkelpotential eingeführt, das das Zusammenfalten von Polymerketten unterdrückt. Dieses Potential (Gleichung (2.13)) versucht den Winkel ϕ zwischen zwei benachbarten Bonds zu strecken.

$$V_{Bw2}(\theta) \; = \; 2\, c_{Bw} \, (1 + \cos(\phi)) \tag{2.13}$$

Damit ein Teilchen nicht eine der Wände des Systems durchdringen kann, muß sein Abstand zur Wand unter Berücksichtigung seiner Orientierung ermittelt werden. Da die Wände des Systems als harte, nur über 'excluded volume'-Wechselwirkung mit den Teilchen wechselwirkende Wände modelliert wurden, ergibt sich nur bei einem Überlappen zwischen Teilchen und Wand ein Energiebeitrag von unendlich.

Das Wandpotential hat somit folgende Form:

$$V_{Wand} \; = \; \{ \begin{array}{ll} \infty & f\ddot{u}r \mid z_p - z_a \mid \leq 0 \\ 0 & sonst \end{array} \tag{2.14}$$

(2.15)

Dabei ist z_p die z-Komponente des Schwerpunktes des Teilchens und z_a die halbe Ausdehnung des Teilchens in z-Richtung. Diese wird mittels

$$z_a \; = \; \sqrt{\frac{\sigma_\perp^2}{4} + \cos^2\phi \left(\frac{\sigma_\parallel^2 - \sigma_\perp^2}{4} \right)} \tag{2.16}$$

berechnet (Herleitung siehe Anhang), wobei $\cos\phi$ der z-Komponente der normierten Teilchenorientierung entspricht und $\sigma_\parallel$ bzw. $\sigma_\perp$ die Länge der großen bzw. der kleinen Achse des Teilchens ist.

Im Rahmen dieser Arbeit wurden (sofern nicht anders angegeben) für die Simulation die folgenden Potentialparameter gewählt:

ϵ_0 und σ_0 wurden beide auf 1 festgelegt und legen somit die Längen- bzw. Energieskala fest. $\sigma_\perp$ wurde zu 1 und $\sigma_\parallel$ zu 3 gewählt, wie es auch die in der Literatur häufigste Wahl dieser Parameter ist (z.B. in [28]). Weiterhin ist es ein guter Kompromiss zwischen einer vernünftig zu simulierenden Systemgröße und einem hinreichend elongierten Teilchen.

Die Potentialstärken der Bondwinkelpotentiale und des Bondlängenpotentials c_{Bl} und c_{Bw} wurden beide auf 10 gesetzt, um eine nicht zu steife Kette zu erhalten. Als Länge eines Bonds im Ruhezustand wurde $d_0 = 4$ gewählt und die maximale Abweichung davon war $d_{bl} = d_0/5 = 0.8$.

2.2 Simulation im NpT-Ensemble

In den durchgeführten Simulationen wurde ein Metropolis [39] Monte Carlo-Algorithmus verwendet. Dabei wird zufällig ein Teilchen ausgewählt und dessen Orts- und/oder Winkelkoordinaten zufällig verändert. Dies entspricht einer zufälligen Verrückung bzw. einer Drehung um eine zufällige Achse. Die Veränderung der Winkelkoordinaten wurde dabei so realisiert, daß das Vektorprodukt aus einem Zufallsvektor mit einer Länge kleiner als eins und der alten Teilchenorientierung, multipliziert mit einem Längenfaktor, zur alten Teilchenorientierung addiert und auf eins normiert wurde.

Danach wird die Energie des betrachteten Teilchens an den neuen Koordinaten E_{neu} mit der Energie des Teilchens an den alten Koordinaten E_{alt} verglichen. Ist die Differenz $\Delta E = E_{neu} - E_{alt} < 0$, so wird der Bewegungsversuch auf jeden Fall (also mit Wahrscheinlichkeit 1) akzeptiert, ist $\Delta E > 0$, wird der Bewegungsversuch nur mit der Wahrscheinlichkeit $\exp(-\Delta E/T)$ (k_B wurde eins gesetzt) angenommen.

Die maximalen Sprungweiten bzw. die Größe des Längenfaktors wurden für jeden Meßlauf so gewählt, daß für beide Bewegungsversuche der Teilchen jeweils eine Akzeptanzrate von $\approx$ 30% erreicht wurde.

Wegen der beschränkten Wechselwirkungsreichweite (Soft-Core-Wechselwirkung der Teilchen) war es möglich die Simulationsbox in mehrere Unterboxen aufzuteilen, deren Kantenlängen so gewählt wurden, daß ein Teilchen nur mit den Teilchen in der gleichen und den 26 benachbarten Unterzellen wechselwirken kann. Die Speicherung der Teilchen erfolgt mittels eines Link-Cell-Algorithmus, d. h. für jede Unterzelle wird eine verkettete Liste geführt, in der alle sich innerhalb dieser Zelle befindlichen Teilchen aufgeführt sind. Somit kann die zur Berechnung der Gesamtenergie nötige Doppelsumme über alle Teilchen-Wechselwirkungen innerhalb der Simulationsbox auf eine Doppelsumme über die Teilchen in den entsprechenden Unterzellen reduziert werden.

Darüber hinaus kam ein Verlet-Algorithmus, der für die Simulation von asphärischen Teilchen modifiziert wurde, zur Anwendung. Dabei wird für jedes Teilchen eine Liste angelegt, für die diejenigen Teilchen aus der Linked-Cell-Struktur ermittelt werden, die tatsächlich mit dem Teilchen wechselwirken können, d. h. es werden alle Teilchen mit Wechselwirkungszentrum innerhalb einer Kugel des Radius der maximalen Wechselwirkungsreichweite um das Wechselwirkungszentrum des betrachteten Teilchens in die Liste aufgenommen.

Die normalerweise (also bei Systemen kugelförmiger Teilchen) um das Zentrum des wechselwirkenden Teilchen gelegte Verletkugel wurde in dieser Arbeit außerdem durch zwei sich überlappende Kugeln ersetzt, die jeweils um die Mitte der Verbindungslinie zwischen der Mitte des Teilchens und seinen Enden gelegt wurden. Dabei ist der Radius dieser Kugeln halb so so groß wie die größte Ausdehnung eines Teilchens plus die maximale Bewegungsweite eines der Kugelzentren.

Die Verlettabelle für ein Teilchen speichert also alle anderen Teilchen, bei denen mindestens eines seiner Verletkugelzentren innerhalb einer der Verletkugeln des ersten Teilchens liegt. Somit ergibt sich gegenüber einer einzigen Kugel um den Mittelpunkt des Teilchens eine erheblich geringere Anzahl von möglichen Wechselwirkungspartnern in der Verlettabelle eines Teilchens und somit eine deutliche Beschleunigung der Simulation, da viele überflüssige Abstandsberechnungen, die zu einem Energiebeitrag von Null führen, entfallen können. Dies führt zu einer weiteren Reduktion der Zeit, die für die Energieberechnung benötigt wird.

Damit nicht jede Änderung einer Teilchenposition eine komplette Neuberechnung aller Verlettabellen und Wechselwirkungsenergien zur Folge hat, wurde die eigentliche Verletkugel um eine Verletschicht erweitert. Diese Schicht ermöglicht es, daß erst dann die Verlettabellen neu berechnet werden müssen, wenn die Summe der beiden größten Veränderungen von Teilchenpositionen seit dem letzten Update der Verlettabellen die Dicke der Verletschicht überschreitet.

Die Dicke dieser auf den Verletkugeln aufgesetzten Verletschicht wurde während der Simulation jeweils so angepaßt, daß die Laufzeit für die einzelnen Simulationsläufe möglicht gering wurde.

In einer weiteren Tabelle wurden für jedes Teilchen zusätzlich auch die aktuellen Wechselwirkungsenergien des Teilchens mit den in seiner Verlet-Liste aufgeführten Wechselwirkungspartnern abgespeichert. Somit konnte innerhalb des Monte Carlo-Schrittes die Berechung der Energie des Teilchens am alten Ort auf eine Summe der Wechselwirkungsenergien dieses Teilchens aus seiner Energietabelle reduziert werden, was ebenfalls zu einer Beschleunigung der Simulation führte.

Die Simulationen wurden, mit Ausnahme einiger Voruntersuchungen, für die das NVT-Ensemble verwendet wurde, im NpT-Ensemble, d.h. bei konstanter Teilchenzahl, konstantem Druck und konstanter Temperatur, durchgeführt. Dafür ist es nötig, auch kollektive Bewegungsversuche durchzuführen, die das Volumen der Simulationsbox und somit auch die Dichte verändern können.

Diese Volumenmoves wurden wie folgt realisiert: Die Grundfläche der Simulationsbox (in x- und y- Richtung wurden jeweils periodische Randbedingungen angenommen) wurde konstant gehalten, und nur in Richtung der z-Achse wurden Fluktuationen der Boxgröße ermöglicht. Eine einfache Reskalierung sämtlicher Kanten der Simulationsbox (Volumenatmung wie z. B. in [40] beschrieben) ist in diesem Fall nicht sinnvoll, da durch eine Reskalierung der x- und y-Achse die Pfropfdichte der auf der Grundfläche verankerten Polymerbürsten verändert würde.

Für das Metropoliskriterium verändert sich bei einem Volumenmove ΔE zu

$$\Delta E = E_{neu} - E_{alt} + p\,\Delta V - N_{sys}\,T\,\ln\left(\frac{z_{neu}}{z_{alt}}\right) \tag{2.17}$$

Dabei ist p der von außen vorgegebene Druck, ΔV die Veränderung des Volumens, N_{sys} die Zahl der Teilchen in der Simulationsbox, T die Temperatur und $z_{neu}\ sowie\ z_{alt}$ die Ausdehnung der Simulationsbox in z-Richtung nach bzw. vor dem Bewegungsversuch. Der zusätzliche dritte Term in Gleichung (2.17) ist die Volumenarbeit gegen den äußeren Druck, und der vierte Term berücksichtigt das für die Teilchen zur Verfügung stehende Volumen vor bzw. nach dem Bewegungsversuch.

Auch für die Volumenmoves (Reskalierung der z-Achse) wurde eine Akzeptanzrate von $\approx$ 30% verwendet. Da dies eine komplette Neuberechnung sämtlicher Wechselwirkungsenergien erzwingt, setzt sich, unter anderem auch aus Effizienzgründen, ein Monte Carlo-Schritt (MCS) im Mittel aus je einem versuchten Volumenmove, N_{sys} Ein-Teilchen-Verschiebungsversuchen und zwei N_{sys} Versuchen, ein Teilchen zu drehen, zusammen, wobei auch hier N_{sys} die Anzahl der Teilchen ist. Die Art des nächsten auszuführenden Bewegungsversuchs wird jeweils zufällig bestimmt. Nach jeweils 500 Monte Carlo-Schritten wurden Meßwerte aufgenommen.

2.3 Die untersuchten Größen

Die folgenden Größen wurden während der Simulationsläufe ermittelt bzw. aus den während der Simulation herausgeschriebenen Konfigurationen gewonnen:

- Innere Energie pro Teilchen U

  Die innere Energie pro Teilchen ist die Summe aller Energiebeiträge innerhalb des Systems auf Grund von Wechselwirkungspotentialen, dividiert durch die Gesamtzahl der im System vorhandenen Teilchen.

- Ordnungsparameter Θ

  Als Ordnungsparameter wurde in dieser Simulation der nematische Ordungsparameter verwendet. Er ist definiert als der größte Eigenwert der Matrix

  $$S_{ij} = \frac{1}{2N} \sum_{n=1}^{N} (3\, x_i^{(n)}\, x_j^{(n)} - \delta_{ij}) \tag{2.18}$$

  wobei die Summe über alle Teilchen in der Simulationsbox N läuft und $i, j \in (1, 2, 3)$ sind.

- Schichtabhängiger Ordnungsparameter $\Theta(d)$

  Auch hierbei handelt es sich um den oben definierten nematischen Ordnungsparameter, wobei die Summe hier nur über alle Teilchen in einer Scheibe der Simulationsbox läuft. Die Dicke dieser Scheibe beträgt $2\sigma_s$.

- Dichte ρ

  Als Dichte wurde die Teilchendichte, also die Anzahl aller Teilchen (Lösungsmittel und Ketten) geteilt durch das Volumen, betrachtet. Da die Grundfläche der Simulationsbox konstant gehalten wurde, entspricht die Veränderung der Dichte der Veränderung der Ausdehnung der Box in z-Richtung bei gleicher Teilchenzahl.

- Schichtabhängige Dichte $\rho(d)$

  Die schichtabhängige Dichte errechnet sich aus der Anzahl der Teilchen in einer Schicht der Simulationsbox geteilt durch das Volumen dieser Scheibe. Die Dicke der Scheiben hierbei beträgt $0.2\sigma_s$.

- Neigungswinkel der Polymerbürsten θ_p

  Der Neigungswinkel der Polymerbürsten ist der über alle Ketten gemittelte Winkel zwischen dem Verbindungsvektor des Kettenkopfes mit den dazugehörenden Endteilchen (Kopf-Ende-Vektor) und der z-Achse.

- Bondlänge l

  Die Bondlänge ist der über alle Bindungen gemittelte Abstand zwischen den Mittelpunkten zweier entlang einer Kette benachbarten Teilchen.

- Winkel zwischen den Bindungen ϕ

  Dies ist der Mittelwert der Winkel, der sich zwischen zwei entlang einer Kette benachbarten Bonds bildet. Pro Kette wird über (Anzahl der Teilchen pro Kette minus zwei) Winkel gemittelt.

- Winkel zwischen Bond und Teilchendirektor θ_1

  Dabei handelt es sich um den Winkel zwischen dem Orientierungsvektor eines Kettenteilchens und den Bindungen zu seinen Nachbarteilchen entlang der Kette.

- Druck p

  Der Drucktensor $p_{\alpha\beta}$ wurde während der Simulation mittels des Virial-Theorems

$$p_{\alpha\beta} = \frac{1}{V}\left(NT\,\underline{\mathbf{1}} + \sum_{\alpha,\beta}\left\{\vec{r_\alpha}\vec{F_\beta}\right\}\right) \tag{2.19}$$

  gemessen und mit dem von außen für die Simulation voreingestellten Druck verglichen.

2.4 Configurational Biased Monte Carlo

Im Laufe dieser Arbeit zeigte sich, daß die Relaxation der Polymerbürsten mittels Einteilchen-Bewegungsversuchen außerordentlich langsam ist. Dies wurde durch die Simulation einer einzelnen Polymerkette im Lösungsmittel deutlich.

Eine einzelne Kette sollte sich unserer Erwartung nach, unabhängig von ihrer Anfangsposition flach entlang der Oberfläche der Wand, auf der sie verankert wurde, legen. Der Grund dafür ist der Umstand, daß der Einfluß einer einzelnen Kette auf die Orientierung gegen den Einfluß des Lösungsmittel vernachlässigt werden kann. Bedingt durch eben diese Lösungsmittelteilchen, die die Kette in ihrer Bewegung hindern, konnte die einzelne Kette jedoch nicht in diesen Zustand gelangen und das System (und somit auch die einzelne Kette) blieb in einem frustrierten Zustand hängen, in dem die Kette einen Neigungswinkel gegen die z-Achse aufwies, ohne sich jedoch weiter neigen zu können.

Damit das System eine bessere Möglichkeit zur Equilibrierung erhält, wurde ein Algorithmus implementiert, der eine komplette Polymerkette aus dem System entfernt und dann neu wachsen läßt (ähnlich Configurational Biased MC). Dabei wurde wie folgt vorgegangen:

Zuerst wurden alle Bonds einer zufällig ausgewählten Kette entfernt. Danach wurden all diejenigen Teilchen ermittelt, zu denen ein Bond mit einer Länge d mit $|d - d_0| < d_{Bl}$ gebildet werden konnte. Aus diesen möglichen neuen Bindungen wurde gemäß der Boltzmann-Verteilung eine neue Bindung für diese Kette gewählt und von diesem neue Teilchen der Kette wiederum nach neuen Bindungen gesucht. Diese Vorgehensweise wurde solange wiederholt, bis wieder eine Kette der ursprünglichen Länge generiert worden war. Über eine abschließende Monte Carlo-Abfrage wurde dann die Entscheidung über Annahme oder Ablehnung der gesamten neu gebauten Kette getroffen.

Diese Art des Vorgehens hat Ähnlichkeit mit dem Configurational Biased Monte Carlo (kurz CBMC). In [41] beschreiben D. Frenkel et. al. eine Variante des CBMC-Algorithmus, die sich mit Modifikationen auf das hier untersuchte Modell anwenden läßt:

Wählt man einen neuen Bond „naiv“ zufällig aus, so wird mit einer sehr großen Wahrscheinlichkeit eine Konfiguration vorgeschlagen, die, zum Beispiel durch Teilchenüberlapp oder Kettendurchdringungen, durch die vorhandenen Potentiale mit einer hohen „Energiestrafe“ belegt wird und somit ein verschwindendes Boltzmanngewicht hat. Demzufolge werden nur sehr wenige dieser Konfigurationen vom MC-Prozeß angenommen.

Bei allen anderen Methoden, bei denen die neuen Bonds „intelligenter“, d.h. energetisch günstiger, generiert werden, ist darauf zu achten, daß die $detailed\ balance$ nicht verletzt wird, d.h. daß für zwei Zustände X und Y die Übergangsrate $K(X \mapsto Y)$ ebenso groß wie die Übergangsrate des umgekehrten Falls ist.

$$K(X \mapsto Y) = K(Y \mapsto X) \tag{2.20}$$

Die Rate $K(X \mapsto Y)$ setzt sich dabei wie folgt zusammen:

$$K(X \mapsto Y) = N_X \cdot P_Y \cdot A(X \mapsto Y) \tag{2.21}$$

Dabei ist N_X die Wahrscheinlichkeit, den Zustand X vorliegen zu haben, P_Y die Wahrscheinlichkeit, den Zustand Y als neuen Zustand vorzuschlagen und $A(X \mapsto Y)$ die Akzeptanzwahrscheinlichkeit für diesen Vorschlag.

Wenn man davon ausgeht, daß die Zustände einer Boltzmannverteilung $N_X \sim \exp(-\beta U(X))$ gehorchen, erkennt man, daß das Metropoliskriterium [39]

$$A(X \mapsto Y) = min\left(1, \frac{P_X \,/\, \exp(-\beta U(X))}{P_Y \,/\, \exp(-\beta U(Y))}\right) \tag{2.22}$$

die Forderung der $detailed\ balance$ erfüllt.

In „einfachen“ Monte Carlo-Algorithmen würden die Zustände mit gleicher Wahrscheinlichkeit vorgeschlagen. Wegen $P_X = P_Y$ hängt die Akzeptanz somit nur noch vom Energieunterschied $\Delta U(X \mapsto Y)$ zwischen den beiden Zuständen ab.

Bei dem hier betrachteten Verfahren geht es darum, für ein Polymer X der Länge l durch Abschneiden und Neuwachsen eines Endes eine neue Konfiguration Y zu gewinnen.

Betrachtet man diesen Vorgang vorübergehend für ein **Gittermodell**, gibt es für einen neu angefügten Bond eine endliche Zahl k von Möglichkeiten.

Die $Rosenbluth - Methode$ [42], ein Polymer neu wachsen zu lassen, sieht vor, aus diesen k Bonds, die an den $i-1$ langen Rumpf angefügt werden können, einen $1 \leq j \leq k$ gemäß seines Boltzmanngewichts $\exp(-\beta u_j^{(i)})$ auszuwählen. $u_j^{(i)}$ ist dabei die innere Energie der Bondposition j und enthält die Wechselwirkung mit den anderen Molekülen, sowie mit den $i-1$ Rumpfmonomeren, nicht aber die abgetrennten Monomere $i+1 ... l$.

Die Wahrscheinlichkeit der entstandenen Konfiguration beträgt

$$P_Y = \prod_{i=1}^{l} \frac{\exp(-\beta u_j^{(i)}(Y))}{Z_i(Y)} \quad Z_i(Y) := \sum_{j=1}^{k} \exp(-\beta u_j^{(i)}(Y)) \tag{2.23}$$

Der zugehörige $Rosenbluth - Faktor$ ist als

$$W_Y := \exp(-\beta u^{(l)}) \prod_{i=1}^{l} \frac{Z_i(Y)}{k} \tag{2.24}$$

definiert. Das Anfangsmonomer $(i = 1)$ wird nicht verändert, weswegen $u^{(l)}(Y) = u^{(l)}(X)$ gilt. Da die Wechselwirkung mit den noch abgetrennten Monomeren in die $u^{(l)}(Y)$ nicht eingeht, ergibt sich die Gesamtenergie der Kette Y zu $U(Y) = \sum_{i=1}^{l} u^{(i)}(Y)$. Hieraus folgt

$$P_Y \,/\, \exp(-\beta U(Y)) = \frac{1}{k^{l-1} W_Y} \quad P_X \,/\, \exp(-\beta U(X)) = \frac{1}{k^{l-1} W_X} \tag{2.25}$$

Wenn die Ketten mit ihrem korrekten Boltzmanngewicht in die Simulation eingehen sollen, müssen sie nach (2.22) mit einer Wahrscheinlichkeit

$$A(X \mapsto Y) = min\left(1, \frac{W_Y}{W_X}\right) \tag{2.26}$$

akzeptiert werden, d.h. entsprechend ihrer Rosenbluth-Faktoren.

Obwohl es sich bei der hier durchgeführten Simulation deutlich nicht um ein Gittermodell handelt, können dennoch viele der Konzepte, die für CBMC auf dem Gitter gelten, direkt übernommen werden. So ist zum Beispiel auch in dieser Simulation die Anzahl der Möglichkeiten, einen neuen Bond zu generieren, endlich. Dies liegt daran, daß die Ketten nicht komplett aus dem System entfernt werden, sondern nur deren Bindungen herausgenommen werden. Somit werden alle ehemaligen Teilchen (mit Ausnahme des Kettenkopfes) der ausgewählten Kette zu normalen Lösungsmittelteilchen.

Für den Neuaufbau der Kette, also die Generierung neuer Bindungen, stehen dadurch nur die vorhandenen Lösungsmittelteilchen zu Verfügung und deren Anzahl ist endlich. Als weitere Einschränkung können nur die Teilchen ausgewählt werden, zu denen Bonds gebildet werden können, deren Länge die Bedingung $|d - d_0| \leq d_{Bl}$ erfüllt – andernfalls divergiert das Bondlängenpotential.

Da sich durch den Aufbruch der alten Kette und die Generierung einer neuen Kette die Teilchenpositionen und somit auch die Paarwechselwirkungen im System nicht verändern, müssen bei der Berechnung der Boltzmann-Gewichte und Rosenbluth-Faktoren für den CBMC-Move auch nur die Bondlängen- und Bondwinkelpotentiale berücksichtigt und somit neu berechnet werden und nicht die Gay-Berne Wechselwirkung.

Der Vorteil, nur die Bindungen zu entfernen und nicht die gesamte Kette, liegt unter anderem in der nahezu verschwindenden Wahrscheinlichkeit, eine neue Kette in eine dichte Lösung wachsen zu lassen. Somit wären die Akzeptanzraten solcher Versuche minimal und würden die Relaxations- und Korrelationszeiten nicht erniedrigen.

Weiterhin müßten die Paarwechselwirkungen berücksichtigt werden – die Durchführung eines „Kettenmoves“ wäre daher auch mit erheblich größerem Rechenaufwand verbunden.

In der Simulation kamen somit die folgenden Formeln zur Anwendung:

Die Wahrscheinlichkeit, einen bestimmten Bond i aus den N möglichen Bonds (d.h. den Bonds, bei denen das Bondlängenpotential einen endlichen Energiebeitrag liefert) als neuen k-ten Bond einer Kette auszuwählen, beträgt

$$p_k^{(i)} = \frac{\exp\left(-\frac{1}{T}\left(V_{Bw1}^{(i)} + V_{Bl}^{(i)} + (1-\delta_{k,1})V_{Bw2}^{(i)}\right)\right)}{\sum\limits_{j=1}^{N} \exp\left(-\frac{1}{T}\left(V_{Bw1}^{(j)} + V_{Bl}^{(j)} + (1-\delta_{k,1})V_{Bw2}^{(j)}\right)\right)} \tag{2.27}$$

und die Wahrscheinlichkeit für die gesamte neue Kette der Länge n (d. h. mit $(n-1)$ Bindungen) beträgt somit

$$P^{(neu)} = \prod_{k=1}^{n-1} p_k^{(i)} \tag{2.28}$$

Weiterhin muß $P^{(alt)}$, die Wahrscheinlichkeit, die ehemals existierende Kette zu bilden, neu berechnet werden. Dies ist deshalb vonnöten, da sich seit der letzten Berechnung dieser Wahrscheinlichkeit die Umgebung der Kette verändert haben kann und somit die Anzahl der potentiellen Alternativketten auch nicht konstant geblieben sein dürfte.

Damit ergibt sich die folgende Metropolisabfrage, die über die Annahme oder Ablehnung einer neu gebildeten Kette entscheidet:

$$A(alt \mapsto neu) = \min\left(1, \frac{P^{(alt)}}{P^{(neu)}} \exp\left(-\frac{\Delta E}{T}\right)\right) \tag{2.29}$$

Dabei ist ΔE die Energiedifferenz zwischen der Konfiguration mit der neuen Kette und der Konfiguration mit der alten Kette. Da sich jedoch durch die Neubildung der Kette keine Teilchenpositionen verändert haben, reduziert sich ΔE auf die Differenz der Beiträge aus den Bondwinkelpotentialen und dem Bondlängenpotential der beiden Konfigurationen.

2.5 Test der Simulationsmethode

Um die Korrektheit des Algorithmus für den Bau neuer Ketten zu überprüfen, wurde bei konstantem Volumen ein System von Polymerbürsten im Lösungsmittel simuliert, bei dem jedoch alle Paarwechselwirkungen zwischen den einzelnen Teilchen ausgeschaltet wurden und die Teilchen nur über Bondlängen- und Bondwinkelpotentialen wechselwirkten. Die Verteilung der Bondwinkel und der Bondlängen sollte dabei mit einer Boltzmann-Verteilung für eine einzelne freie Kette übereinstimmen.

Diese Überprüfung wurde sowohl für ein System mit einer Kette (σ = 0.00694), als auch mit 16 Ketten (σ = 0.111) auf jeder Oberfläche bei einer Temperatur von $T = 0.5$ durchgeführt.

Wie aus den Abbildungen 2.2 und 2.3 zu ersehen ist, entsprechen die Ergebnisse dieser Simulationsläufe den aus der Theorie (Gleichung (2.30) (Herleitung siehe Anhang) bzw. (2.31)) für eine Bondwinkelstärke von $c_{Bw} = 100$ und eine Bondlängenstärke von $c_{Bl} = 10$ erwarteten.

$$P_{Bw}(\theta) \sim e^{-2\,\beta\, c_{Bw}\,(1+|\cos(\theta)|)}\, \frac{\sinh\left(\beta\, c_{Bw}\sqrt{2(1+|\cos(\theta)|)}\right)}{\sqrt{2(1+|\cos(\theta)|)}} \tag{2.30}$$

$$P_{Bl}(d) \sim e^{\frac{1}{2}\,\beta\, c_{Bl}\, d_{Bl}^2\, log\left(1-\frac{(d-d_0)^2}{d_{bl}^2}\right)} \tag{2.31}$$

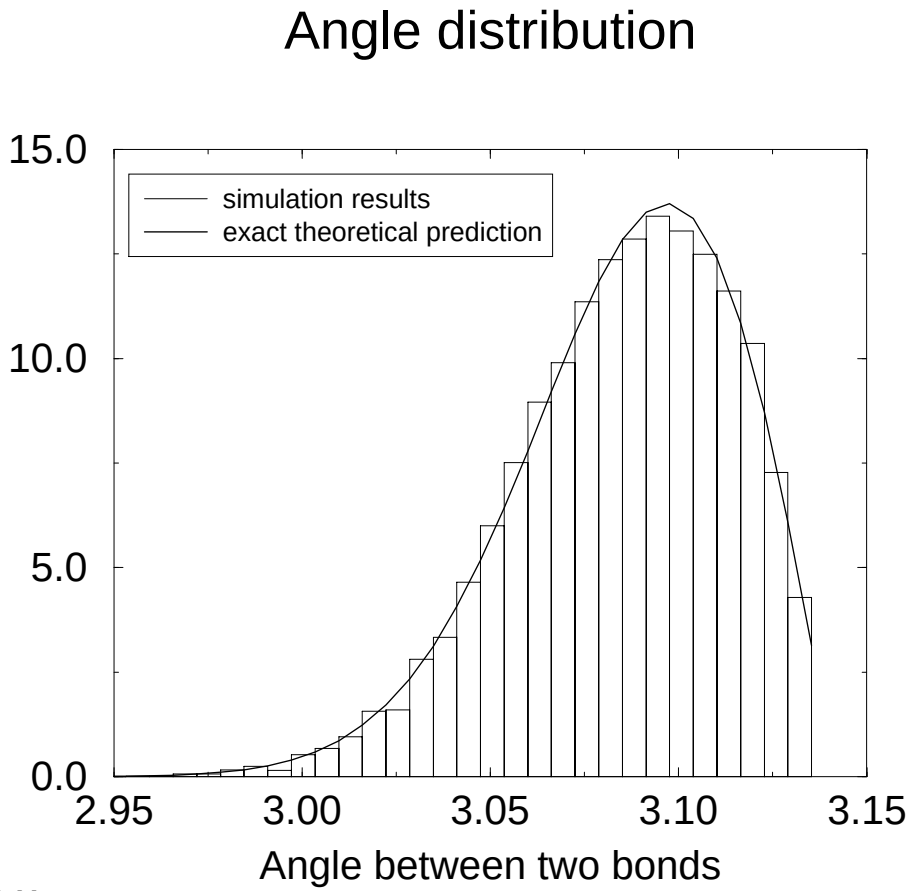

Abbildung 2.2:
Bondwinkelverteilung für ein System wechselwirkungsfreier Bürsten und Vergleich mit der theoretischen Kurve

Die in den Abbildungen 2.2 und 2.3 auftretenden geringen Abweichungen der Simulationsdaten von den exakten Daten sind auf die zu geringe Statistik des Simulationslaufes zurückzuführen.

Im Simulationsalgorithmus wurde die Wahrscheinlichkeit, daß der nächste Bewegungsversuch den „Neubau“ einer Kette enthält, ebenso groß gewählt wie die Wahrscheinlichkeit, einen Volumenmove durchzuführen.

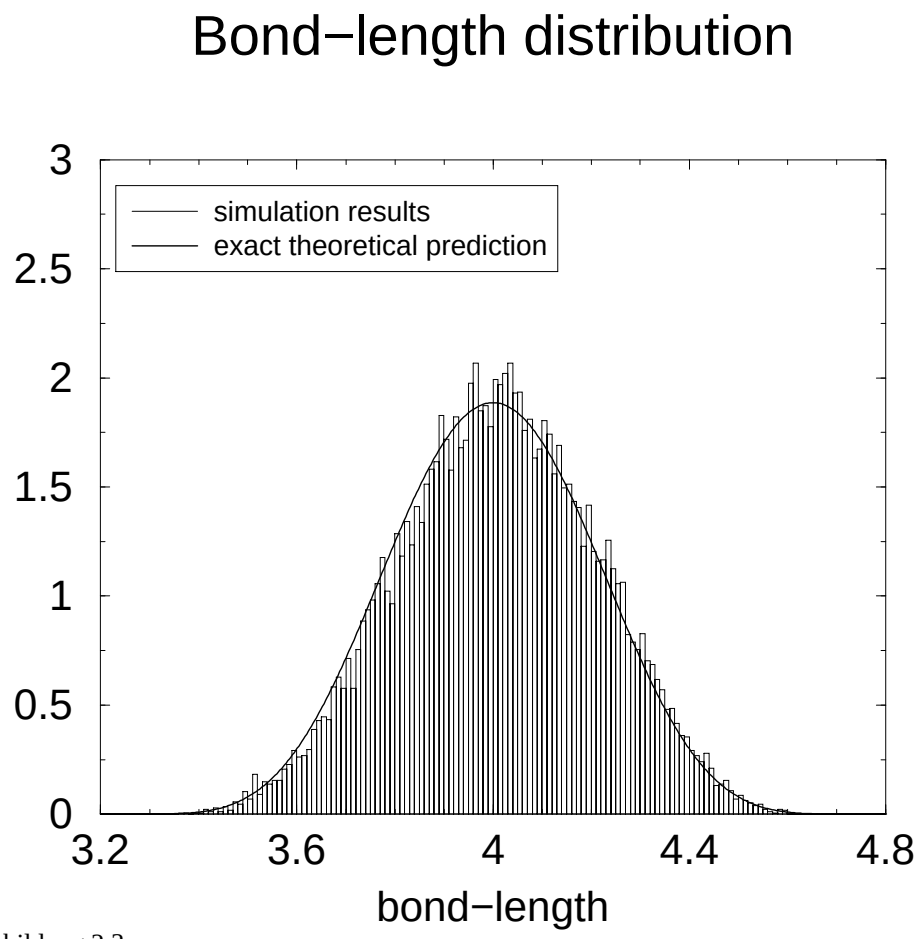

Abbildung 2.3:

Bondlängenverteilung für ein System wechselwirkungsfreier Bürsten und Vergleich mit der theoretischen Kurve

Kapitel 3

Voruntersuchungen

Um für eine Simulation eines komplexen Modells den relevanten Parameterbereich zu erfassen, ist es häufig hilfreich, die einzelnen Teile des Systems zuerst separat zu betrachten. Dadurch kann meist auch ein vernünftigeres Verhältnis zwischen Aufwand (Systemgrößen und Lauflängen) und zu erwartenden Ergebnissen (Dichte der Meßwerte und Genauigkeit) ermittelt werden. Somit ist es dann möglich, günstige Phasenraumpunkte und somit Eingabeparameter für die späteren, komplexeren und rechenzeitintensiveren Simulation zu bestimmen.

3.1 Das Bulk-Verhalten

Im Rahmen dieser Arbeit war es zunächst nötig, das Verhalten der Lösungsmittelteilchen im Bulk zu betrachten, um auf diese Weise ein Phasendiagramm zu gewinnen und ein Gefühl für das Verhalten der Teilchen in der Simulation zu bekommen.

Dafür wurde ein ausschließlich aus Lösungsmittelteilchen bestehendes System in einer Simulationsbox mit periodischen Randbedingungen in allen drei Raumrichtungen und der Möglichkeit, alle Kanten der Simulationsbox unabhängig voneinander zu reskalieren, betrachtet. Um einen Einblick in das Verhalten des Systems bei unterschiedlichen Teilchenzahlen (und die dabei eventuell zu erkennenden Finite-Size-Effekte) zu erhalten, wurde diese Untersuchung bei drei unterschiedlichen Systemgrößen (384, 660 und 980 Lösungsmittelteilchen) durchgeführt.

Die Simulationen wurden dabei als Heizläufe isobar durchgeführt, d.h. die Endkonfiguration eines Simulationslaufes bei niedriger Temperatur wurde als Startkonfiguration für den dem Simulationslauf bei höherer Temperatur vorausgehenden Relaxationslauf bei der neuen Temperatur verwendet.

Bei allen drei betrachteten Systemgrößen konnte ein Phasenübergang zwischen einer isotropen und einer nematischen Phase beobachtet werden (siehe auch Abb. 3.1). Weitere Phasen traten in dem dabei betrachteten Druck- und Temperaturbereich nicht auf.

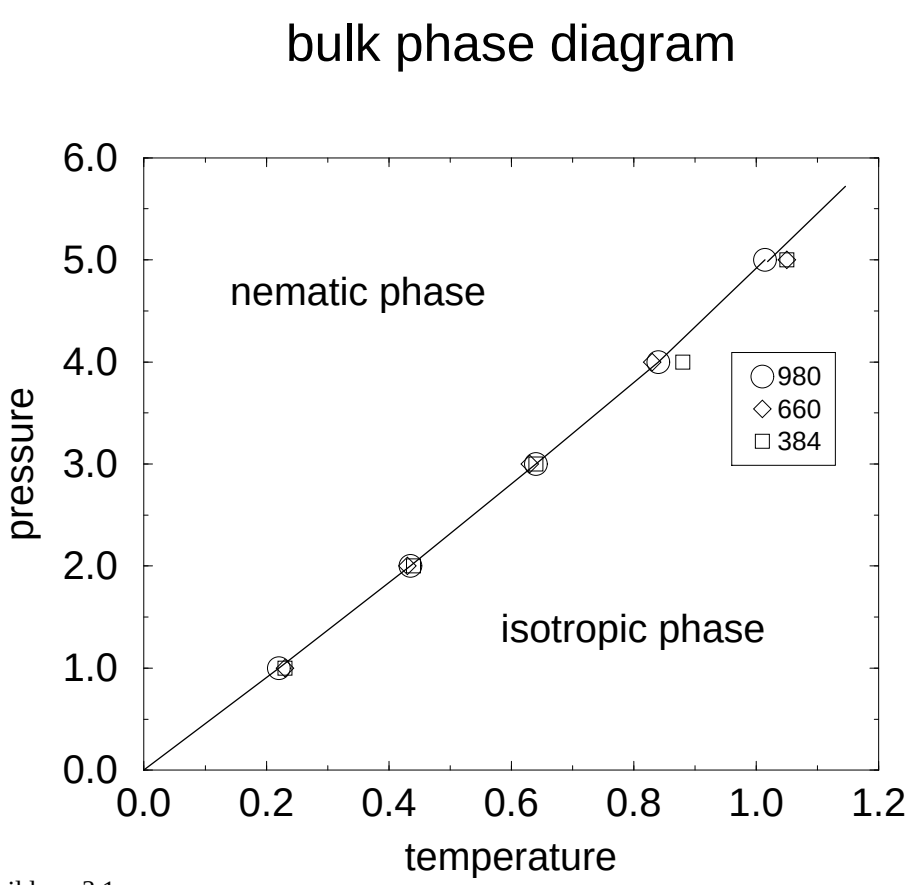

Abbildung 3.1:

Phasendiagramm in der p-T-Ebene für ein System mit 384, 660 und 980 Lösungsmittelteilchen ohne Wände und Bürsten

Dabei ist im gewonnenen Phasendiagramm auch gut zu erkennen, daß die Kurven für alle betrachteten Systemgrößen nahezu deckungsgleich aufeinander liegen, was, zumindest für das Bulk-System, darauf hindeutet, daß Finite-Size-Effekte hier vernachlässigt werden können.

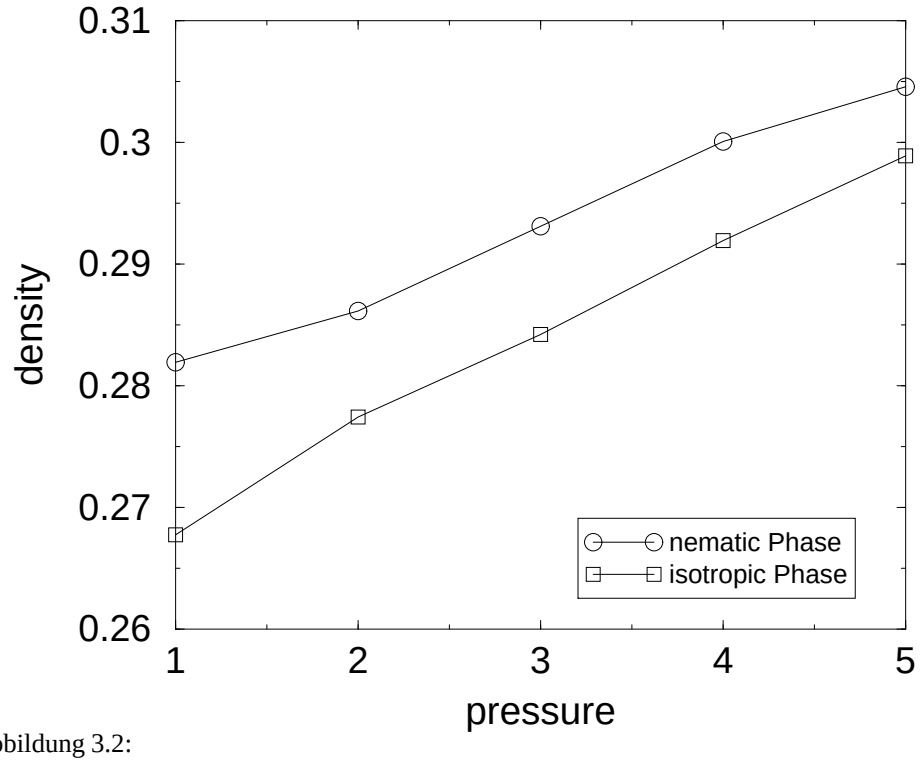

Abbildung 3.2:

Phasendiagramm in der ρ-p-Ebene für ein System mit 980 Lösungsmittelteilchen ohne Wände und Bürsten

In Abbildung 3.2 ist die Dichte am Phasenübergang in der nematischen und in der isotropen Phase gegen den Druck aufgetragen. Der Dichtesprung zwischen den beiden Phasen wird mit steigendem Druck kleiner und wandert mit steigendem Druck zu höheren Dichten. Die Fehlerbalken in dieser Abbildung sind jedoch deutlich größer als die Symbole, was sicherlich auf die geringe Systemgröße zurückzuführen ist. Eine ausgiebigere Untersuchung würde dieses Bild sicherlich verbessern, ist im Rahmen dieser Arbeit allerdings zu Gunsten der eigentlichen Fragestellungen unterblieben.

3.2 Polymerbürsten ohne Lösungsmittel

Um festzustellen, bis zu welcher Pfropfdichte der Polymerbürsten es möglich ist, daß die Kette quillt, d.h. daß sich die Lösungsmittelteilchen zwischen die Ketten setzen können, wurden Simulationen durchgeführt, bei denen lediglich auf den beiden Oberflächen der Simulationsbox verankerte Polymerbürsten ohne Lösungsmittelteilchen simuliert wurden.

Dazu wurden die periodischen Randbedingungen nur in x- und y-Richtung beibehalten und am oberen und unteren Ende der Simulationsbox je eine Wand eingeführt. Die Polymerbürsten wurden dabei mit in beide Raumrichtungen regelmäßigem Abstand (Quadratgitter) auf die als quadratische gewählte Grundfläche der Simulationsbox aufgesetzt.

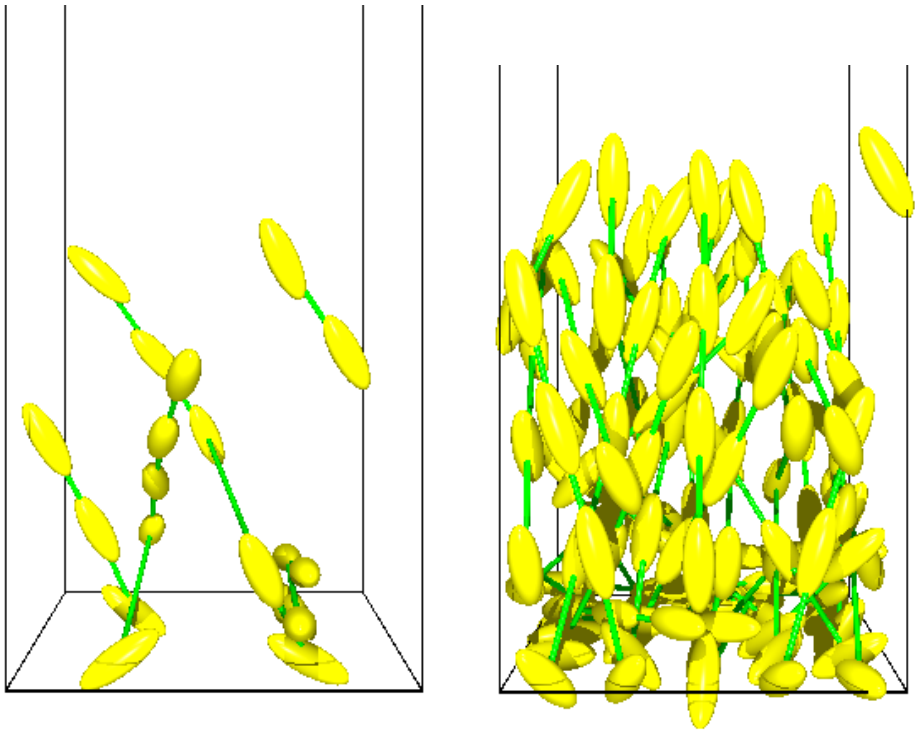

Abbildung 3.3:
Konfigurationsschnappschüsse eines Systems wechselwirkender Ketten für unterschiedliche Pfropfdichten: links $\sigma = 0.028$ – rechts $\sigma = 0.17$

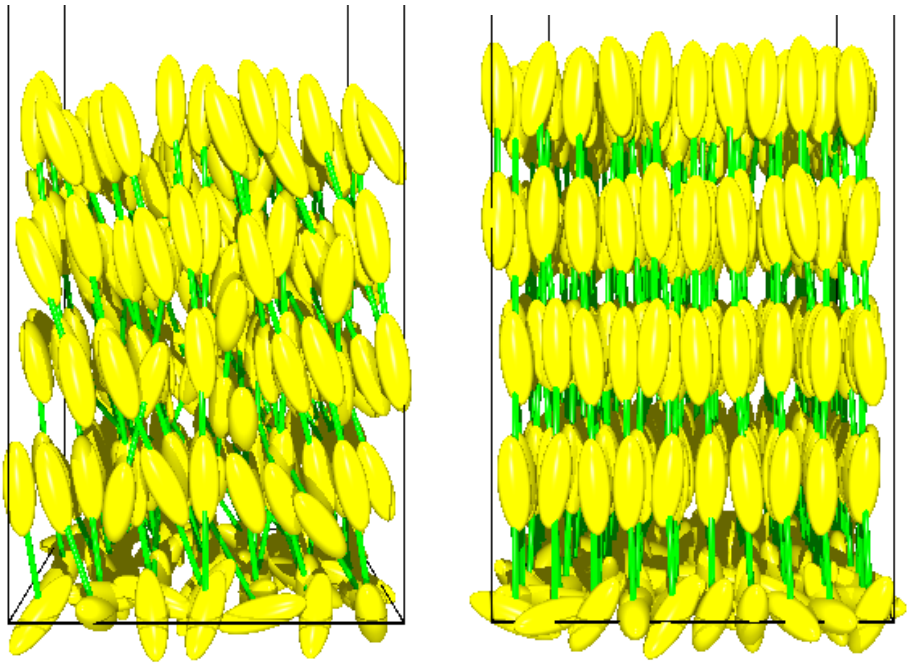

Abbildung 3.4:
Konfigurationsschnappschüsse eines Systems wechselwirkender Ketten für unterschiedliche Pfropfdichten: links σ = 0.34 – rechts σ = 0.69

Damit sich die an gegenüberligenden Wänden verankerten Polymerbürsten nicht gegenseitig beeinflussen und die Simulationsbox nicht zusammenschrumpfen konnte, wurde durch das Abschalten der Volumenmoves ein konstantes, allerdings dafür hinreichend großes, Volumen der Simulationsbox sichergestellt. Somit fanden die Untersuchungen an diesem System im NVT-Ensemble statt.

Die für diese Simulationen verwendete Simulationsbox hatte in der x- bzw. y-Richtung eine Länge von $12\sigma_0$ und in z-Richtung eine Ausdehnung von $40\sigma_0$. Die betrachteten Ketten bestanden aus jeweils fünf Teilchen und wurden bei einer Temperatur von $T = 0.5$ simuliert.

Bei den mit unterschiedlichen Pfropfdichten durchgeführten Simulationsläufen (repräsentative Konfigurationsschnappschüsse siehe Abb. 3.3 und 3.4) konnte unterhalb einer Pfropfdichte von σ = 0.25 Kettenköpfen pro Einheitsfläche ein eher ungeordnetes Verhalten der Ketten beobachtet werden. Oberhalb dieser Dichte präsentierten sich die dann zunehmend senkrecht stehenden Bürsten (siehe auch Abb. 3.5) als recht kompakte Einheit.

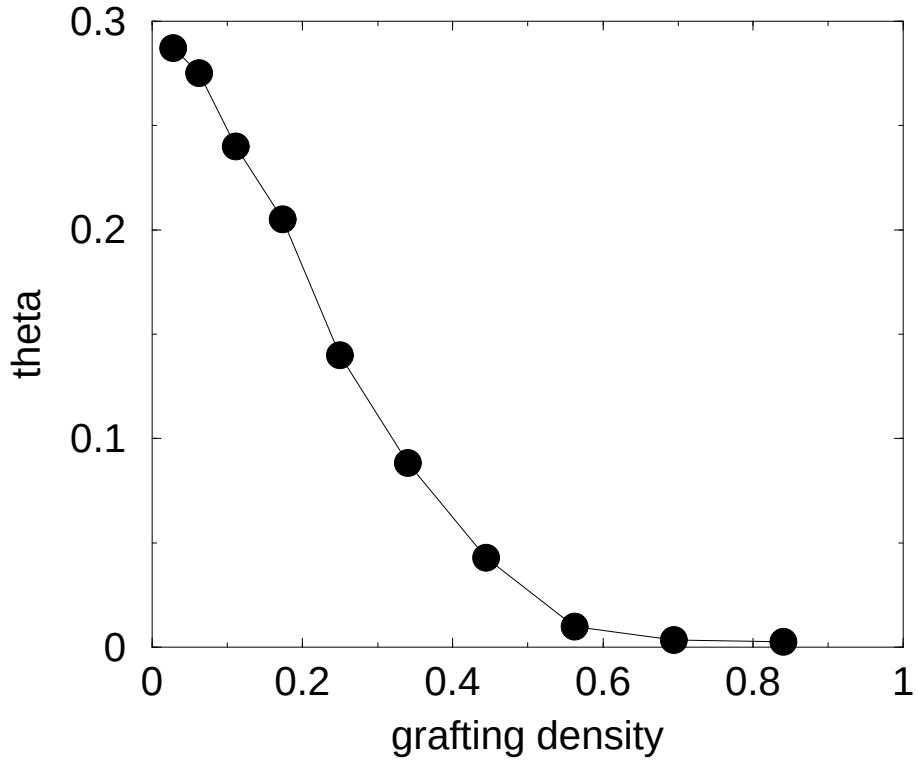

Abbildung 3.5:
Mittlerer Neigungswinkel der Ketten gegen die z-Achse in Abhängigkeit von der Pfropfdichte

Allerdings wurden auch Systeme mit solch hohen Pfropfdichten im Rahmen dieser Arbeit untersucht, da sich das Verhalten der Ketten ohne Lösungsmittel doch recht deutlich vom Verhalten des Systems mit Lösungsmittelteilchen unterscheidet.

3.3 Lösungsmittelteilchen an einer Wand

Für die Simulation eines Systems von Lösungsmittelteilchen zwischen zwei ebenen Wänden, was einer dünnen Schicht entspricht, wurde ebenfalls das in Abschnitt 3.1 eingeführte Modell mit den dort beschriebenen Wechselwirkungen verwendet. Es wurde hier jedoch zwischen zwei in der xy-Ebene befindlichen Wänden betrachtet, d. h. es wurden lediglich in x- und y- Richtung periodische Randbedingungen angenommen, nicht jedoch in z-Richtung.

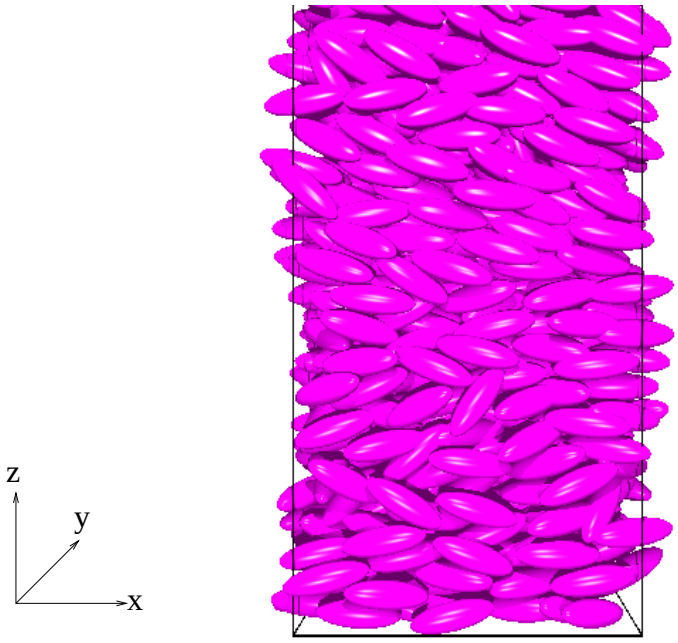

Abbildung 3.6:
Konfigurationsschnappschuß eines Systems von Lösungsmittelteilchen (bei $p = 3$ und $T = 0.5$) an einer Wand (die Wand ist im Bild unten)

Das System wurde bei konstantem Druck weit über die Phasenübergangstemperatur des isotrop-nematischen Phasenüberganges hinaus aufgeheizt, danach wieder abgekühlt und dann bei Temperaturen unterhalb der Übergangstemperatur des Phasenüberganges isotrop-nematisch relaxiert.

Auch hier wurde für die Ausdehnung der Simulationsbox in x- bzw. y-Richtung $12\sigma_0$ gewählt. Für die Drücke $p = 3$ und $p = 4$ wurde das Systme bei der Temperatur $T = 0.5$ simuliert.

Es zeigte sich bereits nach sehr kurzen Simulationsläufen, daß sich die Lösungsmittelteilchen in demjenigen Druck- und Temperaturbereich, der im Bulk eine nematische Phase hervorruft, parallel zu den begrenzenden Wänden, also in der xy-Ebenen, ausrichten, sich also auch in einer nematischen Phase befinden. Diese Ausrichtung ist direkt an der Wand am deutlichsten ausgeprägt und nimmt in z-Richtung zur Mitte der Simulationsbox hin etwas ab (siehe auch Abbildung 3.6).

Allerdings konnten Lösungsmittelteilchen, die an jeweils einer Wand eine einheitliche Ausrichtung besaßen, an der gegenüberliegenden Wand eine andere Orientierung innerhalb der xy-Ebene annehmen. Solche zum Teil auch mehrfach verdrillte Konfigurationen, bei denen die Teilchen an den gegenüberliegenden Wänden in einem Winkel von bis zu 90 Grad zueinander stehen, besitzen, insbesondere bei zueinander senkrechter Orientierung an den Wänden, eine sehr hohe Stabilität gegenüber dem Verschwinden dieser Verdrillung und blieben über mehrere hunderttausend MCS in dieser Konformation.

Um diesem Effekt entgegenzuwirken, wurde zu Beginn der Relaxationsläufe der Systeme mit Wänden (sowohl mit als auch ohne Bürsten) ein zusätzliche Potential angelegt, mit dessen Hilfe die Teilchen entlang der x-Achse ausgerichtet wurden. Somit konnte das Problem der sich an den Wänden in unterschiedliche Richtung ausgerichteten Lösungsmittelteilchen elegant umgangen werden. Für die sich daran anschließenden weiteren Relaxationsläufe und die Meßläufe wurde dieses Potential natürlich abgeschaltet.

Kapitel 4

Ergebnisse

4.1 Systempräparation

Für alle untersuchten Systeme mit auf die Wände gepfropften Polymerketten wurde bei der Simulation wie folgt vorgegangen:

In ein System von Lösungsmittelteilchen, die sich zwischen zwei ebenen Wänden befinden, wurden wechselwirkungsfreie Kettenköpfe auf die Wandflächen aufgebracht. Die Positionen dieser Kettenköpfe wurden so gewählt, daß sie auf jeder der Wände unter Berücksichtigung der periodischen Randbedingungen ein regelmäßiges Quadratgitter bildeten und sich direkt gegenüberstanden. Aus diesen Kettenköpfen wurden dann durch Bildung von Bindungen zu den Lösungsmittelteilchen, also durch das Neuwachsen von Ketten, die Polymerbürsten gebildet.

Als Systemgröße wurde eine Grundfläche von $12\sigma_0$ auf $12\sigma_0$ und eine Lösungsmittelteilchenzahl von 2000 gewählt, was zu einer Ausdehnung der Simulationsbox in z-Richtung (je nach Pfropfdichte) zwischen $43\sigma_0$ bei niedrigen Dichten und $65\sigma_0$ bei hohen Dichten führte. Somit wurde sichergestellt, daß es keinerlei direkte Wechselwirkungen zwischen den an den gegenüberliegenden Oberflächen angebrachten Polymerketten geben kann und zwischen den Polymerketten noch ein ausreichend breiter Bulk-Bereich besteht.

Als Phasenraumpunkt für die Simulation wurde für alle betrachteten Systeme der Punkt $T = 0.5$, $p = 3$ gewählt. Dieser Punkt liegt im nematischen Bereich des Phasendiagrammes, jedoch nicht zu weit von der isotrop-nematischen Phasengrenze entfernt (siehe auch Abb. 3.1), um eine genügend hohe Beweglichkeit der Teilchen zu garantieren. Die Dichte an diesem Phasenraumpunkt beträgt $\rho = 0.313$ und die Energie pro Teilchen ist 0.563.

Nach Ausrichten der Lösungsmittelteilchen entlang der x-Achse durch ein orientierendes Potential (siehe auch Abschnitt 3.3) wurde dieses Potential abgeschaltet und das System equilibiriert. Im Anschluß daran wurde jeweils ein Meßlauf mit mindestens 5 Millionen MCS durchgeführt.

Bei den durchgeführten Läufen wurde die Pfropfdichte der Polymerbürsten zwischen $\sigma = 0.00694$ (entspricht einer Kette auf einer Fläche von $144\sigma_s^2$) und $\sigma = 0.84$ (entspricht $11^2 = 121$ Ketten auf einer Fläche von $144\sigma_s^2$) so variiert, daß immer n^2 Ketten auf einer Fläche von $144\sigma_s^2$ angeordnet waren. Als Kettenlänge der Polymere wurde $n = 5$ gewählt.

Der Zwischenbereich, der 4.2^2 bis 4.5^2 Ketten pro Wandfläche entspricht, wurde durch Variation der Grundfläche bei $5^2 = 25$ Ketten realisiert.

Im folgenden sind Pfropfdichten immer in Einheiten von $1/\sigma_s^2$ angegeben.

4.2 Simulationsergebnisse

Neben dem Einfluß, den die Ketten auf das sie umgebende Lösungsmittel haben, ist es durchaus von Interesse, auch den umgekehrten Einfluß, also den Einfluß des Bulks auf die Bürste, zu untersuchen.

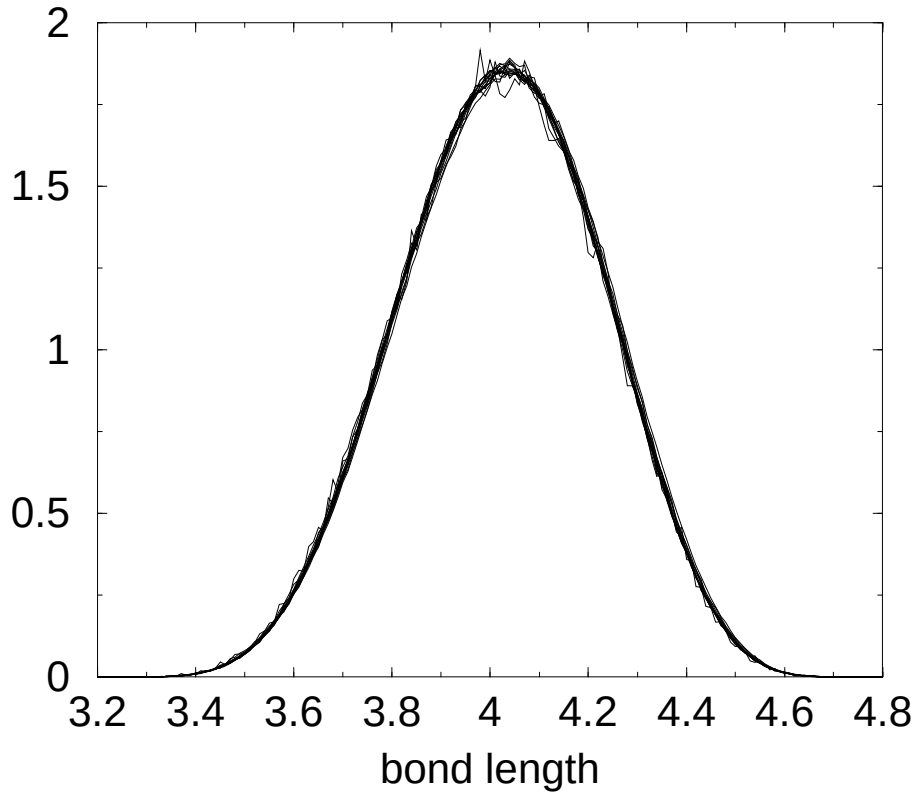

Abbildung 4.1:
Normierte Verteilung der Bondlängen für unterschiedliche Pfropfdichten zwischen $\sigma = 0.00694$ und $\sigma = 0.84$ sowie theoretische Kurve

Zunächst kann man das Histogramm der Verteilung der Längen der Bindungen zwischen zwei, entlang der Kette benachbarten, Kettenteilchen (siehe Abb. 4.1) sowie die Verteilung der Winkel zwischen den Teilchenachsen der Kettenteilchen und den von ihnen ausgehenden Bindungen (siehe Abb. 4.3) betrachten. Weiterhin ist das Histogramm der Verteilung der Winkel zwischen zwei vom selben Teilchen ausgehenden Bindungen (siehe Abb. 4.2) eine aufschlußreiche Größe.

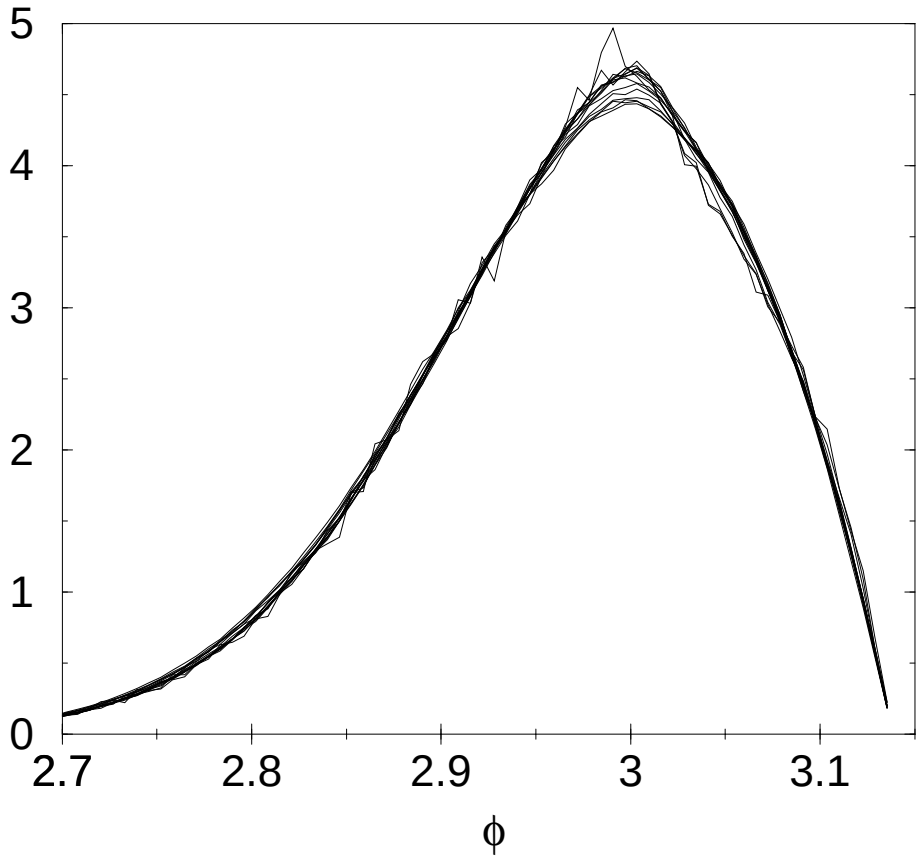

Abbildung 4.2:

Normierte Verteilung der Winkel ϕ zwischen zwei benachbarten Bindungen für alle betrachteten Pfropfdichten sowie theoretische Kurve

Aus den Abbildungen 4.1 bis 4.3 kann man ersehen, daß die entsprechenden Kurven für alle drei betrachteten Größen für unterschiedliche Pfropfdichten nicht wesentlich voneinander abweichen (lediglich in Abbildung 4.2 nimmt die Höhe des Maximums mit steigender Pfropfdichte minimal ab) und sich nahezu völlig überlappen. Auch die theoretischen Kurven lassen sich nicht von den gemessenen Kurven unterscheiden.

Die erkennbaren Fluktuationen lassen sich aus der unterschiedlichen Anzahl von betrachteten Ketten heraus verstehen, da die geringere Anzahl der Meßwerte bei niedrigen Pfropfdichten, bedingt durch die geringere Anzahl der Ketten (und somit auch der Bonds und Winkel), natürlich zu einer schlechteren Statistik führen muß. Dies ändert jedoch nichts an der grundsätzlichen Gleichheit der Meßergebnisse.

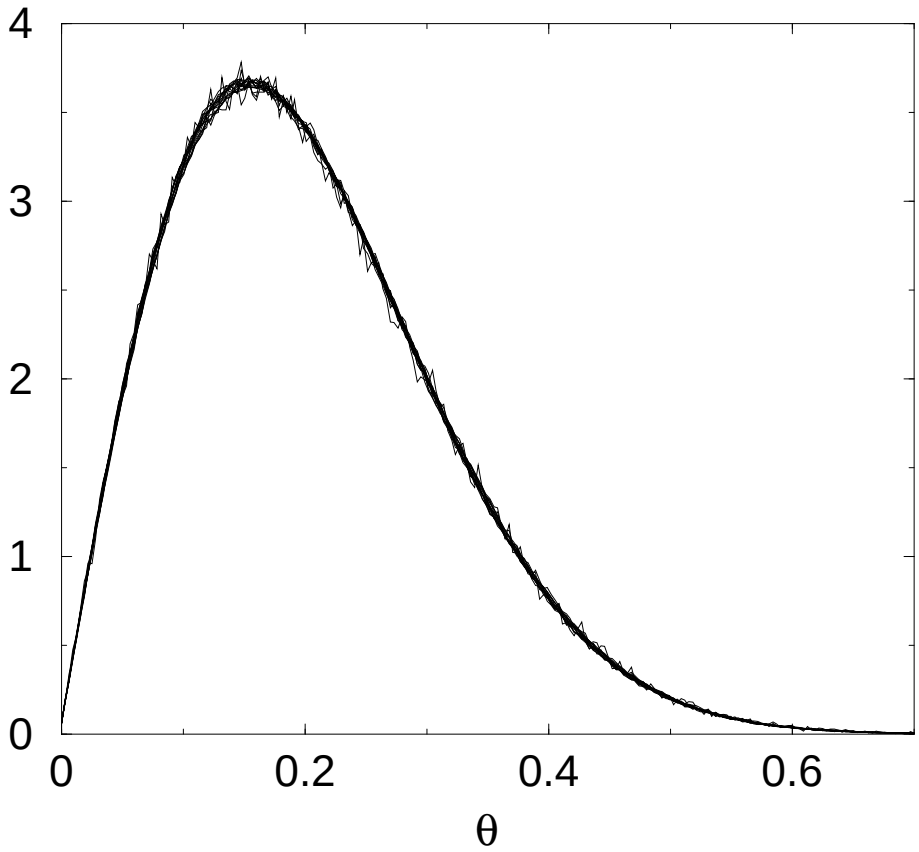

Abbildung 4.3:
Normierte Verteilung der Winkel θ zwischen Teilchenachse und Bindungen zu den nächsten Kettennachbarn für alle betrachteten Pfropfdichten

Daraus kann man erkennen, daß sich das intrinsische Verhalten der Bürste, also das Verhalten der Bondlängen und Bondwinkel, durch die Wahl einer anderen Pfropfdichte nicht verändern, sondern unabhängig davon immer das gleiche Verhalten aufweisen.

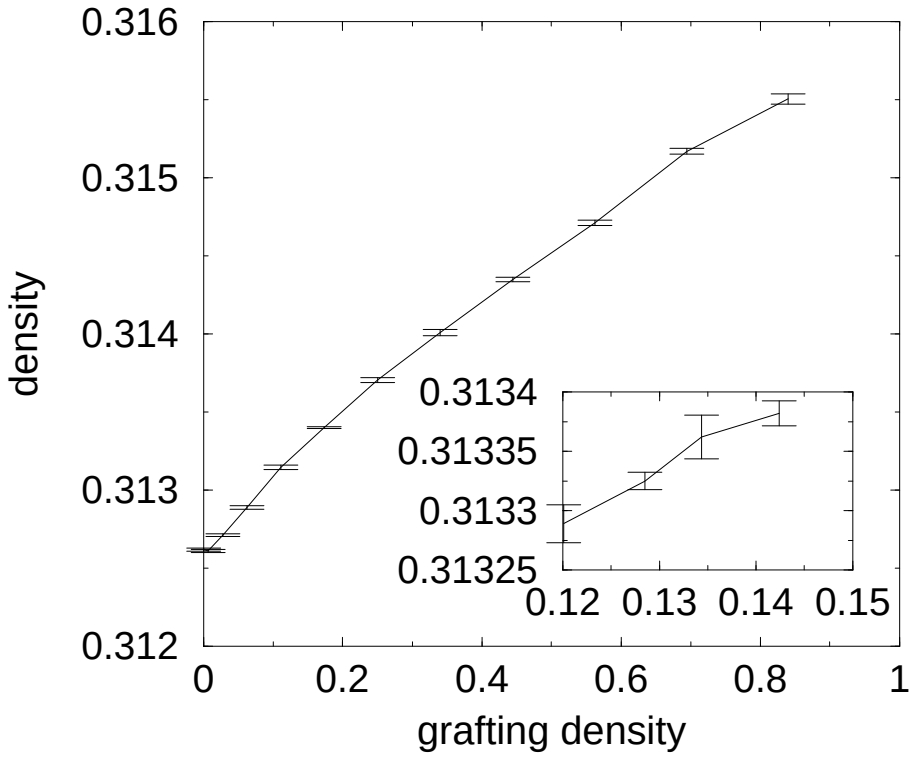

Abbildung 4.4: Mittlere Teilchendichte in Abhängigkeit von der Pfropfdichte

Die mittlere Teilchendichte $\bar{\rho}$ ist über den gesamten Bereich der betrachteten Pfropfdichten nahezu konstant, steigt jedoch zu höheren Pfropfdichten hin leicht an (siehe Abb. 4.4). Da die Teilchendichte bedingt durch die Wände bzw. Wandwechselwirkungen ein nichttriviales Verhalten zeigt (siehe auch Abb. 4.16 auf Seite 53) und an den Wänden Exzessdichten auftreten, kann man die mittlere Dichte als

$$\bar{\rho} = \rho_{Bulk} + \frac{2\,\Delta\rho(\sigma)}{L_z} \tag{4.1}$$

schreiben. Dabei ist ρ_{Bulk} die Teilchendichte des reinen Bulk-Systems, die auch im inneren des Systemes mit Wänden auftritt und $\Delta\rho(\sigma)$ die Exzessdichte.

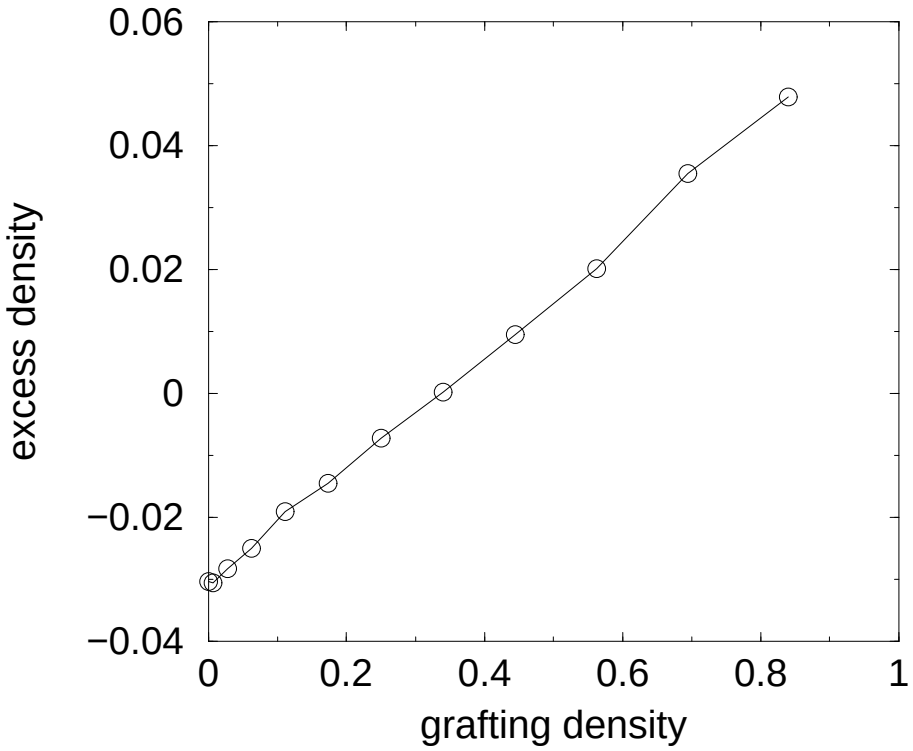

Abbildung 4.5: Exzessdichte $\bar{\rho}$ in Abhängigkeit von der Pfropfdichte

Wenn man Gleichung (4.1) nach $\Delta\rho(\sigma)$ auflöst und dann die Exzessdichte gegen die Pfropfdichte aufträgt, so erhält man Abbildung 4.5. Diese zeigt einen linearen Anstieg der Exzessdichte mit der Pfropfdichte, der für die steigende mittlere Teilchendichte bei steigender Pfropfdichte verantwortlich ist.

Die innere Energie pro Teilchen, die im reinen Bulk-System den Wert 0.563 besitzt, steigt von diesem Wert, den sie auch im System ohne Bürste hat, mit wachsender Pfropfdichte an (siehe Abb. 4.6), obwohl der Anteil der Energie, der von der Wechselwirkung gemäß dem Gay-Berne-Potential stammt, mit wachsender Pfropfdichte leicht abnimmt. Diese Abnahme wird durch den Energiebeitrag der Bondwinkel- und Bondlängenpotentiale aber überkompensiert, wodurch das bereits angesprochene Bild entsteht.

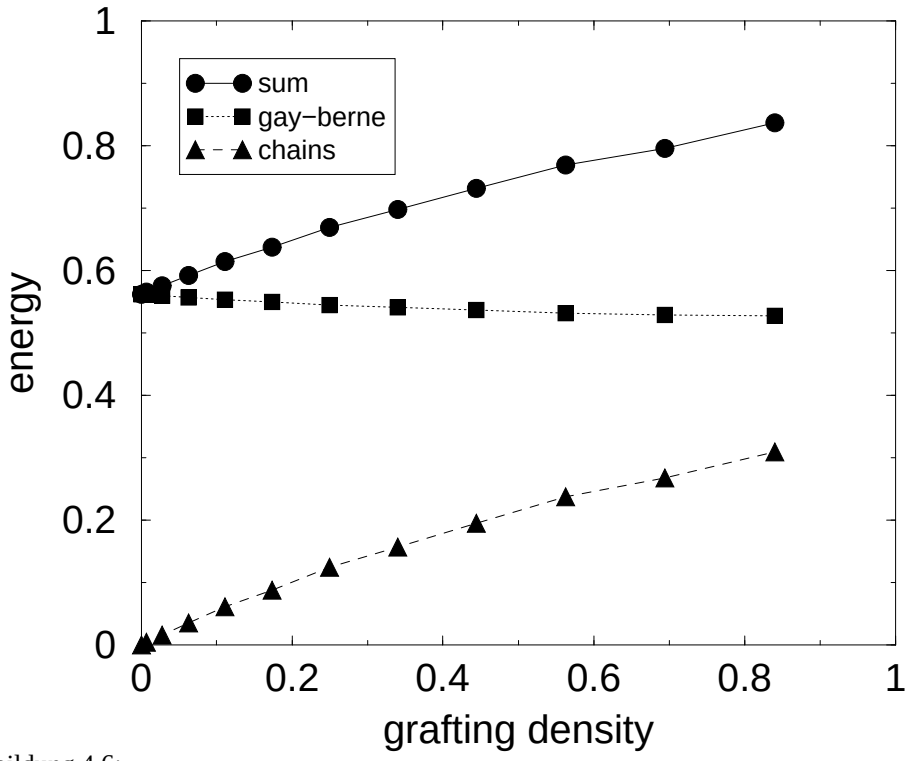

Abbildung 4.6:

Innere Energie pro Teilchen in Abhängigkeit von der Pfropfdichte bei konstanter Grundfläche

Der Verlauf der Energiekurve ist glatt – es treten keinerlei Unstetigkeiten auf, und die Steigung verändert sich nicht. Somit weist nichts an der Energiekurve auf einen Phasenübergang hin.

Bleibt die Anzahl der gepfropften Ketten konstant, während die Größe der Grundfläche verändert wird, um eine Variation der Pfropfdichte zu erreichen, bleibt die innere Energie pro Teilchen innerhalb der Fehlergrenzen konstant (siehe Abb. 4.7). Dies ist, da ja das Ansteigen der Energie durch die höhere Anzahl der Ketten bedingt wird, nicht überraschend.

Der nematische Ordnungsparameter im Lösungsmittel ist bis auf zwei „Einbrüche“ bei $\sigma = 0.00694$ und $\sigma = 0.0625$, zu denen später noch mehr zu sagen sein wird, über den gesamten betrachteten Bereich der Pfropfdichten konstant und liegt im Bereich von 0.75, einem Wert der etwas unter dem im Bulksystem gemessenen Ordnungsparameter von 0.79 liegt. Weiterhin fällt der Ordnungsparameter auch bei der höchsten betrachteten Pfropfdichte deutlich ab.

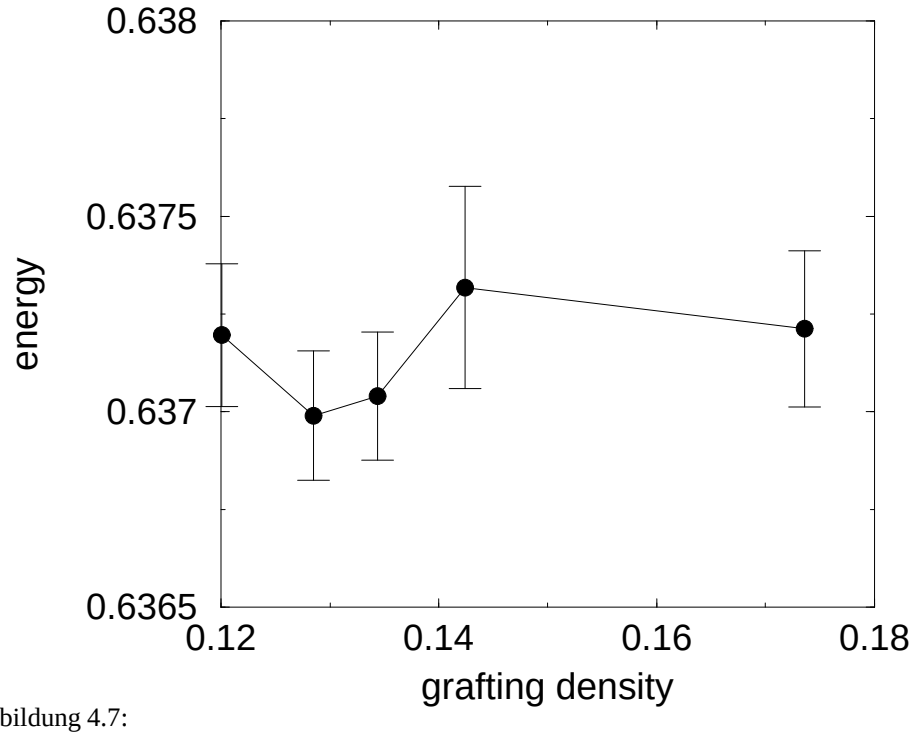

Abbildung 4.7:

Innere Energie pro Teilchen in Abhängigkeit von der Pfropfdichte bei konstanter Kettenanzahl

Der Einbruch am rechten Rand ist darauf zurückzuführen, daß die Bürste bei hoher Pfropfdichte einen größeren Anteil der simulierten Teilchen stellt. Somit ist der Einfluß der Bürste auf den Ordnungsparameter stärker als bei niedrigen Pfropfdichten. Weiterhin zeigt sich, daß bei der höchsten betrachteten Pfropfdichte die Neigungsrichtung von Bulk und Bürste am stärksten auseinander streben (siehe Abb. 4.10). Daher bricht auch der über das gesamte System gemittelte Ordnungsparameter ein.

Eine wichtige Größe für die Beschreibung des Systems ist der Winkel der Teilchenachsen gegen die z-Achse. Wenn man für unterschiedliche Pfropfdichten das normierte Histogramm der Verteilung jener Winkel gegen die z-Achse aufträgt, erhält man Abbildung 4.9.

Im linken oberen Bild von Abbildung 4.9, in dem die Ergebnisse für geringe Pfropfdichten aufgetragen sind, erkennt man einen Peak bei $\Theta = \pi/2$, der für alle vier Kurven nahezu gleich aussieht. Dies entspricht Teilchen, die sich parallel zu den Wänden ausrichten.

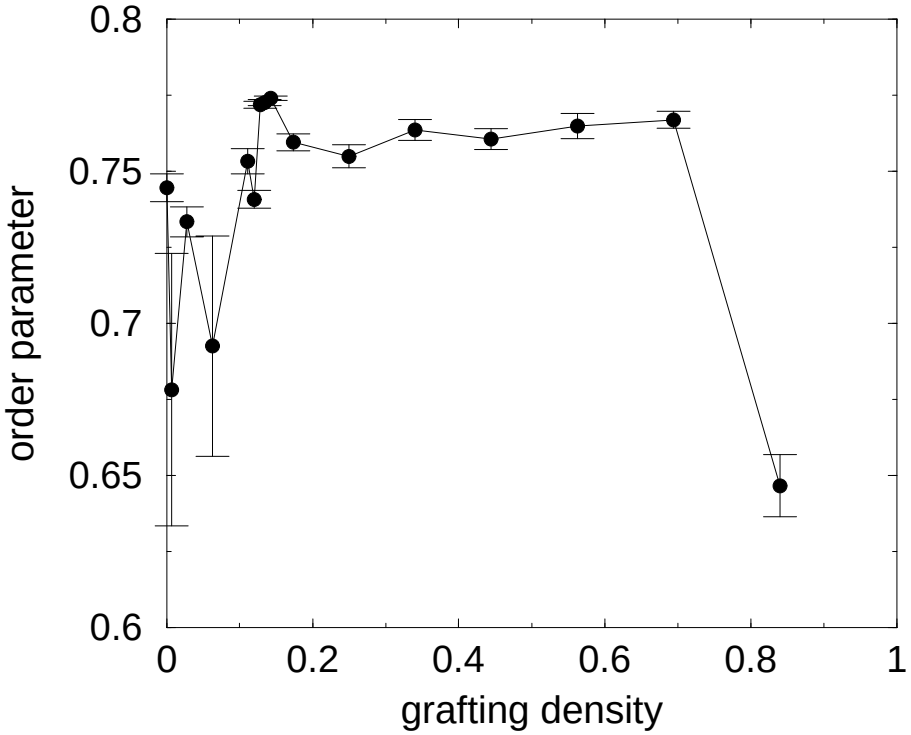

Abbildung 4.8: Nematischer Ordungsparameter in Abhängigkeit von der Pfropfdichte

Erhöht man nun die Pfropfdichte weiter, so erhält man ein verändertes Verhalten. Zwischen einer Dichte von $\sigma = 0.111$ und $\sigma = 0.174$ wandert der Peak, der auch breiter wird, von $\Theta = \pi/2$ zu kleineren Winkeln. Die Teilchen beginnen folglich in diesem Bereich einen Neigungswinkel gegen die Wand einzunehmen. Dieses Verhalten kann man im linken unteren Bild sehen.

Bei noch höheren Pfropfdichten richten sich die Teilchen immer weiter auf und der Peak wandert weiter zu kleineren Winkeln, um schließlich, wie im rechten oberen Bild zu sehen ist, bei einer Pfropfdichte von $\sigma = 0.84$ das Maximum bei $\Theta = 0$ zu erreichen. Die Mehrzahl der Teilchen steht nun senkrecht zur Wand.

Betrachtet man den Bereich der Pfropfdichte, in dem sich das Maximum der Winkelverteilung von $\Theta = \pi/2$ löst, so fällt auf, daß sich dieses Ablösen sprunghaft vollzieht und zur Bildung von einer Art Plateau bei großen Winkeln führt. Im rechten unteren Bild, in dem die Kurven für diesen Zwischenbereich aufgetragen sind, kann man das gut erkennen.

Betrachtet man die Kurven aus Abbildung 4.9 getrennt für die Lösungsmittelteilchen und die Kettenteilchen und normiert sie separat, so ergibt sich daraus Abbildung 4.10.

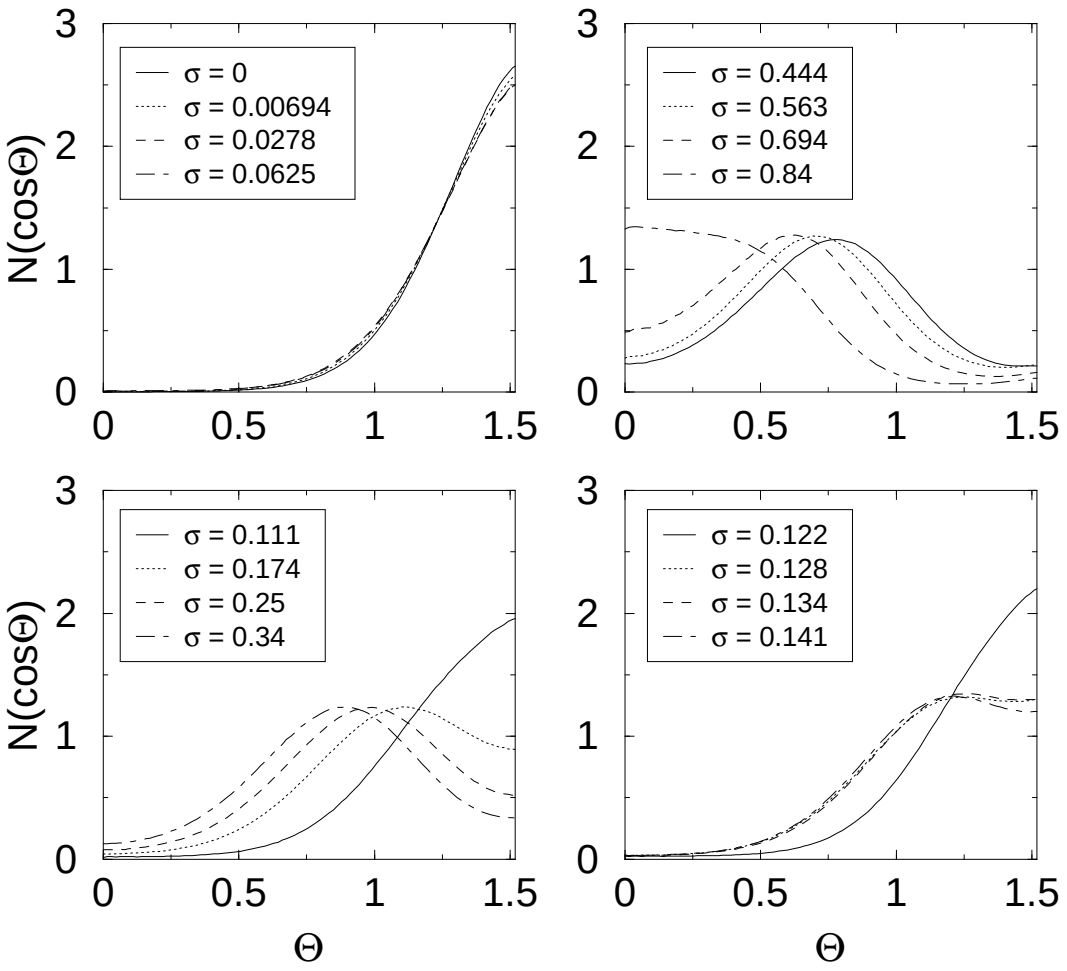

Abbildung 4.9:

Normierte Verteilung des Winkels Θ zwischen Teilchenachse und z-Achse für unterschiedliche Pfropfdichten σ

Auch hier lassen sich klar in beiden Abbildungen Ähnlichkeiten hinsichtlich Verhalten und Phasenübergang erkennen, allerdings gibt es auch deutliche Unterschiede: Zum einen gibt es bei den Kettenteilchen (rechts) einen Untergrund über den gesamten Winkelbereich, der von den Kettenköpfen, denen keine Winkelbeschränkung auferlegt worden ist, herrührt, und zum anderen gibt es im Gegensatz zu den Lösungsmittelteilchen (links) für alle betrachteten Pfropfdichten einen nichtverschwindenden Anteil von parallel zur Grundfläche ausgerichteten Kettenteilchen.

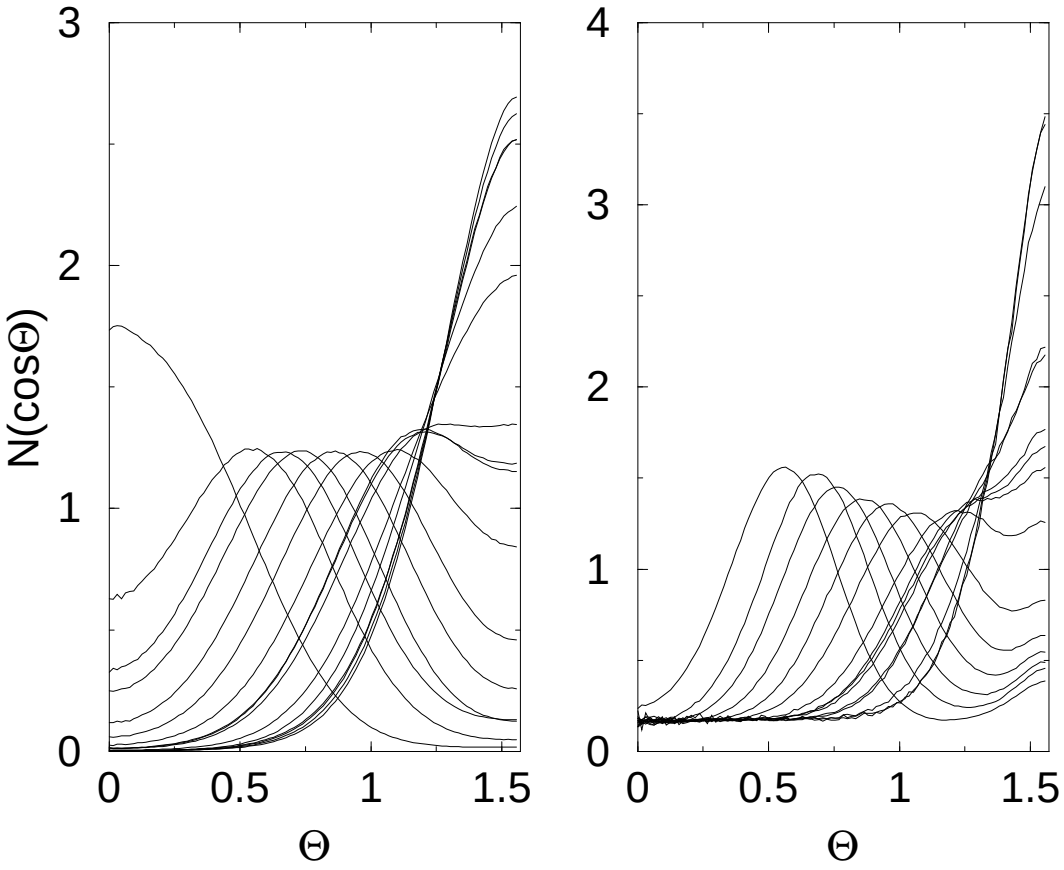

Abbildung 4.10:

Normierte Verteilung des Winkels Θ zwischen Teilchenachse und z-Achse für Lösungsmittelteilchen (links) und Kettenteilchen (rechts) für die Pfropfdichten (von rechts nach links) $\sigma = 0.00694, 0.0278, 0.0625, 0.111, 0.12, 0.128, 0.134, 0.142, 0.174, 0.25, 0.34, 0.444, 0.5625, 0.694, 0.84$

Bei der höchsten betrachteten Pfropfdichte $\sigma = 0.84$ ist bei den Kettenteilchen (rechts) keine senkrechte Ausrichtung wie bei den Lösungsmittelteilchen (links) zu erkennen. Das erklärt das in Abbildung 4.8 gefundenen Verhalten des abfallenden Ordnungsparameters bei dieser Pfropfdichte.

Allen betrachteten Kurven ist eine Verschiebung des Maximalwerts der Winkelverteilung bei den Kettenteilchen gegenüber der Verteilung der Lösungsmittelteilchen zu größeren Winkeln gemein, was sich mit wachsender Pfropfdichte immer deutlicher zeigt.

Man könnte diese Diskrepanz zwischen Bürste und Bulk eventuell durch um die z-Achse rotierende Bürsten erklären, die die Lösungsmittelteilchen somit dazu bewegen, sich senkrecht zu orientieren. Da jedoch wie in Abbildung 4.11 gezeigt (die drei aufgetragenen Kurven sind exemplarisch für alle anderen Kurven gewählt, deren Verhalten ähnlich ist) die Verteilung der Winkel zwischen Teilchenachse und x-Achse keine Gleichverteilung zeigt, sondern deutlich um Null (eine bevorzugte Ausrichtung entlang der x-Achse) ein Maximum hat, kann das nicht die Erklärung des beobachteten Effektes sein.

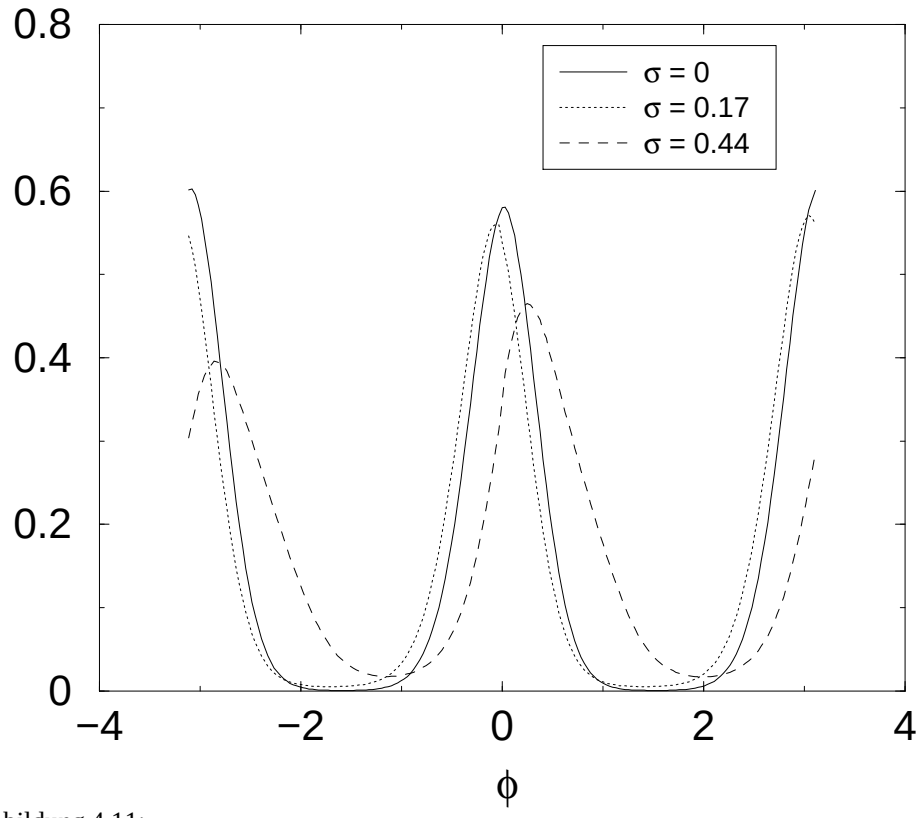

Abbildung 4.11:

Normierte Verteilung des Winkels Φ zwischen Teilchenachse und x-Achse für unterschiedliche Pfropfdichten

Vielmehr ist es eher so, daß sich die Lösungsmittelteilchen an die durch die, gerade bei hohen Pfropfdichten, dicht gepackte Bürste gebildeten Oberfläche besonders günstig senkrecht anlagern können. Somit verankert das Lösungsmittel nicht am Substrat, sondern vielmehr an der durch die Bürste gebildete Oberfläche.

Packungseffekte sind auch der Grund für die selbst bei den höchsten betrachteten Pfropfdichten nicht senkrecht stehende Bürste, da durch das Einhaken eines Kettenteilchens in die Lücke zwischen zwei Teilchen einer benachbarten Kette dieses energetisch besonders günstig liegt. Bei noch höheren Packungsdichten ($\sigma > 1$) kommt es dann auch vor, daß hier die Bürste senkrecht steht.

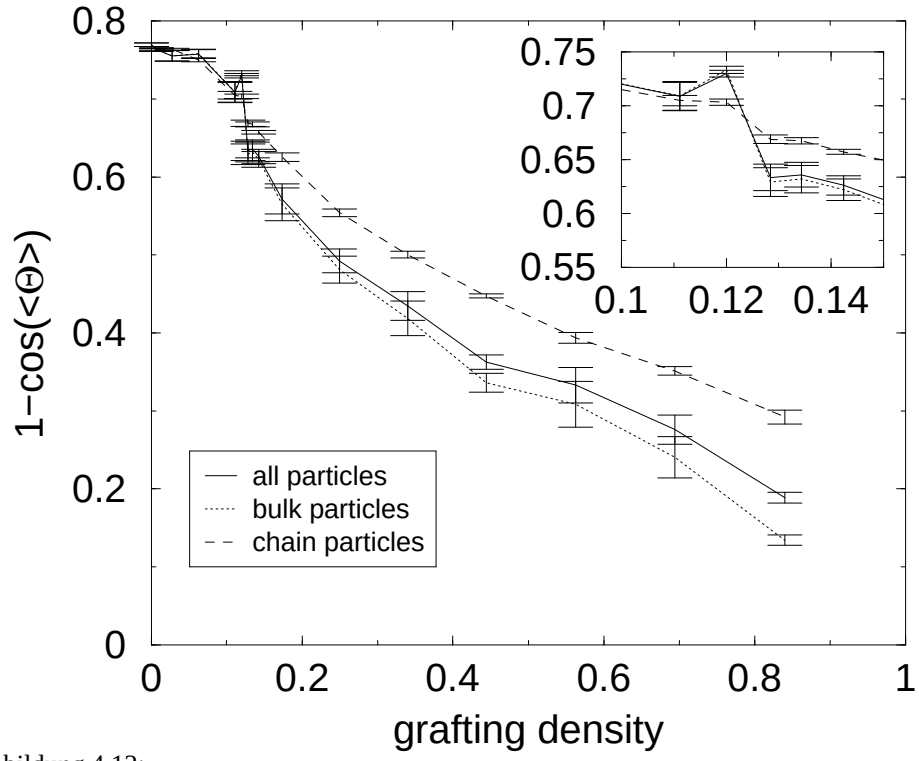

Abbildung 4.12:

Mittlerer Neigungswinkel der Lösungsmittelteilchen in der Auftragung $1 - \cos(< \Theta >)$ gegen die Pfropfdichte σ

Anstelle der Verteilung des Winkels der Teilchenachse gegenüber der z-Achse kann man auch das Verhalten des Mittelwertes der Teilchenachse, sowohl für das gesamte System als auch unterteilt in Lösungsmittelteilchen und Kettenteilchen, in Abhängigkeit von der Pfropfdichte untersuchen (Abb. 4.12). Dabei kann man in der Gesamtkurve und in der Kurve für die Lösungsmittelteilchen einen signifikanten Sprung erkennen. Weiter weisen die Kurven vor bzw. nach dem Sprung unterschiedliche Steigungen auf, was, ebenso wie der Sprung selbst, einen Phasenübergang zwischen einer Phase mit liegenden und einer Phase mit schräg stehenden Lösungsmittelteilchen anzeigt.

Die Kurve für die Kettenteilchen hingegen verläuft offensichtlich glatter und zeigt nur einen kleineren Sprung und eine über den gesamten Bereich nahezu konstante Steigung.

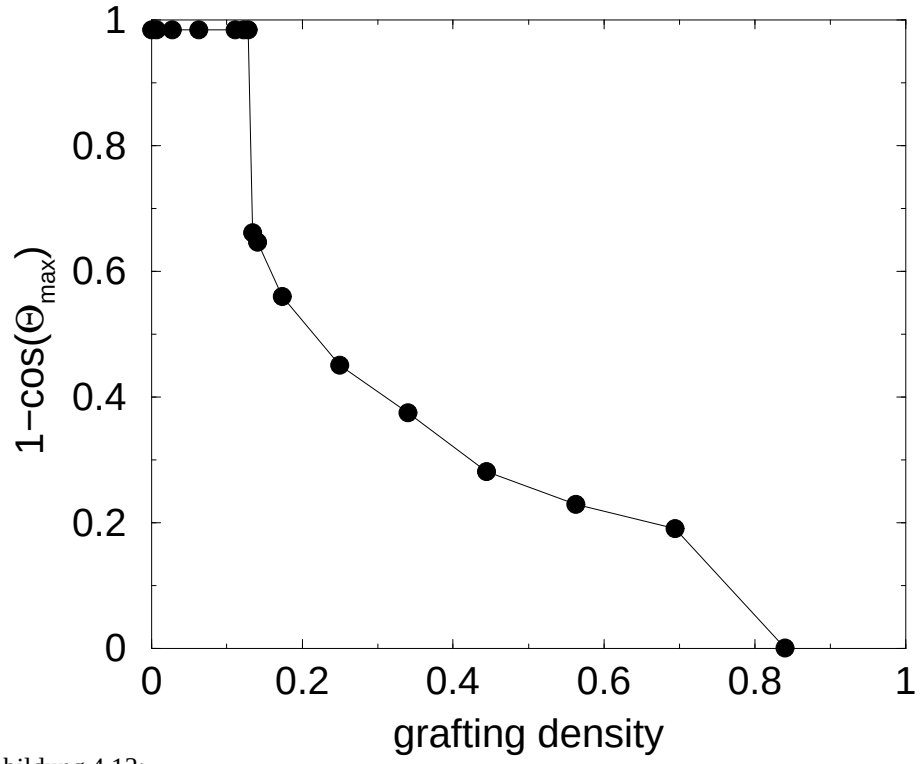

Abbildung 4.13:

Position des Maximums der Verteilung der Neigungswinkel Θ gegen die z-Achse in der Auftragung $1 - \cos(\Theta)$ gegen die Pfropfdichte σ

Das Auseinanderlaufen der bei niedrigen Pfropfdichten nahezu identischen Kurven für das gesamte System und die Lösungsmittelteilchen ist mit dem größeren relativen Anteil der Kettenteilchen bei hohen Pfropfdichten zu erklären.

Noch klarer als in Abbildung 4.12 ist ein Sprung festzustellen, wenn man an Stelle des Mittelwertes des Neigungswinkels gegen die z-Achse die Position des Maximums der Verteilung der Neigungswinkel gegen die Pfropfdichte aufträgt (Abb. 4.13). Die Position des Maximums springt bei einer Pfropfdichte von $\sigma = 0.13$ aus einer flach liegenden Position, also mit einem Winkel von $\pi/2$ gegen die z-Achse, in eine Position mit geringerem Winkel.

Dieser Winkel steigt mit zunehmender Pfropfdichte an und erreicht bei einer Pfropfdichte von $\sigma = 0.84$ die x-Achse, nimmt also den Wert Null an, was einer Ausrichtung parallel zur z-Achse entspricht.

Ob sich der Übergang zu paralleler Ausrichtung allerdings sprunghaft vollzieht oder nicht, kann aus der Abbildung nicht klar bestimmt werden. Auch wenn es so aussehen mag, fehlen doch eine ganze Reihe von Zwischenwerten, ohne die eine solche Aussage keine hinreichende Qualität besäße.

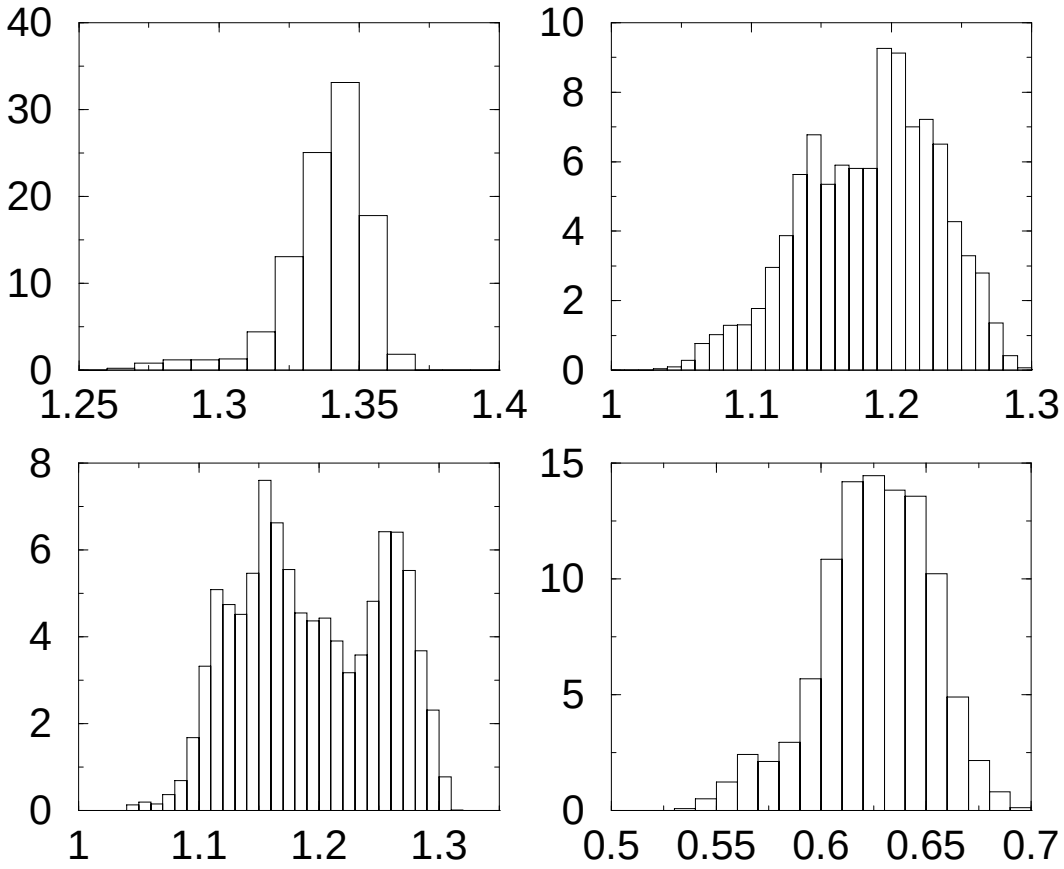

Abbildung 4.14:
Normierte Histogramme der Verteilung des mittleren Winkels zwischen Teilchenachse und der z-Achse für die Pfropfdichten $\sigma = 0$ (links oben), $\sigma = 0.122$ (links unten), $\sigma = 0.134$ (rechts oben) und $\sigma = 0.84$ (rechts unten)

Daher ist es sinnvoll, im weiteren primär den Phasenübergang zur Phase mit schräg liegenden Lösungsmittelteilchen zu betrachten.

Ein Indiz für einen Phasenübergang erster Ordnung ist häufiges Hin- und Herspringen einer Meßgröße zwischen zwei Werten, was sich dann im Histogramm der betrachteten Größe als Verteilung mit zwei Spitzen zeigt, einer sogenannten Doppelpeakstruktur. Stellt man bei einem Vergleich der Flächen unter beiden Peaks fest, daß beide gleich groß sind, dann hat man eine Abschätzung für den Phasenübergangspunkt gefunden.

Um den Punkt, an dem der Phasenübergang zwischen einer Konfiguration mit liegenden Teilchen und Konfigurationen mit Teilchen, die einen Winkel gegen die Grundfläche aufweisen, besser bestimmen zu können, ist es daher hilfreich, sich das Histogramm des mittleren Neigungswinkel der Teilchen gegen die z-Achse anzusehen.

In Abbildung 4.14 ist dies exemplarisch für vier Fälle dargestellt. Links oben und rechts unten sind die Verteilungen für die Pfropfdichten $\sigma = 0$ (ein System ohne Ketten) und $\sigma = 0.84$ aufgetragen. In beiden Teilbildern ist deutlich ein einziger Peak zu erkennen, dessen Maximum bei dem System ohne Ketten einen deutlich höheren Wert, also einen größeren Winkel, hat als im System mit hoher Pfropfdichte.

Die beiden anderen Teilbilder von Abbildung 4.14 zeigen die Verteilung bei Propfdichten von $\sigma = 0.122$ (links unten) bzw. $\sigma = 0.134$ (rechts oben). Bei beiden Dichten ist eine mehr oder minder deutlich ausgeprägte Doppelpeakstruktur in der Verteilung der mittleren Neigungswinkel zu erkennen.

Da die beiden Peaks sich in beiden Teilbildern stark überlappen, ist es schwierig, die Fläche unter den Peaks exakt zu bestimmen, zumal die Statistik für die weiteren Auswertungen dieser Peaks viel zu gering ist, um beispielsweise ein Histogramm-reweighting durchzuführen.

Trotz der schlechten Statistik läßt sich aus diesen beiden Bildern ablesen, daß sich die Pfropfdichte des Phasenübergangs zwischen den beiden hier betrachteten Dichten befinden muß, da bei der geringeren Dichte der linke Peak die größere Fläche und bei der höheren Dichte der rechte Peak die größere Fläche besitzt, und somit die Pfropfdichte, bei der beide Peaks die gleiche Fläche besitzen, dazwischen liegen muß.

Der mittlere Neigungswinkel der Kopf-Ende-Vektoren der Polymerketten gegen die z-Achse in der Auftragung $1 - cos(< \Theta >)$ gegen die Pfropfdichte σ (siehe Abb. 4.15) zeigt ein mit ansteigender Pfropfdichte fallendes Verhalten, bei dem die zunächst lineare negative Steigung mit zunehmender Pfropfdichte deutlich abnimmt, um am Ende einen Plateauwert von ≈ 0.05, was einem Winkel von $\Theta = 0.32$ entspricht, zu erreichen.

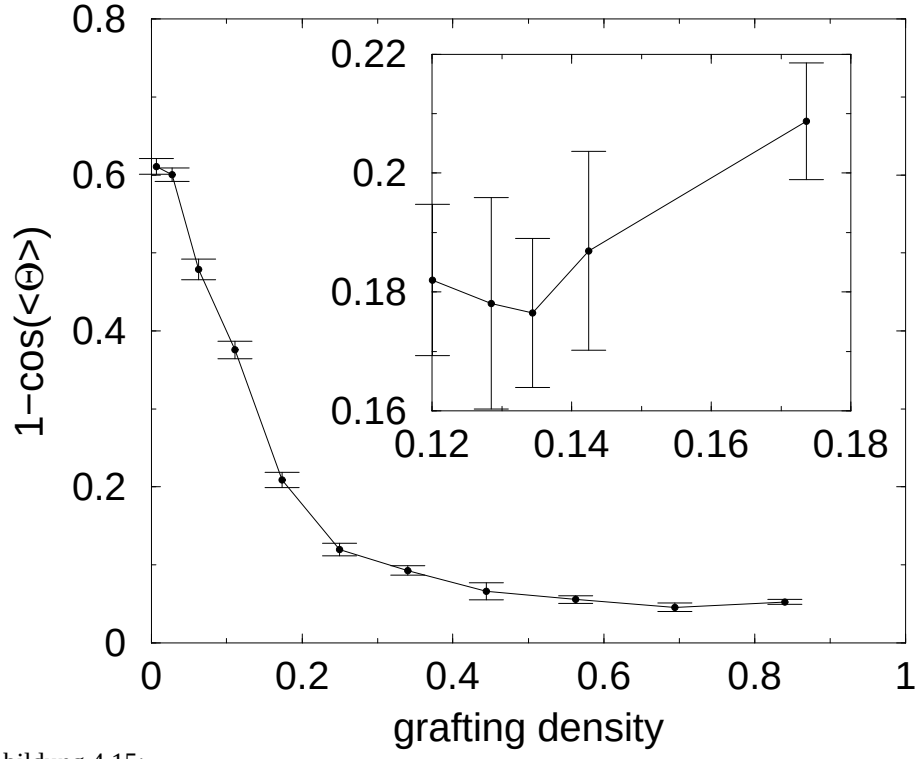

Abbildung 4.15:

Mittlerer Neigungswinkel der Kopf-Ende-Vektoren der Polymerketten gegen die z-Achse in der Auftragung $1 - cos(< \Theta >)$ für unterschiedliche Pfropfdichten

Daraus läßt sich ablesen, daß die Bürste auch bei den höchsten betrachteten Pfropfdichten im Mittel nicht senkrecht auf der Grundfläche steht, sondern einen kleinen Winkel gegen die z-Achse hat. Der Grund hierfür liegt nicht wie man vermuten könnte in der vor der Messung durchgefürten Orientierung der Lösungsmittelteilchen entlang der x-Achse (auch senkrecht aufgesetze Ketten neigten sich) sondern in Einrasteffekten der Ketten ineinander, die es der Bürste erlauben einen energetisch günstigeren Zustand einzunehmen.

Im Falle konstanter Kettenzahl (kleines Bild in Abb. 4.15) fällt der mittlere Winkel nicht mit steigender Pfropfdichte ab, sondern bleibt nahezu konstant.

Abbildung 4.16 zeigt die Gesamtdichteprofile für verschiedene Pfropfdichten. Die Kurven sind zur besseren Unterscheidbarkeit um jeweils 0.2 nach oben verschoben. Am rechten Rand nähern sich die Kurven der Bulk-Dichte von $\rho = 0.313$ an. Weiterhin sind beide Wände in der Simulatiosnbox gleich, so daß es ausreichend ist, sich das Profil an einer Wand anzusehen.

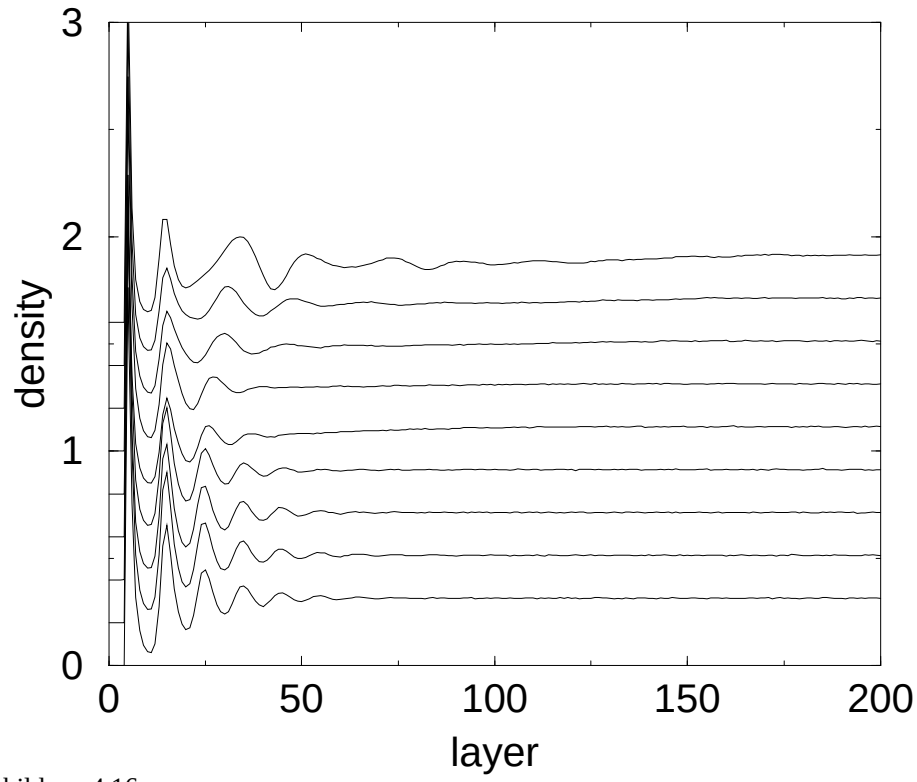

Abbildung 4.16:

Gesamtdichteprofil in der Nähe einer Wand für unterschiedliche Pfropfdichten. Die verschiedenen Kurven entsprechen in der Reihenfolge von oben nach unten den Pfropfdichten $\sigma = 0.84$, 0.563, 0.444, 0.25, 0.174, 0.111, 0.0625, 0.0278, 0 und sind jeweils um 0.2 gegeneinander verschoben.

Die Reichweite der von der Wand hervorgerufene Oszillationen in der Gesamtdichte geht mit steigender Pfropfdichte zunächst etwas zurück, um dann bei hohen Pfropfdichten wieder anzusteigen. Außerdem verschiebt sich bei den höheren Pfropfdichten auch die Frequenz dieser Oszillationen. Um dies näher untersuchen zu können ist es hilfreich die Dichten des Lösungsmittels und der Ketten separat zu betrachten.

Wenn man die Dichteverteilung der Lösungmittelteilchen nahe einer der Wände betrachtet (man kann sich auch hier auf eine der beiden Wände beschränken, da die Verteilung symmetrisch ist), kann man folgendes erkennen:

Bei niedrigen Pfropfdichten bildet sich eine der Dichteverteilung eines Systems ohne Bürsten ähnliche Struktur, also eine Abfolge von um den Mittelwert der Bulkdichte oszillierender, in der Höhe abfallender Peaks mit Abstand σ_s, von denen bei der niedrigsten betrachteten Pfropfdichte $\sigma = 0.00694$ sechs deutlich erkennbar sind, ehe die Kurve auf den Mittelwert der Lösungsmittelteilchendichte abgefallen ist.

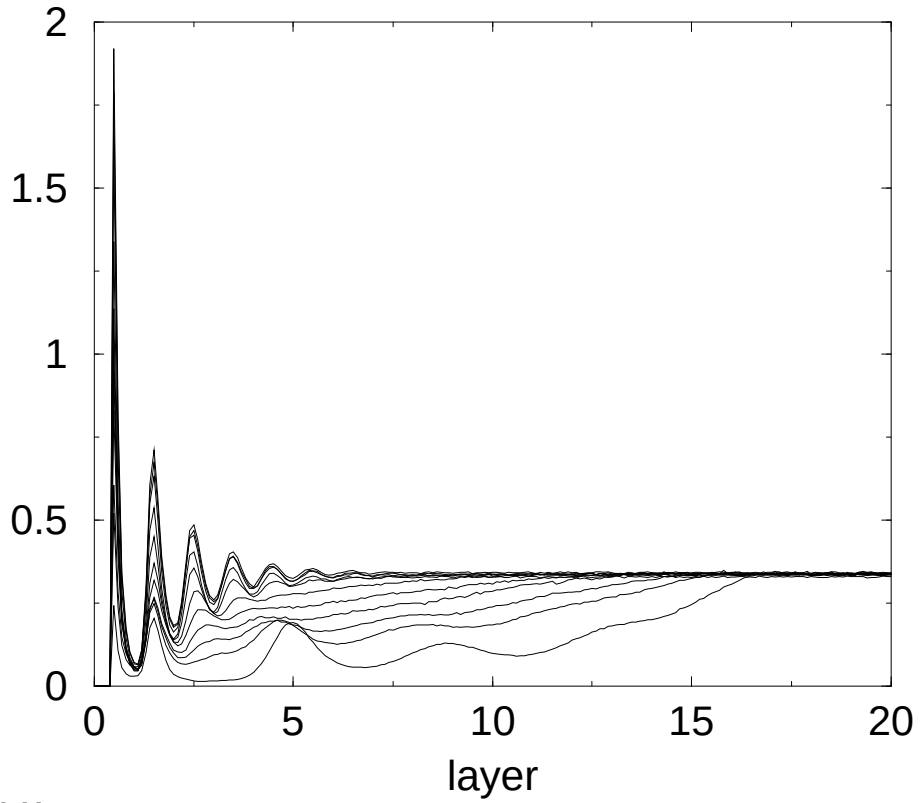

Abbildung 4.17:

Dichteprofil der Lösungsmittelteilchen in der Nähe einer Wand für unterschiedliche Pfropfdichten. Die verschiedenen Kurven entsprechen in der Reihenfolge ihres Abweichens von der Teilchendichte im Lösungsmittel von rechts nach links den Pfropfdichten $\sigma = 0.84, 0.563, 0.444, 0.34, 0.25, 0.174, 0.111, 0.0625, 0.0278, 0.00694, 0$.

Mit steigender Pfropfdichte verändert sich das Verhalten der Dichte dahingehend, daß zum einen sowohl die Anzahl der erkennbaren Peaks als auch deren Höhe abnimmt und zum anderen die Gesamtdichte der Lösungsmittelteilchen innerhalb des Bereiches, den auch die Bürste erreichen kann, abnimmt und den Plateauwert der Lösungsmitteldichte erst mit größerem Abstand von der Wand erreicht.

Bei den höchsten betrachteten Propfdichten erscheint dann neben den Peaks bei $d = 0.5$ und $d = 1.5$, die von flach an der Wand liegenden Teilchen bzw. von einer zweiten Lage darauf ebenfalls flach liegender Teilchen herstammen, eine weitere Peakstruktur mit einem Abstand von $d \simeq d_0$, der mittleren Länge einer Bindung der Bürste zwischen den Peaks, der von den sich innerhalb der mehr und mehr senkrecht zur Wand stehenden Bürste befindenden Lösungsmittelteilchen stammt, die sich in den Lücken zwischen den Ketten aufhalten (eine Sammlung von Konfigurationsschnappschüssen findet sich ab Seite 69).

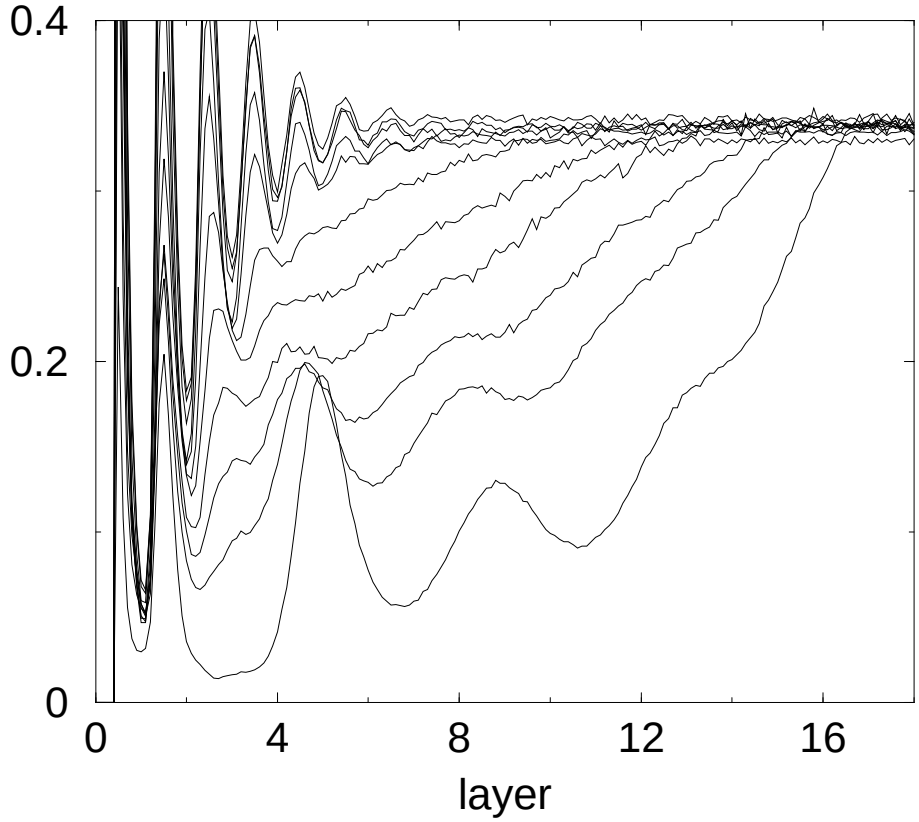

Abbildung 4.18:

Dichteprofil der Lösungsmittelteilchen in der Nähe einer Wand für unterschiedliche Pfropfdichten (Ausschnittsvergrößerung von Abb. 4.17). Die verschiedenen Kurven entsprechen in der Reihenfolge ihres Abweichens von der Teilchendichte im Lösungsmittel von rechts nach links den Pfropfdichten $\sigma = 0.84, 0.563, 0.444, 0.34, 0.25, 0.174, 0.111, 0.0625, 0.0278, 0.00694, 0.$

Der Grund dafür, daß der Abstand dieser Peaks voneinander nicht exakt d_0 beträgt sondern eher darunter liegt, ist die auch bei hohen Pfropfdichten nicht senkrecht stehende Bürste. Dadurch reduziert sich der Abstand zwischen den Peaks um einen Faktor $\cos(\theta)$, wobei θ der mittlere Winkel zwischen den Ketten und der z-Achse ist.

Die Verteilung der z-Komponenten der Bindungen bl_z, die als $bl\cos(\alpha)$ definiert ist, wobei bl die Länge einer Bindung ist und α der Winkel zwischen dieser Bindung und der z-Achse, zeigt zwei ausgeprägte Peaks bei Werten von 0 und 0.5 für die z-Komponente der Bondlänge, die über den gesamten betrachteten Pfropfdichtenbereich erhalten bleiben.

Bei den niedrigsten betrachteten Pfropfdichten ist danach ein Abfall der Verteilung, der, wie im kleinen Bild in Abbildung 4.19 deutlich zu sehen ist, exponentiell verläuft, zu erkennen. Bei höheren Pfropfdichten findet sich zu größeren Werten der z-Komponente der Bondlängen hin ein weiterer breiter Peak.

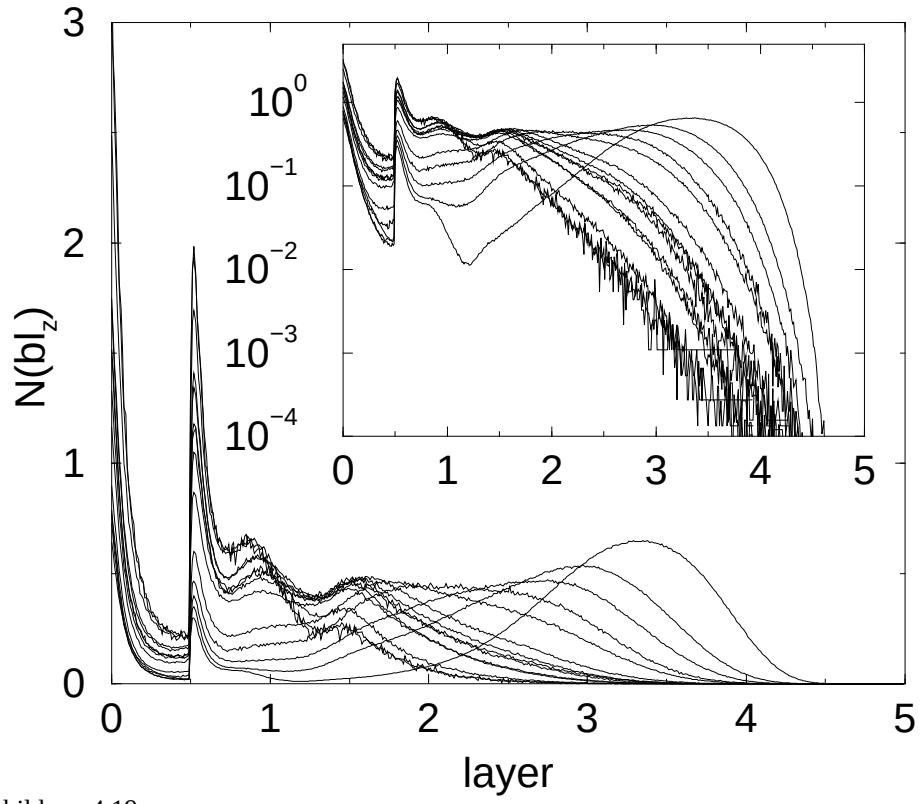

Abbildung 4.19:

Normierte Verteilung der z-Komponenten der Bondlängen für unterschiedliche Pfropfdichten. Die Kurven entsprechen von rechts nach links den Pfropfdichten $\sigma = 0.84, 0.563, 0.444, 0.34, 0.25, 0.174, 0.111, 0.0625, 0.0278, 0.00694, 0$

Die beiden ersten Peaks stammen hauptsächlich von Bindungen, die sich zwischen zwei Kettenteilchen von flach auf dem Boden liegenden Ketten befinden (Peak bei 0), bzw. von Bindungen zwischen Kettenköpfen und dem ersten Kettenteilchen von flach an der Wand liegenden Ketten (Peak bei 0.5).

Daß diese Peaks auch bei hohen Pfropfdichten nicht verschwinden, mag zunächst verwundern, ist jedoch leicht zu verstehen, wenn man die Verteilung der z-Komponenten der Kettenenden (Abb. 4.21) betrachtet.

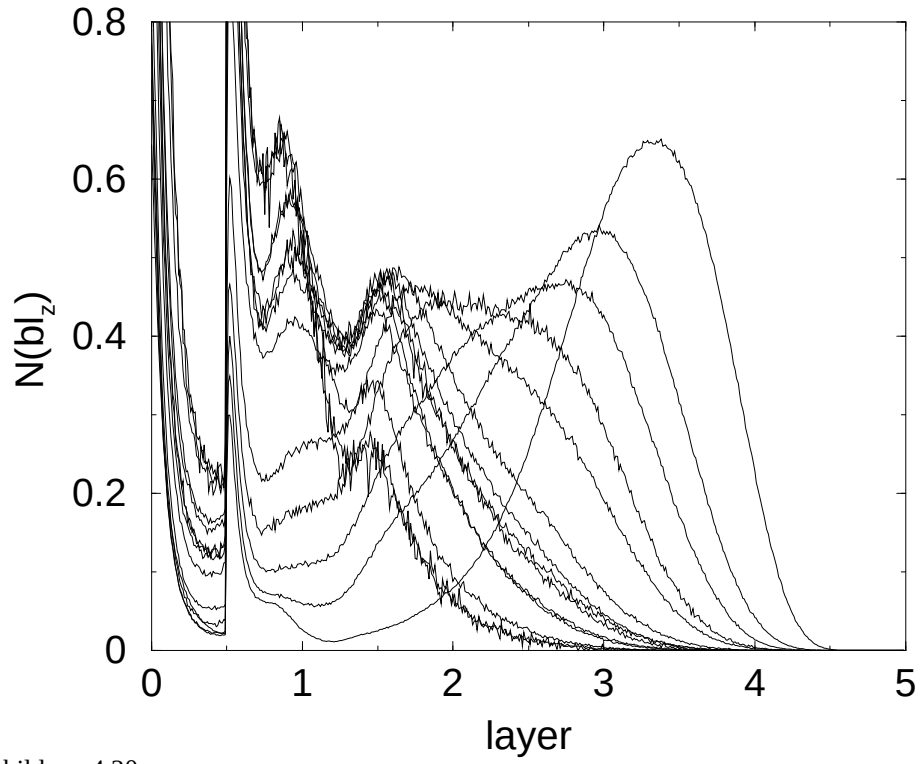

Abbildung 4.20:

Normierte Verteilung der z-Komponenten der Bondlängen für unterschiedliche Pfropfdichten (Ausschnittsvergrößerung von Abb. 4.19). Die Kurven entsprechen von rechts nach links den Pfropfdichten $\sigma = 0.84, 0.563, 0.444, 0.34, 0.25, 0.174, 0.111, 0.0625, 0.0278, 0.00694, 0$

Die Verteilung der z-Komponenten der Kopf-Ende-Vektoren der Ketten liefert ein ähnliches Bild wie die Verteilung der z-Komponenten der Verbindungsbonds. Es existieren relativ scharfe, zu größeren Werten breiter werdende, Peaks bei Werten von $0.5, 1.5, 2.5$ und 3.5, die mit zunehmender Pfropfdichte niedriger werden und schließlich gänzlich verschwinden. Diese Peaks stammen von Ketten bzw. deren Endteilchen, die mit ihren Mittelpunkten auf einer Schicht von $0, 1, 2$ oder 3 Lösungsmittelteilchen liegen und erzeugen somit die Peaks im Abstand $(n + 0.5)\sigma_s$.

Bei niedrigen Pfropfdichten fällt die Kurve nach diesen Peaks stark ab, bei den beiden geringsten betrachtehten Pfropfdichten sogar exponentiell (kleines Bild in Abb. 4.22).

Bei den größeren Pfropfdichten entsteht aus diesem Abfall zu größeren Werten der z-Komponente der Kettenlänge, also aufrechter stehenden Ketten, ein weiterer, deutlich breiterer Peak, der bei den größten betrachteten Pfropfdichten das Diagramm dominiert.

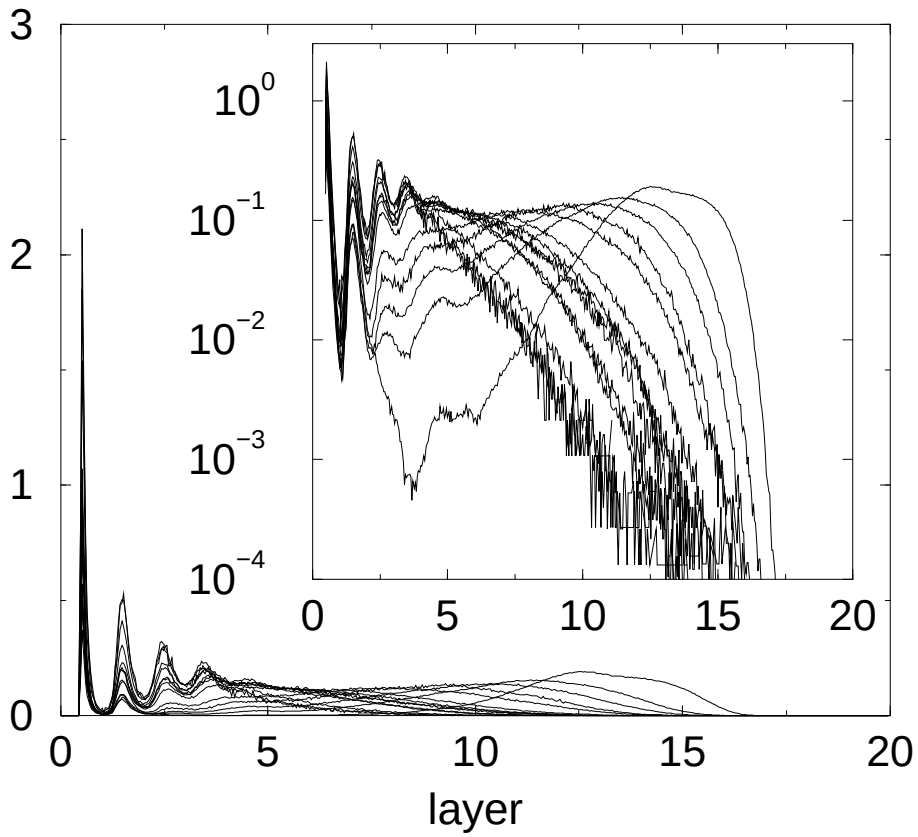

Abbildung 4.21:

Normierte Verteilung der z-Komponenten der Kopf-Ende-Vektoren der Polymerketten für unterschiedliche Pfropfdichten. Die Kurven entsprechen von rechts nach links den Pfropfdichten $\sigma = 0.84$, 0.563, 0.444, 0.34, 0.25, 0.174, 0.111, 0.0625, 0.0278, 0.00694, 0

Obwohl die Höhe der ersten beiden Peaks mit wachsender Pfropfdichte abnimmt, verschwinden sie auch bei größeren Pfropfdichten nicht gänzlich, was zeigt, daß es auch dann noch Ketten gibt, die flach an der Wand liegen.

Dies läßt sich verstehen, wenn man bedenkt, daß die Kettenköpfe völlig wechselwirkungsfrei sind und somit auch keine excluded-volume-Effekte mit ihnen auftreten können. Dadurch existiert ein freies Volumen für unter der ersten Schicht der Kettenteilchen liegende Lösungsmittelteilchen, aus denen natürlich auch Ketten gebildet werden können, die dann zumeist flach an der Wand liegen.

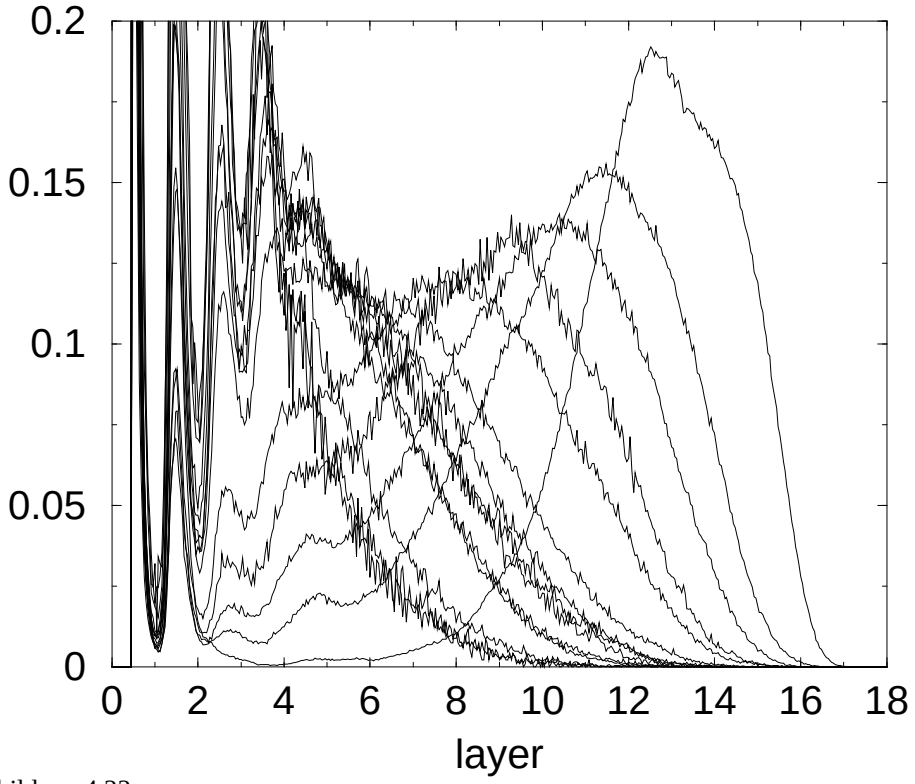

Abbildung 4.22:

Normierte Verteilung der z-Komponenten der Kopf-Ende-Vektoren der Polymerketten für unterschiedliche Pfropfdichten (Ausschnittsvergrößerung von Abb. 4.21). Die Kurven entsprechen von rechts nach links den Pfropfdichten $\sigma = 0.84, 0.563, 0.444, 0.34, 0.25, 0.174, 0.111, 0.0625, 0.0278, 0.00694, 0$

Außer der Verteilung der z-Koordinaten der Kettenenden, die nur eine Aussage über das Verhalten der Kette als ganzes macht, kann man sich die Verteilung der z-Komponenten derjenigen Kettenteilchen ansehen, die weder Kopf- noch Endteilchen sind. Dabei ähneln die Verteilungen für geringe Pfropfdichten sehr den Verteilungen der Kettenenden bei gleicher Pfropfdichte. Das heißt, es existieren sehr scharfe Peaks bei Abständen von $d = 0.5\sigma_s$, $1.5\sigma_s$ und $2.5\sigma_s$, sowie ein von einem kleineren und breiteren Peak bei $3.5\sigma_s$ unterbrochenen Abfall mit wachsendem Abstand von der Wand.

Zu größeren Pfropfdichten hin verändert sich das Bild dahingehend, daß die Verteilung der Kettenteilchen, die eine Überlagerung der Verteilungen der einzelnen Kettenteilchen, die an zweiter, dritter und vierter Stelle auf der Kette sitzen, neben den Peaks bei geringen Abständen, die auch bei niedrigen Pfropfdichten auftreten, eine Reihe von breiten Peaks bei größeren Abständen von der Wand aufweist.

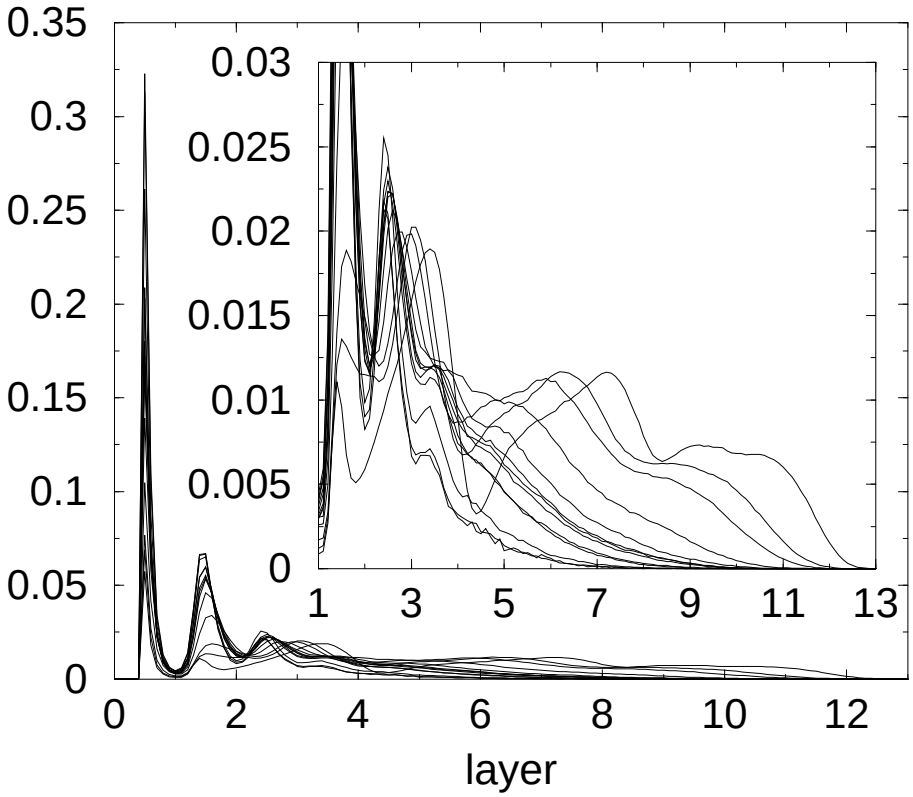

Abbildung 4.23:
Dichteverteilung der Kettenteilchen (ohne Kettenenden) für die Pfropfdichten (von rechts nach links in der Reihenfolge des Ablösens der Kurven von der x-Achse) $\sigma = 0.84, 0.563, 0.444, 0.25, 0.174, 0.141, 0.134, 0.128, 0.122, 0.111, 0.0625, 0.0278, 0.00694$

Abbildung 4.24 zeigt auf eins normierte Histogramme der Verteilungen der z-Komponente des zum nematischen Ordnungsparameter gehörenden Eigenvektors für verschiedene Pfropfdichten.

Für das System ohne Bürste und die Systeme mit den geringsten Pfropfdichten (Abb. 4.24 linke Spalte) läßt sich ein Peak bei $z = 0$ erkennen, der bei den Systemen mit höheren Pfropfdichten (mittlere und rechte Spalte) zu immer größeren Werten von z wandert und bei der höchsten betrachteten Pfropfdichte (Abb. 4.24 rechts unten) sein Maximum bei $z = 1$ hat.

Um den auffälligen Einbruch des Ordnungsparameters für die Dichte $\sigma = 0.00694$ in Abbildung 4.8 zu verstehen, ist es hilfreich, sich die Zeitreihe für die einzelnen Komponenten des Eigenvektors zum nematischen Ordnungsparameter anzusehen. Abbildung 4.25 zeigt diese Zeitreihen über etwa acht Millionen Monte Carlo-Schritte, was 3 Monaten CPU-Zeit auf einem Pentium III mit 800 MHz entspricht.

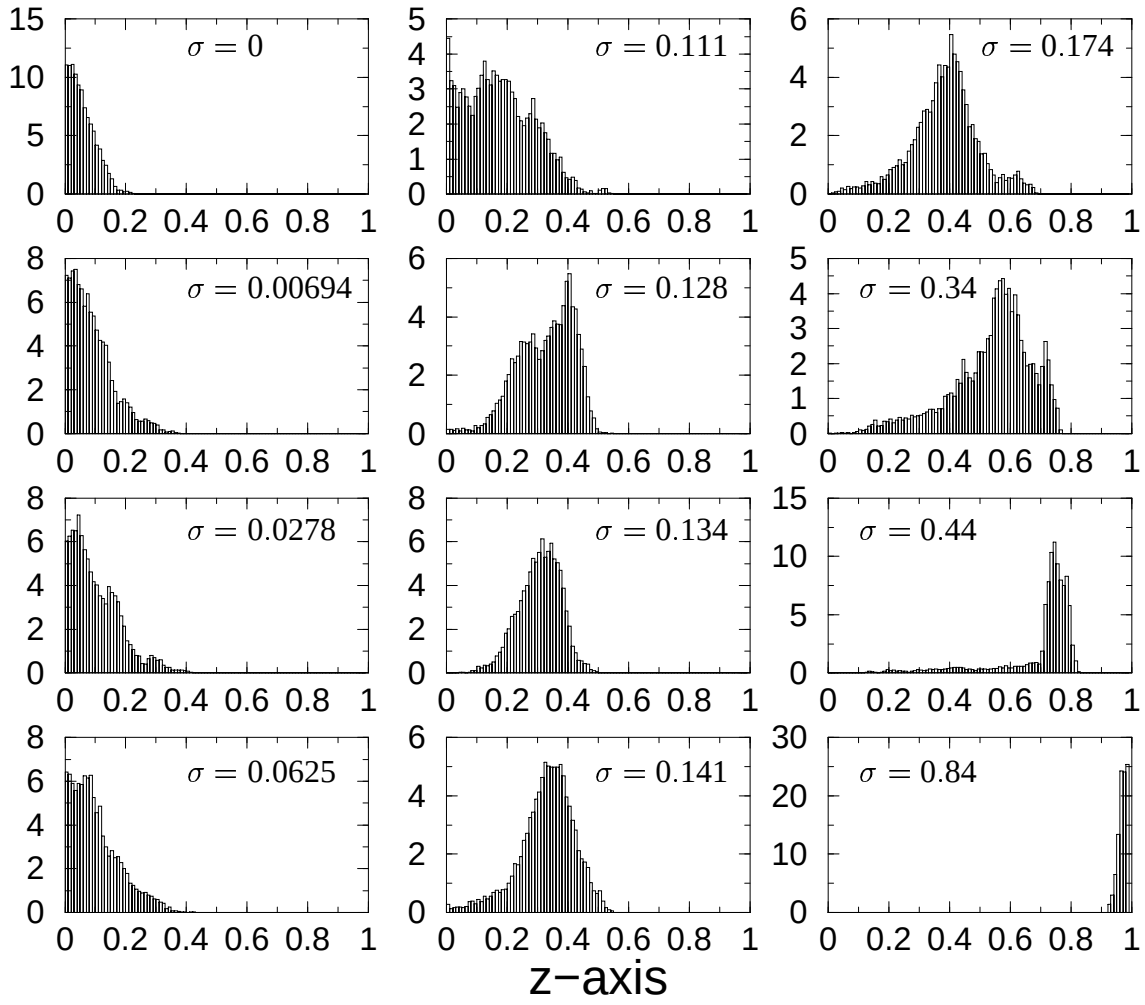

Abbildung 4.24:

Normierte Verteilung der z-Komponente des zum Ordnungsparameter gehörenden Eigenvektors für unterschiedliche Pfropfdichten σ

In der Abbildung fällt auf, daß die Kurve für die y-Komponente zwischenzeitlich stark ansteigt und fast den Wert der x-Komponente erreicht, was einer Ausrichtung entlang der Winkelhalbierenden der xy-Ebene entspräche.

Die Ursache hierfür ist eine Verdrillung des Systems, die nur bei dieser Pfropfdichte beobachtet wurde und die dadurch, daß sich zwischen den Teilchenachsen an den beiden Wänden ein Winkel von 90 Grad einstellt, hervorgerufen wird.

Durch die unterschiedliche Orientierungen der Teilchen an den Wänden wird bei der Mittelung über die Orientierung aller Lösungsmittelteilchen eine Ausrichtung in einer Zwischenrichtung das Ergebnis sein, wie in Abbildung 4.25 auch zu erkennen ist.

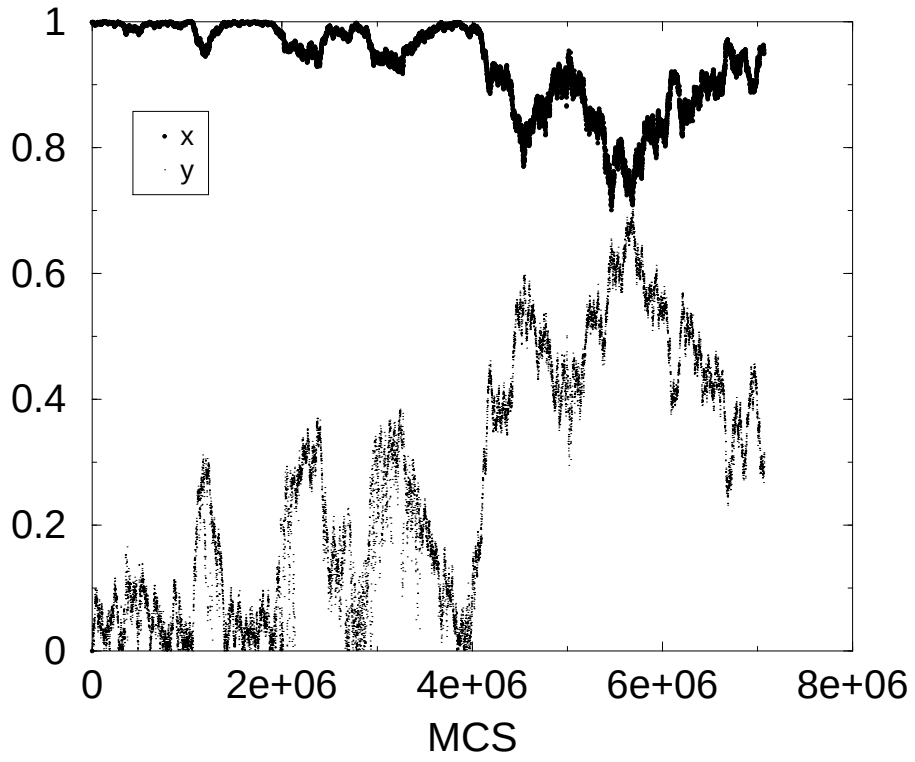

Abbildung 4.25:
Zeitreihe für die x- und y-Komponenten des Eigenvektors zum Ordnungsparameter für eine Pfropfdichte von $\sigma = 0.00694$

Daß sich das System keineswegs global in eine Zwischenrichtung einstellt, wie es ja nach Abbildung 4.25 denkbar wäre, ist in Abbildung 4.26, für die erst Daten nach einer Laufzeit von fünf Millionen MCS berücksichtigt wurden, deutlich zu erkennen. Hier ist die unterschiedliche Orientierung der Teilchen an den Wänden (an der in der Abbildung linken Wand in y-Richtung und an der rechten Wand in x-Richtung) und der Übergangsbereich zwischen den beiden Orientierungen dazwischen zu erkennen.

Wenn man sich den Bereich der Zeitreihe für die Komponenten des Ordnungsparameters vor der einsetzenden Verdrillung betrachtet (Abb. 4.27), so kann man hier keinerlei Unterschiede im Verhalten der Teilchen erkennen. An beiden Wänden sind die Lösungsmittelteilchen entlang der x-Achse orientiert und von einer Verdrillung ist auch noch nichts zu erkennen.

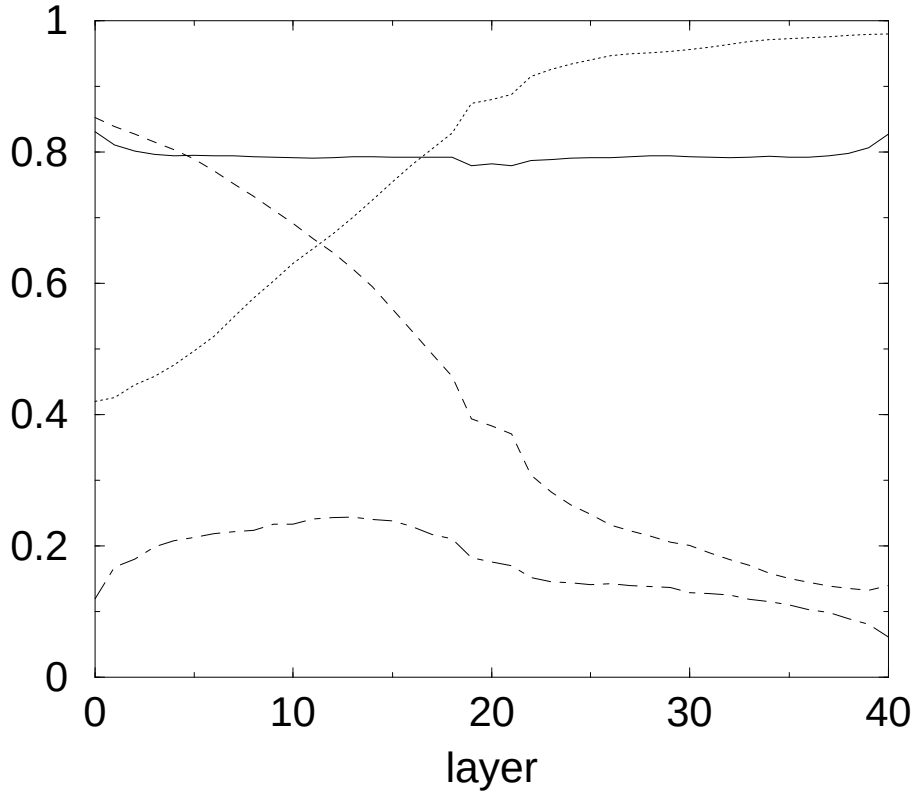

Abbildung 4.26:
Schichtabhängiges
Ordnungsparameterprofil (durchgezogenen Linie) und die x-, y- und z-Komponenten (gepunktete, gestrichelte und gestrichpunktete Linien) des zu diesem Eigenwert gehörenden Eigenvektors für die Pfropfdichte $\sigma = 0.00694$

Das Auftreten dieser Verdrillung ist ein finite-size-Effekt, der durch die geringe Grundfläche der Simulationsbox bedingt ist und bei größeren Systemgrößen nicht mehr auftreten wird, da die Energie, die für eine solche Verdrillung benötigt wird, proportional zur Grundfläche der Simulationsbox ist.

Bei hinreichend langen Laufzeiten sollten jedoch bei allen betrachteten Pfropfdichten der Ordungsparameter die Richtung der Neigung ändern und durch Zwischenzustände laufen, auch wenn das hier nicht beobachtet wurde.

In Abbildung 4.28 finden sich Darstellungen des Ordnungsparameters und der Komponenten des dazugehörigen Eigenvektors für unterschiedliche Pfropfdichten für einzelne Schichten innerhalb des Systems als Funktion des Abstands der betrachteten Schichten von der unteren Wand.

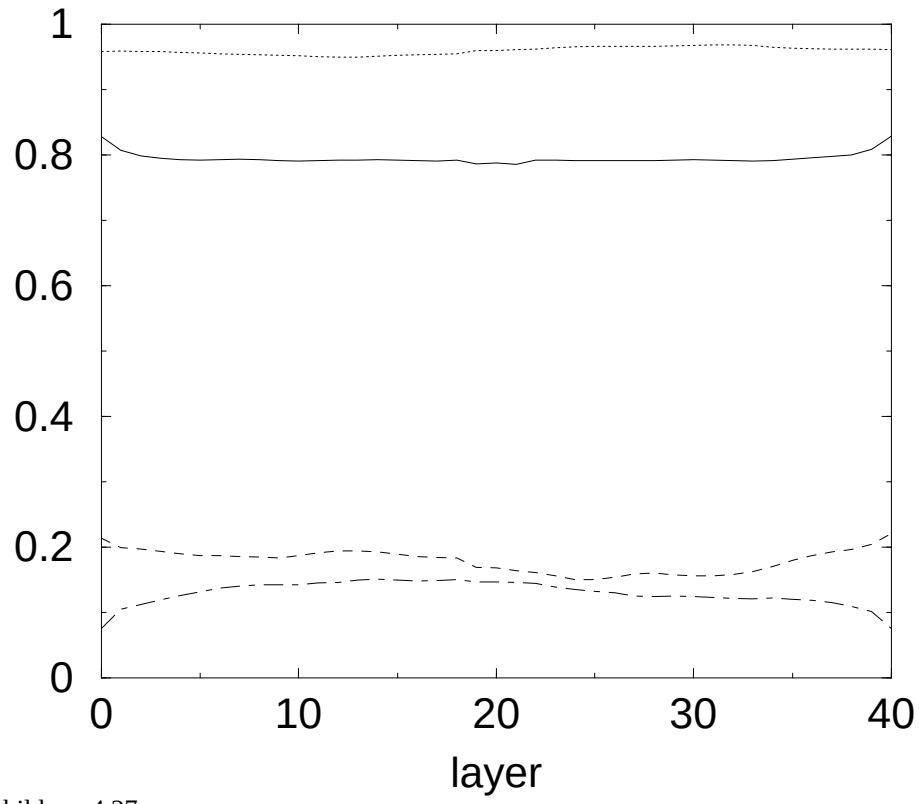

Abbildung 4.27:
Schichtabhängiges Ordnungsparameterprofil (durchgezogenen Linie) und die x-, y- und z-Komponenten (gepunktete, gestrichelte und gestrichpunktete Linien) des zu diesem Eigenwert gehörenden Eigenvektors für die Pfropfdichte $\sigma = 0.00694$

Dabei ist, unabhängig von den Unterschieden in den Komponenten des Eigenvektors, bei allen betrachten Pfropfdichten folgendes gleich: Der Ordungsparameter hat seine Maximalwerte direkt an der Wand und nimmt zur Systemmitte hin ab. Die Komponenten des Eigenvektors zeigen die Richtung der Neigung der Lösungsmittelteilchen an und variieren sehr stark mit der Pfropfdichte.

Bei niedrigen Pfropfdichten (Abb. 4.28 (a) – (f)) ist die z-Komponente nahe Eins und die x- bzw. y- Komponenten nahe Null. Die Abweichungen von diesem Verhalten in den Abbildungen (b) und (d) stammen von Verdrillungen innerhalb des Lösungsmittels, wie sie in (3.3) bereits angesprochen wurden. Dabei verhält sich das System in Bezug auf die Orientierung der Teilchen direkt an der Wand an den beiden Wänden unterschiedlich und reduziert somit die Ordnung im Gesamtsystem (siehe auch Abb. 4.8 und den dortigen Einbruch des Ordnungsparameters bei den zu (b) bzw. (d) gehörigen Pfropfdichten).

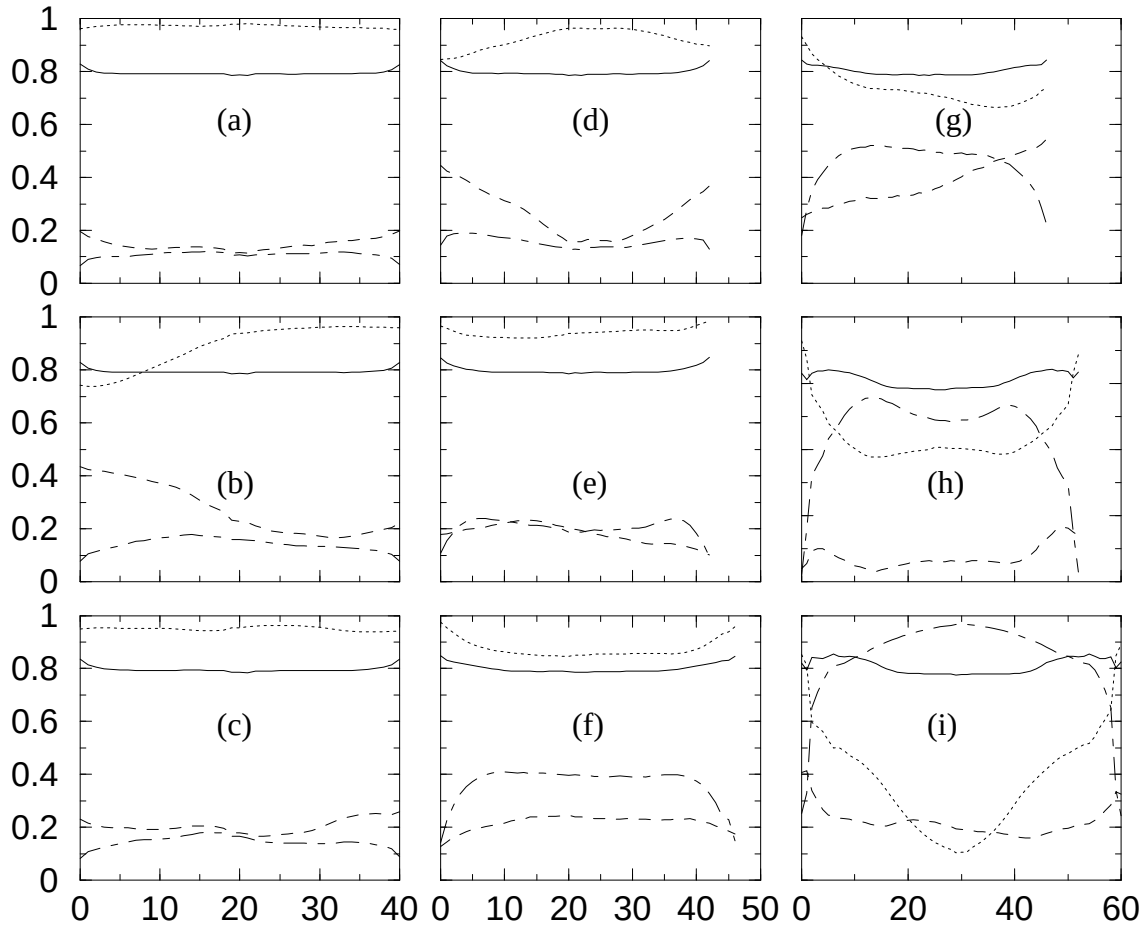

Abbildung 4.28:

Schichtabhängige Ordnungsparameterprofile für unterschiedliche Pfropfdichten. Die Abbildungen gehören dabei zu den folgenden Pfropfdichten: (a) $\sigma = 0$, (b) $\sigma = 0.00694$, (c) $\sigma = 0.0278$, (d) $\sigma = 0.0625$, (e) $\sigma = 0.111$, (f) $\sigma = 0.174$, (g) $\sigma = 0.25$, (h) $\sigma = 0.563$, (i) $\sigma = 0.84$. Die durchgezogenen Linien zeigen den Wert des nematischen Ordnungsparameters, also des größten Eigenwerts der Ordnungsparametermatrix, und die gepunkteten, gestrichelten und gestrichpunkteten Linien die x-, y- und z-Komponenten des zu diesem Eigenwert gehörenden Eigenvektors.

Die Abbildungen für höhere Pfropfdichten (g) – (i) zeigen dagegen ein anderes Verhalten. Hier sinkt die an den Wänden dominierende x-Komponente mit zunehmender Pfropfdichte stark ab und unter den Wert für die z-Komponente.

Dies zeigt, daß sich das System mehrheitlich (also sobald die Wand keinen orientierenden Einfluß mehr ausüben kann) nicht mehr entlang der x-Achse, sondern entlang der z-Achse ausrichtet.

In Teilabbildung (i) kann man außerdem nochmals die Entkopplung von Bürste und Lösungsmittel erkennen. Sie wird deutlich durch die Schulter in der Kurve für die x-Komponente der Richtung des Ordnungsparameters.

In allen Teilbildern von Abbildung 4.28 ist die x-Achse gegenüber der y-Achse nur dadurch ausgezeichnet, daß alle Systeme mit einer Teilchenorientierung entlang der x-Achse aufgesetzt wurden.

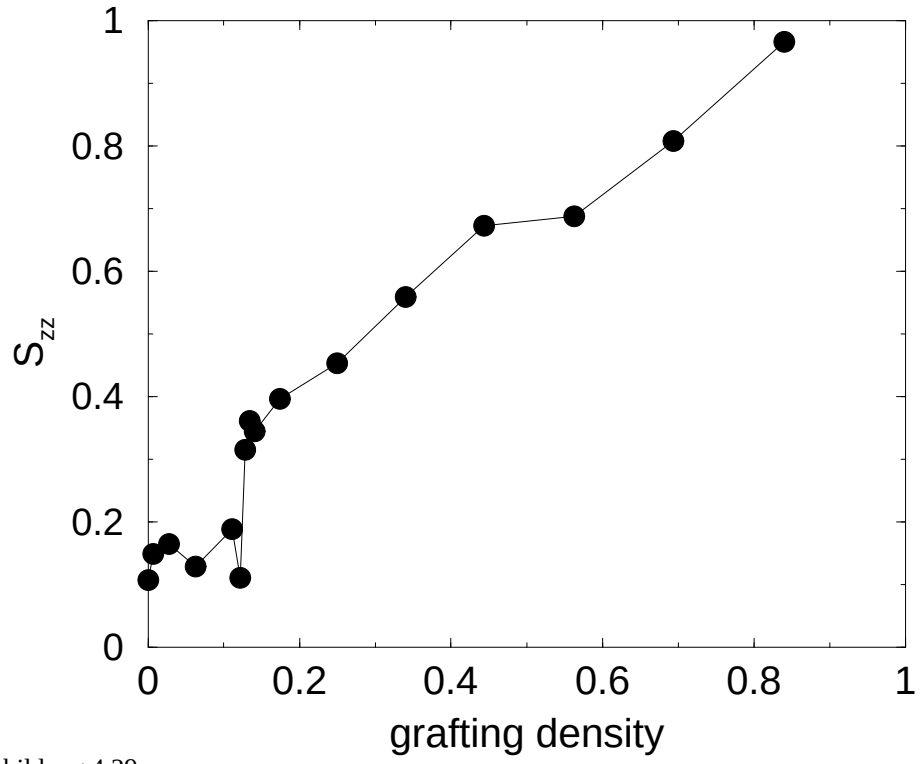

Abbildung 4.29:
z-Komponente des Eigenvektors zum Ordnungsparameters in der Mitte der Schicht in Abhängigkeit von der Pfropfdichte σ

Trägt man die in der Mitte des Bulks gemessene z-Komponente des Eigenvektors zum Ordnungsparameter gegen die Pfropfdichte auf, so erhält man Abbildung 4.29. Die Kurve hat bei niedrigen Pfropfdichten einen konstanten Wert von ≈ 0.15, um dann einen Sprung bei einer Pfropfdichte von $\sigma = 0.13$ aufzuweisen. Danach steigt sie mit wachsender Pfropfdichte linear weiter an um bei $\sigma = 0.84$ einen Wert nahe eins zu erreichen.

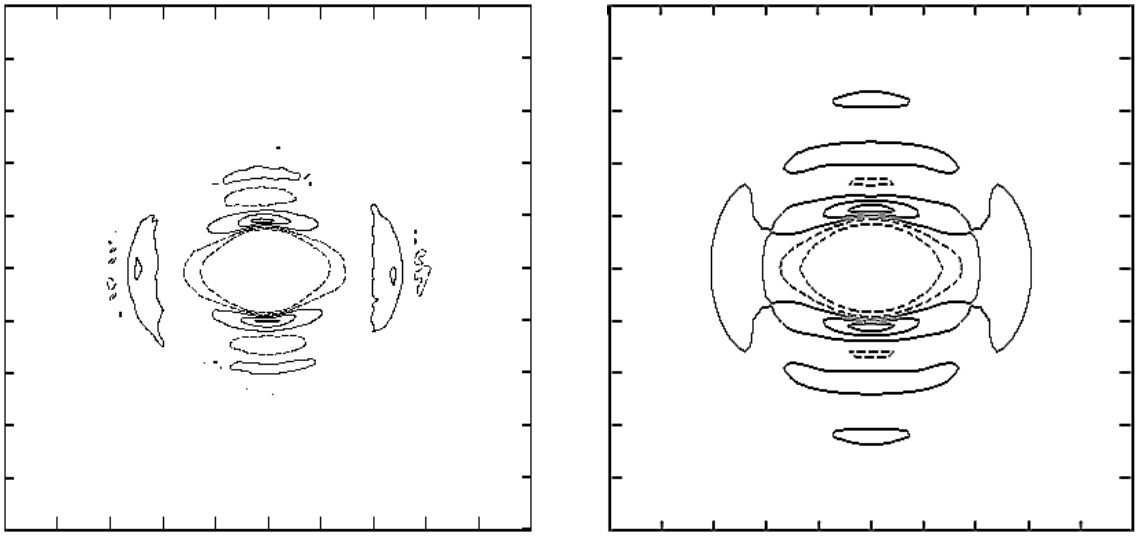

Abbildung 4.30:
Zweidimensionale Paarkorrelationsfunktion in der xy-Ebene aus dieser Simulation (linke Seite) und aus [43] (rechte Seite)

In Abbildung 4.30 ist auf der linken Seite die zweidimensionale Paarkorrelationsfunktion in der xy-Ebene für die Teilchen innerhalb einer Schicht der Dicke σ_s direkt an der Wand aufgetragen. Die gestrichelten Höhenlinien geben dabei Werte unterhalb des Mittelwerts an und die durchgezogenen Linien Werte oberhalb des Mittelwerts.

Um das sich in der Mitte des Teilbildes befindende Teilchen gibt es Peaks für besonders hohe Aufenthaltswahrscheinlichkeiten anderer Teilchen. Man kann die bevorzugte parallele Anordnung der Teilchen zueinander an den ober- bzw. unterhalb der Mitte liegenden Peaks im Abstand 1 schön erkennen. Die beiden Peaks rechts und links der Mitte stammen von Teilchen, die in der nächsthöheren oder -niedrigeren Teilchenlage liegen.

Das rechte Teilbild von Abbildung 4.30 ist die in einer Simulation eines Bulksystem errechnete Paarkorrelationsfunktion [43]. Sie stimmt mit der Korrelationsfunktion an der Wand gut überein.

In den Konfigurationsschnappschüssen (Abb. 4.31 und 4.32) sieht man jeweils auf der linken Seite die Ansicht mit allen undurchsichtigen Lösungsmittelteilchen und auf der jeweils rechten Seite die Ansicht mit durchsichtigen Lösungsmittelteilchen.

Die vier gezeigten Schnappschüsse wurden exemplarisch ausgewählt und illustrieren das bereits vorher beschreibene Verhalten. Bei den Pfropfdichten $\sigma = 0.0278$ und $\sigma = 0.111$ sieht man die liegenden Lösungsmittelteilchen, und bei $\sigma = 0.25$ kann man die Neigung in eine Zwischenrichtung erkennen. Im letzten Bild kann man bei $\sigma = 0.84$ die Entkopplung des Lösungsmittels von der Bürste gut erkennen. Die Bürste steht schräg, und das Lösungsmittel verankert senkrecht auf der durch die Bürste gebildete Oberfläche.

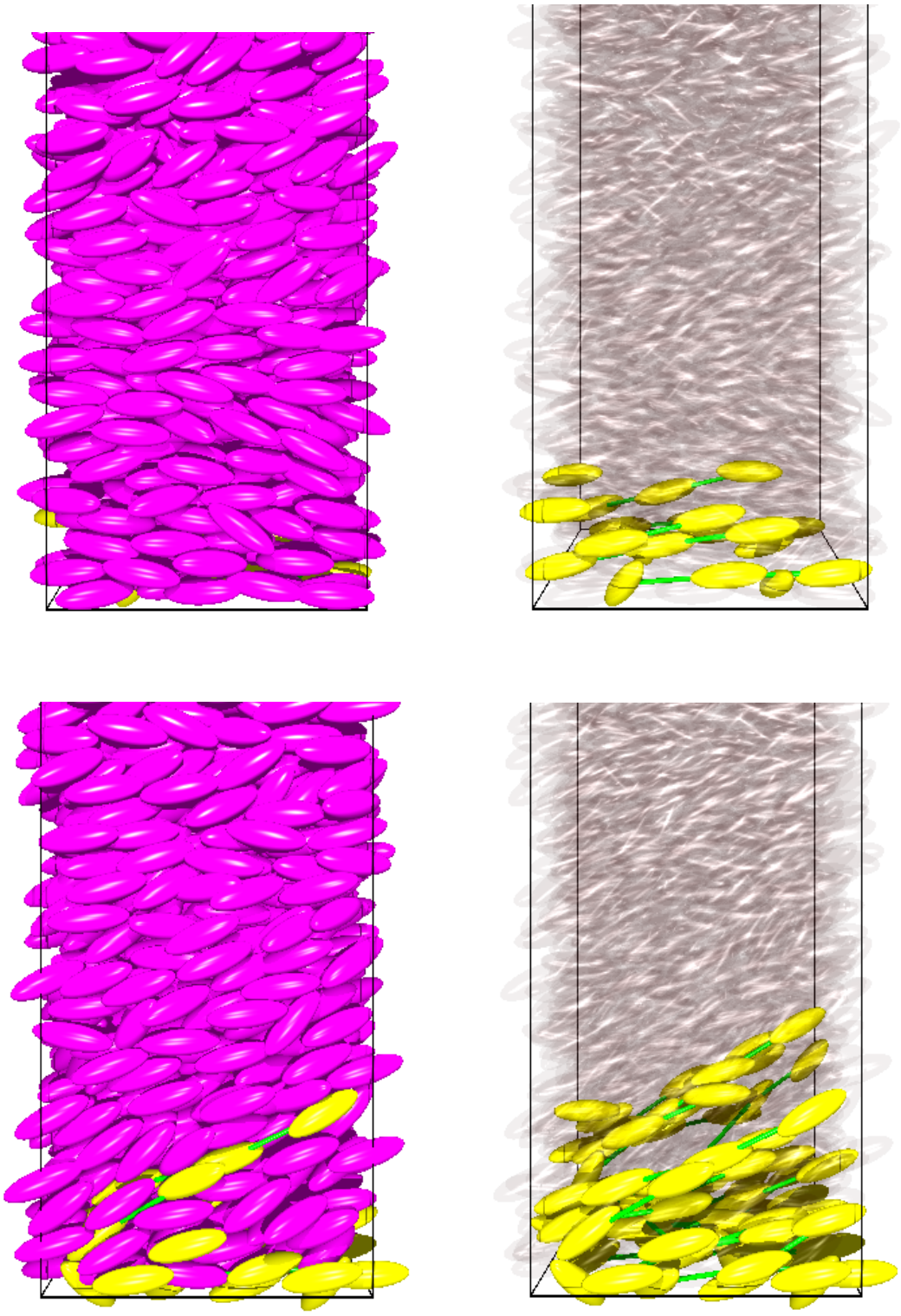

Abbildung 4.31:

Konfigurationsschnapschüsse für die Pfropfdichten $\sigma = 0.0278$ (oben) und $\sigma = 0.111$ (unten)

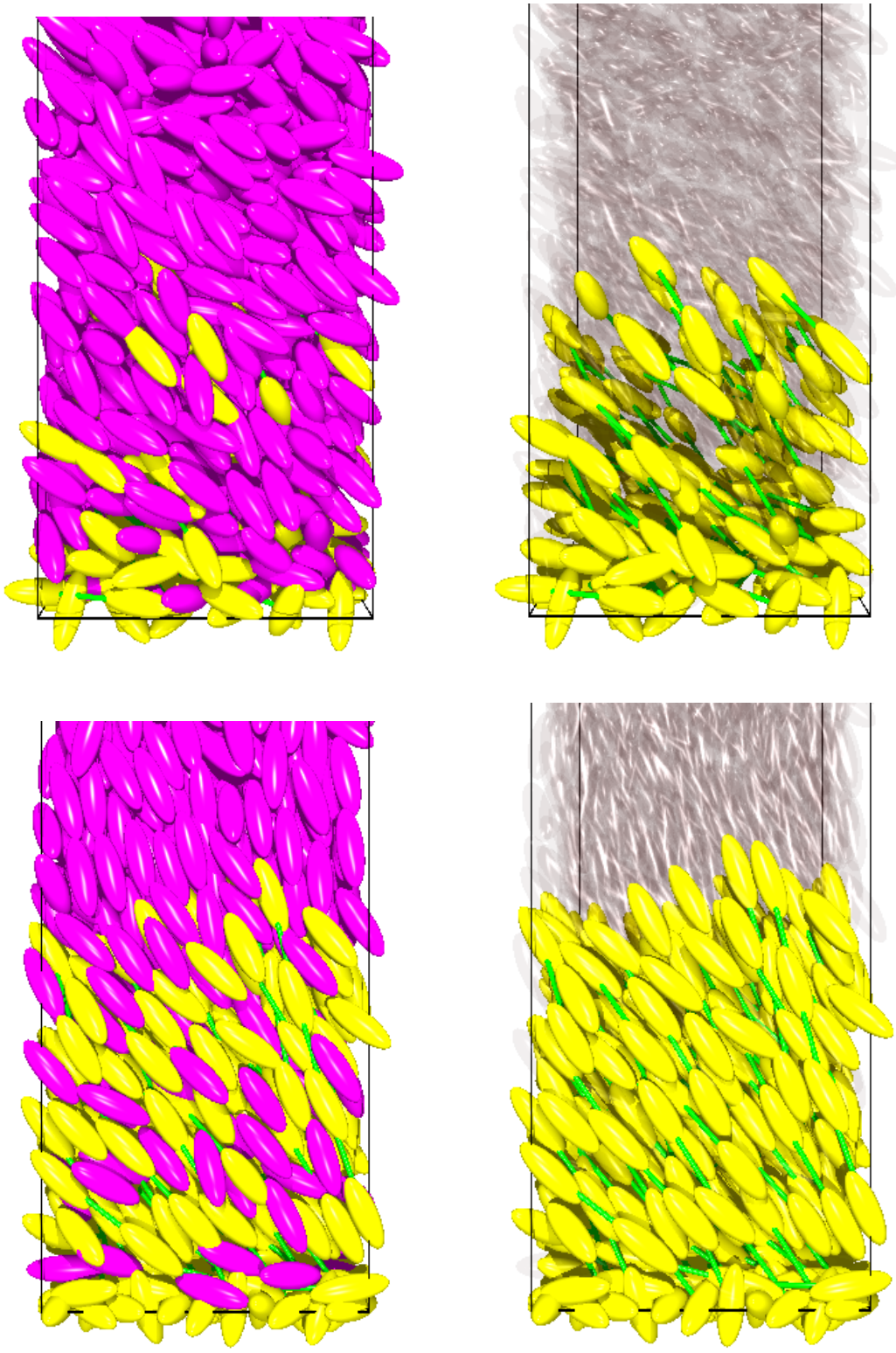

Abbildung 4.32:

Konfigurationsschnapschüsse für die Pfropfdichten $\sigma = 0.25$ (oben) und $\sigma = 0.84$ (unten).

Kapitel 5

Zusammenfassung

In der vorliegenden Arbeit wurde ein System von oberflächengebundenen flüssigkristallinen Polymeren in nematischer Lösung mittels Monte Carlo-Simulationen im NpT-Ensemble untersucht. Dabei wurden die Wechselwirkungen der Lösungsmittelteilchen untereinander und die Wechselwirkung zwischen Lösungsmittelteilchen und Teilchen in der Bürste durch ein im Minimum abgeschnittenes und nach oben verschobenes Gay-Berne-Potential beschrieben. Darüber hinaus wirkten zwischen Teilchen innerhalb einer Kette zusätzliche Bondlängen und Bondwinkelpotentiale.

Die Größe der verwendeten Simulationsbox wurde in x- und y-Richtung festgehalten und konnte in z-Richtung variiert werden. Somit kamen außer den klassischen Monte Carlo-Moves auch kollektive Bewegungsversuche in der Form von Reskalierungen der Simulationsbox und des Neuwachsens von ganzen Ketten zur Anwendung.

Mit den durchgeführten Simulationen konnte gezeigt werden, daß ein Phasenübergang existiert zwischen einer Phase mit flach entlang den Wänden ausgerichteten Teilchen und einer Phase mit Teilchen, deren Orientierung nicht parallel zur Wand ist. Dieser Phasenübergang tritt im hier betrachteten System bei einer Pfropfdichte von $\sigma = 0.13/\sigma_s^2$ auf.

Weiterhin konnte bei hohen Pfropfdichten ($\sigma \geq 0.8/\sigma_s^2$) eine Phase mit einer Ausrichtung der Lösungsmittelteilchen senkrecht zur Grundfläche gefunden werden, obwohl die Kette auch bei diesen Pfropfdichten nicht senkrecht steht.

Die Berechnung der Verankerungstärke[1] (wie in [12]) war auf Grund zu geringer Statistik nicht möglich.

[1]Um die Verankerungsstärke zu bestimmen, benötigt man die elastischen Konstanten. Sie haben die Werte $K_{11} = 0.66 \pm 0.04$, $K_{22} = 0.39 \pm 0.03$, $K_{33} = 2.25 \pm 0.06$ [44].

Experimentelle Arbeiten verwenden in der Regel flüssigkristalline Seitenkettenpolymere für die Modellierung einer Bürste [45, 46, 47, 48], die auch deutlich länger als die in dieser Arbeit untersuchten Ketten sind.

Da allerdings kurze Ketten, wie sie in dieser Arbeit simuliert wurden, experimentell noch nicht untersucht worden sind, ist es schwierig, hier Vergleiche anzustellen.

Als konsequente Fortsetzung dieser Arbeit wäre zuerst eine Verbesserung der Statistik nötig, um den Phasenübergang besser eingrenzen zu können und eventuell die elastischen Konstanten und die Verankerungsstärke zu bestimmen. Weiterhin könnte auch eine Veränderung der einzelnen Systemparameter durchgeführt werden.

Mit hinreichend großer verfügbarer Rechenleistung könnte man zum Beispiel (bei fester Pfropfdichte) durch Variation des Druckes und/oder der Temperatur auch die Abhängigkeit des Überganges von diesen beiden Größen bestimmen.

Weiterhin könnte durch systematische Variation der Größen der Teilchen auf der Kette, der Anzahl der Teilchen auf einer Kette oder der Bondlängen innerhalb der Kette auch deren Einfluß auf den Phasenübergang zwischen geneigter und ungeneigter Phase untersucht werden. Dabei könnte zumindest im ersten Fall allerdings nicht mehr der in dieser Arbeit verwendete Algorithmus zum Wachsen einer Kette verwendet werden. Voruntersuchungen zur Variation der Bondlängen deuten bereits an, daß kürzere Bonds zu einer leichten Verschiebung des Phasenüberganges zu höheren Pfropfdichten führen.

Schließlich wäre auch eine Untersuchung der finite-size-Effekte innerhalb dieses Systems von Interesse. Insbesondere wäre es gut, Systeme zu untersuchen, die so groß sind, daß einzelne Ketten prinzipiell nicht mit ihren periodischen Abbildern wechselwirken können. Dafür wäre aber mindestens eine viermal so große Grundfläche erforderlich und ebenfalls eine viermal so große Teilchenzahl, um die Ausdehnung der Simulationsbox in z-Richtung zu erhalten, um dadurch zu gewährleisten, daß die Bürsten an den gegenüberligenden Wänden nicht direkt miteinander wechselwirken können.

Literaturverzeichnis

[1] P. G. de Gennes, J. Prost
The Physics of Liquid Crystals, Oxford University Press (1993)

[2] A. de Vries
'Structure and Classification of thermotropic liquid crystals' in
Liquid Crystals edt. F. D. Saeva , Decker inc. (1979)

[3] J. Cognard
Alignment of Liquid Crystals and their Mixtures, Gordon and Breach (1992)

[4] H. Yokoyama, S. Kobayashi, H. Kamei
J. Appl. Phys. **61** (1987) 4501

[5] H. Yokoyama
Mol. Cryst. Liq. Cryst. **165** (1988) 265

[6] T. J. Sluckin, A. Poniewierski
in: Fluid Interfacial Phenomena C. A. Croxton (edt.) Wiley (1996)

[7] A. L. Alexe-Ionescu, G. Barbero, A. Ignatov, E. Miraldi
Appl. Phys. A **56** (1993) 453

[8] N. L. Abbott
Surface effects on orientation of liquid crystals
in: Current Opinions in Colloid & Interface Science **2** (1997) 76

[9] J. Stelzer, R. Hirning, H.-R. Trebin
J. Appl. Phys. **74** (1993) 6046

[10] J. Stelzer, L. Longa, H.-R. Trebin
Phys. Rev. E **55** (1997) 7085

[11] T. P. Doerr, P. L. Taylor
Int. J. Mod. Phys. C **10** (1999) 415

[12] D. Andrienko, G. Germano, M. P. Allen
Phys. Rev. E **62** (2000) 6688

[13] T. Gruhn, M. Schoen
J. Chem. Phys. **108** (1998) 9124

[14] G. D. Wall, D. J. Cleaver
Phys. Rev. E **56** (1997) 4306

[15] M. P. Allen
Mol. Phys. **96** (1999) 1391

[16] P. G. de Gennes
J. de Phys. (Paris) **37** (1976) 1443

[17] S. T. Milner
Science **251** (1991) 905

[18] A. Halperin, M. Tirell, T. P. Lodge
Adv. Pol. Sci. **100** (1992) 31

[19] A. Halperin
in: Soft Order in Physical Systems, Y. Rabin and R. Bruinsma (edt.), Plenum Press, New York (1994)

[20] A. Takahashi, M, Kawaguchi
Adv. Pol. Sci. **46** (1986) 1

[21] T. M. Birshtein, Y. V. Liatskaya, E. B. Zhulina
Polymer **31** (1990) 2185

[22] G. J. Fleer, M. A. Cohen-Stuart, J. M. H. M. Scheutjens, T. Cosgrove, B. Vincent
in: Polymers at Interfaces, Chap. 8, Chapman&Hall (1993)

[23] V. M. Amoskov,T. M. Birshtein, V.A. Pryamitsyn
Macromolecules **29** (1996) 7240

[24] T. M. Birshtein, V. M. Amoskov
Comp. Theo. Pol. **10** (2000) 159

[25] B. J. Berne, P. Pechukas
J. Chem. Phys. **16** (1972) 4213

[26] J. G. Gay, B. J. Berne
J. Chem. Phys. **74** (1981) 3316

[27] E. de Miguel, L. F. Rull, M. K. Chalam, K. E. Gubbins, F. van Swol
Mol. Phys. **72** (1990) 593

[28] M. K. Chalam, K. E. Gubbins, E. de Miguel, L. F. Rull
Mol. Sim. **7** (1991) 357

[29] M. P. Allen, J. T. Brown, M. A. Warren
J. Phys. Cond. Matter **8** (1996) 9433

[30] E. de Miguel, E. M. del Rio, J. T. Brown, M. P. Allen
J. Chem. Phys. **105** (1996) 4234

[31] L. F. Rull
Physica A **220** (1995) 113

[32] M. A. Bates, G. R. Luckhurst
J. Chem. Phys. **104** (1996) 6696

[33] G. Ayton, D. Q. Wei, G. N. Patey
Phys. Rev. E. **55** (1997) 447

[34] R. Memmer
Liquid Crystals **27** (2000) 533

[35] J.-H. Kim, S. Kumar, S.-D. Lee
Phys. Rev. E **57** (1998) 5644

[36] J. M. Geary, J. W. Goodby, A. R. Kmetz, J. S. Patel
J. Appl. Phys. **62** (1987) 4100

[37] M. F. Toney, T. P. Russel, J. A. Logan, H. Kiguchi, J. M. Sands, S. K. Kumar
Nature **374** (1995) 709

[38] J. M. Hammersley, D. C. Handscomb
Monte Carlo Methods, Methuen (1964)

[39] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller
J. Chem. Phys. **21** (1953) 1087

[40] M. P. Allen, D. J. Tildesley
Computer Silumations in Chemical Physics, Kluwer Academic Publishers (1993)

[41] D .Frenkel, G.C.A.M. Mooji, B. Smit
J. Phys. Cond. Mat. **3** (1991) 3053

[42] N.M. Rosenbluth, A.W. Rosenbluth
J. Chem. Phys. **23** (1955) 356

[43] Nguyen H. Phuong,F. Schmid
Private Mitteilung

[44] G. Germano
in Vorbereitung

[45] F. Benmouna, B. Peng, J. Rühr, D. Johannsmann
Liq. Cryst. **26** (1999) 1655

[46] B. Peng, D. Johannsmann, J. Rühe
Macromolecules **32** (1999) 6759

[47] B. Peng, J. Rühe, D. Johannsmann
Adv. Mat. **12** (2000) 821

[48] F. Benmouna, B. Peng, J. Gapinski, A. Patkowski, J. Rühe, D. Johannsmann
in Vorbereitung

Anhang A

Berechnung des Abstandes eines Ellipsoids von einer glatten Wand

Ausgehend von der Gleichung zur Berechnung des Schnittpunktes einer Tangente an eine Ellipse mit Mittelpunkt im Koordinatenursprung

$$\frac{xx_0}{a^2} + \frac{yy_0}{b^2} = 1 \tag{A.1}$$

berechnen wir die Koordinaten des Schnittpunktes (x_0, y_0) der Tangente mit den Koordinatenachsen zu

$$x(y=0) = \frac{a^2}{x_0} \tag{A.2}$$

$$y(x=0) = \frac{b^2}{y_0} \tag{A.3}$$

Somit ergibt sich für den Steigungswinkel θ der Tangente

$$\tan\theta = \frac{x_0\, b^2}{y_0\, a^2} \tag{A.4}$$

Löst man Gleichung (A.4) nach y_0 auf und setzt man dann in die Ellipsengleichung im Punkte (x_0, y_0) ein, so ergibt sich für die Koordinaten des Schnittpunktes

$$x_0 = \sqrt{\frac{a^2}{\frac{b^2}{a^2 \tan^2\theta} + 1}} \tag{A.5}$$

$$y_0 = \frac{b^2}{a^2} \frac{1}{\tan\theta} \sqrt{\frac{a^2}{\frac{b^2}{a^2 \tan^2\theta} + 1}} \tag{A.6}$$

Der Abstand g des Koordinatenursprunges zum Schnittpunkt (x_0, y_0) der Tangente läßt sich wie folgt ausdrücken:

$$g = \frac{b^2}{y_0} \cos\theta \tag{A.7}$$

Das Einsetzen von Gleichung (A.6) in diese Gleichung und Umformung führt zu

$$g = \sqrt{a^2 \sin^2\theta + b^2 \cos^2\theta} \tag{A.8}$$

Mit der Ersetzung von a durch $\sigma_{\parallel}$ und b durch $\sigma_{\perp}$ sowie der Verwendung des Winkels $\phi = \pi - \theta$ geht Gleichung (A.8) somit in Gleichung (2.16) über.

Anhang B

Herleitung der theoretischen Winkelverteilung

Die theoretische Verteilung für die Winkel zwischen zwei Bindungen erfüllt

$$P \propto e^{-\beta V(u,u_1,u_2)} \tag{B.1}$$

wobei $V(u, u_1, u_2)$ die Summe aller Winkelpotentiale, u die Orientierung des Teilchens, u_1, u_2 die Richtung der Bindungen und β die inverse Temperatur ist. Setzt man die Potentiale (2.12) und (2.13) unter Berücksichtigung der relativen Richtungen zu u ein, so erhält man

$$P \propto e^{-\beta\, 2\, c_{Bw}\,(1+u_1\cdot u_2)} \int du\, e^{-\beta\, c_{Bw}\,(1-u\cdot u_1+1+u\cdot u_2)} \tag{B.2}$$

Da die Verteilung nicht von der Orientierung des betrachteten Teilchens abhängen soll, wird über u integiert.

Durch Ausintegrieren der Winkel erhält man

$$P \propto e^{-\beta\, 2\, c_{Bw}\,(1+u_1\cdot u_2)} e^{-2\,\beta\, c_{Bw}} 2\pi \int_{-1}^{1} d\tau\, e^{\beta\, c_{Bw}\,|u_1+u_2|\tau} \tag{B.3}$$

und schließlich

$$P \propto e^{-\beta\, 2\, c_{Bw}\,(1+u_1\cdot u_2)}\, e^{-2\,\beta\, c_{Bw}}\, 2\pi \frac{e^{\beta\, c_{Bw}|u_1+u_2|} - e^{\beta\, c_{Bw}|u_1+u_2|}}{\beta\, c_{Bw}|u_1+u_2|} \tag{B.4}$$

was sich auch schreiben läßt als

$$P \propto e^{-\beta\, 2\, c_{Bw}\,(1+u_1\cdot u_2)}\, e^{-2\,\beta\, c_{Bw}}\, \pi \frac{\sinh \beta\, c_{Bw}|u_1+u_2|}{\beta\, c_{Bw}|u_1+u_2|} \tag{B.5}$$

Unter Verwendung der Gleichung

$$|u_1+u_2| = \sqrt{2(1+u_1\cdot u_2)} \tag{B.6}$$

sowie Hineinziehen aller Konstanten in das $\propto$ und durch die Identifizierung von $u_1 u_2$ mit $|\cos(\theta)|$, wobei θ der von den Bindungen gebildete Winkel ist und der Betrag aus der Symmetrie der Teilchen folgt, wird aus Gleichnung (B.5)

$$P \propto e^{-\beta\, 2\, c_{Bw}\,\left(1+|\cos(\theta)|\right)} \frac{\sinh\left(\beta\, c_{Bw}\sqrt{(2\,(1+|\cos(\theta)|))}\right)}{\beta\, c_{Bw}\sqrt{(1+|\cos(\theta)|)}} \tag{B.7}$$

Anhang C

Programmlisting

Das verwendete Simulationsprogramm setzt sich aus den folgenden drei Programmen zusammen:

- nemo.c

  In diesem Programmteil findet die eigentliche Simulation und die Ermittlung von Meßwerten statt.

- r250.c

  Dies ist der verwendete Zufallszahlengenerator.

- matrix.c

  Dieser Programmteil stellt die Routinen zur Verwaltung von mehrdimensionalen dynamischen Feldern zur verfügung.

```
#define  min(x,y ) ((x)<(y) ? (x):(y))

#define  INT_MAX 2147483647

#define MAX_TEILCHEN 3000
#define MAX_KETTEN_LAENGE 10
#define equi_max 1000  /*Anzahl der Bewegungsversuche pro Teilchen,
                         bevor die Polymere gebaut werden */
#define ww_max 300 /* Anzahl der maximalen WW-Partner pro Teilchen */
#define ketten_ww_max 1000
#define c_1 ((char) 1)
#define c_0 ((char) 0)
#define WW_triangle MAX_TEILCHEN*MAX_TEILCHEN/2
#define Schichtdicke 0.1 /* Dicke der Schichten, die fuer
                            schichtabhangige Profile verwendet wird */
#define Schichtdicke_OP 1 /* Dicke der Schichten, die fuer schichtabhaengige
                             Ordnungsparamterprofile verwendet wird */
#define bw_histo_max 500
#define bl_histo_max 500

#include <stdio.h>
#include <math.h>
#include <sys/types.h>
#include <sys/times.h>
#include <time.h>
#include <malloc.h>
#include "matrix.h"
#include "r250.h"


struct part
{
        int         pn ;
        struct part *next ;
} ;
char   tmp[WW_triangle] ;
struct part ***cell,*list ;
unsigned char   *jpx,*jmx,*jpy,*jmy,*jpz,*jmz ;       /* naechste Nachbarzellen */
char   **sflag ; /* Kommandozeilen-Parameter */
double xi,sigma_s,sigma_l,epsilon_0,kappa ;
       /* Parameter fuer Gay-Berne-Potential fuer ll */
double xi_pp,sigma_s_pp,sigma_l_pp,epsilon_0_pp,kappa_pp ;
       /* Parameter fuer Gay-Berne-Potential fuer pp */
double xi_pl,sigma_s_pl,sigma_l_pl,epsilon_0_pl,kappa_pl ;
       /* Parameter fuer Gay-Berne-Potential fuer pl */
double xi_halbe,sigma_l_halbe,sigma_l_halbe_q,sigma_s_halbe_q ;
double xi_halbe_pp,xi_halbe_pl ;
double sprng,cospr ; /* Staerke der Bondlaengen und Bondwinkelpotentiale */
double bondmin,bondcut ; /* Minimun und cutoff des BL-Potentials */
long   laenge,anzahl_polymere,anzahl_teilchen,anzahl_teilchen_mal2 ;
double temp,press,gesamt_energie ;
double max_jump,max_rotate,max_zerr,max_move ; /* Maximale Sprungweite */
double system_size_x,system_size_y,system_size_z ; /* Groesse der Simulationsbox */
double system_size_x_halbe,system_size_y_halbe ;
long   anzahl ; /* Anzahl der freien Teilchen */
double *pseudo_xpos,*pseudo_ypos,*pseudo_zpos ; /* Positionen der Pseudo-Teilchen */
double *pseudo_xbox,*pseudo_ybox,*pseudo_zbox ; /* dito, aber unterBeruecksichtigung
                                                   der periodischen Randbedingungen */
double *pseudo_xpos_0v,*pseudo_ypos_0v,*pseudo_zpos_0v ;/* Positionen der
                                                          Pseudoteilchen nach Update
                                                          der Verlet-Tabelle */
double diff_max,diff_max2 ; /* groesste zwei Sprungweiten */
double rad_verlet,skin_verlet ; /* Radius bzw. Schichtdicke der Verlet-Kugel */
double skin_verlet_check_update ;
double *xpos,*ypos,*zpos ; /* Positionen der Teilchen */
double *xbox,*ybox,*zbox ; /* dito, aber unter Beruecksichtigung der periodischen
                              Randbedingungen */
double *xdir,*ydir,*zdir ; /* Richtung der Teilchen (normiert auf Laenge 1)
                              relativ zur Position */
double *zpos_test,*zbox_test ;
int    *nachb_anz ; /* Anzahl der Nachbarteilchen entlang einer Polymerkette */
int    *nachb_l,*nachb_r ; /* Nummer der Nachbarteilchen entlang einer
                              Polymerkette : INT_MAX heisst freies Teilchen */
unsigned char   *x_zelle,*y_zelle,*z_zelle ; /* Nummer der Zellen in denen sich ein
                                                Pseudo_Teilchen befindet */
unsigned char   zellen_x,zellen_y,zellen_z ; /* Anzahl der Zellen in x,y,z-Richtung
                                                in die die Simulationsbox aufgeteilt
                                                wird */
double cell_size_x,cell_size_y,cell_size_z ; /* Laenge  der Zellen in x,y,z-Richtung */
double **bond_energie ;
double **bond_energie_test ;
long   mcsmax ;
int    iseed ;
double *e_help ;
int    **tab_verlet,*upper_verlet ;
double **ene_verlet,**ene_help ;
unsigned long *histo_theta,*histo_theta_l,*histo_theta_p ;
unsigned long *histo_phi,*histo_phi_l,*histo_phi_p ;
double cutoff,cutoff_pp,cutoff_pl,cutoff_max ;
int    verlet_update_counter ;
double lj_safety ;
double system_size_z_real,system_size_z_real_halbe ;
int Schichtzahl ; /* Anzahl der Schichten */
int Schichtzahl_halbe ;
double Schichtvolumen ; /* Volumen einer Schicht */
int Schichtzahl_OP ; /* Anzahl der Schichten */
int Schichtzahl_halbe_OP ;
double Schichtvolumen_OP ; /* Volumen einer Schicht */
unsigned long *histo_dichte ; /* enthaelt Dichteprofil fuer die Teilchenmittelpunkte */
double ***histo_S ;
double **Schicht_OP ;  /* enthaelt Profil fuer den Ordnungsparameter */
double *Schicht_LEVx ;
double *Schicht_LEVy ;
double *Schicht_LEVz ;
unsigned long *Schicht_dichte ;
int    alles_rausschreiben,savestep ;
unsigned int *chain_heads ;
int    neugleichalt ;
unsigned long bw_histo[bw_histo_max+1] ;
unsigned long bl_histo[bl_histo_max+1] ;
unsigned long blz_histo[bl_histo_max+1] ;
unsigned long ketten_laenge_z_histo[bl_histo_max+1] ;
unsigned long b_d_histo[bw_histo_max+1] ;


double calculate_gesamt_energie() ;

int move() ;
int rot() ;

double ww_energie(double xbo1,double ybo1,double zbo1,
                  double xdi1,double ydi1,double zdi1,
                  double xbo2,double ybo2,double zbo2,
                  double xdi2,double ydi2,double zdi2,
                  int n_anz1,int n_anz2) ;


/*------------------------------------------------------------*/


void print_cell(CL)
struct part *CL;
{
       for(;CL!=NULL;CL=CL->next)
              printf("%i ",CL->pn);
}

/*------------------------------------------------------------*/


void read_parameters()
```

```
/* Hier werden die Startwerte fier einen Run aus der .ein Datei eingelesen und
   daraus einige Startgroessen bestimmt */

{
  FILE *file_1 ;
  char zeile[120],fname[50] ;

  sprintf(fname,"%s.ein",sflag[1]) ;
  file_1=fopen(fname,"r") ;
  if (file_1 == NULL) printf("ERROR opening %s\n",fname) ;

  fgets(zeile, 120, file_1) ;
  sscanf(zeile,"%ld ",&anzahl) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%ld ",&anzahl_polymere) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%ld ",&laenge) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%ld ",&mcsmax) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&temp) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&press) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&system_size_x) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&system_size_y) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&system_size_z) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&sigma_s) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&sigma_l) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&epsilon_0) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&sigma_s_pp) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&sigma_l_pp) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&epsilon_0_pp) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&sigma_s_pl) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&sigma_l_pl) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&epsilon_0_pl) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&sprng) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&bondmin) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&cospr) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&max_jump) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&max_rotate) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&max_zerr) ;
  fgets(zeile,120,file_1) ;

  fclose(file_1) ;

  bondcut = bondmin * 0.2 ;

  kappa = sigma_l/sigma_s ;
  xi = (kappa*kappa - 1) / (kappa*kappa +1) ;
  xi_halbe = xi / 2. ;

  kappa_pp = sigma_l_pp/sigma_s_pp ;
  xi_pp = (kappa_pp*kappa_pp - 1) / (kappa_pp*kappa_pp +1) ;
  xi_halbe_pp = xi_pp / 2. ;

  kappa_pl = sigma_l_pl/sigma_s_pl ;
  xi_pl = (kappa_pl*kappa_pl - 1) / (kappa_pl*kappa_pl +1) ;
  xi_halbe_pl = xi_pl / 2. ;

  anzahl_teilchen = anzahl+anzahl_polymere*laenge ;
  anzahl_teilchen_mal2 = anzahl_teilchen * 2 ;
  cutoff = sigma_s * 0.12246204831 ; /* 6te Wurzel aus 2  - 1 */
  cutoff_pp = sigma_s_pp *  0.12246204831 ; /* 6te Wurzel aus 2  - 1 */
  cutoff_pl = sigma_s_pl *  0.12246204831 ; /* 6te Wurzel aus 2  - 1 */
  sigma_s_halbe_q = (sigma_s) * (sigma_s) * 0.25;
  sigma_l_halbe = 0.5 * (sigma_l) ;
  sigma_l_halbe_q = sigma_l_halbe * sigma_l_halbe ;

  max_move = ((sigma_l_halbe +cutoff) * 0.5 * max_rotate + 1.72*max_jump) ;
  /* Maximal moegliche Bewegungsweite eines Pseudoteilchens in einem Step */

  if (max_zerr > max_move) {
    max_move = max_zerr ;
  }

  cutoff_max = (sigma_l_halbe +cutoff) ;

  if (skin_verlet < max_move) {
    skin_verlet = max_move * 1.5 ;
  }

  skin_verlet_check_update = skin_verlet - max_move ;

  zellen_x  = floor(system_size_x / (cutoff_max + skin_verlet)) ;
  cell_size_x = system_size_x / zellen_x ;
  zellen_y  = floor(system_size_y / (cutoff_max + skin_verlet)) ;
  cell_size_y = system_size_y / zellen_y ;
  zellen_z  = floor(system_size_z / (cutoff_max + skin_verlet)) ;
  cell_size_z = system_size_z / zellen_z ;
  zellen_z = zellen_z + 3 ;


  system_size_x_halbe = system_size_x /2. ;
  system_size_y_halbe = system_size_y /2. ;
  system_size_z_real = zellen_z * cell_size_z ;
  system_size_z_real_halbe = system_size_z_real /2. ;

}

/*------------------------------------------------------------*/

void read_config_parameters()

/* liest die Temperatur und die Anzahl der MCS aus der .ein Datei falls aus einer
   bestehenden Konfiguration gestartet wird */

{

  FILE *file_1 ;
  char zeile[120],fname[50] ;
  int dummyi ;


  sprintf(fname,"%s.ein",sflag[1]) ;
  file_1=fopen(fname,"r") ;
  if (file_1 == NULL) printf("ERROR opening %s\n",fname) ;


  fgets(zeile, 120, file_1) ;
  sscanf(zeile,"%d ",&dummyi) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%d ",&dummyi) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%d ",&dummyi) ;
```

```
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%ld ",&mcsmax) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&temp) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf ",&press) ;

  fclose(file_1) ;

}


/*-------------------------------------------------------------*/


void read_config(double *pmcs0)

/* liest die Parameter fuer eine Konfiguration aus der _e.cnf Datei ein und berechnet
   einige zum Start wichtige Groessen */

{

  FILE *file_1 ;
  char zeile[120],fname[50] ;

  sprintf(fname,"%s_e.cnf",sflag[1]) ;
  file_1=fopen(fname,"r") ;
  if (file_1 == NULL) printf("ERROR opening %s\n",fname) ;

  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%9ld %9ld %9ld %9ld %9lf %9lf",&anzahl,&anzahl_polymere,&laenge,
         &mcsmax,&temp,&press) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%9lf %9lf %9lf",&system_size_x ,&system_size_y,&system_size_z) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%9lf %9lf %9lf",&sigma_l,&sigma_s,&epsilon_0) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%9lf %9lf %9lf",&sigma_l_pp,&sigma_s_pp,&epsilon_0_pp) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%9lf %9lf %9lf",&sigma_l_pl,&sigma_s_pl,&epsilon_0_pl) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%9lf %9lf %9lf %9lf %9lf %9lf",&sprng,&bondmin,&cospr,&max_jump,
         &max_rotate,&max_zerr) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf",pmcs0) ;

  fclose(file_1) ;

  bondcut = bondmin * 0.2 ;

  kappa = sigma_l/sigma_s ;
  xi = (kappa*kappa - 1) / (kappa*kappa +1) ;
  xi_halbe = xi / 2. ;

  kappa_pp = sigma_l_pp/sigma_s_pp ;
  xi_pp = (kappa_pp*kappa_pp - 1) / (kappa_pp*kappa_pp +1) ;
  xi_halbe_pp = xi_pp / 2. ;

  kappa_pl = sigma_l_pl/sigma_s_pl ;
  xi_pl = (kappa_pl*kappa_pl - 1) / (kappa_pl*kappa_pl +1) ;
  xi_halbe_pl = xi_pl / 2. ;

  anzahl_teilchen = anzahl + anzahl_polymere*laenge ;
  anzahl_teilchen_mal2 = anzahl_teilchen * 2 ;
  cutoff = sigma_s * 0.12246204831 ; /* 6te Wurzel aus 2  - 1 */
  cutoff_pp = sigma_s_pp *  0.12246204831 ; /* 6te Wurzel aus 2  - 1 */
  cutoff_pl = sigma_s_pl *  0.12246204831 ; /* 6te Wurzel aus 2  - 1 */
  sigma_s_halbe_q = (sigma_s) * (sigma_s) * 0.25;
  sigma_l_halbe = 0.5 * (sigma_l) ;
  sigma_l_halbe_q = sigma_l_halbe * sigma_l_halbe ;

  max_move = ((sigma_l_halbe +cutoff) * 0.5 * max_rotate + 1.72*max_jump) ;
  /* Maximal moegliche Bewegungsweite eines Pseudoteilchens in einem Step */

  if (max_zerr > max_move) {
    max_move = max_zerr ;
  }

  cutoff_max = (sigma_l_halbe +cutoff) ;

  if (skin_verlet < max_move) {
    skin_verlet = max_move * 1.5 ;
  }

  skin_verlet_check_update = skin_verlet - max_move ;

  zellen_x  = floor(system_size_x / (cutoff_max + skin_verlet)) ;
  cell_size_x = system_size_x / zellen_x ;
  zellen_y  = floor(system_size_y / (cutoff_max + skin_verlet)) ;
  cell_size_y = system_size_y / zellen_y ;
  zellen_z  = floor(system_size_z / (cutoff_max + skin_verlet)) ;
  cell_size_z = system_size_z / zellen_z ;
  zellen_z = zellen_z + 3 ;


  system_size_x_halbe = system_size_x /2. ;
  system_size_y_halbe = system_size_y /2. ;
  system_size_z_real = zellen_z * cell_size_z ;
  system_size_z_real_halbe = system_size_z_real /2. ;

}

/*-------------------------------------------------------------*/


void write_configuration(int j,double mcs,double mcstotal)

/* schreibt eine Konfiguration heraus , je nach Uebergabeparameter in die _s.cnf,
   _m.cnf oder _e.cnf Datei bzw in die _c?????.cnf Dateien */

{
  int  ii ;
  FILE *file_1 ;
  char fname[50],suffix ;
  int nr ;

  if (alles_rausschreiben == 0) {
    suffix = 's' ;
    if (j == 0) suffix = 's' ;
    if (j == 1) suffix = 'm' ;
    if (j == 2) suffix = 'e' ;
    sprintf(fname,"%s_%c.cnf",sflag[1],suffix) ;
  } else {
    if (j == 2) {
      suffix = 'e' ;
      sprintf(fname,"%s_%c.cnf",sflag[1],suffix) ;
    } else {
      suffix = 'c' ;
      nr = mcs/savestep ;
      sprintf(fname,"%s_%c%05d.cnf",sflag[1],suffix,nr) ;
    }
  }
```

```
  file_1=fopen(fname,"w") ;


  fprintf(file_1,"CONFIGURATION %s CREATED BY nemo_neu.C V07.06.\n",fname) ;
  fprintf(file_1,"------------------------------------------------------------\n") ;
  fprintf(file_1,"Monomere  Polymere  Laenge    mcsmax    temp      press\n") ;
  fprintf(file_1,"%-9ld %-9ld %-9ld %-9ld %-9.3f %-9.3f\n" ,anzahl,anzahl_polymere,
          laenge,mcsmax,temp,press) ;
  fprintf(file_1,"Size_x    Size_y    Size_z\n") ;
  fprintf(file_1,"%-9.3f %-9.3f %-9.3f\n",system_size_x,system_size_y,system_size_z) ;
  fprintf(file_1,"sigma_l   sigma_s   epsilon_0 fuer ll\n") ;
  fprintf(file_1,"%-9.3f %-9.3f %-9.3f\n",sigma_l,sigma_s,epsilon_0) ;
  fprintf(file_1,"sigma_l   sigma_s   epsilon_0 epsilon_l epsilon_s fuer pp\n") ;
  fprintf(file_1,"%-9.3f %-9.3f %-9.3f\n",sigma_l_pp,sigma_s_pp,epsilon_0_pp) ;
  fprintf(file_1,"sigma_l   sigma_s   epsilon_0 epsilon_l epsilon_s fuer pl\n") ;
  fprintf(file_1,"%-9.3f %-9.3f %-9.3f\n",sigma_l_pl,sigma_s_pl,epsilon_0_pl) ;
  fprintf(file_1,"sprng     bondmin   cospr     max_jump  max_rot   max_zerr\n") ;
  fprintf(file_1,"%-9.3f %-9.3f %-9.3f %-9.3f %-9.3f %-9.3f\n",sprng,bondmin,cospr,
          max_jump,max_rotate,max_zerr) ;
  fprintf(file_1,"\n") ;
  fprintf(file_1,"%10.2f : mcs at save time \n",mcs) ;
  fprintf(file_1,"%10.2f : total age of this configuration \n",mcstotal+mcs) ;
  fprintf(file_1,"\n") ;

  for (ii=0 ; ii < anzahl_teilchen ;ii++){
    fprintf(file_1,"%-12.6f %-12.6f %-12.6f %-12.6f %-12.6f %-12.6f\n",xbox[ii],
            ybox[ii],zbox[ii],xdir[ii],ydir[ii],zdir[ii]) ;
    fprintf(file_1,"%d %d %d\n",nachb_anz[ii],nachb_l[ii],nachb_r[ii]) ;
  }

  fclose(file_1) ;
}

/*-------------------------------------------------------------*/

void start_norm()

/* setzt beim Start aus einer .ein Datei die Teilchen auf ein SC-Gitter auf */

{

  int    i,j,k,l ;
  double x,y,z ;
  int    anzahl_x,anzahl_y,anzahl_z ;
  double help ;
  int    counter ;
  int    wert ;

  counter = 0 ;
  help = (double) (anzahl + anzahl_polymere * (laenge -1)) ;
  anzahl_z = (int) ((sigma_s/sigma_l)*(system_size_z/system_size_x) *
                    exp(log(help)/3.)) ;
  help = (anzahl + anzahl_polymere * (laenge-1)) / anzahl_z ;
  anzahl_x = (int) sqrt(help) ;
  anzahl_y = (int) help/anzahl_x ;
  x = system_size_x / (anzahl_x+1) ;
  y = system_size_y / (anzahl_y+1) ;
  z = system_size_z / (anzahl_z+1) ;
  anzahl = anzahl_x*anzahl_y*anzahl_z ;
  anzahl_teilchen = anzahl+anzahl_polymere ;
  anzahl_teilchen_mal2 = anzahl_teilchen * 2 ;

  /* setze Loesungsmittelteilchen auf sc-Positionen */

  for (j=0 ;j < anzahl_x ;j++) {
    for (k=0 ;k < anzahl_y ;k++) {
      for (l=0 ;l < anzahl_z ;l++) {

        xpos[l+k*anzahl_z+j*anzahl_z*anzahl_y] = (j+1)*x ;
        ypos[l+k*anzahl_z+j*anzahl_z*anzahl_y] = (k+1)*y ;
        zpos[l+k*anzahl_z+j*anzahl_z*anzahl_y] = (l+1)*z ;
      }
    }
  }

  if (anzahl_polymere >0 ) {
    help = anzahl_polymere /2. ;
    anzahl_x = (int) sqrt(help) ;
    anzahl_y = (int) (help/anzahl_x) ;
    x = system_size_x / anzahl_x ;
    y = system_size_y / anzahl_y ;


    /* setze Polymerbuerstenkoepfe auf obere und untere Grenzflaeche */

    for (j=0 ;j <anzahl_x ;j++) {
      for (k=0 ;k <anzahl_y ;k++) {

        wert = anzahl +k +j*anzahl_y ;

        xpos[wert] = (j+0.5)*x ;
        ypos[wert] = (k+0.5)*y ;
        zpos[wert] = 0 ;
        nachb_anz[wert] = 3 ; /* eigentlich 1 !!!!!!!!!!!!!!!!!!!!!!!!!!!! */
        chain_heads[counter] = wert ;
        counter ++ ;



        wert = wert + (int) help ;

        xpos[wert] = (j+0.5)*x ;
        ypos[wert] = (k+0.5)*y ;
        zpos[wert] = system_size_z ;
        nachb_anz[wert] = 3 ; /* eigentlich 1 !!!!!!!!!!!!!!!!!!!!!!!!!!!! */
        chain_heads[counter] = wert ;
        counter ++ ;


      }
    }
  }

  /* ACHTUNG !!  Noch keine Unterscheidung zwischen Monomeren und Polymeren !! */

  /* richte Loesungsmittelteilchen in z-Richtung aus */

  for (i=0 ;i < anzahl_teilchen ;i++) {
    xdir[i] = 0 ;
    ydir[i] = 0 ;
    zdir[i] = 1 ;
    xbox[i] = xpos[i] ;
    ybox[i] = ypos[i] ;
    zbox[i] = zpos[i] ;
}

  /* Setup der Pseudoteilchen und Zellen */


  for (i=0 ;i<anzahl_teilchen;i++) {
    pseudo_xpos[2*i] = xpos[i] + 0.5*xdir[i]*sigma_l_halbe ;
    pseudo_xpos[2*i+1] = xpos[i] - 0.5*xdir[i]*sigma_l_halbe ;
    pseudo_ypos[2*i] = ypos[i] + 0.5*ydir[i]*sigma_l_halbe ;
    pseudo_ypos[2*i+1] = ypos[i] - 0.5*ydir[i]*sigma_l_halbe ;
    pseudo_zpos[2*i] = zpos[i] + 0.5*zdir[i]*sigma_l_halbe ;
    pseudo_zpos[2*i+1] = zpos[i] - 0.5*zdir[i]*sigma_l_halbe ;
  }

  for (i=0;i<anzahl_teilchen_mal2;i++) {
    pseudo_xbox[i] = pseudo_xpos[i] ;
```

```
    pseudo_ybox[i] = pseudo_ypos[i] ;
    if (pseudo_zpos[i] >0) {
      pseudo_zbox[i] = pseudo_zpos[i] ;
    } else {
      pseudo_zbox[i] = system_size_z_real + pseudo_zpos[i] ;
    }
  }
}



/*-------------------------------------------------------------*/

void make_polymers()

/* baut nach equi_step Schritten die Polymere aus den einzelnen Teilchen
   zusammmen "Wachsen der Polymerketten" */

{

  int ii,i,j,koepfe,counter ;
  double equi_step ;
  double opt_abstand,abstand_q ;
  int opt_nummer,start_nummer ;
  double dx,dy,dz,distx,disty ;
  double bondmin_q ;

  bondmin_q = bondmin * bondmin ;

  equi_step = equi_max*anzahl_teilchen ;

  /* bewege alle Teilchen als ob sie Monomere waeren */

  for (ii=0;ii<equi_step;ii++) {
      counter = move() ;
      counter = rot() ;
  }

  counter = 0 ;

  /* bastele Polymere zusammen */

  for (koepfe = anzahl_teilchen-anzahl_polymere; koepfe < anzahl_teilchen; koepfe++) {

    if (zpos[koepfe] == 0.) {

      /* betrachet nach oben wachsende Koepfe */

      start_nummer = koepfe ;

      for (i=1; i < laenge;i++) {
        opt_nummer = INT_MAX ;
        opt_abstand = 100. ;

        for (j = 0;j<anzahl_teilchen;j++) {
          if (nachb_anz[j] == 0) {
            if ((zbox[j]) > (zbox[start_nummer]+1)) {

              distx = fabs(xbox[j]-xbox[start_nummer]) ;
              dx    = (double) min (distx,system_size_x-distx) ;
              disty = fabs(ybox[j]-ybox[start_nummer]) ;
              dy    = (double) min (disty,system_size_y-disty) ;
              dz    = fabs(zbox[j]-zbox[start_nummer]) ;

              abstand_q = dx*dx+dy*dy+dz*dz ;

              if (fabs(abstand_q-bondmin_q) < opt_abstand) {
                opt_abstand = fabs(abstand_q - bondmin_q) ;
                opt_nummer = j ;

                printf(" %d %d %lf \n",start_nummer,j,opt_abstand) ;

              }
            }
          }


        }

        if (opt_abstand > bondcut*bondcut) {
          printf("Polymere konnten nicht gebaut werden !\n") ;
          exit(1) ;
        }

        if (nachb_anz[start_nummer] != 3)
          nachb_anz[start_nummer] ++ ;
        nachb_anz[opt_nummer] ++ ;
        nachb_l[opt_nummer] = start_nummer ;
        nachb_r[start_nummer] = opt_nummer ;
        start_nummer = opt_nummer ;
      }

    } else {

      /* betrachte nach unten wachsende Koepfe */

      start_nummer = koepfe ;

      for (i=1; i < laenge;i++) {
        opt_nummer = INT_MAX ;
        opt_abstand = 100. ;

        for (j = 0;j<anzahl_teilchen;j++) {
          if (nachb_anz[j] == 0) {
            if ((zbox[j]) < (zbox[start_nummer]-1)) {

              distx = fabs(xbox[j]-xbox[start_nummer]) ;
              dx    = (double) min (distx,system_size_x-distx) ;
              disty = fabs(ybox[j]-ybox[start_nummer]) ;
              dy    = (double) min (disty,system_size_y-disty) ;
              dz    = fabs(zbox[j]-zbox[start_nummer]) ;

              abstand_q = dx*dx+dy*dy+dz*dz ;

              if (fabs(abstand_q-bondmin_q) < opt_abstand) {
                opt_abstand = fabs(abstand_q - bondmin_q) ;
                opt_nummer = j ;

                printf(" %d %d %lf \n",start_nummer,j,opt_abstand) ;

              }
            }
          }


        }


        if (opt_abstand > bondcut*bondcut) {
          printf("Polymere konnten nicht gebaut werden !\n") ;
          exit(1) ;
        }

        if (nachb_anz[start_nummer] != 3)
          nachb_anz[start_nummer] ++ ;
        nachb_anz[opt_nummer] ++ ;
        nachb_l[opt_nummer] = start_nummer ;
        nachb_r[start_nummer] = opt_nummer ;
        start_nummer = opt_nummer ;
      }
```

```
    }
  }
  anzahl = anzahl - (anzahl_polymere*(laenge-1)) ;

}


/*-------------------------------------------------------------*/



void start_config(double *pmcage)

/* liest eine Startkonfiguration ein und berechnet die Positionen der dazugehoerigen
   Pseudoteilchen */

{

  int ii,i ;
  FILE *file_1 ;
  char zeile[120],fname[50] ;
  int counter ;

  counter = 0 ;

  sprintf(fname,"%s_e.cnf",sflag[1]) ;
  file_1=fopen(fname,"r") ;
  if (file_1 == NULL) printf("ERROR opening %s\n",fname) ;

  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  fgets(zeile,120,file_1) ;
  sscanf(zeile,"%lf",pmcage) ;
  fgets(zeile,120,file_1) ;

  for (ii=0 ; ii < anzahl_teilchen ;ii++){
    fgets(zeile,120,file_1) ;
    sscanf(zeile,"%lf %lf %lf %lf %lf %lf",  &xpos[ii],&ypos[ii],&zpos[ii],
           &xdir[ii],&ydir[ii],&zdir[ii]) ;
    fgets(zeile,120,file_1) ;
    sscanf(zeile,"%d %d %d",&nachb_anz[ii],&nachb_l[ii],&nachb_r[ii]) ;
    if (nachb_anz[ii] == 3) {
      chain_heads[counter] = ii ;
      counter ++ ;
      if (zpos[ii] != 0) {
        zpos[ii] = system_size_z ;
      }
    }
  }

  fclose(file_1) ;

  for (ii=0 ; ii <anzahl_teilchen ;ii++) {
    xbox[ii] = fmod((xpos[ii] +50*system_size_x),system_size_x) ;
    ybox[ii] = fmod((ypos[ii] +50*system_size_y),system_size_y) ;
    zbox[ii] = zpos[ii] ;
  }

  /* Setup der Pseudoteilchen und Zellen */


  for (i=0 ;i<anzahl_teilchen;i++) {
    pseudo_xpos[2*i] = xpos[i] + 0.5*xdir[i]*sigma_l_halbe ;
    pseudo_xpos[2*i+1] = xpos[i] - 0.5*xdir[i]*sigma_l_halbe ;
    pseudo_ypos[2*i] = ypos[i] + 0.5*ydir[i]*sigma_l_halbe ;
    pseudo_ypos[2*i+1] = ypos[i] - 0.5*ydir[i]*sigma_l_halbe ;
    pseudo_zpos[2*i] = zpos[i] + 0.5*zdir[i]*sigma_l_halbe ;
    pseudo_zpos[2*i+1] = zpos[i] - 0.5*zdir[i]*sigma_l_halbe ;
  }

  for (i=0;i<anzahl_teilchen_mal2;i++) {
    pseudo_xbox[i] = fmod((pseudo_xpos[i] +50*system_size_x),system_size_x) ;
    pseudo_ybox[i] = fmod((pseudo_ypos[i] +50*system_size_y),system_size_y) ;
    if (pseudo_zpos[i] >0) {
      pseudo_zbox[i] = pseudo_zpos[i] ;
    } else {
      pseudo_zbox[i] = system_size_z_real + pseudo_zpos[i] ;
    }
  }
}


/*-------------------------------------------------------------*/


void addp2cell(int ip,int ix,int iy,int iz)

/* fuegt einer Zelle ein weiteres Teilchen an */

{
  struct part *temp_nr;

  if (cell[ix][iy][iz]==NULL) {
    cell[ix][iy][iz] = &list[ip] ;
  }
  else {
    temp_nr = cell[ix][iy][iz] ;
    cell[ix][iy][iz] = &list[ip] ;
    cell[ix][iy][iz]->next = temp_nr ;
  }
  /*
    print_cell(cell[ix][iy][iz]) ;
    printf("%d %d %d %d \n",ip,ix,iy,iz) ;
    */
}


/*-------------------------------------------------------------*/


void setup_pointers()

/* sortiert die Pseudoteilchen in die linked-cell-Struktur ein */

{
  int ii,ix,iy,iz;

  for (ii=0;ii<anzahl_teilchen_mal2;ii++) {
    ix = (int) floor(pseudo_xbox[ii]/cell_size_x) ;
    iy = (int) floor(pseudo_ybox[ii]/cell_size_y) ;
    iz = (int) floor(pseudo_zbox[ii]/cell_size_z) ;
    addp2cell(ii,ix,iy,iz) ;
    x_zelle[ii] = ix ;
    y_zelle[ii] = iy ;
    z_zelle[ii] = iz ;
```

```
  }
}


/*-------------------------------------------------------------*/

void update_pointers(int ip,int ix,int iy,int iz)

/* verschiebt ein Pseudoteilchen aus einer Zelle in eine andere Zelle */

{
        int ixold,iyold,izold,iprev ;
        struct part *CL ;

        ixold = x_zelle[ip] ;
        iyold = y_zelle[ip] ;
        izold = z_zelle[ip] ;
        if (cell[ixold][iyold][izold]->pn == ip) {
                cell[ixold][iyold][izold] = cell[ixold][iyold][izold]->next ;
                list[ip].next = cell[ix][iy][iz] ;
                cell[ix][iy][iz] = &list[ip] ;
                x_zelle[ip] = ix ;
                y_zelle[ip] = iy ;
                z_zelle[ip] = iz ;
                return ;
        }
        CL = cell[ixold][iyold][izold]->next ;
        iprev = cell[ixold][iyold][izold]->pn ;
        for (;CL!=NULL;CL=CL->next) {
                if (CL->pn == ip) {
                        list[iprev].next = CL->next ;
                        list[ip].next = cell[ix][iy][iz] ;
                        cell[ix][iy][iz] = &list[ip] ;
                        x_zelle[ip] = ix ;
                        y_zelle[ip] = iy ;
                        z_zelle[ip] = iz ;
                        return ;
                }
                iprev = CL->pn ;
        }

        printf("ERROR in update_pointers\n");
}


/*-------------------------------------------------------------*/


int read_seed()

/* liest einen Startwert fuer den Zufallszahlengenrator ein */

{
  int seed ;
  FILE *file_1 ;
  char zeile[120] ;

  file_1 = fopen("seed.rnd","r") ;
  if (file_1==NULL)
    printf(" > read_seed           : ERROR seed.rnd not found\n") ;
  fgets(zeile, 120, file_1) ;
  sscanf(zeile,"%d ",&seed) ;
  fclose(file_1) ;
  return(seed) ;
}
/*-------------------------------------------------------------*/

void write_seed(int seed)

/* schreibt einen neuen seed fuer den Zufallszahlengenerator heraus */

{
  FILE *file_1 ;

  file_1=fopen("seed.rnd","w") ;
  fprintf(file_1,"%d\n",seed) ;
  fclose(file_1) ;
}

/*-------------------------------------------------------------*/

void memory_allocation()

/* reserviert den fuer die Simulation benoetigten Hauptspeicher und besetzt die
   entsprechenden Variablen vor */

{
  int i,j,k ;

  jmx = (unsigned char *) calloc(3*zellen_x,sizeof(unsigned char)) ;
  jpx = (unsigned char *) calloc(3*zellen_x,sizeof(unsigned char)) ;
  jmy = (unsigned char *) calloc(3*zellen_y,sizeof(unsigned char)) ;
  jpy = (unsigned char *) calloc(3*zellen_y,sizeof(unsigned char)) ;
  jmz = (unsigned char *) calloc(3*zellen_z,sizeof(unsigned char)) ;
  jpz = (unsigned char *) calloc(3*zellen_z,sizeof(unsigned char)) ;
  x_zelle = (unsigned char *) calloc(anzahl_teilchen_mal2,sizeof(unsigned char)) ;
  y_zelle = (unsigned char *) calloc(anzahl_teilchen_mal2,sizeof(unsigned char)) ;
  z_zelle = (unsigned char *) calloc(anzahl_teilchen_mal2,sizeof(unsigned char)) ;
  xpos = (double *) calloc(anzahl_teilchen,sizeof(double)) ;
  ypos = (double *) calloc(anzahl_teilchen,sizeof(double)) ;
  zpos = (double *) calloc(anzahl_teilchen,sizeof(double)) ;
  zpos_test = (double *) calloc(anzahl_teilchen,sizeof(double)) ;
  zbox_test = (double *) calloc(anzahl_teilchen,sizeof(double)) ;
  xbox = (double *) calloc(anzahl_teilchen,sizeof(double)) ;
  ybox = (double *) calloc(anzahl_teilchen,sizeof(double)) ;
  zbox = (double *) calloc(anzahl_teilchen,sizeof(double)) ;
  xdir = (double *) calloc(anzahl_teilchen,sizeof(double)) ;
  ydir = (double *) calloc(anzahl_teilchen,sizeof(double)) ;
  zdir = (double *) calloc(anzahl_teilchen,sizeof(double)) ;
  pseudo_xpos = (double *) calloc(anzahl_teilchen_mal2,sizeof(double)) ;
  pseudo_ypos = (double *) calloc(anzahl_teilchen_mal2,sizeof(double)) ;
  pseudo_zpos = (double *) calloc(anzahl_teilchen_mal2,sizeof(double)) ;
  pseudo_xbox = (double *) calloc(anzahl_teilchen_mal2,sizeof(double)) ;
  pseudo_ybox = (double *) calloc(anzahl_teilchen_mal2,sizeof(double)) ;
  pseudo_zbox = (double *) calloc(anzahl_teilchen_mal2,sizeof(double)) ;
  pseudo_xpos_0v = (double *) calloc(anzahl_teilchen_mal2,sizeof(double)) ;
  pseudo_ypos_0v = (double *) calloc(anzahl_teilchen_mal2,sizeof(double)) ;
  pseudo_zpos_0v = (double *) calloc(anzahl_teilchen_mal2,sizeof(double)) ;
  e_help = (double *) calloc(anzahl_teilchen,sizeof(double)) ;
  nachb_l = (int *) calloc(anzahl_teilchen,sizeof(int)) ;
  nachb_r = (int *) calloc(anzahl_teilchen,sizeof(int)) ;
  nachb_anz = (int *) calloc(anzahl_teilchen,sizeof(int)) ;
  list = (struct part *) calloc(anzahl_teilchen_mal2,sizeof(struct part)) ;
  bond_energie = (double **) matrix_alloc(2,sizeof(double),anzahl_teilchen,3) ;
  bond_energie_test = (double **) matrix_alloc(2,sizeof(double),anzahl_teilchen,3) ;
  ene_verlet = (double **) matrix_alloc(2,sizeof(double),anzahl_teilchen,ww_max) ;
  ene_help = (double **) matrix_alloc(2,sizeof(double),anzahl_teilchen,ww_max) ;
  tab_verlet = (int **) matrix_alloc(2,sizeof(double),anzahl_teilchen,ww_max) ;
  upper_verlet = (int *) calloc(anzahl_teilchen,sizeof(int)) ;
  cell = (struct part ****) matrix_alloc(3,sizeof(struct part),3*zellen_x,3*zellen_y,
                                         3*zellen_z);

  if (anzahl_polymere > 0) {
    chain_heads = (unsigned int *) calloc(anzahl_polymere,sizeof(unsigned int)) ;
  }
  if (jpx == NULL) printf("ALLOCATION ERROR \n") ;
  if (jpy == NULL) printf("ALLOCATION ERROR \n") ;
  if (jpz == NULL) printf("ALLOCATION ERROR \n") ;
  if (jmx == NULL) printf("ALLOCATION ERROR \n") ;
  if (jmy == NULL) printf("ALLOCATION ERROR \n") ;
  if (jmz == NULL) printf("ALLOCATION ERROR \n") ;
  if (x_zelle == NULL) printf("ALLOCATION ERROR \n") ;
```

```
if (y_zelle == NULL) printf("ALLOCATION ERROR \n") ;
if (z_zelle == NULL) printf("ALLOCATION ERROR \n") ;
if (xpos == NULL) printf("ALLOCATION ERROR \n") ;
if (ypos == NULL) printf("ALLOCATION ERROR \n") ;
if (zpos == NULL) printf("ALLOCATION ERROR \n") ;
if (zpos_test == NULL) printf("ALLOCATION ERROR \n") ;
if (zbox_test == NULL) printf("ALLOCATION ERROR \n") ;
if (xbox == NULL) printf("ALLOCATION ERROR \n") ;
if (ybox == NULL) printf("ALLOCATION ERROR \n") ;
if (zbox == NULL) printf("ALLOCATION ERROR \n") ;
if (xdir == NULL) printf("ALLOCATION ERROR \n") ;
if (ydir == NULL) printf("ALLOCATION ERROR \n") ;
if (zdir == NULL) printf("ALLOCATION ERROR \n") ;
if (pseudo_xpos == NULL) printf("ALLOCATION ERROR \n") ;
if (pseudo_ypos == NULL) printf("ALLOCATION ERROR \n") ;
if (pseudo_zpos == NULL) printf("ALLOCATION ERROR \n") ;
if (pseudo_xpos_0v == NULL) printf("ALLOCATION ERROR \n") ;
if (pseudo_ypos_0v == NULL) printf("ALLOCATION ERROR \n") ;
if (pseudo_zpos_0v == NULL) printf("ALLOCATION ERROR \n") ;
if (pseudo_xbox == NULL) printf("ALLOCATION ERROR \n") ;
if (pseudo_ybox == NULL) printf("ALLOCATION ERROR \n") ;
if (pseudo_zbox == NULL) printf("ALLOCATION ERROR \n") ;
if (ene_verlet == NULL) printf("ALLOCATION ERROR \n") ;
if (ene_help == NULL) printf("ALLOCATION ERROR \n") ;
if (e_help == NULL) printf("ALLOCATION ERROR \n") ;
if (nachb_l == NULL) printf("ALLOCATION ERROR \n") ;
if (nachb_r == NULL) printf("ALLOCATION ERROR \n") ;
if (nachb_anz == NULL) printf("ALLOCATION ERROR \n") ;
if (list == NULL) printf("ALLOCATION ERROR \n") ;
if (tab_verlet == NULL) printf("ALLOCATION ERROR \n") ;
if (upper_verlet == NULL) printf("ALLOCATION ERROR \n") ;
if (cell == NULL) printf("ALLOCATION ERROR \n") ;
if (bond_energie == NULL) printf("ALLOCATION ERROR \n") ;
if (bond_energie_test == NULL) printf("ALLOCATION ERROR \n") ;
if (anzahl_polymere > 0) {
  if (chain_heads == NULL) printf("ALLOCATION ERROR \n") ;
}

for (i=0;i<3*zellen_x;i++){
  for (j=0;j<3*zellen_y ;j++){
    for (k=0;k<3*zellen_z;k++){
      cell[i][j][k]=NULL;
    }
  }
}


for (i=0 ;i<anzahl_teilchen ;i++) {
  k = 2*i ;
  bond_energie[i][0] = 0 ;
  bond_energie[i][1] = 0 ;
  bond_energie[i][2] = 0 ;
  bond_energie_test[i][0] = 0 ;
  bond_energie_test[i][1] = 0 ;
  bond_energie_test[i][2] = 0 ;
  list[k].pn= k  ;
  list[k+1].pn = k+1 ;
  list[k].next=NULL;
  list[k+1].next= NULL ;
  xpos[i] = 0. ;
  ypos[i] = 0. ;
  zpos[i] = 0. ;
  zpos_test[i] = 0. ;
  zbox_test[i] = 0. ;
  xbox[i] = 0. ;
  ybox[i] = 0. ;
  zbox[i] = 0. ;
  xdir[i] = 0. ;
  ydir[i] = 0. ;
  zdir[i] = 0. ;
  pseudo_xpos[k] = 0. ;
  pseudo_xpos[k+1] = 0. ;
  pseudo_ypos[k] = 0. ;
  pseudo_ypos[k+1] = 0. ;
  pseudo_zpos[k] = 0. ;
  pseudo_zpos[k+1] = 0. ;
  pseudo_xpos_0v[k] = 0. ;
  pseudo_xpos_0v[k+1] = 0. ;
  pseudo_ypos_0v[k] = 0. ;
  pseudo_ypos_0v[k+1] = 0. ;
  pseudo_zpos_0v[k] = 0. ;
  pseudo_zpos_0v[k+1] = 0. ;
  pseudo_xbox[k] = 0. ;
  pseudo_xbox[k+1] = 0. ;
  pseudo_ybox[k] = 0. ;
  pseudo_ybox[k+1] = 0. ;
  pseudo_zbox[k] = 0. ;
  pseudo_zbox[k+1] = 0. ;
  for (j=0;j<ww_max;j++) {
    tab_verlet[i][j] = 0 ;
    ene_verlet[i][j] = 0. ;
    ene_help[i][j] = 0. ;
  }
  e_help[i] = 0. ;
  x_zelle[i] = 0 ;
  y_zelle[i] = 0 ;
  z_zelle[i] = 0 ;
  nachb_l[i] = INT_MAX ;
  nachb_r[i] = INT_MAX ;
  nachb_anz[i] = 0 ;
}

for (i=0;i<zellen_x;i++){
  jpx[i] = (i+1+3*zellen_x)%zellen_x ;
}
for (i=0;i<zellen_x;i++){
  jmx[i] = (i-1+3*zellen_x)%zellen_x ;
}
for (i=0;i<zellen_y ;i++){
  jpy[i] = (i+1+3*zellen_y)%zellen_y ;
}
for (i=0;i<zellen_y ;i++){
  jmy[i] = (i-1+3*zellen_y)%zellen_y ;
}
for (i=0;i<zellen_z;i++){
  jpz[i] = (i+1+3*zellen_z)%zellen_z ;
}
for (i=0;i<zellen_z;i++){
  jmz[i] = (i-1+3*zellen_z)%zellen_z ;
}

/* ausgelagert aus update_verlet */

for (i=0; i<WW_triangle; i++) {
  tmp[i] = c_0;
}
}

/*------------------------------------------------------------*/

void memory_deallocation()

/* gibt den fuer die Simulation benoetigten Hauptspeicher wieder frei */

{

  free(jpx) ;
  free(jmx) ;
  free(jpy) ;
  free(jmy) ;
  free(jpz) ;
  free(jmz) ;
```

```
  free(x_zelle) ;
  free(y_zelle) ;
  free(z_zelle) ;
  free(xdir) ;
  free(ydir) ;
  free(zdir) ;
  free(zpos_test) ;
  free(zbox_test) ;
  free(xbox) ;
  free(ybox) ;
  free(zbox) ;
  free(xpos) ;
  free(ypos) ;
  free(zpos) ;
  free(pseudo_xpos) ;
  free(pseudo_ypos) ;
  free(pseudo_zpos) ;
  free(pseudo_xbox) ;
  free(pseudo_ybox) ;
  free(pseudo_zbox) ;
  free(pseudo_xpos_0v) ;
  free(pseudo_ypos_0v) ;
  free(pseudo_zpos_0v) ;
  free(list) ;
  mfree(cell) ;
  mfree(tab_verlet) ;
  mfree(ene_verlet) ;
  mfree(ene_help) ;
  free(e_help) ;
  free(nachb_l) ;
  free(nachb_r) ;
  free(nachb_anz) ;
  mfree(bond_energie) ;
  mfree(bond_energie_test) ;
  if (anzahl_polymere >0) {
    free(chain_heads) ;
  }
}

/*-------------------------------------------------------------*/

double GB(double xb1,double yb1,double zb1,double xd1,double yd1,double zd1,
          double xb2,double yb2,double zb2,double xd2,double yd2,double zd2)

/* berechnet die Wechselwirkungsenergie zwischen zwei Loesungsmittelteilchen */

{

  double help ;
  double dx,dy,dz,distx,disty,distz,dist,dist_eff,skp1,skp2,skp1p2q,skp1m2q,skp12 ;
  double rx,ry,rz ;
  double sigma,r12,r6 ;
  double GB_energie ;
  double sigma_q,dist_q ;

  /* berechne den Abstand zweier Teilchen */

  distx = xb2 - xb1 ;
  if (fabs(distx) > system_size_x_halbe) {
    if (distx < 0) dx = distx + system_size_x ;
    else dx = distx - system_size_x ;
  } else dx = distx ;

  disty = yb2 - yb1 ;
  if (fabs(disty) > system_size_y_halbe) {
    if (disty < 0) dy = disty + system_size_y ;
    else dy = disty - system_size_y ;
  } else dy = disty ;

  distz = zb2 - zb1 ;
  if (fabs(distz) > system_size_z_real_halbe) {
    if (distz < 0) dz = distz + system_size_z_real ;
    else dz = distz - system_size_z_real ;
  } else dz = distz ;

  dist_q = dx*dx+dy*dy+dz*dz ;

  rx = dx ;
  ry = dy ;
  rz = dz ;


  /* berechne Skalarprodukte zwischen Richtungsvektoren und Abstandsvektor */

  skp1 = (xd1*rx + yd1*ry + zd1*rz) ;
  skp2 = (xd2*rx + yd2*ry + zd2*rz) ;
  skp12 =(xd1*xd2+yd1*yd2+zd1*zd2) ;
  skp1p2q = (skp1+skp2)*(skp1+skp2) ;
  skp1m2q = (skp1-skp2)*(skp1-skp2) ;

  /* berechene "effektiven" Abstand */

  help = xi*skp12 ;

  sigma_q = (sigma_s*sigma_s) /
    (1. - ((xi_halbe/dist_q) * ((skp1p2q/(1.+help)) + (skp1m2q/(1.-help))))) ;

  /* Neue Abfrage und danach erst Wurzelziehen!!! */


  if (dist_q > (sigma_q*(1+cutoff)*(1+cutoff))) {
    return(0.) ;
  }

  dist = sqrt(dist_q) ;
  sigma = sqrt(sigma_q) ;


  if (dist > sigma+cutoff) {
    /* Teilchen ueberschneiden sich nicht */
    return(0.) ;
  }

  dist_eff = dist - sigma + sigma_s ;


  help = (sigma_s/dist_eff) ;

  help = help * help ;

  r6 = help *help *help ;
  r12 = r6*r6 ;


  /* berechne 'Gay-Berne-SC' Potential */

  GB_energie = 4 * epsilon_0 * (r12-r6+0.25)  ;

  return (GB_energie) ;

}

/*-------------------------------------------------------------*/

double GB_pp(double xb1,double yb1,double zb1,double xd1,double yd1,double zd1,
             double xb2,double yb2,double zb2,double xd2,double yd2,double zd2)

/* berechnet die Wechselwirkungsenergie zwischen zweier Polymerkettenteilchen,
   jedoch OHNE Beruecksichtigung des eventuell zwischen diesen Teilchen
   bestehenden Bonds */

{
```

```
double help ;
double dx,dy,dz,distx,disty,distz,dist,dist_eff,skp1,skp2,skp1p2q,skp1m2q,skp12 ;
double rx,ry,rz ;
double sigma,r12,r6 ;
double GB_energie ;
double dist_q,sigma_q ;

/* berechne den Abstand zweier Teilchen */

distx = xb2 - xb1 ;
if (fabs(distx) > system_size_x_halbe) {
  if (distx < 0) dx = distx + system_size_x ;
  else dx = distx - system_size_x ;
} else dx = distx ;

disty = yb2 - yb1 ;
if (fabs(disty) > system_size_y_halbe) {
  if (disty < 0) dy = disty + system_size_y ;
  else dy = disty - system_size_y ;
} else dy = disty ;

distz = zb2 - zb1 ;
if (fabs(distz) > system_size_z_real_halbe) {
  if (distz < 0) dz = distz + system_size_z_real ;
  else dz = distz - system_size_z_real ;
} else dz = distz ;

 dist_q = dx*dx+dy*dy+dz*dz ;

rx = dx ;
ry = dy ;
rz = dz ;


/* berechne Skalarprodukte zwischen Richtungsvektoren und Abstandsvektor */

skp1 = (xd1*rx + yd1*ry + zd1*rz) ;
skp2 = (xd2*rx + yd2*ry + zd2*rz) ;
skp12 =(xd1*xd2+yd1*yd2+zd1*zd2) ;
skp1p2q = (skp1+skp2)*(skp1+skp2) ;
skp1m2q = (skp1-skp2)*(skp1-skp2) ;

/* berechene "effektiven" Abstand */

help = xi_pp*skp12 ;

sigma_q = (sigma_s_pp*sigma_s_pp) /
  (1. - ((xi_halbe_pp/dist_q)  * ((skp1p2q/(1.+help)) + (skp1m2q/(1.-help))))) ;

/* Neue Abfrage und danach erst Wurzelziehen!!! */

if (dist_q > (sigma_q*(1+cutoff_pp)*(1+cutoff_pp))) {
  return(0.) ;
}

dist = sqrt(dist_q) ;
sigma = sqrt(sigma_q) ;



if (dist > sigma+cutoff_pp) return(0.) ;

dist_eff = dist - sigma + sigma_s_pp ;

help = (sigma_s_pp/dist_eff) ;

help = help * help ;

r6 = help *help *help ;
r12 = r6*r6 ;

  /* berechne Gay-Berne Potential */

  GB_energie = 4 * epsilon_0_pp * (r12-r6+0.25) ;

  return (GB_energie) ;

}


/*-------------------------------------------------------------*/

double GB_pl(double xb1,double yb1,double zb1,double xd1,double yd1,double zd1,
             double xb2,double yb2,double zb2,double xd2,double yd2,double zd2)

/* berechnet die Wechselwirkungsenergie zwischen einem Loesungsmittelteilchen und
   einem Polymerkettenteilchen */

{

  double help ;
  double dx,dy,dz,distx,disty,distz,dist,dist_eff,skp1,skp2,skp1p2q,skp1m2q,skp12 ;
  double rx,ry,rz ;
  double sigma,r12,r6 ;
  double GB_energie ;
  double dist_q,sigma_q ;

  /* berechne den Abstand zweier Teilchen */

  distx = xb2 - xb1 ;
  if (fabs(distx) > system_size_x_halbe) {
    if (distx < 0) dx = distx + system_size_x ;
    else dx = distx - system_size_x ;
  } else dx = distx ;

  disty = yb2 - yb1 ;
  if (fabs(disty) > system_size_y_halbe) {
    if (disty < 0) dy = disty + system_size_y ;
    else dy = disty - system_size_y ;
  } else dy = disty ;

  distz = zb2 - zb1 ;
  if (fabs(distz) > system_size_z_real_halbe) {
    if (distz < 0) dz = distz + system_size_z_real ;
    else dz = distz - system_size_z_real ;
  } else dz = distz ;

  dist_q = dx*dx+dy*dy+dz*dz ;

  rx = dx ;
  ry = dy ;
  rz = dz ;


  /* berechne Skalarprodukte zwischen Richtungsvektoren und Abstandsvektor */

  skp1 = (xd1*rx + yd1*ry + zd1*rz) ;
  skp2 = (xd2*rx + yd2*ry + zd2*rz) ;
  skp12 =(xd1*xd2+yd1*yd2+zd1*zd2) ;
  skp1p2q = (skp1+skp2)*(skp1+skp2) ;
  skp1m2q = (skp1-skp2)*(skp1-skp2) ;

  /* berechene "effektiven" Abstand */

  help = xi_pl*skp12 ;

  sigma_q = (sigma_s_pl*sigma_s_pl) /
    (1. - ((xi_halbe_pl/dist_q)  * ((skp1p2q/(1.+help)) + (skp1m2q/(1.-help))))) ;

  /* Neue Abfrage und danach erst Wurzelziehen!!! */
```

```
  if (dist_q > (sigma_q*(1+cutoff_pl)*(1+cutoff_pl))) {
    return(0.) ;
  }

  dist = sqrt(dist_q) ;
  sigma = sqrt(sigma_q) ;


  if (dist > sigma+cutoff_pl) return(0.) ;

  dist_eff = dist - sigma + sigma_s_pl ;

  help = (sigma_s_pl/dist_eff) ;

  help = help * help ;

  r6 = help *help *help ;
  r12 = r6*r6 ;

  /* berechne Gay-Berne Potential */

  GB_energie = 4 * epsilon_0_pl * (r12-r6+0.25) ;

  return (GB_energie) ;

}

/*------------------------------------------------------------*/

double bond_laengen_potential(double xb1,double yb1,double zb1,
                              double xb2,double yb2,double zb2)

/* berechnet die WW-Energie, die auf Grund des Bondlaengenpotentials zwischen zwei
   Teilchen auf einer Polymerkette wirkt. */

{

  double distx,disty,distz,dx,dy,dz,dist ;
  double bola_energie ;
  double bond_laenge ;

  /* berechne den Abstand zweier Teilchen */

  distx = xb2 - xb1 ;
  if (fabs(distx) > system_size_x_halbe) {
    if (distx < 0) dx = distx + system_size_x ;
    else dx = distx - system_size_x ;
  } else dx = distx ;

  disty = yb2 - yb1 ;
  if (fabs(disty) > system_size_y_halbe) {
    if (disty < 0) dy = disty + system_size_y ;
    else dy = disty - system_size_y ;
  } else dy = disty ;

  distz = zb2 - zb1 ;
  if (fabs(distz) > system_size_z_real_halbe) {
    if (distz < 0) dz = distz + system_size_z_real ;
    else dz = distz - system_size_z_real ;
  } else dz = distz ;

  dist = sqrt(dx*dx+dy*dy+dz*dz) ;

  bond_laenge = dist-bondmin ;

  if (fabs(bond_laenge) > bondcut)
    return(1e100) ;

  bola_energie = (-0.5) * bondcut*bondcut*sprng * log(1-((bond_laenge * bond_laenge) / (
bondcut*bondcut))) ;

  return(bola_energie) ;
}

/*------------------------------------------------------------*/

double bond_winkel_potential(double xb1,double yb1,double zb1,
                             double xd1,double yd1,double zd1,
                             double xb2,double yb2,double zb2,
                             double xd2,double yd2,double zd2,
                             int nachb_anz1,int nachb_anz2)

/* berechnet die WW-Energie, die auf Grund des Bondwinkelpotentials zwischen
   ZWEI Teilchen auf einer Polymerkette wirkt.  */


{

  double distx,disty,distz,dx,dy,dz,dist ;
  double cos_a,cos_b ;
  double bowi_energie ;

  /* berechne den Abstand zweier Teilchen */


  distx = xb2 - xb1 ;
  if (fabs(distx) > system_size_x_halbe) {
    if (distx < 0) dx = distx + system_size_x ;
    else dx = distx - system_size_x ;
  } else dx = distx ;

  disty = yb2 - yb1 ;
  if (fabs(disty) > system_size_y_halbe) {
    if (disty < 0) dy = disty + system_size_y ;
    else dy = disty - system_size_y ;
  } else dy = disty ;

  distz = zb2 - zb1 ;
  if (fabs(distz) > system_size_z_real_halbe) {
    if (distz < 0) dz = distz + system_size_z_real ;
    else dz = distz - system_size_z_real ;
  } else dz = distz ;

  dist = sqrt(dx*dx+dy*dy+dz*dz) ;

  if (nachb_anz1 != 3) {
    cos_a = (xd1*dx + yd1*dy + zd1*dz) / dist ;
  } else cos_a = 1 ;
  if (nachb_anz2 != 3) {
    cos_b = (xd2*dx + yd2*dy + zd2*dz) / (-dist) ;
  } else cos_b = 1 ;

  if (cos_a < 0)
    cos_a = cos_a * -1 ;

  if (cos_b < 0)
    cos_b = cos_b * -1 ;


  bowi_energie = cospr *  ((1 - cos_a) + (1 - cos_b)) ;

  return(bowi_energie) ;

}

/*------------------------------------------------------------*/

double bond_winkel_potential_2(double xb1,double yb1,double zb1,
                               double xb2,double yb2,double zb2,
                               double xb3,double yb3,double zb3)
```

```
/* berechnet die WW-Energie, die auf Grund des Bondwinkelpotentials2 zwischen
   DREI Bonds auf einer Polymerkette wirkt.  */


{

  double distx,disty,distz,dx,dy,dz,dist,dx2,dy2,dz2,dist2 ;
  double cos ;
  double bowi_energie ;

  /* berechne den Abstand zweier Teilchen */


  distx = xb2 - xb1 ;
  if (fabs(distx) > system_size_x_halbe) {
    if (distx < 0) dx = distx + system_size_x ;
    else dx = distx - system_size_x ;
  } else dx = distx ;

  disty = yb2 - yb1 ;
  if (fabs(disty) > system_size_y_halbe) {
    if (disty < 0) dy = disty + system_size_y ;
    else dy = disty - system_size_y ;
  } else dy = disty ;

  distz = zb2 - zb1 ;
  if (fabs(distz) > system_size_z_real_halbe) {
    if (distz < 0) dz = distz + system_size_z_real ;
    else dz = distz - system_size_z_real ;
  } else dz = distz ;

  dist = sqrt(dx*dx+dy*dy+dz*dz) ;

  distx = xb3 - xb1 ;
  if (fabs(distx) > system_size_x_halbe) {
    if (distx < 0) dx2 = distx + system_size_x ;
    else dx2 = distx - system_size_x ;
  } else dx2 = distx ;

  disty = yb3 - yb1 ;
  if (fabs(disty) > system_size_y_halbe) {
    if (disty < 0) dy2 = disty + system_size_y ;
    else dy2 = disty - system_size_y ;
  } else dy2 = disty ;

  distz = zb3 - zb1 ;
  if (fabs(distz) > system_size_z_real_halbe) {
    if (distz < 0) dz2 = distz + system_size_z_real ;
    else dz2 = distz - system_size_z_real ;
  } else dz2 = distz ;

  dist2 = sqrt(dx2*dx2+dy2*dy2+dz2*dz2) ;

  cos = (dx*dx2 + dy*dy2 + dz*dz2) / (dist*dist2) ;


  bowi_energie = cospr * 2 * (1+cos) ;

  return(bowi_energie) ;

}

/*-------------------------------------------------------------*/
/*-------------------------------------------------------------*/
/*   M   E   S   S   R   O   U   T   I   N   E   N             */
/*-------------------------------------------------------------*/
/*-------------------------------------------------------------*/

void write_msr_header(FILE *f)

/* schreibt den  Kopfteil jedes Messfiles */

{
  fprintf(f,"DATA FOR %s CREATED BY nemo_neu.c V07.06. \n",sflag[1]) ;
  fprintf(f,"-----------------------------------------------------------\n") ;
  fprintf(f,"Monomere  Polymere  Laenge    mcsmax    temp     press\n") ;
  fprintf(f,"%-9ld %-9ld %-9ld %-9ld %-9.3f %-9.3f \n" ,anzahl,anzahl_polymere,
          laenge,mcsmax,temp,press) ;
  fprintf(f,"Size_x    Size_y    Size_z\n") ;
  fprintf(f,"%-9.3f %-9.3f %-9.3f\n",system_size_x,system_size_y,system_size_z) ;
  fprintf(f,"sigma_l   sigma_s   epsilon_0\n") ;
  fprintf(f,"%-9.3f %-9.3f %-9.3f\n",sigma_l,sigma_s,epsilon_0) ;
  fprintf(f,"sigma_l   sigma_s   epsilon_0 fuer pp\n") ;
  fprintf(f,"%-9.3f %-9.3f %-9.3f\n",sigma_l_pp,sigma_s_pp,epsilon_0_pp) ;
  fprintf(f,"sigma_l   sigma_s   epsilon_0 fuer pl\n") ;
  fprintf(f,"%-9.3f %-9.3f %-9.3f\n",sigma_l_pl,sigma_s_pl,epsilon_0_pl) ;
  fprintf(f,"sprng     bondmin   cospr     max_jump  max_rot   max_zerr\n") ;
  fprintf(f,"%-9.3f %-9.3f %-9.3f %-9.3f %-9.3f %-9.3f\n",sprng,bondmin,cospr,
          max_jump,max_rotate,max_zerr) ;
  fprintf(f,"\n") ;
  fprintf(f,"\n") ;
}

/*-------------------------------------------------------------*/

void memory_allocation_mess()

/* reserviert den Speicher, den die Messroutinen verwenden und besetzt die
   entsprechenden Variablen vor */

{
  int i,j,k ;


  Schichtzahl = (int) ((0.95*system_size_z) / Schichtdicke) ;
  Schichtzahl_halbe = (int) (Schichtzahl / 2) ;
  Schichtzahl = 1 + 2*Schichtzahl_halbe ;
  Schichtvolumen = Schichtdicke*system_size_x*system_size_y ;

  Schichtzahl_OP = (int) ((0.95*system_size_z) / Schichtdicke_OP) ;
  Schichtzahl_halbe_OP = (int) (Schichtzahl_OP / 2) ;
  Schichtzahl_OP = 1 + 2*Schichtzahl_halbe_OP ;
  Schichtvolumen_OP = Schichtdicke_OP*system_size_x*system_size_y ;


  histo_theta = (unsigned long *) calloc(101,sizeof(unsigned long)) ;
  histo_theta_l = (unsigned long *) calloc(101,sizeof(unsigned long)) ;
  histo_theta_p = (unsigned long *) calloc(101,sizeof(unsigned long)) ;
  histo_phi = (unsigned long *) calloc(201,sizeof(unsigned long)) ;
  histo_phi_l = (unsigned long *) calloc(201,sizeof(unsigned long)) ;
  histo_phi_p = (unsigned long *) calloc(201,sizeof(unsigned long)) ;
  histo_dichte = (unsigned long *) calloc(Schichtzahl,sizeof(unsigned long)) ;
  Schicht_dichte = (unsigned long *) calloc(Schichtzahl_OP,sizeof(unsigned long)) ;
  histo_S = (double ***) matrix_alloc(3,sizeof(double),Schichtzahl_OP,3,3) ;
  Schicht_OP = (double **) matrix_alloc(2,sizeof(double),Schichtzahl_OP,3) ;

  Schicht_LEVx = (double *) calloc(Schichtzahl_OP,sizeof(double)) ;
  Schicht_LEVy = (double *) calloc(Schichtzahl_OP,sizeof(double)) ;
  Schicht_LEVz = (double *) calloc(Schichtzahl_OP,sizeof(double)) ;


  if (histo_theta == NULL) printf("ALLOCATION ERROR in mess \n") ;
  if (histo_theta_l == NULL) printf("ALLOCATION ERROR in mess \n") ;
  if (histo_theta_p == NULL) printf("ALLOCATION ERROR in mess \n") ;
  if (histo_phi == NULL) printf("ALLOCATION ERROR in mess \n") ;
  if (histo_phi_l == NULL) printf("ALLOCATION ERROR in mess \n") ;
  if (histo_phi_p == NULL) printf("ALLOCATION ERROR in mess \n") ;
  if (histo_dichte == NULL) printf("ALLOCATION ERROR in mess \n") ;
  if (Schicht_dichte == NULL) printf("ALLOCATION ERROR in mess \n") ;
  if (histo_S == NULL) printf("ALLOCATION ERROR in mess \n") ;
  if (Schicht_OP == NULL) printf("ALLOCATION ERROR in mess \n") ;
  if (Schicht_LEVx == NULL) printf("ALLOCATION ERROR in mess \n") ;
```

```
  if (Schicht_LEVy == NULL) printf("ALLOCATION ERROR in mess \n") ;
  if (Schicht_LEVz == NULL) printf("ALLOCATION ERROR in mess \n") ;



  for(i=0;i<101;i++) {
    histo_theta[i] = 0 ;
    histo_theta_l[i] = 0 ;
    histo_theta_p[i] = 0 ;
    histo_phi[i] = 0 ;
    histo_phi_l[i] = 0 ;
    histo_phi_p[i] = 0 ;
    histo_phi[i+100] = 0 ;
    histo_phi_l[i+100] = 0 ;
    histo_phi_p[i+100] = 0 ;
  }



  for (i=0;i<Schichtzahl;i++) {
    histo_dichte[i] = 0 ;
  }

  for (i=0;i<Schichtzahl_OP;i++) {
    Schicht_dichte[i] = 0 ;
    Schicht_LEVx[i] = 0. ;
    Schicht_LEVy[i] = 0. ;
    Schicht_LEVz[i] = 0. ;
    for (j=0;j<3;j++) {
      for (k=0;k<3;k++) {
        histo_S[i][j][k] = 0. ;
      }
      Schicht_OP[i][j] = 0. ;
    }
  }

  for (i=0;i<bw_histo_max;i++) {
    bw_histo[i] = 0 ;
    b_d_histo[i] = 0 ;
  }

  for (i=0;i<bl_histo_max;i++) {
    bl_histo[i] = 0 ;
    blz_histo[i] = 0 ;
    ketten_laenge_z_histo[i] = 0 ;
  }

}

/*-------------------------------------------------------------*/

void memory_deallocation_mess()

/* gibt den fuer die Messroutinen benoetigten Speicher wieder frei */

{
  free(histo_theta) ;
  free(histo_theta_l) ;
  free(histo_theta_p) ;
  free(histo_phi) ;
  free(histo_phi_l) ;
  free(histo_phi_p) ;
  free(histo_dichte) ;
  free(Schicht_dichte) ;
  mfree(histo_S) ;
  mfree(Schicht_OP) ;
  free(Schicht_LEVx) ;
  free(Schicht_LEVy) ;
  free(Schicht_LEVz) ;
}
```

```
/*-------------------------------------------------------------*/

void setup_mess()
{
/* legt alle Ausgabe-Files fuer die Messroutinen an */

  char fname[50] ;

  FILE *f1,*f2,*f3,*f4,*f5,*f6,*f7,*f8,*f9,*f10,*f11 ;

  sprintf(fname,"%s_ene.dat",sflag[1]);
  f1 = fopen(fname,"w");

  sprintf(fname,"%s_the.dat",sflag[1]);
  f2 = fopen(fname,"w");

  sprintf(fname,"%s_akz.dat",sflag[1]);
  f3 = fopen(fname,"w");

  sprintf(fname,"%s_phi.dat",sflag[1]);
  f4 = fopen(fname,"w");

  sprintf(fname,"%s_bol.dat",sflag[1]);
  f5 = fopen(fname,"w");

  sprintf(fname,"%s_bow.dat",sflag[1]);
  f6 = fopen(fname,"w");

  sprintf(fname,"%s_kor.dat",sflag[1]);
  f7 = fopen(fname,"w");

  sprintf(fname,"%s_dru.dat",sflag[1]);
  f8 = fopen(fname,"w");

  sprintf(fname,"%s_pol.dat",sflag[1]);
  f9 = fopen(fname,"w");

  sprintf(fname,"%s_sch.dat",sflag[1]);
  f10 = fopen(fname,"w");

  sprintf(fname,"%s_sop.dat",sflag[1]);
  f11 = fopen(fname,"w");

  write_msr_header(f1) ;
  fclose(f1) ;

  write_msr_header(f2) ;
  fclose(f2) ;

  write_msr_header(f3) ;
  fclose(f3) ;

  write_msr_header(f4) ;
  fclose(f4) ;

  write_msr_header(f5) ;
  fclose(f5) ;

  write_msr_header(f6) ;
  fclose(f6) ;

  write_msr_header(f7) ;
  fclose(f7) ;

  write_msr_header(f8) ;
  fclose(f8) ;

  write_msr_header(f9) ;
  fclose(f9) ;

  write_msr_header(f10) ;
```

```
  fclose(f10) ;

  write_msr_header(f11) ;
  fclose(f11) ;
}

/*-------------------------------------------------------------*/

void msr_schicht_dichte()

/* berechnet den Ordnungsparameter und die Teilchendichte fuer jede Schicht der
   Simulationsbox */

{
  int i,j,k,l,m ; /* Schleifenzaehler */
  double abstand ; /* Abstand des Teilchens von der nahesten Wand */
  double system_size_z_halbe ;
  double Mittelschicht_volumen ;
  double wieweit ;
  double xd,yd,zd ;
  int bin ; /* Nummer der Schicht */
  int mittel_counter ;


  double help ;
  double a[3][3],v[3][3] ;
  double b[3],z[3],d[3] ;
  int ip,iq,nrot ;
  double tresh,theta,tau,t,sm,s,h,g,c ;


  for (i=0;i<Schichtzahl_OP;i++) {
    Schicht_dichte[i] = 0 ;
    Schicht_LEVx[i] = 0. ;
    Schicht_LEVy[i] = 0. ;
    Schicht_LEVz[i] = 0. ;
    for (j=0;j<3;j++) {
      for (k=0;k<3;k++) {
        histo_S[i][j][k] = 0. ;
      }
      Schicht_OP[i][j] = 0. ;
    }
  }
  /* Alle Messfelder mit Null vorbesetzt */

  system_size_z_halbe = system_size_z / 2. ;
  Mittelschicht_volumen = (system_size_z - Schichtzahl*Schichtdicke)
    * system_size_x * system_size_y ;


  /*erst mal die Schichtabhaengigen Dichten bestimmen */

  wieweit = Schichtzahl_halbe * Schichtdicke ;

  mittel_counter = 0 ;

  for (i=0;i<anzahl_teilchen;i++) {

    if (nachb_anz[i] == 0) {
      /* Beruecksichtige nur freie Teilchen */

      if (zpos[i] < system_size_z_halbe) {
        abstand = zpos[i] ;
        /* zaehle ab unterer Wand */

        if (abstand < wieweit) {

          bin = (int) (abstand / Schichtdicke) ;
          histo_dichte[bin] ++ ;
        } else {
          mittel_counter ++ ;
        }

      } else {

        abstand = system_size_z - zpos[i] ;
        /* zaehle ab oberer Wand */

        if (abstand < wieweit) {

          bin = (Schichtzahl -1) - (int)(abstand / Schichtdicke) ;
          histo_dichte[bin] ++ ;
        } else {
          mittel_counter ++ ;
        }

      }
    }
    histo_dichte[Schichtzahl_halbe] += (int)(mittel_counter*(Schichtvolumen/
                                                    Mittelschicht_volumen)) ;
    /* Normierung der Mittelschicht */


}
wieweit = Schichtzahl_halbe_OP * Schichtdicke_OP ;


for (i=0;i<anzahl_teilchen;i++) {
  /* Schleife ueber alle Teilchen */

  if (nachb_anz[i] == 0) {
    /* Beruecksichtige nur freie Teilchen */

    if (zpos[i] < system_size_z_halbe) {
      abstand = zpos[i] ;
      /* zaehle ab unterer Wand */

      xd = xdir[i] ;
      yd = ydir[i] ;
      zd = zdir[i] ;
      /* Ausrichtung des Teilchens */

      if (abstand < wieweit) {

        bin = (int) (abstand / Schichtdicke_OP) ;
        Schicht_dichte[bin] ++ ;
        histo_S[bin][0][0] += 3*xd*xd -1 ;
        histo_S[bin][0][1] += 3*xd*yd ;
        histo_S[bin][0][2] += 3*xd*zd ;
        histo_S[bin][1][1] += 3*yd*yd -1 ;
        histo_S[bin][1][2] += 3*yd*zd ;
        histo_S[bin][2][2] += 3*zd*zd -1 ;

      } else {
        Schicht_dichte[Schichtzahl_halbe_OP] ++ ;
        histo_S[Schichtzahl_halbe_OP][0][0] += 3*xd*xd -1 ;
        histo_S[Schichtzahl_halbe_OP][0][1] += 3*xd*yd ;
        histo_S[Schichtzahl_halbe_OP][0][2] += 3*xd*zd ;
        histo_S[Schichtzahl_halbe_OP][1][1] += 3*yd*yd -1 ;
        histo_S[Schichtzahl_halbe_OP][1][2] += 3*yd*zd ;
        histo_S[Schichtzahl_halbe_OP][2][2] += 3*zd*zd -1 ;
        }
      /* Und hier wird das Teilchen einsortiert  */

    } else {
      abstand = system_size_z - zpos[i] ;
      /* zaehle ab oberer Wand */

      xd = xdir[i] ;
      yd = ydir[i] ;
      zd = zdir[i] ;
      /* Ausrichtung des Teilchens */
```

```
      if (abstand < wieweit) {
        bin = (Schichtzahl_OP -1) - (int)(abstand / Schichtdicke_OP) ;
        Schicht_dichte[bin] ++ ;
        histo_S[bin][0][0] += 3*xd*xd -1 ;
        histo_S[bin][0][1] += 3*xd*yd ;
        histo_S[bin][0][2] += 3*xd*zd ;
        histo_S[bin][1][1] += 3*yd*yd -1 ;
        histo_S[bin][1][2] += 3*yd*zd ;
        histo_S[bin][2][2] += 3*zd*zd -1 ;

      } else {
        Schicht_dichte[Schichtzahl_halbe_OP] ++ ;
        histo_S[Schichtzahl_halbe_OP][0][0] += 3*xd*xd -1 ;
        histo_S[Schichtzahl_halbe_OP][0][1] += 3*xd*yd ;
        histo_S[Schichtzahl_halbe_OP][0][2] += 3*xd*zd ;
        histo_S[Schichtzahl_halbe_OP][1][1] += 3*yd*yd -1 ;
        histo_S[Schichtzahl_halbe_OP][1][2] += 3*yd*zd ;
        histo_S[Schichtzahl_halbe_OP][2][2] += 3*zd*zd -1 ;
      }
      /* Und hier wird das Teilchen einsortiert  */

    }
  }
}

/* An dieser Stelle wird die jeweils folgende Schicht der aktuellen hinzugefuegt,
   so dass ueber jeweils 2 Schichten gemittelt wird. */
for (l=0;l<Schichtzahl_halbe_OP;l++) {
  m = l + 1 ;
  histo_S[l][0][0] += histo_S[m][0][0] ;
  histo_S[l][0][1] += histo_S[m][0][1] ;
  histo_S[l][0][2] += histo_S[m][0][2] ;
  histo_S[l][1][1] += histo_S[m][1][1] ;
  histo_S[l][1][2] += histo_S[m][1][2] ;
  histo_S[l][2][2] += histo_S[m][2][2] ;
  Schicht_dichte[l] += Schicht_dichte[m] ;
}
for (l=Schichtzahl_OP-1;l>Schichtzahl_halbe_OP;l--) {
  m = l - 1 ;
  histo_S[l][0][0] += histo_S[m][0][0] ;
  histo_S[l][0][1] += histo_S[m][0][1] ;
  histo_S[l][0][2] += histo_S[m][0][2] ;
  histo_S[l][1][1] += histo_S[m][1][1] ;
  histo_S[l][1][2] += histo_S[m][1][2] ;
  histo_S[l][2][2] += histo_S[m][2][2] ;
  Schicht_dichte[l] += Schicht_dichte[m] ;
}


/* Und jetzt brauchen wir noch den Ordnungsparameter... */

for (l=0;l< Schichtzahl_OP;l++) {
  if (Schicht_dichte[l] != 0) {
    histo_S[l][0][0] = histo_S[l][0][0] / (2*Schicht_dichte[l]) ;
    histo_S[l][1][0] = histo_S[l][0][1] / (2*Schicht_dichte[l]) ;
    histo_S[l][2][0] = histo_S[l][0][2] / (2*Schicht_dichte[l]) ;
    histo_S[l][0][1] = histo_S[l][0][1] / (2*Schicht_dichte[l]) ;
    histo_S[l][1][1] = histo_S[l][1][1] / (2*Schicht_dichte[l]) ;
    histo_S[l][2][1] = histo_S[l][1][2] / (2*Schicht_dichte[l]) ;
    histo_S[l][0][2] = histo_S[l][0][2] / (2*Schicht_dichte[l]) ;
    histo_S[l][1][2] = histo_S[l][1][2] / (2*Schicht_dichte[l]) ;
    histo_S[l][2][2] = histo_S[l][2][2] / (2*Schicht_dichte[l]) ;



#define ROTATE(a,i,j,k,l)
      g=a[i][j];h=a[k][l];a[i][j]=g-s*(h+g*tau);a[k][l]=h+s*(g-h*tau);

      for (ip=0;ip<3;ip++) {
        for (iq=0;iq<3;iq++) {
          v[ip][iq] = 0.0 ;
          a[ip][iq] = histo_S[l][ip][iq] ;
        }
        v[ip][ip] = 1.0 ;
        b[ip] = a[ip][ip] ;
        d[ip] = a[ip][ip] ;
        z[ip] = 0. ;
      }

      nrot = 0 ;

      for (i=0;i<50;i++) {

        sm = fabs(a[0][1]) + fabs(a[0][2]) + fabs(a[1][2]) ;

        if (sm == 0.0) break ;

        if (i < 4) {
          tresh = 0.2 * sm / 9. ;
        } else {
          tresh = 0.0 ;
        }

        for (ip=0;ip<2;ip++) {
          for (iq=ip+1;iq<3;iq++) {
            g=100.0*fabs(a[ip][iq]) ;

            if(i>4 && (double)fabs(d[ip]+g) == (double)fabs(d[ip]) &&
               (double)fabs(d[iq]+g) == (double)fabs(d[iq]))
              a[ip][iq] = 0 ;
            else if (fabs(a[ip][iq]) > tresh) {
              h = d[iq]-d[ip] ;
              if ((double)(fabs(h)+g) == (double) fabs(h))
                t=(a[ip][iq])/h ;
              else {
                theta = 0.5*h/(a[ip][iq]) ;
                t = 1.0 / (fabs(theta) + sqrt(1.0+theta*theta)) ;
                if (theta <0.0) t = -t ;
              }

              c = 1.0 / sqrt(1+t*t) ;
              s = t*c ;
              tau = s/(1.0+c) ;
              h = t*a[ip][iq] ;
              z[ip] -= h ;
              z[iq] += h ;
              d[ip] -= h ;
              d[iq] += h ;
              a[ip][iq] = 0.0 ;
              for(j=0;j<=ip-1;j++) {
                ROTATE(a,j,ip,j,iq) ;
              }
              for(j=ip+1;j<=iq-1;j++) {
                ROTATE(a,ip,j,j,iq) ;
              }
              for(j=iq+1;j<3;j++) {
                ROTATE(a,ip,j,iq,j) ;
              }
              for(j=0;j<3;j++) {
                ROTATE(v,j,ip,j,iq) ;
              }

              nrot ++ ;

            }
          }
        }

        for (ip=0;ip<3;ip++) {
          b[ip] += z[ip] ;
          d[ip] = b[ip] ;
          z[ip] = 0.0 ;
```

```
      }
    }

    /* Bestimmung des groessten Eigenwertes und des zugehoerigen EV's */

    if (d[0] > d[1]) {
      Schicht_OP[l][0] = d[0] ;
      help = d[1] ;
      Schicht_LEVx[l] = v[0][0] ;
      Schicht_LEVy[l] = v[0][1] ;
      Schicht_LEVz[l] = v[0][2] ;
    } else {
      Schicht_OP[l][0] = d[1] ;
      help = d[0] ;
      Schicht_LEVx[l] = v[1][0] ;
      Schicht_LEVy[l] = v[1][1] ;
      Schicht_LEVz[l] = v[1][2] ;
    }

    if (Schicht_OP[l][0] < d[2]) {
      Schicht_OP[l][1] = Schicht_OP[l][0] ;
      Schicht_OP[l][0] = d[2] ;
      Schicht_OP[l][2] = help ;
      Schicht_LEVx[l] = v[2][0] ;
      Schicht_LEVy[l] = v[2][1] ;
      Schicht_LEVz[l] = v[2][2] ;

    } else {
      if (help < d[2]) {
        Schicht_OP[l][2] = help ;
        Schicht_OP[l][1] = d[2] ;
      } else {
        Schicht_OP[l][2] = d[2] ;
        Schicht_OP[l][1] = help ;
      }
    }



  }
 }
}

/*-------------------------------------------------------------*/

double msr_energie()

/* berechnet die Energie pro Teilchen */

{
 gesamt_energie = calculate_gesamt_energie() ;
 return (gesamt_energie/anzahl_teilchen) ;
}


/*-------------------------------------------------------------*/

void msr_S(double *O1,double *O2,double *O3,double *S11,double *S12,double *S13,
           double *S22,double *S23,double *S33,double *LEVx,double *LEVy,double *LEVz)

/* berechnet den nematischen Ordnungsparameter */

{
 double help,tausch ;
 double xd,yd,zd ;
 int i,j ;
 double S[3][3],anzahl_malzwei ;
 double a[3][3],v[3][3] ;
 double b[3],z[3],d[3] ;
 int ip,iq,nrot ;
 double tresh,theta,tau,t,sm,s,h,g,c ;

 anzahl_malzwei = 2 * anzahl ;

 for (i=0;i<3;i++) {
   for (j=0;j<3;j++) {
     S[i][j] = 0 ;
   }
 }

 for (i=0;i<anzahl_teilchen;i++) {
   if (nachb_anz[i] == 0) {
     xd = xdir[i] ;
     yd = ydir[i] ;
     zd = zdir[i] ;
     S[0][0] += 3*xd*xd -1 ;
     S[0][1] += 3*xd*yd ;
     S[0][2] += 3*xd*zd ;
     S[1][1] += 3*yd*yd -1 ;
     S[1][2] += 3*yd*zd ;
     S[2][2] += 3*zd*zd -1 ;
   }
 }
 S[0][0] = S[0][0] / anzahl_malzwei ;
 S[1][0] = S[0][1] / anzahl_malzwei ;
 S[2][0] = S[0][2] / anzahl_malzwei ;
 S[0][1] = S[0][1] / anzahl_malzwei ;
 S[1][1] = S[1][1] / anzahl_malzwei ;
 S[2][1] = S[1][2] / anzahl_malzwei ;
 S[0][2] = S[0][2] / anzahl_malzwei ;
 S[1][2] = S[1][2] / anzahl_malzwei ;
 S[2][2] = S[2][2] / anzahl_malzwei ;

 *S11 = S[0][0] ;
 *S12 = S[0][1] ;
 *S13 = S[0][2] ;
 *S22 = S[1][1] ;
 *S23 = S[1][2] ;
 *S33 = S[2][2] ;

 /* Und jetzt bringen wir diesen Tensor auf Diagonalform */


#define ROTATE(a,i,j,k,l)
 g=a[i][j];h=a[k][l];a[i][j]=g-s*(h+g*tau);a[k][l]=h+s*(g-h*tau);

 for (ip=0;ip<3;ip++) {
   for (iq=0;iq<3;iq++) {
     v[ip][iq] = 0.0 ;
     a[ip][iq] = S[ip][iq] ;
     }
   v[ip][ip] = 1.0 ;
   b[ip] = a[ip][ip] ;
   d[ip] = a[ip][ip] ;
   z[ip] = 0. ;
 }

 nrot = 0 ;

 for (i=0;i<50;i++) {

   sm = fabs(a[0][1]) + fabs(a[0][2]) + fabs(a[1][2]) ;

   if (sm == 0.0) break ;

   if (i < 4) {
     tresh = 0.2 * sm / 9. ;
     } else {
       tresh = 0.0 ;
     }
```

```
    for (ip=0;ip<2;ip++) {
      for (iq=ip+1;iq<3;iq++) {
        g=100.0*fabs(a[ip][iq]) ;

        if(i>4 && (double)fabs(d[ip]+g) == (double)fabs(d[ip]) &&
           (double)fabs(d[iq]+g) == (double)fabs(d[iq]))
          a[ip][iq] = 0 ;
        else if (fabs(a[ip][iq]) > tresh) {
          h = d[iq]-d[ip] ;
          if ((double)(fabs(h)+g) == (double) fabs(h))
            t=(a[ip][iq])/h ;
          else {
              theta = 0.5*h/(a[ip][iq]) ;
              t = 1.0 / (fabs(theta) + sqrt(1.0+theta*theta)) ;
              if (theta <0.0) t = -t ;
          }

          c = 1.0 / sqrt(1+t*t) ;
          s = t*c ;
          tau = s/(1.0+c) ;
          h = t*a[ip][iq] ;
          z[ip] -= h ;
          z[iq] += h ;
          d[ip] -= h ;
          d[iq] += h ;
          a[ip][iq] = 0.0 ;
          for(j=0;j<=ip-1;j++) {
            ROTATE(a,j,ip,j,iq) ;
          }
          for(j=ip+1;j<=iq-1;j++) {
            ROTATE(a,ip,j,j,iq) ;
          }
          for(j=iq+1;j<3;j++) {
            ROTATE(a,ip,j,iq,j) ;
          }
          for(j=0;j<3;j++) {
            ROTATE(v,j,ip,j,iq) ;
          }

          nrot ++ ;

        }
      }
    }

    for (ip=0;ip<3;ip++) {
      b[ip] += z[ip] ;
      d[ip] = b[ip] ;
      z[ip] = 0.0 ;
    }
  }

  if (d[0] > d[1]) {
    *O1 = d[0] ;
    help = d[1] ;
    *LEVx = v[0][0] ;
    *LEVy = v[0][1] ;
    *LEVz = v[0][2] ;
} else {
    *O1 = d[1] ;
    help = d[0] ;
    *LEVx = v[1][0] ;
    *LEVy = v[1][1] ;
    *LEVz = v[1][2] ;
}

  if (*O1 < d[2]) {
    *O2 = *O1 ;
    *O1 = d[2] ;
    *O3 = help ;
    *LEVx = v[2][0] ;
    *LEVy = v[2][1] ;
    *LEVz = v[2][2] ;

  } else {
    if (help < d[2]) {
      *O3 = help ;
      *O2 = d[2] ;
    } else {
      *O3 = d[2] ;
      *O2 = help ;
    }
  }

}


/*------------------------------------------------------------*/

double msr_sigma_bondlaenge()

{
  double av_bondlaenge ;
  double dx,dy,dz,disty,distx ;
  int i ;
  int anz_bonds ;
  double q,zahl ;

  anz_bonds = 0 ;
  av_bondlaenge = 0. ;
  q = 0. ;

  for (i = 0; i< anzahl_teilchen;i++) {
    if (nachb_anz[i] != 0) {
      if (nachb_l[i] != INT_MAX) {
        anz_bonds ++ ;

        distx = xbox[i] - xbox[nachb_l[i]] ;
        if (fabs(distx) > system_size_x_halbe) {
          if (distx < 0) dx = distx + system_size_x ;
          else dx = distx - system_size_x ;
        } else dx = distx ;

        disty = ybox[i] - ybox[nachb_l[i]] ;
        if (fabs(disty) > system_size_y_halbe) {
          if (disty < 0) dy = disty + system_size_y ;
          else dy = disty - system_size_y ;
        } else dy = disty ;

        dz = fabs(zbox[i]-zbox[nachb_l[i]]) ;


        zahl = sqrt(dx*dx+dy*dy+dz*dz) ;
        av_bondlaenge = av_bondlaenge + zahl ;
        q = q + zahl * zahl ;
      }
    }
  }

  zahl = sqrt((q / anz_bonds) - ((av_bondlaenge/anz_bonds)*
                                 (av_bondlaenge/anz_bonds))) ;
  return(zahl) ;
}


/*------------------------------------------------------------*/
double msr_bondlaenge()
{
  double av_bondlaenge ;
  double dx,dy,dz,disty,distx ;
```

```
  int i,index ;
  int anz_bonds ;
  double dist ;


  anz_bonds = 0 ;
  av_bondlaenge = 0. ;

  for (i = 0; i< anzahl_teilchen;i++) {
    if (nachb_anz[i] != 0) {
      if (nachb_l[i] != INT_MAX) {
        anz_bonds ++ ;

        distx = xbox[i] - xbox[nachb_l[i]] ;
        if (fabs(distx) > system_size_x_halbe) {
          if (distx < 0) dx = distx + system_size_x ;
          else dx = distx - system_size_x ;
        } else dx = distx ;

        disty = ybox[i] - ybox[nachb_l[i]] ;
        if (fabs(disty) > system_size_y_halbe) {
          if (disty < 0) dy = disty + system_size_y ;
          else dy = disty - system_size_y ;
        } else dy = disty ;

        dz = fabs(zbox[i]-zbox[nachb_l[i]]) ;

        dist =  sqrt(dx*dx+dy*dy+dz*dz) ;

        av_bondlaenge = av_bondlaenge + dist ;

        index = (int) ((dist/bondmin) * 400) ;
        bl_histo[index] ++ ;
      }
    }
  }

  av_bondlaenge = av_bondlaenge / anz_bonds ;
  return(av_bondlaenge) ;
}


/*------------------------------------------------------------*/

double msr_bondwinkel()
{
  double av_bondwinkel ;
  double dx,dy,dz,disty,distx ;
  double dx2,dy2,dz2,disty2,distx2 ;
  double ldir, rdir ;
  int i ;
  int anz_bonds ;

  anz_bonds = 0 ;
  av_bondwinkel = 0. ;

  for (i = 0; i< anzahl_teilchen;i++) {
    if (nachb_anz[i] == 2) {
      anz_bonds ++ ;

      distx = xbox[i] - xbox[nachb_l[i]] ;
      if (fabs(distx) > system_size_x_halbe) {
        if (distx < 0) dx = distx + system_size_x ;
        else dx = distx - system_size_x ;
      } else dx = distx ;

        disty = ybox[i] - ybox[nachb_l[i]] ;
        if (fabs(disty) > system_size_y_halbe) {
          if (disty < 0) dy = disty + system_size_y ;
          else dy = disty - system_size_y ;
        } else dy = disty ;

        dz = (zbox[i]-zbox[nachb_l[i]]) ;

        distx2 = xbox[i] - xbox[nachb_r[i]] ;
        if (fabs(distx2) > system_size_x_halbe) {
          if (distx2 < 0) dx2 = distx2 + system_size_x ;
          else dx2 = distx2 - system_size_x ;
        } else dx2 = distx2 ;

        disty2 = ybox[i] - ybox[nachb_r[i]] ;
        if (fabs(disty2) > system_size_y_halbe) {
          if (disty2 < 0) dy2 = disty2 + system_size_y ;
          else dy2 = disty2 - system_size_y ;
        } else dy2 = disty2 ;

        dz2 = (zbox[i]-zbox[nachb_r[i]]) ;

        ldir = sqrt(dx*dx +dy*dy+dz*dz) ;
        rdir = sqrt(dx2*dx2+dy2*dy2+dz2*dz2) ;

        av_bondwinkel = av_bondwinkel + acos((dx*dx2+dy*dy2+dz*dz2)/(ldir*rdir)) ;

    }
  }
  if (anz_bonds == 0) return(0) ;
  av_bondwinkel = av_bondwinkel / anz_bonds ;
  return(av_bondwinkel) ;
}


/*------------------------------------------------------------*/

double msr_sigma_bondwinkel()
{
  double av_bondwinkel ;
  double dx,dy,dz,disty,distx ;
  double dx2,dy2,dz2,disty2,distx2 ;
  double ldir, rdir ;
  int i ;
  int anz_bonds ;
  double q,zahl ;

  anz_bonds = 0 ;
  av_bondwinkel = 0. ;
  q = 0 ;

  for (i = 0; i< anzahl_teilchen;i++) {
    if (nachb_anz[i] == 2) {
      anz_bonds ++ ;

      distx = xbox[i] - xbox[nachb_l[i]] ;
      if (fabs(distx) > system_size_x_halbe) {
        if (distx < 0) dx = distx + system_size_x ;
        else dx = distx - system_size_x ;
      } else dx = distx ;

        disty = ybox[i] - ybox[nachb_l[i]] ;
        if (fabs(disty) > system_size_y_halbe) {
          if (disty < 0) dy = disty + system_size_y ;
          else dy = disty - system_size_y ;
        } else dy = disty ;

        dz = (zbox[i]-zbox[nachb_l[i]]) ;

        distx2 = xbox[i] - xbox[nachb_r[i]] ;
        if (fabs(distx2) > system_size_x_halbe) {
          if (distx2 < 0) dx2 = distx2 + system_size_x ;
          else dx2 = distx2 - system_size_x ;
        } else dx2 = distx2 ;

        disty2 = ybox[i] - ybox[nachb_r[i]] ;
```

```
        if (fabs(disty2) > system_size_y_halbe) {
          if (disty2 < 0) dy2 = disty2 + system_size_y ;
          else dy2 = disty2 - system_size_y ;
        } else dy2 = disty2 ;

        dz2 = (zbox[i]-zbox[nachb_r[i]]) ;

        ldir = sqrt(dx*dx +dy*dy+dz*dz) ;
        rdir = sqrt(dx2*dx2+dy2*dy2+dz2*dz2) ;

        zahl =  acos((dx*dx2+dy*dy2+dz*dz2)/(ldir*rdir)) ;
        av_bondwinkel = av_bondwinkel + zahl ;
        q = q + zahl *zahl ;
    }
  }
  if (anz_bonds == 0) return(0) ;

  zahl = sqrt((q/ anz_bonds) - ((av_bondwinkel / anz_bonds)*
                                (av_bondwinkel / anz_bonds))) ;

  return(zahl) ;
}


/*-------------------------------------------------------------*/

void write_histo()

/* schreibt die Messhistogramme in die jeweiligen .hst Dateien */

{
  int i ;
  char fname[50] ;
  FILE *f1,*f2,*f3,*f4,*f5 ;

  sprintf(fname,"%s_the.hst",sflag[1]);
  f1 = fopen(fname,"w");

  sprintf(fname,"%s_phi.hst",sflag[1]);
  f2 = fopen(fname,"w");

  sprintf(fname,"%s_sdi.hst",sflag[1]);
  f3 = fopen(fname,"w");

  sprintf(fname,"%s_bl.hst",sflag[1]);
  f4 = fopen(fname,"w");

  sprintf(fname,"%s_bw.hst",sflag[1]);
  f5 = fopen(fname,"w");


  for(i=0;i<=100;i++) {
    fprintf(f1," %lf %ld %ld %ld \n",(i*M_PI*0.5)/100,histo_theta[i],
            histo_theta_l[i],histo_theta_p[i]) ;
  }
  for(i=0;i<=200;i++) {
    fprintf(f2," %lf %ld %ld %ld \n",((i-100)*M_PI)/100,histo_phi[i],
            histo_phi_l[i],histo_phi_p[i]) ;
  }
  for(i=0;i<Schichtzahl;i++) {
    fprintf(f3," %d %lf \n",i,(histo_dichte[i]/(Schichtvolumen))) ;
  }

  for(i=0;i<bl_histo_max;i++) {
    fprintf(f4," %lf %ld %ld %lf %ld \n",((i*bondmin)/400.),bl_histo[i],
            blz_histo[i],(((i*bondmin)*(laenge-1))/400.),ketten_laenge_z_histo[i]) ;
  }

  for(i=0;i<bw_histo_max;i++) {
    fprintf(f5," %lf %ld %lf %ld \n",(i*M_PI)/bw_histo_max,bw_histo[i],
            (i*M_PI)/(bw_histo_max*2),b_d_histo[i]) ;
  }

  fclose(f1) ;
  fclose(f2) ;
  fclose(f3) ;
  fclose(f4) ;
  fclose(f5) ;
}

/*-------------------------------------------------------------*/

void read_histo()

/* liest die Messhistogramme zur Fortschreibung in den Speicher ein */

{
  int i ;
  char fname[50],zeile[120] ;
  FILE *f1,*f2,*f3,*f4,*f5 ;
  double dummy,wert ;

  sprintf(fname,"%s_the.hst",sflag[1]);
  f1 = fopen(fname,"r");

  sprintf(fname,"%s_phi.hst",sflag[1]);
  f2 = fopen(fname,"r");

  sprintf(fname,"%s_sdi.hst",sflag[1]);
  f3 = fopen(fname,"r");

  sprintf(fname,"%s_bl.hst",sflag[1]);
  f4 = fopen(fname,"r");

  sprintf(fname,"%s_bw.hst",sflag[1]);
  f5 = fopen(fname,"r");

  for(i=0;i<=100;i++) {
    fgets(zeile,120,f1) ;
    sscanf(zeile," %lf %ld %ld %ld",&dummy,&histo_theta[i],&histo_theta_l[i],
           &histo_theta_p[i]) ;
  }
  for(i=0;i<=200;i++) {
    fgets(zeile,120,f2) ;
    sscanf(zeile," %lf %ld %ld %ld",&dummy,&histo_phi[i],&histo_phi_l[i],
           &histo_phi_p[i]) ;
  }
  for(i=0;i<Schichtzahl;i++) {
    fgets(zeile,120,f3) ;
    sscanf(zeile," %lf %lf \n",&dummy,&wert) ;
    histo_dichte[i] = (int) (wert * Schichtvolumen) ;
  }

  for(i=0;i<bl_histo_max;i++) {
    fgets(zeile,120,f4) ;
    sscanf(zeile," %lf %ld %ld %lf %ld \n",&dummy,&bl_histo[i],&blz_histo[i],&dummy,
           &ketten_laenge_z_histo[i]) ;
  }

  for(i=0;i<bw_histo_max;i++) {
    fgets(zeile,120,f5) ;
    sscanf(zeile," %lf %ld %lf %ld \n",&dummy,&bw_histo[i],&dummy,&b_d_histo[i]) ;
  }


  fclose(f1) ;
  fclose(f2) ;
  fclose(f3) ;
  fclose(f4) ;
  fclose(f5) ;
}
```

```
/*-------------------------------------------------------------*/

double msr_poly_theta()
{
  /* berechnet den mittleren Neigunswinkel der Polymerbuersten gegen die z-Achse*/

  double wert,distx,disty,distz,dx,dy,dz ;
  int i,next ;


  wert = 0 ;

  for (i=0;i<anzahl_teilchen;i++) {
    if (nachb_anz[i] == 3) {
      next = i ;

      /* Suche Kettenende zu Kettenkopf */

      do {
        next = nachb_r[next] ;
      } while (nachb_anz[next] != 1) ;

      /* Kettenende gefunden */

      distx = xbox[next] - xbox[i] ;
      if (fabs(distx) > system_size_x_halbe) {
        if (distx < 0) dx = distx + system_size_x ;
        else dx = distx - system_size_x ;
      } else dx = distx ;

      disty = ybox[next] - ybox[i] ;
      if (fabs(disty) > system_size_y_halbe) {
        if (disty < 0) dy = disty + system_size_y ;
        else dy = disty - system_size_y ;
      } else dy = disty ;

      distz = zbox[next] - zbox[i] ;
      if (fabs(distz) > system_size_z_real_halbe) {
        if (distz < 0) dz = distz + system_size_z_real ;
        else dz = distz - system_size_z_real ;
      } else dz = distz ;

      wert = wert + atan((dx*dx+dy*dy)/(dz*dz)) ;

    }
  }
  return(wert/anzahl_polymere) ;
}
/*-------------------------------------------------------------*/

double msr_sigma_poly_theta()
{
  /* berechnet die Abweichung des mittleren Neigungswinkel der Polymerbuersten
     gegen die z-Achse */

  double zahl,wert,wert_2,distx,disty,distz,dx,dy,dz ;
  int i,next ;

  wert = 0 ;
  wert_2 = 0 ;

  for (i=0;i<anzahl_teilchen;i++) {
    if (nachb_anz[i] == 3) {
      next = i ;

      /* Suche Kettenende zu Kettenkopf */

      do {
        next = nachb_r[next] ;
      } while (nachb_anz[next] != 1) ;

      /* Kettenende gefunden */

      distx = xbox[next] - xbox[i] ;
      if (fabs(distx) > system_size_x_halbe) {
        if (distx < 0) dx = distx + system_size_x ;
        else dx = distx - system_size_x ;
      } else dx = distx ;

      disty = ybox[next] - ybox[i] ;
      if (fabs(disty) > system_size_y_halbe) {
        if (disty < 0) dy = disty + system_size_y ;
        else dy = disty - system_size_y ;
      } else dy = disty ;

      distz = zbox[next] - zbox[i] ;
      if (fabs(disty) > system_size_z_real_halbe) {
        if (distz < 0) dz = distz + system_size_z ;
        else dz = distz - system_size_z ;
      } else dz = distz ;

      zahl = atan((dx*dx+dy*dy)/(dz*dz)) ;

      wert = wert + zahl ;
      wert_2 = wert_2 + (zahl*zahl) ;

    }
  }

  zahl = sqrt((wert_2/(anzahl_polymere) - ((wert/anzahl_polymere)*(wert/anzahl_polymere)
))) ;

  return(zahl) ;
}
/*-------------------------------------------------------------*/



void msr_theta(double *theta0,double *theta1,double *theta2)
{
  /* berechnet mittleren Neigungswinkel */

  double wert ; /* Winkel des aktuellen Teilchens */
  int i,ihist ;

  *theta0 = 0 ;
  *theta1 = 0 ;
  *theta2 = 0 ;

  for (i=0;i < anzahl_teilchen;i++) {
    wert = acos(fabs(zdir[i])) ; /* Obacht ! */
    ihist = (int) (100*2*wert/M_PI) ;

    *theta0 = *theta0 + wert ;
    histo_theta[ihist] ++ ;

     if (nachb_anz[i] == 0) {
       *theta1 = *theta1 + wert ;
       histo_theta_l[ihist]++ ;
     }
     else {
       *theta2 = *theta2 + wert ;
       histo_theta_p[ihist]++ ;
     }
  }
  *theta0 = *theta0 / anzahl_teilchen ;
  *theta1 = *theta1 / anzahl ;
  if (anzahl_polymere >0 ) {
    *theta2 = *theta2 / (laenge*anzahl_polymere) ;
  } else {
    *theta2 = 0 ;
```

```
  }

 }


/*-------------------------------------------------------------*/

void msr_sigma_theta(double *sigma_theta0,double *sigma_theta1,double *sigma_theta2)
{
  /* berechnet mittleren Neigungswinkel */

  double help[3] ;
  double wert ; /* Winkel des aktuellen Teilchens */
  int i ;

  *sigma_theta0 = 0 ;
  *sigma_theta1 = 0 ;
  *sigma_theta2 = 0 ;
  help[0] = 0 ;
  help[1] = 0 ;
  help[2] = 0 ;

  for (i=0;i < anzahl_teilchen;i++) {
    if (zdir[i] >1) {
      printf("ARGL !!!!") ;
      zdir[i] = 1 ;
      fflush(stdout) ;

    }
     wert = acos(fabs(zdir[i])) ; /* Obacht ! */
     help[0] = help[0] + wert ;
     *sigma_theta0 = *sigma_theta0 + wert*wert ;

     if (nachb_anz[i] == 0) {
       *sigma_theta1 = *sigma_theta1 + wert*wert ;
       help[1] = help [1] + wert ; ;
     }
     else {
       *sigma_theta2 = *sigma_theta2 + wert*wert ;
       help[2] = help[2] + wert ;
     }
  }

  *sigma_theta0 = (*sigma_theta0 / anzahl_teilchen) -
    (help[0]*help[0]/(anzahl_teilchen*anzahl_teilchen)) ;
  *sigma_theta1 = (*sigma_theta1 / anzahl) - (help[1]*help[1]/(anzahl*anzahl)) ;
  if (anzahl_polymere >0) {
    *sigma_theta2 = (*sigma_theta2 / (laenge*anzahl_polymere)) -
      ((help[2]*help[2])/(laenge*anzahl_polymere*laenge*anzahl_polymere)) ;
  } else {
    *sigma_theta2 = 0 ;
  }
}

/*-------------------------------------------------------------*/

void msr_phi(double *phi0,double *phi1,double *phi2)
{
  /* berechnet mittleren Neigungswinkel */

  double wert ; /* Winkel des aktuellen Teilchens */
  int i,ihist ;

  *phi0 = 0 ;
  *phi1 = 0 ;
  *phi2 = 0 ;

  for (i=0;i < anzahl_teilchen;i++) {
    if (xdir[i] != 0) wert = atan2(ydir[i],xdir[i]) ;
    else {
      if (ydir[i] != 0) wert = (M_PI * ydir[i]) / (2. * fabs(ydir[i])) ;
      else wert = 0 ;
    }
    ihist = (int) (100*wert/M_PI) ;

    *phi0 = *phi0 + wert ;
    histo_phi[ihist+100] ++ ;

    if (nachb_anz[i] == 0) {
       *phi1 = *phi1 + wert ;
       histo_phi_l[ihist+100]++ ;
     }
     else {
       *phi2 = *phi2 + wert ;
       histo_phi_p[ihist+100]++ ;
     }
  }
  *phi0 = *phi0 / anzahl_teilchen ;
  *phi1 = *phi1 / anzahl ;
  if (anzahl_polymere >0) {
    *phi2 = *phi2 / (laenge*anzahl_polymere) ;
  } else {
    *phi2 = 0 ;
  }
}

/*-------------------------------------------------------------*/

void msr_sigma_phi(double *sigma_phi0,double *sigma_phi1,double *sigma_phi2)
{
  /* berechnet mittleren Neigungswinkel */

  double help[3] ;
  double wert ; /* Winkel des aktuellen Teilchens */
  int i ;

  *sigma_phi0 = 0 ;
  *sigma_phi1 = 0 ;
  *sigma_phi2 = 0 ;
  help[0] = 0 ;
  help[1] = 0 ;
  help[2] = 0 ;

  for (i=0;i < anzahl_teilchen;i++) {
    if (xdir[i] != 0) wert = atan2(ydir[i],xdir[i]) ;
    else {
      if (ydir[i] != 0) wert = (M_PI * ydir[i]) / (2. * fabs(ydir[i])) ;
      else wert = 0 ;
    }
    help[0] = help[0] + wert ;
    *sigma_phi0 = *sigma_phi0 + wert*wert ;

    if (nachb_anz[i] == 0) {
      *sigma_phi1 = *sigma_phi1 + wert*wert ;
      help[1] = help [1] + wert ; ;
    }
    else {
      *sigma_phi2 = *sigma_phi2 + wert*wert ;
      help[2] = help[2] + wert ;
    }
  }

  *sigma_phi0 = *sigma_phi0 / anzahl_teilchen - help[0]*help[0]/(anzahl_teilchen*anzahl_
teilchen) ;
  *sigma_phi1 = *sigma_phi1 / anzahl - help[1]*help[1]/(anzahl*anzahl) ;
  if (anzahl_polymere >0) {
    *sigma_phi2 = (*sigma_phi2 / (laenge*anzahl_polymere)) - ((help[2]*help[2])/(laenge*
anzahl_polymere*laenge*anzahl_polymere)) ;
  } else {
    *sigma_phi2 = 0 ;
  }
```

```
}


/*------------------------------------------------------------*/

void msr_press(double *xx_p,double *yy_p,double *zz_p,
               double *xy_p,double *xz_p,double *yz_p)

/* Berechnet die Diagonalelemente des Drucktensors */

{

 double help ;
  double dx,dy,dz,distx,disty,distz,dist,dist_eff,skp1,skp2,skp1p2q,skp1m2q,skp12,rx,ry,
rz ;
  double sigma,r11,r5 ;
  double kraft_term_zz,wurzel ;
  double kraft_term_xx,kraft_term_yy ;
  double kraft_term_xy ;
  double kraft_term_xz ;
  double kraft_term_yz ;
   int i,k,j,a ;
  double d_r_z,d_r_y,d_r_x ;
  double d_sigma_x,d_sigma_y,d_sigma_z ;
  double d_E_z,d_E_y,d_E_x ;
  double press_xi,press_epsilon_0,press_sigma_s ;
  double bl ;
  double bondcut_q ;
  double bx,by,bz ;
  double w3 ;

  bondcut_q = bondcut * bondcut ;

  kraft_term_xx = 0. ;
  kraft_term_yy = 0. ;
  kraft_term_zz = 0. ;
  kraft_term_xy = 0. ;
  kraft_term_xz = 0. ;
  kraft_term_yz = 0. ;

  for (i=0 ;i<anzahl_teilchen ;i++) {
    for (k=1 ;k<=upper_verlet[i] ;k++) {
      j = tab_verlet[i][k] ;

      /* berechne den Abstand zweier Teilchen */


      distx = xbox[j] - xbox[i] ;
      if (fabs(distx) > system_size_x_halbe) {
        if (distx < 0) dx = distx + system_size_x ;
        else dx = distx - system_size_x ;
      } else dx = distx ;

      disty = ybox[j] - ybox[i] ;
      if (fabs(disty) > system_size_y_halbe) {
        if (disty < 0) dy = disty + system_size_y ;
        else dy = disty - system_size_y ;
      } else dy = disty ;

      distz = zbox[j] - zbox[i] ;
      if (fabs(distz) > system_size_z_real_halbe) {
        if (distz < 0) dz = distz + system_size_z_real ;
        else dz = distz - system_size_z_real ;
      } else dz = distz ;


      help = dx*dx+dy*dy+dz*dz ;

      dist = sqrt(help) ;

      /* normiere den Abstandsvektor */

      help = 1. / dist ;
      rx = dx * help ;
      ry = dy * help ;
      rz = dz * help ;

      /* berechne Skalarprodukte zwischen Richtungsvektoren und Abstandsvektor */

      skp1 = (xdir[i]*rx + ydir[i]*ry + zdir[i]*rz) ;
      skp2 = (xdir[j]*rx + ydir[j]*ry + zdir[j]*rz) ;
      skp12 =(xdir[i]*xdir[j]+ydir[i]*ydir[j]+zdir[i]*zdir[j]) ;
      skp1p2q = (skp1+skp2)*(skp1+skp2) ;
      skp1m2q = (skp1-skp2)*(skp1-skp2) ;

      /* Unterscheidung zwischen verschiedenen Teilchen */

      a = nachb_anz[i] + nachb_anz[j] ;

      switch(a)
        {
        case 0 :
          press_xi = xi ;
          press_epsilon_0 = epsilon_0 ;
          press_sigma_s = sigma_s ;
          break ;
        case 1 :
          press_xi = xi_pl ;
          press_epsilon_0 = epsilon_0_pl ;
          press_sigma_s = sigma_s_pl ;
          break ;
        default :

          if ((nachb_anz[i] == 3) || (nachb_anz[j] == 3)) {
            press_xi = 1 ;
            press_epsilon_0 = 0 ;
            press_sigma_s = 0 ;
          } else {

            if ((nachb_anz[i] == 0) || (nachb_anz[j] == 0)) {
              press_xi = xi_pl ;
              press_epsilon_0 = epsilon_0_pl ;
              press_sigma_s = sigma_s_pl ;
            }
            else {
              press_xi = xi_pp ;
              press_epsilon_0 = epsilon_0_pp ;
              press_sigma_s = sigma_s_pp ;
            }
          }
        }



      /* berechene "effektiven" Abstand */

      help = press_xi*skp12 ;

      wurzel = sqrt (1. - ((press_xi/2.) * ((skp1p2q/(1.+help)) + (skp1m2q/(1.-help)))))
;

      if (wurzel > 0.) {
        sigma = press_sigma_s / wurzel ;
      } else {
        sigma = 1e-30 ;
      }

      if (dist < (sigma+cutoff)) {
        /* Teilchen ueberschneiden sich  */

        w3 = wurzel*wurzel*wurzel ;
```

```
dist_eff = dist - sigma + press_sigma_s ;


d_sigma_x = (press_sigma_s/w3) * (press_xi/(2.*dist)) *
  ((((skp1+skp2)*(xdir[i]+xdir[j]-rx*(skp1+skp2)))/(1.+help)) +
   (((skp1-skp2)*(xdir[i]-xdir[j]-rx*(skp1-skp2)))/(1.-help))) ;

d_sigma_y = (press_sigma_s/w3) * (press_xi/(2.*dist)) *
  ((((skp1+skp2)*(ydir[i]+ydir[j]-ry*(skp1+skp2)))/(1.+help)) +
   (((skp1-skp2)*(ydir[i]-ydir[j]-ry*(skp1-skp2)))/(1.-help))) ;

d_sigma_z = (press_sigma_s/w3) * (press_xi/(2.*dist)) *
  ((((skp1+skp2)*(zdir[i]+zdir[j]-rz*(skp1+skp2)))/(1.+help)) +
   (((skp1-skp2)*(zdir[i]-zdir[j]-rz*(skp1-skp2)))/(1.-help))) ;

d_r_z = (press_sigma_s * (d_sigma_z-rz)) / (dist_eff * dist_eff) ;
d_r_y = (press_sigma_s * (d_sigma_y-ry)) / (dist_eff * dist_eff) ;
d_r_x = (press_sigma_s * (d_sigma_x-rx)) / (dist_eff * dist_eff) ;

help = (press_sigma_s/dist_eff) ;

r5 = help * help * help * help * help ;
r11 = help * r5 *r5 ;

d_E_z = 4 * press_epsilon_0 * ( 12 * r11 - 6 *r5) * d_r_z ;
d_E_y = 4 * press_epsilon_0 * ( 12 * r11 - 6 *r5) * d_r_y ;
d_E_x = 4 * press_epsilon_0 * ( 12 * r11 - 6 *r5) * d_r_x ;


/* Hier kommen ein paar kleine Testroutinen und abfragen */

/*
if ((12*r11-6*r5) < 0.) {
  printf (" Negative LJ-Klammer !!!! fuer Teilchen %ld %ld : %lf \n",i,j,
  (12*r11-6*r5)) ;
  printf (" Koordinaten Teilchen %d : %lf %lf %lf %lf %lf %lf \n",i,xbox[i],
  ybox[i],zbox[i],xdir[i],ydir[i],zdir[i]) ;
  printf (" Koordinaten Teilchen %d : %lf %lf %lf %lf %lf %lf \n",j,xbox[j],
  ybox[j],zbox[j],xdir[j],ydir[j],zdir[j]) ;
  printf("\n") ;
}

if (dist <(sigma-1)) {
  printf (" Distanz zu klein !!!!! fuer Teilchen  %ld %ld : %lf %lf \n",i,j,
  dist,sigma-1) ;
  printf (" Koordinaten Teilchen %d : %lf %lf %lf %lf %lf %lf \n",i,xbox[i],
  ybox[i],zbox[i],xdir[i],ydir[i],zdir[i]) ;
  printf (" Koordinaten Teilchen %d : %lf %lf %lf %lf %lf %lf \n",j,xbox[j],
  ybox[j],zbox[j],xdir[j],ydir[j],zdir[j]) ;
  printf("\n") ;
}

if ((d_r_z*dz+d_r_y*dy+d_r_x*dx) > 0.) {
  printf (" Summe negativ !!!!!  fuer Teilchen  %ld %ld : %lf \n",i,j,
  (d_r_z*dz+d_r_y*dy+d_r_x*dx)) ;
  printf (" Koordinaten Teilchen %d : %lf %lf %lf %lf %lf %lf \n",i,xbox[i],
  ybox[i],zbox[i],xdir[i],ydir[i],zdir[i]) ;
  printf (" Koordinaten Teilchen %d : %lf %lf %lf %lf %lf %lf \n",j,xbox[j],
  ybox[j],zbox[j],xdir[j],ydir[j],zdir[j]) ;
  printf (" x: %lf %lf \n",d_r_x,dx) ;
  printf (" y: %lf %lf \n",d_r_y,dy) ;
  printf (" z: %lf %lf \n",d_r_z,dz) ;
  printf("\n") ;
}
*/

/* Ende der Tests */

kraft_term_zz += (d_E_z * dz) ;
kraft_term_yy += (d_E_y * dy) ;
kraft_term_xx += (d_E_x * dx) ;
      kraft_term_xy += (d_E_x * dy) ;
      kraft_term_xz += (d_E_x * dz) ;
      kraft_term_yz += (d_E_y * dz) ;
    }
  }
}

for (i = 0;i<anzahl_teilchen;i++) {
  if (nachb_l[i] != INT_MAX) {

    distx = xbox[nachb_l[i]] - xbox[i] ;
    if (fabs(distx) > system_size_x_halbe) {
      if (distx < 0) dx = distx + system_size_x ;
else dx = distx - system_size_x ;
    } else dx = distx ;

    disty = ybox[nachb_l[i]] - ybox[i] ;
    if (fabs(disty) > system_size_y_halbe) {
      if (disty < 0) dy = disty + system_size_y ;
      else dy = disty - system_size_y ;
    } else dy = disty ;


    distz = zbox[nachb_l[i]] - zbox[i] ;
    if (fabs(distz) > system_size_z_real_halbe) {
      if (distz < 0) dz = distz + system_size_z_real ;
      else dz = distz - system_size_z_real ;
    } else dz = distz ;

    help = dx*dx+dy*dy+dz*dz ;

    dist = sqrt(help) ;

    bl = dist - bondmin ;

    bx = dx*sprng*bl*(1/((1-((bl*bl)/bondcut_q))*dist));
    by = dy*sprng*bl*(1/((1-((bl*bl)/bondcut_q))*dist));
    bz = dz*sprng*bl*(1/((1-((bl*bl)/bondcut_q))*dist));

    kraft_term_zz += (bz * dz) ;
    kraft_term_yy += (by * dy) ;
    kraft_term_xx += (bx * dx) ;
    kraft_term_xy += (bx * dy) ;
    kraft_term_xz += (bx * dz) ;
    kraft_term_yz += (by * dz) ;
  }
}

*zz_p = (-(kraft_term_zz)+(anzahl_teilchen*temp)) /
  (system_size_z*system_size_y*system_size_x) ;
*yy_p = (-(kraft_term_yy)+(anzahl_teilchen*temp)) /
  (system_size_z*system_size_y*system_size_x) ;
*xx_p = (-(kraft_term_xx)+(anzahl_teilchen*temp)) /
  (system_size_z*system_size_y*system_size_x) ;
*xy_p = (-(kraft_term_xy)) /(system_size_z*system_size_y*system_size_x) ;
*xz_p = (-(kraft_term_xz)) /(system_size_z*system_size_y*system_size_x) ;
*yz_p = (-(kraft_term_yz)) /(system_size_z*system_size_y*system_size_x) ;

}

/*------------------------------------------------------------*/


double winkel(int i)

     /* Berechnet den Winkel zwischen den beiden Bonds des Teilchens i */

{
  double distx,disty,distz,dx,dy,dz,dist,dx2,dy2,dz2,dist2 ;
  double cos ;
```

```
  double bowi_energie ;
  double xb1,xb2,xb3,yb1,yb2,yb3,zb1,zb2,zb3 ;

  /* berechne den Abstand zweier Teilchen */

  xb1 = xbox[i] ;
  yb1 = ybox[i] ;
  zb1 = zbox[i] ;
  xb2 = xbox[nachb_l[i]] ;
  yb2 = ybox[nachb_l[i]] ;
  zb2 = zbox[nachb_l[i]] ;
  xb3 = xbox[nachb_r[i]] ;
  yb3 = ybox[nachb_r[i]] ;
  zb3 = zbox[nachb_r[i]] ;


  distx = xb2 - xb1 ;
  if (fabs(distx) > system_size_x_halbe) {
    if (distx < 0) dx = distx + system_size_x ;
    else dx = distx - system_size_x ;
  } else dx = distx ;

  disty = yb2 - yb1 ;
  if (fabs(disty) > system_size_y_halbe) {
    if (disty < 0) dy = disty + system_size_y ;
    else dy = disty - system_size_y ;
  } else dy = disty ;

  distz = zb2 - zb1 ;
  if (fabs(distz) > system_size_z_real_halbe) {
    if (distz < 0) dz = distz + system_size_z_real ;
    else dz = distz - system_size_z_real ;
  } else dz = distz ;

  dist = sqrt(dx*dx+dy*dy+dz*dz) ;

  distx = xb3 - xb1 ;
  if (fabs(distx) > system_size_x_halbe) {
    if (distx < 0) dx2 = distx + system_size_x ;
    else dx2 = distx - system_size_x ;
  } else dx2 = distx ;

  disty = yb3 - yb1 ;
  if (fabs(disty) > system_size_y_halbe) {
    if (disty < 0) dy2 = disty + system_size_y ;
    else dy2 = disty - system_size_y ;
  } else dy2 = disty ;

  distz = zb3 - zb1 ;
  if (fabs(distz) > system_size_z_real_halbe) {
    if (distz < 0) dz2 = distz + system_size_z_real ;
    else dz2 = distz - system_size_z_real ;
  } else dz2 = distz ;

  dist2 = sqrt(dx2*dx2+dy2*dy2+dz2*dz2) ;

  cos = (dx*dx2 + dy*dy2 + dz*dz2) / (dist*dist2) ;


  return(acos(cos)) ;

}


/*-------------------------------------------------------------*/

double b_d_winkel(int i,int j)

{
  double distx,disty,distz,dx,dy,dz,dist ;
  double cos ;
  double xb1,xb2,yb1,yb2,zb1,zb2 ;


  xb1 = xbox[i] ;
  yb1 = ybox[i] ;
  zb1 = zbox[i] ;
  xb2 = xbox[j] ;
  yb2 = ybox[j] ;
  zb2 = zbox[j] ;

  distx = xb2 - xb1 ;
  if (fabs(distx) > system_size_x_halbe) {
    if (distx < 0) dx = distx + system_size_x ;
    else dx = distx - system_size_x ;
  } else dx = distx ;

  disty = yb2 - yb1 ;
  if (fabs(disty) > system_size_y_halbe) {
    if (disty < 0) dy = disty + system_size_y ;
    else dy = disty - system_size_y ;
  } else dy = disty ;

  distz = zb2 - zb1 ;
  if (fabs(distz) > system_size_z_real_halbe) {
    if (distz < 0) dz = distz + system_size_z_real ;
    else dz = distz - system_size_z_real ;
  } else dz = distz ;

  dist = sqrt(dx*dx+dy*dy+dz*dz) ;

  cos = (dx*xdir[i] + dy*ydir[i] + dz*zdir[i]) / dist ;


  if (cos < 0) {
    cos = (-1) * cos ;
  }

  return (acos(cos)) ;
}


/*-------------------------------------------------------------*/

void msr_blz_bw_histo()
{

  /* Fuehrt ein Update der blz- und bw- Histogramme durch - das bl-Histogramm wird in
     msr_bond_laenge verarbeitet */
  /* Auch das ketten_laengen_z_hist wird hier bearbeitet ! */
  /* und auch b_d_histo ... */

  int    i,j,index,head,tail,next ;
  double wert,dist ;


  for (i = 0;i<anzahl_teilchen;i++) {
    if ((nachb_anz[i] != 0) && (nachb_anz[i] != 3)) {
      /* Teilchen hat Bonds und ist kein Kopf !*/

      if (nachb_l[i] != INT_MAX) {
        wert = b_d_winkel(i,nachb_l[i]) ;
        index = (int) (wert*bw_histo_max/(M_PI/2)) ;
        b_d_histo[index] ++ ;
      }
      if (nachb_r[i] != INT_MAX) {
        wert = b_d_winkel(i,nachb_r[i]) ;
        index = (int) (wert*bw_histo_max/(M_PI/2)) ;
        b_d_histo[index] ++ ;
```

```
    }
   }
  }


  for (i = 0;i<anzahl_teilchen;i++) {
    if (nachb_anz[i] == 2) {
      wert = winkel(i) ;
      index = (int) (wert*bw_histo_max/M_PI) ;
      bw_histo[index] ++ ;
    }
  }

  for (i= 0; i<anzahl_teilchen;i++) {
    if (nachb_r[i] != INT_MAX) {
      wert = fabs(zbox[i] - zbox[nachb_r[i]]) ;
      index = (int) (((wert/bondmin) * 400)) ;
      blz_histo[index] ++ ;
    }
  }

  for (i = 0; i<anzahl_teilchen;i++) {
    if (nachb_anz[i] == 3) {
      head = i ;
      /* Kettenkopf gefunden - jetzt muessen wir nur noch das passende Ende finden */
      tail = i ;
      for (j=0;j<(laenge-1);j++) {
        next  = nachb_r[tail] ;
        tail = next ;
      }

      wert = fabs(zbox[head] - zbox[tail]) ;
      index = (int) (((wert/((laenge-1)*bondmin)) * 400)) ;
      ketten_laenge_z_histo[index] ++ ;
    }
  }


}



/*-------------------------------------------------------------*/

void mess(int step,double akz_rate_move,double akz_rate_rot,
          double akz_rate_move_kette,double akz_rate_rot_kette,
          double akz_rate_V_move,double akz_rate_ketten_move,
          double akz_rate_ketten_move_same,double skin)

/* DIE Messroutine schlechthin ! Hier werden die Messdateien geoeffnet, erweitert
   und wieder geschlossen */

{
  char fname[50];
  double theta[3],sigma_theta[3] ;
  double phi[3],sigma_phi[3] ;
  double press_xx,press_yy,press_zz ;
  double press_xy,press_xz,press_yz ;
  double bond_summe,m_energie ;
  int i,l ;
  int j,k ;
  double neu ;
  double OP[3],Q[6],E[3] ; /* EW , Matrixelement und groesster EV */

  FILE *f1,*f2,*f3,*f4,*f5,*f6,*f7,*f8,*f9,*f10,*f11 ;

  bond_summe = 0 ;

  sprintf(fname,"%s_ene.dat",sflag[1]);
  f1 = fopen(fname,"a");

  sprintf(fname,"%s_the.dat",sflag[1]);
  f2 = fopen(fname,"a");

  sprintf(fname,"%s_akz.dat",sflag[1]);
  f3 = fopen(fname,"a");

  sprintf(fname,"%s_phi.dat",sflag[1]);
  f4 = fopen(fname,"a");

  sprintf(fname,"%s_bol.dat",sflag[1]);
  f5 = fopen(fname,"a");

  sprintf(fname,"%s_bow.dat",sflag[1]);
  f6 = fopen(fname,"a");

  sprintf(fname,"%s_kor.dat",sflag[1]);
  f7 = fopen(fname,"a");

  sprintf(fname,"%s_dru.dat",sflag[1]);
  f8 = fopen(fname,"a");

  sprintf(fname,"%s_pol.dat",sflag[1]);
  f9 = fopen(fname,"a");

  sprintf(fname,"%s_sch.dat",sflag[1]);
  f10 = fopen(fname,"a");

  sprintf(fname,"%s_sop.dat",sflag[1]);
  f11 = fopen(fname,"a");

  m_energie = msr_energie() ;

  for (i=0;i<anzahl_teilchen;i++) {
    bond_summe += bond_energie[i][0] ; /* Potential zwischen Teilchen und Bond */
    bond_summe += bond_energie[i][2] ; /* Potential zwischen 2 Bonds */

  }

  bond_summe = bond_summe / anzahl_teilchen ;

  fprintf(f1," %d %lf %lf %lf \n",step,m_energie,(m_energie-bond_summe),bond_summe) ;
  fclose(f1) ;

  msr_theta(&theta[0],&theta[1],&theta[2]) ;
  msr_sigma_theta(&sigma_theta[0],&sigma_theta[1],&sigma_theta[2]) ;
  msr_phi(&phi[0],&phi[1],&phi[2]) ;
  msr_sigma_phi(&sigma_phi[0],&sigma_phi[1],&sigma_phi[2]) ;
  msr_press(&press_xx,&press_yy,&press_zz,&press_xy,&press_xz,&press_yz) ;
  msr_S(&OP[0],&OP[1],&OP[2],&Q[0],&Q[1],&Q[2],&Q[3],&Q[4],&Q[5],&E[0],&E[1],&E[2]) ;
  msr_schicht_dichte() ;
  msr_blz_bw_histo() ;



  fprintf(f2," %d %lf %lf %lf %lf %lf %lf \n",step,theta[0],sigma_theta[0],theta[1],
          sigma_theta[1],theta[2],sigma_theta[2]) ;
  fclose(f2) ;

  fprintf(f4," %d %lf %lf %lf %lf %lf %lf \n",step,phi[0],sigma_phi[0],phi[1],
          sigma_phi[1],phi[2],sigma_phi[2]) ;
  fclose(f4) ;

  fprintf(f3," %d %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf \n",step,akz_rate_move,
          akz_rate_move_kette,max_jump,akz_rate_rot,akz_rate_rot_kette,max_rotate,
          akz_rate_V_move,max_zerr,akz_rate_ketten_move,akz_rate_ketten_move_same,
          skin) ;
  fclose(f3) ;

  if (anzahl_polymere !=0) {
    fprintf(f5," %d %lf %lf \n",step,msr_bondlaenge(),msr_sigma_bondlaenge()) ;
  }
```

```
  fclose(f5) ;

  if (anzahl_polymere !=0) {
    fprintf(f6," %d %lf %lf \n",step,msr_bondwinkel(),msr_sigma_bondwinkel()) ;
  }
  fclose(f6) ;

  fprintf(f7," %d %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf \n",step,OP[0],
          OP[1],OP[2],E[0],E[1],E[2],Q[0],Q[1],Q[2],Q[3],Q[4],Q[5]) ;
  fclose(f7) ;

  fprintf(f8," %d %lf %lf %lf %lf %lf %lf %lf %lf \n",step,
          (press_xx+press_yy+press_zz)/3.,press_xx,press_yy,press_zz,system_size_z,
          press_xy,press_xz,press_yz) ;
  fclose(f8) ;

  if (anzahl_polymere !=0) {
    fprintf(f9," %d %lf %lf \n",step,msr_poly_theta(),msr_sigma_poly_theta()) ;
  }
  fclose(f9) ;

  fprintf(f10," %d %d \n",step,Schichtzahl_OP) ;
  for (i=0;i<Schichtzahl_OP;i++) {
    fprintf(f10," %d %lf %lf %lf %lf %lf %ld \n",i,Schicht_OP[i][0],
            Schicht_OP[i][1],Schicht_OP[i][2],Schicht_LEVx[i],Schicht_LEVy[i],
            Schicht_LEVz[i],Schicht_dichte[i]) ;
  }
  fprintf(f10," \n") ;
  fclose(f10) ;

  fprintf(f11," %d %d \n",step,Schichtzahl_OP) ;
  for (i=0;i<Schichtzahl_OP;i++) {
    fprintf(f11," %d %lf %lf %lf %lf %lf %ld \n",i,histo_S[i][0][0],
            histo_S[i][0][1],histo_S[i][0][2],histo_S[i][1][1],histo_S[i][1][2],
            histo_S[i][2][2],Schicht_dichte[i]) ;
  }
  fprintf(f11," \n") ;
  fclose(f11) ;

  write_histo() ;

}
/*-------------------------------------------------------------*/



double ww_energie(double xbo1,double ybo1,double zbo1,
                  double xdi1,double ydi1,double zdi1,
                  double xbo2,double ybo2,double zbo2,
                  double xdi2,double ydi2,double zdi2,
                  int n_anz1,int n_anz2)
{
  /* Berechnet NUR die WW-Energie via G-B */

  double e ;
  int k ;

  k = n_anz1 + n_anz2 ;

  /* gerechnet += 1. ; */

  switch(k)
    {
    case 0 :
      e = GB(xbo1,ybo1,zbo1,xdi1,ydi1,zdi1,xbo2,ybo2,zbo2,xdi2,ydi2,zdi2) ;
      break ;
    case 1 :
      e = GB_pl(xbo1,ybo1,zbo1,xdi1,ydi1,zdi1,xbo2,ybo2,zbo2,xdi2,ydi2,zdi2) ;
      break ;
    default :
      if ((n_anz1 ==3) || (n_anz2 == 3)) {
        /* Koepfe haben keine WW */
        return(0) ;
      }

      if ((n_anz1 == 0) || (n_anz2 == 0)) {
        e = GB_pl(xbo1,ybo1,zbo1,xdi1,ydi1,zdi1,xbo2,ybo2,zbo2,xdi2,ydi2,zdi2) ;
      }
      else {
        e = GB_pp(xbo1,ybo1,zbo1,xdi1,ydi1,zdi1,xbo2,ybo2,zbo2,xdi2,ydi2,zdi2) ;
      }
    }

  return(e) ;
}

/*-------------------------------------------------------------*/

double calculate_energie(double xb,double yb,double zb,
                         double xd,double yd,double zd,int i)

/* berechnet die komplette GB-WW-Energie eines Teilchens mit allen seine WW-Partnern */

{
  int j ;
  double calc_energie ;
  int tzahl ;
  int anzahl_ww ;
  /*  int zaehler ;
      double wert ;

      zaehler = 0 ;
      wert = 0. ;
      */

  calc_energie = 0 ;

  anzahl_ww = tab_verlet[i][0] ;

  for (j=1;j<=anzahl_ww;j++) {
    tzahl = tab_verlet[i][j] ;
    e_help[j] = ww_energie(xb,yb,zb,xd,yd,zd,xbox[tzahl],ybox[tzahl],zbox[tzahl],
                           xdir[tzahl],ydir[tzahl],zdir[tzahl],nachb_anz[i],
                           nachb_anz[tzahl]) ;

    calc_energie = calc_energie + e_help[j] ;

  }
  return(calc_energie) ;
}

/*-------------------------------------------------------------*/



double calculate_gesamt_energie()

/* berechnet die gesamt innere Energie des Systems */

{
  double e ;
  int i,j,k ;
  double wert1,wert2,wert3 ;

  wert1 = 0. ;
  wert2 = 0. ;
  wert3 = 0. ;

  e = 0. ;

  for (i=0;i<anzahl_teilchen;i++) {
    for (k=1;k<=tab_verlet[i][0];k++) {
```

```
      j = tab_verlet[i][k] ;
      ene_verlet[i][k] = ww_energie(xbox[i],ybox[i],zbox[i],xdir[i],ydir[i],zdir[i],
                                    xbox[j],ybox[j],zbox[j],xdir[j],ydir[j],zdir[j],
                                    nachb_anz[i],nachb_anz[j]) ;
      e = e + 0.5 * ene_verlet[i][k] ;
    }

    /* Berechne Bondlaengen-WW und Bondwinkel-WW */

    if (nachb_anz[i] != 0) {

      if (nachb_r[i] != INT_MAX) {
        if (i < nachb_r[i]) {
          wert2 = bond_laengen_potential(xbox[i],ybox[i],zbox[i],xbox[nachb_r[i]],ybox[n
achb_r[i]],zbox[nachb_r[i]]) ;
          wert2 += bond_winkel_potential(xbox[i],ybox[i],zbox[i],xdir[i],ydir[i],zdir[i]
,xbox[nachb_r[i]],ybox[nachb_r[i]],zbox[nachb_r[i]],xdir[nachb_r[i]],ydir[nachb_r[i]],zdi
r[nachb_r[i]],nachb_anz[i],nachb_anz[nachb_r[i]]) ;
          bond_energie[i][1] = wert2 ;
          bond_energie[nachb_r[i]][0] = wert2 ;
          e += wert2 ;
        }
      }
      if (nachb_l[i] != INT_MAX) {
        if (i < nachb_l[i]) {
          wert1 = bond_laengen_potential(xbox[i],ybox[i],zbox[i],xbox[nachb_l[i]],ybox[n
achb_l[i]],zbox[nachb_l[i]]) ;
          wert1 += bond_winkel_potential(xbox[i],ybox[i],zbox[i],xdir[i],ydir[i],zdir[i]
,xbox[nachb_l[i]],ybox[nachb_l[i]],zbox[nachb_l[i]],xdir[nachb_l[i]],ydir[nachb_l[i]],zdi
r[nachb_l[i]],nachb_anz[i],nachb_anz[nachb_l[i]]) ;
          bond_energie[i][0] = wert1 ;
          bond_energie[nachb_l[i]][1] = wert1 ;
          e += wert1 ;
        }
      }

      if (nachb_anz[i] == 2) {
        wert3 = bond_winkel_potential_2(xbox[i],ybox[i],zbox[i],xbox[nachb_l[i]],ybox[na
chb_l[i]],zbox[nachb_l[i]],
                                        xbox[nachb_r[i]],ybox[nachb_r[i]],zbox[nachb_r[i]]
) ;
        bond_energie[i][2] = wert3 ;
        e += wert3 ;
      }
    }
  }
  return(e) ;
}


/*-------------------------------------------------------------*/

void verlet_update()

/* fuehrt ein komplettes Update der Verlettabellen durch */

{
  char *tmp_in[MAX_TEILCHEN],*tmp_ptr;
  int ix,iy,iz,ip,jp,kp;
  int ii,jj,kk,x_array[14],y_array[14],z_array[14];
  int tzahl,tzahl2 ;
  struct part *CL,*CLa;
  double x,y,z;
  double dist,distx,disty,distz,dx,dy,dz;
  int count_lower[MAX_TEILCHEN],*end_ptr[MAX_TEILCHEN] ;


  verlet_update_counter++ ;

  diff_max = diff_max2 = 0.0 ;
  skin_verlet = lj_safety ; /* Schichtdicke der Verletkugel */
  rad_verlet = (cutoff_max + skin_verlet)*(cutoff_max + skin_verlet);

  /* Pseudoteilchenposition zum Zeitpunkt des Verlet-Updates, um nachher aus */
  /* der Drift die Notwendigkeit eines neuen Updates zu bestimmen           */

  for (ii=0; ii<anzahl_teilchen_mal2; ii++) {
    pseudo_xpos_0v[ii] = pseudo_xpos[ii] ;
    pseudo_ypos_0v[ii] = pseudo_ypos[ii] ;
    pseudo_zpos_0v[ii] = pseudo_zpos[ii] ;
  }

  /* 0-Element enthaelt Zahl der Partner in der Kugel, wird auf Null  */
  /* vorbesetzt                                                       */
  for (ii=0; ii<anzahl_teilchen; ii++) {
    tab_verlet[ii][0] = 0;
  }

  /* Logisches Hilfsfeld der Paare mit Abstand<rad_verlet vorbesetzen */

  kk=0;
  for (ii=0; ii<anzahl_teilchen; ii++) {
    tmp_in[ii] = &tmp[kk];
    kk = kk + anzahl_teilchen - 1 - ii;
  }

  for (ix=0; ix<zellen_x; ix++) {
    for (iy=0; iy<zellen_y; iy++) {
      for (iz=0; iz<zellen_z; iz++) {
        CLa = cell[ix][iy][iz];

        /* die Nachbarzellen in den anderen Richtungen werden */
        /* im Schleifenverlauf beruecksichtigt !              */
        x_array[ 0] = ix;        y_array[ 0] = iy;       z_array[ 0] = iz;
        x_array[ 1] = ix;        y_array[ 1] = iy;       z_array[ 1] = jpz[iz];
        x_array[ 2] = ix;        y_array[ 2] = jpy[iy];  z_array[ 2] = jpz[iz];
        x_array[ 3] = ix;        y_array[ 3] = jpy[iy];  z_array[ 3] = iz;
        x_array[ 4] = ix;        y_array[ 4] = jpy[iy];  z_array[ 4] = jmz[iz];
        x_array[ 5] = jpx[ix];   y_array[ 5] = jmy[iy];  z_array[ 5] = jpz[iz];
        x_array[ 6] = jpx[ix];   y_array[ 6] = iy;       z_array[ 6] = jpz[iz];
        x_array[ 7] = jpx[ix];   y_array[ 7] = jpy[iy];  z_array[ 7] = jpz[iz];
        x_array[ 8] = jpx[ix];   y_array[ 8] = jmy[iy];  z_array[ 8] = iz;
        x_array[ 9] = jpx[ix];   y_array[ 9] = iy;       z_array[ 9] = iz;
        x_array[10] = jpx[ix];   y_array[10] = jpy[iy];  z_array[10] = iz;
        x_array[11] = jpx[ix];   y_array[11] = jmy[iy];  z_array[11] = jmz[iz];
        x_array[12] = jpx[ix];   y_array[12] = iy;       z_array[12] = jmz[iz];
        x_array[13] = jpx[ix];   y_array[13] = jpy[iy];  z_array[13] = jmz[iz];

        for ( ;  CLa!=NULL; CLa=CLa->next) {
          /* ......aeussere Schleife ueber alle Monomere..... */
          ip      = CLa->pn  ;
          tzahl = ip >> 1 ;
          tmp_ptr = tmp_in[tzahl];
          x = pseudo_xbox[ip];
          y = pseudo_ybox[ip];
          z = pseudo_zbox[ip];

          for (ii=0; ii<14; ii++) {
            CL = cell[x_array[ii]][y_array[ii]][z_array[ii]];

            for ( ; CL!=NULL; CL=CL->next) {
              /* ...innere Schleife ueber alle Partner... */
              jp = CL->pn;
              tzahl2 = jp >> 1;
              if (tzahl != tzahl2) {

                /* Abstandsbestimmung im period. System */
                distx = fabs(pseudo_xbox[jp]-x);
                dx    = (double) min (distx,system_size_x-distx);
                disty = fabs(pseudo_ybox[jp]-y);
                dy    = (double) min (disty,system_size_y-disty);
                distz = fabs(pseudo_zbox[jp]-z);
```

```
              dz    = (double) min (distz,system_size_z_real-distz);
              dist  = dx*dx + dy*dy + dz*dz;

              if (dist<rad_verlet) {
                kp = tzahl2-tzahl;
                if (kp>0) {
                  tmp_ptr[kp-1]     = c_1;
                } else {
                  tmp_in[tzahl2][-kp-1] = c_1;
                }
              }
            }
          } /* ..................................... */
        }
      }
    }
  }

  /* In jeder Verlet-Kugel kommen zuerst die Monomere mit       */
  /* hoeherem Index danach die mit niedrigerem Index als das    */
  /* Monomer in der Kugelmitte                                  */
  kk=0;
  for (ii=0; ii<anzahl_teilchen; ii++) {
    kp = 0;
    for (jj=ii+1; jj<anzahl_teilchen; jj++) {
      if (tmp[kk]) {
        tmp[kk] = c_0 ;
        kp++;
        tab_verlet[ii][kp]=jj;

      }
      kk++;
    }

    upper_verlet[ii]=kp;
  }

  for (ii=0; ii<anzahl_teilchen; ii++) {
    count_lower[ii] = 0;
    end_ptr[ii]     = &tab_verlet[ii][1+upper_verlet[ii]];
  }

  for (ii=0; ii<anzahl_teilchen; ii++) {
    for (jj=1; jj<=upper_verlet[ii]; jj++) {
      kk=tab_verlet[ii][jj];
      *end_ptr[kk]=ii;
      end_ptr[kk]++; count_lower[kk]++;
    }
  }

  for (ii=0; ii<anzahl_teilchen; ii++) {
    tab_verlet[ii][0] = upper_verlet[ii] + count_lower[ii];
    if (tab_verlet[ii][0] > ww_max) {
      printf("\n ww_max to low !!!!!!!!!!!!!!!!!!!!!!!!!!!!! \n") ;
      exit(1) ;
    }
  }

  gesamt_energie = calculate_gesamt_energie() ;


}


/*-------------------------------------------------------------*/

void put_energie(int i)
/* speichert die Energie des Teilchens i in der Energie-Tabelle */
{
  int j,k ;
```

```
  for (j=1;j<=tab_verlet[i][0];j++) {
    ene_verlet[i][j] = e_help[j] ;
    for (k=1;k<=tab_verlet[tab_verlet[i][j]][0];k++) {
        if (tab_verlet[tab_verlet[i][j]][k] == i) {
          ene_verlet[tab_verlet[i][j]][k] = e_help[j] ;
          break ;
        }
    }
  }
}

/*-------------------------------------------------------------*/

int ketten_move()

/* versucht eine Kette neu zu bauen */

{

  int i,j,k ;
  double rnd ;
  int hit_head ;
  double ketten_wahrscheinlichkeit_alt ;
  double ketten_wahrscheinlichkeit_neu ;
  double bond_wahrscheinlichkeit_alt[MAX_KETTEN_LAENGE] ;
  double bond_wahrscheinlichkeit_neu[MAX_KETTEN_LAENGE] ;
  int alte_kette[MAX_KETTEN_LAENGE] ;
  int neue_kette[MAX_KETTEN_LAENGE] ;
  double alte_ketten_energie[MAX_KETTEN_LAENGE][3] ;
  int counter ;
  int aktuelles_teilchen ;
  int liste[ketten_ww_max] ;
  double wahrsch[ketten_ww_max] ;
  double xdist,ydist,zdist,dist ;
  double xq,yq,zq,distq ;
  double xa,ya,za ;
  double minq,maxq,max ;
  double energie[ketten_ww_max],bw_e[ketten_ww_max],bw_e2[ketten_ww_max],
    bl_e[ketten_ww_max] ;
  double help ;
  int gefunden ;
  double test ;
  double energie_alt,energie_neu ;


  minq = (bondmin - bondcut) * (bondmin - bondcut) ;
  max = (bondmin + bondcut) ;
  maxq = max * max ;

  energie_alt = 0. ;
  energie_neu = 0. ;

  ketten_wahrscheinlichkeit_neu = 1. ;
  ketten_wahrscheinlichkeit_alt = 1. ;

  for (i = 0 ; i< MAX_KETTEN_LAENGE; i++) {
    alte_kette[i] = 0 ;
    neue_kette[i] = 0 ;
    bond_wahrscheinlichkeit_alt[i] = 0. ;
    bond_wahrscheinlichkeit_neu[i] = 0. ;
    alte_ketten_energie[i][0] = 0. ;
    alte_ketten_energie[i][1] = 0. ;
    alte_ketten_energie[i][2] = 0. ;
  }


  /* Waehle zufaelligen Kettenkopf */

  rnd = (double_r250() * anzahl_polymere) ;
  hit_head = chain_heads[(int) rnd] ;
```

```
/* Speichere alte Kette und alte Energien */


i = 0 ;
j = hit_head ;
alte_kette[i] = j ;
while (nachb_r[j] != INT_MAX) {
  i++ ;
  j = nachb_r[j] ;
  alte_kette[i] = j ;
}

for (i=0;i<laenge;i++) {
  alte_ketten_energie[i][0] = bond_energie[alte_kette[i]][0] ;
  alte_ketten_energie[i][1] = bond_energie[alte_kette[i]][1] ;
  alte_ketten_energie[i][2] = bond_energie[alte_kette[i]][2] ;
}

/* Loesche alte Kette und ihre Energie */

for (i=0;i<laenge;i++) {
  nachb_anz[alte_kette[i]] = 0 ;
  nachb_r[alte_kette[i]] = INT_MAX ;
  nachb_l[alte_kette[i]] = INT_MAX ;
  bond_energie[alte_kette[i]][0] = 0. ;
  bond_energie[alte_kette[i]][1] = 0. ;
  bond_energie[alte_kette[i]][2] = 0. ;
}

/* berechne alte Kettenwahrscheinlichkeit */

/* Kopf bleibt Kopf ! */

nachb_anz[hit_head] = 3 ;
aktuelles_teilchen = hit_head ;

for (k=1;k<laenge;k++) {
  counter = 0 ; /* Anzahl der gefunden moeglichen Teilchen fuer den k-ten Bond */
  xa = xbox[aktuelles_teilchen] ;
  ya = ybox[aktuelles_teilchen] ;
  za = zbox[aktuelles_teilchen] ;

  for (j=0;j <anzahl_teilchen;j++) {
    zdist = zbox[j] - za ;
    if (fabs(zdist) <= max) {
      zq = zdist * zdist ;

      xdist = fabs(xbox[j] - xa) ;
      xdist = (double) min (xdist,system_size_x-xdist) ;
      ydist = fabs(ybox[j] - ya) ;
      ydist = (double) min (ydist,system_size_y-ydist) ;

      xq = xdist*xdist ;
      yq = ydist*ydist ;
      distq = xq+yq+zq ;

      if (distq < maxq) {
        if (distq > minq) {
          if (nachb_anz[j] == 0) {
            counter ++ ;
            liste[counter] = j ;
          }
        }
      }
    }
  }

  /* counter enthaelt jetzt die Zahl der in liste stehenden Loesungsmittelteilchen */

  for (i=1;i<=counter;i++) {

    bw_e[i] = bond_winkel_potential(xa,ya,za,xdir[aktuelles_teilchen],
                                    ydir[aktuelles_teilchen],
                                    zdir[aktuelles_teilchen],
                                    xbox[liste[i]],ybox[liste[i]],zbox[liste[i]],
                                    xdir[liste[i]],ydir[liste[i]],zdir[liste[i]],
                                    nachb_anz[aktuelles_teilchen],0) ;
    /* Formal nicht ganz korrekt - liefert aber das richtige Ergebnis
       (nachb_anz[aktuelles_teilchen] ist hier immer Null */
    bl_e[i] = bond_laengen_potential(xa,ya,za,xbox[liste[i]],ybox[liste[i]],
                                     zbox[liste[i]]) ;

    if (k >1) {
      bw_e2[i] = bond_winkel_potential_2(xa,ya,za,xbox[alte_kette[(k-2)]],
                                         ybox[alte_kette[(k-2)]],
                                         zbox[alte_kette[(k-2)]],xbox[liste[i]],
                                         ybox[liste[i]],zbox[liste[i]]) ;
      energie[i] = bw_e[i] + bl_e[i] + bw_e2[i] ;
    } else {
      energie[i] = bw_e[i] + bl_e[i] ;
    }
  }

  help = 0 ;

  for (i=1;i<=counter;i++) {
    help = help + exp((-1)* (energie[i]/temp)) ;
  }

  /* Normierung */

  for (i=1;i<=counter;i++) {
    if (liste[i] == alte_kette[k]) {
      wahrsch[i] = (exp((-1)*(energie[i]/temp))) / help ;
      bond_wahrscheinlichkeit_alt[k] = wahrsch[i] ;
      ketten_wahrscheinlichkeit_alt = ketten_wahrscheinlichkeit_alt *
        bond_wahrscheinlichkeit_alt[k] ;
      aktuelles_teilchen = liste[i] ;
      break ;
    }
  }
}


/* ab hier neue Kette */

/* Kopf bleibt Kopf */

nachb_anz[hit_head] = 3 ;
aktuelles_teilchen = hit_head ;
neue_kette[0] = hit_head ;

for (k=1;k<laenge;k++) {
  counter = 0 ; /* Anzahl der gefunden moeglichen Teilchen fuer den k-ten Bond */
  xa = xbox[aktuelles_teilchen] ;
  ya = ybox[aktuelles_teilchen] ;
  za = zbox[aktuelles_teilchen] ;

  for (j=0;j <anzahl_teilchen;j++) {
    zdist = zbox[j] - za ;
    if (fabs(zdist) <= max) {
      zq = zdist * zdist ;

      xdist = fabs(xbox[j] - xa) ;
      xdist = (double) min (xdist,system_size_x-xdist) ;
      ydist = fabs(ybox[j] - ya) ;
      ydist = (double) min (ydist,system_size_y-ydist) ;

      xq = xdist*xdist ;
      yq = ydist*ydist ;
      distq = xq+yq+zq ;
```

```
    if (distq < maxq) {
      if (distq > minq) {
        if (nachb_anz[j] == 0) {
          counter ++ ;
          liste[counter] = j ;
        }
      }
    }
  }
}

/* counter enthaelt jetzt die Zahl der in liste stehenden Loesungsmittelteilchen */

for (i=1;i<=counter;i++) {
  bw_e[i] = bond_winkel_potential(xa,ya,za,xdir[aktuelles_teilchen],
                                  ydir[aktuelles_teilchen],
                                  zdir[aktuelles_teilchen],xbox[liste[i]],
                                  ybox[liste[i]],zbox[liste[i]],xdir[liste[i]],
                                  ydir[liste[i]],zdir[liste[i]],
                                  nachb_anz[aktuelles_teilchen],0) ;
  bl_e[i] = bond_laengen_potential(xa,ya,za,xbox[liste[i]],ybox[liste[i]],
                                   zbox[liste[i]]) ;

  if (k >1) {
    bw_e2[i] = bond_winkel_potential_2(xa,ya,za,xbox[nachb_l[aktuelles_teilchen]],
                                       ybox[nachb_l[aktuelles_teilchen]],
                                       zbox[nachb_l[aktuelles_teilchen]],
                                       xbox[liste[i]],ybox[liste[i]],
                                       zbox[liste[i]]) ;
    energie[i] = bw_e[i] + bl_e[i] + bw_e2[i] ;
  } else {
    energie[i] = bw_e[i] + bl_e[i] ;
  }
}

help = 0 ;

for (i=1;i<=counter;i++) {
  help = help + exp((-1)* (energie[i]/temp)) ;
}

/* Normierung */

for (i=1;i<=counter;i++) {
  wahrsch[i] = (exp((-1)*(energie[i]/temp))) / help ;
}


rnd = double_r250() ;
gefunden = 0 ;
for (i=1;i<=counter;i++) {
  if ((gefunden == 0) && (rnd<=wahrsch[i])) {
    /* Bond ist ausgewaehlt */
    gefunden = 1 ;
    /* Jetzt muessen wir den neuen Bond nur noch eintragen */
    bond_wahrscheinlichkeit_neu[k] = wahrsch[i] ;
    ketten_wahrscheinlichkeit_neu = ketten_wahrscheinlichkeit_neu *
      bond_wahrscheinlichkeit_neu[k] ;
    neue_kette[k] = liste[i] ;

    if (k==(laenge -1)) {
      nachb_anz[liste[i]] = 1 ;
    } else {
      nachb_anz[liste[i]] = 2 ;
    }
    nachb_r[aktuelles_teilchen] = liste[i] ;
    nachb_l[liste[i]] = aktuelles_teilchen ;
    if (k>1) {
      bond_energie[aktuelles_teilchen][2] = bw_e2[i] ;
    }
    bond_energie[liste[i]][0] = bw_e[i] + bl_e[i] ;
    bond_energie[aktuelles_teilchen][1] = bw_e[i] + bl_e[i] ;


    aktuelles_teilchen = liste[i] ;
    /* gefundenes Teilchen wird neuer "Kopf" */
    break ;

  } else {
    rnd = rnd - wahrsch[i] ;
  }
}


}

/* Jetzt haben wir eine neue Kette ! Wollen wir die auch behalten ? */


for (i=0;i<laenge;i++) {
  energie_alt += alte_ketten_energie[i][0] ;
  energie_alt += alte_ketten_energie[i][2] ;
}

for (i=0;i<laenge;i++) {
  energie_neu += bond_energie[neue_kette[i]][0] ;
  energie_neu += bond_energie[neue_kette[i]][2] ;
}


rnd = double_r250() ;


if (rnd < ((ketten_wahrscheinlichkeit_alt/ketten_wahrscheinlichkeit_neu)*
           exp(-(energie_neu-energie_alt)/temp))) {

  /* neue Kette wird akzeptiert */


  gesamt_energie = gesamt_energie + energie_neu - energie_alt ;


  neugleichalt = 0 ;

  for (i= 0;i<laenge;i++) {
    if (neue_kette[i] != alte_kette[i]) {
      neugleichalt = 1 ;
    }
  }

  return(0) ; /* Beenden mit neu gebauter Kette */

} else {
  /* alte Kette wird beibehalten */


  for (i=0;i<laenge;i++) {
    bond_energie[neue_kette[i]][0] = 0 ;
    bond_energie[neue_kette[i]][1] = 0 ;
    bond_energie[neue_kette[i]][2] = 0 ;
    nachb_anz[neue_kette[i]] = 0 ;
    nachb_r[neue_kette[i]] = INT_MAX ;
    nachb_l[neue_kette[i]] = INT_MAX ;
  }
  /* neue Kette wieder entfernen */

  for (i=0;i<laenge;i++) {
    bond_energie[alte_kette[i]][0] = alte_ketten_energie[i][0] ;
    bond_energie[alte_kette[i]][1] = alte_ketten_energie[i][1] ;
    bond_energie[alte_kette[i]][2] = alte_ketten_energie[i][2] ;
```

```
      nachb_anz[alte_kette[i]] = 2 ;
      nachb_r[alte_kette[i]] = alte_kette[i+1] ;
      nachb_l[alte_kette[i]] = alte_kette[i-1] ;

      if (i==0) {
        nachb_anz[alte_kette[i]] = 3 ;
        nachb_l[alte_kette[i]] = INT_MAX ;
      }
      if (i== (laenge -1)) {
        nachb_anz[alte_kette[i]] = 1 ;
        nachb_r[alte_kette[i]] = INT_MAX ;
      }
    }
    /* alte Kette wieder eingefuegt */

    return(1) ; /* Beenden mit alter Kette */
  }
}

/*-------------------------------------------------------------*/


int V_move()

/* Fuehrt einen Volumenmove in z-Richtung aus */

{
  double rnd,faktor ;
  double move_energie ;
  double energie_alt ;
  double system_size_z_test ;
  double ediff ;
  int i,j,k,l ;
  int tzahl ;
  double diff1,diff2 ;
  int zwei_i,zwei_ip1 ;
  int ix1,ix2,iy1,iy2,iz1,iz2 ;
  double neu ;
  double a,b ;
  double ausl_z ;



  for (i=0;i<anzahl_teilchen;i++) {
    bond_energie_test[i][0] = 0 ;
    bond_energie_test[i][1] = 0 ;
    bond_energie_test[i][2] = 0 ;
  }



  move_energie = 0. ;
  energie_alt = 0. ;

  rnd = (2*double_r250()-1)*max_zerr ;

  system_size_z_test = system_size_z + rnd ;

  faktor = system_size_z_test / system_size_z ;

  /* Skaliere z-Koordinate der Teilchen neu */

  if (faktor > 1.) {

    for (i=0 ;i<anzahl_teilchen ;i++) {
      zpos_test[i] = zpos[i] * faktor ;
      zbox_test[i] = zpos_test[i] ;
    }
  } else {

  for (i=0 ;i<anzahl_teilchen ;i++) {
    zpos_test[i] = zpos[i] * faktor ;
    zbox_test[i] = zpos_test[i] ;
    if ((zpos_test[i] < sigma_l) || (zpos_test[i] > (system_size_z-sigma_l))) {
      ausl_z =  sqrt(sigma_s_halbe_q + zdir[i] * zdir[i] *
                     (sigma_l_halbe_q-sigma_s_halbe_q))  ;
      /* Abstand des Schwerpunktes von einer Ebene parallel zur z-Achse */

      if ((nachb_anz[i] != 3) && ((zpos_test[i] - ausl_z <= 0) ||
                                  (zpos_test[i] + ausl_z  >= system_size_z))) {
        return(1) ;
      } /* Teilchen wurde durch Boxstauchung nach oben oder unten aus der Box
           herausgeschoben */
    }

  }

}

/* Berechne Energie neu ! */

/* Durch das Skalieren der z-Achse wechselwirken die Teilchen nicht mit mehr oder
   anderen Teilchen als vorher
   => nur die Nachbarteilchen betrachten, die in der Verlettabelle stehen ! */


for (i=0 ;i<anzahl_teilchen ;i++) {
  for (j=1 ;j<=tab_verlet[i][0] ;j++) {
    tzahl = tab_verlet[i][j] ;
    ene_help[i][j] = ww_energie(xbox[i],ybox[i],zbox_test[i],xdir[i],ydir[i],
                                zdir[i],xbox[tzahl],ybox[tzahl],zbox_test[tzahl],
                                xdir[tzahl],ydir[tzahl],zdir[tzahl],
                                nachb_anz[i],nachb_anz[tzahl]) ;

    move_energie += ene_help[i][j] ;
  }
}

move_energie = move_energie / 2. ;
/* alle WW werden doppelt gezaehlt => Korrektur ! */

for (i=0;i<anzahl_teilchen;i++) {
  if (nachb_anz[i] != 0) {
    if (nachb_l[i] != INT_MAX) {
      bond_energie_test[i][0] = 0.5*bond_laengen_potential(xbox[i],ybox[i],
                                                           zbox_test[i],
                                                           xbox[nachb_l[i]],
                                                           ybox[nachb_l[i]],
                                                           zbox_test[nachb_l[i]]) ;

      bond_energie_test[i][0] += 0.5*bond_winkel_potential(xbox[i],ybox[i],
                                                           zbox_test[i],xdir[i],
                                                           ydir[i],zdir[i],
                                                           xbox[nachb_l[i]],
                                                           ybox[nachb_l[i]],
                                                           zbox_test[nachb_l[i]],
                                                           xdir[nachb_l[i]],
                                                           ydir[nachb_l[i]],
                                                           zdir[nachb_l[i]],
                                                           nachb_anz[i],
                                                           nachb_anz[nachb_l[i]]) ;
      move_energie += bond_energie_test[i][0] ;
    }

    if (nachb_r[i] != INT_MAX) {
      bond_energie_test[i][1] = 0.5*bond_laengen_potential(xbox[i],ybox[i],
                                                           zbox_test[i],
                                                           xbox[nachb_r[i]],
                                                           ybox[nachb_r[i]],
                                                           zbox_test[nachb_r[i]]) ;
```

```
      bond_energie_test[i][1] += 0.5*bond_winkel_potential(xbox[i],ybox[i],
                                                          zbox_test[i],
                                                          xdir[i],ydir[i],zdir[i],
                                                          xbox[nachb_r[i]],
                                                          ybox[nachb_r[i]],
                                                          zbox_test[nachb_r[i]],
                                                          xdir[nachb_r[i]],
                                                          ydir[nachb_r[i]],
                                                          zdir[nachb_r[i]],
                                                          nachb_anz[i],
                                                          nachb_anz[nachb_r[i]]) ;
      move_energie += bond_energie_test[i][1] ;
    }
    if (nachb_anz[i] == 2) {
      bond_energie_test[i][2] = bond_winkel_potential_2(xbox[i],ybox[i],
                                                        zbox_test[i],
                                                        xbox[nachb_l[i]],
                                                        ybox[nachb_l[i]],
                                                        zbox_test[nachb_l[i]],
                                                        xbox[nachb_r[i]],
                                                        ybox[nachb_r[i]],
                                                        zbox_test[nachb_r[i]]) ;
      move_energie += bond_energie_test[i][2] ;
    }
  }

}


/* Ermittele alte Energie */

energie_alt = gesamt_energie ;


/* Berechne Energiedifferenz */

ediff = move_energie - energie_alt +
  /* + p * dV */
  (press * (system_size_z_test - system_size_z) * system_size_x * system_size_y) -
  /* - N * k_B * T * ln(Kantenlaenge_neu / Kantenlaenge_alt) */
  (anzahl_teilchen * log(faktor) * temp) ;


/* Metropolis - Test */

if (ediff/temp > 70.) {
  return(1) ;
}

if (ediff > 0.){
  if (double_r250() > exp(-ediff/temp)) {
    return(1) ;
  }
}

/* Move akzeptiert */

gesamt_energie = move_energie ;

/* Aktualisiere Koordinaten ,Linked-Cells-Struktur und WW - Tabelle */

cell_size_z = cell_size_z * faktor ;
system_size_z = system_size_z_test ;
system_size_z_real = zellen_z * cell_size_z ;
system_size_z_real_halbe = system_size_z_real * 0.5 ;

for (i=0 ;i<anzahl_teilchen ;i++) {

  zwei_i = 2*i ;
  zwei_ip1 = zwei_i + 1 ;

  zpos[i] = zpos_test[i] ;
  zbox[i] = zbox_test[i] ;

  pseudo_xpos[zwei_i] = xpos[i] + 0.5*xdir[i]*sigma_l_halbe ;
  pseudo_xpos[zwei_ip1] = xpos[i] - 0.5*xdir[i]*sigma_l_halbe ;
  pseudo_ypos[zwei_i] = ypos[i] + 0.5*ydir[i]*sigma_l_halbe ;
  pseudo_ypos[zwei_ip1] = ypos[i] - 0.5*ydir[i]*sigma_l_halbe ;
  pseudo_zpos[zwei_i] = zpos[i] + 0.5*zdir[i]*sigma_l_halbe ;
  pseudo_zpos[zwei_ip1] = zpos[i] - 0.5*zdir[i]*sigma_l_halbe ;

  pseudo_xbox[zwei_i] = fmod((pseudo_xpos[zwei_i] +50*system_size_x),
                             system_size_x) ;
  pseudo_xbox[zwei_ip1] = fmod((pseudo_xpos[zwei_ip1] +50*system_size_x),
                               system_size_x) ;
  pseudo_ybox[zwei_i] = fmod((pseudo_ypos[zwei_i] +50*system_size_y),
                             system_size_y) ;
  pseudo_ybox[zwei_ip1] = fmod((pseudo_ypos[zwei_ip1] +50*system_size_y),
                               system_size_y) ;
  if (pseudo_zpos[zwei_i] >0) {
    pseudo_zbox[zwei_i] = pseudo_zpos[zwei_i] ;
  } else {
    pseudo_zbox[zwei_i] = system_size_z_real + pseudo_zpos[zwei_i] ;
  }
 if (pseudo_zpos[zwei_ip1] >0) {
    pseudo_zbox[zwei_ip1] = pseudo_zpos[zwei_ip1] ;
  } else {
    pseudo_zbox[zwei_ip1] = system_size_z_real + pseudo_zpos[zwei_ip1] ;
  }
}


for(i=0 ;i<anzahl_teilchen ;i++) {
  for (j=1 ;j<=tab_verlet[i][0] ;j++) {
    ene_verlet[i][j] = ene_help[i][j] ;
  }
}

for(i=0;i<anzahl_teilchen;i++) {
  bond_energie[i][0] = 2*bond_energie_test[i][0] ;
  bond_energie[i][1] = 2*bond_energie_test[i][1] ;
  bond_energie[i][2] = bond_energie_test[i][2] ;
}


for (i=0 ;i<anzahl_teilchen ;i++) {

  zwei_i = 2*i ;
  zwei_ip1 = zwei_i + 1 ;

  diff1 = (pseudo_xpos[zwei_i]-pseudo_xpos_0v[zwei_i]) *
    (pseudo_xpos[zwei_i]-pseudo_xpos_0v[zwei_i]) +
    (pseudo_ypos[zwei_i]-pseudo_ypos_0v[zwei_i]) *
    (pseudo_ypos[zwei_i]-pseudo_ypos_0v[zwei_i]) +
    (pseudo_zpos[zwei_i]-pseudo_zpos_0v[zwei_i]) *
    (pseudo_zpos[zwei_i]-pseudo_zpos_0v[zwei_i]) ;

  diff2 = (pseudo_xpos[zwei_ip1]-pseudo_xpos_0v[zwei_ip1]) *
    (pseudo_xpos[zwei_ip1]-pseudo_xpos_0v[zwei_ip1]) +
    (pseudo_ypos[zwei_ip1]-pseudo_ypos_0v[zwei_ip1]) *
    (pseudo_ypos[zwei_ip1]-pseudo_ypos_0v[zwei_ip1]) +
    (pseudo_zpos[zwei_ip1]-pseudo_zpos_0v[zwei_ip1]) *
    (pseudo_zpos[zwei_ip1]-pseudo_zpos_0v[zwei_ip1]) ;

  if (diff2 > diff1) {
    diff1 = diff2 ;
  }


  if (diff1>diff_max2) {
    if (diff1>diff_max) {
      diff_max2 = diff_max ;
```

```
        diff_max  = diff1 ;
      } else {
        diff_max2 = diff1 ;
      }
    }
  }

  /* Verlet-Tabelle wird upgedatet wenn die beiden groessten        */
  /* Monomer-Drifts von Pseudoteilchen seit letztem Update zusammen */
  /* die Schichtdicke  skin_verlet_check_update  ueberschreiten.    */

  if ((sqrt(diff_max2)+sqrt(diff_max)) >= skin_verlet_check_update) {
    /*      printf("UPDATE !!!!!! %lf %lf \n",diff_max,diff_max2) ;
     */
    verlet_update() ;
  }

  return(0) ;

}


/*-------------------------------------------------------------*/

int rot()

     /* dreht ein zufaellig ausgewaehltes Teilchen um einen zufaelligen Winkel */
     /* Durch Rotation eines Teilchens wird bond_winkel_potential_2 nicht beruehrt
        - kann also voellig ignoriert werden */

{

  int hitm,hitm2,hitm2p1 ;
  double rnd,u,v,w ;
  double zdir_neu,ydir_neu,xdir_neu,energie_alt,energie_neu,ediff ;
  int ix1,ix2,iy1,iy2,iz1,iz2,j ;
  double ausl_z,help1,help2,diff1,diff2  ;
  char check ;
  int ergebnis ;
  double help ;


  ergebnis = 0 ;
  energie_alt = 0. ;
  help1 = 0. ;
  help2 = 0. ;
  u = 2. ;
  v = 2. ;
  w = 2. ;

  /* Waehle zufaelliges Teilchen aus, das bewegt werden soll */

  hitm = (int) (double_r250()*anzahl_teilchen);

  if (nachb_anz[hitm] > 0) ergebnis = 2 ;

  hitm2 = 2 * hitm ;
  hitm2p1 = hitm2 + 1 ;

  /* fuehre Zufallsbewegung fuer gewaehltes Monomer aus */


  /* Suche Vektor innerhalb Einheitskugel */

  while  (u*u+v*v+w*w > 1) {

    u = 2.*double_r250()-1.;

    v = 2.*double_r250()-1.;

    w = 2.*double_r250()-1.;
```

```
}


xdir_neu = xdir[hitm] + max_rotate * (ydir[hitm]*w -zdir[hitm]*v) ;
ydir_neu = ydir[hitm] + max_rotate * (zdir[hitm]*u -xdir[hitm]*w) ;
zdir_neu = zdir[hitm] + max_rotate * (xdir[hitm]*v -ydir[hitm]*u) ;


/* Normierung des neuen Vektors */


help = 1./sqrt(xdir_neu*xdir_neu + ydir_neu*ydir_neu + zdir_neu*zdir_neu) ;

xdir_neu = xdir_neu * help ;
ydir_neu = ydir_neu * help ;
zdir_neu = zdir_neu * help ;


ausl_z =  sqrt(sigma_s_halbe_q + zdir_neu * zdir_neu *
                (sigma_l_halbe_q-sigma_s_halbe_q))  ;
/* Abstand des Schwerpunktes von einer Ebene parallel zur z-Achse */

if ((nachb_anz[hitm] != 3) && ((zpos[hitm] - ausl_z <= 0) ||
                               (zpos[hitm] + ausl_z  >= system_size_z))) {
  return(ergebnis+1) ;
} /* Teilchen wurde nach oben oder unten aus der Box herausbewegt bzw. gedreht */


 /* Berechne neue SC-Energie des Teilchens */

energie_neu = calculate_energie(xbox[hitm],ybox[hitm],zbox[hitm],xdir_neu,
                                ydir_neu,zdir_neu,hitm) ;

/* Berechne neue Bondwinkel-und Bondlaegenenergien */

if (nachb_anz[hitm] != 0) {
  if (nachb_l[hitm] != INT_MAX) {
    help1 = bond_laengen_potential(xbox[hitm],ybox[hitm],zbox[hitm],
                                   xbox[nachb_l[hitm]],ybox[nachb_l[hitm]],
                                   zbox[nachb_l[hitm]]) ;

    help1 += bond_winkel_potential(xbox[hitm],ybox[hitm],zbox[hitm],xdir_neu,
                                   ydir_neu,zdir_neu,xbox[nachb_l[hitm]],
                                   ybox[nachb_l[hitm]],zbox[nachb_l[hitm]],
                                   xdir[nachb_l[hitm]],ydir[nachb_l[hitm]],
                                   zdir[nachb_l[hitm]],nachb_anz[hitm],
                                   nachb_anz[nachb_l[hitm]]) ;
    energie_neu += help1 ;
  }

  if (nachb_r[hitm] != INT_MAX) {
    help2 = bond_laengen_potential(xbox[hitm],ybox[hitm],zbox[hitm],
                                   xbox[nachb_r[hitm]],ybox[nachb_r[hitm]],
                                   zbox[nachb_r[hitm]]) ;

    help2 += bond_winkel_potential(xbox[hitm],ybox[hitm],zbox[hitm],xdir_neu,
                                   ydir_neu,zdir_neu,xbox[nachb_r[hitm]],
                                   ybox[nachb_r[hitm]],zbox[nachb_r[hitm]],
                                   xdir[nachb_r[hitm]],ydir[nachb_r[hitm]],
                                   zdir[nachb_r[hitm]],nachb_anz[hitm],
                                   nachb_anz[nachb_r[hitm]]) ;
    energie_neu += help2 ;
  }
}


/* Ermittele alte Energie */

for (j=1;j<=tab_verlet[hitm][0];j++) {
  energie_alt += ene_verlet[hitm][j] ;
```

```
}

/* energie_alt = get_energie(hitm) ;*/

energie_alt += bond_energie[hitm][0] ;
energie_alt += bond_energie[hitm][1] ;



ediff = energie_neu - energie_alt ;

/* Metropolisabfrage  */

if (ediff/temp > 70.) {
  return(ergebnis+1);
}

if (ediff > 0.){
  if (double_r250() > exp(-ediff/temp)) {
    return(ergebnis+1);
  }
}

/* Move akzeptiert !!!! */

put_energie(hitm) ;

if (nachb_l[hitm] != INT_MAX) {
  bond_energie[hitm][0] = help1 ;
  bond_energie[nachb_l[hitm]][1] = help1 ;
}

if (nachb_r[hitm] != INT_MAX) {
  bond_energie[hitm][1] = help2 ;
  bond_energie[nachb_r[hitm]][0] = help2 ;
}

gesamt_energie = gesamt_energie + ediff ;


/* Berechne die neue Positionen der dem Teilchen zugeordneten Pseudoteilchen */

pseudo_xpos[hitm2] = xpos[hitm] + 0.5*xdir_neu*sigma_l_halbe ;
pseudo_xpos[hitm2p1] = xpos[hitm] - 0.5*xdir_neu*sigma_l_halbe ;
pseudo_ypos[hitm2] = ypos[hitm] + 0.5*ydir_neu*sigma_l_halbe ;
pseudo_ypos[hitm2p1] = ypos[hitm] - 0.5*ydir_neu*sigma_l_halbe ;
pseudo_zpos[hitm2] = zpos[hitm] + 0.5*zdir_neu*sigma_l_halbe ;
pseudo_zpos[hitm2p1] = zpos[hitm] - 0.5*zdir_neu*sigma_l_halbe ;

pseudo_xbox[hitm2] = fmod((pseudo_xpos[hitm2] +50*system_size_x),system_size_x) ;
pseudo_xbox[hitm2p1] = fmod((pseudo_xpos[hitm2p1] +50*system_size_x),system_size_x) ;
pseudo_ybox[hitm2] = fmod((pseudo_ypos[hitm2] +50*system_size_y),system_size_y) ;
pseudo_ybox[hitm2p1] = fmod((pseudo_ypos[hitm2p1] +50*system_size_y),system_size_y) ;
if (pseudo_zpos[hitm2] >0) {
  pseudo_zbox[hitm2] = pseudo_zpos[hitm2] ;
} else {
  pseudo_zbox[hitm2] = system_size_z_real + pseudo_zpos[hitm2] ;
}
if (pseudo_zpos[hitm2p1] >0) {
  pseudo_zbox[hitm2p1] = pseudo_zpos[hitm2p1] ;
} else {
  pseudo_zbox[hitm2p1] = system_size_z_real + pseudo_zpos[hitm2p1] ;
}

xdir[hitm] = xdir_neu ;
ydir[hitm] = ydir_neu ;
zdir[hitm] = zdir_neu ;

ix1 = (int) (pseudo_xbox[hitm2]/cell_size_x) ;
ix2 = (int) (pseudo_xbox[hitm2p1]/cell_size_x) ;
iy1 = (int) (pseudo_ybox[hitm2]/cell_size_y) ;
iy2 = (int) (pseudo_ybox[hitm2p1]/cell_size_y) ;
iz1 = (int) (pseudo_zbox[hitm2]/cell_size_z) ;
iz2 = (int) (pseudo_zbox[hitm2p1]/cell_size_z) ;

 /* Hat Pseudoteilchen die Zelle gewechselt ???? */

if ((ix1!=x_zelle[hitm2])||(iy1!=y_zelle[hitm2])||(iz1!=z_zelle[hitm2]))
  update_pointers(hitm2,ix1,iy1,iz1) ;

if ((ix2!=x_zelle[hitm2p1])||(iy2!=y_zelle[hitm2p1])||(iz2!=z_zelle[hitm2p1]))
  update_pointers(hitm2p1,ix2,iy2,iz2) ;


/* evtl. Verlet-Tabelle updaten (nur komplett, denn andere   */
/* Monomere wurden vermutl. auch schon bewegt !)             */

diff1 = (pseudo_xpos[hitm2]-pseudo_xpos_0v[hitm2]) *
  (pseudo_xpos[hitm2]-pseudo_xpos_0v[hitm2]) +
  (pseudo_ypos[hitm2]-pseudo_ypos_0v[hitm2]) *
  (pseudo_ypos[hitm2]-pseudo_ypos_0v[hitm2]) +
  (pseudo_zpos[hitm2]-pseudo_zpos_0v[hitm2]) *
  (pseudo_zpos[hitm2]-pseudo_zpos_0v[hitm2]) ;

diff2 = (pseudo_xpos[hitm2p1]-pseudo_xpos_0v[hitm2p1]) *
  (pseudo_xpos[hitm2p1]-pseudo_xpos_0v[hitm2p1]) +
  (pseudo_ypos[hitm2p1]-pseudo_ypos_0v[hitm2p1]) *
  (pseudo_ypos[hitm2p1]-pseudo_ypos_0v[hitm2p1]) +
  (pseudo_zpos[hitm2p1]-pseudo_zpos_0v[hitm2p1]) *
  (pseudo_zpos[hitm2p1]-pseudo_zpos_0v[hitm2p1]) ;

if (diff2 > diff1) {
  diff1 = diff2 ;
}

if (diff1>diff_max2) {
  if (diff1>diff_max) {
    diff_max2 = diff_max ;
    diff_max  = diff1 ;
  } else {
    diff_max2 = diff1 ;
  }

  /* Verlet-Tabelle wird upgedatet wenn die beiden groessten        */
  /* Monomer-Drifts von Pseudoteilchen seit letztem Update zusammen */
  /* die Schichtdicke  skin_verlet_check_update  ueberschreiten.    */

  if ((sqrt(diff_max2)+sqrt(diff_max)) >= skin_verlet_check_update) {
/*      printf("UPDATE !!!!!! %lf %lf \n",diff_max,diff_max2) ;
    fflush ;
*/
    verlet_update();
  }

}

return(ergebnis) ;
}


/*------------------------------------------------------------*/

int move()

/* verschiebt ein zufaellig gewaehltes Teilchen in jede Raumrichtung um einen
   jeweils zufaelligen Wert */

{

  int hitm,hitm2,hitm2p1 ;
  double zpos_neu,ypos_neu,xpos_neu,zbox_neu,ybox_neu,xbox_neu ;
  double energie_alt,energie_neu,ediff ;
```

```
int ix1,ix2,iy1,iy2,iz1,iz2,j ;
double ausl_z,help1,help2,help3,help4,help5,diff1,diff2  ;
char check ;
double xdiff,ydiff,zdiff ;
int ergebnis ;
int marke1,marke2 ;


ergebnis = 0 ;
energie_alt = 0. ;
help1 = 0. ;
help2 = 0. ;
help3 = 0. ;
help4 = 0. ;
help5 = 0. ;
marke1 = 0 ;
marke2 = 0 ;


/* Waehle zufaelliges Teilchen aus, das bewegt werden soll */

hitm = (int) (double_r250()*anzahl_teilchen);

if (nachb_anz[hitm] > 0) ergebnis = 2 ;

if (nachb_anz[hitm] == 3) return(3) ;
/* Kopfmonomere koennen nicht bewegt werden */


hitm2 = 2 * hitm ;
hitm2p1 = hitm2 + 1 ;


/* fuehre Zufallsbewegung fuer gewaehltes Monomer aus */

zdiff = (2.*double_r250()-1.) * max_jump ;
zpos_neu = zpos[hitm] + zdiff ;

ydiff = (2.*double_r250()-1.) * max_jump ;
ypos_neu = ypos[hitm] + ydiff ;

xdiff = (2.*double_r250()-1.) * max_jump ;
xpos_neu = xpos[hitm] + xdiff ;

xbox_neu = fmod((xpos_neu +50*system_size_x),system_size_x) ;
ybox_neu = fmod((ypos_neu +50*system_size_y),system_size_y) ;
zbox_neu = zpos_neu ;

ausl_z =  sqrt(sigma_s_halbe_q + zdir[hitm] * zdir[hitm] *
          (sigma_l_halbe_q-sigma_s_halbe_q))  ;
/* Abstand des Schwerpunktes von einer Ebene parallel zur z-Achse */

if ((zpos_neu - ausl_z <= 0) || (zpos_neu + ausl_z  >= system_size_z)) {
  return(ergebnis + 1) ;
} /* Teilchen wurde nach oben oder unten aus der Box herausbewegt */


/* Berechne neue SC-Energie des Teilchens */

energie_neu = calculate_energie(xbox_neu,ybox_neu,zbox_neu,xdir[hitm],ydir[hitm],
                                zdir[hitm],hitm) ;

/* Berechne neue Bondwinkel-und Bondlaegenenergien */

if (nachb_anz[hitm] != 0) {
  if (nachb_l[hitm] != INT_MAX) {
    help1 = bond_laengen_potential(xbox_neu,ybox_neu,zbox_neu,
                                   xbox[nachb_l[hitm]],ybox[nachb_l[hitm]],
                                   zbox[nachb_l[hitm]]) ;

    help1 += bond_winkel_potential(xbox_neu,ybox_neu,zbox_neu,xdir[hitm],
                                   ydir[hitm],zdir[hitm],xbox[nachb_l[hitm]],
                                   ybox[nachb_l[hitm]],zbox[nachb_l[hitm]],
                                   xdir[nachb_l[hitm]],ydir[nachb_l[hitm]],
                                   zdir[nachb_l[hitm]],nachb_anz[hitm],
                                   nachb_anz[nachb_l[hitm]]) ;
    energie_neu += help1 ;

    if (nachb_l[nachb_l[hitm]] != INT_MAX) {
      marke1 = 1 ;
      help4 = bond_winkel_potential_2(xbox[nachb_l[hitm]],ybox[nachb_l[hitm]],
                                      zbox[nachb_l[hitm]],
                                      xbox[nachb_l[nachb_l[hitm]]],
                                      ybox[nachb_l[nachb_l[hitm]]],
                                      zbox[nachb_l[nachb_l[hitm]]],
                                      xbox_neu,ybox_neu,zbox_neu) ;
      energie_neu += help4 ;
    }

  }

  if (nachb_r[hitm] != INT_MAX) {
    help2 = bond_laengen_potential(xbox_neu,ybox_neu,zbox_neu,
                                   xbox[nachb_r[hitm]],ybox[nachb_r[hitm]],
                                   zbox[nachb_r[hitm]]) ;

    help2 += bond_winkel_potential(xbox_neu,ybox_neu,zbox_neu,xdir[hitm],
                                   ydir[hitm],zdir[hitm],xbox[nachb_r[hitm]],
                                   ybox[nachb_r[hitm]],zbox[nachb_r[hitm]],
                                   xdir[nachb_r[hitm]],ydir[nachb_r[hitm]],
                                   zdir[nachb_r[hitm]],nachb_anz[hitm],
                                   nachb_anz[nachb_r[hitm]]) ;
    energie_neu += help2 ;

    if (nachb_r[nachb_r[hitm]] != INT_MAX) {
      marke2 = 1 ;
      help5 = bond_winkel_potential_2(xbox[nachb_r[hitm]],ybox[nachb_r[hitm]],
                                      zbox[nachb_r[hitm]],xbox_neu,ybox_neu,
                                      zbox_neu,xbox[nachb_r[nachb_r[hitm]]],
                                      ybox[nachb_r[nachb_r[hitm]]],
                                      zbox[nachb_r[nachb_r[hitm]]]) ;
      energie_neu += help5 ;
    }
  }
  if (nachb_anz[hitm] == 2) {
    help3 = bond_winkel_potential_2(xbox_neu,ybox_neu,zbox_neu,xbox[nachb_l[hitm]],
                                    ybox[nachb_l[hitm]],zbox[nachb_l[hitm]],
                                    xbox[nachb_r[hitm]],ybox[nachb_r[hitm]],
                                    zbox[nachb_r[hitm]]) ;
    energie_neu += help3 ;
  }

}

/* Ermittele alte Energie */

for (j=1;j<=tab_verlet[hitm][0];j++) {
  energie_alt += ene_verlet[hitm][j] ;
}

energie_alt += bond_energie[hitm][0] ;
energie_alt += bond_energie[hitm][1] ;
energie_alt += bond_energie[hitm][2] ;
if(marke1 == 1) {
  energie_alt += bond_energie[nachb_l[hitm]][2] ;
}
if(marke2 == 1) {
  energie_alt += bond_energie[nachb_r[hitm]][2] ;
}

ediff = energie_neu - energie_alt ;
```

```
/* Metropolisabfrage  */

if (ediff/temp > 70.) {
  return(ergebnis+1);
}

if (ediff > 0.){
  if (double_r250() > exp(-ediff/temp)) {
    return(ergebnis+1);
  }
}

/* Move akzeptiert !!!! */

put_energie(hitm) ;


if (nachb_l[hitm] != INT_MAX) {
  bond_energie[hitm][0] = help1 ;
  bond_energie[nachb_l[hitm]][1] = help1 ;
}

if (nachb_r[hitm] != INT_MAX) {
  bond_energie[hitm][1] = help2 ;
  bond_energie[nachb_r[hitm]][0] = help2 ;
}

if (nachb_anz[hitm] == 2) {
  bond_energie[hitm][2] = help3 ;
}

if (marke1 == 1) {
  bond_energie[nachb_l[hitm]][2] = help4 ;
}

if (marke2 == 1) {
  bond_energie[nachb_r[hitm]][2] = help5 ;
}


gesamt_energie = gesamt_energie + ediff ;


/* Berechne die neuen Positionen der dem Teilchen zugeordneten Pseudoteilchen */

pseudo_xpos[hitm2] = pseudo_xpos[hitm2] +xdiff ;
pseudo_xpos[hitm2p1] = pseudo_xpos[hitm2p1] +xdiff ;
pseudo_ypos[hitm2] = pseudo_ypos[hitm2] +ydiff ;
pseudo_ypos[hitm2p1] = pseudo_ypos[hitm2p1] +ydiff ;
pseudo_zpos[hitm2] = pseudo_zpos[hitm2] +zdiff ;
pseudo_zpos[hitm2p1] = pseudo_zpos[hitm2p1] +zdiff ;



pseudo_xbox[hitm2] = fmod((pseudo_xpos[hitm2] +50*system_size_x),system_size_x) ;
pseudo_xbox[hitm2p1] = fmod((pseudo_xpos[hitm2p1] +50*system_size_x),system_size_x) ;
pseudo_ybox[hitm2] = fmod((pseudo_ypos[hitm2] +50*system_size_y),system_size_y) ;
pseudo_ybox[hitm2p1] = fmod((pseudo_ypos[hitm2p1] +50*system_size_y),system_size_y) ;

if (pseudo_zpos[hitm2] >0) {
  pseudo_zbox[hitm2] = pseudo_zpos[hitm2] ;
} else {
  pseudo_zbox[hitm2] = system_size_z_real + pseudo_zpos[hitm2] ;
}
if (pseudo_zpos[hitm2p1] >0) {
  pseudo_zbox[hitm2p1] = pseudo_zpos[hitm2p1] ;
} else {
  pseudo_zbox[hitm2p1] = system_size_z_real + pseudo_zpos[hitm2p1] ;
}

  xpos[hitm] = xpos_neu ;
  ypos[hitm] = ypos_neu ;
  zpos[hitm] = zpos_neu ;
  xbox[hitm] = xbox_neu ;
  ybox[hitm] = ybox_neu ;
  zbox[hitm] = zbox_neu ;

  ix1 = (int) (pseudo_xbox[hitm2]/cell_size_x) ;
  ix2 = (int) (pseudo_xbox[hitm2p1]/cell_size_x) ;
  iy1 = (int) (pseudo_ybox[hitm2]/cell_size_y) ;
  iy2 = (int) (pseudo_ybox[hitm2p1]/cell_size_y) ;
  iz1 = (int) (pseudo_zbox[hitm2]/cell_size_z) ;
  iz2 = (int) (pseudo_zbox[hitm2p1]/cell_size_z) ;


  /* Hat Pseudoteilchen die Zelle gewechselt ???? */

  if ((ix1!=x_zelle[hitm2])||(iy1!=y_zelle[hitm2])||(iz1!=z_zelle[hitm2]))
    update_pointers(hitm2,ix1,iy1,iz1) ;

  if ((ix2!=x_zelle[hitm2p1])||(iy2!=y_zelle[hitm2p1])||(iz2!=z_zelle[hitm2p1]))
    update_pointers(hitm2p1,ix2,iy2,iz2) ;



  /* evtl. Verlet-Tabelle updaten (nur komplett, denn andere   */
  /* Monomere wurden vermutl. auch schon bewegt !)             */

  diff1 = (pseudo_xpos[hitm2]-pseudo_xpos_0v[hitm2]) *
    (pseudo_xpos[hitm2]-pseudo_xpos_0v[hitm2]) +
    (pseudo_ypos[hitm2]-pseudo_ypos_0v[hitm2]) *
    (pseudo_ypos[hitm2]-pseudo_ypos_0v[hitm2]) +
    (pseudo_zpos[hitm2]-pseudo_zpos_0v[hitm2]) *
    (pseudo_zpos[hitm2]-pseudo_zpos_0v[hitm2]) ;

  diff2 = (pseudo_xpos[hitm2p1]-pseudo_xpos_0v[hitm2p1]) *
    (pseudo_xpos[hitm2p1]-pseudo_xpos_0v[hitm2p1]) +
    (pseudo_ypos[hitm2p1]-pseudo_ypos_0v[hitm2p1]) *
    (pseudo_ypos[hitm2p1]-pseudo_ypos_0v[hitm2p1]) +
    (pseudo_zpos[hitm2p1]-pseudo_zpos_0v[hitm2p1]) *
    (pseudo_zpos[hitm2p1]-pseudo_zpos_0v[hitm2p1]) ;

  if (diff2 > diff1) {
    diff1 = diff2 ;
  }

  if (diff1>diff_max2) {
    if (diff1>diff_max) {
      diff_max2 = diff_max ;
      diff_max  = diff1 ;
    } else {
      diff_max2 = diff1 ;
    }

    /* Verlet-Tabelle wird upgedatet wenn die beiden groessten        */
    /* Monomer-Drifts von Pseudoteilchen seit letztem Update zusammen */
    /* die Schichtdicke  skin_verlet_check_update  ueberschreiten.    */

    if ((sqrt(diff_max2)+sqrt(diff_max)) >= skin_verlet_check_update) {
/*      printf("UPDATE !!!!!! %lf %lf \n",diff_max,diff_max2) ;
      fflush ;
*/
      verlet_update();
    }

  }

  return(ergebnis) ;
}
```

```
/*------------------------------------------------------------*/

void redefine_neighbor_cells()

/* definiert die Nachbarzellen innerhalb der link-cell-Struktur neu. Wird eigentlich
   nur benoetigt, wenn sich die Anzahl der Zellen aendert. */

{
  int i;

  for (i=0;i<zellen_x;i++){
    jpx[i] = (i+1+3*zellen_x)%zellen_x ;
  }
  for (i=0;i<zellen_x;i++){
    jmx[i] = (i-1+3*zellen_x)%zellen_x ;
  }
  for (i=0;i<zellen_y ;i++){
    jpy[i] = (i+1+3*zellen_y)%zellen_y ;
  }
  for (i=0;i<zellen_y ;i++){
    jmy[i] = (i-1+3*zellen_y)%zellen_y ;
  }
  for (i=0;i<zellen_z;i++){
    jpz[i] = (i+1+3*zellen_z)%zellen_z ;
  }
  for (i=0;i<zellen_z;i++){
    jmz[i] = (i-1+3*zellen_z)%zellen_z ;
  }
}


/*------------------------------------------------------------*/


void optimize_skin(double T_so_alt, double T_so_neu)

#define INCMAX 1.10
#define DECMAX 0.90

/* versucht das Programm duch Variation der Verlet-Schichtdicke moeglichst flott
   laufen zu lassen */


{ static double fact = INCMAX;
  static int sdc = 0;
  double skin_try,fact_old;
  int ncx,ncy,ncz,i1,nx,ny,nz,sdc_old;

  sdc_old = sdc;  fact_old = fact;

  if (T_so_neu>T_so_alt) {
    /* Programm wurde durch letzte Skin-Aenderung langsamer !         */
    /* 2. Richtung der Aenderung umkehren                             */
    /* 1. Naehe des Optimums ==> Schrittweite verkleinern             */
    sdc = 0;
    if ((fact>1.005)||(fact<0.995)) {
      fact = sqrt(fact);
    }

    fact = 1.0/fact;
  }
  else {
    sdc++;
    /* dreimal in gleiche Richtung --> Vergroesserung des Faktors     */
    if (sdc>2)
      if ((fact<INCMAX)&&(fact>DECMAX)) {
        fact = fact*fact;
      }
  }

  /* Neue Skin-Dicke berechnen                                        */

  skin_try = skin_verlet*fact;

  ncx = floor(system_size_x/(cutoff_max+skin_try));
  ncy = floor(system_size_y/(cutoff_max+skin_try));
  ncz = floor(system_size_z/(cutoff_max+skin_try));

  if ((ncx>=2)&&(ncy>=2)&&(ncz>=2)) {

    cell_size_x = system_size_x/ncx;
    cell_size_y = system_size_y/ncy;
    cell_size_z = system_size_z/ncz;
    ncz = ncz +3 ;

    /* Neue Skin-Dicke verarbeiten                                    */
    if ((ncx != zellen_x)||(ncy != zellen_y)||(ncz != zellen_z)) {
      zellen_x = ncx;
      zellen_y = ncy;
      zellen_z = ncz;
      redefine_neighbor_cells();
    }

    system_size_z_real = zellen_z * cell_size_z ;
    system_size_z_real_halbe = system_size_z_real * 0.5 ;

    skin_verlet = lj_safety = skin_try;
    /* printf("%lf %lf %lf %lf\n",mcs,T_so_neu,skin_verlet,fact); */

    for (i1 = 0; i1<anzahl_teilchen_mal2; i1++) {
      if (pseudo_zpos[i1] < 0)
        pseudo_zbox[i1] = system_size_z_real + pseudo_zpos[i1] ;
      nx = pseudo_xbox[i1]/cell_size_x;
      ny = pseudo_ybox[i1]/cell_size_y;
      nz = pseudo_zbox[i1]/cell_size_z;
      if ((nx != x_zelle[i1])||(ny != y_zelle[i1])||(nz != z_zelle[i1]))
        update_pointers(i1,nx,ny,nz);
    }

    verlet_update();
    verlet_update_counter--;
  }
  else {
    fact = fact_old;
    sdc = sdc_old;
  }
  skin_verlet_check_update = skin_verlet - max_move ;

}
/*------------------------------------------------------------*/

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

{
  long   step ;
  double mcs,mcs0,mcstotal,mcsave ;
  double akz_rate_move,akz_rate_rot,zellen ;
  double akz_rate_move_kette,akz_rate_rot_kette ;
  double akz_rate_V_move, akz_rate_ketten_move,akz_rate_ketten_move_same ;
  long   move_counter ; /* zaehlt erfolgreiche Bewegungsversuche */
  long   rot_counter ;
  long   rot_counter_kette ;
  long   move_counter_kette ;
  long   ketten_move_counter ;
  long   ketten_move_counter_same ;
  long   V_move_counter ;
  long   norm, itime ;
  int    samples,autoskin ;
  struct tms u_start,u_end ;
  struct tms so_start,so_end ;
  double T_so_alt,T_so_neu ;
  time_t t_start,t_end;
  double wall_secs,CPU_secs;
```

```
FILE   *Pro ;
int    i,j ;
double zahl ;
double oops,a,b ;
int    hilfe ;
long   lc_move_moves,ketten_move_moves ;
long   lc_rot_moves,ketten_rot_moves ;
long   mcs_laenge,mcs_loop ;
double rnd ;
double rot_border,move_border,V_move_border ;
long   move_try,rot_try,V_move_try,ketten_move_try ;


double alt_energie ;

/*  INITIALISIERUNG  */

Pro = fopen("skin","r");
fscanf(Pro,"%lf",&skin_verlet);
fscanf(Pro,"%d",&autoskin);
fclose(Pro);
printf(" > Verlet skin set to   : %7.5lf",skin_verlet);
if (autoskin) printf("  automatic\n");
else printf("  fixed\n");

lj_safety = skin_verlet ;

sflag = argv ;


lc_move_moves = 0 ;
ketten_move_moves = 0 ;
lc_rot_moves = 0 ;
ketten_rot_moves = 0 ;
rot_counter = 0 ;
move_counter = 0 ;
rot_counter_kette = 0 ;
move_counter_kette = 0 ;
V_move_counter = 0 ;
ketten_move_counter = 0 ;
ketten_move_counter_same = 0 ;
savestep = 500 ;
samples = 0 ;
mcs = 0 ;
mcs0 = 0 ;
mcstotal = 0 ;
zellen = 0 ;
move_try = 0 ;
rot_try = 0 ;
V_move_try = 0 ;
ketten_move_try = 0 ;



iseed = read_seed() ; /* lade Startwert fuer Zufallsgenerator */
r250_srandom(iseed) ; /* starte Zufallsgenerator */

if (sflag[2][3] == 'c') {
  alles_rausschreiben = 1 ;
} else {
  alles_rausschreiben = 0 ;
}

if (sflag[2][0] == 'f') read_parameters() ;
if (sflag[2][0] == 'c') {
  read_config(&mcs0) ;
  read_config_parameters() ;
}

if (anzahl_teilchen > MAX_TEILCHEN) {
  printf("\n Zu viele Teilchen !!!!!!!!!!!!!!!!!!!!! \n") ;
  exit(1) ;
}

time(&t_start);        /* Systemzeit bei Simulationsbeginn speichern  */
times(&u_start);

memory_allocation() ;

if (sflag[2][0] == 'f') {
  start_norm() ;
  setup_pointers() ;
  verlet_update() ;
  make_polymers() ;
  verlet_update() ;
  verlet_update_counter = 0 ;
}
if (sflag[2][0] == 'c') {
  start_config(&mcstotal) ;
  setup_pointers() ;
  verlet_update() ;
  verlet_update_counter = 0 ;
}

if ((sflag[2][1] == 'm') || (sflag[2][1] == 'r')) {
  memory_allocation_mess() ;

  if (sflag[2][0] == 'f') setup_mess() ;
  if ((sflag[2][0] == 'c') && (sflag[2][2] == 'c')) read_histo() ;
  if ((sflag[2][0] == 'c') && (sflag[2][2] == 'n')) {
    setup_mess() ;
    mcs0 = 0 ;
  }
}


write_configuration (0,mcs,mcstotal) ;
mcsave = mcs0 ;
norm =  (int) savestep * anzahl_teilchen ;


alt_energie = gesamt_energie ;

/*  HAUPTSCHLEIFE  */

times(&so_start); T_so_alt = 0.0;

mcs_laenge = 3*anzahl_teilchen + 2 ;
rot_border = (2*(double)anzahl_teilchen) / (double)mcs_laenge ;
move_border = rot_border + ((double)anzahl_teilchen/(double)mcs_laenge) ;
V_move_border = move_border + (1/(double)mcs_laenge) ;

for (step = (long)mcs0 ;step < mcsmax+mcs0 ;step++) {

  for (mcs_loop=0;mcs_loop < mcs_laenge;mcs_loop++) {
    rnd = double_r250() ;


    if (rnd < rot_border) {
      switch (rot())
        {
        case 0 : rot_counter++ ;
        case 1 : lc_rot_moves ++ ;
          break ;
        case 2 : rot_counter_kette++ ;
        case 3 : ketten_rot_moves++ ;
        }
    } else {
      if (rnd < move_border) {
        switch (move())
          {
          case 0 : move_counter++ ;
```

```
      case 1 : lc_move_moves ++ ;
        break ;
      case 2 : move_counter_kette++ ;
      case 3 : ketten_move_moves++ ;
      }
  } else {
    if (rnd < V_move_border) {
      if (V_move() == 0) V_move_counter++ ;
      V_move_try ++ ;
    } else {
      neugleichalt = 1 ;
      if (anzahl_polymere > 0) {
        if (ketten_move() == 0) ketten_move_counter ++ ;
        if (neugleichalt == 0) ketten_move_counter_same ++ ;
        ketten_move_try ++ ;
      }
    }
  }
}


if ((step >= mcsave) && ((sflag[2][1] == 'm')|| (sflag[2][1] == 'r'))) {

  times(&so_end);
  T_so_neu = (so_end.tms_utime - so_start.tms_utime);
  if (T_so_alt==0.0) T_so_alt = T_so_neu;

  if (lc_move_moves != 0) {
    akz_rate_move = (double) move_counter/lc_move_moves ;
  } else {
    akz_rate_move = 0 ;
  }
  move_counter = 0 ;
  lc_move_moves = 0 ;

  if (lc_rot_moves != 0) {
    akz_rate_rot = (double) rot_counter/lc_rot_moves ;
  } else {
    akz_rate_rot = 0 ;
  }
  rot_counter = 0 ;
  lc_rot_moves = 0 ;

  if (ketten_move_moves != 0) {
    akz_rate_move_kette = (double) move_counter_kette/ketten_move_moves ;
  } else {
    akz_rate_move_kette = 0 ;
  }
  move_counter_kette = 0 ;
  ketten_move_moves = 0 ;

  if (ketten_rot_moves != 0) {
    akz_rate_rot_kette = (double) rot_counter_kette/ketten_rot_moves ;
  } else {
    akz_rate_rot_kette = 0 ;
  }
  rot_counter_kette = 0 ;
  ketten_rot_moves = 0 ;


  if (V_move_try !=0 ) {
    akz_rate_V_move = (double) V_move_counter/V_move_try ;
  } else {
    akz_rate_V_move = 0 ;
  }
  V_move_counter = 0 ;
  V_move_try = 0 ;

  if ( ketten_move_try != 0) {
    akz_rate_ketten_move = (double) ketten_move_counter/ketten_move_try ;
  } else {
    akz_rate_ketten_move = 0 ;
  }
  ketten_move_counter = 0 ;


  if (ketten_move_try != 0) {
    akz_rate_ketten_move_same = (double) ketten_move_counter_same/
      ketten_move_try ;
  } else {
    akz_rate_ketten_move_same = 0 ;
  }
  ketten_move_counter_same = 0 ;
  ketten_move_try = 0 ;

  samples ++ ;


  mess(step,akz_rate_move,akz_rate_rot,akz_rate_move_kette,akz_rate_rot_kette,
       akz_rate_V_move,akz_rate_ketten_move,akz_rate_ketten_move_same,
       skin_verlet) ;
  write_configuration(1,(double)step,mcstotal) ;
  mcsave = mcsave + savestep ;

  if (sflag[2][1] == 'r') {

    /* Loeding'sche Akzeptanzratenanpassung */

    max_zerr = max_zerr * (1 + 0.3 *(akz_rate_V_move - 0.3)) ;
    max_jump = max_jump * (1 + 0.3 *(akz_rate_move - 0.3)) ;
    max_rotate = max_rotate * (1 + 0.3 *(akz_rate_rot - 0.3)) ;

    /* Beschraenke maximale Drehweite */

    if (max_rotate > M_PI_2) {
      max_rotate = M_PI_2 ;
    }

    max_move = ((sigma_l_halbe +cutoff) * 0.5 * max_rotate + 1.72*max_jump) ;

    if (max_zerr > max_move) {
      max_move = max_zerr ;
    }

    skin_verlet_check_update = skin_verlet - max_move ;

      /* Folgende Abfrage kollidiert natuerlich mit der Skindickenoptimierung
         - vermeidet aber unendliche Laufzeiten.... */

      if (skin_verlet_check_update < max_move/2.) {
        skin_verlet = max_move * 1.5 ;
        skin_verlet_check_update = skin_verlet - max_move ;
      }

    }

    if (autoskin) optimize_skin(T_so_alt,T_so_neu);
    T_so_alt = T_so_neu;
    times(&so_start);
  }
 }

}

write_configuration(2,(double)step,mcstotal) ;

write_seed(r250_random()) ;
/* speichere neuen Startwert fuer Zufallsgenerator */

if ((sflag[2][1] == 'm') || (sflag[2][1] == 'r')) {

  if (lc_move_moves != 0) {
```

```
    akz_rate_move = (double) move_counter/lc_move_moves ;
  } else {
    akz_rate_move = 0 ;
  }
  move_counter = 0 ;
  lc_move_moves = 0 ;

  if (lc_rot_moves != 0) {
    akz_rate_rot = (double) rot_counter/lc_rot_moves ;
  } else {
    akz_rate_rot = 0 ;
  }
  rot_counter = 0 ;
      lc_rot_moves = 0 ;

  if (ketten_move_moves != 0) {
    akz_rate_move_kette = (double) move_counter_kette/ketten_move_moves ;
  } else {
    akz_rate_move_kette = 0 ;
  }
  move_counter_kette = 0 ;
  ketten_move_moves = 0 ;

  if (ketten_rot_moves != 0) {
    akz_rate_rot_kette = (double) rot_counter_kette/ketten_rot_moves ;
  } else {
    akz_rate_rot_kette = 0 ;
  }
  rot_counter_kette = 0 ;
  ketten_rot_moves = 0 ;


  if (V_move_try !=0 ) {
    akz_rate_V_move = (double) V_move_counter/V_move_try ;
       } else {
         akz_rate_V_move = 0 ;
       }
  V_move_counter = 0 ;
  V_move_try = 0 ;

  if ( ketten_move_try != 0) {
    akz_rate_ketten_move = (double) ketten_move_counter/ketten_move_try ;
  } else {
    akz_rate_ketten_move = 0 ;
  }
  ketten_move_counter = 0 ;


  if (ketten_move_try != 0) {
    akz_rate_ketten_move_same = (double) ketten_move_counter_same/ketten_move_try ;
  } else {
    akz_rate_ketten_move_same = 0 ;
  }
  ketten_move_counter_same = 0 ;
  ketten_move_try = 0 ;

  mess(step,akz_rate_move,akz_rate_rot,akz_rate_move_kette,akz_rate_rot_kette,
       akz_rate_V_move,akz_rate_ketten_move,akz_rate_ketten_move_same,skin_verlet) ;
  memory_deallocation_mess() ;
}
memory_deallocation() ;

time(&t_end);                     /* Systemzeit bei Schleifen-Ende */
wall_secs  = (t_end-t_start);     /* Zeit fuer Schleifendurchlauf  */

times(&u_end);
CPU_secs  = (u_end.tms_utime - u_start.tms_utime);
CPU_secs  = CPU_secs/100.;
itime = (step-mcs0)* anzahl_teilchen ;

printf(" > main                :");
printf(" %-10.2f CPU seconds to execute\n",CPU_secs);
printf(" > main                :");
printf(" %-10.2f attempts / CPU second\n",itime/CPU_secs);

printf(" > main                :");
printf(" %-10.2f wall seconds to execute\n",wall_secs);
printf(" > main                :");
printf(" %-10.2f attempts / wall second\n",itime/wall_secs);

printf("%5d Verlet updates in %5d MCSs --> rate %15.6lf\n",
       verlet_update_counter,(int) (itime/anzahl_teilchen),
       (double) (verlet_update_counter*anzahl_teilchen)/itime);


}
```

```
/*****************************************************************/
/*                                                               */
/*   File: r250.c                                                */
/*                                                               */
/*   The R250 random number generator routines.                  */
/*                                                               */
/*****************************************************************/

static char _ID_[] = "$Id: r250.c,v 1.8 90/10/31 14:57:50 reger Exp $";

/* #define int long */
#define INT16

/* Comment out nextline if sizeof(int) == 2 ( 16 Bit integer )  */
#undef  INT16

#include <math.h>
#include <stdio.h>

#ifndef INT_MAX
#ifdef INT16
#define INT_MAX         0x7FFF
#else
#define INT_MAX                 0x7FFFFFFF
#endif
#endif

#define INVERSE_INT_MAX         (1.0/INT_MAX)

#define         TYPE_3          3               /* x**31 + x**3 + 1 */
#define         BREAK_3         128
#define         DEG_3           31
#define         SEP_3           3

#define P 250
#define Q 103

static int m[P]   =
          {  0x7be9c1bd, 0x088aa102, 0x3d38509b, 0x746b9fbe, 0x2d04417f,
             0x775d4351, 0x53c48d96, 0x02b26e0b, 0x418fedcf, 0x19dbc19e,
             0x78512adb, 0x1a1f5e2b, 0x307d7761, 0x6584c1f0, 0x24e3c36f,
             0x2232310f, 0x2dac5ceb, 0x106e8b5a, 0x5a05a938, 0x5e6392b6,
             0x66b90348, 0x75264901, 0x4174f402, 0x618b18a4, 0x3a6ab4bf,
             0x3c4bc289, 0x2657a9a9, 0x4e68589b, 0x09648aa6, 0x3fc489bb,
             0x1c1b715c, 0x054e4c63, 0x484f2abd, 0x5953c1f8, 0x79b9ec22,
             0x75536c3d, 0x50b10549, 0x4d7e79b8, 0x7805da48, 0x1240f318,
             0x675a3b56, 0x70570523, 0x2c605143, 0x17d7b2b7, 0x55dbc713,
             0x514414b2, 0x3a09e3c7, 0x038823ff, 0x61b2a00c, 0x140f8cff,
             0x61ebb6b5, 0x486ba354, 0x0935d600, 0x2360aab8, 0x29f6bbf8,
             0x43a08abf, 0x5fac6d41, 0x504e65a2, 0x1208e35b, 0x6910f7e7,
             0x1012ef5d, 0x2e2454b7, 0x6e5f444b, 0x58621a1b, 0x077816af,
             0x6819306d, 0x4db58658, 0x58291bf8, 0x3597aa25, 0x45bb60a0,
             0x6a6a0f11, 0x1cf1e57c, 0x361265c4, 0x16ca6054, 0x34c99833,
             0x0bee2cd7, 0x680e7507, 0x6ed37bfa, 0x0f7650d6, 0x49c11513,
             0x02e308f9, 0x7162078c, 0x122cb868, 0x0c18defa, 0x14c2b244,
             0x3c237460, 0x4fb969b9, 0x746f1f85, 0x0c71da02, 0x61c24d14,
             0x5d80176d, 0x1c84c960, 0x0fe6a1cc, 0x4bdf5bb8, 0x74e6e37b,
             0x175eb87b, 0x33f88c25, 0x429c69d3, 0x6f87d474, 0x6990364a,
             0x0857ca73, 0x59f1e385, 0x06821bc6, 0x3e6a3037, 0x70bc43d9,
             0x3b4bb3fa, 0x4a585d0f, 0x58cab8e0, 0x2a1f2ff4, 0x59ceade5,
             0x228bcdf4, 0x2d0238ee, 0x4b30b571, 0x34b8865c, 0x391b17e8,
             0x5ff367b5, 0x70dbfabc, 0x08d481a1, 0x5462873b, 0x7d4dd4bf,
             0x6a96ceb6, 0x31e29ea8, 0x19d29e1f, 0x7a7d7082, 0x7dc1fa60,
             0x0eb9819a, 0x11dc28fd, 0x31ba8685, 0x5155eb6d, 0x0163fd71,
             0x1b4abccf, 0x59adb5e0, 0x5b55e0f6, 0x21ccd896, 0x1817e618,
             0x4c1224d0, 0x5d188c90, 0x62704327, 0x24dcddb0, 0x0737bc84,
             0x3c3ef10c, 0x4768aba4, 0x3439f572, 0x076fa67e, 0x7c213200,
             0x6d550d5a, 0x67630e33, 0x6cfd2cbd, 0x76298efc, 0x3bc5956e,
             0x6a4b017c, 0x60c05db2, 0x6da83416, 0x041d9f9b, 0x5b3dce34,
             0x6b6a2e76, 0x12d72135, 0x6d19f731, 0x1d24b4fb, 0x642d0ca2,
             0x6e7df4a3, 0x386f71cb, 0x3ddac282, 0x49d3d599, 0x5a3c4a61,
             0x55f2a89a, 0x15e5fa69, 0x3754d6f1, 0x3862ebc1, 0x3ac2d81a,
             0x3e8c9375, 0x74a1dcce, 0x022b83be, 0x72c688e8, 0x7c11834c,
             0x7e4cb5bf, 0x601b9642, 0x6374917f, 0x6b49e27c, 0x5645253e,
             0x1f3a26ee, 0x5594e3f8, 0x370582f0, 0x0ce25b04, 0x59b28393,
             0x12435124, 0x784c897b, 0x6c89a4c8, 0x7f5d4856, 0x15713e76,
             0x50b6b16a, 0x6ddb3cf9, 0x4de0b041, 0x0e9173ec, 0x37af1292,
             0x281cfaa2, 0x64841c87, 0x4d950cfc, 0x5f71d193, 0x1ce70848,
             0x0857e516, 0x1dfe6509, 0x1188e516, 0x0a8368d4, 0x10c4edf1,
             0x0d9a6862, 0x08d01e93, 0x70e08433, 0x710ef9e2, 0x741a010f,
             0x4725a972, 0x104920d0, 0x49aee507, 0x7e2b2c62, 0x1d2b7bd4,
             0x2361689a, 0x106e7d87, 0x1578054f, 0x0feb0d62, 0x0fcbc5dd,
             0x2ae943c6, 0x60a1becc, 0x7da702d6, 0x78c9f407, 0x6f3332b9,
             0x35561568, 0x20e6eeaa, 0x53b74f40, 0x02eb2264, 0x0058c03d,
             0x709e5788, 0x0b43077a, 0x1e572546, 0x02273c9f, 0x15c6704f,
             0x2f1c1337, 0x0fc1a501, 0x1e968ee2, 0x1ffc976b, 0x00d09ee3,
             0x12b08ff2, 0x672240dd, 0x1119bfb3, 0x5c5f74f9, 0x654d6d3f,
             0x2e453b88, 0x7fc0dd94, 0x75bbeac6, 0x43bd40d7, 0x0fabeaf6 };

static int *k_static = m + P;
static int *j_static = m + P-Q;
static int *end      = m + P-1;

static  int     randtbl[ DEG_3 ]        =
                      { 0x9a319039, 0x32d9c024, 0x9b663182, 0x5da1f342,
                        0xde3b81e0, 0xdf0a6fb5, 0xf103bc02, 0x48f340fb,
                        0x7449e56b, 0xbeb1dbb0, 0xab5c5918, 0x946554fd,
                        0x8c2e680f, 0xeb3d799f, 0xb11ee0b7, 0x2d436b86,
                        0xda672e2a, 0x1588ca88, 0xe369735d, 0x904f35f7,
                        0xd7158fd6, 0x6fa6f051, 0x616e6b96, 0xac94efdc,
                        0x36413f93, 0xc622c298, 0xf5a42ab8, 0x8a88d77b,
                                    0xf5ad9d0e, 0x8999220b, 0x27fb47b9 };

static  int            *fptr                   = randtbl + SEP_3;
static  int            *rptr                   = randtbl;
static  int            *end_ptr                = randtbl + DEG_3;

static  int            *state                  = randtbl;

static  int             rand_deg                = DEG_3;
static  int             rand_sep                = SEP_3;

static int
xrandom()

{
        int            i;

        *fptr += *rptr;
        i = (*fptr >> 1) & INT_MAX;    /* chucking least random bit */
        if(  ++fptr  >=  end_ptr  )  {
            fptr = state;
            ++rptr;
        }
        else  {
            if(  ++rptr  >=  end_ptr  )  rptr = state;
        }
        return( i );
}

static void
xsrandom( x )

        unsigned int           x;
{
        register  int           i;
        /*  int int             random( void );  */
        int             xrandom();

        state[ 0 ] = x;
        for( i = 1; i < rand_deg; i++ )  {
            state[i] = (1103515245*state[i - 1] + 12345) & INT_MAX;
```

```
        }
        fptr = &state[ rand_sep ];
        rptr = &state[ 0 ];
        for( i = 0; i < 10*rand_deg; i++ )  xrandom();
}

void
r250_srandom ( seed )

    unsigned int seed;

{
        register int  i;

        xsrandom( seed );

        for ( i=0; i<P; ++i) m[i] = xrandom();

        k_static = m + P;
        j_static = m + P-Q;
}

int
r250_random ()

{

    if (--k_static < m) k_static = end;
    if (--j_static < m) j_static = end;

    return (*k_static ^= *j_static);
}

float
float_r250 ()

{
    if (--k_static < m) k_static = end;
    if (--j_static < m) j_static = end;

    return ( (*k_static ^= *j_static) * INVERSE_INT_MAX );
}

double
double_r250 ()

{
    if (--k_static < m) k_static = end;
    if (--j_static < m) j_static = end;

    return ( (*k_static ^= *j_static) * INVERSE_INT_MAX );
}

void
r250_save ( table )

    int table[];

{
    int i;

    for ( i = 0; i < P; i++ )
        table[i] = m[i];

    table[P  ] = k_static - m;
    table[P+1] = j_static - m;
}

void
r250_restart ( table )

    int table[];

{
    int i;

    for ( i = 0; i < P; i++ )
        m[i] = table[i];

    k_static = m + table[P  ];
    j_static = m + table[P+1];
}

double
gauss_r250 ( mean, variance )

    double              mean, variance;

{
    static int          empty = 1;
    static double       store;
    double              v1, v2, r, root;

    if ( empty ) {

        do {

            /*  Inline double_random()  */

            if (--k_static < m) k_static = end;
            if (--j_static < m) j_static = end;
            v1 = (*k_static ^= *j_static) * INVERSE_INT_MAX;

            v1 = 2.0 * v1 - 1.0;

            /*  Inline double_random()  */

            if (--k_static < m) k_static = end;
            if (--j_static < m) j_static = end;
            v2 = (*k_static ^= *j_static) * INVERSE_INT_MAX;

            v2 = 2.0 * v2 - 1.0;

            r  = v1 * v1 + v2 * v2;

        } while ( r > 1.0 );

        root  = variance * sqrt( -2.0 * log(r) / r );
        store = v2 * root + mean;
        empty = 0;

        return ( v1 * root + mean );

    } else {

        empty = 1;
        return ( store );

    }

}
```

```
/*****************************************************************/
/*                                                               */
/*   File: r250.c                                                */
/*                                                               */
/*   A simple multiplicative generator from Kalos & Whitlock.    */
/*                                                               */
/*****************************************************************/

static int      rand_seed = 1;

int
rand ()

{
    rand_seed = ( 1812433253 * rand_seed + 314159265 ) & INT_MAX;
    return  ( rand_seed );
}

float
float_rand ()

{
    rand_seed = ( 1812433253 * rand_seed + 314159265 ) & INT_MAX;
    return  ( rand_seed * INVERSE_INT_MAX );
}


void
Srand ( seed )

    int  seed;

{
    rand_seed = seed;
}

int
rand_save ()

{
    return ( rand_seed );
}

#define MAX(a,b) ((a)>(b)?(a):(b))

void
r250_vector (x, n)

register int    *x;
int              n;

{
    register int        *j, *k, *l;

    j = j_static;
    k = k_static;

    while ( n > 0 ) {

        if ( k == m ) k = m + P;
        l = MAX( k-n, m+Q );
/*  printf("\n I: N = %d, k,l : %d %d %d %d", n, k, l, m, m+P); fflush(stdout);  */
        if ( k > l ) {
            n -= k-l;
#ifdef __STDC__
#pragma ivdep
#endif
            while ( k > l )
                *x++ = (*--k ^= *--j);
        }

        if ( j == m ) j = m + P;
        l = MAX( j-n, m+P-Q );
/*  printf(" ==> N = %d, j,l : %d %d", n, j, l); fflush(stdout); */
        if ( j > l ) {
            n -= j-l;
#ifdef __STDC__
#pragma ivdep
#endif
            while ( j > l )
                *x++ = (*--k ^= *--j);
        }

    }

    j_static = j;
    k_static = k;

}

void
float_r250_vector (x, n)

register float  *x;
int              n;

{
    register int        *j, *k, *l;

    j = j_static;
    k = k_static;

    while ( n > 0 ) {

        if ( k == m ) k = m + P;
        l = MAX( k-n, m+Q );
        if ( k > l ) {
            n -= k-l;
#ifdef __STDC__
#pragma ivdep
#endif
            while ( k > l )
                *x++ = (*--k ^= *--j) * INVERSE_INT_MAX;
        }

        if ( j == m ) j = m + P;
        l = MAX( j-n, m+P-Q );
        if ( j > l ) {
            n -= j-l;
#ifdef __STDC__
#pragma ivdep
#endif
            while ( j > l )
                *x++ = (*--k ^= *--j) * INVERSE_INT_MAX;
        }

    }

    j_static = j;
    k_static = k;

}
```

```
/* matrix.c */
/* %W%   %E% */
#define PROTOCOL
#define STANDARD_C
#define MAXBLOCKS 1000
#define MAXDIM 10
#define NULL    0
#ifdef NO_STANDARD_C
#include <varargs.h>
#endif
#ifdef STANDARD_C
#include <stdarg.h>
#endif
#include <sys/types.h>
#include <stdio.h>
#include <assert.h>
#include "matrix.h"

#ifdef STANDARD_C
void *matrix_alloc
(
        size_t dim,
        size_t el_size,
    ...
)
{
        size_t cant_length[MAXDIM];
        va_list pvar;
        size_t  i,j;
        size_t  data_mem,aux_mem,mem_help,offset;
        void    **pointer,*ptr;
        char    *block, *address;
        if(dim<0 || dim>MAXDIM) return(NULL);
        if(dim==0)
        {
                ptr=(void *)calloc(1,el_size);
        mallprot(ptr,(int)el_size,ptr);
                assert(ptr!=NULL);
                return(ptr);
        }
        va_start(pvar,el_size);
        for(i=0;i<dim;i++) cant_length[i]=va_arg(pvar,size_t);
        if(dim==1)
        {
                ptr=(void *)calloc(cant_length[0],el_size);
        mallprot(ptr,(int)(cant_length[0]*el_size),ptr);
                assert(ptr!=NULL);
                return(ptr);
        }
        va_end(pvar);
#endif

#ifdef  NO_STANDARD_C
void *matrix_alloc
(
        va_alist
)
        va_dcl
{
        va_list pvar;
        size_t dim;
        size_t el_size;
        size_t cant_length[MAXDIM];
        size_t  i,j;
        size_t  data_mem,aux_mem,mem_help,offset;
        void    **pointer,*ptr;
        char    *block,*address;
        va_start(pvar);
        dim=va_arg(pvar,size_t);
        el_size=va_arg(pvar,size_t);
        if(dim<0 || dim>MAXDIM) return(NULL);
        if(dim==0)
        {
                ptr=(void *)calloc(1,el_size);
        mallprot(ptr,(int)el_size,ptr);
                assert(ptr!=NULL);
                return(ptr);
        }
        for(i=0;i<dim;i++)
        {
                cant_length[i]=va_arg(pvar,size_t);
        }
        if(dim==1)
        {
                ptr=(void *)calloc(cant_length[0],el_size);
        mallprot(ptr,(int)(cant_length[0]*el_size),ptr);
                assert(ptr!=NULL);
                return(ptr);
        }
        va_end(pvar);
#endif

        data_mem=1;
        for(i=0;i<dim;i++) data_mem*=cant_length[i];
        data_mem*=el_size;
        aux_mem=0;mem_help=1;
        for(i=0;i<dim-1;i++)
        {
                mem_help*=cant_length[i];
                aux_mem+=mem_help;
        }
        aux_mem*=sizeof(void *);
        block=(char *) calloc(aux_mem+data_mem,1);
    mallprot(block,(int)(aux_mem+data_mem),block);
        assert(block!=NULL);
/*
        assert(sizeof(char)==1);
*/
        if(block==NULL) return(NULL);
        pointer=(void **)block;
        address=block+cant_length[0]*sizeof(void *);
        mem_help=1;
        for(i=1;i<dim;i++)
        {
                offset=cant_length[i]*(i<(dim-1)?sizeof(void *):el_size);
                mem_help*=cant_length[i-1];
                for(j=0;j<mem_help;j++)
                {
                        *pointer++=(void *)address;
                        address  +=offset;
                }
        }
        return(block);
}

#ifdef STANDARD_C
void *harve_alloc
(
        size_t dim,
        size_t el_size,
    ...
)
{
        size_t cant_length[MAXDIM];
        size_t *cord_length;
        size_t cords;
        va_list pvar;
        size_t  i,j;
        size_t  data_mem,aux_mem,mem_help,offset;
        void    **pointer,*ptr;
        char    *block, *address;
        if(dim<0 || dim>MAXDIM) return(NULL);
```

```
if(dim==0)
{
        ptr=(void *)calloc(1,el_size);
    mallprot(ptr,(int)el_size,ptr);
        assert(ptr!=NULL);
        return(ptr);

    }
    va_start(pvar,el_size);
    for(i=0;i<dim-1;i++) cant_length[i]=va_arg(pvar,size_t);
    cord_length=va_arg(pvar,size_t *);
    if(dim==1)
    {
        ptr=(void *)calloc(cord_length[0],el_size);
    mallprot(ptr,(int)(cord_length[0]*el_size),ptr);
        assert(ptr!=NULL);
        return(ptr);

    }
    va_end(pvar);

#endif

#ifdef  NO_STANDARD_C
void *harve_alloc
(
    va_alist

)

{
    va_dcl

    va_list pvar;
    size_t dim;
    size_t el_size;
    size_t cant_length[MAXDIM];
    size_t *cord_length;
    size_t cords;
    size_t  i,j;
    size_t  data_mem,aux_mem,mem_help,offset;
    void    **pointer,*ptr;
    char    *block,*address;
    va_start(pvar);
    dim=va_arg(pvar,size_t);
    el_size=va_arg(pvar,size_t);
    if(dim<0 || dim>MAXDIM) return(NULL);
    if(dim==0)
    {
        ptr=(void *)calloc(1,el_size);
    mallprot(ptr,(int)el_size,ptr);
        assert(ptr!=NULL);
        return(ptr);

    }
    for(i=0;i<dim-1;i++) cant_length[i]=va_arg(pvar,size_t);
    cord_length = va_arg(pvar,size_t *);
    if(dim==1)
    {
        ptr=(void *)calloc(cord_length[0],el_size);
    mallprot(ptr,(int)(cord_length[0]*el_size),ptr);
        assert(ptr!=NULL);
        return(ptr);

    }
    va_end(pvar);

#endif

    data_mem=0;
    cords=1;
    for(i=0;i<dim-1;i++) cords*=cant_length[i];
    if(cords<=0) return(NULL);
    for(i=0;i<cords;i++) data_mem+=cord_length[i];
    data_mem*=el_size;
    aux_mem=0;mem_help=1;
    if(dim>2) for(i=0;i<dim-2;i++)
    {
        mem_help*=cant_length[i];
        aux_mem+=mem_help;
    }
    aux_mem+=cords;
    aux_mem*=sizeof(void *);
    block=(char *) calloc(aux_mem+data_mem,1);
    mallprot(block,(int)(aux_mem+data_mem),block);
    assert(block!=NULL);

/*
    assert(sizeof(char)==1);
*/
    if(block==NULL) return(NULL);
    pointer=(void **)block;
    address=block+cant_length[0]*sizeof(void *);
    mem_help=1;
    for(i=1;i<dim-1;i++)
    {
        mem_help*=cant_length[i-1];
        offset=cant_length[i]*sizeof(void *);
        for(j=0;j<mem_help;j++)
        {
            *pointer++=(void *)address;
            address  +=offset;
        }
    }
    for(i=0;i<cords;i++)
    {
        offset=cord_length[i]*el_size;
        *pointer++=(void *) address;
        address+=offset;
    }
    return(block);
}

void mallprot
(
    void    *newblck,
int     size,   *oldblck
)
{
    static  char   *block[MAXBLOCKS];
  static  int  sizes[MAXBLOCKS];
    static  int     init=0;
    int             i,indx=0,last=0;
    int             ptr1_is_new=1,ptr2_is_new=1;
    int             totsize=0;
    if(init==0)
    {
        init=1;
        for(i=0;i<MAXBLOCKS;i++)
        {
            block[i]=NULL;
            sizes[i]=0;
        }
    }
    if(newblck==NULL||oldblck==NULL)
    {
        printf
        (
    " > mallprot\t\t: allocation failed! (%i bytes)\n",size
        );
        fflush(stdout);
        return;
    }
    /*check if ptr are new*/
    for(i=0;i<MAXBLOCKS;i++)
    {
        if(newblck==block[i])       ptr1_is_new=0;
        if(oldblck=block[i])        {ptr2_is_new=0;indx=i;}
        if(block[i]!=NULL)          last=i;
        totsize+=sizes[i];
    }
```

```
        last++;

        if      (ptr1_is_new&&ptr2_is_new&&size>=0) /*alloc*/
        {
                assert(newblck==oldblck);
                assert(last<MAXBLOCKS);
#ifdef PROTOCOL
                printf
                (
            " > alloc\t\t: %i new bytes ==> %i total bytes\n",size,totsize+size
                );
                fflush(stdout);
#endif
                block[last]=(char *) newblck;
                sizes[last]=size;
        }
        else if (!ptr2_is_new&&size>=0) /*realloc*/
        {
                if(ptr1_is_new) {assert(last<MAXBLOCKS);}
                else            assert(newblck==oldblck);
#ifdef PROTOCOL
                printf
                (
            " > realloc\t\t: %i bytes ==> %i total bytes\n",
                        size-sizes[indx],totsize+size-sizes[indx]
                );
                fflush(stdout);
#endif
                block[indx]=(char *) newblck;
                sizes[indx]=size;
        }
        else if (!ptr1_is_new&&!ptr2_is_new&&size<0)    /*free*/
        {
                assert(newblck==oldblck);
#ifdef PROTOCOL
                printf
                (
            " > free\t\t\t: %i bytes ==> %i total bytes\n",
                        sizes[indx],totsize-sizes[indx]
                );
                fflush(stdout);
#endif
                for(i=indx;i<MAXBLOCKS-1;i++)
                {
                        block[i]=block[i+1];
                        sizes[i]=sizes[i+1];
                }
                block[MAXBLOCKS-1]=NULL;
                sizes[MAXBLOCKS-1]=0;
        }
        else printf
        (
        " > mallprot\t\t: error in mallprot (%i,%i,%i)\n",
                newblck,size,oldblck
        );
        return;
}

#if(0)
/*int main()*/
/*int test_matrix_alloc()*/
int test_matrix_alloc()
{
        int     indx,i,j,k,l;
        int     *linear;
        int     ***test;
        int     list1[1]={3},sum1=3;
        int     list2[3]={1,2,3},sum2=6;
        int     list3[3][5]={15,14,13,12,11,10,9,8,7,6,5,4,3,2,1},sum3=120;
        int     *crumm0,*crumm1,**crumm2,***crumm3;

        test=(int ***) matrix_alloc(3,sizeof(int),3,5,10);
        linear=&(test[0][0][0]);
        for(indx=0;indx<3*5*10;indx++)
        {
            *linear++=indx;
        }
        for(i=0;i<3;i++)
        for(j=0;j<5;j++)
        for(k=0;k<10;k++)
           printf("%ld %ld %ld : %ld \n",i,j,k,test[i][j][k]);
             crumm0=(int *)  harve_alloc(0,sizeof(int));
             crumm1=(int *)  harve_alloc(1,sizeof(int),list1);
             crumm2=(int **) harve_alloc(2,sizeof(int),3,list2);
             crumm3=(int ***)harve_alloc(3,sizeof(int),3,5,list3);
             crumm0[0]=0;
             for(i=0;i<sum1;i++) crumm1[i]=i;
             for(i=0;i<sum2;i++) crumm2[0][i]=i;
             for(i=0;i<sum3;i++) crumm3[0][0][i]=i;
             printf("%ld \n",crumm0[0]);
             for(i=0;i<3;i++) printf("%ld: %ld\n",i,crumm1[i]);
             for(i=0;i<3;i++) for(j=0;j<list2[i];j++)
                        printf("%ld %ld: %ld\n",i,j,crumm2[i][j]);
             for(i=0;i<3;i++)
             for(j=0;j<5;j++)
             for(k=0;k<list3[i][j];k++)
                        printf("%ld %ld %ld: %ld\n",i,j,k,crumm3[i][j][k]);
             free((char *) test);
             return(1);
}

int test_mallprot()
/*int main()*/
{
        int     *p0,*p1;
        p0=(int *)calloc(1000,sizeof(int));
        mallprot(p0,1000*sizeof(int),p0);
        p1=(int *)realloc(p0,2000*sizeof(int));
        mallprot(p1,2000*sizeof(int),p0);
        free(p1);
        mallprot(p1,-1,p1);
}
#endif
```