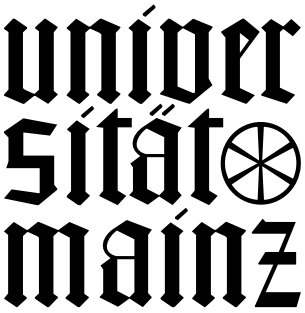

Johannes Gutenberg-Universität Mainz
Institut für Theoretische Festkörperphysik
Prof. Dr. Rolf Schilling

Diplomarbeit

Test der Modenkopplungstheorie in zwei Dimensionen

zur Erlangung des Grades eines

Diplom-Physikers

vorgelegt von

Gerolf Ziegenhain

In der vorliegenden Diplomarbeit wird die Gültigkeit der Modenkopplungstheorie in zwei Dimensionen an monodispersen und binären (dipolaren) harten Scheiben überprüft. Die zentralen Resultate der Arbeit sind ein schneller und stabiler Algorithmus zur Iteration der statischen Strukturfaktoren in zwei Dimensionen, eine Diskussion der statischen Korrelatoren der genannten Systeme und weiter die Untersuchung des Glasüberganges der beiden monodispersen Systeme.

Inhaltsverzeichnis

1. Einführung

Allgemeines zu Gläsern In der vorliegenden Arbeit werden wir physikalische und strukturelle Eigenschaften von zweidimensionalen Flüssigkeiten und Gläsern studieren. Betrachten wir eine unterkühlte Flüssigkeit, deren Kristallisation bei Abkühlen oder Verdichten verhindert wird. Es entstehen Frustrationen, woraus für die unterkühlte Flüssigkeit Eigenschaften eines Festkörpers, wie z.B. Scherspannung, bei gleichzeitig fehlender langreichweitiger Ordnung folgen. Diese strukturellen Eigenschaften werden wir durch Korrelationsfunktionen beschreiben. Die Kapitel zwei bis vier beschäftigen sich mit diesen statischen Korrelatoren und den strukturellen statischen Eigenschaften von zweidimensionalen Flüssigkeiten und Gläsern.

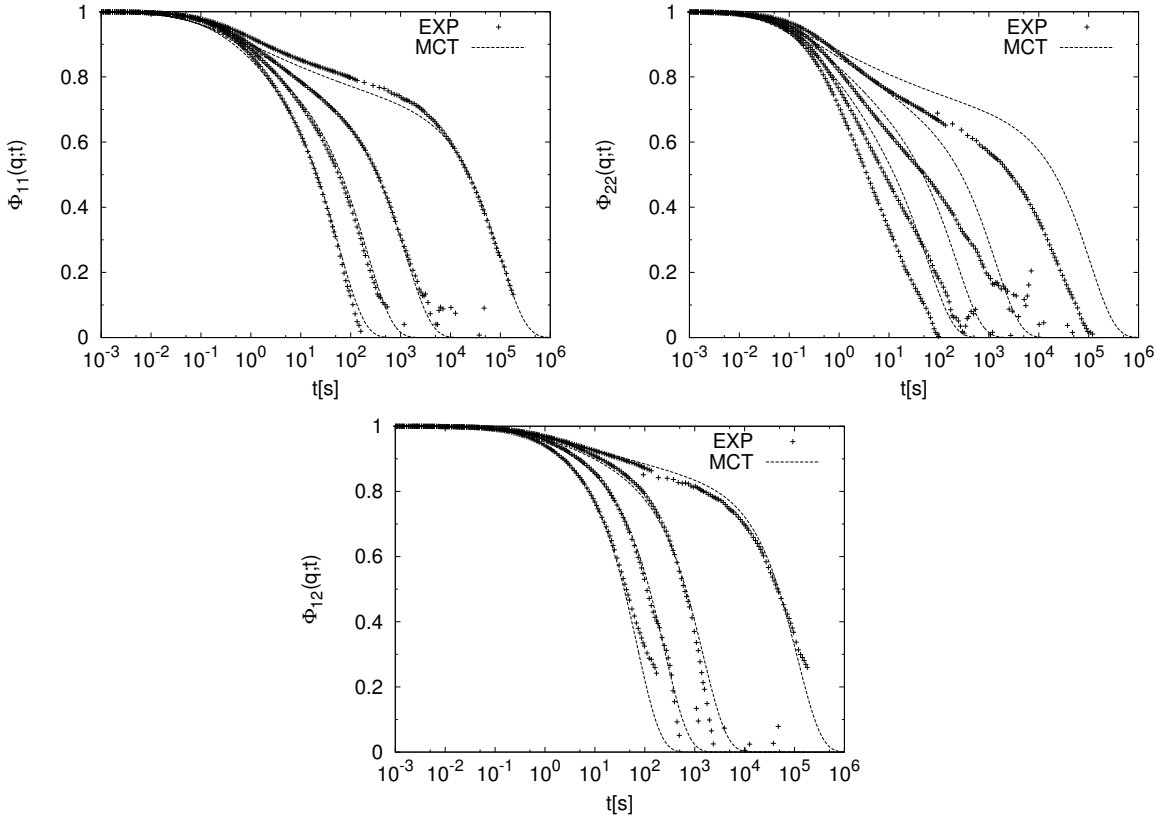

Abbildung 1.1.: Intermediäre Streufunktion $\Phi_{\alpha\beta}(q;t)$ für binäre harte Kugeln in drei Dimensionen bei einem Radienverhältnis $\delta = 0.2$ und einer Konzentration $x_1 = 0.2$: MCT aus [Voi03] bei $\eta = \{0.46; 0.475; 0.497; 0.51\}$ an der Stelle $q\sigma_{22} = 5.4$ und EXP aus [WvM01] bei $\eta = \{0.51; 0.53; 0.55; 0.57\}$ an der Stelle $q\sigma_{22} = 6.0$; dabei werden die Korrelatoren mit wachsender Packungsdichte langreichweitiger; sämtliche Bezeichnungen sind später in diesem Kapitel eingeführt

Als Glasübergang bezeichnet man den Übergang von ergodischer Flüssigkeits- zu nichtergodischer

Glas-Dynamik. Im Gegensatz zu thermodynamischen Phasenübergängen ist der aus der Modenkopplungstheorie gelieferte Glasübergang nicht mit Divergenzen thermodynamischer Größen verbunden, woraus die Problematik der Definition des Glasübergangs folgt. Erster Indikator für einen Glasübergang ist das Auftreten eines Plateaus in dem *zeitabhängigen Strukturfaktor*

$$S(q;t) := \frac{1}{N}\langle \rho^*(\vec{q};t=0)\rho(\vec{q};t)\rangle$$

wobei die fouriertransformierte Teilchendichte $\rho(\vec{q};t) := \sum_{i=1}^{N} e^{i\vec{q}\cdot\vec{x}_i(t)}$ und N die Gesamtteilchenzahl ist. Genauso kann für Systeme zweier Teilchensorten der *zeitabhängige Strukturfaktor* definiert werden als

$$S_{\alpha\beta}(q;t) := \frac{1}{N}\left\langle \rho^*_\alpha(\vec{q};t=0)\rho_\beta(\vec{q};t)\right\rangle$$

Dabei indizieren $\alpha,\beta \in 1,2$ die beiden Teilchensorten und die fouriertransformierte zeitabhängige Teilchendichte der Sorte α ist über die Teilchenzahl N_α der Sorte α definiert als $\rho_\alpha(\vec{q};t) := \sum_{i=1}^{N_\alpha} e^{i\vec{q}\cdot\vec{x}_i(t)}$. Es gilt dabei für die Gesamtteilchenzahl $N = N_1 + N_2$.

Weiter lässt sich die *intermediäre Streufunktion*, die dem auf den Zeitpunkt $t = 0$ normierten Strukturfaktor entspricht, definieren:

$$\begin{aligned}
\Phi(q;t) &:= \frac{\langle \rho^*(\vec{q};t=0)\rho(\vec{q};t)\rangle}{\langle \rho^*(\vec{q};t=0)\rho(\vec{q};t=0)\rangle} \\
\Phi_{\alpha\beta}(q;t) &:= \frac{\left\langle \rho^*_\alpha(\vec{q};t=0)\rho_\beta(\vec{q};t)\right\rangle}{\left\langle \rho^*_\alpha(\vec{q};t=0)\rho_\beta(\vec{q};t=0)\right\rangle}
\end{aligned}$$

In Abb. 1.1 ist die intermediäre Streufunktion für das System binärer harter Kugeln in drei Dimensionen dargestellt[1]. Das in allen drei Komponenten auftretende Plateau wird erzeugt durch den *Cage-Effekt*, bei dem nächste Nachbarn einen Käfig (*Cage*) um ein Teilchen bilden und damit das von dem Teilchen erreichbare Phasenraumvolumen einschränken. Effektiv wird dadurch die Dynamik des Systems verlangsamt und eine Relaxation der Flüssigkeit verhindert.

Charakteristisch für ein Glas ist insbesondere der Langzeitlimes des dynamischen Strukturfaktors $S(q;t)$ für Systeme bestehend aus einer Teilchensorte bzw. $S_{\alpha\beta}(q;t)$ für Systeme bestehend aus zwei Teilchensorten, der bei stetiger Änderung der Systemparameter einen unstetigen Übergang von flüssigartigem zu glasartigem Verhalten aufweist. Dieser Langzeitlimes wird auch als *Nichtergodizitätsparameter* (NEP) bezeichnet:

$$\begin{aligned}
f(q) &:= \lim_{t\to\infty} \Phi(q;t) \\
F_{\alpha\beta}(q) &:= \lim_{t\to\infty} S_{\alpha\beta}(q;t)
\end{aligned}$$

Dabei ist der monodisperse[2] NEP auf den statischen Strukturfaktor normiert. Der Glasübergang kann am NEP abgelesen werden:

[1] Dabei ist die Definition von $\Phi_{\alpha\beta}(q;t) := \frac{\langle \rho^*_\alpha(\vec{q};t=0)\rho_\beta(\vec{q};t)\rangle}{\langle \rho^*_\alpha(\vec{q};t=0)\rho_\beta(\vec{q};t=0)\rangle}$ nur genau dann sinnvoll, wenn $\left\langle \rho^*_\alpha(\vec{q};t=0)\rho_\beta(\vec{q};t=0)\right\rangle \neq 0$. Daher wurde in Abb. 1.1 ein festes q so gewählt, dass dies erfüllt ist.

[2] Wir bezeichnen ein System bestehend aus Teilchen einer Sorte als monodispers und eines bestehend aus zwei Teilchensorten als binär.

$$f(q)\begin{cases} \neq 0 & \text{Glas} \\ = 0 & \text{Flüssigkeit} \end{cases}$$
$$F_{\alpha\beta}(q)\begin{cases} \neq 0 & \text{Glas} \\ = 0 & \text{Flüssigkeit} \end{cases}$$

Da also der Glasübergang am NEP abgelesen werden kann, ist das Ziel der vorliegenden Arbeit, schließlich den NEP für zweidimensionale Systeme zu berechnen. Um dies zu bewerkstelligen benötigen wir die Modenkopplungstheorie.

Modenkopplungstheorie Der Glasübergang an sich ist bis heute noch immer nicht umfassend verstanden. Jedoch können wichtige Erkenntnisse über ihn bereits mit Hilfe der Modenkopplungstheorie (im Folgenden kurz MCT) gewonnen werden. Ausgehend vom Projektionsformalismus von Mori und Zwanzig [For83] kann ein exaktes Bewegungsgleichungssystem der dynamischen Strukturfaktoren aufgestellt werden (siehe Abschnitt 5.2). Der in diesem Gleichungssystem auftretende Gedächtniskern kann mittels der MCT approximiert werden [UBS84] und es ergibt sich ein geschlossenes Integro-DGL-System der dynamischen Strukturfaktoren (Abschnitt 5.3). Mit diesem Gleichungssystem können die Nichtergodizitätsparameter in Kapitel 7 bestimmt werden [Göt89]. An deren Änderung unter Variation der Systemparameter kann der Glasübergang abgelesen werden (siehe Kapitel 7). Beachtenswert ist dabei, dass die MCT eine mikroskopische Theorie ist, die ihre Aussagen allein aus der Kenntnis der Wechselwirkung der Teilchen im System zieht. Denn die MCT ermöglicht $S(q;t)$ aus dem statischen Strukturfaktor $S(q)$ zu bestimmen, der seinerseits mittels PY- bzw. HNC-Approximation aus dem Potential der Teilchen berechenbar ist. Das heißt, wir müssen zunächst Lösungen für die statischen Korrelatoren der jeweiligen Systeme finden, bevor mittels der MCT die NEP und damit der Glasübergang untersucht werden kann. Nachteil der MCT ist einerseits, dass sie auf mehreren Näherungen beruht, deren Güte bisher noch nicht abgeschätzt werden konnte. Andererseits, dass die in dieser Arbeit betrachtete idealisierte MCT nicht der Möglichkeit Rechnung trägt, dass wegen des Cage-Effekts gefangene Teilchen wieder aus dem Käfig entkommen können. Wir werden die MCT und den Glasübergang zweidimensionaler Systeme in den Kapiteln fünf bis sieben beschreiben.

Zweidimensionale Systeme Hinreichend untersucht wurde die Gültigkeit der MCT bisher in drei Dimensionen für monodisperse und binäre harte Kugeln [Voi03] (siehe auch Abb. 1.1), ebenso für monodisperse dipolare harte Kugeln [SS97, TTL01]. Harte Kugeln sind physikalisch besonders interessant, weil Flüssigkeiten oft bei hoher Dichte - d.h. bei kleinen mittleren Abständen der Teilchen - durch das Potential der harten Teilchen dominiert werden. Die Frage nach der Gültigkeit der MCT in zwei Dimensionen wurde motiviert durch die experimentelle Realisierung eines zweidimensionalen Glasformers aus einer binären Mischung dipolarer harter Scheiben [HKM05]. Bei diesem System handelt es sich um ein einfaches Glas in dem Sinne, dass es nur aus Kolloiden und nicht etwa Molekülen besteht. In Abb. 1.2 ist der über alle Teilchen gemittelte zeitabhängige Strukturfaktor aus dem Experiment für verschieden starke dipolare Wechselwirkung dargestellt. Wie in drei Dimensionen (Abb. 1.1) tritt ein Cage-Effekt und somit glasartiges Verhalten auf und es stellt sich die Frage, ob und wie gut dieses System mit der MCT beschrieben werden kann.

Wir geben nun einen Überblick über die von uns in dieser Arbeit betrachteten physikalischen Systeme. Eine tiefergehende Diskussion findet sich dann in Kapitel 4. Um die Frage zu klären, inwiefern

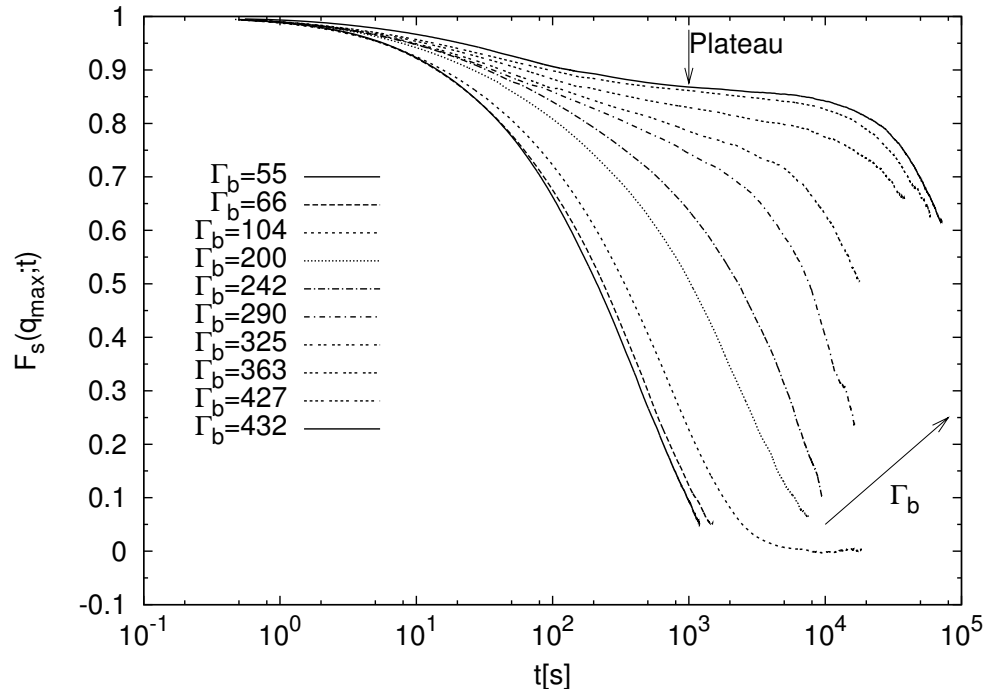

Abbildung 1.2.: Selbstanteil der intermediären Streufunktion $F_s(q;t) := \frac{1}{N}\left\langle \sum_{j=1}^{N} e^{i\vec{q}\cdot[\vec{x}_j(t)-\vec{x}_j(t=0)]} \right\rangle$ für binäre zweidimensionale magnetische Kolloide aus dem Experiment [Hun01] gemittelt über alle Teilchen; Der Pfeil mit Γ_b zeigt in Richtung wachsendem Γ_b.

die MCT in zwei Dimensionen gültig ist, betrachten wir zunächst das einfachst mögliche System monodisperser zweidimensionaler harter Scheiben. Das Potential ist hinreichend bekannt und lautet:

$$V(r) := \begin{cases} \infty & \frac{r}{\sigma} \leq 1 \\ 0 & \frac{r}{\sigma} > 1 \end{cases}$$

Wir sind bei der Definition gleich zu reduzierten Einheiten übergegangen und haben auf den Teilchendurchmesser σ skaliert. Das heißt, der Ort wird im folgenden auf $\frac{r}{\sigma}$ und der Impuls auf $q\sigma$ skaliert. Systemparameter der harten Scheiben in zwei Dimensionen ist, wie im dreidimensionalen Fall, allein die *Packungsdichte*[3]

$$\eta := \rho\sigma^2\frac{\pi}{4}$$

wobei die Dichte ρ durch die Anzahl N der Teilchen pro Fläche F gegeben ist: $\rho := \frac{N}{F}$. Eine tiefer gehende Beschreibung dieses Systems findet sich in Abschnitt 4.1.

Weiter können diese harten Scheiben durch ein repulsiv dipolares Potential ergänzt werden (zur Motivation aus dem Experiment siehe Abschnitt 4.2), was auch bereits in drei Dimensionen untersucht wurde [TTL01]. Der repulsiv dipolare Anteil des Potentials ist:

$$V(\frac{r}{\sigma} > 1) := \frac{\mu_0}{4\pi}\frac{\chi^2 B^2}{(\frac{r}{\sigma})^3\sigma^3}$$

[3] Sie kann allgemein über das D-dimensionale Volumen V_D, die Dichte $\rho = \frac{N}{V_D}$ und den Teilchendurchmesser σ definiert werden als $\eta := V_D\left(\frac{\sigma}{2}\right)^D\rho$ und gibt den Bedeckungsgrad des Systems an (d.h. $0 \leq \eta \leq 1$).

Dabei ist B das äußere magnetische Feld und χ die magnetische Suszeptibilität der Teilchen. Dieses System lässt sich durch die Packungsdichte η und die Temperatur T (gemessen in Kelvin) kontrollieren (siehe Abschnitt 4.2). Wir werden zeigen (Abschnitt 2.1, Abschnitt 4.2), dass das System bei geringen Packungsdichten bereits durch einen Parameter bestimmt ist: die monodisperse *mittlere magnetische Wechselwirkungsenergie*

$$\Gamma_m := \frac{\mu_0}{4\pi} B^2 \frac{(\rho\pi)^{\frac{3}{2}}}{k_B T} \chi^2$$

Sie ist ein einheitenloses Maß für die Stärke der magnetischen Wechselwirkung zwischen zwei Teilchen im mittleren Abstand $\bar{r} := \rho^{-1/2}\sigma$. k_B ist dabei wie üblich die Boltzmann-Konstante und μ_0 die magnetische Permeabilitätskonstante. Wir werden dieses System näher in Abschnitt 4.2 betrachten.

Die Verallgemeinerung der MCT auf binäre Systeme in drei Dimensionen wurde bereits durchgeführt [Voi03] und wird von uns in Abschnitt 5.3 für zweidimensionale Systeme erweitert. Die von uns betrachteten binären Systeme haben einen Harte-Scheiben-Potentialanteil. Seien die Radien der Teilchen r_1 und r_2, so können dann formal drei Durchmesser

$$\sigma_{\alpha\beta} := r_\alpha + r_\beta$$

mit $\alpha, \beta \in \{1,2\}$ definiert werden. Die binären Systeme werden wir immer auf den mittleren Durchmesser σ_{12} normieren, das heißt, der Ort wird in $\frac{r}{\sigma_{12}}$ und der Impuls in $q\sigma_{12}$ gemessen. Dann kann das Potential der binären harten Scheiben angegeben werden:

$$V_{\alpha\beta}(r) := \begin{cases} \infty & \frac{r}{\sigma_{12}} \leq \frac{\sigma_{\alpha\beta}}{\sigma_{12}} \\ 0 & \frac{r}{\sigma_{12}} > \frac{\sigma_{\alpha\beta}}{\sigma_{12}} \end{cases}$$

Sei weiter mit der Gesamtteilchenzahl N und den Teilchenzahlen N_α

$$x_\alpha := \frac{N_\alpha}{N}$$

die *Konzentration der Teilchensorte* α definiert und $\rho := \frac{N}{F}$ die Gesamtdichte des Systems. Dann kann die *Packungsdichte* für binäre Systeme eingeführt werden:

$$\eta := \rho \frac{\pi}{4} [x_1 \sigma_{11}^2 + x_2 \sigma_{22}^2]$$

Außerdem definieren wir das *Radienverhältnis*

$$\delta := \frac{r_1}{r_2}$$

Das System binärer harter Scheiben wird durch die Packungsdichte η, das Radienverhältnis δ und den *Mischungsparameter* $x_1 := \frac{N_1}{N}$ kontrolliert. Dabei muss nur eine Konzentration angegeben werden, denn $x_1 + x_2 = 1$. Für eine tiefergehende Beschreibung verweisen wir auf Abschnitt 4.3.

Wie im monodispersen Fall kann auch binär das System harter Scheiben durch ein repulsiv dipolares Potential

$$V_{\alpha\beta}(\frac{r}{\sigma_{12}} > \frac{\sigma_{\alpha\beta}}{\sigma_{12}}) := \frac{\mu_0}{4\pi} \frac{\chi_\alpha \chi_\beta B^2}{(\frac{r}{\sigma_{12}})^3 \sigma_{12}^3}$$

ergänzt werden. Dabei sind χ_α die magnetischen Suszeptibilitäten der Teilchen. Die physikalischen Parameter dieses Systems sind dann die Packungsdichte η, der Mischungsparameter x_1, das Radienverhältnis δ und das *Verhältnis der magnetischen Suszeptibilitäten*:

$$\delta_\chi := \frac{\chi_1}{\chi_2}$$

Für kleine Packungsdichten lässt sich das System der binären dipolaren harten Scheiben wieder allein durch die *mittlere magnetische Wechselwirkungsenergie* kontrollieren, die binär definiert ist als

$$\Gamma_\mathrm{b} := \frac{\mu_0}{4\pi} B^2 \frac{(\rho\pi)^{\frac{3}{2}}}{k_B T} \left(x_1\chi_1 + x_2\chi_2\right)^2$$

Das System binärer dipolarer harter Scheiben wird in Abschnitt 4.4 näher beschrieben.

Gliederung Die Kapitel zwei bis vier behandeln die strukturellen statischen Eigenschaften zweidimensionaler Kolloidsysteme. Zunächst werden wir in Kapitel 2 zentrale Größen wie Korrelatoren und den Strukturfaktor definieren, deren Eigenschaften skizzieren und schließlich ein geschlossenes Gleichungssystem für die statischen Strukturfaktoren angeben. Deren numerische Behandlung schließt sich im darauf folgenden Kapitel 3 an, denn im Gegensatz zum dreidimensionalen Fall kann das geschlossene Gleichungssystem für die statischen Korrelatoren nicht mehr analytisch gelöst werden. Die Numerik der Statik in zwei Dimensionen erweist sich jedoch als äußerst fragil, was zunächst überraschend erscheinen mag, da die Dimensionalität letzten Endes nur über eine Fourier-Integration eingeht (2.15). Wir werden also die Problematik der üblicherweise verwendeten Methoden beschreiben (Abschnitt 3.3) und einen gegen Divergenzen stabilen Algorithmus vorstellen (Abschnitt 3.5). Resultate der statischen Strukturfaktoren und der Paarkorrelationsfunktionen für die vier verschiedenen Systeme finden sich in Kapitel 4.

In den Kapiteln fünf bis sieben beschreiben wir die Dynamik und das Glasübergangsverhalten der zweidimensionalen Kolloidsysteme. In Kapitel 5 geben wir eine aphoristische Einführung in die MCT und werden das bis auf die Dimensionalität der Vektoren von der Dimension unabhängige Modenkopplungsfunktional für monodisperse und binäre einfache Flüssigkeiten herleiten. Die konkrete Erweiterung der MCT auf zwei Dimensionen erfolgt dann im thermodynamischen Limes (Abschnitt 5.5). Abschließend wird das geschlossene Gleichungssystem für die Nichtergodizitätsparameter präsentiert. In Kapitel 6 widmen wir uns der numerischen Iteration der Nichtergodizitätsparameter und stellen schließlich in Kapitel 7 die Resultate der MCT vor.

Im Anhang befinden sich unter anderem die Listings des zentral neuen LM-Algorithmus zur Iteration der statischen Korrelatoren in zwei Dimensionen und der Algorithmus zur Iteration der Nichtergodizitätsparameter (Kapitel B). Alle in dieser Arbeit erzeugten Graphen können mit den im Anhang aufgeführten Programmen reproduziert werden.

2. Theorie der statischen Korrelatoren

In diesem zweiten Kapitel wollen wir einen Überblick über die zur strukturellen Beschreibung von Flüssigkeiten und Gläsern essentiellen Größen geben. Dazu gehören die Paarkorrelationsfunktion, die direkte Korrelationsfunktion und der statische Strukturfaktor. Von den beiden letzteren werden wir kurz die asymptotischen Grenzwerte in Ort und Impuls angeben. Außerdem wird sich die Definition eines neuen Korrelators γ im nächsten Kapitel als nützlich erweisen. Wir werden weiter die Ornstein-Zernicke (OZ) Gleichung, die Percus-Yevick (PY) und die Hypernetted-Chain (HNC) Approximation angeben, mit deren Hilfe schließlich ein geschlossenes Gleichungssystem für die statischen Strukturfaktoren formuliert wird. Außerdem sollen die wichtigsten, aus den Korrelatoren zu gewinnenden, physikalischen Einsichten Erwähnung finden. Abkürzender Weise werden die Korrelatoren gleich für zweidimensionale binäre Systeme formuliert.

Konventionen Zunächst wollen wir einige Konventionen für ein beliebiges binäres System treffen. Mit griechischen Indices $\alpha, \beta \in \{1,2\}$ werden wir die Teilchensorten unterscheiden. Es sei N die Gesamtteilchenzahl, F die Gesamtfläche des Systems, $\rho := \frac{N}{F}$ die Gesamtdichte, N_α die Zahl der Teilchen der Sorte α und $\rho_\alpha := \frac{N_\alpha}{F}$ die partielle Dichte der Teilchensorte α. Weiterhin bezeichne $x_\alpha := \frac{N_\alpha}{N}$ die Konzentration der Teilchensorte α. Impuls- und Ortsraum werden wird durch Groß- und Kleinschreibung ($F(q)$ und $f(r)$) unterscheiden. Außerdem benötigen wir sowohl die Matrixdarstellung $\mathbf{f}(r)$ eines Korrelators, als auch dessen Matrixelemente $f_{\alpha\beta}(r)$ mit den o.g. griechischen Indices. Vektoren $\vec{a}$ sind zweidimensional aufzufassen und werden mit Pfeilen gekennzeichnet.

2.1. Korrelationsfunktionen

Lediglich die für die Iteration der statischen Strukturfaktoren wichtigen Aussagen werden hier vorgestellt. Für tiefer gehende Darstellungen verweisen wir auf Lehrbücher (wie z.B. [HM86]).

Wir definieren die *2-Teilchendichte* $\rho^{(2)}_{\alpha\beta}(\vec{x}_i, \vec{x}_j)$ und die *partielle mikroskopische Dichte* $\rho_\alpha(\vec{x})$ als

$$\begin{aligned}
\rho^{(2)}_{\alpha\beta}(\vec{x}_i, \vec{x}_j) &:= \frac{1}{Z_N(F,T)} \frac{N_\alpha! N_\beta!}{(N_\alpha - 2)!(N_\beta - 2)!} \int_{F_\alpha} \int_{F_\beta} \prod_{n \neq i}^{N_\alpha} \prod_{m \neq j}^{N_\beta} d^2 x_n^\alpha d^2 x_m^\beta e^{-\beta V(\vec{x}_1^{(\alpha)}, \ldots, \vec{x}_{N_\beta}^{(\beta)})} \\
\rho_\alpha(\vec{x}) &:= \sum_{n=1}^{N_\alpha} \delta(\vec{x} - \vec{x}_n^{(\alpha)})
\end{aligned}$$

Implizit haben wir damit $\beta := \frac{1}{k_B T}$, die Flächen F_α, F_β und ein Potential $V(\vec{x}_1^{(\alpha)}, \ldots, \vec{x}_{N_\beta}^{(\beta)})$ definiert. Ab

jetzt soll

$$\langle * \rangle := \frac{1}{Z(F,T;N_\alpha,N_\beta)} \int_{F_\alpha} \prod_{n=1}^{N_\alpha} d^2x_n^\alpha \int_{F_\beta} \prod_{m=1}^{N_\beta} d^2x_m^\beta * e^{-\beta V(\vec{x}_1^{(\alpha)},\ldots,\vec{x}_{N_\beta}^{(\beta)})}$$

die thermodynamische Mittelung mit

$$Z(F,T;N_\alpha,N_\beta) := \int_{F_\alpha} \prod_{n=1}^{N_\alpha} d^2x_n^\alpha \int_{F_\beta} \prod_{m=1}^{N_\beta} d^2x_m^\beta e^{-\beta V(\vec{x}_1^{(\alpha)},\ldots,\vec{x}_{N_\beta}^{(\beta)})}$$

bezeichnen.

Dichte-Dichte-Korrelations- und Paarverteilungsfunktion Nun können wir die *partielle Dichte-Dichte-Korrelationsfunktion* über die partielle mikroskopische Dichte definieren:[1]

$$G_{\alpha\beta}(\vec{x}) := \frac{1}{N} \int_F d^2x' \left\langle \rho_\alpha(\vec{x}+\vec{x}')\rho_\beta(\vec{x}') \right\rangle \tag{2.1}$$

Sie lässt sich aufteilen in einen Selbstanteil $G_{\alpha\beta}^{(s)}(\vec{x})$, der die Korrelation eines Teilchens mit sich selbst angibt, und einen Zweieranteil $G_{\alpha\beta}^{(d)}(\vec{x})$, der die Korrelation zweier Teilchen gleicher oder verschiedener Sorte enthält: $G_{\alpha\beta}(\vec{x}) = G_{\alpha\beta}^{(s)}(\vec{x}) + G_{\alpha\beta}^{(d)}(\vec{x})$. Mit der Delta-Distribution $\delta(\vec{x})$ lauten die beiden Anteile dann:

$$\begin{aligned}
G_{\alpha\beta}^{(s)}(\vec{x}) &:= x_\alpha \delta_{\alpha\beta}\delta(\vec{x}) \\
G_{\alpha\beta}^{(d)}(\vec{x}) &:= \frac{\delta_{\alpha\beta}}{N} \sum_{n\neq m}^{N_\alpha} \left\langle \delta(\vec{x} - [\vec{x}_n^{(\alpha)} - \vec{x}_m^{(\beta)}]) \right\rangle + \frac{1-\delta_{\alpha\beta}}{N} \sum_{n=1}^{N_\alpha}\sum_{m=1}^{N_\beta} \left\langle \delta(\vec{x} - [\vec{x}_n^{(\alpha)} - \vec{x}_m^{(\beta)}]) \right\rangle \\
&= \frac{1}{N} \int_F d^2x' \rho_{\alpha\beta}^{(2)}(|\vec{x}-\vec{x}'|)
\end{aligned}$$

Die *Paarverteilungsfunktion* ist für homogene Systeme definiert als

$$g_{\alpha\beta}^{(2)}(\vec{x}_i,\vec{x}_j) := \frac{\rho_{\alpha\beta}^{(2)}(\vec{x}_i,\vec{x}_j)}{\rho_\alpha\rho_\beta} \tag{2.2}$$

Für ein isotropes[2] System lässt sich leicht zeigen, dass $\rho_{\alpha\beta}^{(2)}(\vec{x}_i,\vec{x}_j) = \rho_{\alpha\beta}^{(2)}(|\vec{x}_i-\vec{x}_j|)$. Dies impliziert, dass die beiden Korrelatoren (2.1) und (2.2) nur vom Abstand $r := |\vec{x}_i - \vec{x}_j|$ abhängig sind. Die Ortsabhängigkeit wird daher ab jetzt mit r bezeichnet.

Da $g_{\alpha\beta}^{(2)}(r) = 1 - \frac{1}{N}$ für $r \to \infty$ folgt für $N \to \infty$: $g_{\alpha\beta}^{(2)}(r) = 1$. Die Korrelation zwischen den Teilchen im Unendlichen soll aber auf Null abgefallen sein; also definieren wir zusätzlich die *totale Korrelationsfunktion*

$$h_{\alpha\beta}(r) := g_{\alpha\beta}^{(2)}(r) - 1 \tag{2.3}$$

[1] Hierbei weichen wir kurz von unserer Konvention für Orts- und Impulsraum ab, da die Dichte-Dichte-Korrelationsfunktion nur in diesem Abschnitt gebraucht wird.

[2] Die von uns betrachteten Systeme sind isotrop.

Der Zweieranteil der partiellen Dichte-Dichte-Autokorrelationsfunktion kann mit der Paarverteilungsfunktion $\mathbf{g}(r)$ umgeschrieben werden zu:

$$G^{(d)}_{\alpha\beta}(r) = \frac{\rho_\alpha \rho_\beta}{\rho} g_{\alpha\beta}(r)$$

Legt man den Koordinatenursprung zentral in ein festes Teilchen, so ist die Paarkorrelationsfunktion eine Wahrscheinlichkeitsdichte, die ein Maß dafür darstellt, wie viele Teilchen in einem Abstand r vom betrachteten Teilchen im Mittel zu finden sein werden. Ihre Maxima geben gerade die mittleren Positionen der nächsten, übernächsten etc. Nachbarn an, deren Breite ein Maß ihrer Lokalisierung ist. Da $\mathbf{g}(r)$ nun eine Wahrscheinlichkeitsdichte ist, gibt $2\pi\rho \int_{r_1}^{r_2} \mathbf{g}(r) r dr$ die Anzahl der Teilchen im Intervall $[r_1, r_2]$ an. Durch Integration zwischen den lokalen Minima kann so insbesondere die Anzahl der jeweiligen nächsten Nachbarn errechnet werden (siehe (4.2)).

Der statische Strukturfaktor Der statische Strukturfaktor ist im Gegensatz zu den beiden vorherigen Korrelatoren im Impulsraum definiert. Genauer ist er über die Dichtefluktuation $\delta\rho_\alpha(\vec{q}) = \rho_\alpha(\vec{q}) - \langle \rho_\alpha(\vec{q}) \rangle = \rho_\alpha(\vec{q}) - \delta_{\vec{q},\vec{0}} N_\alpha$ definiert als:

$$S_{\alpha\beta}(\vec{q}) := \frac{1}{N} \left\langle \delta\rho_\alpha(\vec{q})^* \delta\rho_\beta(\vec{q}) \right\rangle = \frac{1}{N} \left\langle \rho_\alpha(\vec{q})^* \rho_\beta(\vec{q}) \right\rangle - \frac{\delta_{\vec{q},\vec{0}} N^*_\alpha N_\beta}{N} \tag{2.4}$$

Das heißt, dass die fouriertransformierte Dichte-Dichte-Autokorrelationsfunktion dem statischen Strukturfaktor für $q \neq 0$ entspricht: $S_{\alpha\beta}(\vec{q}) = \mathcal{F}\left[G_{\alpha\beta}(\vec{x})\right](\vec{q})$. Ab hier soll für $S_{\alpha\beta}(\vec{q})$ immer $\vec{q} \neq \vec{0}$ gelten.

Da die Dichte-Dichte-Korrelationsfunktion nur abhängig vom Abstand r ist, lässt sich leicht zeigen, dass auch deren Fouriertransformierte nur von $q := |\vec{q}|$ abhängig ist. Wegen der Linearität der Fouriertransformation überträgt sich auch die Aufteilung in Selbst- und Zweieranteil so, dass $S_{\alpha\beta}(q) = S^{(s)}_{\alpha\beta}(q) + S^{(d)}_{\alpha\beta}(q)$ mit:

$$\begin{aligned} S^{(s)}_{\alpha\beta}(q) &= \delta_{\alpha\beta} x_\alpha \\ S^{(d)}_{\alpha\beta}(q) &= \frac{\rho_\alpha \rho_\beta}{\rho} h_{\alpha\beta}(q) \end{aligned} \tag{2.5}$$

Zusammenhang $\rho \leftrightarrow T$ Für Potentiale der Form $V(r) \sim r^{-n}$ lässt sich allgemein für ein d-dimensionales System ein Zusammenhang zwischen Temperatur und Dichte zeigen: Für gewisse Temperatur-Dichte-Paare ist der Strukturfaktor gleich. Wir werden diese Äquivalenz für monodisperse Systeme kurz herleiten. Zunächst gilt: $S(q) = 1 + \rho H(q) = 1 + \rho \int_{\mathbb{R}} d\vec{r} e^{-i\vec{q}\cdot\vec{r}} g(r) = 1 + N \left\langle e^{i\vec{q}\cdot[\vec{r}_1 - \vec{r}_2]} \right\rangle$. Mit der Substitution $\bar{r} := \rho^{-\frac{1}{d}}$ und $\bar{T} := T\vec{r}^{\,n}$ folgt sofort:

$$S(q;T;\rho) = \tilde{S}\left(\bar{q} = q\rho^{\frac{1}{d}}; \bar{T} = T\rho^{\frac{n}{d}}\right) \tag{2.6}$$

Also zeigt (2.6), dass $S(q;T,\rho)$ mit der Skalenfunktion $\tilde{S}(\bar{q};\bar{T})$ allein von $\bar{T} = T\rho^{\frac{n}{d}}$ abhängt. Diese Aussage gilt analog für binäre Systeme. Auf diesem Zusammenhang fußt die Definition der mittleren magnetischen Wechselwirkung in (4.7) und (4.14) als physikalisch relevantem Systemparameter.

Die direkte Korrelationsfunktion Die direkte Korrelationsfunktion $c_{\alpha\beta}(q)$ werden wir implizit definieren. Zur Motivation betrachten wir das quantenmechanische Problem, den Propagator $G(t) = e^{-\frac{i}{\hbar}H(t)t}$ für den gestörten Hamiltonoperator $H(t) = H_0(t) + V$ nach dem ungestörten Propagator $G_0(t) = e^{-\frac{i}{\hbar}H_0(t)t}$ zu entwickeln. Dies geht leicht über die inversen laplacetransformierten Operatoren: Bezeichne $\hat{G}(z) := \mathcal{L}[G(t)] = i\int_{\mathbb{R}^+} dt G(t)e^{izt} = -\frac{\hbar}{\hbar z - H}$ und analog $\hat{G}_0(z)$ die laplacetransformierten Propagatoren, so kann der inverse laplacetransformierte Propagator des Systems dargestellt werden als $\hat{G}^{-1}(z) = \hat{G}_0^{-1}(z) + \frac{V}{\hbar}$. Hier soll nun die direkte Korrelationsfunktion die Störung in Form der indirekten Wechselwirkung zwischen Teilchen über die nächsten Nachbarn darstellen. Wir definieren sie, indem wir den inversen statischen Strukturfaktor über seinen inversen Selbstanteil darstellen. In Matrixschreibweise heißt das:

$$\mathbf{S}^{-1}(q) =: \mathbf{S}^{(s)^{-1}}(q) - \rho\mathbf{c}(q) \tag{2.7}$$

Der Korrelator γ Um die Definition von γ zu motivieren, betrachten wir das zweidimensionale System monodisperser harter Scheiben, deren Durchmesser mit σ bezeichnet sei. Gehen wir weiter zu reduzierten Einheiten über und geben Ort und Impuls in Einheiten des Durchmessers an ($\frac{r}{\sigma}$ und $q\sigma$), so ist das zugehörige Potential (4.1) an der Stelle $r\sigma^{-1} = 1$ unstetig. Diese Unstetigkeit im Potential bedingt eine Unstetigkeit in den Korrelatoren $h(r)$ und $c(r)$. Für $h(r)$ ist das aus der Percus-Yevick- (2.11) oder Hypernetted-Chain-Approximation (2.10) ersichtlich. Die Unstetigkeit in $c(r)$ wird unten diskutiert. Wie wir im nächsten Kapitel (Abschnitt 3.4) beschreiben werden, ist es jedoch notwendig, eine stetige Funktion zu iterieren. Diese Unstetigkeit bei $r\sigma^{-1} = 1$ tritt nicht auf in

$$\gamma(r) := \mathbf{h}(r) - \mathbf{c}(r) \tag{2.8}$$

Um die Stetigkeit von $\gamma(r)$ zu beweisen, greifen wir auf die Ornstein-Zernicke-Gleichung (OZ) (siehe Abschnitt 2.2) zurück. Angenommen $h(r)$ sei eine stetige Funktion, die nur bei $r\sigma^{-1} = 1$ eine Unstetigkeit der Höhe Δ_h hat, dann kann eine Zerlegung vorgenommen werden: $h(r) =: [1 - \theta(r\sigma^{-1} - 1)]h_1(r\sigma^{-1}) + \theta(r\sigma^{-1} - 1)h_2(r\sigma^{-1})$ mit $\Delta_h = h_2(\frac{r}{\sigma}^+) - h_1(\frac{r}{\sigma}^-)$. In erster Ordnung gilt nach OZ (2.9): $C(q) \approx H(q)$. Weil aber weiter die Fouriertransformation stetig ist, folgt aus der approximierten OZ-Gleichung, dass auch $c(r)$ eine Unstetigkeit bei $r\sigma^{-1} = 1$ mit gleicher Sprunghöhe hat. Kurz: $\Delta_h = \Delta_c$. Zerlegt man analog zu $h(r)$ nun auch $c(r) =: [1 - \theta(r\sigma^{-1} - 1)]c_1(r) + \theta(r\sigma^{-1} - 1)c_2(r)$ mit $\Delta_c = c_2(\frac{r}{\sigma}^+) - c_1(\frac{r}{\sigma}^-)$, so folgt sofort: $\lim_{\varepsilon\to 0}[\gamma(r\sigma^{-1} + \varepsilon) - \gamma(r\sigma^{-1} - \varepsilon)] = \Delta_h - \Delta_c = 0$ und die Stetigkeit von $\gamma(r)$ ist gezeigt. Der Beweis für binäre Systeme verläuft analog.

2.2. Ornstein-Zernicke-Gleichung

Die Ornstein-Zernicke-Gleichung lässt sich wie folgt in binärer Schreibweise herleiten (zu Gunsten der Übersichtlichkeit verzichten wir auf die Impulsabhängigkeit):

$$\delta_{\alpha\beta} = (\mathbf{S}\mathbf{S}^{-1})_{\alpha\beta} = \sum_\gamma [x_\alpha\delta_{\alpha\gamma} + \frac{\rho_\alpha\rho_\gamma}{\rho}H_{\alpha\gamma}][\frac{1}{x_\gamma}\delta_{\gamma\beta} - \rho C_{\gamma\beta}] \Leftrightarrow$$

$$\frac{\rho_\alpha}{\rho}\sum_\gamma \rho_\gamma H_{\alpha\gamma}\frac{\delta_{\gamma\beta}}{x_\gamma} = \delta_{\alpha\beta} - x_\alpha \sum_\gamma \frac{\delta_{\alpha\gamma}\delta_{\gamma\beta}}{x_\gamma} + x_\alpha \rho \sum_\gamma \delta_{\alpha\gamma} C_{\gamma\beta} \Leftrightarrow$$

$$\rho H_{\alpha\beta} = \rho C_{\alpha\beta} + \rho \sum_\gamma H_{\alpha\gamma} C_{\gamma\beta} \rho_\gamma \Leftrightarrow H_{\alpha\beta}(q) = C_{\alpha\beta}(q) + \sum_\gamma H_{\alpha\gamma}(q)\rho_\gamma C_{\gamma\beta}(q)$$

Damit ist die Ornstein-Zernicke-Gleichung gefunden. Führen wir nun noch die Matrizen $\mathbf{P} := diag(\rho_1, \rho_2)$ und $\mathbb{1} := diag(1,1)$ ein, so lässt sich die Ornstein-Zernicke-Gleichung in Matrixform schreiben:

$$\mathbf{C}(q) = [\mathbb{1} + \mathbf{H}(q)\mathbf{P}]\mathbf{C}(q) \tag{2.9}$$

2.3. Percus-Yevick und Hypernetted-Chain

PY- und HNC-Approximation Für die (diagrammatische) Begründung der PY- und HNC-Näherung verweisen wir auf [Ste, HM86] und geben mit der Potentialmatrix $\mathbf{v}(r) := \begin{pmatrix} v_{11}(r) & v_{12}(r) \\ v_{21}(r) & v_{22}(r) \end{pmatrix}$ direkt an:

$$\text{HNC} \qquad \mathbf{g}(r) = e^{-\beta \mathbf{v}(r)} \cdot e^{\mathbf{h}(r) - \mathbf{c}(r)} \tag{2.10}$$

$$\text{PY} \qquad \mathbf{g}(r) = e^{-\beta \mathbf{v}(r)} \cdot [\mathbf{g}(r) - \mathbf{c}(r)] \tag{2.11}$$

Dabei bedeutet die Operation $\cdot$, dass es sich hierbei um eine komponentenweise Multiplikation handelt: $\mathbf{c} = \mathbf{a} \cdot \mathbf{b} :\Leftrightarrow c_{ij} = a_{ij} b_{ij}$. Wie zu erkennen ist, ergibt sich die PY-Gleichung durch Linearisieren der HNC-Gleichung bezüglich $[\mathbf{h} - \mathbf{c}]$, woraus resultiert, dass die HNC- der PY-Approximation überlegen ist [HM86]. Beide Approximationen geben den korrekten Verlauf $\mathbf{g}(r) \sim e^{-\beta \mathbf{v}(r)}$ für $\rho \ll 1$ wieder; insbesondere werden beide Näherungen für $\rho \to 0$ exakt. Beides werden wir in Kapitel 4 überprüfen.

PY exakt gelöst Wurde die PY-Approximation in drei Dimensionen für monodisperse Systeme noch exakt gelöst [Wer63],

$$c(r) = \begin{cases} -\frac{(1+2\eta)^2}{(1-\eta)^4} + \frac{6\eta(1+\frac{\eta}{2})^2}{(1-\eta)^4} r - \frac{\eta(1+2\eta)^2}{2(1-\eta)^4} r^3 & r \le 1 \\ 0 & r > 1 \end{cases} \tag{2.12}$$

so ist dies generell in geraden Dimensionen nicht möglich [MB87]. Die direkte Korrelationsfunktion (2.12) ist in reduzierten Einheiten angegeben. Für binäre Systeme konnte diese Lösung verallgemeinert werden [Leb63]. Insbesondere im vorliegenden zweidimensionalen Fall muss die PY-Approximation numerisch behandelt werden. Diese numerischen Resultate werden wir in Abschnitt 4.1 mit den exakt PY-approximierten Resultaten in drei Dimensionen vergleichen.

2.4. Asymptotik der Korrelatoren

Bei der Diskussion der asymptotischen Grenzwerte der Korrelatoren werden wir uns auf die Paarverteilungsfunktion $\mathbf{g}(r)$ und den Strukturfaktor $\mathbf{S}(q)$ beschränken. Wir betrachten die Grenzwerte für große und kleine Orte r bzw. Impulse q. Weiter geben wir an, wie sich die Korrelatoren bei kleiner Dichte ρ verhalten.

Grenzwerte der Paarverteilungsfunktion $\mathbf{g}(r)$ Da für $r \to \infty$ die Zweiteilchendichte separiert und $\left\langle \rho_\alpha(r^{(\alpha)}) \right\rangle = \rho_\alpha$ gilt, folgt:

$$\lim_{r\to\infty} g_{\alpha\beta}(r) = 1$$

Wie zuvor bereits erwähnt soll die Korrelation zweier Teilchen allerdings in diesem Falle Null sein, was direkt auf die Definition der totalen Korrelationsfunktion $\mathbf{h}(r) := \mathbf{g}(r) - 1$ führt. Es ist weiter klar, dass keine zwei Teilchen in sehr geringem Abstand zu finden sein werden. Dies gilt insbesondere für die von uns betrachteten Systeme, die alle einen Harte-Scheiben-Anteil im Potential haben. Für unsere binären Systeme mit den Teilchendurchmessern $\sigma_{\alpha\beta} := r_\alpha + r_\beta$ ist dann stets $V_{\alpha\beta}(\frac{r}{\sigma_{12}} \leq \frac{\sigma_{\alpha\beta}}{\sigma_{12}}) = \infty$. Damit gilt $G^{(d)}_{\alpha\beta}(\frac{r}{\sigma_{12}} \leq \frac{\sigma_{\alpha\beta}}{\sigma_{12}}) = 0$ und es folgt sofort:

$$g_{\alpha\beta}(r) = 0 \qquad \frac{r}{\sigma_{12}} \leq \frac{\sigma_{\alpha\beta}}{\sigma_{12}}$$

Grenzwerte des Strukturfaktors $\mathbf{S}(q)$ Wir diskutieren zunächst den Fall $q \to \infty$. Wegen der Linearität der thermodynamischen Mittelung kann der Selbstanteil des Strukturfaktors abgespalten werden:

$$\begin{aligned}
\lim_{q\to\infty} S_{\alpha\beta}(q) &= \lim_{q\to\infty} S^{(s)}_{\alpha\beta}(q) + \lim_{q\to\infty} S^{(d)}_{\alpha\beta}(q) \qquad (2.13)\\
S^{(s)}_{\alpha\beta}(q) &:= \frac{1}{N}\sum_{n=1}^{N_\alpha}\sum_{m=1}^{N_\beta}\left\langle \delta_{\alpha\beta}\delta_{nm} e^{-i\vec{q}\cdot(\vec{x}_n^{(\alpha)} - \vec{x}_m^{(\beta)})}\right\rangle = \delta_{\alpha\beta} x_\alpha\\
S^{(d)}_{\alpha\beta}(q) &:= \frac{1}{N}\sum_{n=1}^{N_\alpha}\sum_{m=1}^{N_\beta}\left\langle \left(\delta_{\alpha\beta}\left[1-\delta_{nm}\right] + \left[1-\delta_{\alpha\beta}\right]\right) e^{-i\vec{q}\cdot(\vec{x}_n^{(\alpha)} - \vec{x}_m^{(\beta)})}\right\rangle
\end{aligned}$$

Dabei kann der Term $S^{(d)}_{\alpha\beta}(q)$, der nur verschiedene Teilchen beinhaltet, in (2.13) im Limes $q \to \infty$ mit dem Lemma von Riemann-Lebesgue [Olv74] berechnet werden und ergibt Null. Daher folgt mit (2.5) für den Strukturfaktor im Limes $q \to \infty$:

$$\lim_{q\to\infty} S_{\alpha\beta}(q) = \lim_{q\to\infty} \delta_{\alpha\beta} x_\alpha = \delta_{\alpha\beta} x_\alpha$$

Weiterhin gilt für $q \to 0$:

$$\lim_{q\to 0} S_{\alpha\beta}(q) = \frac{1}{N}\left(\sum_{n=1}^{N_\alpha}\sum_{m=1}^{N_\beta}\left\langle e^{-i\vec{q}\cdot(\vec{x}_n^{(\alpha)} - \vec{x}_m^{(\beta)})}\right\rangle - \delta_{\vec{q},\vec{0}} N_\alpha N_\beta\right) = 0$$

Geringe Dichten Mit Hilfe dieser Relationen kann die Frage beantwortet werden, wie sich die Korrelatoren mit der Dichte ändern. Dazu rufen wir uns zunächst ins Gedächtnis, dass die Dichte mit dem mittleren Abstand und dem mittleren Impuls der Teilchen über $\rho \to 0 \Leftrightarrow \bar{r} \to \infty \Leftrightarrow \bar{q} \to 0$ zusammenhängt. Schließen wir nun die OZ-Gleichung mit der PY-Approximation, so folgt unter der physikalisch sinnvollen Annahme, dass $\lim_{r\to\infty} \mathbf{V}(r) = \mathbf{0}$ für kleine Dichten:

$$\begin{aligned} \mathbf{c}(\bar{r}) &\approx [e^{-\beta\mathbf{0}} - 1] \cdot [\mathbf{g}(\bar{r}) - \mathbf{c}(\bar{r})] = \mathbf{0} \\ \mathbf{H}(\bar{q}) &= [\mathbb{1} + \mathbf{PC}(\bar{q})]^{-1}\mathbf{C}(\bar{q}) \approx \mathbf{C}(\bar{q}) \end{aligned}$$

Setzt man voraus, dass $\mathbf{V}(r)$ stetig fällt, bedeutet dies, dass die Korrelatoren allesamt mit sinkender Dichte flacher werden. Für den Strukturfaktor ist das besonders anschaulich, weil er bei Streuexperimenten die Struktur des Targets beschreibt.

2.5. Geschlossenes Gleichungssystem

Wir werden nun zwei geschlossene Gleichungssysteme (GLS) zur numerischen Iteration der binären statischen Korrelatoren angeben. Für den monodispersen Fall verweisen wir auf [HM86, Gil79]. Das erste GLS dient zur Iteration der Korrelatoren **h**,**c**. Im nächsten Kapitel werden wir zeigen, dass es notwendig ist zu dem bei $\frac{r}{\sigma} = 1$ stetigen Korrelator γ überzugehen, der zum zweiten GLS führt. Der statische Strukturfaktor kann vermöge (2.5) berechnet werden.

Da die OZ-Gleichung von uns direkt im Impulsraum und die PY- bzw. HNC-Approximation im Ortsraum formuliert ist, wird das GLS durch die Fouriertransformation[3] (FT) $\mathcal{F}[\mathbf{f}(\vec{r})]$

$$\begin{aligned} \mathbf{F}(\vec{q}) &= \mathcal{F}[\mathbf{f}(\vec{r})] := \frac{1}{2\pi}\int_{\mathbb{R}^2} d\vec{r} e^{i\vec{q}\cdot\vec{r}}\mathbf{f}(\vec{r}) \\ \mathbf{f}(\vec{r}) &= \mathcal{F}^{-1}[\mathbf{F}(\vec{q})] := \frac{1}{2\pi}\int_{\mathbb{R}^2} d\vec{q} e^{-i\vec{q}\cdot\vec{r}}\mathbf{F}(\vec{q}) \end{aligned} \tag{2.14}$$

und ihre Inverse $\mathcal{F}^{-1}[\mathcal{F}[\mathbf{f}(\vec{r})]] = \mathbf{f}(\vec{r})$ geschlossen. Im Gegensatz zum dreidimensionalen Fall kann der Winkelanteil nicht mehr herausintegriert werden. In drei Dimensionen kann die Volumenintegration in Kugelkoordinaten geschrieben werden als $\int_{\mathbb{R}^3} d^3 r f(r) = \int_0^\pi d\vartheta \int_0^{2\pi} d\varphi \int_{\mathbb{R}^+} dr r^2 \sin(\vartheta) f(r)$. Da weiter $f(r)$ eine nur vom Radius abhängige Funktion ist, lässt sich durch Substitution der Winkelanteil herausintegrieren zu $4\pi \int_{\mathbb{R}^+} dr r^2 f(r)$. In zwei Dimensionen kann das Flächenintegral in Polarkoordinaten formuliert werden: $\int_{\mathbb{R}^2} d^2 r f(r) = \int_{\mathbb{R}^+} dr r f(r) \int_0^{2\pi} d\varphi e^{iqr\cos(\varphi)}$. Führen wir mit $J_0(x) := \frac{1}{\pi}\int_0^\pi d\varphi e^{ix\cos(\varphi)}$ die nullte Besselfunktion ein, für die aufgrund ihrer Symmetrie $\int_0^{2\pi} d\varphi e^{ix\cos(\varphi)} = 2\pi J_0(x)$ gilt, so folgt für die FT:

$$\begin{aligned} \mathcal{F}[\mathbf{f}(r)] &= \int_{\mathbb{R}^+} dr r \mathbf{f}(r) J_0(qr) \\ \mathcal{F}^{-1}[\mathbf{F}(q)] &= \int_{\mathbb{R}^+} dq q \mathbf{F}(q) J_0(qr) \end{aligned} \tag{2.15}$$

[3] Die FT ist hier symmetrisch definiert.

Diese spezielle Form der FT bezeichnet man auch als *Hankel-Transformation* (HT). Nun können wir die GLS formulieren, die wir im nächsten Kapitel numerisch behandeln werden.

Iteration von h Zusammen mit der FT (2.15), der OZ-Gleichung

$$\mathbf{H}(q) = \mathbf{C}(q)[\mathbb{1} - \mathbf{PC}(q)]^{-1} \tag{2.16}$$

und der HNC- bzw PY-Approximation

$$\begin{aligned} \mathbf{h}(r) &= e^{-\beta\mathbf{v}(r)} \cdot e^{\mathbf{h}(r)-\mathbf{c}(r)} + 1 \\ \mathbf{h}(r) &= e^{-\beta\mathbf{v}(r)} \cdot [\mathbf{h}(r) + 1 - \mathbf{c}(r)] + 1 \end{aligned} \tag{2.17}$$

ergibt sich ein geschlossenes GLS für die Korrelatoren **h** und **c**.

Iteration von γ Das geschlossene GLS für die Korrelatoren γ und **c** ergibt sich abermals aus der FT (2.15), der OZ-Gleichung in der Form

$$\Gamma(q) = \mathbf{C}(q)\mathbf{PC}(q)[\mathbb{1} - \mathbf{PC}(q)]^{-1} \tag{2.18}$$

und der HNC- bzw PY-Approximation

$$\begin{aligned} \mathbf{c}(r) &= e^{-\beta\mathbf{v}(r)+\gamma(r)} - \gamma(r) - 1 \\ \mathbf{c}(r) &= [1 + \gamma(r)] \cdot [e^{-\beta\mathbf{v}(r)} - 1] \end{aligned} \tag{2.19}$$

3. Numerik der statischen Korrelatoren

Dieses Kapitel stellt die drei von uns verwendeten Algorithmen zur Iteration der statischen Korrelatoren vor. Zunächst diskretisieren wir Orts- und Impulsraum und führen die Fourier-Bessel-Transformation ein. Nach einigen allgemein für alle Algorithmen gültigen Vorbemerkungen beschreiben wir, wie das reine Picardverfahren deutlich verbessert werden kann. Wir werden aufzeigen, dass dieses noch nicht in der Lage ist, Lösungen im für die Glasphysik relevanten Bereich starker Wechselwirkungen zu finden. Einen Ausweg für die monodispersen (dipolaren) harten Scheiben werden wir dann mit dem Algorithmus von Gillan beschreiten, der jedoch die binären Systeme, wegen des zu hohen Rechenaufwands, nicht zu lösen vermag. Schließlich beschreiben wir noch einen kombinierten Gillan-/Levenberg-Marquardt-Algorithmus (kurz LM-Algorithmus), mit dem es uns gelang, auch die binären Systeme zu lösen. Die Struktur der Algorithmen werden wir mit Hilfe von Pseudo-Code beschreiben und verweisen für die konkrete Realisierung auf den Quelltext im Anhang. Die Algorithmen sind hier so formuliert, dass sie möglichst klar die zentralen Punkte verdeutlichen. Für jeden Algorithmus werden wir kurz darstellen, welche Parameter er hat und wie er auf ihre Variation reagiert. Die Parameter der physikalischen Systeme werden wir im nächsten Kapitel diskutieren.

3.1. Diskretisierung und Fourier-Bessel-Transformation

Konventionen Für die Diskretisierung wollen wir wieder eine Konvention treffen: i, j, k sollen im folgenden Orts- und m, n, o Impulsindices beschreiben, die jeweils zwischen 0 und $M-1$ der Anzahl der Stützstellen variieren[1], die im Orts- und Impulsraum gleich groß gewählt wurde. Die Extrema von r und q nennen wir: $r_{min} := r_{i=0}$, $R := r_{i=M-1}$ und $q_{min} := q_{m=0}$, $Q := q_{m=M-1}$. Weiter bezeichnen wir mit $f_i := f(r_i)$ und $F_m := F(q_m)$ diskretisierte Funktionen im Orts- und Impulsraum und definieren $\mathbf{f} := (f_0, ..., f_{M-1})$ und $\mathbf{F} := (F_0, ..., F_{M-1})$.

Diskretisierung Wie vor Gleichung (2.15) erwähnt, kann der Winkelanteil nicht wie in drei Dimensionen herausintegriert werden und steckt in der Besselfunktion $J_0(x)$. Das führt zu einem Problem bei der Diskretisierung. Angenommen wir wählen eine äquidistante Diskretisierung der Form $r_i = r_0 + i\Delta r$ und $q_m = q_0 + m\Delta q$. Dann gilt für die FBT $F_m \sim \sum_{j=0}^{M-1} f_j J_0(r_j q_m)$. Das aus der Diskretisierung folgende Argument $r_j q_m$ der Besselfunktion hat dabei nicht deren Periode. Daraus folgen Finite-Size-Effekte, die sich in kleinen Oszillationen der Korrelatoren äußern. Diese Problematik kann jedoch leicht umgangen werden, indem man eine nicht-äquidistante Diskretisierung mit der Periode der nullten Besselfunktion vornimmt wie in [Lad68] beschrieben. Genau wie dort beschrieben haben wir die Stützstellen so gewählt, dass das Argument der Besselfunktion deren Periode hat:

[1] Dabei wurde der erste Index Null gewählt, um Konsistenz mit den C-Programmen zu wahren.

$$\begin{aligned} r_i &:= \frac{\lambda_i}{Q} \\ q_m &:= \frac{\lambda_m}{R} \end{aligned} \tag{3.1}$$

Wir haben je nach System $R = 20...50$ (für die dipolaren Systeme teilweise auch größer) gewählt und dann den maximalen Impuls mittels $Q = \frac{\lambda_{M-1}}{R}$ errechnet. Die Nullstellen der Besselfunktion J_0 lassen sich nach [GR80] auf mindestens 10^{-5} genau mittels[2]

$$\lambda_i \approx \frac{\pi(4i-1)}{4} + \frac{1}{2\pi(4i-1)} - \frac{31}{6\pi^3(4i-1)} + \frac{3779}{15\pi^5(4i-1)^5}$$

berechnen. Im folgenden Pseudo-Code wollen wir diese Diskretisierung an M Stützstellen durch *discretization* r_i, q_m, M bezeichnen.

Fourier-Bessel-Transformation Um die HT auf die Korrelatoren erfolgreich wirken zu lassen, müssen sie lediglich schnellfallend sein. Dass die Korrelatoren diese Eigenschaft haben, ist in [Göt89, HM86] gezeigt. Definieren wir nun die HT asymmetrisch wie in [Lad68][3], so lautet sie diskretisiert:

$$\begin{aligned} F_m &= \frac{4\pi}{Q^2} \sum_{j=0}^{M-1} f_j \frac{J_0(q_m r_j)}{J_1^2(Q r_j)} \\ f_i &= \frac{1}{\pi R^2} \sum_{n=0}^{M-1} F_n \frac{J_0(q_n r_i)}{J_1^2(q_n R)} \end{aligned} \tag{3.2}$$

Die Äquivalenz der FBT zur HT (2.15) für eine beliebige Funktion $f(r)$ lässt sich über eine *Fourier-Bessel-Entwicklung* der zu transformierenden Funktion zeigen. Gehen wir zunächst von unendlich vielen Stützstellen aus, so lautet die diskretisierte HT $f_i = \sum_{m=0}^{\infty} J_0(q_m r_i) \frac{q_m}{2\pi} F_m$ und die Fourier-Bessel-Entwicklung derselben Funktion ist $f_i = \sum_{n=0}^{\infty} J_0(q_n r_i) \cdot \frac{2}{J_1^2(q_n)} \int_0^1 dt\, t f(t) J_0(q_n t)$. Die Äquivalenz lässt sich über elementare Rechnung zeigen, indem das Integral im letzteren Ausdruck mit F_n in Verbindung gebracht wird. (3.2) heißt daher auch *Fourier-Bessel-Transformation*. Um den Rechenaufwand zu minimieren, haben wir die $2M^2$ Konstanten $const(m,j) := \frac{4\pi}{Q^2} \frac{J_0(q_m r_j)}{J_1^2(Q r_j)}$ und $const(i,n) := \frac{1}{\pi R^2} \frac{J_0(q_n r_i)}{J_1^1(q_n R)}$ bei jedem Durchlauf nur einmal berechnet. In den Algorithmen werden wir sowohl Hin-, als auch Rücktransformation mit *FBT* bezeichnen.

[2] Wir haben ohne Einfluss auf den Algorithmus und dessen konvergente Lösungen die Nullstellen im Levenberg-Marquardt-Algorithmus mit Hilfe von [GSL05] approximiert, um Rechenzeit bei der Initialisierung einzusparen.

[3] Im Gegensatz zu [Lad68] haben wir den letzten Summanden $M-1$ nicht vernachlässigt.

3.2. Vorbemerkungen

Reduzierte Einheiten Um unnötige Rechenzeit und Rundungsfehler zu vermeiden und die Übersichtlichkeit zu wahren, gehen wir nun zu reduzierten Einheiten über. Der Ort r wird in Einheiten des Teilchendurchmessers σ gemessen und damit der Impuls in Einheiten σ^{-1}. Weiter benutzen wir, wie in Kapitel 1, die Packungsdichte $\eta := n\sigma^2\frac{\pi}{4}$, die den Bedeckungsgrad angibt. Der für die dipolaren Systeme bei geringen Packungsdichten physikalisch relevante Kontrollparameter ist die mittlere magnetische Wechselwirkung Γ (siehe Kapitel 1) - nicht zu verwechseln mit der fouriertransformierten des Korrelators γ.

Konventionen Da wir in diesem Kapitel Iterationsverfahren beschreiben wollen, führen wir einen Index zur Unterscheidung der Iterationsschritte ein. Sei $f(r)$ eine beliebige zu iterierende Funktion, dann bezeichne $f^{(n)}(r)$ die Funktion nach dem n-ten Iterationsschritt. Führt man einen *Iterationsoperator*

$$I[f^{(n)}(r)] := f^{(n+1)}(r) \tag{3.3}$$

ein, so gilt für jeden Fixpunkt (FP) $I[f^{(\mathrm{fix})}(r)] = f^{(\mathrm{fix})}(r)$. Wir gehen ohne Beweis davon aus, dass unsere Iterationsoperatoren, die durch die GLS in Abschnitt 2.5 gegeben sind, einen eindeutigen physikalischen FP liefern, falls der Startwert $f^{(0)}(r)$ nur nahe genug diesem erwarteten FP gewählt ist.

Symmetrie der Lösungen Nun soll $\mathbf{f}(r)$ einen zu iterierenden matrixwertigen Korrelator darstellen. Die Symmetrie des Potentials (4.10) (bzw. dipolar (4.13)) bezüglich der Teilchenvertauschung überträgt sich auf die FPe der GLS:

$$V_{\alpha\beta}(r) = V_{\beta\alpha}(r) \quad \Rightarrow \quad f_{\alpha\beta}^{(\mathrm{fix})}(r) = f_{\beta\alpha}^{(\mathrm{fix})}(r)$$

Diese Symmetrie ist allerdings im Allgemein bei einem n-ten Iterationsschritt, der noch nicht der gesuchte FP ist, gebrochen. Konvergiert die Folge der $f_{\alpha\beta}^{(n)}(r)$ jedoch gegen den FP, so gilt (ohne Beweis): $|f_{\alpha\beta}^{(n)}(r) - f_{\beta\alpha}^{(n)}(r)| - |f_{\alpha\beta}^{(n+1)}(r) - f_{\beta\alpha}^{(n+1)}(r)| \to 0$. Diese Konvergenz der Symmetrieeigenschaft kann man sich zu Nutze machen, um das binäre Iterationsverfahren stabiler und schneller zu machen, indem nach jedem Iterationsschritt symmetrisiert wird mittels:

$$f_{\alpha\beta}^{(n),sym}(r) = \delta_{\alpha\beta} f_{\alpha\beta}^{(n)}(r) + \frac{1-\delta_{\alpha\beta}}{2}\left(f_{\alpha\beta}^{(n)}(r) + f_{\beta\alpha}^{(n)}(r)\right) \tag{3.4}$$

Weil eine mögliche Asymmetrie in den Korrelatoren nur durch die OZ-Gleichung (2.16) oder (2.18) entsteht, genügt es einmal nach Anwenden dieser Gleichung zu symmetrisieren. In den folgenden Algorithmen werden wir diese Symmetrisierung *symmetrize* nennen. Weiter kann die Symmetrie dann auch in der PY- bzw. HNC-Gleichung und der FBT verwendet werden, indem diese nur für die obere Dreiecksmatrix ausgewertet werden.

Konvergenzkriterium Die *relative Normänderung* zweier aufeinander folgender Iterationsschritte

$$||\mathbf{f}^{(n)}|| := \frac{\sqrt{\sum_{\alpha,\beta}\sum_i |f^{(n)}_{\alpha\beta,i} - f^{(n-1)}_{\alpha\beta,i}|^2}}{\sqrt{\sum_{\alpha,\beta}\sum_i |f^{(n)}_{\alpha\beta,i}|^2}}$$

liefert uns ein gutes Konvergenzkriterium mit $||\mathbf{f}^{(n)}|| \in [0,1]$. Es gilt also approximativ $\mathbf{f}^{(n)} = \mathbf{f}^{(\mathrm{fix})}$ genau dann, wenn $||\mathbf{f}^{(n)}|| < \delta$, wobei als Schranke $\delta = 10^{-10}$ gewählt wurde. Außerdem ist die relative Normänderung ein Maß dafür, wie weit der aktuelle Iterationsschritt noch von der konvergenten Lösung entfernt ist.

Separation des langreichweitigen Anteils Eine Aufteilung der Korrelatoren in kurz- und langreichweitigen Anteil, wie in [Lad78] für ein zweidimensionales Elektronen-Gas, haben wir nicht vorgenommen. Numerische Tests mit kurzreichweitigeren Potentialen wie Lenard-Jones und attraktiv dipolaren harten Scheiben wiesen die gleiche Problematik wie die repulsiv dipolaren harten Scheiben auf. Es treten jeweils vergleichbare numerische Instabilitäten auf, die wir exemplarisch in Abschnitt 3.3 kurz beschreiben werden. Weiter sind die Korrelatoren alle auf $< 10^{-6}$ bei unserer Wahl von R abgefallen. Das ist ein weiterer Indikator dafür, dass die Problematik nicht in der Langreichweitigkeit der dipolaren Systeme liegt. Für monodisperse harte Scheiben lassen sich mit dem Picardverfahren bereits konvergente Lösungen finden, wenn R so gewählt wird, dass die resultierenden Korrelatoren nur auf $\sim 10^{-1}$ abgefallen sind.

3.3. Versagen des Picardverfahrens

In diesem Abschnitt wollen wir die Picardmethode zur Iteration des Gleichungssystems (2.16), (2.17) und (3.2) vorstellen. Da die Picardmethode selbst hinreichend bekannt ist, verweisen wir auf [WHP92, HM86] und legen nur den von uns verwendeten adaptiven Picardalgorithmus für binäre Systeme dar. Aussagen über das Konvergenzverhalten des Picardverfahrens finden sich in [GSL05, WHP92]. Der Wahl der Startwerte widmen wir ein eigenes Unterkapitel. Am Ende dieses Abschnittes beschreiben wir, für welche Systeme der vorliegende Algorithmus konvergiert und exemplarisch einmal, wie divergenten Lösungen aussehen können.

Picard-Algorithmus

Es ist klar, dass die Picard-Iteration genau dann (schnell) konvergiert, wenn $||\mathbf{f}^{(n)}|| \to 0$ für wachsendes n. Sämtliche Verbesserungen (inklusive Abschnitt 3.4 und Abschnitt 3.5) werden darauf abzielen, dafür zu sorgen, dass

$$\forall n: \quad ||\mathbf{f}^{(n)}|| \leq ||\mathbf{f}^{(n-1)}|| \tag{3.5}$$

Das wird für das in diesem Abschnitt behandelte reine Picardverfahren so aussehen, dass wir eine adaptive Iterationsschrittweite und eine adaptive Stützstellenwahl verwenden. Bevor wir den Algorithmus angeben, wollen wir diese beiden Begriffe erklären.

Adaptive Iterationsschrittweite Angenommen für ein n gilt nicht mehr (3.5). Dann heißt das mit anderen Worten, dass die Änderung der zu iterierenden Funktion durch Anwenden des Iterationsoperators zu groß geworden ist. Also schränken wir die Änderung durch einen Faktor α ein, indem wir die $(n-1)$-te mit der (n)-ten Lösung mischen:

$$\mathbf{f}^{(n)} = \alpha I\,[\mathbf{f}^{(n-1)}] + (1-\alpha)\mathbf{f}^{(n-1)} \tag{3.6}$$

Wählt man den Faktor α nun statisch, werden entweder (3.5) nicht erfüllende Schritte nicht stark genug gedämpft oder der Algorithmus wird unbrauchbar langsam. Ausweg liefert die adaptive Wahl von α: Läuft die Picardmethode erst einmal in die richtige Richtung, d.h. $||\mathbf{f}^{(n)}|| \leq ||\mathbf{f}^{(n-1)}|| \leq ||\mathbf{f}^{(n-2)}|| \leq$ ..., so kann α erhöht werden um den Faktor δ_α^+. Umgekehrt muss α mit δ_α^- verringert werden, falls (3.5) für ein n einmal nicht gelten sollte. In diesem Falle bietet es sich an, den betreffenden Schritt zurückzunehmen (*Backtrace-Verfahren*):

$$||\mathbf{f}^{(n)}|| \geq ||\mathbf{f}^{(n-1)}|| \quad \rightarrow \quad \mathbf{f}^{(n)} = \mathbf{f}^{(n-1)}$$

Anfangs wurde $\alpha := 10^{-15}$ gewählt, um zu gewährleisten, dass die ersten Iterationsschritte in die richtige Richtung starten. Die Kontrollparameter wurden als $\delta_\alpha^+ := 1.01$ und $\delta_\alpha^- := 0.98$ gewählt[4]. Weiter geben wir mir $\delta_\alpha \leq 1$ eine obere Schranke für das Wachstum des Parameters α an. Wir geben nun die Funktion *step* an, die eine gemischte Lösung und einen neuen Mischparameter liefert.

Algorithm 1 step

Require: $\alpha, \tilde{\mathbf{h}}^{(n)}, \mathbf{h}^{(n-1)}$
Require: $||\mathbf{h}^{(n-1)}||, ||\mathbf{h}^{(n-2)}||, ||\mathbf{h}^{(n-3)}||$

if $||\tilde{\mathbf{h}}^{(n)}|| \leq ||\mathbf{h}^{(n-1)}|| \leq ||\mathbf{h}^{(n-2)}|| \leq ||\mathbf{h}^{(n-3)}||$ **then**
  $\alpha = \alpha\delta_\alpha^+$
else
  $\alpha = \alpha\delta_\alpha^-$
end if

if $||\tilde{\mathbf{h}}^{(n)}|| \leq ||\mathbf{h}^{(n-1)}||$ **then**
  $\mathbf{h}^{(n)} = \alpha\tilde{\mathbf{h}}^{(n)} + (1-\alpha)\mathbf{h}^{(n-1)}$
else
  $\mathbf{h}^{(n)} = \mathbf{h}^{(n-1)}$
end if

return $\mathbf{h}^{(n)}, \alpha$

Adaptive Stützstellenwahl[5] Die Idee ist hier, die Anzahl der Stützstellen sukzessive zu erhöhen. Das heißt wir beginnen unser Iterationsverfahren mit $M^{(1)}$ Stützstellen und erhöhen die Zahl der Stützstellen auf $M^{(2)}$, sobald ein FP gefunden wurde. Mit Hilfe von Abschnitt A.1 oder Abschnitt A.2 müssen die an $M^{(1)}$ Stellen bekannten Funktionen dann für $M^{(2)}$ Stützstellen interpoliert werden. So wird weiter verfahren, bis schließlich die gewünschte Anzahl von M Stützstellen erreicht ist. Es gibt hauptsächlich zwei Gründe, warum die adaptive Stützstellenwahl das Picardverfahren schneller

[4] $\delta_\alpha^+ = \delta_\alpha^-$ wurde vermieden, um lokale Bistabilitäten zu vermeiden.
[5] Diese Methode wird in der Literatur oft als *Multigridverfahren* bezeichnet.

und stabiler macht. Der erste Grund ist die starke Abhängigkeit der Picardmethode vom Startwert. Wählt man zunächst sehr wenige Stützpunkte, bekommt man mit wenig Rechenaufwand durch Interpolation einen guten Startwert für den nächsten Durchlauf. Zweiter Grund ist die in dieser Methode automatisch enthaltene Glättung der zu iterierenden Funktionen. Bei einer kleinen Anzahl Stützstellen $M^{(g)}$ sind Divergenzen provozierende schnelle Oszillationen per se nicht möglich und wird bei Konvergenz die Anzahl auf $M^{(g+1)}$ erhöht, ergibt sich durch die stetige Interpolation ein glatter Startwert. Wir haben die Anzahl der Stützstellen von $M^{(0)} = 256$ bis zu $M = 3072$ laufen lassen[6] und dadurch die binären harten Scheiben lösen können. Im Algorithmus werden wir die Anzahl der Stützstellen im jeweiligen Durchlauf $M^{(g)}$ nennen. Und die Interpolation durch *interpolate* beschreiben. Der Einfluss von M auf die konvergierten Lösungen äußert sich vor allem in der Präzision der Peakdarstellung. Weil aber gerade das erste Maximum von $S(q)$ stark das Verhalten des Nichtergodizitätsparameters $f(q)$ bestimmt (siehe Abschnitt 6.3), müssen die Peaks durch möglichst viele Stützstellen dargestellt werden.

Der Picard-Algorithmus Nun können wir den Picard-Algorithmus angeben. Wie in (2.17) zu erkennen ist, nehmen wir implizit zwei Startwerte an. Wir wählen zunächst $\mathbf{c}^{(0)}(r) = 0$ und geben $\mathbf{h}^{(0)}$ durch die Mayerfunktion $e^{-\beta \mathbf{V}(r)} - 1$ vor. An dieser Stelle sei kurz bemerkt, dass $\mathbf{V}(r)$ nur einmal berechnet wird um Rechenzeit zu sparen.

Algorithm 2 Picard Iteration

Require: discretization $r_i, q_m, M^{(1)}$
Require: FBT, symmetrize, step, interpolate

1: set startvalues $\mathbf{c}^{(0)}, \mathbf{h}^{(0)}$
2: **repeat**
3: **repeat**
4: $\mathbf{c}^{(n-1)}, \mathbf{h}^{(n-1)} \xrightarrow{\text{PY/HNC}} \tilde{\mathbf{h}}^{(n)}$
5: $\tilde{\mathbf{h}}^{(n)} \xrightarrow{\text{FBT}} \mathbf{H}^{(n)}$
6: $\mathbf{H}^{(n)} \xrightarrow{\text{OZ}} \mathbf{C}^{(n)}$
7: symmetrize $\mathbf{C}^{(n)}$
8: $\mathbf{C}^{(n)} \xrightarrow{\text{FBT}} \mathbf{c}^{(n)}$
9: $\alpha, \tilde{\mathbf{h}}^{(n)}, \mathbf{h}^{(n-1)}, ||\mathbf{h}^{(n-1)}||, ||\mathbf{h}^{(n-2)}||, ||\mathbf{h}^{(n-3)}|| \xrightarrow{\text{step}} \mathbf{h}^{(n)}, \alpha$
10: **until** $||\mathbf{h}^{(n)}|| < \delta \quad \wedge \quad \alpha > \delta_\alpha$
11: discretization $r_i, q_m, M^{(g)}$
12: $\mathbf{c}^{(0)}, \mathbf{h}^{(0)} \xleftarrow{\text{interpolate}} \mathbf{c}^{(\text{fix})}, \mathbf{h}^{\text{fix})}$
13: **until** $M^{(\text{g})} = M$

Die Wahl des richtigen Startwertes

Wie bereits erwähnt konvergiert der Algorithmus nur dann, wenn der Startwert gut gewählt wurde. Betrachten wir nun monodisperse Systeme harter Scheiben, so ist der Algorithmus in der von

[6] Für $M^{(0)} < 64$ ergaben sich keine sinnvollen Lösungen und für $64 \leq M^{(0)} \leq 256$ konnte kein zusätzlicher Vorteil mehr aus der Methode gezogen werden.

uns verwendeten Form bei niedrigen Packungsdichten $\eta \leq 0.5$ sehr stabil gegen Variation des Startwertes. Packungsdichten von $\eta > 0.6$ können hingegen nur mit hohem Rechenaufwand und gutem Startwert erreicht werden. Die Stabilität bei geringer Dichte kann man sich zu Nutze machen, indem die Lösung einer Packungsdichte $\eta_1 < \eta_2$ als Startwert für letztere verwendet wird. Mit Hilfe dieser Methodik konnten die Systeme bishin zu einer Packungsdichte von $\eta \leq 0.63$ gelöst werden. Ganz analog kann bei den dipolaren Systemen verfahren werden: Ausgehend vom Harte-Scheiben-Limes hoher Temperaturen, kann die Lösung eines Systems mit der Temperatur $T_1 > T_2$ als Startwert für das gleiche System mit der Temperatur T_2 verwendet werden. Auch hier zeigen sich schnell die Grenzen dieses Vorgehens: Eine minimale mittlere magnetische Wechselwirkung (4.14) $\Gamma = 10^{-2}$ konnte nicht überschritten werden. Diese Vorgehensweise ist auch für die Iteration der Nichtergodizitätsparameter in Abschnitt 6.3 wichtig, denn sie verringert die Anzahl der nötigen Iterationsschritte bishin zur konvergenten Lösung. Weitere mögliche Startwerte für das binäre dipolare System sind durch das Experiment [HKM05] gegeben. Glättet man diese experimentellen Daten mittels Abschnitt A.1 und verwendet sie als Startwert, ergeben sich allerdings trotzdem noch keine konvergenten Lösungen. Auch mit bikubischen Splines (siehe Abschnitt A.2) extrapolierte Startwerte konvergenter Lösungen bei geringerer Wechselwirkung führen nicht zum Erfolg.

Fazit

Divergente Lösungen Wir bezeichnen eine Lösung als divergent, wenn $||f^{(n)}|| \to \infty$ gilt. In unserem Falle haben sich die Divergenzen so geäußert, dass $||c^{(n)}|| \sim n$ linear divergiert und $h^{(n)}$ immer schneller oszilliert. In Abb. 3.1 geben wir ein Beispiel für divergente Korrelatoren. Die Oszillationen in h beginnen zunächst im Bereich $r\sigma^{-1} < 1$ und pflanzen sich dann im Laufe der Picard-Schritte zu größeren r fort. Daher ist es eine weitere Verfeinerung, die bekannten Limites der Korrelatoren auszunutzen und $h(r) = -1$ für alle $r\sigma^{-1} < 1$ zu setzen. Doch weder das, noch der Übergang zum glatten Korrelator γ führten zur Konvergenz. Präzise ließ sich allerdings nicht festmachen, woran die Divergenzen lagen. Zunächst könnte man die zu hoher Dichte hin steigenden Peaks der Korrelatoren verantwortlich machen; Gegenargument ist allerdings, dass der Sprung in der Mayerfunktion harter Scheiben bei $r\sigma^{-1} = 1$, der direkt eine Unstetigkeit in c verursacht, keine Divergenzen erzeugt. Wie bereits in der Vorbemerkung beschrieben, liegt die Ursache auch nicht in der größeren Reichweite der Wechselwirkung in zwei Dimensionen. Obwohl sich die einfache Picardmethode ohne adaptive Stützpunktwahl in drei Dimensionen sowohl für dipolare harte Kugeln [SS98] als auch für wesentlich komplexere Systeme [RS05, SS98, TTL01] bewährt hat, kann sie für zweidimensionale Systeme nicht mehr verwendet werden. Der Grund dafür liegt letztlich in der FBT, denn für dreidimensionale Systeme lässt sich der Winkelanteil herausintegrieren.

Konvergente Lösungen Wie bereits zuvor erwähnt, konnten mittels dieses Algorithmus die monodispersen harten Scheiben bishin zu einer Packungsdichte von $\eta \leq 0.63$ gelöst werden und die binären harten Scheiben bishin zu $\eta \leq 0.65$. Warum der Algorithmus für die binären Systeme bei höheren Dichten noch konvergiert, ist nicht ersichtlich. Dipolare Systeme konnten nur im monodispersen Fall gelöst werden. Und auch die monodispersen dipolaren Systeme konnten lediglich bis zu einer mittleren magnetischen Energie (4.7) $\Gamma < 10^{-2}$ gelöst werden. Der monodisperse Grenzfall des binären Systems[7] war ebenso numerisch instabil. Prinzipiell erwies sich die HNC-Approximation in

[7] Das heißt die Teilcheneigenschaften werden gleich gewählt.

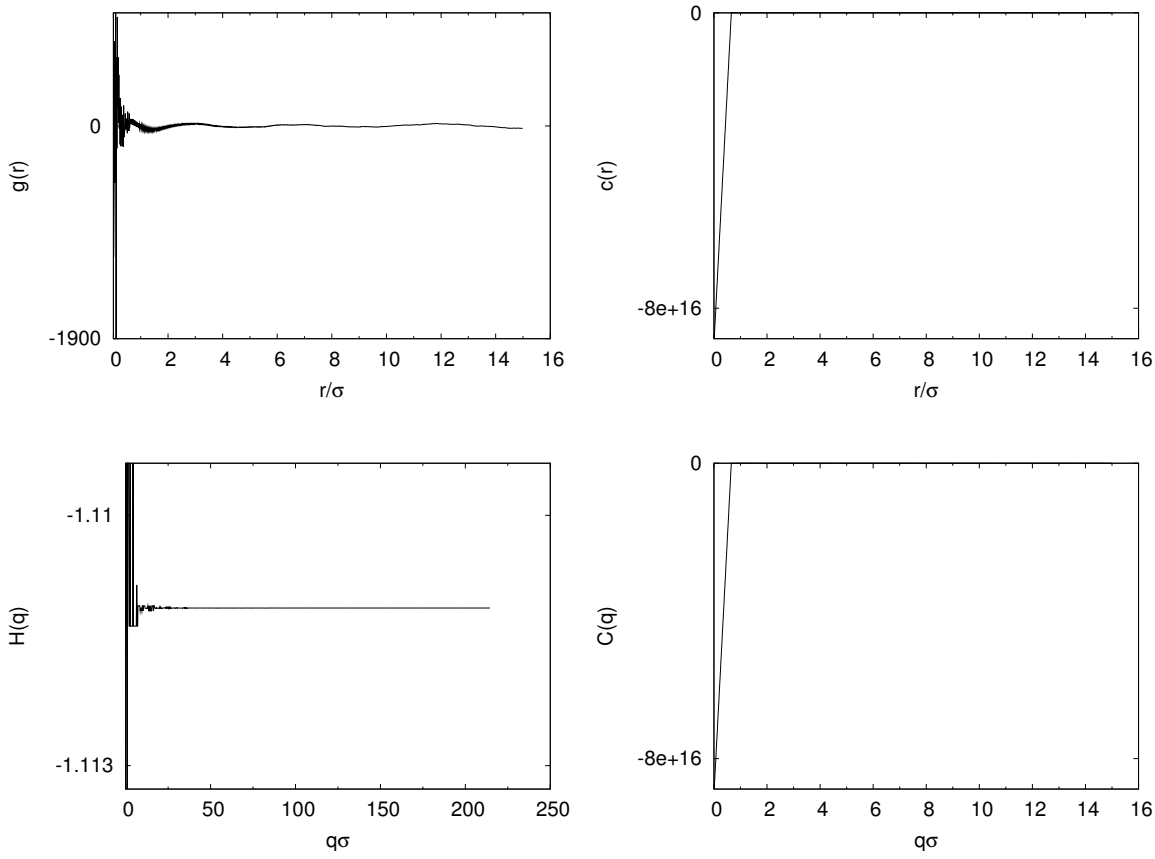

Abbildung 3.1.: Divergente Korrelatoren monodisperser harter Scheiben bei $\eta = 0.71$

Kombination mit dem Picardverfahren als numerisch instabiler als die PY-Approximation. Insbesondere bei starker Wechselwirkung (hohe Packungsdichte und/oder niedrige Temperatur) zeigte sich die HNC- der PY-Approximation unterlegen.

3.4. Gillans Ausweg

Nun werden wir den Algorithmus von Gillan [Gil79] vorstellen, der durch eine Kombination von Picard-Iteration und Newton-Rhapson-Verfahren (NR) Divergenzen vermeiden soll. Dazu wird zunächst zu der glatten Funktion γ übergegangen und das GLS lautet nun (2.18), (2.19) und (3.2). Grundidee ist dabei das NR zu verwenden, um die Picardmethode im Sinne von (3.5) zu stabilisieren. Bevor wir die Problematik bei der Verallgemeinerung auf zwei Teilchensorten darlegen, stellen wir den monodispersen Algorithmus vor, weil sein Verständnis notwendig für den von uns entwickelten Algorithmus in Abschnitt 3.5 ist. Abschließen werden wir den Abschnitt mit einem Vergleich zur Picardmethode.

Gillans Algorithmus

Für die Beschreibung der Problematik des Picard-Algorithmus in drei Dimensionen verweisen wir auf [Gil79] und halten fest, dass insbesondere für zweidimensionale Systeme Divergenzen bei hohen Dichten und niedrigen Temperaturen auftreten. In [Gil79] ist der Algorithmus für dreidimensionale Probleme formuliert; die Übertragung auf zwei Dimensionen ist einfach und äußert sich im Wesentlichen nur in der FBT und der Jacobimatrix des NR. Wir werden wieder zunächst einige Vorbemerkungen zum Algorithmus machen und ihn dann im Pseudo-Code angeben.

Aufteilung in Grob- und Feinanteil Wie eingangs bereits erwähnt, ist der zentrale neue Punkt die Aufteilung in einen Grob- und Feinanteil. Dabei soll die Aufteilung derart sein, dass gleichzeitig die Vorteile der Picard- und der Newton-Methode ausgenutzt werden können. Das heißt, dass der grobe Verlauf möglichst gut mit dem Newton-Verfahren behandelt wird, das für kleine Anzahlen von Stützstellen sehr schnell und insbesondere stabil ist. Die vielen feinen Änderungen sollen mit dem Picardverfahren erzeugt werden, das für diese gut konvergiert und einen geringeren Rechenaufwand benötigt. Gesucht sind nun also Funktionen, die eine solche Aufteilung ermöglichen; naheliegend ist eine stückchenweise Mittelwertbildung, um den groben Verlauf zu erfassen. Bezeichne wie gewohnt i den diskretisierten Ort und weiter $0 \leq \alpha < \nu$ die α-te Basisfunktion, so lautet eine mögliche Basis[8]:

$$\alpha = 0 \qquad P_i^{\alpha=0} = \begin{cases} \frac{i_1 - i}{i_1} & 0 \leq i \leq i_1 \\ 0 & i_1 < i < M \end{cases} \tag{3.7}$$

$$\alpha = 1 \qquad P_i^{\alpha=1} = \begin{cases} \frac{i}{i_1} & 0 \leq i \leq i_1 \\ \frac{i_2 - i}{i_2 - i_1} & i_i < i \leq i_2 \\ 0 & i_2 < i < M \end{cases}$$

[8] Die Formel in [Gil79] ist falsch.

$$\alpha \geq 2 \qquad P_i^\alpha = \begin{cases} 0 & 0 \leq i \leq i_{\alpha-2} \\ \frac{i-i_{\alpha-2}}{i_{\alpha-1}-i_{\alpha-2}} & i_{\alpha-2} < i \leq i_{\alpha-1} \\ \frac{i_{\alpha-i}}{i_\alpha - i_{\alpha-1}} & i_{\alpha-1} < i \leq i_\alpha \\ 0 & i_\alpha < i < M \end{cases}$$

Diese Basen[9] erfüllen $0 \leq P_i^\alpha \leq 1$ für alle α, i und können allein aus einer gegebenen Menge von $i_0 \ldots i_{\nu-1}$ konstruiert werden. In Abb. 3.2 sind vier solcher Basen dargestellt.

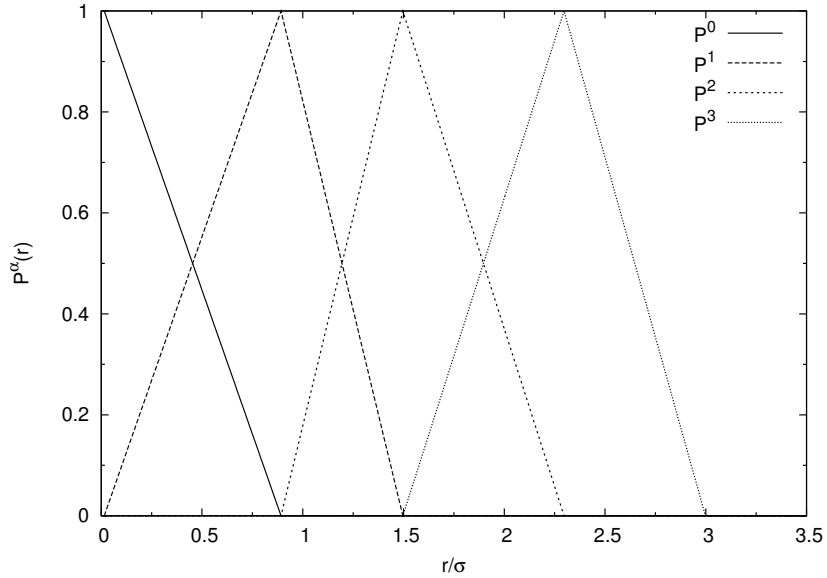

Abbildung 3.2.: Basisfunktionen $P^\alpha(r)$; der untere Graph ist eine Vergrößerung des oberen im Intervall $-1 \leq \gamma(r) \leq 2$

Die dazu konjugierte Basis bezeichnen wir mit Q^α und definieren sie über die Orthogonalität $\sum_i Q_i^\alpha P_i^\beta = \delta_{\alpha\beta}$. Damit lässt sich die konjugierte Basis bestimmen:

$$Q_i^\alpha = \sum_\beta \left(\left[\sum_j P_j^\alpha P_j^\beta \right]^{-1} \right)_{\alpha\beta} P_i^\beta \tag{3.8}$$

Im Algorithmus verwenden wir die Funktion *construct-basis*, die aus einer gegebenen Menge $i_0 \ldots i_{\nu-1}$ und diskretisierten Orten $r_0 \ldots r_{M-1}$ die Basen P^α und Q^α konstruiert. Die dabei auftretende Matrixinversion wurde analytisch durchgeführt. Mit Hilfe dieser Basen kann nun der Korrelator γ im Ortsraum in seinen Grobanteil a_α und seinen Feinanteil $\Delta\gamma_i$ zerlegt werden mit:

$$\begin{aligned} a_\alpha &= \sum_{i=0}^{M-1} \gamma_i Q_i^\alpha \\ \Delta\gamma_i &= \gamma_i - \sum_{\alpha=0}^{\nu-1} a_\alpha P_i^\alpha \end{aligned} \tag{3.9}$$

[9] Mit Basen bezeichnen wir im folgenden die Basisfunktionen P_i^α und Q_i^α.

Damit ergibt sich automatisch:

$$\gamma_i = \Delta\gamma_i + \sum_{\alpha=0}^{\nu-1} a_\alpha P_i^\alpha \tag{3.10}$$

Nun sollen diese beiden Anteile getrennt behandelt werden, indem die Feinanteilkoeffizienten $\Delta\gamma_i$ mit Picard und die Grobanteilkoeffizienten a_α mit dem NR behandelt werden. Um unnötige Indices zu vermeiden, zählen wir die Iterationsschritte nicht mehr durch, sondern unterscheiden zwei aufeinander folgende Schritte mit f und $f^{'}$. Die Konvergenzbedingungen lauten nun also

$$||a, a^{'}|| \quad := \quad \frac{\sqrt{\sum_{\alpha=0}^{\nu-1} |a_\alpha - a'_\alpha|^2}}{\sqrt{\sum_{\alpha=0}^{\nu-1} |a_\alpha|^2}} < \delta_{\text{coarse}} \tag{3.11}$$

$$||\Delta\gamma, \Delta\gamma^{'}|| \quad := \quad \frac{\sqrt{\sum_{i=0}^{M-1} |\Delta\gamma_i - \Delta\gamma'_i|^2}}{\sqrt{\sum_{i=0}^{M-1} |\Delta\gamma_i|^2}} < \delta_{\text{fine}} \tag{3.12}$$

Wie zuvor beim Picard-Algorithmus wählen wir $\delta_{\text{fine}} = 10^{-10}$ und weiter die Grobanteilschranke kleiner mit $\delta_{\text{coarse}} = 10^{-15}$, weil diese für die Konvergenz des Feinanteils verantwortlich ist.

Elementarschritt und Iteration Unter einem elementaren Schritt verstehen wir einen Picarddurchlauf durch die Gleichungen (2.18), (2.19) und (3.2):

$$\gamma \to c \to C \to \Gamma \to \gamma^{'} \quad \Leftrightarrow: \quad \gamma \longrightarrow \gamma^{'} \tag{3.13}$$

Dieser elementare Schritt liefert nach Zerlegung einen neuen Feinanteil- $\Delta\gamma_i^{'}$ und Grobanteil-Koeffizienten $a_\alpha^{'}$. An dieser Stelle sei bemerkt, dass der elementare Schritt selbst noch keine Iteration darstellt. Die aus ihm gelieferten Koeffizienten werden nun getrennt behandelt. Für die Koeffizienten $a_\alpha^{'}$ liefert das NR neue Grobanteile mittels:

$$a_\alpha = a_\alpha - \sum_{\beta=0}^{\nu-1} (J^{-1})_{\alpha\beta} \left[a_\alpha - a_\alpha^{'} \right] \tag{3.14}$$

mit der Jacobimatrix

$$J_{\alpha\beta} := \frac{\partial}{\partial a_\beta} \left[a_\alpha - a_\alpha^{'} \right]$$

Sobald ein neuer Satz a_α so errechnet wurde, kann mit Hilfe von (3.10) die Komposition $a_\alpha, \Delta\gamma_i \to \gamma_i$ gebildet werden, mit welcher wieder ein Elementarschritt durchgeführt wird. Das wird solange wiederholt, bis das Konvergenzkriterium für den Grobanteil (3.11) erfüllt ist. Erst wenn das der Fall ist, wird ein Picardschritt im Feinanteil gemacht, indem $\Delta\gamma_i = \Delta\gamma_i^{'}$ gesetzt wird.

Jacobimatrix für zwei Dimensionen analytisch behandelt Für die Newtonschritte ist es notwendig die Jacobimatrix analytisch zu kennen. Mit Hilfe des GLS werden wir einen solchen Ausdruck nun herleiten. Zunächst gilt

$$J_{\alpha\beta} = \delta_{\alpha\beta} - \frac{\partial a_\alpha^{'}}{\partial a_\beta}$$

und mit Hilfe der Basen folgt weiter

$$\frac{\partial a_\alpha^{'}}{\partial a_\alpha} = \sum_{i,j=0}^{M-1} Q_i^\alpha \frac{\partial \gamma_i^{'}}{\partial \gamma_j} P_j^\beta$$

Die Ableitungen können aber mit Hilfe der Kettenregel unter Beachtung der Lokalität der OZ- und PY-/HNC-Gleichungen umgeschrieben werden zu

$$\frac{\partial \gamma_i^{'}}{\partial \gamma_j} = \sum_{m=0}^{M-1} \frac{\partial \gamma_i^{'}}{\partial \Gamma_m} \frac{\partial \Gamma_m}{\partial C_m} \frac{\partial C_m}{\partial c_j} \frac{\partial c_j}{\partial \gamma_j}$$

Nun kann das GLS ausgenutzt werden und es folgt schließlich mit PY:

$$\begin{aligned} J_{\alpha\beta} &= \delta_{\alpha\beta} - \left(\frac{2}{QR}\right)^2 \sum_{m=0}^{M-1} \left(-1 + \frac{1}{(-1+\rho C_m)^2}\right) \times \\ &\quad \times \sum_{i,j=0}^{M-1} Q_j^\alpha P_i^\beta \frac{J_0(q_m r_j) J_0(q_m r_i)}{J_1^2(q_m R) J_1^2(Q r_i)} \left(e^{-\beta V_i} - 1\right) \end{aligned} \tag{3.15}$$

Die Konstante $\mathrm{const}(m) = \left(\frac{2}{QR}\right)^2 \sum_{i,j=0}^{M-1} \frac{J_0(q_m r_j) J_0(q_m r_i)}{J_1^2(q_m R) J_1^2(Q r_i)} \left(e^{-\beta V_i} - 1\right)$ berechnen wir eingangs nur einmal, um Rechenaufwand einzusparen. Das Ergebnis für die HNC-Approximation ist analog.

Begrenzung der Schrittweiten Bevor wir den Algorithmus angeben, bemerken wir noch, dass analog zum Picard-Algorithmus die Schrittweiten in Fein- und Grobanteil beschränkt werden müssen, falls die jeweilige Norm nicht kontinuierlich gegen Null gehen sollte. Im Gegensatz zur reinen Picard-Methode sind hier diese Schritte in die falsche Richtung aufgrund der Stabilität der Newton-Methode die Seltenheit und es genügt bereits nur diesen jeweiligen Schritt zu beschränken mittels:

$$f^{'} = ||f, f^{'}||f^{'} + \left[1 - ||f, f^{'}||\right] f$$

Diese Beschränkungsvorschrift wurde sowohl für den Grob-, als auch den Feinanteil verwendet.

Die Wahl der richtigen Basisfunktionen Zentrale Idee dieses Algorithmus ist eine sinnvolle Aufteilung in Grob- und Feinanteil, so dass die Vorteile der Picard- mit denen der Newton-Methode kombiniert werden können. Es ist sofort klar, dass die Basen nicht äquidistant gewählt werden können, sondern der zu iterierenden Funktion angepasst werden müssen. Denn einerseits soll die Anzahl der Basen nicht zu klein sein, um die Stabilität des Newton-Verfahrens möglichst optimal zu nutzen und andererseits darf die Zahl der Basen nicht zu groß sein, da der Rechenaufwand des Newton-Verfahrens

Algorithm 3 Gillan Algorithmus

Require: discretization r_i, q_m, M

Require: construct-basis, symmetrize

set startvalue γ_i

first decomposition: $\gamma_i \longrightarrow a_\alpha, \Delta\gamma_i$

repeat

  repeat

    $a_\alpha, \Delta\gamma_i \longrightarrow \gamma_i$

    $\gamma_i \longrightarrow \gamma_i^{'}, C_m$

    symmetrize $C_m, \gamma_i^{'}$

    $\gamma_i^{'} \longrightarrow a_\alpha^{'}, \Delta\gamma_i^{'}$

    $C_m, const(m) \longrightarrow J_{\alpha\beta}^{-1}$

    New coarse estimate: $a_\alpha = a_\alpha - \sum_\beta (J^{-1})_{\alpha\beta}[a_\alpha - a_\alpha^{'}]$

    if $\neg||a, a^{'}|| \rightarrow 0$ **then**

      $a_\alpha = ||a, a^{'}||a_\alpha + [1 - ||a, a^{'}||]a_\alpha^{'}$

    end if

  until $||a_\alpha, a_\alpha^{'}|| < \delta_{\text{coarse}}$

  New fine estimate: $\Delta\gamma_i = \Delta\gamma_i^{'}$

  if $\neg||\Delta\gamma, \Delta\gamma^{'}|| \rightarrow 0$ **then**

    $\Delta\gamma_i = ||\Delta\gamma, \Delta\gamma^{'}||\Delta\Gamma_i + [1 - ||\Delta\gamma, \Delta\gamma^{'}||]\Delta\Gamma_i^{'}$

  end if

until $||\Delta\gamma, \Delta\gamma_i^{'}|| < \delta_{\text{fine}}$

sonst zu groß wird. Je höher die Dichte bzw. je niedriger die Temperatur ist, desto kritischer wird die Wahl. Ist die Menge der Basen nicht gut gewählt, ergibt sich eine schlechte Aufteilung zwischen den beiden Iterationsalgorithmen, die sich in divergenten Lösungen äußert. Um einen guten Satz Basisfunktionen zu finden, bedienen wir uns der Funktion γ_i bei geringer Dichte.

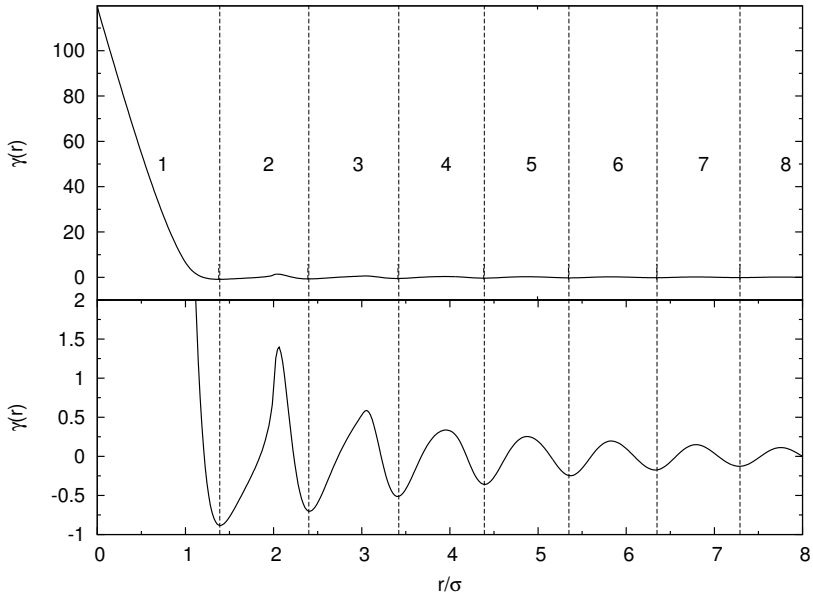

Abbildung 3.3.: $\gamma(r)$ für monodisperse harte Scheiben bei $\eta = 0.75$.

γ_i hat ein Maximum bei $r_{i=0}$ und fällt dann schnell ab. Selbst für dipolare Systeme fällt $\gamma(r)$ noch mit r^{-3} für große r. Wir teilen auf in Intervalle zwischen ihren lokalen Minima und wählen dann je nach ihrer Oszillationsstärke im jeweiligen Intervall die Basisfunktionen aus. Je stärker sich γ in einem Intervall ändert, desto größer muss die Zahl der Basen sein, um zu gewährleisten, dass möglichst viel des groben Kurvenverlaufes mit dem Newtonverfahren iteriert wird. Wie in Abb. 3.3 zu erkennen, ist das erste Maximum in γ sehr hoch; wählt man in diesem ersten Intervall jedoch die Zahl der Basen sehr hoch, können trotzdem Divergenzen auftreten. Das passiert genau dann, wenn pro Basis P^α weniger als zwei Stützstellen vorhanden sind. Um einen Satz von Basen zu testen kann man wie folgt verfahren: Die betrachtete Funktion γ wird zerlegt und allein aus ihrem Grobanteil wieder zusammengesetzt: $\gamma_i = \sum_{\alpha=0}^{\nu-1} P_i^\alpha a_\alpha$. Die resultierende Abweichung beider Funktionen dient als Maß für die Güte der Basen. Die Anzahl der verwendeten Basisfunktionen ist nach oben dadurch beschränkt, dass die Jacobimatrix jedesmal berechnet und invertiert werden muss. Für rein harte Scheiben genügen 20 Basisfunktionen, um konvergente Lösungen zu erhalten, für dipolare harte Scheiben sind es bereits 45.

Die Wahl des richtigen Startwertes Im Gegensatz zum Picardverfahren im letzten Abschnitt liefert Gillans Algorithmus unabhängig vom Startwert eine konvergente Lösung. Weiter konnte keine wesentliche Rechenzeitersparnis durch geschickte Wahl der Startwerte erzielt werden. Bereits nach den ersten Picardschritten haben die Korrelatoren ihre grobe Gestalt angenommen.

Verallgemeinerung auf zwei Teilchensorten

Die Verallgemeinerung des Algorithmus auf binäre Systeme ist in unserem Falle besonders einfach, weil die zwei Teilchensorten im Experiment auf etwa gleich großen Längenskalen wechselwirken [HKM05]. Das ermöglicht uns, den gleichen Satz Basen für alle drei Matrixkomponenten zu verwenden. Die Basisentwicklung lässt sich also direkt komponentenweise schreiben und damit ist der binäre Algorithmus vollkommen analog zum monodispersen. Problematisch ist jedoch die Kopplung der drei Matrixkomponenten der Korrelatoren in der Jacobimatrix. Wir weichen von der Konvention für die Matrixschreibweise in diesem Unterabschnitt ab und benennen die Teilchenindices mit $f^{\mu\nu}$. In jedem Newton-Schritt wird der neue Grobanteil berechnet vermöge:

$$a_\alpha^{\mu\nu} = a_\alpha^{\mu\nu} - \sum_{\beta=0}^{\nu-1}\sum_{\rho,\sigma=0}^{1}\left\{\left(\frac{\partial}{\partial a_\beta^{\rho\sigma}}\left[a_\alpha^{\mu\nu} - a_\alpha^{'\mu\nu}\right]\right)^{-1}\right\}_{\alpha\beta}\left(a_\beta^{\rho\sigma} - a_\beta^{'\rho\sigma}\right)$$

Die dabei zu invertierende Jacobimatrix muss wieder analytisch berechnet werden mit:

$$J_{\alpha\beta}^{\mu\nu,\rho\sigma} = \delta_{\alpha\beta}\delta_{\mu\rho}\delta_{\nu\sigma} - \sum_{i,j=0}^{M-1}\sum_{\zeta,\xi=0}^{1} Q_i^\alpha \frac{\partial\gamma_i^{'\rho\sigma}}{\partial\gamma_j^{\zeta\xi}} P_j^\beta$$

Analog zum monodispersen Fall wird die Ableitung mittels Kettenregel und dem GLS berechnet und es ergibt sich insgesamt:

$$\begin{aligned} J_{\alpha\beta}^{\mu\nu,\rho\sigma} &= \delta_{\alpha\beta}\delta_{\mu\rho}\delta_{\nu\sigma} - \left(\frac{2}{QR}\right)^2 \sum_{i,j=0}^{M-1} Q_i^\alpha P_j^\beta \left(e^{-\beta V_i^{\rho\sigma}} - 1\right) \times \\ &\quad \times \sum_{m=0}^{M-1} \frac{J_0(q_m r_i) J_0(q_m r_j)}{J_1^2(q_m R) J_1^2(Q r_i)} \frac{\partial\Gamma_m^{\mu\nu}}{\partial C_m^{\rho\sigma}} \end{aligned}$$

Mit Hilfe der OZ-Gleichung kann die verbleibende Ableitung $\frac{\partial\Gamma_m^{\mu\nu}}{\partial C_m^{\rho\sigma}}$ berechnet werden. Nutzt man die Symmetrie aus, so verbleiben 9 unabhängige Elemente in diesem Tensor zu berechnen. Insgesamt ergibt sich damit aber ein extrem hoher Rechenaufwand, der sowohl aus der Anzahl der Tensorelemente, als auch aus der Multiplikation mit den (vorher abgespeicherten) Besselfunktionen folgt,[10] denn jeder Picardschritt benötigt viele Newtonschritte. Der Gillan Algorithmus ist also nicht für binäre Systeme verwendbar, weil die analytische Berechnung der Jacobimatrix zu rechenaufwendig wird.

Fazit

Die Methode von Gillan bietet im Vergleich zur Picardmethode aus dem letzten Abschnitt einen deutlichen Geschwindigkeitsvorteil. Die Stabilität dieser Methode erlaubt es, den Glasübergang an monodispersen harten Scheiben zu untersuchen. Doch noch immer steht die numerische Lösung der dipolaren

[10] Um eine grobe Idee zu geben: Braucht der Algorithmus für ein monodisperses System Bruchteile einer Sekunde für einen Newton-Schritt, so sind es für das binäre System mit zwei identischen Teilchensorten bei den gleichen Parametern knapp 30 Sekunden.

Systeme aus. Weiter ist es wegen des zu hohen Rechenaufwands nicht möglich, die binären Systeme mit dieser Methode zu iterieren.

Vergleich zum Multigridverfahren Dem Algorithmus von Gillan liegt zwar prinzipiell die gleiche Überlegung wie dem Multigridverfahren zu Grunde: Die Divergenzen des reinen Picardverfahrens können vermieden werden, wenn der grobe Verlauf der Kurven stabil bleibt. Für monodisperse Systeme zeigt sich der Algorithmus von Gillan trotzdem bei starken Wechselwirkungen überlegen. Die Hauptursache dafür mag darin liegen, dass wir uns für das Multigridverfahren nur auf das Picardverfahren verlassen haben. Die Anzahl der nötigen Iterationsschritte verringerte sich außerdem drastisch: Brauchte Picard alleine noch in der Größenordnung von 100.000 Schritte bis zur Konvergenz, so gelang es dem Gillan Algorithmus für das gleiche System bereits in wenigen Sekunden mit einigen 100 Schritten zur Konvergenz zu gelangen.

Konvergente Lösungen Mittels dieses Algorithmus konnten für die monodispersen harten Scheiben Lösungen bei genügend hoher Dichte gefunden werden, um einen Glasübergang zu finden (siehe Abschnitt 7.1). Die dipolaren Scheiben können zwar bereits bishin zu wesentlich stärkeren mittleren magnetischen Wechselwirkungen $\Gamma \leq 2$ gelöst werden, was jedoch noch nicht ausreichend für die Untersuchung des Glasüberganges ist (siehe Abschnitt 7.2).

3.5. Erweiterung mit Levenberg-Marquardt

In diesem letzten Abschnitt stellen wir den schließlich für alle Systeme von uns verwendeten Algorithmus vor. Er ergänzt Gillans Algorithmus um die binären Systeme und beschleunigt die Konvergenz der monodispersen Systeme abermals drastisch. Wie zuvor legen wir zunächst die Details des Algorithmus dar und geben ihn dann im Pseudo-Code an. Genau wie die anderen beiden Algorithmen haben wir ihn sowohl monodispers, als auch binär implementiert. Jedoch werden wir hier nur den binären Fall darstellen. Im Anhang (Kapitel B) befinden sich die Listings des monodispersen und binären LM-Algorithmus.

Levenberg-Marquardt-Algorithmus

Ausgehend von Gillans Algorithmus ist eine Möglichkeit gesucht, die binären Systeme zu lösen. Die Problematik war der ungeheure Rechenaufwand für die Iteration des Grobanteils gewesen. Es liegt also nahe, eine andere Methode zur Iteration des Grobanteils zu verwenden. Eine die Anforderungen erfüllende Methode ist die Levenberg-Marquardt-Methode [Wut05], die gleich mehrere Vorteile bietet. Zuerst einmal ist kein analytischer Ausdruck für die Jacobimatrix notwendig, und weiter ist sie wesentlich stabiler und schneller als die Newton-Methode [WHP92, GSL05]. Sie ist eine Kombination aus verschiedenen Algorithmen, zwischen denen stetig umgeschaltet wird, je nachdem wie groß die relative Normänderung ist. Die hohe Konvergenzgeschwindigkeit des Marquardt-Algorithmus (im folgenden kurz LM) resultiert z.B. aus einem Steepest-Descent-Algorithmus, der für große Normänderungen angewandt wird. Für eine tiefergehende Darstellung verweisen wir auf [WHP92]. Wir werden im Algorithmus den Marquardt-Teil durch eine Funktion *minimize* beschreiben. Diese Funktion bekommt einen Satz Grobanteilkoeffizienten $a^{(0)}_{\alpha=0}, ..., a^{(0)}_{\alpha=\nu-1}$, minimiert dann den Vektor

$d_\alpha := a_\alpha^{(n)} - a_\alpha^{(n+1)}$ unter Beachtung des GLS mittels des Marquardt-Algorithmus und liefert schließlich einen Satz $a_{\alpha=0},...,a_{\alpha=\nu-1}$ konvergierter Grobanteilkoeffizienten. Für diese Funktion ist eine Routine *elementarystep* nötig, die für einen Satz $a_{\alpha=0}^{(n)},...,a_{\alpha=\nu-v}^{(n)}$ die Koeffizienten $a_{\alpha=0}^{(n+1)},...,a_{\alpha=\nu-1}^{(n+1)}$ und deren Differenzen $a_{\alpha=0}^{(n)} - a_{\alpha=0}^{(n+1)},...,a_{\alpha=\nu-1}^{(n)} - a_{\alpha=\nu-1}^{(n+1)}$ vermöge eines Elementarschrittes berechnet. Um jedoch einen Elementarschritt durchführen zu können, muss auch ein Feinanteil $\Delta\gamma$ übergeben werden; er bleibt wie bei Gillan so lange fest, bis der Grobanteil konvergiert ist. Weil diese Funktion das zentrale GLS beinhaltet geben wir sie an.

Algorithm 4 elementarystep

Require: $a_{\alpha=0}^{(n)},...,a_{\alpha=\nu-1}^{(n)}$
Require: $\Delta\gamma_{i=0},...,\Delta\gamma_{i=M-1}$
Require: symmetrize

1: $\Delta\gamma, a^{(n)} \longrightarrow \gamma$
2: $\gamma \longrightarrow \gamma'$
3: symmetrize γ'
4: $\gamma' \longrightarrow \Delta\gamma', a^{(n+1)}$
5: $d_\alpha = a_\alpha^{(n)} - a_\alpha^{(n+1)}$
6: **return** $a_{\alpha=0}^{(n+1)},...,a_{\alpha=\nu-1}^{(n+1)} \quad \wedge \quad d_{\alpha=0},...,d_{\alpha=\nu-1}$

Anschluss an den Gillan-Algorithmus Genau wie im letzten Abschnitt wird der Korrelator γ mittels der gleichen Basisfunktionen P^α und Q^α zerlegt in Grob- und Feinanteil. Ebenso bleiben der elementare und der Picard-Schritt die selben. Nach einem Elementarschritt wird jedoch der Grobanteil nun mit der Marquardtmethode bestimmt. Den Quellcode dieses Algorithmus haben wir aus [Wut05] übernommen und dann modifiziert.

Adaptive Basiswahl Mit einer dem Multigridverfahren ähnlichen Überlegung lässt sich der Algorithmus weiter beschleunigen. Wir rufen uns dazu zunächst ins Gedächtnis, dass Picard genau dann gut funktioniert, wenn der Startwert nur gut genug ist bzw. der Algorithmus in die richtige Richtung läuft. Das heißt, dass anfangs die Stabilität aus dem Marquardt-Algorithmus gezogen werden muss. Die Aufteilung zwischen Marquardt und Picard wird aber durch die Wahl der Basen reguliert. Ist nun die Anzahl der Basen anfangs sehr hoch, so wird der Rechenaufwand stark auf die wegen des Steepest-Descent-Algorithmus sehr stabile Marquardt-Methode verlagert. Nach einigen Elementarschritten ist jedoch bereits die Konvergenz in einem gewissen Rahmen gewährleistet und der Nachteil der Marquardt-Methode tritt zu Tage: Aufgrund des involvierten Steepest-Descent-Algorithmus werden in jedem Falle einige LM-Schritte berechnet, selbst wenn die relative Normänderung sehr gering ist. Genau an diesen Schritten kann Rechenzeit eingespart werden, indem die Zahl der Basen dynamisch reduziert wird. Dadurch wird sukzessive der Aufwand vom Marquardt- auf den Picard-Algorithmus verlagert. Wir haben die Anzahl immer dann reduziert, wenn die Feinnorm $||\Delta\gamma||$ sich um eine Größenordnung verringert hat. Wir nennen dieses Kriterium abkürzend einfach $\Delta||\Delta\gamma|| <$ criterium im Algorithmus. Die einfachste Realisierung der adaptiven Basiswahl besteht darin, immer die letzte Basisfunktion abzuschneiden, denn für große r haben die Korrelatoren wenig Struktur. Zu Beginn sind Basisfunktionen bei großen r jedoch sehr wohl nötig, um Divergenzen zu vermeiden, denn durch die FBT wird das GLS nichtlokal.

Adaptive Präzision im Levenberg-Marquardt-Anteil Für den LM-Algorithmus müssen Schranken für die Genauigkeit vorgegeben werden. Diese Schranken beinhalten die Präzision mit der die Jacobimatrix berechnet wird und verschiedene Iterationsschrittweiten. Für die genaue Beschreibung der Parameter verweisen wir auf [Wut05]. Wichtig ist, dass die Konvergenz deutlich beschleunigt wird, wenn anfangs mit einer geringen Genauigkeit begonnen wird, die dann sukzessive erhöht wird, wenn die Feinnorm $||\Delta\gamma||$ sich um eine Größenordnung verringert. Diese Überlegung steht nicht im Widerspruch zur adaptiven Basiswahl, denn diese regelt die Aufteilung zwischen Grob- und Feinanteil. Hier geht es vielmehr darum, den Grobanteil nur so genau zu berechnen, wie es für die Konvergenz des Feinanteils nötig ist. Es hat sich numerisch gezeigt, dass für die Grobanteilgenauigkeit die Wahl einer um fünf Größenordnungen kleineren als der Normänderung des Feinanteils, einen Rechenzeitvorteil bei hinreichender Stabilität bietet.

Algorithm 5 Levenberg-Marquardt

Require: discretization r_i, q_m, M
Require: construct-basis, minimize, symmetrize

set startvalue γ_i
first decomposition: $\gamma_i \longrightarrow a_\alpha, \Delta\gamma_i$
repeat
  minimize: $(\Delta\gamma, a) \mapsto a^{'}$
  if $\neg||a, a^{'}|| \rightarrow 0$ **then**
    $a_\alpha = ||a, a^{'}||a_\alpha + [1 - ||a, a^{'}||]a^{'}_\alpha$
  end if
  $\Delta\gamma, a^{'} \longrightarrow \gamma$
  $\gamma \longrightarrow \gamma^{'}$
  symmetrize $\gamma^{'}$
  $\gamma^{'} \longrightarrow \Delta\gamma^{'}, a^{'}$
  if $\neg||\Delta\gamma, \Delta\gamma^{'}|| \rightarrow 0$ **then**
    $\Delta\gamma_i = ||\Delta\gamma, \Delta\gamma^{'}||\Delta\Gamma_i + [1 - ||\Delta\gamma, \Delta\gamma^{'}||]\Delta\Gamma^{'}_i$
  end if
  if $\Delta||\Delta\gamma, \Delta\gamma^{'}|| <$ criterium **then**
    adjust accuracy for lm
    adjust basis functions
  end if
until $||\Delta\gamma, \Delta\gamma^{'}_i|| < \delta_{\text{fine}}$

Fazit

Zunächst einmal ist mit diesem Algorithmus eine numerische Methode gefunden, die in der Lage ist, konvergente Lösungen für alle von uns betrachteten Systeme zu liefern. Dabei reagiert das Konvergenzverhalten sehr sensibel auf die Wahl der Basen. Weiter konnte durch die beschriebenen Verfeinerungen im Vergleich zum Gillan-Algorithmus abermals eine deutliche Rechenzeitersparnis erreicht werden. Erst mit Hilfe dieses schnellen und stabilen Algorithmus ist es möglich, systematisch die statischen Strukturfaktoren in zwei Dimensionen bei starker Wechselwirkung zu untersuchen.

4. Resultate der statischen Korrelatoren

In diesem Kapitel werden wir in vier Abschnitten die Statik der von uns betrachteten Systeme vorstellen. In jedem dieser Abschnitte werden wir zunächst jeweils die Potentiale und ihre Parameter einführen. Zwar sind die Potentiale in der Einleitung (Kapitel 1) bereits kurz erwähnt worden; wir werden sie hier dennoch der Übersichtlichkeit halber nochmals auflisten. Anschließend diskutieren wir die mit Hilfe des LM-Algorithmus (Abschnitt 3.5) gewonnenen Resultate für die statische Paarverteilungsfunktion $\mathbf{g}(r)$ und den statischen Strukturfaktor $\mathbf{S}(q)$. Wir beschränken uns dabei auf diese beiden Korrelatoren, da für die MCT (siehe Kapitel 5) nur die Kenntnis vom Strukturfaktor nötig ist und weiter die physikalisch leicht zu interpretierende Paarverteilungsfunktion durch FBT aus dem Strukturfaktor hervorgeht. Des weiteren werden wir Vergleiche zu bereits bekannten Ergebnissen, insbesondere dem Experiment [HKM05], ziehen.

4.1. Monodisperse harte Scheiben

Für die monodispersen harten Scheiben in zwei Dimensionen wurden zwar bereits die statischen Paarverteilungsfunktionen numerisch errechnet [Lad68], dennoch wollen wir die Statik dieses Systems der Vollständigkeit halber hier kurz diskutieren, da wir den statischen Strukturfaktor für die MCT (siehe Kapitel 5) benötigen werden. Außerdem werden wir einige allgemeine Aussagen über $g(r)$ und $S(q)$ anhand dieses Systems diskutieren. Weiter werden wir die mit PY gewonnenen Lösungen in zwei und drei Dimensionen miteinander vergleichen.

Potential und Parameter Das Wechselwirkungspotential der monodispersen harten Scheiben lautet:

$$V(r) := \begin{cases} \infty & \frac{r}{\sigma} \leq 1 \\ 0 & \frac{r}{\sigma} > 1 \end{cases} \tag{4.1}$$

Das System der monodispersen harten Scheiben lässt sich allein durch die Packungsdichte $\eta := \frac{\pi}{4}\rho\sigma^2$ (siehe Kapitel 1) kontrollieren. Dabei stellt $\rho = \frac{N}{F}$ wieder die Teilchendichte und σ den Durchmesser der Teilchen dar.

Variation der Packungsdichte In Abb. 4.1 und Abb. 4.2 ist die Paarverteilungsfunktion aus PY- bzw HNC-Approximation für verschiedene Packungsdichten dargestellt. Dabei wurden repräsentativ zwei hohe Packungsdichten $\eta = \{0.59; 0.55\}$ und drei niedrige Packungsdichten $\eta = \{0.4; 0.2; 0.01\}$ gewählt. Da im Intervall $0 \leq \frac{r}{\sigma} < 1$ $g(r) \equiv 0$ per Definition des Potentials gilt, werden die Graphen erst bei $\frac{r}{\sigma} = 1$ begonnen. Mit der Packungsdichte ändern sich auch Position und Höhe der Extrema. Die

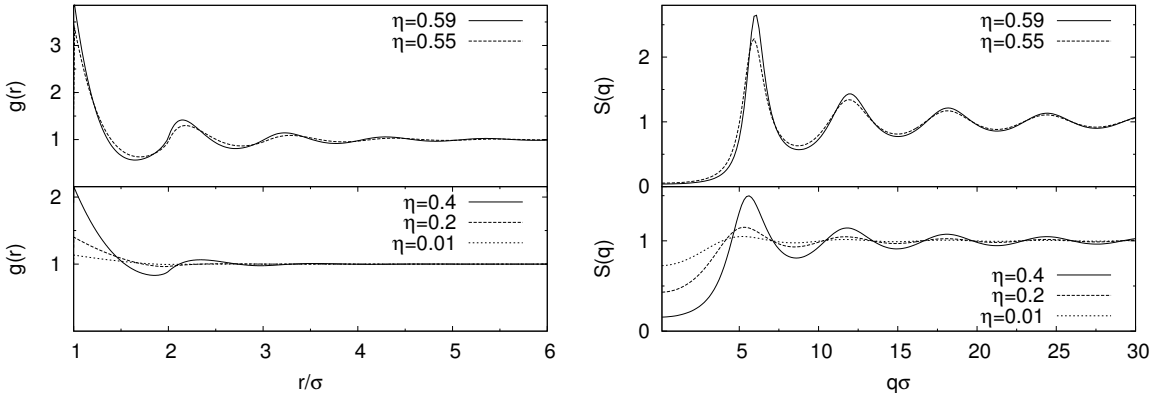

Abbildung 4.1.: $g(r)$ und $S(q)$ für monodisperse harte Scheiben mit PY - Abhängigkeit von der Packungsdichte

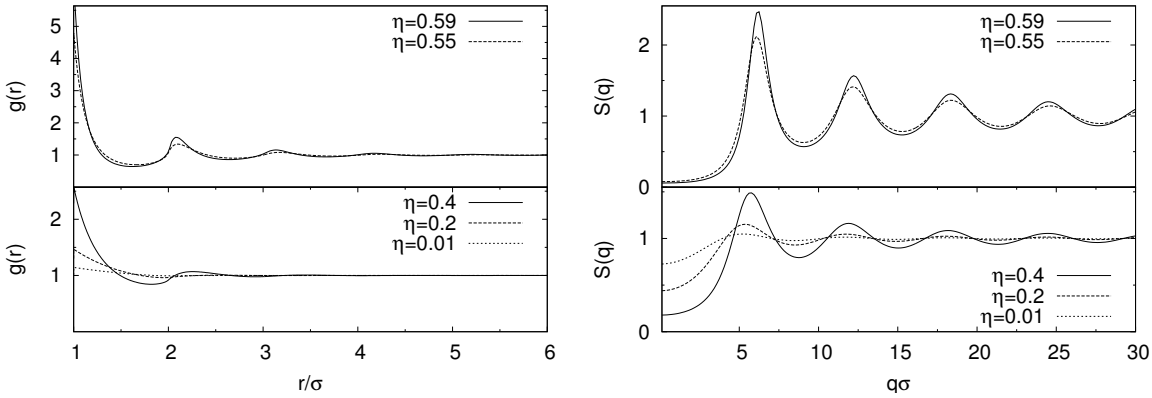

Abbildung 4.2.: $g(r)$ und $S(q)$ für monodisperse harte Scheiben mit HNC - Abhängigkeit von der Packungsdichte

mit der Packungsdichte wachsende Ausprägung der Extrema lässt sich erklären, wenn man beachtet, dass die Packungsdichte mit dem mittleren Abstand $\bar{r}$ der Teilchen über

$$\bar{r} = \sqrt{\frac{\pi}{\eta}}\frac{\sigma}{2}$$

zusammenhängt. Je dichter die Teilchen aber gepackt sind, desto mehr nähert sich ihr mittlerer Abstand der unstetigen Stelle $\frac{r}{\sigma} = 1$ des Potentials (4.1) an. Das heißt, die Teilchen wechselwirken mit wachsender Packungsdichte stärker miteinander und die Struktur wird deutlicher (siehe auch Abschnitt 2.4). Der Abfall der Kurve mit wachsendem r bedeutet, dass die Korrelation eines Teilchen mit den n-ten nächsten Nachbarn stärker ist, als die mit den $n+1$-ten nächsten Nachbarn. Weiter ist die Breite des n-ten Maximum ein Maß für die Lokalisierung der n-ten nächsten Nachbarn. Unter Lokalisierung der n-ten nächsten Nachbarn wollen wir die Schwankung der jeweiligen n-ten Nächstnachbarnabstände verstehen. Mit wachsender Packungsdichte wird die Lokalisierung also stärker, da die Breite kleiner wird. Weiter verschieben sich die Peakpositionen mit wachsender Packungsdichte zu kleinerem r, da sie ein Maß für den mittleren Abstand des jeweiligen Nächstnachbarringes sind. Besonders die ersten beiden Peaks bei $\frac{r}{\sigma} = 1$ und $\frac{r}{\sigma} \approx 2\frac{\bar{r}}{\sigma}$ weisen eine mit dem Auge deutlich sichtbare Asymmetrie auf. Ihre physikalische Deutung werden wir am ersten Peak bei $\frac{r}{\sigma} = 1$ vornehmen. Angenommen ein beliebiges festes Teilchen habe erste nächste Nachbarn. Diese nächsten Nachbarn äußern sich sowohl im ersten Peak bei $\frac{r}{\sigma} = 1$, als auch in dem Dip bei $\frac{r}{\sigma} \approx 1.6$. Dieses lokale Minimum mit $g(r) < 1$ ist dadurch zu erklären, dass die z nächsten Nachbarn selbst eine gewisse Fläche $\tilde{F} := z\frac{\pi\sigma^2}{2}$ haben. In dieser ausgeschlossenen Fläche $\tilde{F}$ kann sich kein weiteres Teilchen aufhalten. Die Paarverteilungsfunktion gibt nun aber gerade die statistisch gemittelte Verteilung der Teilchen an und muss damit den Effekt der durch die nächsten Nachbarn ausgeschlossenen Fläche $\tilde{F}$ in einem lokalen Minimum widerspiegeln. Das heißt allgemeiner, dass beliebige nächste Nachbarn sowohl ein Maximum, als auch ein Minimum in $g(r)$ erzeugen. Nun sind die ersten nächsten Nachbarn stärker lokalisiert, als die übernächsten Nachbarn, was auch aus der Frage nach der Unterscheidung zwischen den einzelnen Nächstnachbarringen folgt. Weil aber das Minimum der n-ten nächsten Nachbarn an das Maximum der $n+1$-ten nächsten Nachbarn angrenzt, tritt genau an dieser Stelle eine Asymmetrie auf.

Betrachten wir die Abhängigkeit der Paarverteilungsfunktion von der Packungsdichte etwas genauer. Da nach obigen Bemerkungen die Breite eines Maximums nicht unabhängig von seiner Höhe ist, berechnen wir nun die mittlere Zahl der ersten nächsten Nachbarn $Z_{2\mathrm{d}}$. Sie ergibt sich durch eine Integration zwischen den zwei ersten lokalen Minima $r_{\mathrm{min}1}$ und $r_{\mathrm{min}2}$ der Paarverteilungsfunktion:

$$z_{2\mathrm{d}}(\eta) := 8\eta \int_{r_{\mathrm{min}1}}^{r_{\mathrm{min}2}} drrg(r) \tag{4.2}$$

Betrachten wir nun die zwei dimensionale Paarverteilungsfunktion in Abb. 4.4 bei den Packungsdichten $\eta = 0.1$ und $\eta = 0.52$. Führen wir eine numerische Integration zwischen den ersten Minima durch, so folgt für die Anzahl der nächsten Nachbarn

$$\begin{aligned} z_{2\mathrm{d}}(\eta = 0.1) &= 1.26 \\ z_{2\mathrm{d}}(\eta = 0.5) &= 5.25 \end{aligned} \tag{4.3}$$

Das heißt es ergibt sich, wie zu erwarten, für eine höhere Packungsdichte eine größere Anzahl nächster Nachbarn. Im Grenzfall sehr kleiner Packungsdichten wird sich das System einem idealen Gas annähern. Für ein ideal homogenes System sollte dann die Zahl der nächsten Nachbarn 4 sein. Die

niedrigere Zahl nächster Nachbarn lässt sich aufgrund von Teilchenfluktuationen erklären. Im Grenzfall hoher Packungsdichten nähert sich das System der dichtesten Packung mit in zwei Dimensionen maximal sechs nächsten Nachbarn an.

Im statischen Strukturfaktor (siehe Abb. 4.3) beeinflusst die Packungsdichte ebenfalls Höhe, Breite und Position der Extrema. Die Position der Maxima ändert sich wie erwartet: Bei höherer Packungsdichte wird der mittlere Impuls $\bar{q}$ größer und damit verschieben sich die Maxima, die gerade bei $\frac{2\pi}{\bar{r}}$ liegen, zu größeren q hin. Die Ausprägung der Extrema ändert sich mit der Packungsdichte qualitativ analog zu denen der Paarverteilungsfunktion. Weiter kann an der Stelle $q = 0$ aus dem Strukturfaktor die Kompressibilität des Systems abgelesen werden [HM86]. Mit steigender Packungsdichte η kann das System schlechter komprimiert werden und die Kompressibilität sinkt, was im Einklang mit dem Verhalten des Strukturfaktors in Abb. 4.3 ist.

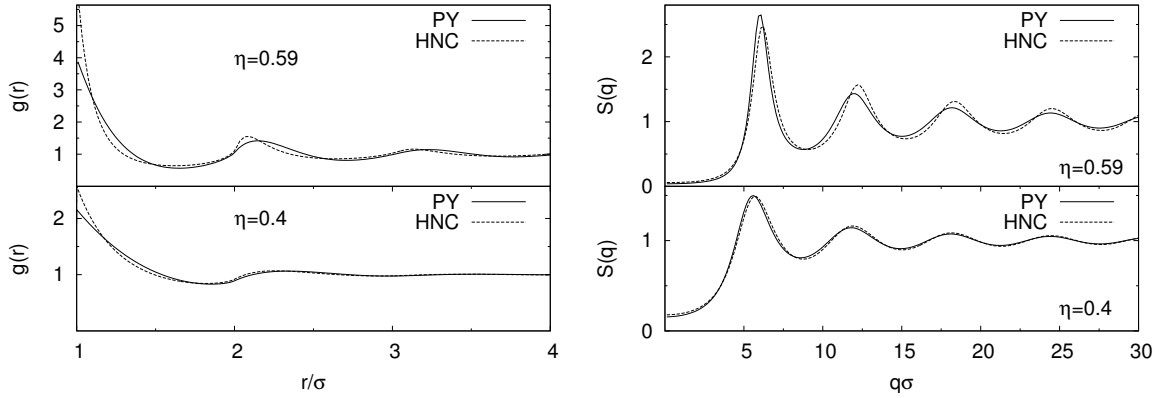

Abbildung 4.3.: $g(r)$ und $S(q)$ für monodisperse harte Scheiben - Vergleich zwischen PY und HNC bei verschiedenen Packungsdichten

Vergleich von Percus-Yevick und Hypernetted-Chain Wie in Abb. 4.3 zu erkennen ist, liefert die HNC-Approximation schärfer ausgeprägte Strukturen als PY. Insbesondere bei hohen Packungsdichten tritt diese Abweichung auf. Es ist bekannt [HM86], dass die PY-Approximation für hohe Packungsdichten schlechter wird. Das liegt daran, dass letztere eine Linearisierung der HNC ist. Numerisch ist die HNC-Approximation für harte Scheiben in zwei Dimensionen wesentlich anfälliger für Divergenzen (im Sinne von Abschnitt 3.3), als die PY-Approximation. Die Ursache dafür liegt allerdings nicht an der größeren Struktur der Korrelatoren im Vergleich zu PY, sondern wie wir im nächsten Abschnitt (Abschnitt 4.2) zeigen werden, in der Unstetigkeit des Potentials (4.1) des Systems bei $\frac{r}{\sigma} = 1$.

Diese Resultate sowohl aus PY, als auch aus HNC sind in guter Übereinstimmung mit den experimentellen Daten in [MBvG03].

Vergleich zu dreidimensionalen harten Kugeln Wir werden nun die PY-Lösungen für die zwei- und dreidimensionalen harten Teilchen miteinander vergleichen. Dazu betrachten wir zunächst Systeme gleicher Packungsdichte η, die in drei Dimensionen als $\eta = \rho\sigma^3\frac{\pi}{6}$ definiert ist. Die Lösungen in zwei Dimensionen wurden mit Hilfe des LM-Algorithmus iteriert. Der dreidimensionale Strukturfaktor $S(q)$ kann mit Hilfe von OZ aus dem exakten Ergebnis für $c(r)$ [Wer63] gewonnen werden.

Die Paarkorrelationsfunktion wurde mittels des analytischen Ausdrucks in [SH70][1] errechnet. Sowohl $S(q)$, als auch $g(r)$ lassen sich damit für dreidimensionale harte Kugeln analytisch als Funktion der Packungsdichte η bestimmen. In Abb. 4.4 ist die Paarverteilungsfunktion für harte Scheiben und Kugeln bei Packungsdichten von $\eta = 0.1$ und $\eta = 0.5$ angegeben. Für die höhere Packungsdichte von $\eta = 0.5$ ergeben sich deutlichere Unterschiede zwischen den beiden Systemen in Position und Höhe der Maxima. Wie zu erwarten nähern sich beide Systeme bei geringerer Packungsdichte einem idealen Gas an und die Ausprägung der Struktur der Korrelatoren lässt damit nach. Dennoch bleiben die strukturellen Unterschiede der beiden Systeme sichtbar. Die unterschiedlichen Extrema in den Korrelatoren der beiden Systeme resultieren hauptsächlich aus dem unterschiedlichen mittleren Abstand der Teilchen in den Systemen: In drei Dimensionen gilt $\bar{r}_{3d} = (\eta\frac{6}{\pi})^{-1/3}\sigma$, in zwei Dimensionen dagegen: $\bar{r}_{2d} = (\eta\frac{4}{\pi})^{-1/2}\sigma$. Für die Packungsdichte von $\eta = 0.5$ ergibt sich dann $\bar{r}_{3d} \approx 1.01\sigma$ und $\bar{r}_{2d} \approx 1.25\sigma$ (in Abb. 4.4 sind diese mittleren Abstände durch Pfeile gekennzeichnet). Die Systeme haben also verschiedene mittlere Abstände.

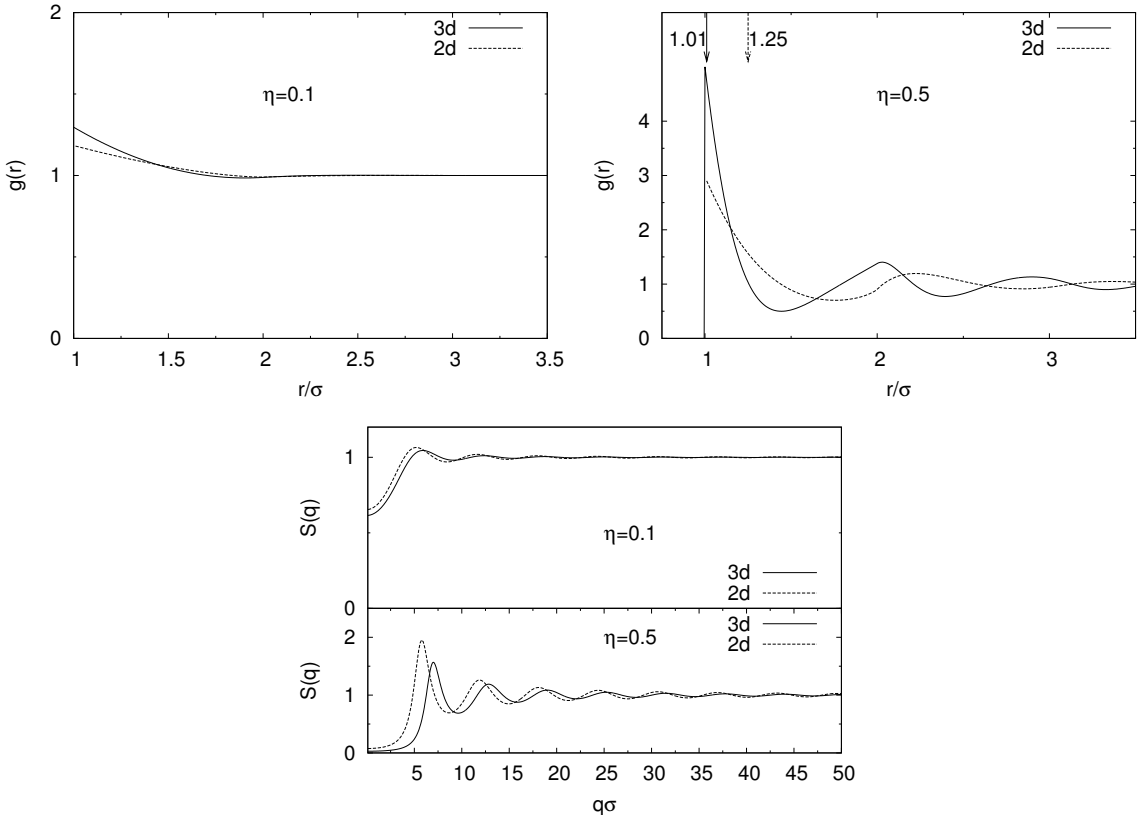

Abbildung 4.4.: $g(r)$ und $S(q)$ für monodisperse harte Teilchen; die Pfeile in $g(r)$ bei $\eta = 0.5$ kennzeichnen die mittleren Abstände $\frac{\bar{r}_{3d}}{\sigma} = 1.01$ und $\frac{\bar{r}_{2d}}{\sigma} = 1.25$ - Vergleich zwischen 2d und 3d mit PY bei zwei Packungsdichten

Damit ist die Motivation für eine weitere Fragestellung gegeben: Wie unterscheiden sich die Korrelatoren der Systeme bei einem ähnlichen mittleren Abstand $\bar{r}$? Dazu wählen wir eine Packungsdichte in der Umgebung der für den Glasübergang kritischen Packungsdichte: $\eta_{3d} = 0.52$ (aus [Göt89]) und

[1] Die approximative Formel für die Nullstellen t_i wird erst für Packungsdichten im Bereich von $\eta \approx 10^{-2}$ gut. Für die vorliegenden höheren Packungsdichten wurden sie numerisch bestimmt.

$\eta_{2d} = 0.71$ (aus Abschnitt 7.1), d.h. $\bar{r}_{3d} \approx 1.02\sigma$ und $\bar{r}_{2d} \approx 1.02\sigma$. Die Paarverteilungsfunktion und der Strukturfaktor sind in Abb. 4.5 abgebildet. In der Paarverteilungsfunktion der beiden Systeme zeigt sich eine bessere Übereinstimmung der Extremstellen als in Abb. 4.4; gleiches gilt für den Strukturfaktor. Dies untermauert den zuvor beschriebenen Zusammenhang zwischen mittlerem Abstand und Extremstellen. Die noch verbleibenden Abweichungen in der Position haben ihren Ursprung in der unterschiedlichen dichtesten Anordnung in zwei und drei Dimensionen. Stimmt die Paarverteilungsfunktion qualitativ für die beiden Systeme bei gleichem mittleren Abstand noch recht gut überein, so zeigen sich jedoch deutliche Unterschiede in den Maxima der Strukturfaktoren. Die harten Scheiben haben deutlich höhere Peaks als die harten Kugeln.

Wieder kann die Anzahl der nächsten Nachbarn mittels Integration zwischen den ersten beiden lokalen Minima r_{min1} und r_{min2} der Paarkorrelationsfunktion errechnet werden. In drei Dimensionen ist die maximale Zahl nächster Nachbarn 12 und es gilt

$$z_{3d}(\eta) := 24\eta \int_{r_{\text{min1}}}^{r_{\text{min2}}} dr r^2 g(r) \tag{4.4}$$

woraus sich für die in Abb. 4.4 angegebenen Paarverteilungsfunktionen ergibt

$$\begin{aligned} z_{3d}(\eta = 0.1) &= 4.99 \\ z_{3d}(\eta = 0.5) &= 9.56 \end{aligned}$$

Das heißt einerseits ergibt sich die zu erwartende Abhängigkeit der Zahl der nächsten Nachbarn von der Packungsdichte und weiter ist bei gleichem Bedeckungsgrad η die Zahl der nächsten Nachbarn in drei Dimensionen größer, als in zweien (4.3).

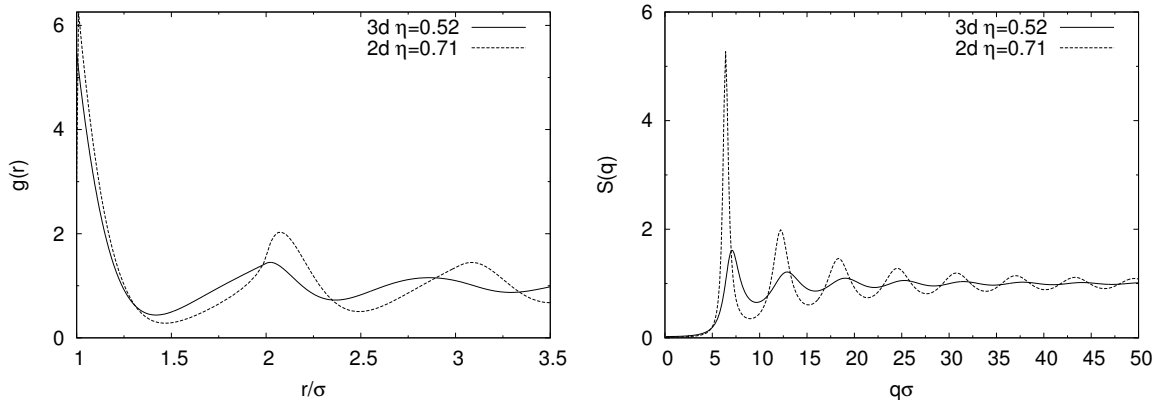

Abbildung 4.5.: $g(r)$ und $S(q)$ für monodisperse harte Teilchen - Vergleich zwischen 2d und 3d mit PY bei verschiedenen Packungsdichten und gleichem mittleren Abstand $\bar{r} = 1.01\sigma$

4.2. Monodisperse dipolare harte Scheiben

Potential und Parameter Motiviert durch das Experiment [HKM05] benutzen wir die dipolare Wechselwirkung zwischen zwei magnetischen Teilchen. Sei Teilchen 1 im Koordinatenursprung und

Teilchen zwei am Ort $\vec{r}$, dann erzeugt Teilchen eins aufgrund des von außen anliegenden Feldes $\vec{B}$ ein Magnetfeld

$$\vec{B}_1(\vec{r}) = \frac{\mu_0}{4\pi}\frac{3(\vec{r}\cdot\vec{M}_1)\vec{r} - r^2\vec{M}_1}{r^5} \tag{4.5}$$

am Ort von 2, wobei $\vec{M}_\alpha = \chi_\alpha \vec{B}$ das magnetische Moment des Teilchens α ist. Die Wechselwirkung zwischen den beiden Teilchen lautet $V(\vec{r}) = -\vec{M}_2 \cdot \vec{B}_1$. In dem von uns betrachteten Fall sind die Teilchen in einer Ebene, ihre magnetischen Momente sind parallel ausgerichtet mit $\vec{r} \perp \vec{M}_{1,2}$ und das dipolare Potential ist damit repulsiv. Insgesamt lautet das Potential dann:

$$V(r) := \begin{cases} \infty & \frac{r}{\sigma} \leq 1 \\ \frac{\mu_0}{4\pi}\frac{(\chi B)^2}{(\frac{r}{\sigma})^3\sigma^3} & \frac{r}{\sigma} > 1 \end{cases} \tag{4.6}$$

Die magnetische Suszeptibilität $\chi = 6.6[\mathrm{p}\frac{\mathrm{Am}^2}{\mathrm{T}}]$ und auch der Teilchenradius $r = 1.4[\mu\mathrm{m}]$ wurden entsprechend der kleinen Teilchen aus [HKM05] gewählt. Weiter wurde das äußere Magnetfeld $B = 1[\mathrm{mT}]$ gesetzt und lediglich die Temperatur $T[\mathrm{K}]$ wird variiert. Die für ein System charakteristische Temperatur kann dann mittels $T_{\mathrm{char}} = \frac{V(\bar{r})}{k_B}[\mathrm{K}]$ berechnet werden. Für die Numerik wurde immer η und T vorgegeben (siehe unten). Der physikalisch relevante Kontrollparameter des Systems ist für kleine Packungsdichten (siehe weiter unten) die mittlere Wechselwirkungsenergie Γ_{m} des Systems, die für den monodispersen Fall lautet

$$\Gamma_{\mathrm{m}} := \frac{\mu_0}{4\pi}B^2\frac{(\rho\pi)^{\frac{3}{2}}}{k_B T}\chi^2 \tag{4.7}$$

Dabei ist $\rho = \frac{N}{F}$ die Teilchendichte, k_B die Boltzmannkonstante und μ_0 die magnetische Permeabilitätskonstante. Die Definition der mittleren magnetischen Wechselwirkungsenergie beinhaltet die bereits gezeigte (siehe (2.6)) Äquivalenz verschiedener Temperatur-Dichte-Paare.

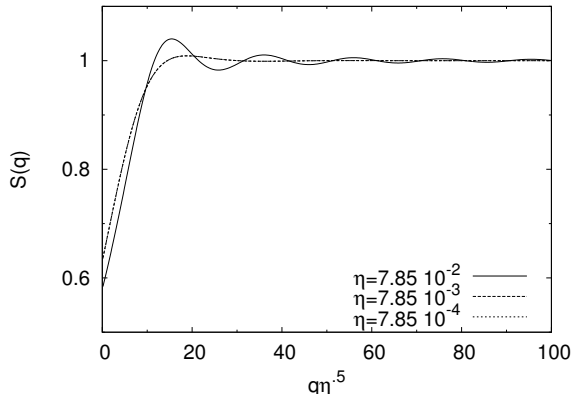

Abbildung 4.6.: $S(q)$ für monodisperse dipolare harte Scheiben bei $\Gamma_{\mathrm{m}} = 0.1$ aus PY bei verschiedenen Packungsdichten - Güte von Γ_{m} als Systemparameter bei den Packungsdichten $\eta = \{7.85\cdot10^{-2}; 7.85\cdot10^{-3}; 7.85\cdot10^{-4}\}$

Variation der Packungsdichte und Äquivalente Strukturfaktoren In (2.6) hatten wir gezeigt, dass es für ein reines r^{-n}-Potential äquivalente Strukturfaktoren gibt. Ausgenutzt haben wir das dann in obiger Definition (4.7) der mittleren magnetischen Wechselwirkungsenergie. Diesen physikalischen

Parameter wollen wir nun etwas genauer betrachten. Wird die Packungsdichte des Systems der harten dipolaren Scheiben variiert, stellt sich neben den bereits für die monodispersen Scheiben diskutierten Effekten noch ein weiterer ein. Da durch die Packungsdichte gleichzeitig der mittlere Abstand geändert wird, ist entweder bei hohen Packungsdichten das Harte-Scheiben-Potential oder bei geringen Packungsdichten das dipolare Potential dominant. Wie in (2.6) gezeigt, gibt es für reine dipolare Wechselwirkung eine physikalische Äquivalenz bestimmter Temperatur-Dichte-Paare. Weil nun aber das Potential auch einen Harte-Scheiben-Anteil hat, ist zu erwarten, dass diese Äquivalenz nur für geringe Packungsdichten gilt, damit der mittlere Teilchenabstand groß ist und der harte Scheiben-Anteil vernachlässigt werden kann. Um dies kurz numerisch zu verifizieren betrachten wir ein System dipolarer harter Scheiben mit $\Gamma_m = 0.1$ bei den Packungsdichten $\eta = \{7.85 \cdot 10^{-2}; 7.85 \cdot 10^{-3}; 7.85 \cdot 10^{-4}\}$. Die zu diesen Packungsdichten gehörenden mittleren Abstände lauten $\bar{r}\sigma^{-1} = \{3.16; 10; 31.62\}$. An den Strukturfaktoren in Abb. 4.6 ist zu erkennen, dass die Äquivalenzaussage erst ab einem mittleren Abstand von $\bar{r}\sigma^{-1} \approx 10$ wahr ist: Die Strukturfaktoren der Systeme mit den beiden sehr kleinen Packungsdichten liegen direkt aufeinander. Damit ist klar, warum für die Numerik sowohl η als auch T vorgegeben werden müssen.

Variation der Temperatur Die dipolare Wechselwirkung kann mittels der Temperatur in ihrer Stärke kontrolliert werden. Um die Auswirkung der Temperatur auf die beiden Korrelatoren $S(q)$ und $g(r)$ zu dokumentieren, betrachten wir ein System dipolarer harter Scheiben mit einer niedrigen Packungsdichte $\eta = 0.004$ (siehe Abb. 4.7) und eines mit einer hohen Packungsdichte $\eta = 0.67$ (siehe Abb. 4.8). Für sehr hohe Temperaturen nähern sich die Korrelatoren der dipolaren harten Scheiben immer mehr denen der reinen harten Scheiben an. Betrachten wir nun zunächst die Paarverteilungsfunktion: Mit wachsender Wechselwirkungsstärke steigt das erste Maximum an. Da mit wachsender Wechselwirkungsstärke der mittlere Abstand der Teilchen abnimmt, sind insbesondere die ersten nächsten Nachbarn weniger stark lokalisiert. Da jedoch die Anzahl der nächsten Nachbarn gleich bleibt, wird der Peak kleiner. Wie zu erwarten war, wird die Korrelation der Teilchen bei wachsender Wechselwirkung immer langreichweitiger.

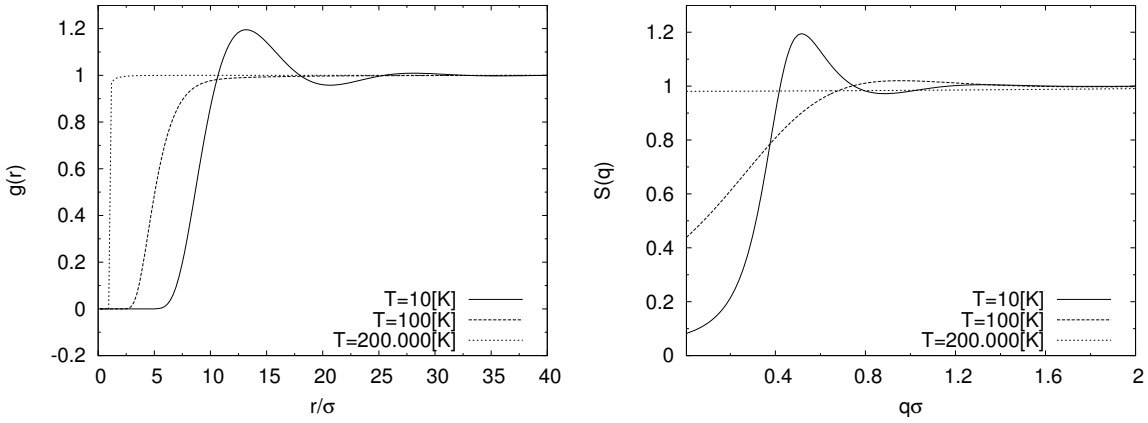

Abbildung 4.7.: $g(r)$ und $S(q)$ für monodisperse dipolare harte Scheiben aus PY - Temperaturabhängigkeit bei niedriger Packungsdichte $\eta = 0.004$; dabei ist $\Gamma_m = \{6.2 \cdot 10^{-1}; 6.2 \cdot 10^{-2}; 3 \cdot 10^{-5}\}$

Vergleich von Percus-Yevick und Hypernetted-Chain Wie bei den reinen harten Scheiben ergibt sich, dass die HNC-Approximation stärker ausgeprägte Extrema liefert. Untersuchen wir weiter das

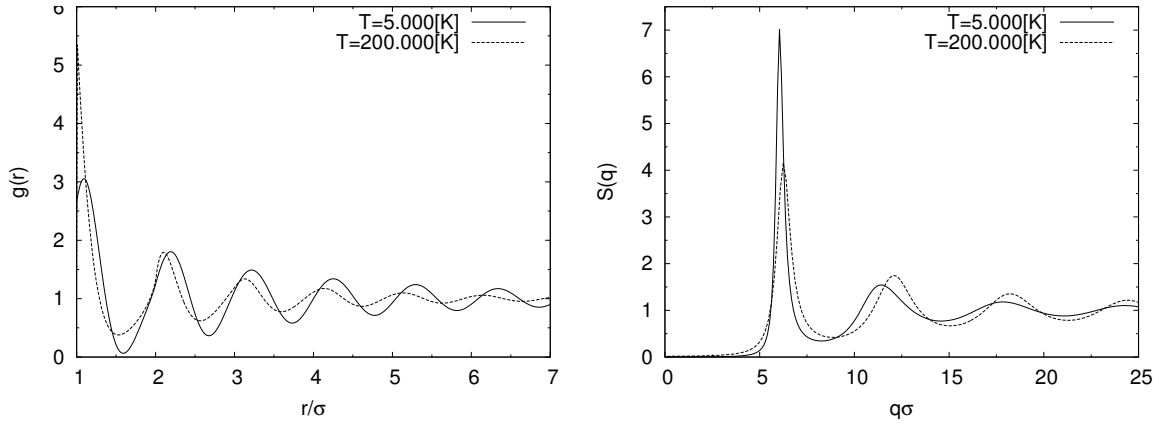

Abbildung 4.8.: $g(r)$ und $S(q)$ für monodisperse dipolare harte Scheiben aus PY - Temperaturabhängigkeit bei hoher Packungsdichte $\eta = 0.67$; dabei ist $\Gamma_m = \{2.6; 6.6 \cdot 10^{-2}\}$

Konvergenzverhalten der PY- und HNC-Approximation, so konvergiert die HNC-Approximation für starke dipolare Wechselwirkung wesentlich schneller als PY. Die Ursache dafür muss in der kleiner werdenden Unstetigkeit des Potentials bei $\frac{r}{\sigma} = 1$ liegen. Das Potential geht in beiden Approximationen nur über den Faktor $e^{-\beta V(r)}$ ein. Für harte Scheiben ergibt sich für diesen Faktor bei $\frac{r}{\sigma} = 1$ ein Sprung von 0 auf 1; für die dipolaren harten Scheiben wird dieser Sprung mit steigender Temperatur immer kleiner (siehe Abb. 4.9).

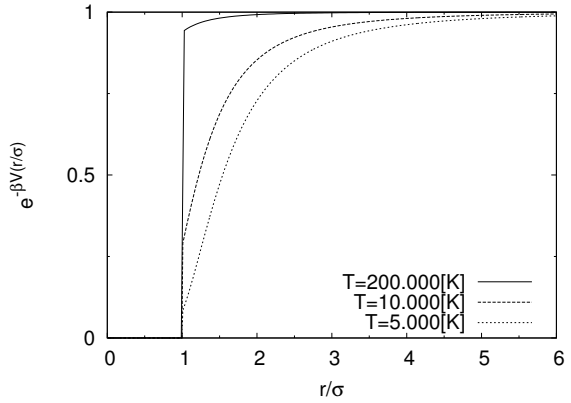

Abbildung 4.9.: Faktor $e^{-\beta V(r)}$ für das Potential dipolarer harter Scheiben bei verschiedenen Temperaturen; dabei ist $\Gamma_m = \{2.7; 1.3; 6.6 \cdot 10^{-2}\}$

4.3. Binäre harte Scheiben

Potential und Parameter Wir betrachten nun ein System mit zwei Sorten harter Scheiben. Die Teilchen haben die Radien r_1 und r_2. Damit können die drei Durchmesser

$$\sigma_{\alpha\beta} := r_\alpha + r_\beta \tag{4.8}$$

definiert werden. Nun muss zunächst eine neue Skalierung eingeführt werden: Die reduzierten Einheiten beziehen sich bei den binären Systemen im folgenden auf den mittleren Durchmesser $\sigma_{12} := r_1 + r_2$. Das heißt der Ort r wird in Einheiten $\frac{r}{\sigma_{12}}$ und der Impuls in Einheiten $q\sigma_{12}$ gemessen. Des weiteren definieren wir das Radienverhältnis

$$\delta := \frac{r_1}{r_2} \tag{4.9}$$

Das Potential der binären harten Scheiben lautet dann:

$$\mathbf{V}_{\alpha\beta}(r) := \begin{cases} \infty & \frac{r}{\sigma_{12}} \leq \frac{\sigma_{\alpha\beta}}{\sigma_{12}} \\ 0 & \frac{r}{\sigma_{12}} > \frac{\sigma_{\alpha\beta}}{\sigma_{12}} \end{cases} \tag{4.10}$$

Die Parameter des Systems sind die *Packungsdichte*

$$\eta := \rho\frac{\pi}{4}[x_1\sigma_{11}^2 + x_2\sigma_{22}^2] \tag{4.11}$$

die Konzentrationen der Teilchensorten x_1, x_2 und das Radienverhältnis δ. O.B.d.A. geben wir nur eine Konzentration x_1 vor, denn es gilt $x_2 = 1 - x_1$. In der Definition der binären Packungsdichte ist $\rho = \frac{N}{F}$. Auf einen expliziten Vergleich zwischen HNC und PY verzichten wir hier und halten fest, dass erstere wieder stärker ausgeprägte Strukturen bei geringerer Konvergenzstabilität liefert.

Variation der Konzentration Da wir die Packungsdichte über die Konzentrationen definiert haben, und diese dem Algorithmus zusammen mit der Konzentration x_1 vorgeben, wird mit der Variation von x_1 die Dichte ρ des Systems immer so angepasst, dass die Packungsdichte η und damit der Bedeckungsgrad des Systems gleich bleibt. Betrachten wir nun also den Einfluss einer Variation der Teilchenkonzentrationen auf die Korrelatoren.

In Abb. 4.10 ist die Paarverteilungsfunktion binärer harter Scheiben für verschiedene Teilchenkonzentrationen dargestellt. Zunächst einmal sind die ersten Maxima der Matrixkomponenten wegen der Normierung auf σ_{12} gegeneinander verschoben mit $M_{11}(\delta) = \frac{2\delta}{\delta+1}$, $M_{22}(\delta) = \frac{2}{\delta+1}$ und $M_{12} = 1$. Dass die Paarverteilungsfunktion in den Fällen $x_1 = 1$ und $x_1 = 0$ wieder in das System einfacher monodisperser harter Scheiben übergeht ist klar und auch in Abb. 4.10 ersichtlich. Dabei haben die beiden Grenzfälle verschiedene Längenskalen, da $\delta = 0.5$ gewählt wurde. Besonders stark äußert sich der Einfluss der Mischung wenn $x_1 \approx 0.5$ ist. Dann ist in Abb. 4.10 zu erkennen, dass sich eine Unterstruktur bildet, deren Extremstellen sich an den Stellen befinden, an denen jeweils die beiden Grenzfälle $x_1 = 0$ und $x_1 = 1$ ihre Extremstellen haben. Anschaulich trägt diese Unterstruktur der Tatsache Rechnung, dass ein festes Teilchen einer Sorte von Teilchen beider Sorten umgeben sein wird. Betrachten wir exemplarisch die 11-Komponente der Paarverteilungsfunktion. Berechnen wir nun die Zahl der nächsten Nachbarn mit (4.12) in den Intervallen $I^{(1)} := [0.6, 1.33]$ und $I^{(2)} := [1.33, 1.79]$ für $\frac{r}{\sigma}$, so ergibt sich für die Zahl der nächsten Nachbarn $z_{2d;11}^{(1)} = 3.74$ und $z_{2d;11}^{(2)} = 2.74$. Berechnen wir weiter den mittleren Abstand dieser beiden Nächstnachbarringe über das Maximum der Paarverteilungsfunktion in dem entsprechenden Intervall, so folgt, dass bei $\bar{r}^{(1)} = 1.39\sigma$ im Mittel ein Teilchen mehr als bei $\bar{r}^{(r)} = 2.25\sigma$ zu finden sein wird. Anschaulich ist das so zu verstehen, dass der erste Nächstnachbarring der kleinen Teilchen aufgespalten wird, da angrenzende große Teilchen ein Teil der vorhandenen Fläche ausschließen.

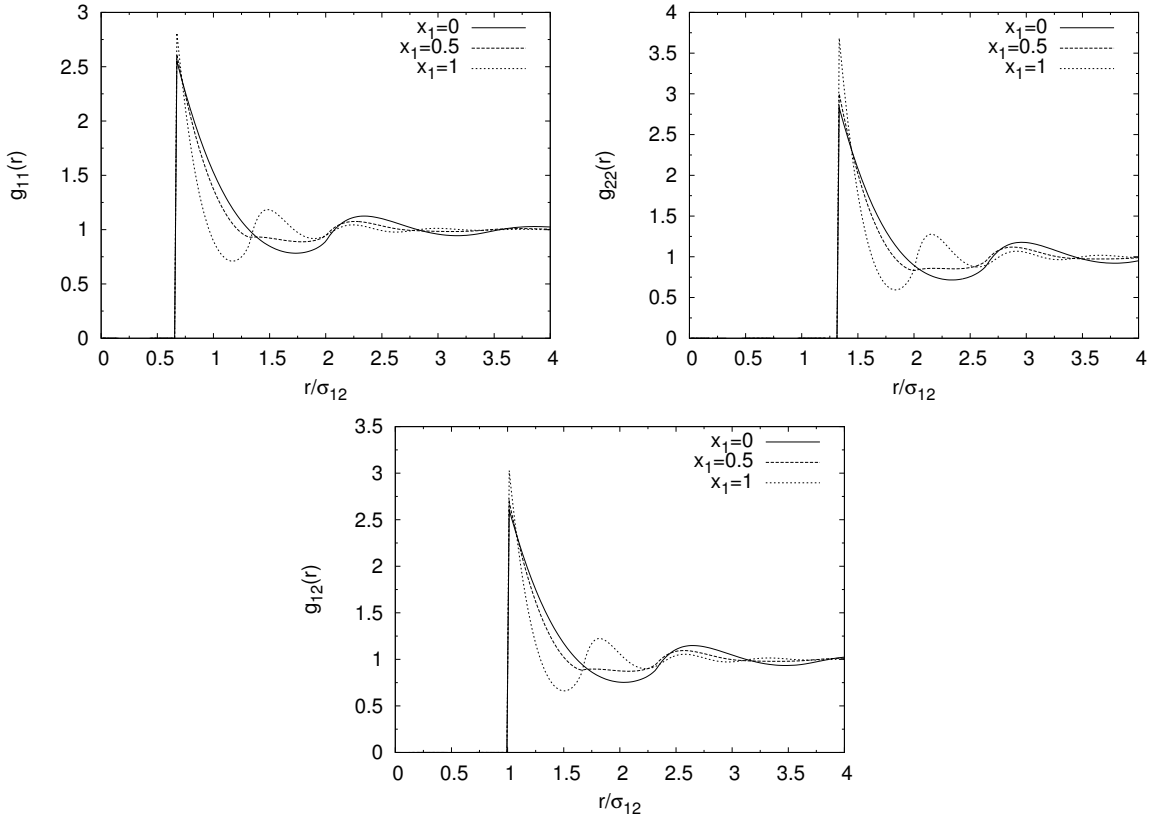

Abbildung 4.10.: $\mathbf{g}(r)$ für binäre harte Scheiben aus PY - Verschiedene Teilchenkonzentrationen bei Radienverhältnis $\delta = 0.5$ und Packungsdichte $\eta = 0.5$

Die Extrema der 11-Komponenten sind offensichtlich alle kleiner, als diejenigen der 22-Komponenten (siehe Abb. 4.10), da $r_1 = \delta r_2 < r_2$.

Betrachten wir nun wieder die Anzahl der nächsten Nachbarn um die Abhängigkeit der Peaks von der Konzentration genauer zu untersuchen. Dabei folgt der Normierungsfaktor aus der Definition der binären Packungsdichte (4.11).

$$z_{2d;\alpha\beta} = 2\eta \frac{(\delta+1)^2}{x_1(\delta^2-1)+1} \int_{r_{min1}}^{r_{min2}} drrg_{\alpha\beta}(r) \tag{4.12}$$

Die numerischen Resultate sind in Abb. 4.1 zusammengestellt. Betrachten wir die Diagonalelemente

	$x_1 = 0$	$x_1 = 0.5$	$x_1 = 1$
$z_{2d;11}$	6.02	6.48	6.71
$z_{2d;12}$	7.72	7.7	7.32
$z_{2d;22}$	9.31	6.6	4.79

Tabelle 4.1.: Anzahl nächster Nachbarn für binäre harte Scheiben bei $\delta = 0.5$ und $\eta = 0.5$ - Variation der Teilchenkonzentrationen

der nächsten Nachbarn in Abb. 4.1. Wie zu erwarten wird $z_{2d;11}$ mit wachsendem x_1 analog zu den monodispersen harten Scheiben größer, da $\eta_1 = x_1\eta$ wächst. Konsistent dazu fällt $z_{2d;22}$ mit wachsendem x_1 ab.

Interpretieren wir nun die Diagonalelemente der Korrelatoren als reine monodisperse Systeme und die Nebendiagonalelemente analog als rein gemischte Systeme[2], so stehen $z_{2d;11}$ und $z_{2d;22}$ im Widerspruch zu den monodispersen reinen harten Scheiben, für die maximal sechs nächste Nachbarn existieren. Diese Interpretation ist, wie bereits zuvor angedeutet, nicht richtig. Die Wechselwirkung der verschiedenen Teilchensorten miteinander steckt eben nicht nur in der gemischten Komponente der Korrelatoren, sondern auch implizit in deren Diagonalelementen. Denn zwischen zwei Teilchen einer Sorte kann sich durchaus ein Teilchen der anderen Sorte befinden. Betrachtet man nun die Korrelation der beiden gleichen Teilchen, so steckt das dazwischenliegende Teilchen der anderen Sorte implizit durch eine ausgeschlossene Fläche in den Korrelatoren. Betrachten wir nochmals den monodispersen Grenzfall mit $x_1 = 0$ bzw. $x_1 = 1$. Im Widerspruch zur physikalischen Anschauung sind die Paarverteilungsfunktionen (siehe Abb. 4.10) $g_{11}(r) \neq 0$ bzw. $g_{22}(r) \neq 0$ und die Zahl der nächsten Nachbarn (siehe Abb. 4.1) sind $z_{2d;11} \neq 0$ bzw. $z_{2d;22} \neq 0$. Ursache dafür ist die Kopplung der Matrixkomponenten durch die OZ-Gleichung (siehe Abschnitt 2.2).

Betrachten wir nun die zugehörigen Strukturfaktoren $\mathbf{S}(q)$ in Abb. 4.11. Wie in Kapitel 2 beschrieben, gilt in Übereinstimmung mit den numerischen Resultaten für $q \to \infty$ $S_{\alpha\beta}(q) = \delta_{\alpha\beta}x_\alpha$. Dass weiter für den Extremfall $x_1 = 0$ der Strukturfaktor pathologischer Weise nur Null oder Eins sein kann, ist aus (2.5) ersichtlich, denn $\frac{\rho_\alpha\rho_\beta}{\rho} = x_1x_2\rho = 0$. Die unterschiedlichen ersten Maximalstellen der Matrixkomponenten des Strukturfaktors lassen sich aus den mittleren Abständen der Teilchen erklären. Analog zur Paarverteilungsfunktion ist die 11-Komponente des Strukturfaktors weniger stark ausgeprägt, als die 22-Komponente. Weiter ist die Oszillation der 11-Komponente um $\delta = 0.5$ kleiner, als die der 22-Komponente. Da die großen Teilchen der Sorte zwei das System dominieren ist klar, dass für kleines x_1 die Komponente $S_{22}(q)$ immer näher an die harten Scheiben rückt.

[2]Das heißt Teilchensorte eins kann immer nur mit Teilchensorte zwei wechselwirken.

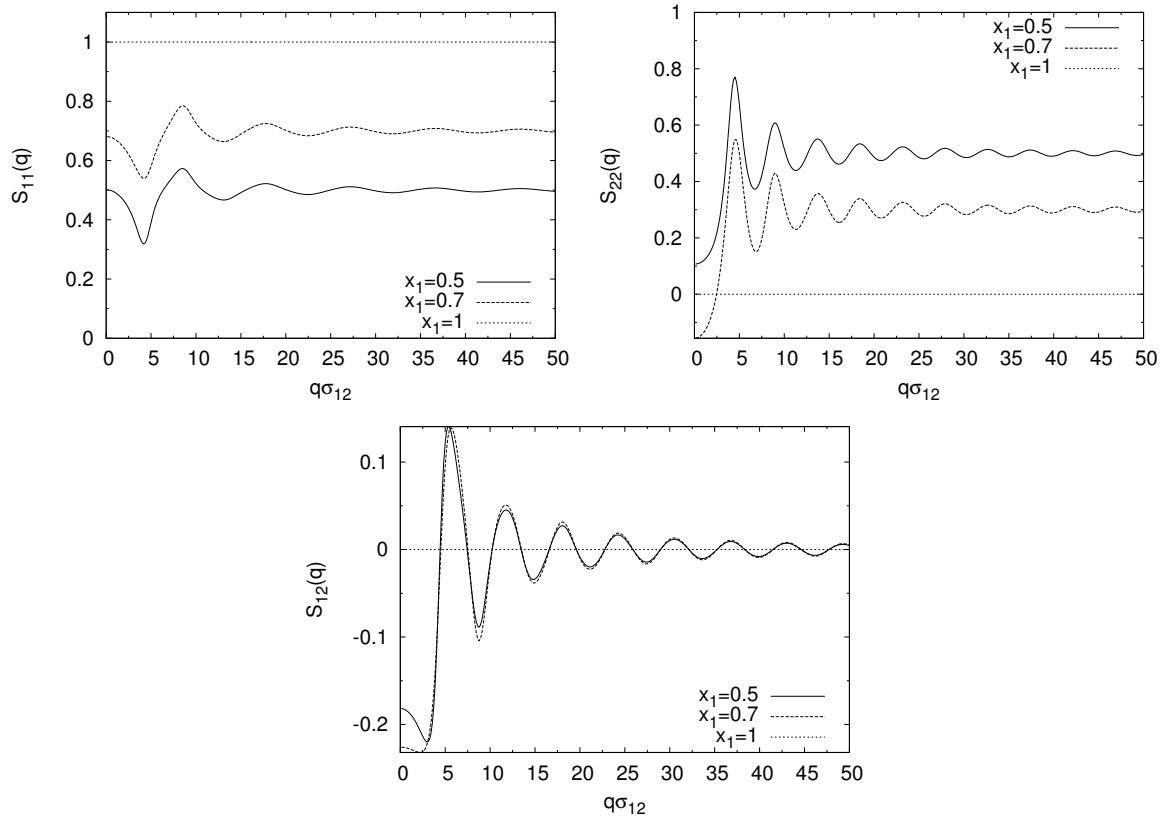

Abbildung 4.11.: $\mathbf{S}(q)$ für binäre harte Scheiben aus PY - Verschiedene Teilchenkonzentrationen bei Radienverhältnis $\delta = 0.5$ und Packungsdichte $\eta = 0.5$

Vergleichen wir den Strukturfaktor des binären Systems harter Scheiben mit dem des Systems monodisperser harter Scheiben (z.B. Abb. 4.1), ist eine zusätzliche Strukturierung ersichtlich. Genau wie bei der bereits zuvor diskutierten Paarverteilungsfunktion entsteht sie durch die implizite Wechselwirkung der verschiedenen Teilchensorten. Im Gegensatz zu rein monodispersen Systemen gibt es in polydispersen im allgemein verschiedene charakteristische Längenskalen. Diese Längenskalen der einzelnen Komponenten mischen nun miteinander über die OZ-Gleichung (2.9) und es ergeben sich Unterstrukturen, die nicht mehr allein durch monodisperse Systeme erklärt werden können.

Variation des Radienverhältnisses Betrachten wir nun eine Variation im Radienverhältnis. Da $\delta = \frac{r_1}{r_2}$ äquivalent zu δ^{-1} bei Teilchenvertauschung ist, betrachten wir nur den Fall $0 < \delta \leq 1$. Für hohe Packungsdichten ist die Wechselwirkung der harten Scheiben stark und somit ist zu erwarten, dass sich eine Änderung der Längenskalen hier umso deutlicher äußert. Für niedrige Packungsdichten hingegen, wird diese Abhängigkeit kleiner.

Insbesondere für das experimentelle System [HKM05] binärer dipolarer harter Scheiben (Abschnitt 4.4) ist das Verhältnis der magnetischen Suszeptibilitäten δ_χ (siehe Kapitel 1) relevanter. Unter dieser Motivation betrachten wir im folgenden abkürzend nur hohe Packungsdichten.

Mit dem Radienverhältnis ändert sich die Unstetigkeitsstelle des Potentials (4.10) und damit ändern sich die Extremalstellen in den Korrelatoren. Daraus ergibt sich eine leicht veränderte Form der Korrelatoren in Position und Höhe. Betrachten wir die Diagonalelemente der Matrix $\mathbf{g}(r)$ für drei Radienverhältnisse bei $\eta = 0.24$ und $x_1 = 0.5$ in Abb. 4.12. Der Sprung im Potential (4.10) ist abhängig vom Radienverhältnis und verursacht wie bereits erwähnt die Verschiebung des Sprunges in $\mathbf{g}(r)$: $M_{11}(\delta) = \frac{2\delta}{\delta+1}$, $M_{22}(\delta) = \frac{2}{\delta+1}$ und für die gemischte Komponente $M_{12} = 1$. Das ist in Abb. 4.12 ersichtlich.

Mit $\delta \to 1$ geht das System in die monodispersen harten Scheiben über, was im Vergleich zu den Graphen aus Abschnitt 4.1 folgt. Insbesondere verschwindet die Strukturierung der Korrelatoren aufgrund der Polydispersität. Wie zu erwarten ist die Unterstruktur genau dann besonders ausgeprägt, wenn die Teilchen in etwa gleich groß sind. Ansonsten sind die spezifischen Längenskalen so verschieden, dass sie mit dem Auge in den Korrelatoren nicht mehr gut zu erkennen sind.

Der Vollständigkeit halber geben wir auch die zugehörigen Strukturfaktoren an. Die Verschiebung der Diagonalelemente mit δ ist analog zur Paarverteilungsfunktion mit der Normierung auf σ_{12} zu erklären.

Vergleich zu dreidimensionalen binären harten Kugeln Wie bei den monodispersen harten Scheiben wollen wir einen Vergleich zu bereits bekannten Resultaten in drei Dimensionen ziehen. Dazu betrachten wir die Paarverteilungsfunktion dreidimensionaler harter binärer Kugeln [GV03]. Wie für das monodisperse System (Abb. 4.4 und Abb. 4.5) geben wir die Paarverteilungsfunktion in zwei Dimensionen für zwei verschiedene Packungsdichten an. Aus [GV03] übernehmen wir $\eta = 0.516$ für die dreidimensionalen harten Kugeln. Für das zweidimensionale System der binären harten Scheiben wählen wir dann einmal $\eta = 0.516$, woraus für den über alle Teilchen gemittelten mittleren Abstand folgt, dass $\bar{r}_{3d} \approx 1.01\sigma_{12}$ und $\bar{r}_{2d} \approx 1.25\sigma_{12}$. Weiter wählen wir $\eta = 0.71$ als zweite Packungsdichte für das zweidimensionale System, damit $\bar{r}_{3d} \approx \bar{r}_{2d} = 1.01\sigma_{12}$.

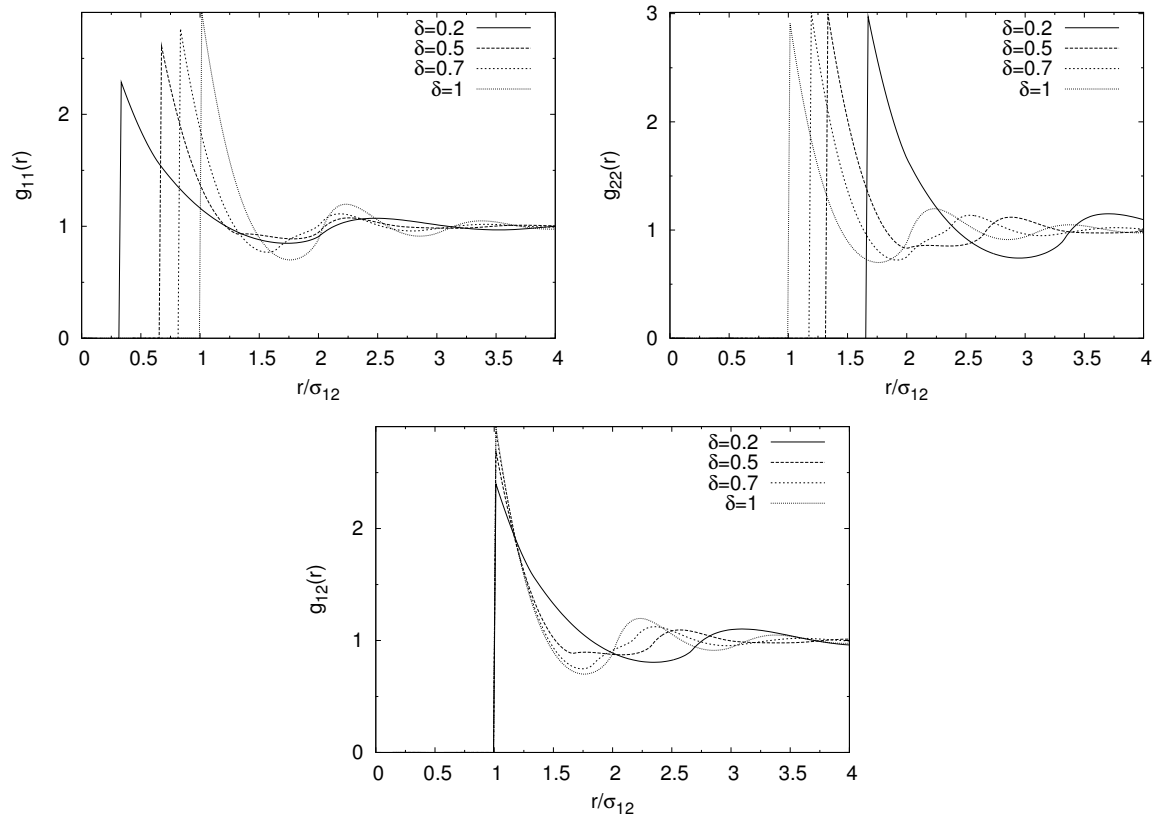

Abbildung 4.12.: $\mathbf{g}(r)$ für binäre harte Scheiben aus PY - Verschiedene Radienverhältnisse bei Teilchenkonzentration $x_1 = 0.5$ und Packungsdichte $\eta = 0.24$

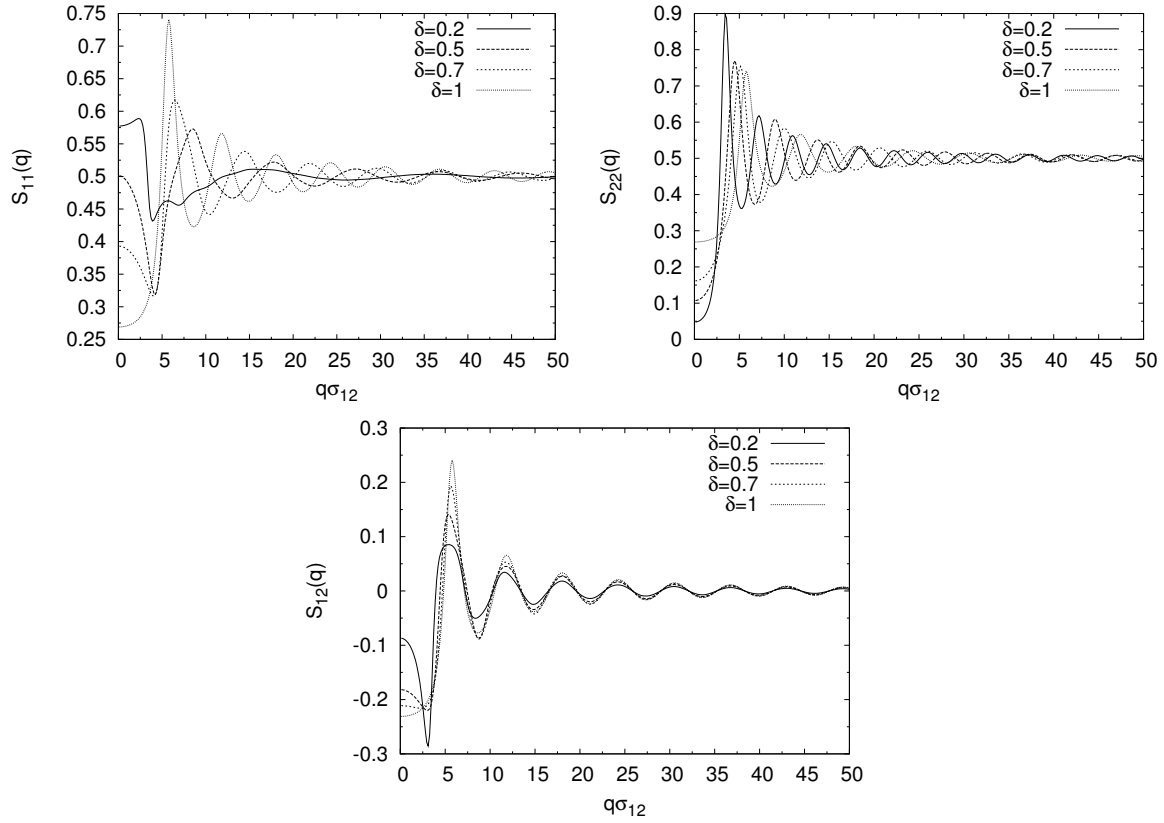

Abbildung 4.13.: $\mathbf{S}(q)$ für binäre harte Scheiben aus PY - Verschiedene Radienverhältnisse bei Teilchenkonzentration $x_1 = 0.5$ und Packungsdichte $\eta = 0.24$

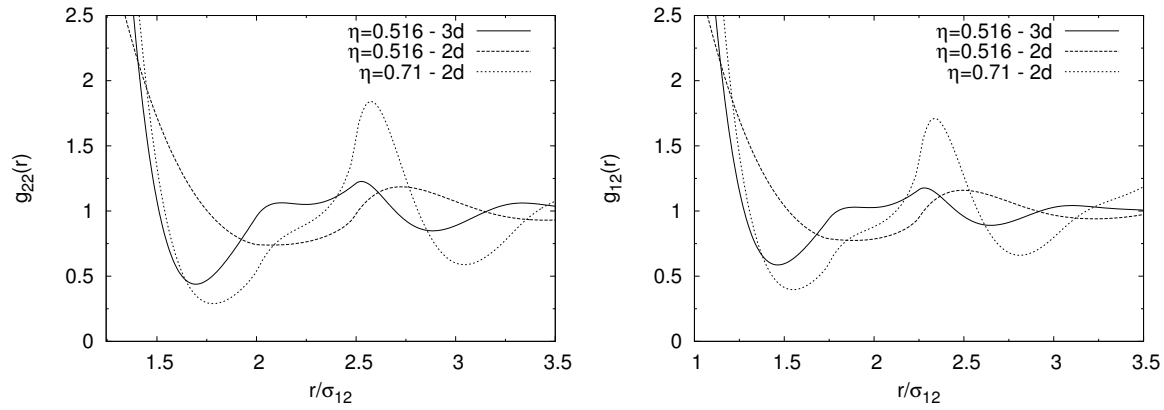

Abbildung 4.14.: $\mathbf{g}(r)$ für binäre harte Teilchen bei $\delta = 0.6$ und $x_1 = 0.2$ - Vergleich zwischen 2d und 3d mit PY bei gleicher Packungsdichte $\eta = 0.516$ und gleichem mittleren Abstand ($\eta = 0.71$)

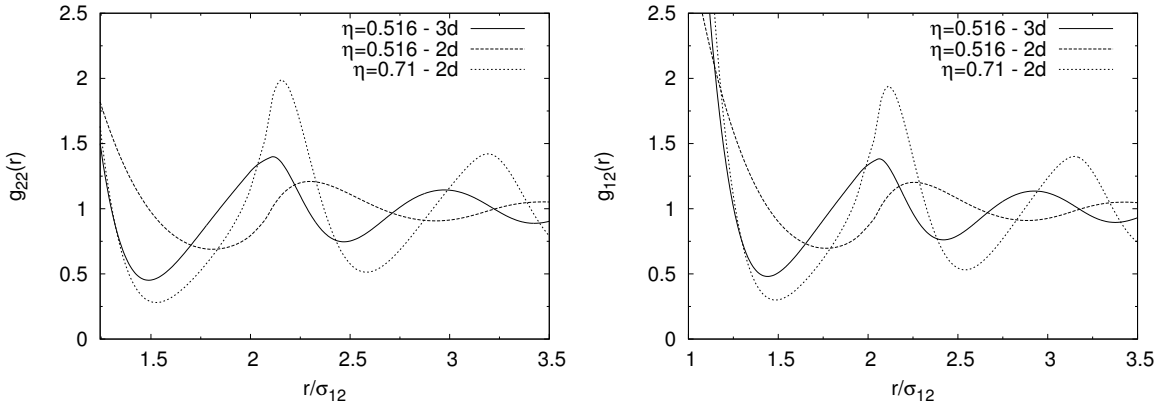

Abbildung 4.15.: $\mathbf{g}(r)$ für binäre harte Teilchen bei $\delta = 0.9$ und $x_1 = 0.2$ - Vergleich zwischen 2d und 3d mit PY bei gleicher Packungsdichte $\eta = 0.516$ und gleichem mittleren Abstand ($\eta = 0.71$)

Betrachten wir also die Paarverteilungsfunktionen in (Abb. 4.14 und Abb. 4.15)[3]. Wie nach den Resultaten der monodispersen harten Teilchen zu erwarten (Abb. 4.4), sind alle Komponenten der zweidimensionalen Paarverteilungsfunktionen langreichweitiger, als die dreidimensionalen bei gleicher Packungsdichte. Ursache dafür ist wieder der in zwei Dimensionen größere mittlere Abstand der Teilchen. Damit ist auch ersichtlich, warum bei gleichen Packungsdichten in zwei Dimensionen weniger Unterstruktur aufgrund der Polydispersität zu erkennen ist, als in drei Dimensionen. Ist der mittlere Abstand der Teilchen in beiden Dimensionen gleich, zeigt sich wie bei dem System monodisperser harter Teilchen (Abb. 4.5), dass die Extremstellen der Korrelatoren besser übereinstimmen. Genau wie dort sind die Extrema der Paarverteilungsfunktion jedoch in zwei Dimensionen höher, was aus den unterschiedlichen Packungseffekten resultiert. Vergleichen wir weiter den Einfluss des Radienverhältnisses auf die Unterstruktur in der Paarverteilungsfunktion, so ergibt sich in beiden Dimensionen die gleiche Tendenz: Die Strukturierung ist besonders stark zu erkennen, wenn $\delta \approx 0.5$ ist. Sowohl im monodispersen Grenzfall (siehe Abb. 4.15 für $\delta = 0.9$), als auch für stark unterschiedliche Radienverhältnisse wird die Strukturierung kleiner (für zwei Dimensionen in Abb. 4.12).

Wir halten also fest, dass für zweidimensionale binäre Systeme harter Teilchen ebenfalls eine Unterstruktur in den Korrelatoren existiert, die jedoch im Vergleich zu drei Dimensionen aufgrund des verschiedenen mittleren Abstandes erst bei höheren Packungsdichten zu Tage tritt.

4.4. Binäre dipolare harte Scheiben

Potential und Parameter Das Potential der binären dipolaren harten Scheiben lautet analog zu den monodispersen dipolaren harten Scheiben:

[3] Dabei sind nur die gemischte Komponente und der Komponente der großen Teilchen dargestellt, da die großen Teilchen das System physikalisch dominieren und daher nur diese Komponenten des dreidimensionales Systems zur Verfügung standen (siehe [GV03]).

$$V_{\alpha\beta}(r) := \begin{cases} \infty & \frac{r}{\sigma_{12}} \leq \frac{\sigma_{\alpha\beta}}{\sigma_{12}} \\ \frac{\mu_0}{4\pi} \frac{\chi_\alpha \chi_\beta B^2}{(\frac{r}{\sigma_{12}})^3 \sigma_{12}^3} & \frac{r}{\sigma_{12}} > \frac{\sigma_{\alpha\beta}}{\sigma_{12}} \end{cases} \tag{4.13}$$

Auch hier sind die Teilcheneigenschaften nach dem Experiment [HKM05] gewählt. Die magnetischen Suszeptibilitäten lauten $\chi_1 := 6.6[\mathrm{p}\frac{\mathrm{Am}^2}{\mathrm{T}}]$ und $\chi_2 := 62[\mathrm{p}\frac{\mathrm{Am}^2}{\mathrm{T}}]$, und die Radien $r_1 := 1.4[\mu\mathrm{m}]$ und $r_2 := 2.3[\mu\mathrm{m}]$. Genau wie für die monodispersen dipolaren harten Scheiben ist der Kontrollparameter des Systems Γ_b, wenn die Packungsdichte klein genug ist. Γ_b ist wieder die mittlere magnetische Wechselwirkungsenergie, die für das binäre System noch mit der mittleren magnetischen Suszeptibilität gewichtet werden muss:

$$\Gamma_\mathrm{b} := \frac{\mu_0}{4\pi} B^2 \frac{(\rho\pi)^{\frac{3}{2}}}{k_B T} \left(x_1\chi_1 + [1-x_1]\chi_2\right)^2 \tag{4.14}$$

Dabei ist $\rho = \frac{N}{F}$ wieder die Gesamtteilchendichte, k_B die Boltzmannkonstante und μ_0 die magnetische Permeabilitätskonstante. Für die Numerik wurden jedoch wieder η (wie zuvor für binäre harte Scheiben definiert) und $T[\mathrm{K}]$ bei festem Magnetfeld $B[\mathrm{mT}]$ vorgegeben. Der Grund dafür ist wieder (siehe Abschnitt 4.2), dass die Äquivalenz der Strukturfaktoren nur für niedrige Packungsdichten gilt. Die Normierung der Orte und Impulse ist wie bei den reinen binären harten Scheiben auf den mittleren Durchmesser $\sigma_{12} = r_1 + r_2$ bezogen.

Wie bereits zuvor erwähnt dominiert bei geringen Packungsdichten die magnetische Wechselwirkung. Da die Packungsdichte des Experimentes [HKM05] sehr gering ist, betrachten wir abkürzend hier nur eine Variation der magnetischen Suszeptibilitäten, denn für die kleinen Packungsdichten ist der Einfluss des Radienverhältnisses auf die Korrelatoren gering. Prinzipiell ändern sich die Korrelatoren der binären dipolaren harten Scheiben unter Variation von δ wie die binären reinen harten Scheiben.

Es sei weiter kurz angemerkt, dass das Konvergenzverhalten von PY und HNC für die binären Systeme genau wie bei dem monodispersen Fall beschrieben ist: Für stark repulsiv dipolare Wechselwirkung ist der Faktor $e^{-\beta \mathbf{V}(r)}$ eine nahezu stetige Funktion vom Ort und die HNC-Approximation konvergiert wesentlich schneller, als die PY-Approximation.

Variation des Suszeptibilitätenverhältnisses Wir diskutieren zunächst den Einfluss von δ_χ auf die Korrelatoren. Dazu wählen wir χ_2 aus dem Experiment [HKM05] und geben δ_χ vor, so dass $\chi_1 = \delta_\chi \chi_2$ folgt. Betrachten wir zunächst die Paarverteilungsfunktion in Abb. 4.16. Wie zu erwarten sind für $\delta_\chi = 1$ alle Komponenten gleich, denn das Radienverhältnis $\delta = 0.4$ ist bei $\eta = 0.02$ vernachlässigbar. Weiter sind in der 22-Komponente die Änderungen mit δ_χ im ersten Maximum geringer, da χ_2 fest vorgegeben wurde und somit nur der indirekte Einfluss der Teilchensorte eins sichtbar wird. Im Gegensatz dazu ändert sich die erste Maximalstelle der 11-Komponente stark, wobei sie sich als Maß für den mittleren Abstand der ersten nächsten Nachbarn mit sinkender Wechselwirkungsstärke zu geringerem $\frac{r}{\sigma_{12}}$ verschiebt.

Betrachten wir wieder die Zahl der nächsten Nachbarn in Abb. 4.2, um die ersten Peaks zu interpretieren. Die Anzahl der nächsten Nachbarn der 11-Komponente wächst mit steigender Wechselwirkungsstärke, da sich die Scheiben dann stärker abstoßen und somit ein Effekt wie bei Erhöhen der Packungsdichte resultiert.

	$\delta_\chi = 0.1$	$\delta_\chi = 0.5$	$\delta_\chi = 1.0$
$z_{2d;11}$	2.51	5.07	6.82
$z_{2d;12}$	5.92	6.63	6.83
$z_{2d;22}$	9.37	7.5	6.84

Tabelle 4.2.: Anzahl nächster Nachbarn für binäre dipolare harte Scheiben bei $\delta = 0.4$, $x_1 = 0.3$, $B = 1[\mathrm{mT}]$, $T = 1.000[\mathrm{K}]$ und $\eta = 0.02$ - Variation des Suszeptibilitätenverhältnisses δ_χ bei gegebenem $\chi_2 = 62[\mathrm{p}\frac{\mathrm{Am}^2}{\mathrm{T}}]$; Γ_b steigt mit δ_χ und die Werte lauten $\Gamma_\mathrm{b} = \{40; 54; 75\}$

Wie bereits an der schwächeren Änderung der Anzahl nächster Nachbarn in der 22-Komponente erkennbar, basiert sie auf einem anderen Effekt. Mit gutmütigem Blick lässt sich in Abb. 4.12 erkennen, dass der Einfluss der Änderung von δ_χ immer kleiner wird. Die Ursache liegt also in der impliziten Wechselwirkung der verschiedenen Teilchensorten. Anders als die 11-Komponente kann die 22-Komponente nicht verschwinden, da $x_2 = 1 - x_1$ und χ_2 fest vorgegeben sind.

Betrachten wir nochmals die Paarverteilungsfunktion in Abb. 4.16 im Vergleich zu den binären reinen harten Scheiben (Abb. 4.12). Es zeigt sich, dass die Strukturierung aufgrund der Polydispersität der dipolaren harten Scheiben wesentlich geringer ist. Nebenmaxima bei einem Mischungverhältnis $\delta_\chi = 0.5$ wie bei den harten Scheiben bei $\delta = 0.5$ treten gar nicht auf. Die Ursache dafür muss in der Langreichweitigkeit der dipolaren Wechselwirkung liegen.

Die Änderungen der zugehörigen Strukturfaktoren in Abb. 4.17 sind konsistent mit den Abhängigkeiten der Paarverteilungsfunktion. Die Argumentationen übertragen sich analog zu den zuvor besprochenen Systemen auf den Strukturfaktor. Vergleichen wir die strukturellen Unterschiede der 11- und der 22-Komponente mit denen der 11- und der 22-Komponenten reiner binärer harter Scheiben (Abb. 4.13), ist ersichtlich, dass sich die verschiedenen Längenskalen der Teilchensorten ähnlich äußern. Die 22-Komponente der größeren bzw. magnetisch stärker wechselwirkenden Teilchen ähnelt in ihrem Verlauf eher dem monodispersen Fall (Abb. 4.7), als die 11-Komponente, denn diese Teilchen dominieren das System.

Korrelatoren Vergleich zum Experiment Betrachten wir zum Abschluss dieses Kapitels die statischen Korrelatoren des Experimentes [HKM05]. Da die Näherung Γ_b in diesem Packungsdichtebereich als Systemparameter zu verwenden noch nicht gut ist (siehe Abb. 4.6), werden wir hier Γ_b immer zusammen mit einer Temperatur angeben. Aus dem Experiment wählen wir exemplarisch einen Datensatz zu $\Gamma_\mathrm{b}^\mathrm{EXP} = 78$ und $T = 292[K]$. Die numerischen Lösungen aus PY wurden bei $\Gamma_\mathrm{b}^\mathrm{PY} = 11.43$ und $T = 2.000[\mathrm{K}]$ bzw. $\Gamma_\mathrm{b}^\mathrm{PY} = 5.71$ und $T = 4.000[\mathrm{K}]$ gewonnen.

Sehr qualitativ gesehen ergibt sich eine Übereinstimmung der Resultate. So stimmt die relative Anordnung der Matrixkomponenten untereinander in Position und Höhe grob überein. Auch die Reichweite dieser Korrelatoren ist vergleichbar. Jedoch sind die experimentellen Maxima wesentlich höher als die numerischen. Dies ist auch bereits der zentrale Unterschied, denn mit höheren numerischen Maxima ist auch die Reichweite der Korrelatoren wesentlich länger. Weiter wird die experimentell beobachtete Unterstruktur aufgrund der Polydispersität numerisch nicht aus der dipolaren Wechselwirkung reproduziert. Das heißt aber, dass die reine (in sich konsistente) dipolare Wechselwirkung das Experiment nicht ausreichend beschreibt. Diese Tatsache ist jedoch auch nicht weiter verwunderlich, wenn wir beachten, dass das Potential, über das die Teilchen wechselwirken nur dann die rein dipolare Form (4.13) hat, wenn die magnetischen Momente parallel zueinander und die Teilchen alle in einer Ebene sind. Falls nicht alle Teilchen exakt in einer Ebene sind, ergeben sich sofort Wechselwirkungsterme, die das

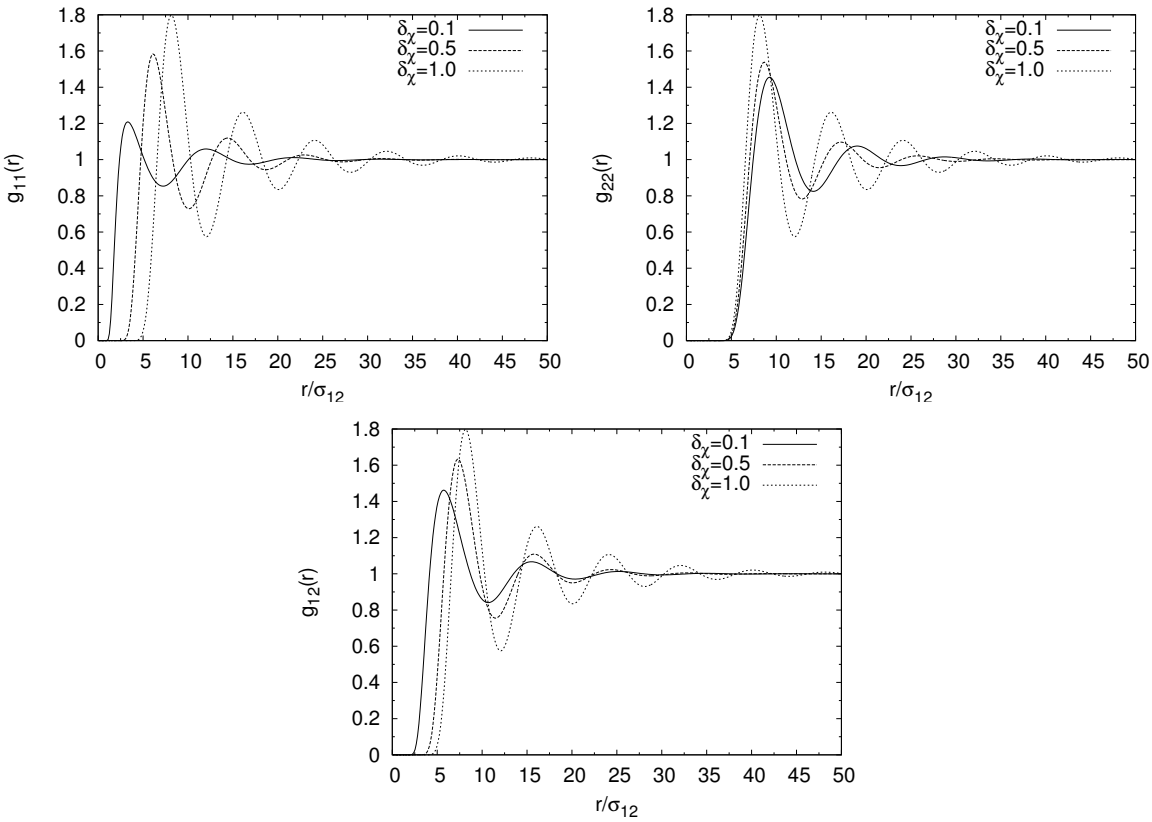

Abbildung 4.16.: $\mathbf{g}(r)$ für binäre dipolare harte Scheiben bei $\eta = 0.02$, $x_1 = 0.3$, $\delta = 0.4$, $B = 1[\mathrm{mT}]$, $T = 1.000[\mathrm{K}]$ - Variation des Suszeptibilitätenverhältnisses δ_χ bei gegebenen $\chi_2 = 62[\mathrm{p}\frac{\mathrm{Am}^2}{\mathrm{T}}]$; Γ_b steigt mit δ_χ und die Werte lauten $\Gamma_\mathrm{b} = \{40; 54; 75\}$

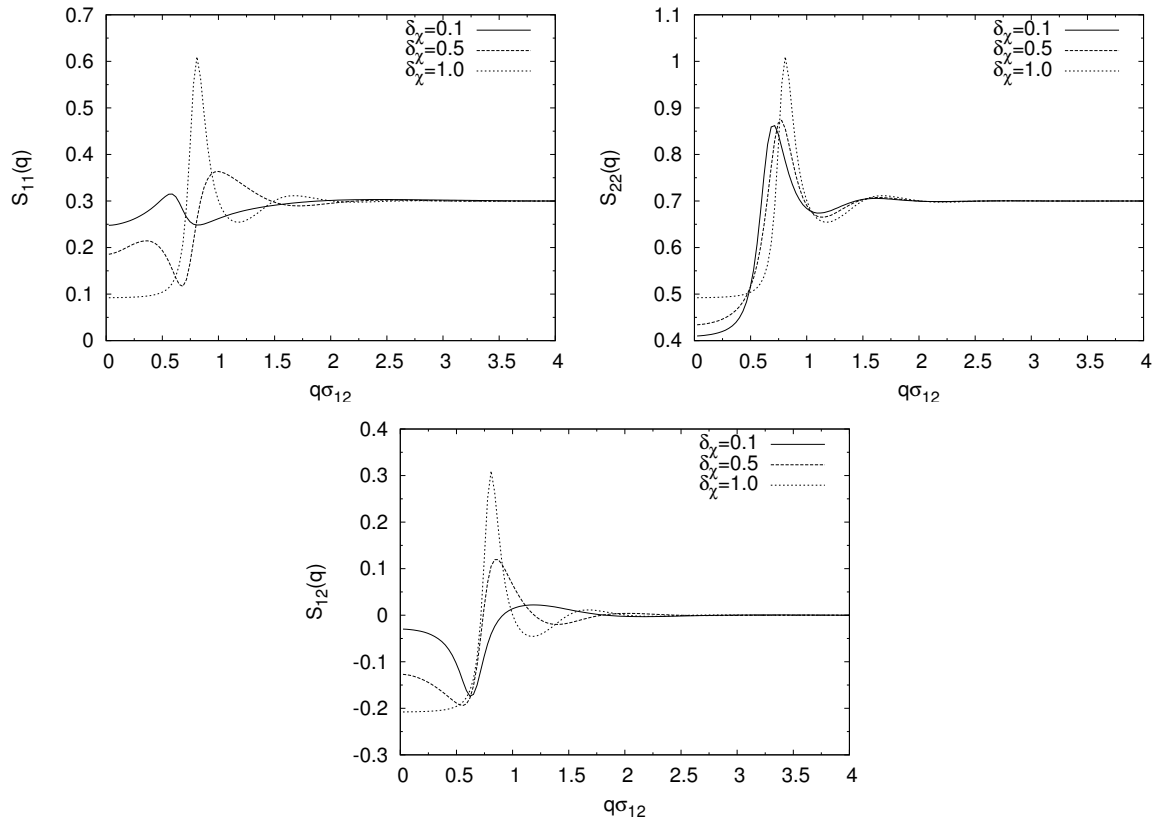

Abbildung 4.17.: $\mathbf{S}(q)$ für binäre dipolare harte Scheiben bei $\eta = 0.02$, $x_1 = 0.3$, $\delta = 0.4$, $B = 1[\mathrm{mT}]$, $T = 1.000[\mathrm{K}]$ - Variation des Suszeptibilitätenverhätnisses δ_χ bei gegebenen $\chi_2 = 62[\mathrm{p}\frac{\mathrm{Am}^2}{\mathrm{T}}]$; Γ_b steigt mit δ_χ und die Werte lauten $\Gamma_\mathrm{b} = \{40; 54; 75\}$

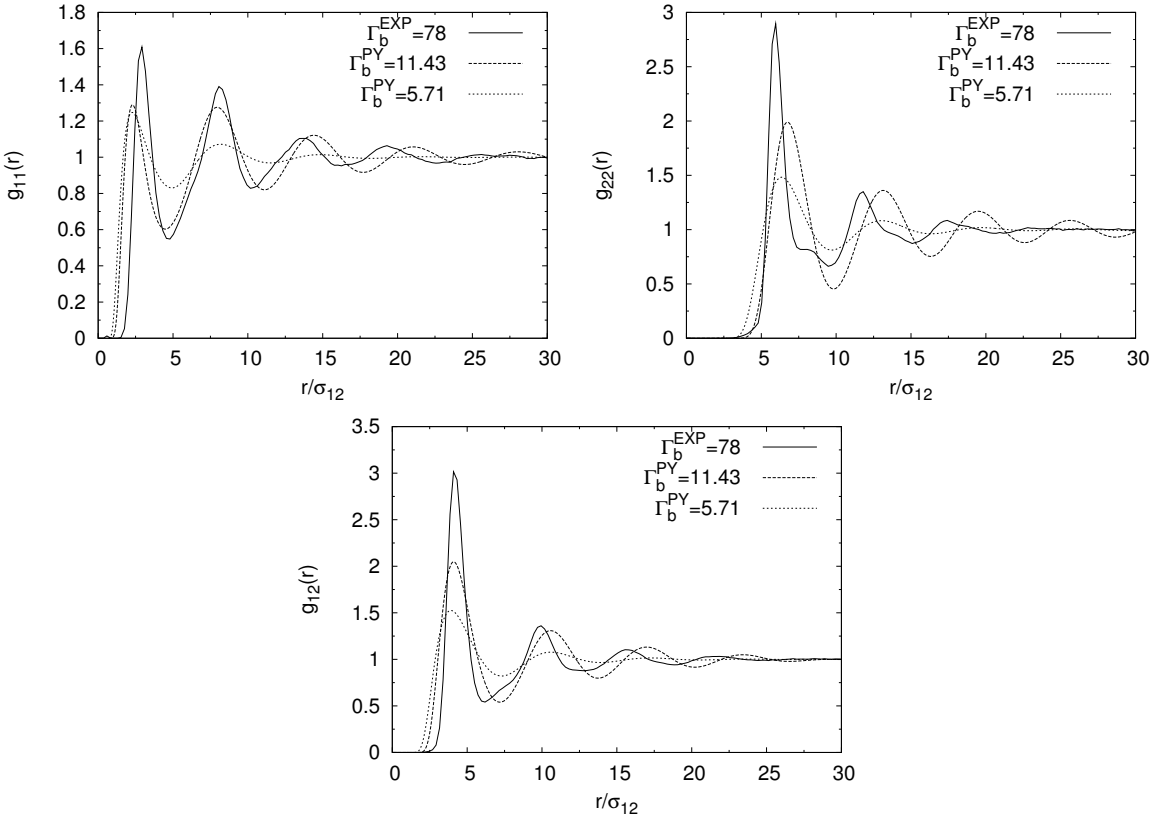

Abbildung 4.18.: $\mathbf{g}(r)$ für binäre harte Scheiben aus dem Experiment [HKM05] und aus PY im Vergleich; $B = 1.47[\mathrm{mT}]$, $\eta = 0.0413$, $x_1 = 0.508$ bei $\Gamma_b^{\mathrm{EXP}} = 78$ (d.h. $T = 292.941[\mathrm{K}]$) und bei $\Gamma_b^{\mathrm{PY}} = 11.4$ (d.h. $T = 2.000[\mathrm{K}]$) und $\Gamma_b^{\mathrm{PY}} = 5.7$ (d.h. $T = 4.000[\mathrm{K}]$)

Potential verringern. Das ist bereits am magnetischen Feld eines Teilchens (4.5) abzulesen. Außerdem kann bei starker Packung $\eta > 1$ werden, da die verwendete Packungsdichte in zwei Dimensionen definiert ist. Dass genau dieser Punkt des Haltens der Teilchen in einer Ebene experimentelle Probleme verursacht ist bestätigt [Fuc06].

Prinzipiell stellt das aber keine Einschränkung an den zu untersuchenden Glasübergang der binären dipolaren Systeme dar. Der monodisperse Glasübergang liegt je nach Packungsdichte bei $\Gamma_m \approx 2..5$ (siehe Abschnitt 7.2). Da das System der binären dipolaren harten Scheiben von den Teilchen mit der größeren magnetischen Suszeptibilität majorisiert wird, ist die Erwartung, dass der Glasübergang der binären dipolaren harten Scheiben ebenfalls in diesem Bereich liegt: $\Gamma_\mathrm{b} \approx \Gamma_\mathrm{m}$.

5. Modenkopplungstheorie

Ziel dieses Kapitels ist das Aufstellen eines Gleichungssystems für den dynamischen Strukturfaktor $\mathbf{S}(q;t)$. Wir werden in diesem Kapitel die Modenkopplungsapproximation für binäre Systeme in zwei Dimensionen formulieren, um schließlich ein geschlossenes Gleichungssystem für den Nichtergodizitätsparameter $\mathbf{F}(q)$ (bzw. monodispers $f(q)$) zu erhalten[1]. Wie bereits in Kapitel 2 beziehen wir uns direkt auf binäre zweidimensionale Systeme. Die relevanten grundlegenden Begriffe der zeitabhängigen Korrelatoren führen wir in Abschnitt 5.1 ein. Weiter stellen wir in Abschnitt 5.2 den Projektorformalismus von Mori und Zwanzig vor, mit dem auf die langsamen Bewegungen eines Systems projeziert werden kann. Mit seiner Hilfe kann ein exaktes Gleichungssystem für den dynamischen Strukturfaktor (5.15) aufgestellt werden. In diesem Gleichungssystem steckt die Dynamik des Systems allein im Relaxationskern (5.32) (bzw. monodispers (5.29)), der dann in Abschnitt 5.3 mittels der Modenkopplungsapproximation genähert werden kann. Schließlich wird in Abschnitt 5.5 der thermodynamische Limes gebildet und ein geschlossenes Gleichungssystem für den zeitabhängigen Strukturfaktor präsentiert. Abschließend geben wir in Abschnitt 5.6 ein Gleichungssystem für den Nichtergodizitätsparameter an. Einige allgemeine Aussagen der Modenkopplungstheorie (MCT) werden wir aphoristisch in Abschnitt 5.4 angeben. Wie bereits in Kapitel 2 formulieren wir die Aussagen zunächst ohne auf spezielle Einheiten Bezug zu nehmen. Damit ist insbesondere $\rho := \frac{N}{F}$ im folgenden eine Teilchendichte.

5.1. Zeitabhängige Korrelatoren

Nachdem wir in Kapitel 2 ein geschlossenes Gleichungssystem für den statischen Strukturfaktor $\mathbf{S}(q)$ angegeben haben, das alleine durch das Potential $\mathbf{V}(r)$ bestimmt ist, werden wir nun zum dynamischen d.h. zeitabhängigen Strukturfaktor $\mathbf{S}(q;t)$ übergehen. Dazu betrachten wir zunächst beliebige zeitabhängige Korrelatoren, die über ein Skalarprodukt der Form $\langle a^*|b\rangle$ definiert sind. Sei $A_\alpha(t)$ mit $\alpha \in \{1,2\}$ eine beliebige dynamische Variable[2] und $A^*_\alpha(t)$ die zu ihr komplex konjugierte dynamische Variable. Ganz allgemein ist die Korrelationsmatrix einer dynamischen Variable $A_\alpha(t)$ dann definiert als

$$C_{\alpha\beta}(t) := \left\langle A^*_\alpha(t)|A_\beta(t=0)\right\rangle \tag{5.1}$$

Da wir uns im Geltungsbereich der klassischen Mechanik befinden wird eine beliebige dynamische Variable $A_\alpha(t)$ durch den Liouville-Operator $\mathcal{L}$ propagiert, der über die Poisson-Klammer der dynamischen Variable und der Hamiltonfunktion H definiert ist als

[1] Obwohl der Nichtergodizitätsparameter im Impulsraum definiert ist, bezeichnen wir den monodispersen normierten NEP mit $f(q)$ und den binären nicht normierten NEP mit $\mathbf{F}(q)$ - siehe (5.42).

[2] Die folgenden Aussagen lassen sich auch allgemein mit $\alpha = 1...m$ formulieren. Wir wählen jedoch direkt die Formulierung für binäre Systeme.

$$\frac{d}{dt}A_\alpha(t) = \{A_\alpha(t), H\} =: i\mathcal{L}A_\alpha(t) \tag{5.2}$$

Damit gilt dann formal $A_\alpha(t) = e^{i\mathcal{L}t}A_\alpha(t=0)$ und die Korrelationsmatrix kann mit dem Liouville-Operator umgeschrieben werden zu

$$C_{\alpha\beta}(t) = \left\langle A_\alpha^* | e^{-i\mathcal{L}t} | A_\beta \right\rangle \tag{5.3}$$

Abkürzend bezeichne $A_\alpha := A_\alpha(t=0)$. Mit Hilfe der Laplace-Transformation

$$\hat{C}_{\alpha\beta}(z) = \int_0^\infty dt e^{izt} C_{\alpha\beta}(t) \qquad \Im(z) > 0 \tag{5.4}$$

kann die Korrelationsmatrix in den Bildbereich transformiert werden und lautet dann:

$$\hat{C}_{\alpha\beta}(z) = i\left\langle A_\alpha^* | \left(z - \mathcal{L}\right)^{-1} | A_\beta \right\rangle \tag{5.5}$$

Nun soll der Anschluss dieser allgemeingültigen Aussagen an die statistische Mechanik vollzogen werden. Dazu wird wie in [Göt89] die thermodynamische Gleichgewichtsmittelung $\langle\cdot\rangle_{\text{td}}$ als Skalarprodukt über dem Hilbertraum der thermodynamischen Observablen interpretiert, denn die thermodynamische Mittelung erfüllt die drei Skalarprodukteigenschaften, die aus elementarer Rechnung mit der Definition der thermodynamischen Mittelung folgen. Kurzum:

$$\left\langle A_\alpha^* A_\beta \right\rangle_{\text{td}} = \left\langle A_\alpha^* | A_\beta \right\rangle$$

5.2. Projektionsformalismus von Mori und Zwanzig

Wir werden in diesem Abschnitt zuerst mittels eines Projektionsformalismus die Mori-Zwanzig-Gleichung herleiten. Sie ermöglicht es uns, auf die langsamen Variablen eines Systems zu projezieren und damit die Berechnung der Korrelationsmatrix zu vereinfachen. Da unser Interesse in der Kenntnis des zeitabhängigen Strukturfaktors besteht werden wir auf ihn den Projektorformalismus dann anwenden.

Projektionsformalismus von Mori und Zwanzig Gesucht ist im folgenden eine Möglichkeit, die Berechnung der Korrelationsmatrix (5.5) zu vereinfachen. Dies kann mit einer Basisentwicklung von $\mathcal{L}$ bezüglich des Projektors $\mathcal{P}$ auf die Variablen A_α erreicht werden. Dazu gehen wir analog zu [For83] vor und definieren zunächst den erwähnten Projektor $\mathcal{P}$ auf den Variablensatz A_α

$$\mathcal{P} := \sum_{\rho,\sigma=1}^{2} \left|A_\rho\right\rangle \left(\left(\left\langle A_\alpha^* | A_\beta \right\rangle \right)^{-1} \right)_{\rho\sigma} \left\langle A_\sigma^* \right| \tag{5.6}$$

Sein orthogonales Komplement ist dann $\mathcal{Q} := 1 - \mathcal{P}$. Die beiden erfüllen die Operatoridentitäten $\mathcal{Q}^2 = \mathcal{Q}$, $\mathcal{P}^2 = \mathcal{P}$ und $\mathcal{Q}\mathcal{P} = \mathcal{P}\mathcal{Q} = 0$. Die zentrale Idee ist nun, den Liouville-Operator mittels des

Projektors (5.6) zu vereinfachen. Da die Vollständigkeit also erfüllt ist, kann mit $\mathcal{P}$ eine Basisentwicklung vorgenommen werden:

$$\mathcal{L} = \mathcal{L}\mathcal{P} + \mathcal{L}\mathcal{Q}$$

und die Korrelationsmatrix lautet dann

$$\hat{C}_{\alpha\beta}(z) = i\left\langle A_\alpha^* | [z - \mathcal{L}\mathcal{P} - \mathcal{L}\mathcal{Q}]^{-1} | A_\beta \right\rangle$$

Nun kann die Operatoridentität

$$(X+Y)^{-1} = X^{-1} - Y(X[X+Y])^{-1}$$

mit $X := i\mathcal{L}\mathcal{Q} - iz$ und $Y := i\mathcal{L}\mathcal{P}$ verwendet werden, um die Inverse in der Korrelationsmatrix zu entwickeln. Dann kann weiter in dieser entwickelten Inversen nach den auftretenden Termen $\mathcal{L}\mathcal{Q}$ formal in eine geometrische Reihe entwickelt werden. Damit ergibt sich dann eine Form für die Korrelationsmatrix, in der Potenzen von $\mathcal{L}\mathcal{Q}$ auftreten, die nicht mehr invertiert werden müssen. Da die Korrelationsmatrix für den Variablensatz A_α berechnet werden soll, auf den $\mathcal{P}$ projeziert und $\mathcal{Q}$ orthogonal zu $\mathcal{P}$ ist, können die Terme mit $\mathcal{L}\mathcal{Q}$ vernachlässigt werden. Es ergibt sich insgesamt für die Korrelationsmatrix die Mori-Zwanzig-Gleichung

$$\hat{C}_{\alpha\beta}(z) = -\sum_{\rho=1}^{2} \left(\left[z\delta_{\alpha\beta} + \Omega_{\alpha\beta} + \hat{\Sigma}_{\alpha\beta}(z) \right]^{-1} \right)_{\alpha\rho} \left\langle A_\rho^* | A_\beta \right\rangle \tag{5.7}$$

mit

$$\begin{aligned} \Omega_{\alpha\beta} &:= i\sum_{\rho=1}^{2} \left\langle \dot{A}_\alpha^* | A_\rho \right\rangle \left(\left(\left\langle A_\alpha^* | A_\beta \right\rangle \right)^{-1} \right)_{\rho\beta} \\ \hat{\Sigma}_{\alpha\beta}(z) &:= i\sum_{\rho=1}^{2} \left\langle \dot{A}_\alpha^* | \mathcal{Q} [z - \mathcal{Q}\mathcal{L}\mathcal{Q}]^{-1} \mathcal{Q} | \dot{A}_\rho \right\rangle \left(\left(\left\langle A_\alpha^* | A_\beta \right\rangle \right)^{-1} \right)_{\rho\beta} \end{aligned}$$

In $\hat{\Sigma}_{\alpha\beta}(z)$ taucht noch der zu $\mathcal{P}$ orthogonale Projektor $\mathcal{Q}$ auf. Dieser Anteil lässt sich jedoch nicht mehr mit obiger Argumentation weiter vereinfachen, da der Liouville-Operator $\mathcal{L}' := \mathcal{Q}\mathcal{L}\mathcal{Q}$ jetzt auf die dynamischen Variablen $\mathcal{Q}\dot{A}_\alpha := \mathcal{Q}\frac{dA_\alpha}{dt}$ wirkt und die beiden damit nicht orthogonal zueinander sind. Physikalisch beinhaltet der Projektor $\mathcal{Q}$ alle Freiheitsgrade, die nicht in den Variablen A_α stecken und beschreibt damit anschaulich ein Teilchenbad, mit dem die dynamische Variable A_α wechselwirkt.

Wegen der Zeitumkehrinvarianz der thermodynamischen Mittelung gilt allgemein $\Omega_{\alpha\alpha} \equiv 0$, denn angenommen A_α ist gerade (ungerade) in t, so ist $\dot{A}_\alpha$ ungerade (gerade) in t.

Projektionsformalismus angewandt auf den Strukturfaktor Der Strukturfaktor $S_{\alpha\beta}(q)$ wurde bereits in (2.4) definiert. Betrachten wir wieder den Fall $q \neq 0$ und bezeichnen wieder $A_\alpha(t=0) =: A_\alpha$. Dann lautet der zeitabhängige Strukturfaktor

$$\hat{S}_{\alpha\beta}(q;z) = \frac{1}{N} \left\langle \rho_\alpha^*(\vec{q}) | [\mathcal{L} - z]^{-1} | \rho_\beta(\vec{q}) \right\rangle \tag{5.8}$$

Weiter definieren wir noch die zeitabhängige *fouriertransformierte Teilchendichte*

$$\rho_\alpha(\vec{q};t) := \sum_{n=1}^{N_\alpha} e^{i\vec{q}\cdot\vec{x}_n^{(\alpha)}(t)} \tag{5.9}$$

und den *Teilchenstrom*

$$\vec{j}_\alpha(\vec{q};t) := \sum_{n=1}^{N_\alpha} \dot{\vec{x}}_n^{(\alpha)}(t) e^{i\vec{q}\cdot\vec{x}_n^{(\alpha)}(t)} \tag{5.10}$$

Weiter kann dann der *longitudinale Teilchenstrom* definiert werden:

$$j_\alpha(\vec{q};t) := \frac{\vec{q}}{q}\cdot\vec{j}_\alpha(\vec{q};t) \tag{5.11}$$

Gesucht ist nun vor allem der Strukturfaktor zu einer beliebigen Zeit t, die groß gegen die mittlere Stoßzeit der Teilchen ist. Das heißt, dass nicht die mikroskopischen schnellen Bewegungen der Teilchen, sondern die Entwicklung der langsamen Variablen gesucht ist. Damit ist also motiviert, den Projektorformalismus von Mori und Zwanzig auf den Strukturfaktor anzuwenden. Die langsamen Variablen sind dabei Teilchendichte (5.9) und der longitudinale Teilchenstrom (5.11).

Führen wir also zunächst den Projektor auf die Teilchendichten zum Zeitpunkt $t = 0$ ein:

$$\mathcal{P}_\rho := \sum_{\delta,\sigma=1}^{2} |\rho_\delta(\vec{q})\rangle \left(\left(\langle \rho_\alpha^*(\vec{q}) | \rho_\beta(\vec{q}) \rangle \right)^{-1} \right)_{\delta\sigma} \langle \rho_\sigma^*(\vec{q})| \tag{5.12}$$

Damit folgt nach (5.7) für den dynamischen Strukturfaktor zunächst:

$$\begin{aligned}
\hat{S}_{\alpha\beta}(q;z) &= -\sum_{\sigma=1}^{2} \left(\left[z\delta_{\alpha\beta} + \Omega_{\alpha\beta} + \hat{\Sigma}_{\alpha\beta}(z) \right] \right)^{-1}_{\alpha\sigma} \langle \rho_\sigma^*(\vec{q}) | \rho_\beta(\vec{q}) \rangle \\
\Omega_{\alpha\beta} &:= i \sum_{\sigma=1}^{2} \langle \dot{\rho}_\alpha^*(\vec{q}) | \rho_\sigma(\vec{q}) \rangle \left(\left(\langle \rho_\alpha^*(\vec{q}) | \rho_\beta(\vec{q}) \rangle \right)^{-1} \right)_{\sigma\beta} \\
\hat{\Sigma}_{\alpha\beta}(z) &:= i \sum_{\sigma=1}^{2} \langle \dot{\rho}_\alpha^*(\vec{q}) | Q_\rho [z - Q_\rho \mathcal{L} Q_\rho]^{-1} Q_\rho | \dot{\rho}_\sigma(\vec{q}) \rangle \left(\left(\langle \rho_\alpha^*(\vec{q}) | \rho_\beta(\vec{q}) \rangle \right)^{-1} \right)_{\sigma\beta}
\end{aligned}$$

Wir führen weiter den Projektor $\mathcal{P}_j$ auf die Teilchenströme zum Zeitpunkt $t = 0$ ein, da $\hat{\Sigma}_{\alpha\beta}(z)$ noch die langsame Variable der longitudinalen Teilchenströme $j_\alpha(\vec{q})$ beinhaltet:

$$\mathcal{P}_j := \sum_{\delta,\sigma=1}^{2} |j_\delta(\vec{q})\rangle \left(\left(\langle j_\alpha^*(\vec{q}) | j_\beta(\vec{q}) \rangle \right)^{-1} \right)_{\delta\sigma} \langle j_\sigma^*(\vec{q})| \tag{5.13}$$

Mit diesem Projektor kann $\hat{\Sigma}_{\alpha\beta}(z)$ mittels der Mori-Zwanzig-Gleichung (5.7) entwickelt werden. Definieren wir analog zum dynamischen Strukturfaktor

$$J_{\alpha\beta}(q) := \frac{m}{Nk_BT} \left\langle j^*_\alpha(\vec{q}) | j_\beta(\vec{q}) \right\rangle \tag{5.14}$$

so folgt insgesamt in Matrixschreibweise für den dynamischen Strukturfaktor:

$$\hat{\mathbf{S}}(q;z) = -[z\mathbb{1} - [z\mathbb{1} + \hat{\mathbf{M}}(q;z)\mathbf{J}^{-1}(q)]^{-1}\mathbf{J}^{-1}(q)\mathbf{S}^{-1}(q)]^{-1}\mathbf{S}(q) \tag{5.15}$$

mit dem *Relaxationskern*

$$\hat{M}_{\alpha\beta}(q;z) := i \left\langle \frac{dj^*_\alpha(\vec{q})}{dt} | \mathcal{Q}\left[z - \mathcal{Q}\mathcal{L}\mathcal{Q}\right]^{-1}\mathcal{Q} | \frac{dj_\beta(\vec{q})}{dt} \right\rangle \tag{5.16}$$

Dabei ist

$$\mathcal{Q} := \mathcal{Q}_\rho \mathcal{Q}_j \tag{5.17}$$

Die Dynamik des Systems steckt nun also nur noch im Relaxationskern $\hat{\mathbf{M}}$, der im folgenden Abschnitt mittels der Modenkopplungsapproximation genähert werden soll.

5.3. Modenkopplungsfunktional

In diesem Abschnitt werden wir beschreiben, wie der Relaxationskern (5.16) genähert werden kann. Die Rechnungen werden wir wieder für ein binäres zweidimensionales System formulieren, das Ergebnis aber auch für monodisperse zweidimensionale Systeme angeben.

Zusammenhang zwischen $\frac{d}{dt} j_\alpha(\vec{q};t) \leftrightarrow \frac{d^2}{dt^2}\rho_\alpha(\vec{q};t) \leftrightarrow \rho(\vec{q}_1;t)\rho(\vec{q}_2;t)$ Um die MCT zu verstehen, ist es zunächst nötig, die Zustände im Relaxationskern (5.16) zu betrachten. Für die Teilchenströme gilt die Kontinuitätsgleichung

$$\dot{\rho}_\alpha(\vec{q};t) = i\vec{q} \cdot \vec{j}_\alpha(\vec{q};t) \tag{5.18}$$

Damit gilt für die longitudinalen Teilchenströme insbesondere: $\dot{\rho}(\vec{q};t) = iq j_\alpha(\vec{q};t)$. Und damit folgt:

$$\frac{d}{dt} j_\alpha(\vec{q};t) = \frac{1}{iq}\frac{d^2}{dt^2}\rho_\alpha(\vec{q};t) \tag{5.19}$$

Die zweite Zeitableitung der fouriertransformierten Teilchendichte lässt sich elementar berechnen und zerfällt in einen kinetischen (kin.) und einen kräfteabhängigen Anteil:

$$\ddot{\rho}_\alpha(\vec{q};t) = \sum_{n=1}^{N_\alpha} \left(\underbrace{\frac{i\vec{q}}{m} \cdot \dot{\vec{p}}_n^{(\alpha)}(t)}_{=:A} - \text{kin.} \right) e^{i\vec{q}\cdot\vec{x}_n^{(\alpha)}(t)} \tag{5.20}$$

Nun kann $\dot{\vec{p}}_n^{(\alpha)}$ mit einer mikroskopischen geschwindigkeitsunabhängigen Kraft identifiziert werden. Damit existiert aber zu dieser Kraft ein Potential und es gilt $\dot{\vec{p}}_n^{(\alpha)} = -\frac{\partial}{\partial \vec{x}_n^{(\alpha)}} V(\vec{x}_1^{(\alpha)}, ..., \vec{x}_{N_\alpha}^{(\alpha)})$. Angenommen es handelt sich dabei um ein Paarpotential, so kann der Term A in (5.20) mit dem fouriertransformierten Paarpotential $\tilde{V}(\vec{q})$ umgeschrieben werden zu

$$A = \frac{1}{2F}\vec{q} \cdot \sum_{\vec{q}_1+\vec{q}_2=\vec{q}} \left(\tilde{V}(\vec{q}_1)\vec{q}_1 + \tilde{V}(\vec{q}_2)\vec{q}_2\right) \rho(\vec{q}_1;t)\rho(\vec{q}_2;t)$$

Das heißt also, dass der die Langzeitdynamik bestimmende Anteil der Zustände im Relaxationskern über miteinander gekoppelte Dichten (gekoppelte Moden) dargestellt werden kann. Eine tiefergehende Darstellung findet sich in [Sch94].

Projektor auf gekoppelte Moden Mit der Kontinuitätsgleichung (5.19) wird der Relaxationskern im Zeitbereich zunächst umgeformt zu

$$M_{\alpha\beta}(q;t) = \frac{1}{q^2}\left\langle \ddot{\rho}^*_\alpha(\vec{q}) | \mathcal{Q} e^{-i\mathcal{Q}\mathcal{L}\mathcal{Q}t}\mathcal{Q} | \ddot{\rho}_\beta(\vec{q})\right\rangle$$

Der Projektor $\mathcal{P}$ auf die gekoppelten Moden lautet

$$\begin{aligned}\mathcal{P} \quad := \quad & \sum_{\alpha,\beta,\alpha',\beta'=1}^{2} \sum_{\vec{q}_1,\vec{q}_2,\vec{q}_1',\vec{q}_2'} \left|\rho_\alpha(\vec{q}_1)\rho_\beta(\vec{q}_2)\right\rangle \times \\ & \times \left(\left(\left\langle \rho^*_\alpha(\vec{q}_1)\rho^*_\beta(\vec{q}_2) | \rho_{\alpha'}(\vec{q}_1')\rho_{\beta'}(\vec{q}_2')\right\rangle\right)^{-1}\right)_{\alpha\beta,\alpha'\beta'} \left\langle \rho^*_{\alpha'}(\vec{q}_1')\rho^*_{\beta'}(\vec{q}_2')\right|\end{aligned}$$

Dabei ist $\left(\left(\left\langle \rho^*_\alpha(\vec{q}_1)\rho^*_\beta(\vec{q}_2) | \rho_{\alpha'}(\vec{q}_1')\rho_{\beta'}(\vec{q}_2')\right\rangle\right)^{-1}\right)_{\alpha\beta,\alpha'\beta'}$ eine noch zu bestimmende Normierung. Die erste Näherung ist nun mittels $\mathcal{P}$ auf die gekoppelten Moden zu projezieren

$$\begin{aligned} M_{\alpha\beta}(q;t) \quad \approx \quad & \frac{1}{q^2}\left\langle \ddot{\rho}^*_\alpha(\vec{q}) | \mathcal{Q}\mathcal{P} e^{-i\mathcal{Q}\mathcal{L}\mathcal{Q}t}\mathcal{P}\mathcal{Q} | \ddot{\rho}_\beta(\vec{q})\right\rangle \\ = \quad & \frac{1}{q^2} \sum_{\zeta,\xi,\zeta',\xi'=1}^{2} \sum_{\rho,\sigma,\rho',\sigma'=1}^{2} \sum_{\vec{q}_1,\vec{q}_2,\vec{q}_1',\vec{q}_2'} \sum_{\vec{q}_3,\vec{q}_4,\vec{q}_3',\vec{q}_4'} \left\langle \ddot{\rho}^*_\alpha(\vec{q}) | \mathcal{Q} | \rho_\zeta(\vec{q}_1)\rho_\xi(\vec{q}_2)\right\rangle \times \\ & \times \left(\left(\left\langle \rho^*_\alpha(\vec{q}_1)\rho^*_\beta(\vec{q}_2) | \rho_{\alpha'}(\vec{q}_1')\rho_{\beta'}(\vec{q}_2')\right\rangle\right)^{-1}\right)_{\zeta\xi,\zeta'\xi'} \times \\ & \times \left\langle \rho^*_{\zeta'}(\vec{q}_1')\rho^*_{\xi'}(\vec{q}_2') | e^{-i\mathcal{Q}\mathcal{L}\mathcal{Q}t} | \rho_\rho(\vec{q}_3)\rho_\sigma(\vec{q}_4)\right\rangle \times \\ & \times \left(\left(\left\langle \rho^*_\alpha(\vec{q}_3)\rho^*_\beta(\vec{q}_4) | \rho_{\alpha'}(\vec{q}_3')\rho_{\beta'}(\vec{q}_4')\right\rangle\right)^{-1}\right)_{\rho\sigma,\rho'\sigma'} \times \\ & \times \left\langle \rho^*_{\rho'}(\vec{q}_3')\rho^*_{\sigma'}(\vec{q}_4') | \mathcal{Q} | \ddot{\rho}_\beta(\vec{q})\right\rangle \end{aligned} \tag{5.21}$$

Diese Terme müssen nun einzeln betrachtet werden. Da die Rechnung analog zum monodispersen dreidimensionalen Fall ist, verweisen wir auf [Göt89, Sch94] und kürzen auf die wichtigsten Ideen ab.

Faktorisierung der Dichten Betrachten wir zuerst den verbliebenen zeitabhängigen Term in (5.21). Die in ihm enthaltenen Zustände sind gerade die gekoppelten Dichte-Moden. Die zweite Näherung besteht nun darin, zu fordern, dass diese Korrelationsfunktion faktorisiert; diese Näherung ist bis jetzt jedoch noch nicht abgeschätzt worden. Es ergibt sich dann für diesen Term

$$\begin{aligned}\left\langle \rho^*_{\zeta'}(\vec{q}_1')\rho^*_{\xi'}(\vec{q}_2')|e^{-i\mathcal{Q}\mathcal{L}\mathcal{Q}t}|\rho_\rho(\vec{q}_3)\rho_\sigma(\vec{q}_4)\right\rangle &\approx N^2 S_{\zeta'\rho}(\vec{q}_1';t)S_{\xi'\rho}(\vec{q}_2';t)[\delta_{\vec{q}_1',\vec{q}_3}\delta_{\vec{q}_2',\vec{q}_4}\delta_{\zeta'\rho}\delta_{\xi'\rho}]+\\ &\quad +N^2 S_{\zeta'\sigma}(\vec{q}_1';t)S_{\xi'\sigma}(\vec{q}_2';t)[\delta_{\vec{q}_1',\vec{q}_4}\delta_{\vec{q}_2',\vec{q}_3}\delta_{\zeta'\sigma}\delta_{\xi'\sigma}]\end{aligned}$$

Normierung des Projektors Mit obiger Approximation lässt sich die Normierung von $\mathcal{P}$ nun aus der Projektoridentität $\mathcal{P}^2=\mathcal{P}$ approximativ berechnen, denn für $t=0$ stimmen die beiden Terme überein. Weiter fügen wir einen Faktor 8 ein, der die vier Impuls- und die zwei Teilchensummationen in (5.21) normiert.

$$\begin{aligned}\left(\left\langle \rho^*_\alpha(\vec{q}_1)\rho^*_\beta(\vec{q}_2)|\rho_{\alpha'}(\vec{q}_1')\rho_{\beta'}(\vec{q}_2')\right\rangle^{-1}\right)_{\zeta\xi,\zeta'\xi'} &\approx \frac{1}{8N^2}\Big\{[S_{\zeta\zeta'}(\vec{q}_1)S_{\xi\zeta'}(\vec{q}_2)]^{-1}[\delta_{\zeta\zeta'}\delta_{\xi\zeta'}\delta_{\vec{q}_1\vec{q}_1'}\delta_{\vec{q}_2\vec{q}_2'}]+\\ &\quad +[S_{\zeta\xi'}(\vec{q}_1)S_{\xi\xi'}(\vec{q}_2)]^{-1}[\delta_{\zeta\xi'}\delta_{\xi\xi'}\delta_{\vec{q}_1\vec{q}_2'}\delta_{\vec{q}_2\vec{q}_1'}]\Big\}\end{aligned}$$

Paarmodenterm Die beiden verbliebenen Terme in (5.21)[3] können jeweils mit dem zu $\mathcal{P}$ orthogonalen Projektor $\mathcal{Q}':=1-\mathcal{P}_\rho-\mathcal{P}_j$ in drei Terme zerlegt werden. Der Term $\langle\ddot{\rho}^*_\alpha(\vec{q})|\mathcal{P}_j|\rho_\zeta(\vec{q}_1)\rho_\xi(\vec{q}_2)\rangle$ ist dabei Null, da der Teilchenstrom gerade und die Teilchendichte ungerade in t ist und das Skalarprodukt unter Zeittranslation invariant ist. Die zwei anderen Terme werden jeweils mittels der Hermitizität von $\mathcal{L}$ und der OZ-Gleichung berechnet [Göt89] und es ergibt sich insgesamt:

$$\begin{aligned}\langle\ddot{\rho}^*_\alpha(\vec{q})|\mathcal{Q}|\rho_\zeta(\vec{q}_1)\rho_\xi(\vec{q}_2)\rangle &= \frac{N_\alpha k_B T}{m}\delta_{\vec{q}-\vec{q}_1,\vec{q}_2}S_{\alpha\zeta}(\vec{q}_1)S_{\alpha\xi}(\vec{q}_2)\times\\ &\quad\times\Big\{\vec{q}\cdot\Big[\vec{q}\frac{\delta_{\zeta\xi}}{x_\alpha}-\vec{q}_1\rho c_{\alpha\zeta}(\vec{q}_1)-\vec{q}_2\rho c_{\alpha\xi}(\vec{q}_2)\Big]+\\ &\quad+\sum_{\gamma,\delta=1}^{2}\frac{q^2}{2}\langle\rho^*_\delta(\vec{q})|\rho_\zeta(\vec{q}_1)\rho_\xi(\vec{q}_2)\rangle\times\\ &\quad\times[NS_{\alpha\xi}(\vec{q})S_{\alpha\zeta}(\vec{q}_1)S_{\alpha\xi}(\vec{q}_2)]^{-1}\Big\}\end{aligned}\tag{5.22}$$

Die letzte Näherung der MCT besteht nun darin, den Term mit dem thermodynamischen Mittelwert der drei Dichten $\delta_{\vec{q}-\vec{q}_1,\vec{q}_2}\sum_{\gamma,\delta=1}^{2}\frac{q^2}{2}\langle\rho^*_\delta(\vec{q})|\rho_\zeta(\vec{q}_1)\rho_\xi(\vec{q}_2)\rangle[NS_{\alpha\xi}(\vec{q})S_{\alpha\zeta}(\vec{q}_1)S_{\alpha\xi}(\vec{q}_2)]^{-1}$ gegen den Term $\delta_{\vec{q}-\vec{q}_1,\vec{q}_2}q^2\frac{\delta_{\zeta\xi}}{x_\alpha}$ wegfallen zu lassen. In [HM86, Göt89] ist diese Superpositionsapproximation erläutert.

[3] Zeilen eins und fünf.

Diese Approximation ist für unser System nicht abgeschätzt. Für harte Kugeln in drei Dimensionen wurde allerdings gezeigt [JLBL89], dass der Einfluss dieser Dreierkorrelation die kritische Packungsdichte nur im Bereich von $\sim 1\%$ ist. Jetzt können alle Terme zusammengesetzt werden und es ergibt sich der Relaxationskern.

Monodisperser Relaxationskern Die Berechnung des monodispersen Relaxationskernes in drei Dimensionen findet sich in [Göt89] und ist in zwei Dimensionen vollkommen analog. Da sich die thermische Geschwindigkeit $v^2_{\text{therm}} := \frac{k_B T}{m}$ herauskürzt, wenn der Relaxationskern in (5.15) eingesetzt wird, definieren wir

$$m(q;t) := \frac{S(q)}{q^2}\frac{m}{k_B T}M(q;t) \tag{5.23}$$

Außerdem definieren wir den Impuls

$$\vec{p} := \vec{q} - \vec{k}$$

Durch diese Definition kann die im Relaxationskern verbleibende Summation über zwei Impulse $\vec{k}$ und $\vec{p}$ mit $\vec{k} + \vec{p} = \vec{q}$ umgeschrieben werden in eine Summation über $\vec{k}$, da über $\vec{p} = \vec{q} - \vec{k}$ implizit summiert wird. Weiter definieren wir den auf den Zeitpunkt $t = 0$ normieren Korrelator (motiviert aus der Mori-Zwanzig-Gleichung für $\mathbf{S}(q)$ (5.15))

$$\Phi(q;t) := \frac{S(q;t)}{S(q)} \tag{5.24}$$

$\Phi(q;t)$ wird als *intermediäre Streufunktion* (siehe Kapitel 1) bezeichnet. Es ergibt sich insgesamt für den *monodispersen Relaxationskern*:

$$m(q;t) = \frac{S(q)\rho}{q^2 2F}\sum_{\vec{k}} S(k)S(p)\{\vec{q}\cdot[\vec{k}c(k) + \vec{p}c(p)]\}^2\Phi(k;t)\Phi(p;t) \tag{5.25}$$

Binärer Relaxationskern Analog zum monodispersen Fall definieren wir

$$m_{\alpha\beta}(q;t) := \frac{S_{\alpha\beta}(q)}{q^2}\frac{m}{k_B T}\frac{1}{(x_\alpha x_\beta)^2}M_{\alpha\beta}(q;t) \tag{5.26}$$

Wobei $x_\alpha := \frac{N_\alpha}{N}$ die Konzentration der Teilchensorte α bezeichne. Wir formulieren die binäre Modenkopplungsgleichung für den Strukturfaktor $S(q;t)$ und der *binäre Relaxationskern* lautet:

$$\begin{aligned} m_{\alpha\beta}(q;t) \quad = \quad & \frac{S_{\alpha\beta}(q)\rho}{q^2 x_\alpha x_\beta 2F}\sum_{\vec{k}}\sum_{\rho\rho',\sigma\sigma'=1}^{2} S_{\rho\rho'}(k;t)S_{\sigma\sigma'}(p;t)\times \\ & \times\vec{q}\cdot[\vec{k}c_{\alpha\rho}(k)\delta_{\alpha\sigma} + \vec{p}c_{\alpha\sigma}(p)\delta_{\alpha\rho}]\times \\ & \times\vec{q}\cdot[\vec{p}c_{\beta\rho'}(p)\delta_{\beta\sigma'} + \vec{k}c_{\beta\sigma'}(k)\delta_{\beta\rho'}] \end{aligned} \tag{5.27}$$

Die beiden Resultate (5.25) und (5.27) sehen bis auf die Dimensionalität der Vektoren genauso aus, wie in drei Dimensionen (siehe [Göt89, Voi03]). Allgemein geht die Dimension des Systems erst explizit ein, wenn der thermodynamische Limes im Relaxationskern durchgeführt wird.

Das Modenkopplungsfunktional In der Literatur [Göt89] werden abkürzend *Vertizes* definiert, mit deren Hilfe sich der Relaxationskern übersichtlicher schreiben lässt. Betrachten wir zuerst den monodispersen Fall und führen den Vertex

$$V(\vec{q};\vec{k},\vec{p}) := \frac{1}{q}\{\vec{q}\cdot[\vec{k}c(k)+\vec{p}c(p)]\} \tag{5.28}$$

ein. Es lässt sich zunächst der Relaxationskern mittels (5.28) schreiben als

$$m(q;t) = \frac{\rho}{2F}S(q)\sum_{\vec{k}} S(k)S(p)V(\vec{q};\vec{k},\vec{p})V(\vec{q};\vec{k},\vec{p})\Phi(k;t)\Phi(p;t) \tag{5.29}$$

Nun identifizieren wir die rechte Seite von (5.29) mit einem Funktional: Dem *Modenkopplungsfunktional*

$$m(q;t) =: \mathcal{F}\left[\{V(\vec{q};\vec{k},\vec{p})\},\{\Phi(q;t)\}\right](q) \tag{5.30}$$

Analog kann für binäre Systeme ein Vertex

$$V_{\alpha\beta\gamma}(\vec{q};\vec{k},\vec{p}) := \frac{1}{q}\left(\vec{q}\cdot\vec{k}c_{\alpha\beta}(k)\delta_{\alpha\gamma}+\vec{q}\cdot\vec{p}c_{\alpha\gamma}(p)\delta_{\alpha\beta}\right) \tag{5.31}$$

definiert werden [Voi03][4] mit dem dann der Relaxationskern lautet

$$m_{\alpha\beta}(q;t) = \frac{\rho}{2Fx_\alpha x_\beta}S_{\alpha\beta}(q)\sum_{\vec{k}}\sum_{\rho\rho',\sigma\sigma'=1}^{2} S_{\rho\rho'}(k;t)S_{\sigma\sigma'}(p;t)V_{\alpha\rho\rho'}(\vec{q};\vec{k},\vec{p})V_{\beta\sigma\sigma'}(\vec{q};\vec{k},\vec{p}) \tag{5.32}$$

Genau wie für die monodispersen Systeme kann nun die rechte Seite von (5.32) mit dem *Modenkopplungsfunktional* identifiziert werden

$$m_{\alpha\beta}(q;t) =: \mathcal{F}_{\alpha\beta}[\{V_{\alpha\beta\gamma}(\vec{q};\vec{k},\vec{p})\},\{S_{\alpha\beta}(\vec{q};t)\}](q) \tag{5.33}$$

Aus dem Modenkopplungsfunktional, das von den Vertizes und Strukturfaktoren abhängt, lässt sich also der NEP eines Systems approximativ errechnen.

[4] Wir haben dabei im Vertex die Korrelation der drei Dichten weggelassen, da sie bereits im Paarmodenterm (5.22) vernachlässigt wurden.

5.4. Aussagen der Modenkopplungstheorie

Wir wollen nun einige allgemeinen Aussagen der MCT angeben. Dazu betrachten wir abkürzend nur monodisperse Systeme. Führen wir den Normierungsfaktor

$$\Omega(q) := q\sqrt{\frac{k_B T}{mS(q)}}$$

analog zu [Göt89] ein, mit dem $m(q;t) = \Omega(q)^2 M(q;t)$ gilt, so lautet die Mori-Zwanzig-Gleichung für monodisperse Systeme

$$\hat{\Phi}(q;z) = -\left[z - \Omega(q)^2\left[z \pm i\nu(q) + \Omega(q)^2 \hat{m}(q;z)\right]^{-1}\right]^{-1}$$

Eine Laplace-Rücktransformation dieser Gleichung liefert im Zeitbereich eine Faltung

$$\ddot{\Phi}(q;t) + \Omega(q)^2\Phi(q;t) + \nu(q)\dot{\Phi}(q;t) + \Omega(q)^2\int_0^t d\tau m(q;\tau)\dot{\Phi}(q;t-\tau) = 0 \tag{5.34}$$

Es handelt sich in (5.34) um eine Integro-DGL zweiter Ordnung mit einem Faltungsintegral und den durch die Eigenschaften der Korrelationsfunktion $\Phi(q;t)$ vorgegebenen Anfangsbedingungen $\Phi(q;0) = 1$ und $\dot{\Phi}(q;0) = 0$. Diese Gleichung kann allerdings nicht einfach als harmonischer Oszillator mit einer Reibung $\nu(q)$ interpretiert werden. Die Integro-DGL (5.34) hat eine instantane Komponente in Form des Reibungstermes, der aus schnellen Stößen zwischen Teilchen resultiert und charakteristisch für Gase bei geringen Packungsdichten ist. Weiter gibt es eine retardierende Komponente in Form des Faltungsintegrales, die für große Zeiten dominiert und das kollektive Verhalten der Teilchen darstellt. Die Bezeichnung von $m(q;t)$ als Gedächtniskern wird in (5.34) besonders anschaulich, denn in $m(q;t)$ steckt die Information, wie sich das System zu einer Zeit τ mit $0 \leq \tau \leq t$ verhalten hat.

Insbesondere beschreibt das Integral den *Cage-Effekt*. Als Cage-Effekt bezeichnet man die Tatsache, dass einzelne Teilchen in einem von anderen Teilchen gebildeten Käfig gefangen sein können. Die in dieser Arbeit betrachtete idealisierte MCT beschreibt allerdings nicht den *Hopping-Effekt*, durch den gefangene Teilchen bei $T < T_c$ bzw $\eta > \eta_c$ aus dem Käfig wieder ausbrechen können. Dabei ist η_c die kritische Packungsdichte und T_c ist die kritische Temperatur, an der das System einen Glasübergang aufweist (siehe Abschnitt 5.6). Wie bereits in der Einleitung (Kapitel 1) erwähnt zerfällt $\Phi(q;t)$ in zwei Stufen. Das Plateau in $\Phi(q;t)$ ist gerade durch den Cage-Effekt zu erklären.

Prinzipiell ist es möglich mit (5.34) eine Lösung für $\Phi(q;t)$ für alle Zeiten t zu finden. Wir sind jedoch nur an dem für den Glasübergang kritischen Parameter interessiert. Daher brauchen wir (siehe Kapitel 1) nur den Langzeitlimes $t \to \infty$ für $\Phi(q;t)$ zu betrachten. Das führt direkt auf die Definition der NEP (Abschnitt 5.6).

5.5. Zweidimensionale Impulsintegration

Nun soll die Impulssummation im Relaxationskern (5.27) (bzw. monodispers (5.25)) berechnet werden. Bis jetzt wurde alles für endliche Systeme formuliert; jetzt wird der thermodynamische Limes

durchgeführt. Der thermodynamische Limes einer Funktion $f(N,F,\rho)$ ist wie üblich definiert als $\lim_{N,F\to\infty;\rho=const} f(N,F,\rho)$. Die Summe kann nun in ein Integral konvertiert werden

$$\lim_{N,F\to\infty;\rho=const} \frac{1}{F}\sum_{\vec{q}} f(\vec{q}) = \lim_{N,F\to\infty;\rho=const} \frac{1}{4\pi^2}\sum_{\vec{q}} \frac{4\pi^2}{F} f(\vec{q}) = \frac{1}{4\pi^2}\int_{\mathbb{R}^2} d^2q f(\vec{q}) \tag{5.35}$$

Damit lautet der Relaxationskern (5.27) im thermodynamischen Limes

$$\begin{aligned} m_{\alpha\beta}\quad(q;t) = \quad & \frac{\rho}{q^2 x_\alpha x_\beta 8\pi^2} S_{\alpha\beta}(q)\int_{\mathbb{R}^2} d^2k \sum_{\rho\rho',\sigma\sigma'=1}^{2} S_{\rho\rho'}(k;t)S_{\sigma\sigma'}(p;t)\vec{q}\cdot[\vec{k}c_{\alpha\rho}(\vec{k})\delta_{\alpha\sigma}+\vec{p}c_{\alpha\sigma}(\vec{p})\delta_{\alpha\rho}]\times \\ \times \quad & \vec{q}\cdot[\vec{p}c_{\beta\rho'}(\vec{p})\delta_{\beta\sigma'}+\vec{k}c_{\beta\sigma'}(\vec{k})\delta_{\beta\rho'}] \end{aligned} \tag{5.36}$$

Dieses Integral kann allerdings immer noch nicht direkt berechnet werden. Zur weiteren Vereinfachung rufen wir uns ins Gedächtnis, dass $\vec{p}$ und $\vec{k}$ mit einem von außen gegebenen $\vec{q}$ über $\vec{k}+\vec{p}=\vec{q}$ zusammenhängen. Gesucht ist also eine Möglichkeit, die Integration $\int_{\mathbb{R}^2} d^2k f(\vec{q},\vec{k},\vec{p})$ mit der Nebenbedingung $\vec{p}=\vec{q}-\vec{k}$ zu vereinfachen. Dazu definieren wir ein zweidimensionales kartesisches Koordinatensystem mit dem Einheitsvektor $\hat{x}_1 \parallel \vec{q}$. Das ist in Abb. 5.1 dargestellt. Dann lauten mit

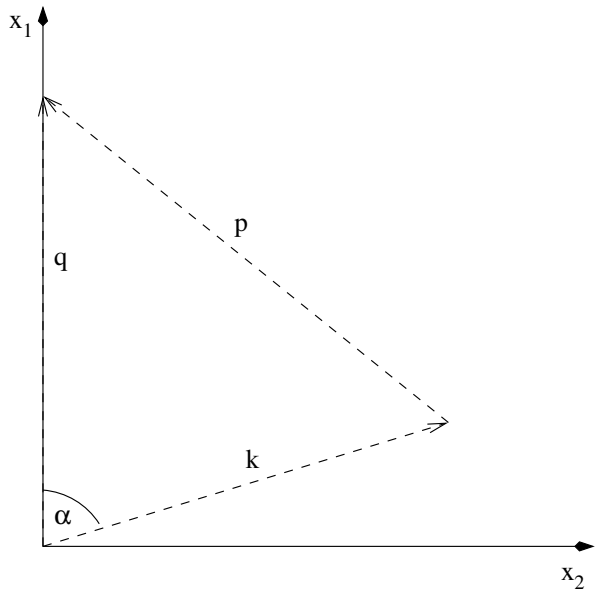

Abbildung 5.1.: Koordinatentransformation der Impulsintegration im Modenkopplungsfunktional

$0 \le \alpha < \pi$ die Vektoren im neuen Koordinatensystem

$$\begin{aligned} \vec{q} &= (q_1,q_2)^T = q(1,0)^T \\ \vec{k} &= (k_1,k_2)^T = k(\cos\alpha,\sin\alpha)^T \end{aligned}$$

Es gilt weiter $p^2 = q^2+k^2-2qk\cos\alpha$. Und damit folgt dann für die Komponenten von $\vec{k}$:

$$
\begin{aligned}
k_1(k,p) &= \frac{1}{2q}\left(q^2+k^2-p^2\right) \\
k_2(k,p) &= \pm\frac{1}{2q}\sqrt{4q^2k^2-\left(q^2+k^2-p^2\right)^2}
\end{aligned}
\tag{5.37}
$$

Damit kann direkt die Jacobideterminante der Koordinatentransformation berechnet werden:

$$
\frac{\partial(k_1,k_2)}{\partial(k,p)} = 2\frac{2kp}{\sqrt{4q^2k^2-\left(q^2+k^2-p^2\right)^2}}
\tag{5.38}
$$

Aus der Mehrwertigkeit der Wurzel in (5.37) folgt der zusätzliche Faktor 2 in der Jacobideterminante (5.38).

Mit Hilfe dieser Koordinatentransformation gilt dann

$$
\begin{aligned}
\int_{\mathbb{R}^2} d^2k f(\vec{q},\vec{k},\vec{p}) &= \int_{\mathbb{R}^+} dk \int_{|q-k|}^{q+k} dp \frac{\partial(k_1,k_2)}{\partial(k,p)} f(\vec{q},\vec{k},\vec{p}) \\
&= \frac{2}{q}\int_{\mathbb{R}^+} dkk \int_{|q-k|}^{q+k} dpp \frac{f(\vec{q},\vec{k},\vec{p})}{\sqrt{k^2-\left(\frac{q^2+k^2-p^2}{2q}\right)^2}}
\end{aligned}
\tag{5.39}
$$

Jetzt können wir den Relaxationskern im thermodynamischen Limes mit der Transformation (5.39) umformen. Der Vollständigkeit halber geben wir sowohl das monodisperse

$$
\begin{aligned}
m(q;t) &= \frac{S(q)}{q^5}\frac{\rho}{16\pi^2}\int_{\mathbb{R}^+} dkkS(k)\Phi(k;t)\int_{|q-k|}^{q+k} dppS(p)\Phi(p;t)\times \\
&\quad\times\frac{|(q^2-p^2+k^2)c(k)+(q^2-k^2+p^2)c(p)|^2}{\sqrt{k^2-[\frac{q^2-p^2+k^2}{2q}]^2}}
\end{aligned}
\tag{5.40}
$$

als auch das binäre Ergebnis an:

$$
\begin{aligned}
m_{\alpha\beta}(q;t) &= \frac{S_{\alpha\beta}(q)}{q^5}\frac{\rho}{x_\alpha x_\beta 16\pi^2}\sum_{\rho\rho',\sigma\sigma'=1}^{2}\int_{\mathbb{R}^+} dkkS_{\rho\rho'}(k;t)\int_{|q-k|}^{q+k} dppS_{\sigma\sigma'}(p;t)\times \\
&\quad\times\frac{1}{\sqrt{k^2-[\frac{q^2-p^2+k^2}{2q}]^2}}[(q^2+k^2-p^2)c_{\alpha\rho}(\vec{k})\delta_{\alpha\sigma}+(q^2-k^2+p^2)c_{\alpha\sigma}(\vec{p})\delta_{\alpha\rho}]\times \\
&\quad\times[(q^2-k^2+p^2)c_{\beta\rho'}(\vec{p})\delta_{\beta\sigma'}+(q^2+k^2-p^2)c_{\beta\sigma'}(\vec{k})\delta_{\beta\rho'}]
\end{aligned}
\tag{5.41}
$$

Dabei wurden die verbleibenden Skalarprodukte $\vec{q}\cdot\vec{k}$ und $\vec{q}\cdot\vec{p}$ zwischen den Impulsen elementar berechnet.

5.6. Der Nichtergodizitätsparameter[5]

Wie bereits in der Einleitung (Kapitel 1) erwähnt, geht der Korrelator $S_{\alpha\beta}(q;t)$ in der Glasphase mit der Zeit nicht mehr gegen Null. Also definieren wir den Grenzwert dieses Korrelators als Funktion des Impulses und bezeichnen ihn als NEP:

$$\begin{aligned} F_{\alpha\beta}(q) &:= \lim_{t\to\infty} S_{\alpha\beta}(q;t) \\ f(q) &:= \lim_{t\to\infty} \Phi(q;t) = \lim_{t\to\infty} \frac{S(q;t)}{S(q)} \end{aligned} \tag{5.42}$$

Zur besseren Unterscheidung des normierten monodispersen NEP und des nicht normierten binären NEP verwenden wir $f(q)$ und $F_{\alpha\beta}(q)$. Dabei wurde $F_{\alpha\beta}(q)$ nicht auf $S_{\alpha\beta}(q)$ normiert, da die Nebendiagonalelemente von $S_{\alpha\beta}(q)$ für große q gegen Null gehen und im Relaxationskern auch über große Impulse summiert wird. Auf eine hemitesche Normierung der Form $\Phi(q;t) := \mathbf{S}(q)^{1/2}\mathbf{S}(q;t)\mathbf{S}(q)^{1/2}$ haben wir verzichtet. Nun wollen wir die Modenkopplungsgleichung für den Limes $t \to \infty$ betrachten, um für den NEP ein einfacheres Gleichungssystem als in (5.34) beschrieben zu erhalten. Nach dem Anfangswertsatz gilt $\lim_{t\to\infty} f(t) = -\lim_{z\to 0} z\hat{f}(z)$ und damit für den NEP $F_{\alpha\beta}(q) = -\lim_{z\to 0} z\hat{S}_{\alpha\beta}(q;z)$. Betrachten wir zunächst wieder die Mori-Zwanzig-Gleichung (5.15) und schreiben sie für die intermediäre Streufunktion um:

$$\hat{\mathbf{S}}(q;z) = -[z\mathbf{S}(q)^{-1} - \mathbf{S}(q)^{-1}[z\mathbf{J}(q)^{-1} + \hat{\mathbf{M}}(q;z)]^{-1}\mathbf{S}^{-1}(q)]^{-1} \tag{5.43}$$

Damit ergibt sich durch elementare Rechnung für den NEP inversenfrei geschrieben

$$\mathbf{F}(q) = \mathbf{S}(q)\,[\mathbf{m}(q)]\,[\mathbf{S}(q) - \mathbf{F}(q)] \tag{5.44}$$

Die inversenfreie Gleichung (5.44) hat dabei gegenüber einer Schreibweise mit Inversen Matrizen den Vorteil, dass die Rechenzeit geringer ist. Der Relaxationskern im Langzeitlimes lautet mit der Vertexschreibweise und (5.26)

$$\begin{aligned} m_{\alpha\beta}(q) &:= \lim_{t\to\infty} m_{\alpha\beta}(q;t) \\ &= \frac{\rho}{2Fx_\alpha x_\beta} S_{\alpha\beta}(q) \sum_{\vec{k}} \sum_{\rho\rho',\sigma\sigma'=1}^{2} V_{\alpha\rho\sigma}(\vec{q};\vec{k},\vec{p}) V_{\beta\rho'\sigma'}(\vec{q};\vec{k},\vec{p}) F_{\rho\rho'}(k) F_{\sigma\sigma'}(p) \\ &= \frac{S_{\alpha\beta}(q)}{q^3} \frac{\rho}{x_\alpha x_\beta 16\pi^2} \sum_{\rho\rho',\sigma\sigma'=1}^{2} \int_{\mathbb{R}^+} dkk F_{\rho\rho'}(k) \int_{|q-k|}^{q+k} dpp F_{\sigma\sigma'}(p) \times \\ &\quad \times \frac{1}{\sqrt{k^2 - [\frac{q^2-p^2+k^2}{2q}]^2}} V_{\alpha\rho\rho'}(\vec{q};\vec{k},\vec{p}) V_{\beta\sigma\sigma'}(\vec{q};\vec{k},\vec{p}) \end{aligned} \tag{5.45}$$

[5] Im folgenden kurz mit NEP bezeichnet.

Für den Fall der monodispersen Systeme lautet die Mori-Zwanzig-Gleichung der intermediären Streufunktion

$$\hat{\Phi}(q;z) = -\frac{1}{z - \frac{\Omega(q)^2}{z+\Omega(q)^2 \hat{m}(q;z)}}$$

Und für den NEP gilt:

$$f(q) = [1 - f(q)]m(q) \tag{5.46}$$

mit dem Relaxationskern

$$\begin{aligned} m(q) &:= \lim_{t\to\infty} m(q;t) \\ &= \frac{\rho}{2F} S(q) \sum_{\vec{k}} S(k)S(p)V(\vec{q};\vec{k},\vec{p})V(\vec{q};\vec{k},\vec{p})f(k)f(p) \\ &= \frac{S(q)}{q^3}\frac{\rho}{16\pi^2}\int_{\mathbb{R}^+} dkkS(k)f(k)\int_{|q-k|}^{q+k} dppS(p)f(p)\times \\ &\quad \times \frac{1}{\sqrt{k^2 - [\frac{q^2-p^2+k^2}{2q}]^2}} V(\vec{q};\vec{k},\vec{p})V(\vec{q};\vec{k},\vec{p}) \end{aligned} \tag{5.47}$$

Dabei wurde analog zum binären Fall auch in (5.46) eine divisionsfreie Formulierung gewählt, um den Rechenaufwand zu verringern. Damit ist durch (5.44) und (5.45) (bzw. (5.46) und (5.47)) ein geschlossenes GLS gegeben, mit dem es möglich ist, allein aus der Kenntnis des statischen Strukturfaktors[6] $S_{\alpha\beta}(q)$ den NEP $F_{\alpha\beta}(q)$ zu berechnen. Die dazu notwendige Numerik stellen wir im folgenden Kapitel vor.

Kritische Parameter Wie bereits zuvor erwähnt fällt in der Glasphase der Korrelator $S_{\alpha\beta}(q;t)$ für große t nicht auf Null ab. Damit ist insbesondere der NEP ungleich Null. Betrachten wir nun das System monodisperser harter Scheiben, dann gibt es eine kritische Packungsdichte η_c, oberhalb derer der NEP $f(q)$ immer ungleich Null ist. Unterhalb der kritischen Packungsdichte gilt[7] dann $f(q) \approx 0$. Damit kann also der Glasübergang an der Norm des NEP abgelesen werden. Betrachten wir weiter das monodisperse System dipolarer harter Scheiben, das einen temperaturgetriebenen Glasübergang aufweist. Dort gilt ebenso für die Temperaturen oberhalb einer kritischen Temperatur T_c, dass der NEP auf Null abgefallen ist, während er für $T < T_c$ ungleich Null ist. Zusammengefasst heißt das also, dass mittels der MCT aus der Langzeitentwicklung des dynamischen Strukturfaktors der Glasübergang abgelesen werden kann. Wir wollen im folgenden unter dem *kritischen NEP* den NEP eines Systems am kritischen Parameter verstehen.

Abschließend betrachten wir kurz die Temperaturabhängigkeit des NEP und halten fest, dass nach [Göt89] für den monodispersen Fall gilt:

[6] Die direkte Korrelationsfunktion $c(q)$ kann aus $S(q)$ mit OZ leicht bestimmt werden, z.B. monodispers: $c(q) = \frac{S(q)(S(q)-1)}{\rho}$

[7] Der NEP fällt numerisch auf $< 10^{-14}$ ab.

$$f_q = \begin{cases} f_q^c + c_q\sqrt{T_c - T} & T < T_c \\ 0 & T > T_c \end{cases} \tag{5.48}$$

Ein analoges Resultat ergibt sich für die binären Systeme.

Äquivalente Strukturfaktoren oder ist die Dichte ausgezeichnet? Erinnern wir uns an Gleichung (2.6) und betrachten abkürzend o.B.d.A. ein monodisperses System. Dann gilt insbesondere für rein dipolare Potentiale $V(r) \sim r^{-3}$ in zwei Dimensionen:

$$S(q;T;\rho) = \tilde{S}\left(\bar{q} = q\rho^{\frac{1}{2}}; \bar{T} = T\rho^{\frac{3}{2}}\right)$$

Die Frage ist nun, ob die Dichte in (5.47) ausgezeichnet ist, oder ob der Relaxationskern ebenso wie der Strukturfaktor für gewisse Temperatur-Dichte-Kombination das physikalisch gleiche Verhalten aufweist. Führen wir dazu in (5.47) die Substitution $q \mapsto \bar{q} = q\rho^{1/2}$ in allen Impulsen durch, so folgt:

$$\begin{aligned} m(q) &= \frac{S(\bar{q})}{\bar{q}^3}\frac{\rho^{3/2}}{16\pi^2}\int_{\mathbb{R}^+} d\bar{k}\bar{k}\frac{1}{\rho}S(\bar{k})f(\bar{k})\int_{|\bar{q}-\bar{k}|}^{\bar{q}+\bar{k}} d\bar{p}\bar{p}\frac{1}{\rho}S(\bar{p})f(\bar{p})\times \\ &\quad \times\frac{1}{\sqrt{\frac{1}{\rho}}\sqrt{\bar{k}^2 - [\frac{\bar{q}^2-\bar{p}^2+\bar{k}^2}{2\bar{q}}]^2}}\left[\frac{\vec{\bar{q}}}{\bar{q}}\cdot\left\{\frac{\vec{\bar{k}}}{\rho^{1/2}}\frac{S(\bar{k})[S(\bar{k})-1]}{\rho} + \frac{\vec{\bar{p}}}{\rho^{1/2}}\frac{S(\bar{p})[S(\bar{p})-1]}{\rho}\right\}\right]^2 \\ &= \frac{S(\bar{q})}{\bar{q}^3}\frac{1}{16\pi^2}\int_{\mathbb{R}^+} d\bar{k}\bar{k}S(\bar{k})f(\bar{k})\int_{|\bar{q}-\bar{k}|}^{\bar{q}+\bar{k}} d\bar{p}\bar{p}S(\bar{p})f(\bar{p})\times \\ &\quad \times\frac{1}{\sqrt{\bar{k}^2 - [\frac{\bar{q}^2-\bar{p}^2+\bar{k}^2}{2\bar{q}}]^2}}\left[\frac{\vec{\bar{q}}}{\bar{q}}\cdot\left\{\vec{\bar{k}}S(\bar{k})[S(\bar{k})-1] + \vec{\bar{p}}S(\bar{p})[S(\bar{p})-1]\right\}\right]^2 \end{aligned} \tag{5.49}$$

Das heißt aber, dass die Dichte nicht explizit in das Modenkopplungsfunktional eingeht und auch für den Langzeitlimes[8] das System physikalisch allein durch den Parameter Γ_m bestimmt ist. Damit ist insbesondere zu erwarten, dass für das System der monodispersen dipolaren harten Scheiben (Abschnitt 7.2) für geringe Packungsdichten der Glasübergang unabhängig von der Packungsdichte wird und immer bei einem festen Γ_m^c stattfindet.

[8] Die Aussage gilt auch allgemein für binäre Systeme zu einer beliebigen Zeit t.

6. Numerik der Modenkopplung

In diesem Kapitel werden wir die numerischen Methoden vorstellen, die zur Iteration des Gleichungssystems (5.44) und (5.45) (bzw. monodispers (5.46) und (5.47)) genutzt wurden, um den NEP zu bestimmen. Die Dynamik erweist sich numerisch wesentlich gutmütiger als die Statik. Daher genügt es, einen reinen Picard-Algorithmus zur Iteration des jeweiligen GLS zu verwenden. Zunächst gehen wir jedoch in Abschnitt 6.1 auf die Problematik der Diskretisierung ein. Dann beschreiben wir in Abschnitt 6.2, wie die numerische Integration durchgeführt wird. Abschließend stellen wir in Abschnitt 6.3 den Algorithmus zur Iteration des NEP und die verwendete Methode zur Suche nach den kritischen Parametern vor.

Es sei vorab angemerkt, dass zwar der Algorithmus für binäre Systeme implementiert wurde, allerdings die vorhandene Rechenzeit nicht mehr ausreichte um Glasübergänge der binären zweidimensionalen Systeme im Rahmen der Diplomarbeit zu untersuchen (siehe Kapitel 8). Daher wollen wir in diesem Kapitel zwar die Algorithmen für ein binäres System formulieren, allerdings die Parameterabhängigkeiten nur für monodisperse Systeme diskutieren.

In diesem Kapitel werden sämtliche Indices (analog zu Kapitel 3) in Konsistenz zum Quelltext der Programme bei Null beginnen.

6.1. Diskretisierung des Gleichungssystems

Das Gleichungssystem sowohl für monodisperse, als auch für binäre Systeme enthält im thermodynamischen Limes zwei Integrationen, die numerisch behandelt werden müssen. Weil wir im Gegensatz zu dem statischen GLS (2.18), (2.19) und (3.2) keine FBT mehr benötigen, können wir auf eine äquidistante Diskretisierung zurückgreifen und damit eine Simpsonformel zur Integration verwenden (siehe Abschnitt 6.2). Wir wählen also N diskrete Impulse im Intervall $[q_0, q_{N-1}]$, die wir mit m indizieren wollen:

$$q_m := q_0 + m\frac{q_{N-1} - q_0}{N} \qquad 0 \leq m \leq N-1 \tag{6.1}$$

Dabei nennen wir im folgenden die Schrittweite im Impulsraum auch $\Delta q := \frac{q_{N-1}-q_0}{N}$. Den maximalen Impuls bezeichnen wir im nächsten Kapitel mit $Q := q_{N-1}$ und verwenden q_{N-1} hier nur, um Konsistenz mit der Stützstellenzahl zu wahren. Bezeichnen wir im folgenden mit $G_m := G(q_m)$ eine im Impulsraum diskretisierte Funktion $G(q_m)$, so lautet der erste Teil des diskretisierten GLS:

$$\begin{aligned} f_m &= (1-f_m)\, m_m \\ \mathbf{F}_m &= \mathbf{S}_m\,[\mathbf{m}_m]\,[\mathbf{S}_m - \mathbf{F}_m] \end{aligned} \tag{6.2}$$

Nun soll weiter die Integration über k, p in (5.45) (bzw. monodispers (5.47)) diskretisiert werden. Das impliziert aber, dass der Strukturfaktor und die direkte Korrelationsfunktion an mit (6.1) äquidistant diskretisierten Stellen im Impulsraum bekannt sein müssen. Weil diese jedoch nach (3.1) nicht an diesen Punkten bekannt sind, muss interpoliert werden. Die Interpolation beschreiben wir im Anhang (Abschnitt A.2).

6.2. Numerische Impulsintegration

Betrachten wir nun zunächst die Grenzen der verbleibenden Integration in (5.45) (bzw. monodispers (5.47)). Zu diskretisieren ist ein Integral der Form:

$$\begin{aligned} g_q &:= \int_{\mathbb{R}^+} dkk \int_{|q-k|}^{q+k} dppG(q;k,p) \\ &\approx \int_{q_0}^{q_{N-1}} dkk \int_{|q-k|}^{q+k} dppG(q;k,p) \end{aligned} \tag{6.3}$$

Diskretisieren wir nun die Impulse k, p analog zu (6.1), so ist in (6.3) ersichtlich, dass für die obere Grenze $q+k$ der p-Integration eine weitere Approximation nötig ist, um zu gewährleisten, dass $S(q)$ nur bishin zu q_{N-1} bekannt sein muss:

$$g_q \approx \int_{q_0}^{q_{N-1}} dkk \int_{\max(q_0,|q-k|)}^{\min(q_{N-1},q+k)} dppG(q;k,p)$$

Damit ist festgelegt, in welchen Intervallen die Impulse q, k, p laufen:

$$\begin{aligned} &q_0 \le q_m, k_n \le q_{N-1} \\ &\max(q_0, |q_m - k_n|) \le p_l \le \min(q_{N-1}, q_m + k_n) \end{aligned}$$

Die beiden Integrationen sollen nun mit der gleichen *Drei-Punkt-Simpson-Formel* [WHP92] numerisch berechnet werden. Es gilt allgemein für eine beliebige Funktion $h(k)$:

$$\begin{aligned} \int_{q_0}^{q_{N-1}} dkh(k) &= \Delta k \left[\frac{3}{8}h_0 + \frac{7}{6}h_1 + \frac{23}{24}h_2 + h_3 + ... + \right. \\ &\quad \left. + h_{N-4} + \frac{23}{24}h_{N-3} + \frac{7}{6}h_{N-2} + \frac{3}{8}h_{N-1} \right] + O\left(N^{-4}\right) \end{aligned}$$

Dabei ist $\Delta k := \frac{k_{N-1}-k_0}{N}$. Das heißt, das Integral wird eine N-Punkt-Summe mit den *renormierten Summanden* $\tilde{h}_n$ wobei $\tilde{h}_0 := \frac{3}{8}h_0$, $\tilde{h}_1 := \frac{7}{6}h_1$, $\tilde{h}_2 := \frac{23}{24}h_2$ und $\tilde{h}_{N-3} := \frac{3}{8}h_{N-3}$, $\tilde{h}_{N-2} := \frac{7}{6}h_{N-2}$, $\tilde{h}_{N-1} := \frac{23}{24}h_{N-1}$ und sonst $\tilde{h}_n = h_n$:

$$\int_{q_0}^{q_{N-1}} dkh(k) \approx \Delta k \sum_{n=0}^{N-1} \tilde{h}_n$$

Die äußere k-Integration lässt sich damit bereits berechnen. Für die p-Integration muss noch ein Kriterium für die Integrationsgrenzen angegeben werden. Für die untere Grenze gilt $p_0 := \max(q_0, |q-k|)$. Sind nun q,k diskretisiert und werden mit q_m, k_n bezeichnet, so folgt

$$p_{\min} = \max(q_0, |q_m - k_n|) = \max(q_0, |m-n|\Delta q)$$

Analog gilt

$$p_{\max} = \min(q_{N-1}, 2q_0 + (m+n)\Delta q)$$

Da alle drei Impulse in der gleichen Form diskretisiert werden, stellen wir nun den diskretisierten Impuls p_l über $p_l = q_0 + l\Delta q$ dar. Damit können die Grenzen über den Index l formuliert werden und es gilt:

$$\begin{aligned} l_{\min}(m,n) &:= \begin{cases} 0 & |m-n|\Delta q < q_0 \\ |m-n| - [\frac{q_0}{\Delta q}] & |m-n|\Delta q \geq q_0 \end{cases} \\ l_{\max}(m,n) &:= \begin{cases} N-1 & (m+n)\Delta q > q_{N-1} - 2q_0 \\ m+n+[\frac{q_0}{\Delta q}] & (m+n)\Delta q \leq q_{N-1} - 2q_0 \end{cases} \end{aligned}$$

Dabei sind die eckigen Klammern $[\cdot]$ die Gaussklammern, die eine beliebige Zahl abrundend in eine natürliche Zahl konvertieren. Nun kann das Integral (6.2) mittels der Drei-Punkt-Simpson-Formel geschrieben werden als:

$$g_m \approx \Delta q^2 \sum_{n=0}^{N-1} k_n \sum_{l=l_{\min}}^{l_{\max}} p_l G_{m;n,l} \tag{6.4}$$

Schließlich folgt für den Relaxationskern für monodisperse Systeme

$$\begin{aligned} m_m &= \frac{S_m \Delta q^2}{q_m^5} \frac{\rho}{16\pi^2} \sum_{n=0}^{N-1} \tilde{k}_n S_n f_n \sum_{l=l_{\min}}^{l_{\max}} \tilde{p}_l S_l f_l \times \\ &\quad \times \frac{|(q_m^2 - p_l^2 + k_n^2)c_n + (q_m^2 - k_n^2 + p_l^2)c_l|^2}{\sqrt{k_n^2 - [\frac{q_m^2 - p_l^2 + k_n^2}{2q_m}]^2}} \end{aligned} \tag{6.5}$$

und für binäre Systeme

$$\begin{aligned} m_{\alpha\beta;m} &= \frac{S_{\alpha\beta;m}\Delta q^2}{q_m^5} \frac{\rho}{x_\alpha x_\beta 16\pi^2} \sum_{\rho\rho',\sigma\sigma'=1}^{2} \sum_{n=0}^{N-1} \tilde{k}_n F_{\rho\rho';n} \sum_{l=l_{\min}}^{l_{\max}} \tilde{p}_l F_{\sigma\sigma';l} \frac{1}{\sqrt{k_n^2 - [\frac{q_m^2 - p_l^2 + k_n^2}{2q_m}]^2}} \times \\ &\quad \times [(q_m^2 + k_n^2 - p_l^2)c_{\alpha\rho;n}\delta_{\alpha\sigma} + (q_m^2 - k_n^2 + p_l^2)c_{\alpha\sigma;l}\delta_{\alpha\rho}] \times \\ &\quad \times [(q_m^2 - k_n^2 + p_l^2)c_{\beta\rho';l}\delta_{\beta\sigma'} + (q_m^2 + k_n^2 - p_l^2)c_{\beta\sigma';n}\delta_{\beta\rho'}] \end{aligned} \tag{6.6}$$

Betrachten wir diese Ergebnisse, so wird klar, dass die Diskretisierung so gewählt werden muss, dass nur q_{N-1} und N vorgegeben werden, denn q_0 liegt implizit durch die untere Grenze der k-Integration

fest. Weiter wählen wir $q_0 := \frac{1}{2}\frac{q_{N-1}}{N}$, damit die Stützstellen der Integration immer in der Mitte der Intervalle liegen. Abschließend sei erwähnt, dass die Anzahl der Stützstellen auch in der p-Integration für die von uns gewählten Parameter q_{N-1} und N immer größer sechs ist und damit auch für die innere Integration die Drei-Punkt-Simpson-Formel angewandt werden kann.

6.3. Algorithmus zur Iteration der NEP

Wir geben den zur Iteration der NEP verwendeten Algorithmus direkt im Pseudo-Code an. Wesentlicher Kern ist eine reine Picard-Iteration mit adaptiver Iterationsschrittweite (siehe Kapitel 3). Das Konvergenzkriterium ist auch hier durch (3.12) gegeben, wobei $\delta = 10^{-8}$ gewählt wurde. Im Algorithmus greifen wir auf zwei Funktionen *MCT* und *step* zurück. Die Funktion *MCT* beschreibt einen Iterationsschritt durch das GLS (6.3) und (6.6). *step* ist analog zu Kapitel 3 eine Funktion, die die n-te mit der $n+1$-ten Lösung mischt und den Schrittweitenparameter dynamisch α anpasst.

Algorithm 6 Iteration des NEP

Require: $S_{\alpha\beta}(q)$ for any M points $q_0 \leq q_m \leq q_{M-1}$
Require: Packing fraction η
Require: Target discretization N, q_{N-1}

1: Interpolate $S_{\alpha\beta}(q) \longrightarrow S_{\alpha\beta}(q)$ at target discretization
2: $S_{\alpha\beta}(q) \longrightarrow C_{\alpha\beta}(q)$
3: Startvalue for $F^{(0)}_{\alpha\beta}(q)$
4: **repeat**
5: $\quad F^{(n-1)}_{\alpha\beta}(q) \xrightarrow{\text{MCT}} \tilde{F}^{(n)}_{\alpha\beta}(q)$
6: $\quad \tilde{F}^{(n)}_{\alpha\beta}(q), F^{(n-1)}_{\alpha\beta}(q), \alpha \xrightarrow{\text{step}} F^{(n)}_{\alpha\beta}(q), \alpha$
7: **until** $||F^{(n)}_{\alpha\beta}(q)|| < \delta \quad \wedge \quad \alpha > \delta_\alpha$
8: **return** $F_{\alpha\beta}(q), ||F_{\alpha\beta}(q)||$

Abhängigkeit des Algorithmus vom Startwert Wir wollen kurz die Abhängigkeit des Algorithmus von seinen Parametern diskutieren. Zunächst betrachten wir den Startwert $F^{(0)}_{\alpha\beta}(q)$. Wir haben als Startwerte $F^{(0)}_{\alpha\beta}(q) = S_{\alpha\beta}(q)$ und $F^{(0)}_{\alpha\beta}(q) = F^{(\text{conv})}_{\alpha\beta}(q)$ gewählt. Dabei ist $F^{(\text{conv})}_{\alpha\beta}(q)$ ein bereits konvergierter NEP. Im Gegensatz zur Statik hat die Wahl des Startwertes für die Dynamik keinen Einfluss darauf, ob die Lösung konvergiert oder nicht. Das GLS (6.3) und (6.6) erweist sich also in diesem Sinne als numerisch stabiler. Allerdings gibt es je nach Startwert noch deutliche Unterschiede in der Anzahl der Iterationsschritte bishin zur Konvergenz. Um die Anzahl der nötigen Iterationsschritte und damit den Rechenaufwand zu minimieren gehen wir analog zur Statik vor: Der NEP ist eine Funktion der Temperatur und der Packungsdichte $F_{\alpha\beta}(q;\eta_1,T_1)$, woraus sich für $T_1 \approx T_2$, $\eta_1 \approx \eta_2$ als in dieser Hinsicht guter Startwert ergibt: $F^{(0)}_{\alpha\beta}(q;\eta_2,T_2) = F_{\alpha\beta}(q;\eta_1,T_1)$.

Abhängigkeit des Algorithmus von den Integrationsparametern Als Integrationsparameter wollen wir q_{N-1}, N und auch q_0 bezeichnen. Wir diskutieren die Abhängigkeit des Algorithmus von den

Integrationsparametern am System der monodispersen harten Scheiben. Betrachten wir zunächst die Abhängigkeit von q_0. Angenommen wir wählen $q_0 \neq \frac{1}{2}\frac{q_{N-1}}{N}$. Dann ist q_0 aber nicht mehr durch Null festgelegt und somit die untere Integrationsgrenze verändert worden. Dadurch entsteht ein Fehler in der Integrationsapproximation (6.4), der sich in einer Abweichung der kritischen Packungsdichte von 10^{-2} äußert. Damit ist also klar, dass q_0 wie beschrieben als $q_0 = \frac{1}{2}\frac{q_{N-1}}{N}$ gewählt werden muss. Werfen wir nun einen Blick auf den maximalen Impuls q_{N-1}. Dazu betrachten wir einen Strukturfaktor für monodisperse harte Scheiben im Glas (siehe Abschnitt 7.1) bei einer Packungsdichte $\eta = 0.72$ (Abb. 6.1). Wie dort ersichtlich wird, ist der Strukturfaktor sehr langreichweitig. Es zeigt sich numerisch,

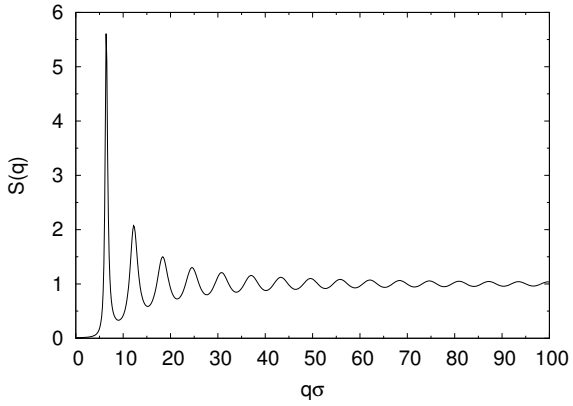

Abbildung 6.1.: $S(q)$ für monodisperse harte Scheiben aus PY bei $\eta = 0.72$

dass die kritische Packungsdichte η_c für dieses System nur im Bereich 10^{-3} variiert, wenn q_{N-1} zwischen $30 \leq q_{N-1} \leq 100$ gewählt wird. Die Ursache für diese schwache Abhängigkeit von q_{N-1} lässt sich an dem nicht-linearen GLS nur schwer ablesen. Zusammenfassend lässt sich sagen, dass sich die kritischen Packungsdichten bereits auf 10^{-2} genau berechnen lassen, wenn die Integration über das erste Maximum von S_q ausreichend gut ist. Die Anzahl der Stützstellen N ist für die monodispersen harten Scheiben weniger kritisch, als die Wahl von q_{N-1}. Dennoch wird sie insbesondere für die dipolaren harten Scheiben wichtig, da das erste Maximum in $S(q)$ wegen der Gewichtung mit q^{-1} im Relaxationskern den stärksten Einfluss auf den NEP hat. Bei sinkender Packungsdichte wird der Glasübergang der dipolaren harten Scheiben stärker temperaturgetrieben. Das äußert sich im ersten Maximum des Strukturfaktors darin, dass er sich zu kleineren q verschiebt und schmaler und höher wird. Ist dieses erste Maximum durch die Stützstellen nicht gut genug dargestellt, so ergeben sich starke Abweichungen der Größenordnung 10^3[K] der kritischen Temperatur T_c (siehe Abschnitt 7.2). Numerisch ausreichend ist eine Zahl von $N = 1500$ Stützstellen.

Abhängigkeit der Resultate von der Integrationsroutine und der Interpolation Um die Abhängigkeit des Glasüberganges und des NEP von der Integrationsroutine und der Interpolation zu überprüfen, haben wir einen Algorithmus von Sperl [Spe06] verwendet. In diesem werden die statischen Korrelatoren nur linear interpoliert und weiter eine einfache Trapezregel [Spe00] zur Integration verwendet. Setzen wir für einen Vergleich $q_{N-1} := 50$ und $N := 500$ und betrachten wieder das System monodisperser harter Scheiben. Die statischen Strukturfaktoren aus MHNC[1] des Systems übernehmen wir von Brader [Bra06] (siehe auch Abschnitt 7.1). Zwischen den bekannten Strukturfaktoren wird im vorliegenden Algorithmus mit bikubischen Splines interpoliert. Es ergeben sich zwei ver-

[1] Modified Hypernetted-Chain - siehe Abschnitt 7.1.

schiedene kritische Packungsdichten: Sperls Algorithmus liefert dann eine kritische Packungsdichte $\eta_c^{\mathrm{SPERL}} = 0.7057$. Aus dem vorliegendem Algorithmus ergibt sich eine kritische Packungsdichte von $\eta_c = 0.712$. Diese recht starke Abweichung lässt sich hauptsächlich auf die unterschiedlichen Interpolationsroutinen zurückführen: Das erste Maximum des Strukturfaktors beeinflusst die kritische Packungsdichte stark; da dieses erste Maximum von $S(q)$ relativ hoch und schmal ist (z.B. Abb. 6.1), fallen Unterschiede in der Interpolation dort umso stärker ins Gewicht.

Vergleichen wir nun noch die kritischen NEP aus den beiden Algorithmen in Abb. 6.2. Eingezeichnet sind die beiden kritischen NEP und weiter ein NEP aus unserem Algorithmus für $q_{N-1} = 25$ und $N = 250$ bei einer Packungsdichte $\eta = 0.7168$. Es ergibt sich ein deutlicher Unterschied im Verlauf der kritischen NEP für kleine Impulse. Da wir mit unserem Algorithmus das Verhalten von Sperls kritischem NEP für kleine Impulse reproduzieren können, wenn die obere Integrationsgrenze schlechter - also q_{N-1} kleiner - gewählt wird, gehen wir davon aus, dass es sich bei dem Kleinimpulsverhalten in Sperls kritischem NEP um ein numerisches Artefakt handelt [Spe06].

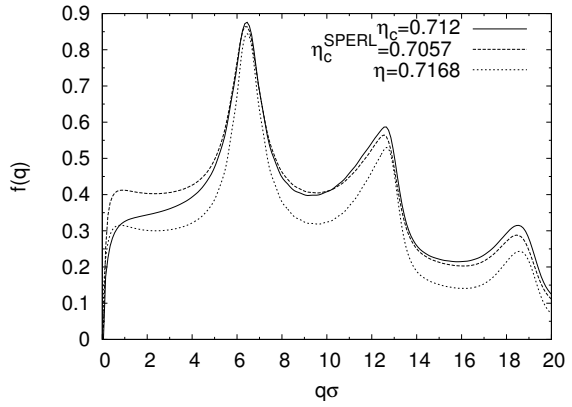

Abbildung 6.2.: Vergleich der NEP monodisperser harter Scheiben - Kurve zu $\eta_c = 0.712$ bei $N = 500$, $Q = 500$; zu $\eta = 0.7168$ bei $N = 250$, $Q = 25$; zu $\eta_c^{\mathrm{SPERL}} = 0.7057$ aus dem Algorithmus von Sperl [Spe06]

Suche nach der kritischen Packungsdichte Nun formulieren wir noch den Algorithmus zur Suche nach der kritischen Packungsdichte η_c und der kritischen Temperatur T_c eines Systems. Eine vollständige Automatisierung mittels eines Programms ist dabei nicht möglich, da an vielen Stellen per Hand an den Parametern justiert werden muss. Wir formulieren die Methode nur für das Finden der kritischen Packungsdichte, denn das Finden der kritischen Temperatur verläuft analog. Sei also eine Packungsdichte $\eta^{(0)}$ bei fester Teilchenkonzentration x_1 und ein zugehöriger Strukturfaktor $S_{\alpha\beta}(q;\eta^{(0)})$ gegeben. Der Index (0) kennzeichne dabei die Nullte vorgegebene Packungsdichte. Dann wählen wir für die erste Iteration als Startwert für den NEP den Strukturfaktor. Ist bei Konvergenz der Iteration die Norm $||f_{\alpha\beta}(q;\eta^{(0)})|| > 10^{-2}$, so befindet sich das System noch im Glas und die Packungsdichte wird um δ verringert: $\eta^{(1)} = \eta^{(0)} - \delta$. Der Startwert für den NEP ist nun $f^{(0)}_{\alpha\beta}(q;\eta^{(1)}) = f_{\alpha\beta}(q;\eta^{(0)})$. Der zugehörige Strukturfaktor $S_{\alpha\beta}(q;\eta^{(1)})$ wird interpoliert. So verfahren wir, bis schließlich ein Intervall für die kritische Packungsdichte angegeben werden kann: $\eta_1 \leq \eta_c \leq \eta_2$. Dann werden in diesem Intervall weitere Strukturfaktoren mit dem LM-Algorithmus berechnet, die Schrittweite δ angepasst und analog ein kleineres Intervall gesucht.

7. Resultate der Nichtergodizitätsparameter

In diesem Kapitel werden wir die Resultate der NEP für die monodispersen Systeme präsentieren. Mit Hilfe dieser Resultate lässt sich der Glasübergang der Systeme untersuchen. Wir werden für das System monodisperser harter Scheiben in zwei Dimensionen die kritische Packungsdichte des Glasüberganges angeben. Für das System der monodispersen dipolaren harten Scheiben geben wir eine kritische Glasübergangskurve an, die zu jeweils fester Packungsdichte eine zugehörige kritische Temperatur angibt. Wie bereits im vorigen Kapitel erwähnt reichte die vorhandene Rechenzeit jedoch leider nicht mehr aus, um die NEP der binären Systeme im Rahmen dieser Diplomarbeit zu untersuchen.

Da wir die diskretisierten Impulse in der Form (6.1) in diesem Kapitel nicht explizit benötigen, bezeichnen wir den maximalen Impuls nun mit $Q := q_{N-1}$.

7.1. Monodisperse harte Scheiben

Zunächst werden wir die monodispersen harten Scheiben betrachten. Experimentell ist für dieses System kein Glasübergang beobachtet worden, denn die Glasbildung harter Teilchen wird erst durch Polydispersität hervorgerufen. Da jedoch $S(q)$ leicht polydisperser Systeme recht gut durch $S(q)$ monodisperser Systeme approximiert wird, macht es Sinn das monodisperse Modell trotzdem zu untersuchen. Dabei ist auch von Interesse, welcher wichtigen Unterschiede zwischen zwei- und dreidimensionalen Systemen existieren. Da pyhsikalischer Parameter dieses Systems die Packungsdichte ist, diskutieren wir zunächst die NEP bei verschiedenen Packungsdichten und geben dann die kritische Packungsdichte η_c für den Glasübergang an. Weiter werden wir die Abhängigkeit der kritischen NEP und der kritischen Packungsdichte von den statischen Strukturfaktoren aus verschiedenen Näherungen (hier PY und MHNC) diskutieren. Am Ende des Abschnittes stellen wir die Unterschiede zu den dreidimensionalen harten Kugeln heraus.

Die Abhängigkeiten des Algorithmus zur Iteration der NEP wurden bereits (siehe Kapitel 6) diskutiert. Wir wählen für die harten Scheiben die Diskretisierung so, dass $N := 500$ und $Q := 50$.

NEP für verschiedene Packungsdichten Betrachten wir zunächst die Abhängigkeit der NEP von der Packungsdichte: $f(q;\eta)$. Nach Abschnitt 5.6 ist zu erwarten, dass der NEP bei Packungsdichten nahe des Glasüberganges eine Wurzelabhängigkeit zeigt:

$$f_q = \begin{cases} f_q^c + c_q\sqrt{\eta - \eta_c} & \eta > \eta_c \\ 0 & \eta < \eta_c \end{cases}$$

Diese Abhängigkeit des NEP überträgt sich insbesondere auf dessen Norm $||f_q(\eta)||$, was in Abb. 7.1 dargestellt ist. Die Norm ist dabei definiert als

$$||f_q|| := \sqrt{\sum_{i=0}^{N-1} |f(q_i)|^2}$$

Der Index q an f_q bedeutet dabei, dass die Norm bezüglich des Impulses gebildet wurde. Es wurde eine Funktion der Form $j(\eta) = j^c + k\sqrt{\eta - \eta_c}$ im Intervall $0.7 \leq \eta \leq 0.72$ angefittet. Dabei entspricht $||f_q(\eta)|| =: j(\eta)$, $||f_q^c|| =: j^c$ und $||c_q|| =: k$. Es ergibt sich für die Fitparameter: $j^c \approx 5.22 \pm 0.1$, $k \approx 33.41 \pm 0.78$, $\eta_c \approx 0.7042 \pm 10^{-4}$ bei einem reduzierten $\chi^2 < 10^{-4}$. Wie zu erwarten ergibt sich auch für die zweidimensionalen harten Scheiben die Abhängigkeit vom kritischen Parameter über eine Wurzel.

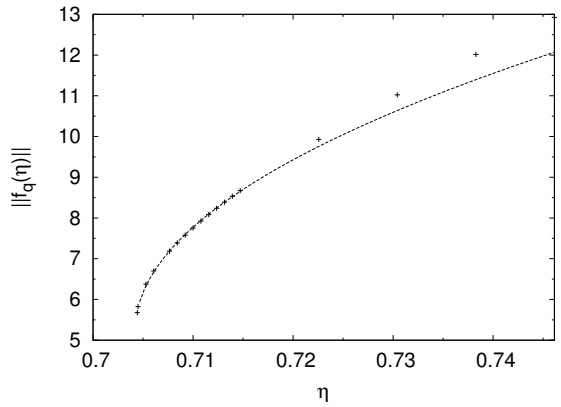

Abbildung 7.1.: $||f_q(\eta)||$ für monodisperse harte Scheiben bei Packungsdichten nahe dem Glasübergang

Betrachten wir nun den NEP selbst bei verschiedenen Packungsdichten $\eta \geq \eta_c$. In Abb. 7.2 sind einige NEP bei verschiedenen Packungsdichten mit ihren zugehörigen Strukturfaktoren aus PY dargestellt. Auch dort lässt sich die zuvor diskutierte Abhängigkeit des NEP von der Packungsdichte ablesen: Für $\eta_1 > \eta_2$ gilt offensichtlich für jeden Impuls q: $f(q;\eta_1) > f(q;\eta_2)$. Weiter ist für die hohe Packungsdichte $\eta = 0.746$ zu erkennen, dass der NEP langreichweitiger und größer wird. Das ist physikalisch auch verständlich, denn im Extremfall sehr hoher Packungsdichten muss der NEP gegen eins konvergieren [Göt89]. Deutlich zu erkennen ist in Abb. 7.2 auch, dass der erste Peak beim mittleren Impuls $\bar{q}$ des NEP nahe bei eins bleibt, das heißt $S(\bar{q};t=\infty) \approx S(\bar{q})$. Betrachten wir weiter die zugehörigen Strukturfaktoren, so wird klar, dass sehr kleine Unterschiede zwischen ihnen sehr verschiedene NEP liefern. Das bestätigt aber insbesondere, dass allein am Strukturfaktor nicht abgelesen werden kann, ob sich ein System in der Glasphase befindet oder nicht. Dass diese starken Unterschiede in den NEP nicht auf eine zu kleine Anzahl Stützstellen zurückzuführen sind, wurde in Abschnitt 6.3 diskutiert.

Weiter können aus dem NEP der Strukturfaktor $S(q;t=\infty) = f(q) \cdot S(q)$ und die zugehörige Paarverteilungsfunktion $g(r;t=\infty) = 1 + \mathcal{F}\left[\frac{S(q;t=\infty)-1}{\eta \cdot 4/\pi}\right]$ für den Langzeitlimes errechnet werden. In Abb. 7.3 geben wir die beiden für eine Packungsdichte von $\eta = 0.707$ an. In der Paarverteilungsfunktion sind kaum Änderungen zu erkennen[1]. Insbesondere sind $g(r)$ und $g(r;t=\infty)$ in Phase; das heißt aber, dass die mittleren Abstände der Teilchen untereinander gleich bleiben. Im Strukturfaktor $S(q;t=\infty)$ hingegen ist ein Unterschied zum Strukturfaktor bei $t=0$ ersichtlich. $S(q;t=\infty)$ fällt mit steigendem

[1] Die kleinen Abweichungen resultieren aus der für die FBT nötigen Interpolation, denn $S(q;t=\infty)$ wird aus dem äquidistanten $f(q)$ berechnet.

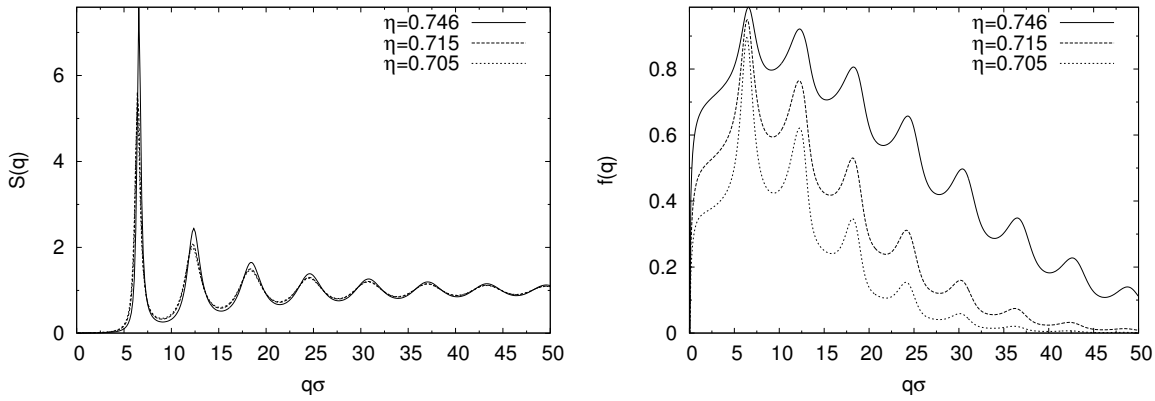

Abbildung 7.2.: $S(q)$ und $f(q)$ für monodisperse harte Scheiben bei verschiedenen Packungsdichten $\eta = 0.746$, $\eta = 0.715$ und $\eta = 0.705$

Impuls ab und es gilt $\forall q : S(q;t) \leq S(q)$. Letzteres resultiert daraus, dass für den NEP immer $f(q) \leq 1$ $\forall q$ gilt. Diese Tatsache kann numerisch bestätigt werden (siehe Abb. 7.2) und ist analytisch gezeigt in [Göt89]. Weiter ist $S(q)$ in Konsistenz zur Paarverteilungsfunktion in Phase mit $S(q;t=\infty)$. Damit ist der NEP monodisperser harter Scheiben ebenfalls in Phase mit dem Strukturfaktor.

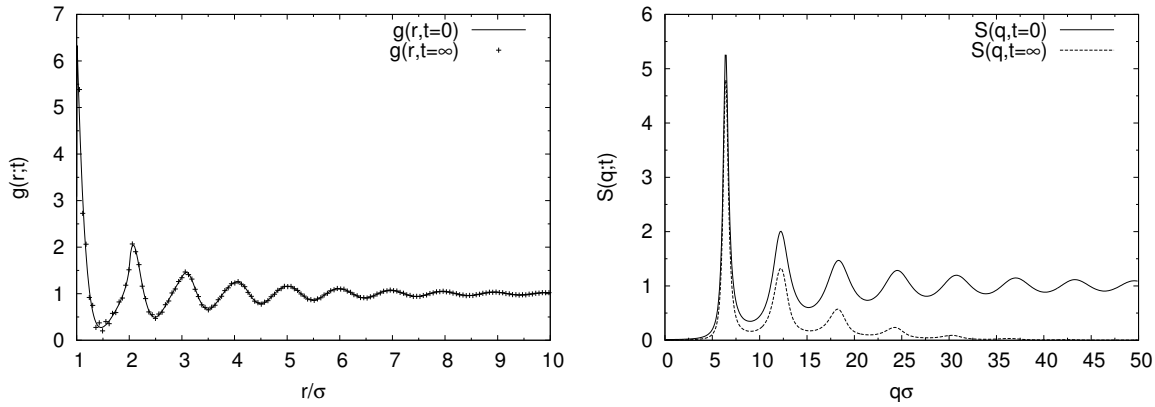

Abbildung 7.3.: $g(r;t)$ und $S(q;t)$ für monodisperse harte Scheiben bei $\eta = 0.707$ für $t = 0$ und Langzeitlimes $t = \infty$.

Kritische Packungsdichte aus PY Wir geben nun die kritische Packungsdichte, welche für statische Strukturfaktoren aus der PY-Approximation erhalten wurde, an. Es ergibt sich

$$\eta_c^{\mathrm{PY}} = 0.704 \pm 10^{-3} \tag{7.1}$$

Dabei bedeutet $\pm 10^{-3}$, dass die kritische Packungsdichte mittels Intervallschachtelung auf vier Stellen genau berechnet wurde. Der Einfluss der Diskretisierung in unserem Algorithmus wurde im vorigen Kapitel beschrieben und liegt ebenfalls im Bereich 10^{-3}.

Kritische Packungsdichte aus MHNC Wir wollen nun den Einfluss der statischen Strukturfaktoren auf die NEP diskutieren. Dazu ziehen wir Strukturfaktoren hinzu, die mittels modifizierter Hypernetted-Chain-Approximation (MHNC) [Lad73] gewonnen wurden [Bra06]. Zwischen den aus [Bra06] gegebenen Strukturfaktoren wurde mit bikubischen Splines (siehe Abschnitt A.2) interpoliert. In Abb. 7.4 sind zwei exemplarische Strukturfaktoren beider Näherungen dargestellt. Wie zu erkennen ist, hat der Strukturfaktor aus PY deutlichere Extrema. Das lässt bereits vermuten, dass die kritische Packungsdichte aus PY niedriger ist. In der Tat ergibt sich für die kritische Packungsdichte

$$\eta_c^{\mathrm{MHNC}} = 0.712 \pm 10^{-3} \tag{7.2}$$

Dabei bedeutet $\pm 10^{-3}$ wieder, dass die kritische Packungsdichte auf vier Stellen genau eingegrenzt wurde.

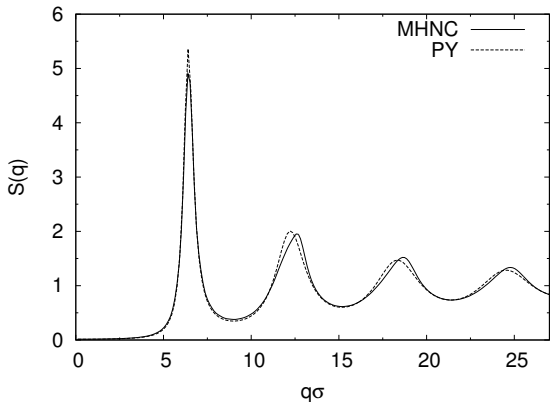

Abbildung 7.4.: $S(q)$ für monodisperse harte Scheiben bei $\eta = 0.707$ - Vergleich zwischen PY und MHNC

Vergleich zu harten Kugeln in drei Dimensionen Zunächst wollen wir die kritischen Packungsdichten für dreidimensionale harte Kugeln referieren. Für Strukturfaktoren aus PY-Approximation ergibt sich $\eta_c^{3\mathrm{d}} = 0.516$ [GFV04] und für Strukturfaktoren aus Molekular-Dynamik folgt $\eta_c^{3\mathrm{d}} = 0.546$ [GFV04]. Es zeigt sich also ein deutlicher Unterschied der kritischen Packungsdichten zwischen zwei und drei Dimensionen.

Die Ursache dafür, dass in zwei Dimensionen die Packungsdichte für einen Glasübergang wesentlich höher als in drei Dimensionen sein muss, liegt in den verschiedenen Packungseffekten. Rufen wir uns dazu den Vergleich der Statik zwei- und dreidimensionaler harter Teilchen in Abschnitt 4.1 ins Gedächtnis. Wir hatten dort in Abb. 4.4 die Paarverteilungsfunktion der zwei- und dreidimensionalen Systeme bei gleicher Packungsdichte betrachtet und festgestellt, dass die mittleren Abstände der Systeme aufgrund der Packungseffekte verschieden sind. Weiter hatten wir in Abb. 4.5 den Strukturfaktor der beiden Systeme bei gleichem mittleren Abstand $\frac{\bar{r}}{\sigma} = 1.02$ dargestellt. Die aus diesem mittleren Abstand resultierenden Packungsdichten von $\eta^{3\mathrm{d}} = 0.52$ und $\eta^{2\mathrm{d}} = 0.71$ zeigen, dass der Glasübergang der beiden Systeme bei etwa gleichem mittleren Abstand stattfindet. Der mathematische Grund dafür, dass die zweidimensionalen harten Scheiben erst bei einer wesentlich höheren Packungsdichte einen Glasübergang aufweisen, liegt im Phasenvolumenfaktor des Relaxationskernes (5.46): In zwei Dimensionen geht bei den Integrationen über p und k jeweils nur ein linearer Impulsfaktor ein, während es in drei Dimensionen bereits ein quadratischer Impulsfaktor ist. Damit muss

aber in zwei Dimensionen der Strukturfaktor für einen Glasübergang wesentlich höhere Maxima haben, was in Abb. 4.5 bestätigt werden kann.

7.2. Monodisperse dipolare harte Scheiben

In diesem Abschnitt werden wir den Glasübergang der monodispersen dipolaren harten Scheiben diskutieren. Da die Systemparameter nun die Packungsdichte und die Temperatur sind, ist zu erwarten, dass bei Packungsdichten $\eta < \eta_c^{\text{h.S.}}$ unterhalb der kritischen Packungsdichte der reinen harten Scheiben ein temperaturgetriebener Glasübergang zu finden sein wird. Je geringer die Packungsdichte η ist, desto stärker muss die dipolare Wechselwirkung sein, um einen Glasübergang zu beobachten. Wir haben in der vorliegenden Arbeit zunächst den Glasübergang der monodispersen dipolaren harten Scheiben nur bei hohen Packungsdichten untersucht, denn je niedriger die Packungsdichte ist, desto größer ist die numerische Anstrengung. Auf diese numerische Problematik gehen wir weiter unten ein.

Einfluss der dipolaren Wechselwirkung auf die NEP Betrachten wir vorab die NEP der monodispersen dipolaren harten Scheiben bei den Packungsdichten $\eta = 0.39$ (siehe Abb. 7.5) und $\eta = 0.67$ (siehe Abb. 7.6) für verschiedene Temperaturen. Wie in beiden Graphen zu sehen ist, wächst der NEP mit sinkender Temperatur. Das heißt für ein beliebiges festes $q\sigma$ und $T_1 > T_2$ gilt: $f(q;T_1) < f(q;T_2)$. Das entspricht auch der oben genannten Erwartung, dass für das System mit steigender dipolarer Wechselwirkungsenergie ein Glasübergang zu beobachten sein wird. Analog zu den harten Scheiben werden die NEP mit sinkender Wechselwirkungsstärke kleiner. Bleiben wir bei der Temperaturabhängigkeit der NEP und vergleichen die NEP bei den beiden Packungsdichten. Offensichtlich ist die Temperaturabhängigkeit des NEP bei der höheren Packungsdichte geringer. Das ist physikalisch auch verständlich, da bei hohen Packungsdichten der Glasübergang dichtegetrieben ist. Je niedriger die Packungsdichte ist, desto stärker dominiert also die dipolare Wechselwirkung (vgl. Abb. 7.5 und Abb. 7.6).

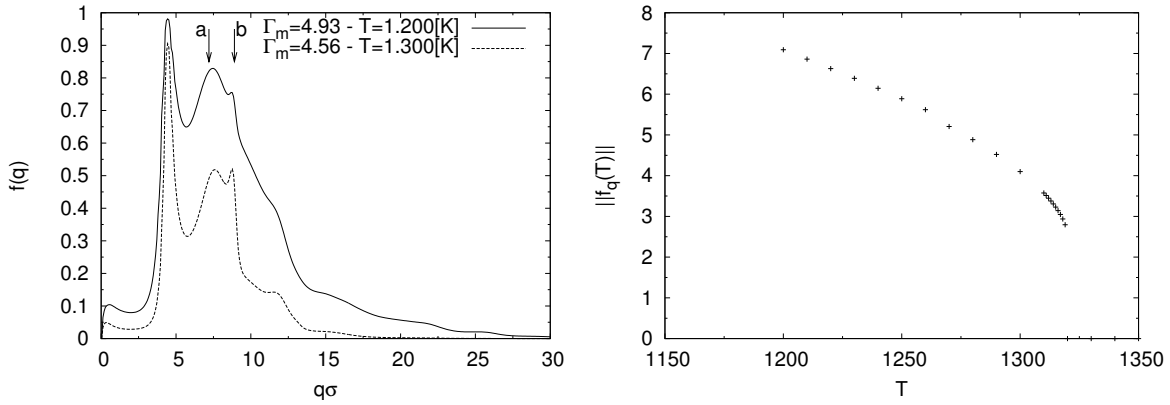

Abbildung 7.5.: NEP monodisperser dipolarer harter Scheiben und dessen Norm bei $\eta = 0.39$; die Pfeile an *a* und *b* kennzeichnen das Doppelmaximum - Variation der Temperatur

Weiter äußert sich die Dichteabhängigkeit der NEP darin, dass die Reichweite mit sinkender

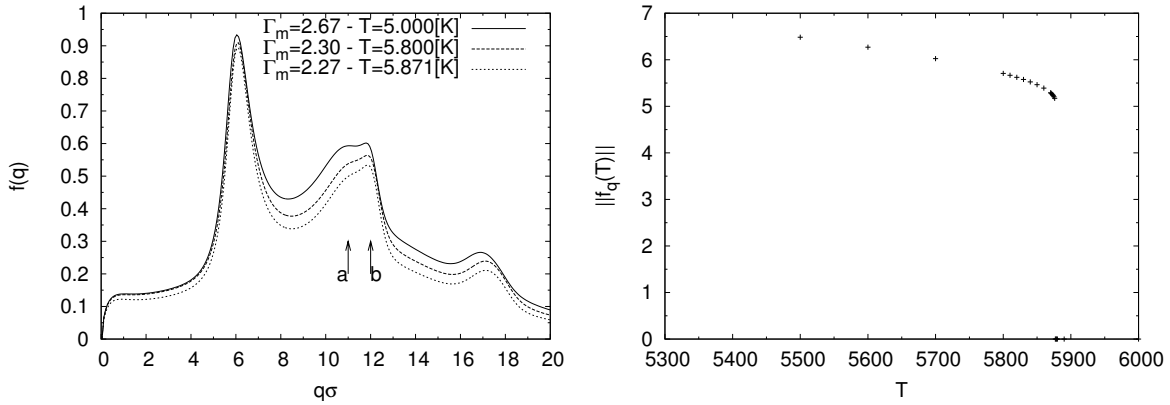

Abbildung 7.6.: NEP monodisperser dipolarer harter Scheiben und dessen Norm bei $\eta = 0.67$; die Pfeile an a und b kennzeichnen das Doppelmaximum - Variation der Temperatur

Packungsdichte kleiner wird. Das heißt, die Maxima verschieben sich mit geringerer Packungsdichte zu kleineren $q\sigma$. Diese Tatsache resultiert direkt aus den zugehörigen statischen Strukturfaktoren (Abb. 7.10) und führt zu numerischen Schwierigkeiten: Da mit kleinerer Packungsdichte der Strukturfaktor immer kurzreichweitiger und das erste Maximum gleichzeitig wegen der steigenden dipolaren Wechselwirkung immer höher wird (siehe Abb. 7.10), bestimmt dieses mehr und mehr den Glasübergang des Systems. Gleichzeitig wächst jedoch mit stärkerer dipolarer Wechselwirkung auch das erste Maximum im stetigen Korrelator $\gamma(r)$, der in (2.8) definiert wurde (siehe Abb. 7.7). Das heißt aber, dass gleichzeitig im Orts- und Impulsraum bei kleinen Argumenten die Anzahl der Stützstellen hoch sein muss. Weil aber wegen der Diskretisierung (3.1) R und Q nicht unabhängig voneinander gewählt werden können, bleibt nur die Möglichkeit M zu erhöhen, um beide Peaks möglichst gut darzustellen. Wir haben für die Glasübergangskurve die Statik mit $M = 2000$ gelöst[2].

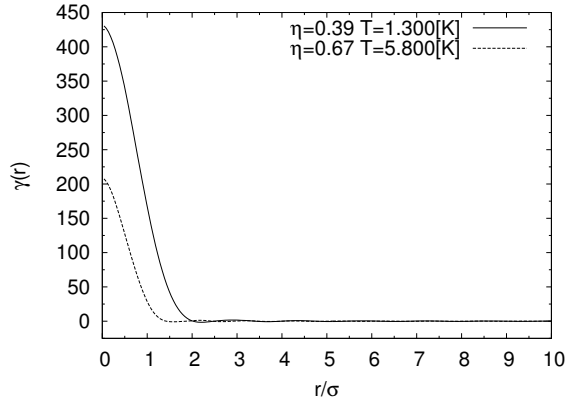

Abbildung 7.7.: Korrelator $\gamma(r)$ für monodisperse dipolare harte Scheiben bei $\eta = 0.39$ und $T = 1300[\mathrm{K}]$ ($\Gamma_\mathrm{m} = 4,93$) bzw. $\eta = 0.67$ und $T = 5800[\mathrm{K}]$ ($\Gamma_\mathrm{m} = 2.3$)

Nun wollen wir auf das Doppelmaximum des zweiten Peaks im dipolaren NEP eingehen, das wir in den Graphen jeweils durch a und b gekennzeichnet haben. Vorab sei erwähnt, dass aufgrund seiner

[2]Das führt auf den uns zur Verfügung stehenden Rechnern zu einem Zeitaufwand von knapp drei Stunden pro statischem Korrelatorensatz $S(q)$, $g(r)$.

Unabhängigkeit von der Diskretisierung (Abschnitt 6.3) weitgehend ausgeschlossen werden kann, dass es sich bei diesem Doppelpeak um ein numerisches Artefakt handelt.

Betrachten wir zunächst den NEP reiner harter Scheiben in Abb. 7.8 für die Packungsdichten $\eta = 0.67$ und $\eta = 0.7$. Dort ist zu erkennen, dass auch in der Flüssigkeitsphase, in welcher der NEP numerisch nahezu auf Null abgefallen ist, die Extremalstellen erhalten bleiben. Das heißt, das sich die Extremstellen des NEP monodisperser harter Scheiben mit der Packungsdichte kaum ändern (siehe auch Abb. 7.2), selbst wenn die kritische Packungsdichte unterschritten wurde. Vergleichen wir nun also den NEP der harten Scheiben im Glas bei $\eta = 0.7$ mit den NEP dipolarer harter Scheiben bei einer Packungsdichte $\eta = 0.67$ im Glas mit $\Gamma_\mathrm{m} = 2.29$ und $\Gamma_\mathrm{m} = 2.65$ in Abb. 7.8. Betrachten wir nun die Struktur des zweiten Peaks der NEP, so ist ersichtlich, dass für die dipolaren harten Scheiben ein Maximum bei b nahe dem der reinen harten Scheiben auftritt. Zusätzlich hat der dipolare NEP ein Maximum a, das bei den reinen harten Scheiben nicht vorhanden ist. Um dieses Maximum a weiter zu untersuchen, betrachten wir nochmals die NEP bei den zwei Packungsdichten $\eta = 0.39$ (Abb. 7.5) und $\eta = 0.67$ (Abb. 7.6) für verschiedene Temperaturen. Vergleichen wir nun das Wachstum des Maximums a mit dem des Maximums b jeweils in Abhängigkeit von der Temperatur, so wächst a stärker mit der Temperatur, als b. Das wäre auch zu erwarten, weil ersteres ein zusätzlicher Effekt des dipolaren Systems ist. Weiter ist offensichtlich nahe der kritischen Temperatur das Maximum a bei geringerer Packungsdichte größer. Das deutet abermals darauf hin, dass dieses Maximum aus der dipolaren Wechselwirkung resultiert, die bei den geringen Packungsdichten immer dominanter wird. Es scheint also, als äußere sich in diesem Doppelmaximum ein Konkurrenzprozess zwischen dichte- und temperaturgetriebenem Glasübergang. Um diesen Effekt jedoch hinreichend zu verstehen, muss das dipolare System noch für geringere Packungsdichten untersucht werden (siehe Kapitel 8). Ziehen wir zum Vergleich jedoch weiter die Strukturfaktoren reiner harter Scheiben aus Abb. 4.1 hinzu, so ist dort zu erkennen, dass die Maximalstellen des Strukturfaktors weniger mit der Packungsdichte variieren, als die Maximalstellen der Strukturfaktoren der dipolaren harten Scheiben. Das heißt aber, dass es sich bei diesem Doppelpeak nicht um eine triviale Abhängigkeit vom reinen Harte-Scheiben-Anteil handelt. Ein ähnliches Phänomen, dass die MCT einen NEP in der Glasphase liefert, der eine Struktur aufweist, die in dem statischen Strukturfaktor des Systems nicht zu erkennen ist, wurde bereits für molekulare Flüssigkeiten beobachtet [CS03]. Dort wurde versucht, diesen Peak im Strukturfaktor des Molekülschwerpunktes durch eine indirekte Wechselwirkung (wie für die binären Systeme in Abschnitt 4.3 beschrieben) zu erklären.

Wie bereits bei den reinen harten Scheiben, geben wir noch den Langzeitlimes von $S(q;t)$ (Abb. 7.9) im Vergleich zum Zeitpunkt $t = 0$ (Abb. 7.10) an. Genau wie dort fallen die $S(q;t = \infty) = f(q) \cdot S(q)$ in Konsistenz zum jeweiligen NEP auf Null ab. Interessant ist hierbei aber im direkten Vergleich des Zeitpunktes $t = 0$ und $t = \infty$, dass der strukturelle Verlauf des Langzeitlimes anders ist. Zum Zeitpunkt $t = 0$ ist an den mit den Pfeilen gekennzeichneten Stellen des Doppelpeaks im NEP nichts zu sehen. Im Langzeitlimes von $S(q;t)$ taucht dann jedoch die zusätzliche Struktur auf. Es sei an dieser Stelle erwähnt, dass sich die Paarverteilungsfunktion genau wie bei den monodispersen reinen harten Scheiben im Langzeitlimes fast nicht ändert. Insbesondere gibt es keinen signifikanten strukturellen Unterschied zwischen $g(r;t = 0)$ und $g(r;t = \infty)$, der die Struktur im zweiten Maximum erklären würde.

Glasübergangskurve Abschließend geben wir die kritischen Temperaturen T_c zu verschiedenen Packungsdichten η_c in einem Graphen an. In Abb. 7.11 ist also die Glasübergangskurve für das System der monodispersen dipolaren harten Scheiben dargestellt. Es ergibt sich der für große Packungsdichten

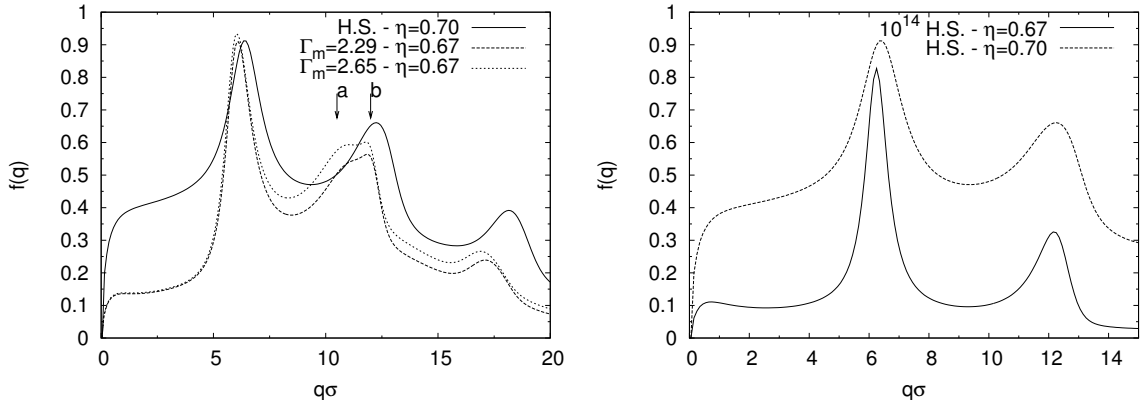

Abbildung 7.8.: NEP monodisperser harter Scheiben bei $\eta = 0.7$ und monodisperser dipolarer harter Scheiben bei $\eta = 0.67$ und $\Gamma_m = 2.29$ bzw. $\Gamma_m = 2.66$ im Vergleich; *a* und *b* kennzeichnen den Einfluss der dipolaren Wechselwirkung - Vergleich der Struktur der NEP monodisperser harter Scheiben bei $\eta = 0.7$ und $\eta = 0.67$ wobei der NEP bei $\eta = 0.67$ mit 10^{14} skaliert wurde

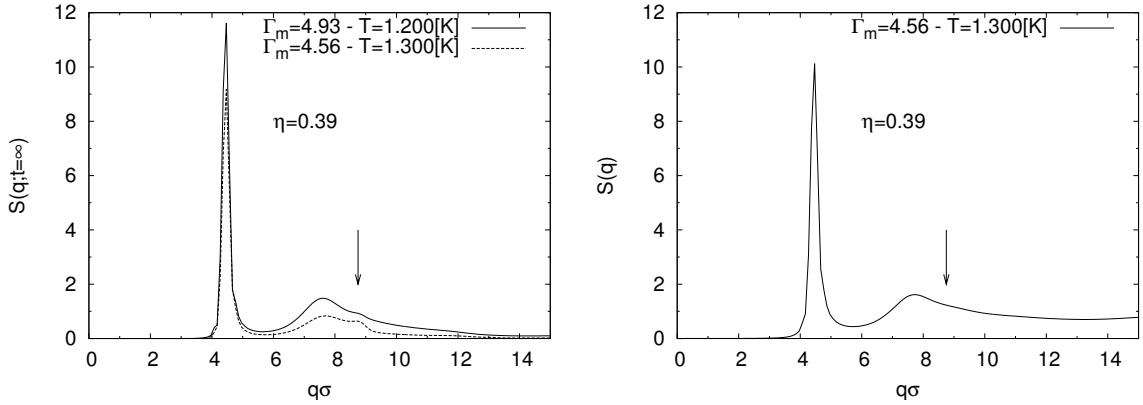

Abbildung 7.9.: $S(q;t)$ monodisperser dipolarer harter Scheiben bei $\eta = 0.39$ zum Zeitpunkt $t = 0$ und $t = \infty$ bei verschiedenen Temperaturen; die Pfeile kennzeichnen jeweils das zweite Maximum des NEP

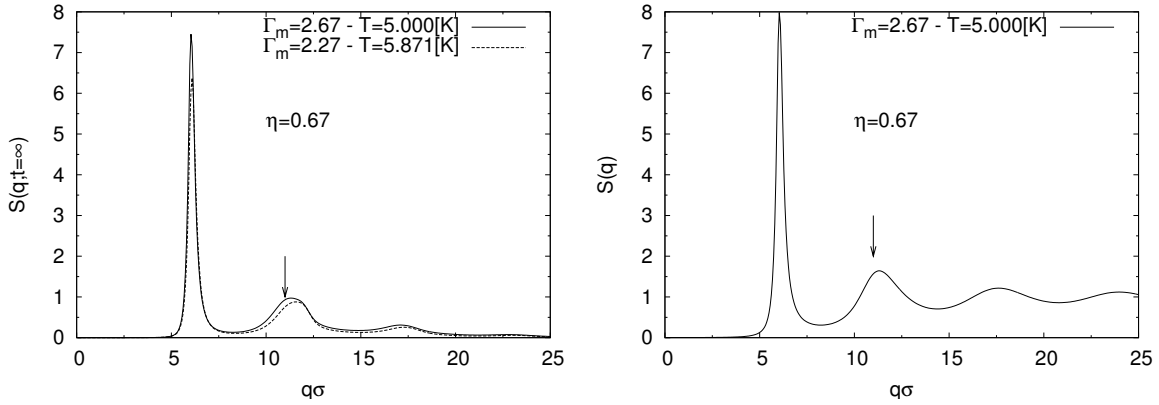

Abbildung 7.10.: $S(q;t)$ monodisperser dipolarer harter Scheiben bei $\eta = 0.67$ zum Zeitpunkt $t = 0$ und $t = \infty$ bei verschiedenen Temperaturen; die Pfeile kennzeichnen jeweils das zweite Maximum des NEP

erwartete Verlauf: Mit sinkender Packungsdichte fällt die kritische Temperatur ab, bzw. die kritische mittlere magnetische Wechselwirkungsenergie steigt. Je näher die Packungsdichte des Systems von unten an die kritische Packungsdichte $\eta_c = 0.7044$ der reinen harten Scheiben rückt, desto größer wird die kritische Temperatur. In Konsistenz zu den reinen harten Scheiben ergibt sich für die dipolaren harten Scheiben im Limes hoher Temperaturen (bei $T = 10^{23}[\mathrm{K}]$) bei $\eta_c = 0.7044$ ein Glasübergang. Betrachten wir nun den Verlauf der Glasübergangskurve etwas genauer. Nach (5.49) ist zu erwarten, dass $T_c(\eta_c) \sim \eta_c^{3/2}$ bzw. $\Gamma_{\mathrm{m}}^{\mathrm{c}}(\eta_c) = \mathrm{const}$. Das ist jedoch in Abb. 7.11 nicht zu beobachten, denn die dargestellten Packungsdichten η sind $\eta > 10^{-3}$. Damit ist Γ_m noch nicht der das System bestimmende Parameter (Abb. 4.6).

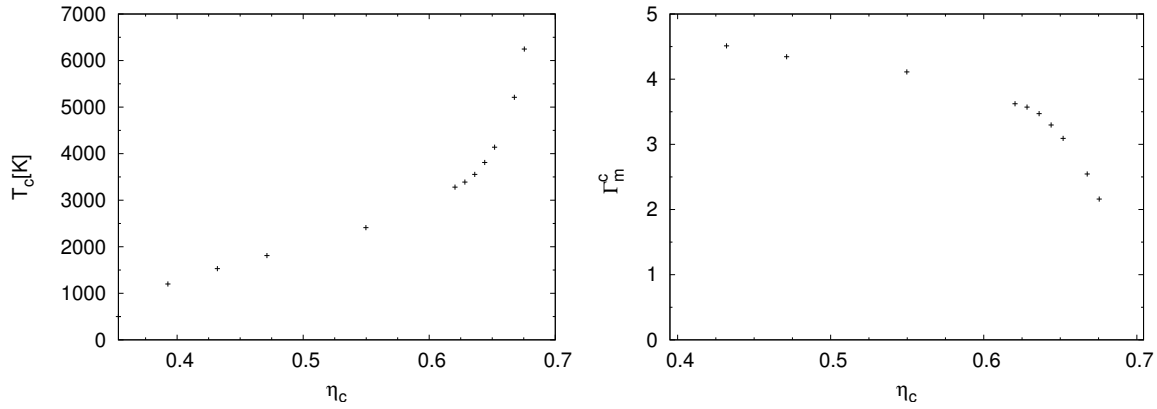

Abbildung 7.11.: Kritische Temperatur T_c und kritische mittlere magnetische Wechselwirkungsenergie $\Gamma_{\mathrm{m}}^{\mathrm{c}}$ in Abhängigkeit von der kritischen Packungsdichte η_c für monodisperse dipolare harte Scheiben

8. Fazit und Ausblick

Fazit Ziel der vorliegenden Diplomarbeit war es, die Gültigkeit der Modenkopplungstheorie in zwei Dimensionen zu überprüfen. Aus dem Experiment [HKM05] motiviert ist dabei insbesondere die Untersuchung des Glasüberganges eines binären Systems dipolarer harter Scheiben (Kapitel 1). Wir haben dazu die Modenkopplungstheorie für monodisperse und binäre Systeme formuliert (Kapitel 5) und die NEP für monodisperse Systeme untersucht (Kapitel 7). Die NEP der binären Systeme haben wir mangels Rechenzeit nicht mehr analysieren können.

Um den Glasübergang dieser monodispersen und binären zweidimensionalen Systeme jedoch zu untersuchen, mussten zunächst deren statische Korrelatoren numerisch berechnet werden. Da die bisher bekannten Algorithmen weder die nötige Stabilität ([HM86], Abschnitt 3.3), noch die nötige Geschwindigkeit ([Gil79], Abschnitt 3.4) besaßen, haben wir mit dem LM-Algorithmus eine Methode entwickelt, die statischen Korrelatoren zweidimensionaler Systeme nahe des Glasüberganges zu berechnen (Abschnitt 3.5).

Gleichzeitig mit der Entwicklung dieses Algorithmus haben wir Unterschiede zwischen der PY- und HNC-Approximation bezüglich Stabilität und konvergierter Lösungen untersucht, und strukturelle Eigenschaften der Korrelatoren der vier Modellsysteme diskutiert (Kapitel 4). Dabei sind wir insbesondere auf Unterschiede zu dreidimensionalen Systemen monodisperser [SH70] und binärer [**?**] harter Kugeln eingegangen. Das zweidimensionale System hat bei gleicher Packungsdichte eine geringere Struktur, als das dreidimensionale, da der mittlere Abstand in zwei Dimensionen größer ist. Für gleichen mittleren Abstand ist die Struktur in zwei Dimensionen wesentlich höher. Die Ursache für diese Unterschiede liegt in den verschiedenen Packungseffekten in zwei und drei Dimensionen. Weiter haben wir den Einfluss der Binärität auf die Korrelatoren in zwei Dimensionen untersucht und herausgefunden, dass ein dipolares Harte-Scheiben-Potential der Form (4.13) bei kleinen Packungsdichten keine Unterstrukturierung der Korrelatoren hervorruft, wie es das reine Harte-Scheiben-Potential tut. Die Statik abschließend haben wir die Paarverteilungsfunktion der binären dipolaren harten Scheiben aus PY bei $\Gamma_\mathrm{b} = 5.71$ und $\Gamma_\mathrm{b} = 11.43$ mit experimentellen Resultaten [HKM05] bei $\Gamma_\mathrm{b} = 78$ verglichen (Abb. 4.18) und dabei zwei Klassen von Abweichungen festgestellt: Erstens ist in den numerischen Resultaten keine Unterstruktur der Korrelatoren aufgrund der Binärität wie bei den binären reinen harten Scheiben zu erkennen. Experimentell ergeben sich jedoch Nebenmaxima, die aus der Binärität des Systems resultieren müssen. Zweitens ist die Reichweite der Korrelatoren aus der Numerik wesentlich größer, woraus wir folgern, dass dem Experiment keine reine dipolare Wechselwirkung zugrunde liegt. Diese Vermutung wird untermauert von [Fuc06], wonach es experimentell problematisch ist, die Scheiben exakt in einer Ebene zu halten.

Wir haben die Modenkopplungstheorie auf binäre Systeme in zwei Dimensionen erweitert (Abschnitt 5.5) und die Numerik entwickelt, um die Nichtergodizitätsparameter zu berechnen (Kapitel 6), mit denen sich die kritischen Parameter des Glasüberganges bestimmen lassen. Für die zweidimensionalen monodispersen harten Scheiben haben wir eine kritische Packungsdichte für den Glasübergang mit PY ($\eta_c^\mathrm{PY} = 0.7044$) und MHNC ($\eta_c^\mathrm{MHNC} = 0.712$) bestimmt (Abschnitt 7.1). Im Vergleich zu den

dreidimensionalen harten Kugeln ($\eta_c = 0.516$ [GFV04]) ist die kritische Packungsdichte der zweidimensionalen harten Scheiben dabei wegen der dimensionsabhängigen Packungseffekte deutlich höher. Weiter haben wir den Glasübergang der zweidimensionalen monodispersen dipolaren harten Scheiben untersucht (Abschnitt 7.2). Wir haben die Glasübergangskurve $T_c(\eta_c)$ bzw. $\Gamma_m^c(\eta_c)$ für $0.4 \leq \eta_c \leq 0.67$ bestimmt (Abb. 7.11), die bei diesen hohen Packungsdichten der physikalischen Erwartung gerecht wird. Dabei ist die kritische mittlere magnetische Wechselwirkungsenergie $\Gamma_m^c = 2, ..., 5$ in diesem Packungsdichtebereich noch sehr abhängig von der Packungsdichte, da der mittlere Abstand der Scheiben $\bar{r} \approx \sigma$ ist und somit der Harte-Scheiben-Anteil des Potentials dominiert. In den zugehörigen Nichtergodizitätsparametern (Abb. 7.8) haben wir überraschend einen Doppelpeak im zweiten Maximum entdeckt, der sich aus einem dipolaren Anteil und einem Harte-Scheiben-Anteil zusammensetzt.

Ausblick Da der größte Aufwand der Arbeit darin bestand, die Numerik der Statik in den Griff zu bekommen, ist mit dieser Diplomarbeit gewissermaßen erst der Grundstein für eine systematische Untersuchung der zweidimensionalen Glasübergänge mit der Modenkopplungstheorie gelegt.

Formulieren wir zunächst den Ausblick für die bereits erzielten Resultate. Der Vergleich der statischen Korrelatoren mit dem Experiment kann durch Abändern des dipolaren Potentials weiter getrieben werden. Die Frage ist dabei, ob es durch Ergänzen des Potentials um einen attraktiven Anteil numerisch möglich ist, eine bessere Übereinstimmung mit der experimentellen Paarverteilungsfunktion zu erzielen. Und inwiefern eine binäre Unterstruktur in den Korrelatoren erzeugt werden kann. Weiter ist dann die Frage, ob die Idealisierung, dass sich die Teilchen alle in einer Ebene befinden, aufgegeben werden muss.

Sehr interessant ist weiter der Verlauf der kritischen Glasübergangskurve der monodispersen dipolaren harten Scheiben bei niedrigeren Packungsdichten, denn dort dominiert die reine dipolare Wechselwirkung. Zu erwarten wäre in diesem Bereich, dass der Glasübergang bei einem konstanten Γ_m stattfindet, das heißt die kritische Temperatur T_c sollte als Funktion der Packungsdichte mit $T_c(\eta_c) \sim \eta_c^{3/2}$ verlaufen. Sobald die kritische Glasübergangskurve gefunden ist, kann ein Vergleich zu den dreidimensionalen dipolaren Kugeln [SS98] gezogen werden.

In diesem Zusammenhang stellt sich auch die Frage nach dem Doppelpeak in den NEP der monodispersen dipolaren harten Scheiben. Kann die Tendenz bestätigt werden, dass Peak *b* mit sehr kleiner Packungsdichte verschwindet? Weiter ist die Frage, ob in rein dipolaren monodispersen Scheiben nur Peak *a* zu sehen ist.

Abschließend bleibt noch die Frage nach dem Glasübergangsverhalten der binären Systeme zu klären, die im Rahmen dieser Diplomarbeit leider mangels Rechenzeit nicht mehr geklärt werden konnte.

A. Weitere Algorithmen

A.1. Einlesen unterbestimmter[1] Funktionen

Im Zuge der Wahl experimenteller Daten als Startwert für die Iterationsalgorithmen war es notwendig, Kurven mit $M^{(exp)}$ äquidistanten Stützstellen einzulesen. Ist nun $M^{(exp)} < M$, so müssen die Daten interpoliert werden. An dieser Stelle haben wir auf die FBT mit einem Tiefpassfilter zurückgegriffen.

Zur Unterscheidung zwischen eingelesener und gesuchter Funktion, indizieren wir erstere mit i und die gesuchte Funktion mit j. Aus einer gegebenen Datei werden zunächst $M^{(exp)}$ 2-Tupel der Form $\left(r_i^{(exp)}, f_i^{(exp)}\right)$ eingelesen. Gesucht ist nun die Funktion f_j an den M Stellen r_j. Dazu benötigen wir die Zuordnung $g : r_i \mapsto i$. Weil für die einzulesenden Daten $R^{(exp)} \leq R$ gilt, kann jeder eingelesene Funktionswert $f_i^{(exp)}$ anhand seines zugehörigen Ortes $r_i^{(exp)}$ einem f_j zugeordnet werden vermöge $j = argmin_{0\leq k\leq M-1}(|r_k - r_i^{(exp)}|)$. Setzt man weiter alle restlichen $f_j = 0$, so ergibt sich eine strubbelige Funktion $\vec{f} = (f_0, ..., f_{M-1})$, deren Fouriertransformierte $\vec{F}$ Anteile bei hohen Impulsen hat. Um diese Beträge zu verlieren, definieren wir einen Tiefpassoperator[2] $F_m^{'} = \mathcal{T}_m[F_m]$, der die aufgrund der Stückelung entstandenen hohen Impulse abschneidet. Bezeichnen wir mit $\vec{f} = (f_0 ... f_{M-1})$ und lassen den Tiefpassoperator nun wirken mit $\vec{f}^{'} = \mathcal{F}^{-1}\mathcal{T}(q)\mathcal{F}\,\vec{f}$, so ergibt sich die gewünschte glatte Funktion.

Algorithm 7 Einlesen unterbestimmter Funktionen

Require: $\exists$ INFILE: $(r_i^{(exp)}, f_i^{(exp)}) \quad \forall 0 \leq i \leq M^{(exp)-1}$
Require: $R^{(exp)} \leq R$
Require: $\exists (r_j, f_j = 0) \quad \forall 0 \leq j \leq M-1$

1: **repeat**
2: INFILE $\rightarrow \left(r_i^{(exp)}, f_i^{(exp)}\right)$
3: $j = argmin_{0\leq k\leq M-1}\left(|r_k - r_i^{(exp)}|\right)$
4: $f_j = f_i^{(exp)}$
5: **until** eof (INFILE)
6: $\vec{f}^{'} = \mathcal{F}^{-1}\mathcal{T}\,\mathcal{F}\,\vec{f}$

[1] Unterbestimmt heißt in diesem Falle, dass die Zahl der Stützstellen der eingelesenen Funktion kleiner als gewünscht ist.

[2] Wir haben $\mathcal{T}(q)$ als einen glatten Abfall von Q_1 zu Q_2 auf Null mittels $t(q) := \begin{cases} 1 & 0 \leq q \leq Q_1 \\ \frac{\arctan(10 - \frac{10q}{Q_2 - Q_1})}{\arctan(10)} & Q_1 < q < Q_2 \\ 0 & Q_2 \leq q \end{cases}$ gewählt und so mögliche Oszillationen bei großen r_j vermieden.

A.2. Interpolation mit bikubischen Splines

Angenommen eine beliebige Funktion $f(x,y)$ ist an den Stellen x_i, y_j mit $0 \leq i \leq M-1, 0 \leq j \leq N-1$ bekannt. Gesucht sei eben diese Funktion $f(x,y)$ an den Stellen $\tilde{x}_m, \tilde{y}_n$ mit $0 \leq m \leq \tilde{M}-1, 0 \leq n \leq \tilde{N}-1$. Damit es sich um ein Interpolationsproblem handelt, gelte weiter $x_0 \leq \tilde{x}_m \leq x_{M-1}$, $y_0 \leq \tilde{y}_n \leq y_{N-1}$. Dann kann die Funktion an den gesuchten Stellen mittels bikubischer Interpolation berechnet werden. Das hat den Vorteil, dass sich sowohl die Funktion selbst, als auch ihre Ableitung zwischen den bekannten Stellen stetig ändert. Für eine nähere Beschreibung des Interpolationsalgorithmus verweisen wir auf [WHP92]. Der von uns verwendete Algorithmus ist dieser Quelle entlehnt.

B. Listings und CD

Der Vollständigkeit halber sind im folgenden die wichtigsten Algorithmen im Quelltext (ISO-C99) angegeben. Es handelt sich dabei um den LM-Algorithmus und die Routine zur Iteration der monodispersen NEP. Um Platz einzusparen wurden dabei sämtliche Kommentare entfernt. Auf der beigefügten CD findet sich der kommentierte Quelltext dieser Programme und weiter die Gillan-Methode für monodisperse und binäre Systeme. Außerdem sind alle in dieser Arbeit enthaltenen Graphen als GNUPlot-Skripte mit zugehörigen Datensätzen enthalten. Eine PDF- und PS-Version der Arbeit ist ebenfalls auf der CD zu finden.

B.1. Levenberg-Marquardt-Algorithmus monodispers

lm_mono.c

```
// compile with: gcc -Wall -pedantic -funroll-loops -std=c99 -lgsl -lgslcblas -l
m -O3
#include <stdio.h>
#include <stdlib.h>
#define __USE_GNU 1
#define _GNU_SOURCE
#include <math.h>
long double j0l(long double x);
long double j1l(long double x);
#include <getopt.h>
#include <string.h>
#include <assert.h>
#include <stdarg.h>
#include <time.h>


#define __USE_GSL 1
#ifdef __USE_GSL
#include <gsl/gsl_sf_bessel.h>
#else
long double gsl_sf_bessel_zero_J0 (int number) {
        return M_PI1/4.0*(4.0*number-1.0) +
                1.0/(2.0*M_PI*(4.0*number-1.0)) -
                31.0/(6.0*powl(M_PI*(4.0*number-1.0),3.0)) +
                3779.0/(15.0*powl(M_PI*(4.0*number-1.0),5.0));
}
#endif

#define NDEBUG
#define SYSTEM 2
#define APPROXIMATION 1
#define STARTVALUE 1
//#define ASYMPTOTE 2
//#define DYN_COARSE
//#define DYN_COARSE_2
//#define PIC_AFTER
//#define DYN_BASIS
//#define DYN_BASIS_1
//#define NEW_BASIS
//#define SLOW_PICARD
#define BASIS_LINE


int out (const char *format, ...) {
        va_list arg;
        int done;

        va_start (arg, format);
        done = vfprintf (stdout, format, arg);
        fflush (stdout);
        va_end (arg);

        return done;
}

void die (const char *format, ...) {
        va_list arg;
        int done;

        va_start (arg, format);
        done = vfprintf (stdout, format, arg);
        fflush (stdout);
        va_end (arg);

        exit (-1);
}
```

lm_mono.c

```
typedef struct {
        long double *matrix;
        long double *ptr, *ptr1;
        unsigned int size;
} t_matrix;

t_matrix init_matrix (unsigned int size) {
        t_matrix matrix = {
                .size = size,
                .matrix = (long double *) calloc ((size_t)(size), (size_t)(sizeo
f (long double))),
                .ptr = NULL
        };
        if (!matrix.matrix)
                exit (-1);
        return matrix;
}

void cleanup_matrix (t_matrix matrix) {
        free (matrix.matrix);
}


long double *alloc_matrix_1 (int anz_1) {
        long double *m;
        int i;

        m = (long double *) malloc ((size_t) (anz_1 * sizeof (long double)));
        if (!m)
                die ("allocation failure in alloc_matrix_1");

        for (i = 0; i < anz_1; i++)
                m[i] = 0.0;

        return m;
}

long double **alloc_matrix_2(int anz_1,int anz_2) {
        int i;
        long double **m;

        m = (long double **) malloc((size_t) (anz_1 * sizeof(long double *)));
        if (!m)
                die ("allocation failure in alloc_matrix_2");
        for (i = 0; i < anz_1; i++)
                m[i] = alloc_matrix_1(anz_2);
        return m;
}

void free_matrix_1(long double *m) {
        free(m);
}

void free_matrix_2(long double **m,int anz_1) {
        int i;

        for (i = 0; i < anz_1; i++)
                free_matrix_1(m[i]);
        free(m);
}


typedef struct {
        long double *function;
        long double *coarse, *fine;
        long double **P_, **Q_;
        int gitter, nu;
        long double sum;
```

lm_mono.c

```
} t_composite;

typedef struct {
        long double *F_r, *F_q;
        int gitter;
        long double **bessel;
        long double cons;
} t_fbt;

typedef struct {
        long double *Cr, *Cq, *Gammar, *Gammaq;
        long double *r_;
        long double p;
        int gitter, nu;
        long double *mayer, *Vr;
        t_fbt fbt1, fbt2;
        t_composite comp, decomp;
} lm_data_type;

typedef struct {
        long double *coarse;
        long double *fine;
        int coarse_dim, fine_dim;
        int *pic, *nr;
        long double *r_;
        long double **P_;
} t_coarse_fine;

typedef struct {
        long double *Gamma_r, *Gamma_q;
        long double *C_r, *C_q;
        long double *r_, *q_;
        int *gitter;
        int *iteration;
        long double *p;
} t_functions;


#include "marquardt.c"


void spline (t_matrix x, t_matrix y, t_matrix y2) {
#define __SPLINE__
        unsigned int i, k;
        long double p, qn, sig, un;
#ifdef __SPLINE__
        printf ("=> Init"); fflush (stdout);
        t_matrix u = init_matrix (x.size-1);
#endif

        u.matrix[0] = 0.0;

#ifdef __SPLINE__
        printf (" for"); fflush (stdout);
#endif
        for (i = 1; i < x.size-1; i++) {
                sig = (x.matrix[i]-x.matrix[i-1])/(x.matrix[i+1]-x.matrix[i-1]);
                p = sig*y2.matrix[i-1]+2.0;
                y2.matrix[i] = (sig-1.0)/p;
                u.matrix[i] = (y.matrix[i+1]-y.matrix[i])/(x.matrix[i+1]-x.matri
x[i])-(y.matrix[i]-y.matrix[i-1])/(x.matrix[i]-x.matrix[i-1]);
                u.matrix[i] = (6.0*u.matrix[i]/(x.matrix[i+1]-x.matrix[i-1])-sig
*u.matrix[i-1])/p;
        }

        qn = un = 0.0;

#ifdef __SPLINE__
```

lm_mono.c

```
        printf (" for"); fflush (stdout);
#endif
        y2.matrix[y2.size-1] = (un-qn*u.matrix[y2.size-2])/(qn*y2.matrix[y2.size
-2]+1.0);
        for (k = y2.size-2; k > 0; k--)
                y2.matrix[k] = y2.matrix[k]*y2.matrix[k+1]+u.matrix[k];

#ifdef __SPLINE__
        printf (" Cleanup"); fflush (stdout);
#endif
        cleanup_matrix (u);
}

long double splint (t_matrix x, t_matrix y, t_matrix y2, long double x_) {
        int klo = 0, khi = x.size-1, k;
        long double h, b, a;

        while (khi-klo > 1) {
                k = (khi+klo) >> 1;
                if (x.matrix[k] > x_)
                        khi = k;
                else
                        klo = k;
        }

        h = x.matrix[khi]-x.matrix[klo];
        if (h == 0.0)
                return (long double)(0.0);
        a = (x.matrix[khi]-x_)/h;
        b = (x_-x.matrix[klo])/h;
        return (long double)(a*y.matrix[klo]+b*y.matrix[khi]+((a*a*a-a)*y2.matri
x[klo]+(b*b*b-b)*y2.matrix[khi])*(h*h)/6.0);
}


void PY (long double *C_r, long double *Gamma_r, long double *mayer, int gitter)
 {
        unsigned int i;

        for (i = 0; i < gitter; i++)
                C_r[i] = mayer[i] * (1.0+Gamma_r[i]);
}

void HNC (long double *C_r, long double *Gamma_r, long double *Vr, int gitter) {
        unsigned int i;

        for (i = 0; i < gitter; i++)
                C_r[i] = expl (Vr[i] + Gamma_r[i])-Gamma_r[i]-1.0;
}

void FBT1 (t_fbt fbt1) {
        unsigned int m, i;

        for (m = 0; m < fbt1.gitter; m++) {
                fbt1.F_q[m] = 0.0;
                for (i = 0; i < fbt1.gitter; i++)
                        fbt1.F_q[m] += (long double) (fbt1.F_r[i] * fbt1.bessel[
m][i]);
                fbt1.F_q[m] *= fbt1.cons;
        }
}

void OZ (long double *C_q, long double *Gamma_q, long double p, int gitter) {
        unsigned int i;

        for (i = 0; i < gitter; i++)
                Gamma_q[i] = C_q[i]*p*C_q[i]/(1.0-p*C_q[i]);
}
```

lm_mono.c

```
270 void FBT2 (t_fbt fbt2) {
        unsigned int i, m;

        for (i = 0; i < fbt2.gitter; i++) {
                fbt2.F_r[i] = 0.0;
275             for (m = 0; m < fbt2.gitter; m++)
                        fbt2.F_r[i] += (long double) (fbt2.F_q[m] * fbt2.bessel[
m][i]);
                fbt2.F_r[i] *= fbt2.cons;
        }
}
280



int find_i (long double *r_, long double X, int gitter) {
        int i = gitter-1, j;
285     long double minimum = r_[i];

        for (j = 0; j < gitter; j++)
                if (fabsl(r_[j]-X) <= minimum) {
                        minimum = fabsl(r_[j]-X);
290                     i = j;
                }

        return i;
}
295
void invert (long double **matrix, long double **inverse, int dimension) {
        int k, i, j;
        double min_value = powl (10.0, -10.0);

300     for (i = 0; i < dimension; i++)
                if (matrix[i][i] < min_value) {
                        die ("Invertierfehler: (Diagonalelemet %d) = %.64Lf", i, matrix[i][i]
);
                        return;
                }
305
        long double **unity = alloc_matrix_2 (dimension, dimension);
        for (i = 0; i < dimension; i++) for (j = 0; j < dimension; j++)
                unity[i][j] = (long double)(!(i-j));

310     long double xmult;
        for (i = 0; i < dimension-1; i++) {
                for (j = i+1; j < dimension; j++) {
                        xmult = matrix[j][i]/matrix[i][i];
                        for (k = i+1; k < dimension; k++) {
315                             matrix[j][k] -= xmult*matrix[i][k];
                        }
                        matrix[j][i] = xmult;
                        for (k = 0; k < dimension; k++) {
                                unity[j][k] -= xmult*unity[i][k];
320                     }
                }
        }
        long double sum;
        for (i = 0; i < dimension; i++) {
325             inverse[dimension-1][i] = unity[dimension-1][i]/matrix[dimension
-1][dimension-1];
                for (j = dimension-2; j >= 0; j--) {
                        sum = unity[j][i];
                        for (k = j+1; k < dimension; k++) {
                                sum -= matrix[j][k]*inverse[k][i];
330                     }
                        inverse[j][i] = sum/matrix[j][j];
                }
        }
```

lm_mono.c

```
335         free_matrix_2 (unity, dimension);
    }

    void decomposite (t_composite decomp) {
            unsigned int i, n;
340
    #ifdef _DECOMPOSITE_
            out ("\n c: n");
    #endif
            for (n = 0; n < decomp.nu; n++)
345                 decomp.coarse[n] = 0.0;

    #ifdef _DECOMPOSITE_
            out ("a");
    #endif
350         for (n = 0; n < decomp.nu; n++) for (i = 0; i < decomp.gitter; i++)
                    decomp.coarse[n] += decomp.Q_[n][i]*decomp.function[i];

    #ifdef _DECOMPOSITE_
            out ("f");
355 #endif
            for (i = 0; i < decomp.gitter; i++) {
                    decomp.sum = 0.0;
                    for (n = 0; n < decomp.nu; n++)
                            decomp.sum += decomp.coarse[n]*decomp.P_[n][i];
360                 decomp.fine[i] = decomp.function[i]-decomp.sum;
            }
    #ifdef _DECOMPOSITE_
            out (".");
    #endif
365 }

    void composite (t_composite comp) {
            unsigned int i, n;

370         for (i = 0; i < comp.gitter; i++) {
    #ifdef _COMPOSITE_
                    out ("\ni:%d");
    #endif
                    comp.function[i] = comp.fine[i];
375 #ifdef _COMPOSITE_
                    out (" %Lf", comp.function[i]);
    #endif
                    for (n = 0; n < comp.nu; n++) {
                            comp.function[i] += comp.P_[n][i]*comp.coarse[n];
380 #ifdef _COMPOSITE_
                            out (" n:%d %Lf", n, comp.function[i]);
    #endif
                    }
            }
385 }

    void pq_init (long double *r_i_a, unsigned int nu, long double *r_, unsigned int
     gitter, long double **P_, long double **Q_) {
            unsigned int i, m, n;

390         int i_[nu];
            for (n = 0; n < nu; n++) {
                    i_[n] = find_i (r_, r_i_a[n], (int)gitter);
    #ifdef _PQ_INIT_
                    out ("i: %d %Lf", i_[n], r_i_a[n]);
395 #endif
            }

            for (n = 0; n < nu; n++)
                    if (n == 0) {
400                         for (i = 0; i < gitter; i++) {
                                    P_[0][i] = 0.0;
                                    if (i <= i_[0]) P_[0][i] = (long double) (i_[0]-
```

lm_mono.c

```
i) / (i_[0]);
                        }
                } else if (n == 1) {
405                     for (i = 0; i < gitter; i++) {
                                P_[n][i] = 0.0;
                                if (0 <= i && i <= i_[1]) P_[n][i] = (long doubl
e)(i)/(long double)(i_[1]);
                                else if (i_[1] <= i && i <= i_[2]) P_[n][i] = (l
ong double)(i_[2]-i)/(long double)(i_[2]-i_[1]);
                        }
410             } else {
                        for (i = 0; i < gitter; i++) {
                                P_[n][i] = 0.0;
                                if (i_[n-1] <= i && i <= i_[n]) P_[n][i] = fabsl
 ((long double) (i-i_[n-1]) / (i_[n-1]-i_[n]));
                                else if (i_[n] <= i && i <= i_[n+1]) P_[n][i] =
fabsl ((long double) (i_[n+1]-i) / (i_[n+1]-i_[n]));
415                     }
                }

#ifdef _PQ_INIT_
        FILE *p_outfile = fopen ("p_outfile", "w");
420     for (n = 0; n < nu; n++)
                for (i = 0; i < gitter; i++)
                        fprintf (p_outfile, "\n%d %d %Lf", n, i, P_[n][i]);
        fclose (p_outfile);
#endif
425
        long double sum;
        long double **PP = alloc_matrix_2 (nu, nu);
        for (m = 0; m < nu; m++) for (n = 0; n < nu; n++) {
                sum = 0.0;
430             for (i = 0; i < gitter; i++)
                        sum += P_[m][i]*P_[n][i];
                PP[m][n] = sum;
        }

435     long double **RR = alloc_matrix_2 (nu, nu);
        invert (PP, RR, nu);

        for (m = 0; m < nu; m++)
                for (i = 0; i < gitter; i ++) {
440                     sum = 0.0;
                        for (n = 0; n < nu; n++)
                                sum += RR[m][n]*P_[n][i];
                        Q_[m][i] = sum;
                }
445
        free_matrix_2 (RR, nu);
        free_matrix_2 (PP, nu);
}


450


void write_function (long double *function, int gitter, long double *vector, cha
r *name, int iteration) {
        char filename[255];
        sprintf (filename, "%s__%d.dat", name, iteration);
455     FILE *function_file = fopen (filename, "w");
        int i;

        for (i = 0; i < gitter; i++)
                fprintf (function_file, "\n%.64Lf  %.64Lf", vector[i], function[i])
;
460
        fclose (function_file);
}

void coarse_fine_out (t_coarse_fine cf) {
```

lm_mono.c

```
465             char filename_coarse[255], filename_fine[255];
                sprintf (filename_coarse, "coarse__pic%d_nr%d", *cf.pic, *cf.nr);
                sprintf (filename_fine, "fine__pic%d_nr%d", *cf.pic, *cf.nr);
                FILE *coarse_file = fopen (filename_coarse, "w"), *fine_file = fopen (f
ilename_fine, "w");

470             int i, n;

                for (i = 0; i < cf.fine_dim; i++)
                        fprintf (fine_file, "\n%.64Lf %.64Lf", cf.r_[i], cf.fine[i]);

475             long double tmp;
                for (i = 0; i < cf.fine_dim; i++) {
                        fprintf (coarse_file, "\n%.64Lf ", cf.r_[i]);
                        tmp = 0.0;
                        for (n = 0; n < cf.coarse_dim; n++)
480                             tmp += cf.P_[n][i]*cf.coarse[n];
                        fprintf (coarse_file, "%.64Lf ", tmp);
                }

                fclose (coarse_file);
485             fclose (fine_file);
}

void function_out (t_functions m) {
        int i;
        long double *tmp = alloc_matrix_1 (*m.gitter);
490
        for (i = 0; i < *m.gitter; i++)
                tmp[i]= m.Gamma_r[i] + m.C_r[i];
        write_function (tmp, *m.gitter, m.r_, "Hr", *m.iteration);
495     for (i = 0; i < *m.gitter; i++)
                tmp[i] = m.Gamma_q[i] + m.C_q[i];
        write_function (tmp, *m.gitter, m.q_, "Hq", *m.iteration);

        write_function (m.Gamma_r, *m.gitter, m.r_, "Gammar", *m.iteration);
500     write_function (m.Gamma_q, *m.gitter, m.q_, "Gammaq", *m.iteration);

        write_function (m.C_r, *m.gitter, m.r_, "Cr", *m.iteration);
        write_function (m.C_q, *m.gitter, m.q_, "Cq", *m.iteration);

505     for (i = 0; i < *m.gitter; i++)
                tmp[i] = 1.0+(*m.p)*(m.Gamma_q[i]+m.C_q[i]);
        write_function (tmp, *m.gitter, m.q_, "Sq", *m.iteration);

        free_matrix_1 (tmp);
510 }

void read_function (long double *function, long double *r_, int gitter, char *na
me) {
        unsigned int rows = 0;
        unsigned int i;
515     FILE *function_file = fopen (name, "r");

        if (!function_file)
                die ("\nfunction_file: %s", name);
        long double tmp;
520     do {
                fscanf (function_file, "%Lf %Lf\n", &tmp, &tmp);
                rows++;
        } while (!feof (function_file));
        tmp = 0;
525     fclose (function_file);

        function_file = fopen (name, "r");
        if (!function_file)
                die ("\nfunction_file: %s", name);
530     if (gitter < rows)
                die ("\nKeine Interpolation nötig: %d %d", gitter, rows);
```

lm_mono.c

```
            t_matrix r_in = init_matrix (rows);
            t_matrix f_in = init_matrix (rows);
            t_matrix deriv = init_matrix (rows);
535         for (i = 0, r_in.ptr = r_in.matrix, f_in.ptr = f_in.matrix; i < rows; i+
    +, r_in.ptr++, f_in.ptr++)
                    fscanf (function_file, "%Lf %Lf\n", r_in.ptr, f_in.ptr);

            spline (r_in, f_in, deriv);
            for (i = 0; i < (unsigned int)gitter; i++)
540                 function[i] = splint (r_in, f_in, deriv, r_[i]);

            cleanup_matrix (r_in);
            cleanup_matrix (f_in);
            cleanup_matrix (deriv);
545 }

    void return_mayer (long double *mayer, long double *r_, long double *Vr, unsigne
    d int gitter, long double density, long double temperature, long double b_field)
     {
            unsigned int i;

550         long double kb = 1.3806505e-23;
            long double chi = 6.2e-12;
            long double mu_ = 1.0e-7;
            long double T = temperature;
            long double B = b_field;
555         long double r = 1.4e-6;

            long double sigma = r+r;
            double norm = powl (sigma,-3.0);

560         long double V_ = -1.0*mu_*chi*chi*B*B/(kb*T)*norm;
            out ("\n-> Reduziertes Potential: V:%g", V_);

    #if SYSTEM == 1
            for (i = 0; i < gitter; i++) {
565                 mayer[i] = (r_[i] < 1.0) ? -1.0 : 0.0;
                    Vr[i] = (r_[i] < 1.0) ? -1.0/0.0 : 0.0;
            }
    #elif SYSTEM == 2
            for (i = 0; i < gitter; i++) {
570                 mayer[i] = (r_[i] < 1.0) ? -1.0 : expl (V_*powl(r_[i],-3.0))-1.0
    ;
                    Vr[i] = (r_[i] < 1.0) ? -1.0/0.0 : V_*powl(r_[i],-3.0);
            }
    #elif SYSTEM == 3
            long double epsilon = 1.0;
575

            V_ = -temperature;

            for (i = 0; i < gitter; i++) {
                    mayer[i] = (r_[i] < .5) ? -1.0 : expl (V_*(-powl(sigma/r_[i],6.0
    )+powl(sigma/r_[i],12.0)))-1.0;
580                 Vr[i] = (r_[i] < .5) ? -1.0/0.0 : V_*(-powl(sigma/r_[i],6.0)+pow
    l(sigma/r_[i],12.0));
            }
    #elif SYSTEM == 4
            for (i = 0; i < gitter; i++) {
                    mayer[i] = (r_[i] < 1.0) ? -1.0 : expl (-2.0*V_*powl(r_[i],-3.0)
    )-1.0;
585                 Vr[i] = (r_[i] < 1.0) ? -1.0/0.0 : -2.0*V_*powl(r_[i],-3.0);
            }
    #endif
            out (" mittlerer Abstand:%g", pow(density,-.5));
            out (" Vk(r_):%g", Vr[find_i(r_,pow(density,-.5)/kb,(int)gitter)]);
590         out ("\n-> Gamma: %g", 10e-7/(kb*T)*B*B*chi*chi*powl(density*powl(sigma,-2
    .0)*M_PIl,3.0/2.0));
    }
```

lm_mono.c

```
595 void lm_evaluate_default (double* coarse, int nu, double* diff, void *data, int
    *info) {
    #ifdef _EVAL_
            out ("e");
    #endif
            unsigned int i;
600         lm_data_type *d = (lm_data_type*)data;

            for (i = 0; i < nu; i++)
                    d->comp.coarse[i] = coarse[i];

605         composite (d->comp);

    #if APPROXIMATION == 1
            PY (d->Cr, d->Gammar, d->mayer, d->gitter);
    #elif APPROXIMATION == 2
610         HNC (d->Cr, d->Gammar, d->Vr, d->gitter);
    #endif

    #if ASYMPTOTE == 1
            for (i = 0; i < d->gitter; i++)
615                 if (d->r_[i] <= 1.0)
                            d->Cr[i] = -1.0-*d->Gammar[i];
                    else
                            i = d->gitter;
    #elif ASYMPTOTE == 2
620         for (i = 0; i < d->gitter; i++)
                    if (d->r_[i] <= pow (d->p,-.5))
                            d->Cr[i] = -1.0-d->Gammar[i];
                    else
                            i = d->gitter;
625 #endif

            d->fbt1.F_r = d->Cr; d->fbt1.F_q = d->Cq;
            FBT1 (d->fbt1);
            OZ (d->Cq, d->Gammaq, d->p, d->gitter);
630         d->fbt2.F_q = d->Gammaq; d->fbt2.F_r = d->Gammar;
            FBT2 (d->fbt2);
            decomposite (d->decomp);

            for (i = 0; i < nu; i++)
635                 diff[i] = d->comp.coarse[i]-d->decomp.coarse[i];

    #ifdef _EVAL_
            i = 3;
            out ("\n%d c%Lf f%Lf c%Lf f%Lf", i, d->comp.coarse[i], d->comp.fine[i], d->d
    ecomp.coarse[i], d->decomp.fine[i]);
640 #endif
    }



645 int main (int argc, char* argv[]) {
            int v;
            int i,j;
            int m, n;

650

            out ("\n-> Parameter"); fflush (stdout);
            int gitter;
            long double p;
            long double Rmax;
655         long double temperature, b_field;

    #if SYSTEM == 1
            out ("\n-> Harte Scheibchen");
```

lm_mono.c

```
#ifndef BASIS_LINE
        if ((int)argc != 1+3)
                die ("\n<cmd> <density> <rmax> <m>\n");
#else
        if ((int)argc <= 1+3)
                die ("\n<cmd> <density> <rmax> <m> [p...]\n");
#endif
        p = (long double) atof (argv[1]);
        gitter = (unsigned int)(atoi(argv[3]));
        Rmax = (unsigned int)(atoi(argv[2]));

        temperature = b_field = 0.0;

        out (" p:%Lf Rmax:%Lf gitter:%d", p, Rmax, gitter);
#ifdef BASIS_LINE
        int NU = (int)argc - (1+3);
        int nu = NU;
        long double *r_i_a = (long double*) alloc_matrix_1 ((unsigned int)NU);
        for (i = 0; i < NU; i++)
                r_i_a[i] = (long double)(atof(argv[1+3+i]));

#else
#ifndef NEW_BASIS
        #define NU 20
        int nu = NU;
        long double r_i_a[NU] =          {.2, .4, .65, .8, .9,
                                  1.5, 1.7,
                                  2.1, 2.4, 2.7,
                                  3.2, 3.5, 3.8,
                                  4.2, 4.5, 4.9, 5.2, 5.8, 6.3, 7.0};
#endif
#endif
#elif SYSTEM == 2
        out ("\n-> Dipolare harte Scheibchen");

#ifndef BASIS_LINE
        if ((int)argc != 1+5)
                die ("\n<cmd> <density> <temperature> <b> <rmax> <m>\n");
#else
        if ((int)argc <= 1+5)
                die ("\n<cmd> <density> <temperature> <b> <rmax> <m> [p_...] \n");
#endif
        p = (long double) atof (argv[1]);
        temperature = (long double) atof (argv[2]);
        b_field = (long double) atof (argv[3]) * powl (10.0, -3.0);

        gitter = (unsigned int)(atoi(argv[5]));
        Rmax = (unsigned int)(atoi(argv[4]));

        out (" p:%Lf temperature:%Lf b:%Lf Rmax:%Lf gitter:%d", p, temperature, b_field, Rm
ax, gitter);
#ifdef BASIS_LINE
        int NU = (int)argc - (1+5);
        int nu = NU;
        long double *r_i_a = (long double*) alloc_matrix_1 ((unsigned int)NU);
        for (i = 0; i < NU; i++)
                r_i_a[i] = (long double)(atof(argv[6+i]));
#else
#ifndef NEW_BASIS
        #define NU 29

        int nu = NU;
        long double r_i_a[NU] ={.2, .4, .65, .9, 1.5, 1.7, 2.1, 2.3, 2.7, 3.0, 3
.5, 4.0, 4.5, 5.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0
, 18.0, 19.0, 20.0, 25.0};
#endif
#endif
#elif SYSTEM == 3
        out ("\n-> LJ ");
```

lm_mono.c

```
        if ((int) argc != 1+2)
                die ("\n<cmd> <density> <temperature>\n");
        p = (long double) atof (argv[1]);
        temperature = (long double) atof (argv[2]);
        b_field = 0.0;

        gitter = .25*1024;
        Rmax = 10.0;

        out (" p:%Lf temperature:%Lf Rmax:%Lf gitter:%d", p, temperature, Rmax, gitter);
#ifndef NEW_BASIS
        #define NU 20
        int nu = NU;
        long double r_i_a[NU] = {.05, .1, .15, .2, .3, .4, .7, .9, 1.2, 1.5, 2.0
, 2.3, 2.7, 3.0, 3.5, 4.0, 4.5, 5.0, 7.0, 9.0 };
#endif
#elif SYSTEM == 4
        out ("\n-> Dipolare attraktive harte Scheibchen");

        if ((int)argc != 1+3)
                die ("\n<cmd> <density> <temperature> <b>\n");
        p = (long double) atof (argv[1]);
        temperature = (long double) atof (argv[2]);
        b_field = (long double) atof (argv[3]) * powl (10.0, -3.0);

        gitter = 2*1024;
        Rmax = 30.0;

        out (" p:%Lf temperature:%Lf b:%Lf Rmax:%Lf gitter:%d", p, temperature, b_field, Rm
ax, gitter);
#ifndef NEW_BASIS
        #define NU 28
        int nu = NU;
        long double r_i_a[NU] = {.1, .2, .3, .5, .7, .9, 1.2, 1.5, 1.7, 2.0, 2.3
, 2.7, 3.0, 3.3, 3.7, 4.0, 4.3, 4.7, 5.0, 5.5, 6.0, 6.5, 7.0, 8.0, 9.0, 10.0, 12
.0, 15.0};
#endif
#endif


#ifdef NEW_BASIS
        out ("\n-> Neue Methode der Basisfuntionserzeugung");
        int nu = 17;
        long double *r_i_a = malloc ((size_t)(nu*sizeof(long double)));
        long double *r_i_a_reservoir = malloc ((size_t)((nu*(nu+1)/2)*sizeof(lon
g double))),
                *r_i_a_ptr = r_i_a_reservoir;
        r_i_a_reservoir[0] = .1;
        r_i_a_reservoir[1] = .2;
        r_i_a_reservoir[2] = .3;
        r_i_a_reservoir[3] = .45;
        r_i_a_reservoir[4] = .65;
        r_i_a_reservoir[5] = .9;
        r_i_a_reservoir[6] = 1.1;
        r_i_a_reservoir[7] = 1.5;
        r_i_a_reservoir[8] = 1.9;
        r_i_a_reservoir[9] = 2.2;
        r_i_a_reservoir[10] = 2.5;
        r_i_a_reservoir[11] = 2.7;
        r_i_a_reservoir[12] = 3.2;
        r_i_a_reservoir[13] = 4.5;
        r_i_a_reservoir[14] = 6.0;
        r_i_a_reservoir[15] = 8.0;
        r_i_a_reservoir[16] = 10.0;
        // nu = 16
        r_i_a_reservoir[17+0] = .1;
        r_i_a_reservoir[17+1] = .2;
        r_i_a_reservoir[17+2] = .3;
        r_i_a_reservoir[17+3] = .45;
```

lm_mono.c

```
790         r_i_a_reservoir[17+4] = .65;
            r_i_a_reservoir[17+5] = .9;
            r_i_a_reservoir[17+6] = 1.1;
            r_i_a_reservoir[17+7] = 1.5;
            r_i_a_reservoir[17+8] = 1.9;
795         r_i_a_reservoir[17+9] = 2.5;
            r_i_a_reservoir[17+10] = 3.0;
            r_i_a_reservoir[17+11] = 3.5;
            r_i_a_reservoir[17+12] = 4.5;
            r_i_a_reservoir[17+13] = 6.5;
800         r_i_a_reservoir[17+14] = 8.0;
            r_i_a_reservoir[17+15] = 10.0;
            // nu = 15
            r_i_a_reservoir[17+16+0] = .1;
            r_i_a_reservoir[17+16+1] = .2;
805         r_i_a_reservoir[17+16+2] = .3;
            r_i_a_reservoir[17+16+3] = .45;
            r_i_a_reservoir[17+16+4] = .65;
            r_i_a_reservoir[17+16+5] = .9;
            r_i_a_reservoir[17+16+6] = 1.1;
810         r_i_a_reservoir[17+16+7] = 1.5;
            r_i_a_reservoir[17+16+8] = 1.9;
            r_i_a_reservoir[17+16+9] = 2.5;
            r_i_a_reservoir[17+16+10] = 3.0;
            r_i_a_reservoir[17+16+11] = 3.5;
815         r_i_a_reservoir[17+16+12] = 4.5;
            r_i_a_reservoir[17+16+13] = 6.5;
            r_i_a_reservoir[17+16+14] = 8.5;
            // nu = 14
            r_i_a_reservoir[17+16+15+0] = .1;
820         r_i_a_reservoir[17+16+15+1] = .2;
            r_i_a_reservoir[17+16+15+2] = .3;
            r_i_a_reservoir[17+16+15+3] = .45;
            r_i_a_reservoir[17+16+15+4] = .65;
            r_i_a_reservoir[17+16+15+5] = .9;
825         r_i_a_reservoir[17+16+15+6] = 1.1;
            r_i_a_reservoir[17+16+15+7] = 1.5;
            r_i_a_reservoir[17+16+15+8] = 1.9;
            r_i_a_reservoir[17+16+15+9] = 2.5;
            r_i_a_reservoir[17+16+15+10] = 3.0;
830         r_i_a_reservoir[17+16+15+11] = 4.5;
            r_i_a_reservoir[17+16+15+12] = 5.5;
            r_i_a_reservoir[17+16+15+13] = 8.5;
            // nu = 13
            r_i_a_reservoir[17+16+15+14+0] = .1;
835         r_i_a_reservoir[17+16+15+14+1] = .2;
            r_i_a_reservoir[17+16+15+14+2] = .3;
            r_i_a_reservoir[17+16+15+14+3] = .45;
            r_i_a_reservoir[17+16+15+14+4] = .65;
            r_i_a_reservoir[17+16+15+14+5] = .9;
840         r_i_a_reservoir[17+16+15+14+6] = 1.1;
            r_i_a_reservoir[17+16+15+14+7] = 1.5;
            r_i_a_reservoir[17+16+15+14+8] = 1.9;
            r_i_a_reservoir[17+16+15+14+9] = 2.5;
            r_i_a_reservoir[17+16+15+14+10] = 3.0;
845         r_i_a_reservoir[17+16+15+14+11] = 5.5;
            r_i_a_reservoir[17+16+15+14+12] = 8.5;
            // nu = 12
            r_i_a_reservoir[17+16+15+14+13+0] = .1;
            r_i_a_reservoir[17+16+15+14+13+1] = .2;
850         r_i_a_reservoir[17+16+15+14+13+2] = .3;
            r_i_a_reservoir[17+16+15+14+13+3] = .45;
            r_i_a_reservoir[17+16+15+14+13+4] = .65;
            r_i_a_reservoir[17+16+15+14+13+5] = .9;
            r_i_a_reservoir[17+16+15+14+13+6] = 1.1;
855         r_i_a_reservoir[17+16+15+14+13+7] = 1.5;
            r_i_a_reservoir[17+16+15+14+13+8] = 1.9;
            r_i_a_reservoir[17+16+15+14+13+9] = 2.5;
            r_i_a_reservoir[17+16+15+14+13+10] = 4.7;
```

lm_mono.c

```
            r_i_a_reservoir[17+16+15+14+13+11] = 8.5;
860         // nu = 11
            r_i_a_reservoir[17+16+15+14+13+12+0] = .1;
            r_i_a_reservoir[17+16+15+14+13+12+1] = .25;
            r_i_a_reservoir[17+16+15+14+13+12+2] = .3;
            r_i_a_reservoir[17+16+15+14+13+12+3] = .45;
865         r_i_a_reservoir[17+16+15+14+13+12+4] = .65;
            r_i_a_reservoir[17+16+15+14+13+12+5] = .9;
            r_i_a_reservoir[17+16+15+14+13+12+6] = 1.1;
            r_i_a_reservoir[17+16+15+14+13+12+7] = 1.5;
            r_i_a_reservoir[17+16+15+14+13+12+8] = 2.5;
870         r_i_a_reservoir[17+16+15+14+13+12+9] = 4.5;
            r_i_a_reservoir[17+16+15+14+13+12+10] = 8.0;
            // nu = 10
            r_i_a_reservoir[17+16+15+14+13+12+11+0] = .1;
            r_i_a_reservoir[17+16+15+14+13+12+11+1] = .25;
875         r_i_a_reservoir[17+16+15+14+13+12+11+2] = .45;
            r_i_a_reservoir[17+16+15+14+13+12+11+3] = .65;
            r_i_a_reservoir[17+16+15+14+13+12+11+4] = .9;
            r_i_a_reservoir[17+16+15+14+13+12+11+5] = 1.4;
            r_i_a_reservoir[17+16+15+14+13+12+11+6] = 2.2;
880         r_i_a_reservoir[17+16+15+14+13+12+11+7] = 3.0;
            r_i_a_reservoir[17+16+15+14+13+12+11+8] = 4.5;
            r_i_a_reservoir[17+16+15+14+13+12+11+9] = 8.0;
            // nu = 9
            r_i_a_reservoir[17+16+15+14+13+12+11+10+0] = .1;
885         r_i_a_reservoir[17+16+15+14+13+12+11+10+1] = .25;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+2] = .45;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+3] = .65;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+4] = .9;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+5] = 1.2;
890         r_i_a_reservoir[17+16+15+14+13+12+11+10+6] = 1.9;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+7] = 2.5;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+8] = 4.9;
            // nu = 8
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+0] = .1;
895         r_i_a_reservoir[17+16+15+14+13+12+11+10+9+1] = .3;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+2] = .65;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+3] = .9;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+4] = 1.2;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+5] = 1.7;
900         r_i_a_reservoir[17+16+15+14+13+12+11+10+9+6] = 2.5;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+7] = 4.9;
            // nu = 7
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+8+0] = .1;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+8+1] = .3;
905         r_i_a_reservoir[17+16+15+14+13+12+11+10+9+8+2] = .65;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+8+3] = .9;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+8+4] = 1.2;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+8+5] = 1.9;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+8+6] = 3.5;
910         // nu = 6
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+8+7+0] = .25;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+8+7+1] = .65;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+8+7+2] = .9;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+8+7+3] = 1.2;
915         r_i_a_reservoir[17+16+15+14+13+12+11+10+9+8+7+4] = 2.4;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+8+7+5] = 3.0;
            // nu = 5
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+8+7+6+0] = .5;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+8+7+6+1] = .9;
920         r_i_a_reservoir[17+16+15+14+13+12+11+10+9+8+7+6+2] = 1.5;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+8+7+6+3] = 2.3;
            r_i_a_reservoir[17+16+15+14+13+12+11+10+9+8+7+6+4] = 3.0;

            Rmax = 100.0;
925 #endif

            out ("\n-> Basen (%d):", nu);
```

lm_mono.c

```
            for (i = 0; i < nu; i++)
                    out (" %.1Lf", r_i_a[i]);
930
            out ("\n--> Mittlerer Abstand: %lf", pow(p,-.5));
            if (pow(p,-.5) < 1.0)
                    die ("!\n");

935
            out ("\n--> Funktionen initialisieren");
            long double *C_r = alloc_matrix_1 (gitter);
            long double *C_q = alloc_matrix_1 (gitter);
            long double *Gamma_r = alloc_matrix_1 (gitter);
940         long double *Gamma_q = alloc_matrix_1 (gitter);


            out ("\n--> Besselfunktionen");
            long double Qmax = gsl_sf_bessel_zero_J0 (gitter+1) / Rmax;
945         long double *r_ = alloc_matrix_1 (gitter);
            for (i = 0; i < gitter; i++)
                    r_[i] = (long double) (gsl_sf_bessel_zero_J0 (i+1) / Qmax);
            long double *q_ = alloc_matrix_1 (gitter);
            for (i = 0; i < gitter; i++)
950                 q_[i] = (long double) (gsl_sf_bessel_zero_J0 (i+1) / Rmax);

            t_fbt fbt1 = {
                    .gitter = gitter,
                    .cons = 4.0*M_PI1/(Qmax*Qmax),
955                 .bessel = alloc_matrix_2 (gitter, gitter)
            };
            for (m = 0; m < gitter; m++) for (i = 0; i < gitter; i++)
                    fbt1.bessel[m][i] = j0l (q_[m] * r_[i]) / powl (j1l (r_[i] * Qma
    x), 2.0);

960         t_fbt fbt2 = {
                    .gitter = gitter,
                    .cons = 1.0/(M_PI1*Rmax*Rmax),
                    .bessel = alloc_matrix_2 (gitter, gitter)
            };
965         for (m = 0; m < gitter; m++) for (i = 0; i < gitter; i++)
                    fbt2.bessel[m][i] = j0l (q_[m] * r_[i]) / powl (j1l (q_[m] * Rma
    x), 2.0);


            out ("\n--> Mayerfunktion");
970         long double *mayer = alloc_matrix_1 (gitter);
            long double *Vr = alloc_matrix_1 (gitter);
            return_mayer (mayer, r_, Vr, gitter, p, temperature, b_field);
            write_function (mayer, gitter, r_, "mayer", 0);
    #ifdef _MAYER_
975         goto clean;
    #endif


            out ("\n--> Startwerte");
980 #if STARTVALUE == 1
            out (" Gamma(r) == 0");
            for (i = 0; i < gitter; i++)
                    Gamma_r[i] = 0.0;
    #elif STARTVALUE == 2
985         out (" Gamma(r) = 'Fitfunktion'");
            for (i = 0; i < gitter; i++)
                    Gamma_r[i] = 20.0*expl (-powl (r_[i], 2.0)/4.0);
    #elif STARTVALUE == 3
            out (" Gamma(r) einlesen aus 'startvalue'");
990         read_function (Gamma_r, r_, gitter, "startvalue");
    #elif STARTVALUE == 4
            out (" Gamma(r) == Mayerfkt");
            for (i = 0; i < gitter; i++)
                    Gamma_r[i] = mayer[i];
```

lm_mono.c

```
995 #endif


    #ifdef DYN_BASIS
            r_i_a_ptr = r_i_a_reservoir;
1000        for (n = 0; n < nu; n++, r_i_a_ptr++)
                    r_i_a[n] = *r_i_a_ptr;
            out ("\n--> %d Basen: ", nu);
            for (n = 0; n < nu; n++)
                    out (" %.3Lf", r_i_a[n]);
1005 #endif


    #ifdef NEW_BASIS
            long double **P_ = alloc_matrix_2 (nu, gitter),
1010                **Q_ = alloc_matrix_2 (nu, gitter);
            pq_init (r_i_a, (unsigned int)nu, r_, (unsigned int)gitter, P_, Q_);
    #else
            out ("\n--> Zerlegungsbasis ");
    #ifndef BASIS_LINE
1015        int i_[NU];
    #else
            int *i_ = (int *) calloc ((size_t)NU, (size_t)sizeof (int));
    #endif
            for (n = 0; n < nu; n++)
1020                i_[n] = find_i (r_, r_i_a[n], gitter);

            out ("P_n^i ");
            long double **P_ = alloc_matrix_2 (nu, gitter);
            for (n = 0; n < nu; n++)
1025                if (n == 0) {
                            for (i = 0; i < gitter; i++) {
                                    P_[0][i] = 0.0;
                                    if (i <= i_[1]) P_[0][i] = (long double) (i_[1]-
    i) / (long double)(i_[1]);
                            }
1030                } else if (n == 1) {
                            for (i = 0; i < gitter; i++) {
                                    P_[n][i] = 0.0;
                                    if (0 <= i && i <= i_[1]) P_[n][i] = (long doubl
    e)(i)/(long double)(i_[1]);
                                    else if (i_[1] <= i && i <= i_[2]) P_[n][i] = (l
    ong double)(i_[2]-i)/(long double)(i_[2]-i_[1]);
1035                        }
                    } else {
                            for (i = 0; i < gitter; i++) {
                                    P_[n][i] = 0.0;
                                    if (i_[n-1] <= i && i <= i_[n]) P_[n][i] = fabsl
     ((long double) (i-i_[n-1]) / (i_[n-1]-i_[n]));
1040                                if (i_[n] <= i && i <= i_[n+1]) P_[n][i] = fabsl
    ((long double) (i_[n+1]-i) / (i_[n+1]-i_[n]));
                            }
                    }

            out ("Q_n^i ");
1045        long double **Q_ = alloc_matrix_2 (nu, gitter);

            long double sum;
            long double **PP = alloc_matrix_2 (nu, nu);
            for (m = 0; m < nu; m++) for (n = 0; n < nu; n++) {
1050                sum = 0.0;
                    for (i = 0; i < gitter; i++)
                            sum += P_[m][i]*P_[n][i];
                    PP[m][n] = sum;
            }
1055
            long double **RR = alloc_matrix_2 (nu, nu);
            invert (PP, RR, nu);
```

lm_mono.c

```
        for (m = 0; m < nu; m++)
                for (i = 0; i < gitter; i ++) {
                        sum = 0.0;
                        for (n = 0; n < nu; n++)
                                sum += RR[m][n]*P_[n][i];
                        Q_[m][i] = sum;
                }
#endif

#define _BASIS_
#ifdef _BASIS_
        FILE *basis_file = fopen ("basis", "w");
        if (!basis_file)
                die ("\nbasis_file: basis\n");
        for (i = 0; i < gitter; i++) {
                fprintf (basis_file, "\n%Lf ", r_[i]);
                for (m = 0; m < nu; m++)
                        fprintf (basis_file, "%Lf ", P_[m][i]);
                for (m = 0; m < nu; m++)
                        fprintf (basis_file, "%Lf ", Q_[m][i]);
        }
        fclose (basis_file);
#endif

        t_composite comp = {
                .function = Gamma_r,
                .coarse = alloc_matrix_1 (nu),
                .fine = alloc_matrix_1 (gitter),
                .P_ = P_,
                .Q_ = Q_,
                .gitter = gitter,
                .nu = nu
        };

        t_composite decomp = {
                .function = Gamma_r,
                .coarse = alloc_matrix_1 (nu),
                .fine = alloc_matrix_1 (gitter),
                .P_ = P_,
                .Q_ = Q_,
                .gitter = gitter,
                .nu = nu
        };

#ifndef NEW_BASIS
        free_matrix_2 (RR, nu);
        free_matrix_2 (PP, nu);
#endif



        out ("\n-> Hauptschleife");
        int pic = 0;
        long double norm_fine = 10.0, norm_fine_old = 10.0;
        long double norm_fine_max = 1e-10;
        long double norm_sum = 0.0;

        t_functions functions = {
                .Gamma_r = Gamma_r,
                .Gamma_q = Gamma_q,
                .C_r = C_r,
                .C_q = C_q,
                .r_ = r_,
                .q_ = q_,
                .gitter = &gitter,
                .iteration = &pic,
                .p = &p
        };
```

lm_mono.c

```
#define __STARTFUNCTION
#ifdef __STARTFUNCTION
        function_out (functions);
#endif

#ifdef _cf_
        t_coarse_fine coarse_fine = {
                .coarse = decomp.coarse,
                .fine = decomp.fine,
                .coarse_dim = &nu,
                .fine_dim = gitter,
                .nr = &nr,
                .pic = &pic,
                .r_ = r_,
                .P_ = P_
        };
        coarse_fine.coarse_dim = coarse_fine.coarse_dim;
#endif


#ifdef TEST
        goto clean;
#endif

        out ("\n-> Erste Zerlegung");
        decomposite (comp);
#ifdef DYN_BASIS_1
        for (i = 0; i < gitter; i++)
                decomp.fine[i] = comp.fine[i];
        for (n = 0; n < nu; n++)
                decomp.coarse[n] = comp.coarse[n];
#endif

        lm_data_type data = {
                .Cr = C_r, .Cq = C_q, .Gammar = Gamma_r, .Gammaq = Gamma_q,
                .mayer = mayer, .Vr = Vr,
                .r_ = r_,
                .p = p,
                .gitter = gitter, .nu = nu,
                .fbt1 = fbt1, .fbt2 = fbt2,
                .comp = {
                        .function = Gamma_r,
                        .coarse = alloc_matrix_1 (nu),
                        .fine = alloc_matrix_1 (gitter),
                        .P_ = P_, .Q_ = Q_,
                        .gitter = gitter, .nu = nu
                },
                .decomp = {
                        .function = Gamma_r,
                        .coarse = alloc_matrix_1 (nu),
                        .fine = alloc_matrix_1 (gitter),
                        .P_ = P_, .Q_ = Q_,
                        .gitter = gitter, .nu = nu
                }
        };

        time_t start, now;
        time (&start);

        j = v = 1.0;

        lm_control_type control;
        lm_initialize_control (&control);

        control.epsilon = control.ftol = control.xtol = control.gtol = 1e-25;


#ifdef DYN_COARSE
        out ("\n-> Grobanteilgenauigkeit adaptiv anpassen");
```

lm_mono.c

```
            double dyn_coarse_step = 1e-1,
                   dyn_coarse = 1e-3,
                   dyn_coarse_max = 1e-12;
1200        double norm_fine_max_dyn_coarse = 1e-10;
            control.epsilon = dyn_coarse;
            control.ftol = dyn_coarse;
            control.xtol = dyn_coarse;
            control.gtol = dyn_coarse;
1205    #endif
    #ifdef DYN_COARSE_2
            out ("\n--> Grobanteilgenauigkeit adaptiv anpassen (2)");
            double dyn_coarse_step = 1e-1,
                   dyn_coarse = 1e-12,
1210               dyn_coarse_min = 1e-1;
            double norm_fine_max_dyn_coarse = 1e-1;
            control.epsilon = dyn_coarse;
            control.ftol = dyn_coarse;
            control.xtol = dyn_coarse;
1215        control.gtol = dyn_coarse;
    #endif
    #ifdef DYN_BASIS
            double norm_fine_max_dyn_basis = 1e-5;
    #endif
1220 #ifdef DYN_BASIS_1
            double norm_fine_max_dyn_basis = 1e-3;
    #endif

    #ifdef DYN_BASIS_1
1225        out ("\n--> Neu Basis (1)");
                            composite (data.decomp);

                            free_matrix_2 (P_, nu);
                            free_matrix_2 (Q_, nu);
1230
                            nu=7;
                            data.nu = data.comp.nu = data.decomp.nu = comp.nu = deco
    mp.nu = nu;
                            data.comp.coarse = (long double *) realloc (data.comp.co
    arse, (size_t)(nu*sizeof(long double)));
                            data.decomp.coarse = (long double *) realloc (data.decom
    p.coarse, (size_t)(nu*sizeof(long double)));
1235                        comp.coarse = (long double *) realloc (comp.coarse, (siz
    e_t)(nu*sizeof(long double)));
                            decomp.coarse = (long double *) realloc (decomp.coarse,
    (size_t)(nu*sizeof(long double)));
                            r_i_a = (long double *)malloc ((size_t)(nu*sizeof(long d
    ouble)));
                            for (m = 17, j = 0; m > nu; m--)
                                    j += m;
1240                        for (n = 0, r_i_a_ptr = r_i_a_reservoir+j; n < nu; n++,
    r_i_a_ptr++)
                                    r_i_a[n] = *r_i_a_ptr;


                            P_ = alloc_matrix_2 (nu, gitter);
                            Q_ = alloc_matrix_2 (nu, gitter);
1245                        pq_init (r_i_a, (unsigned int)nu, r_, (unsigned int)gitt
    er, P_, Q_);
                            data.comp.P_ = data.decomp.P_ = comp.P_ = decomp.P_ = P_
    ;
                            data.comp.Q_ = data.decomp.Q_ = comp.Q_ = decomp.Q_ = Q_
    ;


                            decomposite (data.decomp);
1250                        for (i = 0; i < gitter; i++)
                                    comp.fine[i] = data.decomp.fine[i];

                            out ("\n--> %d Basen:", nu);
                            for (n = 0; n < nu; n++)
1255                                out (" %.3Lf", r_i_a[n]);
```

lm_mono.c

```
    #endif



1260        out ("\n--> Hauptschleife");
            unsigned int pic_max = 10000;
            unsigned int out_period = 1;
            double out_period_norm = 1e-1;
            do {
1265                pic++;
                    out ("\n%d", pic);

                    for (i = 0; i < gitter; i++)
                            data.comp.fine[i] = comp.fine[i];
1270
                    out (" lm %d ", nu);
                    lm_minimize (nu, nu, (double *)comp.coarse, lm_evaluate_default,
     lm_print_default, &data, &control);

                    norm_fine_old = norm_fine;
1275                norm_fine = norm_sum = 0.0;
                    for (i = 0; i < gitter; i++) {
                            norm_fine += powl ((comp.fine[i]-data.decomp.fine[i]), 2
    .0);
                            norm_sum += data.decomp.fine[i];
                    }
1280                norm_fine = sqrt (norm_fine)/norm_sum;
                    out (" fine: %g", (double)norm_fine);

                    if (norm_fine_old < norm_fine) {
                            out (" !! ");
1285                        for (i = 0; i < gitter; i++)
                                    comp.fine[i] = .1*data.decomp.fine[i] + (1.0-.1)
    *comp.fine[i];
                    } else {
                            for (i = 0; i < gitter; i++)
                                    comp.fine[i] = data.decomp.fine[i];
1290                }

    #ifdef DYN_BASIS_1
                    if (norm_fine < norm_fine_max_dyn_basis && nu < 16) {
                            composite (data.decomp);
1295
                            free_matrix_2 (P_, nu);
                            free_matrix_2 (Q_, nu);


                            nu++;
1300                        data.nu = data.comp.nu = data.decomp.nu = comp.nu = deco
    mp.nu = nu;
                            data.comp.coarse = (long double *) realloc (data.comp.co
    arse, (size_t)(nu*sizeof(long double)));
                            data.decomp.coarse = (long double *) realloc (data.decom
    p.coarse, (size_t)(nu*sizeof(long double)));
                            comp.coarse = (long double *) realloc (comp.coarse, (siz
    e_t)(nu*sizeof(long double)));
                            decomp.coarse = (long double *) realloc (decomp.coarse,
    (size_t)(nu*sizeof(long double)));
1305                        r_i_a = (long double *)malloc ((size_t)(nu*sizeof(long d
    ouble)));
                            for (m = 17, i = 0; m > nu; m--)
                                    i += m;
                            for (n = 0, r_i_a_ptr = r_i_a_reservoir+i; n < nu; n++,
    r_i_a_ptr++)
                                    r_i_a[n] = *r_i_a_ptr;
1310
                            P_ = alloc_matrix_2 (nu, gitter);
                            Q_ = alloc_matrix_2 (nu, gitter);
                            pq_init (r_i_a, (unsigned int)nu, r_, (unsigned int)gitt
    er, P_, Q_);
```

lm_mono.c

```
                        data.comp.P_ = data.decomp.P_ = comp.P_ = decomp.P_ = P_
    ;
1315                    data.comp.Q_ = data.decomp.Q_ = comp.Q_ = decomp.Q_ = Q_
    ;


                        decomposite (data.decomp);
                        for (i = 0; i < gitter; i++)
                                comp.fine[i] = data.decomp.fine[i];
1320
                        out ("\n--> %d Basen:", nu);
                        for (n = 0; n < nu; n++)
                                out (" %.3Lf", r_i_a[n]);
                }
1325 #endif

    #ifdef DYN_BASIS
                if (norm_fine < norm_fine_max_dyn_basis && nu == 14) {
                        composite (data.decomp);
1330
                        free_matrix_2 (P_, nu);
                        free_matrix_2 (Q_, nu);


                        nu = 14;
1335                    data.nu = data.comp.nu = data.decomp.nu = comp.nu = deco
    mp.nu = nu;
                        data.comp.coarse = (long double *) realloc (data.comp.co
    arse, (size_t)(nu*sizeof(long double)));
                        data.decomp.coarse = (long double *) realloc (data.decom
    p.coarse, (size_t)(nu*sizeof(long double)));
                        comp.coarse = (long double *) realloc (comp.coarse, (siz
    e_t)(nu*sizeof(long double)));
                        decomp.coarse = (long double *) realloc (decomp.coarse,
    (size_t)(nu*sizeof(long double)));
1340                    r_i_a = (long double *)malloc ((size_t)(nu*sizeof(long d
    ouble)));
                        for (n = 0; n < nu; n++, r_i_a_ptr++)
                                r_i_a[n] = *r_i_a_ptr;


                        P_ = alloc_matrix_2 (nu, gitter);
1345                    Q_ = alloc_matrix_2 (nu, gitter);
                        pq_init (r_i_a, (unsigned int)nu, r_, (unsigned int)gitt
    er, P_, Q_);
                        data.comp.P_ = data.decomp.P_ = comp.P_ = decomp.P_ = P_
    ;
                        data.comp.Q_ = data.decomp.Q_ = comp.Q_ = decomp.Q_ = Q_
    ;


1350                    decomposite (data.decomp);
                        for (i = 0; i < gitter; i++)
                                comp.fine[i] = data.decomp.fine[i];


                        out ("\n--> %d Basen:", nu);
1355                    for (n = 0; n < nu; n++)
                                out (" %.3Lf", r_i_a[n]);
                }
    #endif
    #ifdef DYN_COARSE
1360                if (norm_fine < dyn_coarse && dyn_coarse > dyn_coarse_max) {
                        dyn_coarse *= dyn_coarse_step;
                        control.epsilon = dyn_coarse;
                        control.ftol = dyn_coarse;
                        control.xtol = dyn_coarse;
1365                    control.gtol = dyn_coarse;
                        out ("\n--> Grobanteilgenauigkeit anpassen %g\n", dyn_coarse);
    #ifdef DEBUG
    #endif
                }
1370 #endif
    #ifdef DYN_COARSE_2
```

lm_mono.c

```
                if (norm_fine < norm_fine_max_dyn_coarse && dyn_coarse < dyn_coa
    rse_min) {
                        dyn_coarse /= dyn_coarse_step;
                        control.epsilon = dyn_coarse;
1375                    control.ftol = dyn_coarse;
                        control.xtol = dyn_coarse;
                        control.gtol = dyn_coarse;
                        out ("\n--> Grobanteilgenauigkeit anpassen %.15lf", dyn_coarse);
                }
1380 #endif

    #ifdef DEBUG
                function_out (functions);
    #else
1385                if ((pic % 100) == 0)
                        function_out (functions);
    #endif


1390                if (norm_fine < out_period_norm) {
                        function_out (functions);
                        out_period_norm = .1*norm_fine;
                        out_period *= 10;
                }
1395                if ((pic % out_period) == 0)
                        function_out (functions);

                time (&now);
                out (" time: %.0lf", difftime (now, start));
1400 #ifdef DYN_COARSE
                } while (norm_fine > norm_fine_max && pic < pic_max);
    #else
                } while (norm_fine > norm_fine_max && pic < pic_max);
    #endif
1405
    #ifdef DYN_COARSE
                norm_fine_max_dyn_coarse = 0.0;
    #endif
                goto clean;
1410 clean:
                out ("\n--> Cleanup");
                function_out (functions);

                free_matrix_1 (C_r);
1415            free_matrix_1 (C_q);
                free_matrix_1 (mayer);
                free_matrix_1 (Vr);
                free_matrix_1 (Gamma_r);
                free_matrix_1 (Gamma_q);
1420 #ifdef PIC_AFTER
                free_matrix_1 (Gamma__);
    #endif

                free_matrix_1 (r_);
1425            free_matrix_1 (q_);
                free_matrix_2 (fbt1.bessel, gitter);
                free_matrix_2 (fbt2.bessel, gitter);

    #ifdef NEW_BASIS
1430            free_matrix_1 (r_i_a);
                free_matrix_1 (r_i_a_reservoir);
    #endif

                free_matrix_2 (P_, nu);
1435            free_matrix_2 (Q_, nu);

                free_matrix_1 (data.comp.fine);
                free_matrix_1 (data.comp.coarse);
                free_matrix_1 (data.decomp.fine);
```

lm_mono.c

```
1440            free_matrix_1 (data.decomp.coarse);

                free_matrix_1 (comp.fine);
                free_matrix_1 (comp.coarse);
                free_matrix_1 (decomp.fine);
1445            free_matrix_1 (decomp.coarse);

        #ifdef BASIS_LINE
                free (i_);
                free (r_i_a);
1450    #endif

                out ("\n--> Fertig\n");
                if (pic >= pic_max)
                        return (-1);
1455
                return (pic);
        }
```

B.2. Levenberg-Marquardt-Algorithmus binär

lm_bin.c

```
// compile with: gcc -Wall -pedantic -funroll-loops -std=c99 -lgsl -lgslcblas -l
m -O3
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#define __USE_GNU 1
#define _GNU_SOURCE 1
double j0(double x);
double j1(double y);
#include <math.h>
#include <getopt.h>
#include <string.h>

#include <time.h>

#include <assert.h>
#define NDEBUG

#define __USE_GSL 1
#ifdef __USE_GSL
#include <gsl/gsl_sf_bessel.h>
#else
long double gsl_sf_bessel_zero_J0 (int number) {
        return M_PIl/4.0*(4.0*number-1.0) +
                1.0/(2.0*M_PIl*(4.0*number-1.0)) -
                31.0/(6.0*powl(M_PIl*(4.0*number-1.0),3.0)) +
                3779.0/(15.0*powl(M_PIl*(4.0*number-1.0),5.0));
}
#endif


#define SYSTEM 6
//#define FINE_RESCALE
#define APPROXIMATION 1
#define BASIS_LINE
#define PACKING_FRACTION
//#define DYN_COARSE
//#define SYM 2
//#define ASYMPTOTE
#define STARTVALUE 2
#define SYMMETRIC_FT
#define MARQUARDT
//#define DYN_COARSE
//#define T_PATH



typedef struct {
        double *matrix;
        double *ptr, *ptr1;
        unsigned int size;
} t_matrix;

typedef struct {
        t_matrix mr;
        t_matrix mq;
        t_matrix bessel;
        int gitter;
        double cons;
} t_fbt;

typedef struct {
        t_matrix function;
        t_matrix coarse, fine;
        t_matrix P_, Q_;
        unsigned int gitter, nu;
        double sum;
} t_composite;
```

lm_bin.c

```
typedef struct {
        t_matrix P_;
        unsigned int gitter, nu;
        t_matrix r_;
} t_basis_out;

typedef struct {
        t_matrix coarse;
        t_matrix fine;
        unsigned int *pic;
        unsigned int gitter, nu;
        t_matrix r_;
        t_matrix P_;
} t_coarse_fine;

typedef struct {
        t_matrix Gammar, Gammaq, Cr, Cq;
        t_matrix r_, q_;
        unsigned int *pic;
        double *x_s, *x_b;
        double *p;
} t_matrices;

typedef struct {
        int rows;
        double stepsize;
} t_read_matrix;

typedef struct {
        t_matrix Cr, Cq, Gammar, Gammaq;
        double p_s, p_b;
        t_matrix mayer, Vr;
        t_fbt fbt1, fbt2;
        t_composite comp, decomp;
} lm_data_type;



int out (const char *format, ...) {
        va_list arg;
        int done;

        va_start (arg, format);
        done = vfprintf (stdout, format, arg);
        fflush (stdout);
        va_end (arg);

        return done;
}

void die (const char *format, ...) {
        va_list arg;
        int done;

        va_start (arg, format);
        done = vfprintf (stdout, format, arg);
        fflush (stdout);
        va_end (arg);

        exit (-1);
}



t_matrix init_matrix (unsigned int size) {
        t_matrix matrix = {
                .size = size,
                .matrix = (double *) calloc ((size_t)(size), (size_t)(sizeof (do
```

lm_bin.c

```
uble))),
                .ptr = NULL,
                .ptr1 = NULL
        };
        if (!matrix.matrix)
                die ("\nMatrixinit");
        return matrix;
}

void cleanup_matrix (t_matrix matrix) {
        free (matrix.matrix);
}


#include "marquardt.c"


void PY (t_matrix Cr, t_matrix Gammar, t_matrix mayer) {
        unsigned int i;
        for (i = 0, Cr.ptr = Cr.matrix, Gammar.ptr = Gammar.matrix, mayer.ptr =
mayer.matrix; i < Cr.size; i++, Cr.ptr++, Gammar.ptr++, mayer.ptr++)
                *Cr.ptr = *mayer.ptr*(1.0+*Gammar.ptr);
}

void HNC (t_matrix Cr, t_matrix Gammar, t_matrix Vr) {
        unsigned int i;
        for (i = 0, Cr.ptr = Cr.matrix, Gammar.ptr = Gammar.matrix, Vr.ptr = Vr.
matrix; i < Cr.size; i++, Cr.ptr++, Gammar.ptr++, Vr.ptr++)
                *Cr.ptr = exp (*Vr.ptr+*Gammar.ptr)-*Gammar.ptr-1.0;
}

void FBT1 (t_fbt f) {
        unsigned int r, q;

        double *ptr11_r, *ptr12_r, *ptr21_r, *ptr22_r;
        double *ptr11_q, *ptr12_q, *ptr21_q, *ptr22_q;

        for (q = 0, f.bessel.ptr = f.bessel.matrix, ptr11_q = f.mq.matrix, ptr12
_q = ptr11_q+1, ptr21_q = ptr12_q+1, ptr22_q = ptr21_q+1;
                        q < f.gitter; q++, ptr11_q+=4, ptr12_q+=4, ptr21_q+=4, p
tr22_q+=4) {
                *ptr11_q = *ptr12_q = *ptr21_q = *ptr22_q = 0.0;
                for (r = 0, ptr11_r = f.mr.matrix, ptr12_r = ptr11_r+1, ptr21_r
= ptr12_r+1, ptr22_r = ptr21_r+1;
                                        r < f.gitter; r++, ptr11_r+=4, ptr12_r+=4, ptr21
_r+=4, ptr22_r+=4, f.bessel.ptr++) {
                        *ptr11_q += *ptr11_r**f.bessel.ptr;
                        *ptr12_q += *ptr12_r**f.bessel.ptr;
#ifndef SYMMETRIC_FT
                        *ptr21_q += *ptr21_r**f.bessel.ptr;
#endif
                        *ptr22_q += *ptr22_r**f.bessel.ptr;
                }
                *ptr11_q *= f.cons;
                *ptr12_q *= f.cons;
#ifdef SYMMETRIC_FT
                *ptr21_q = *ptr12_q;
#else
                *ptr21_q *= f.cons;
#endif
                *ptr22_q *= f.cons;
        }
}

void OZ (t_matrix Cq, t_matrix Gammaq, double ps, double pb) {
        unsigned int i;
        double z1;
```

lm_bin.c

```
        Gammaq.ptr = Gammaq.matrix;
        double *ptr11, *ptr12, *ptr21, *ptr22;
        ptr11 = Cq.matrix;
        ptr12 = Cq.matrix+1;
        ptr21 = Cq.matrix+2;
        ptr22 = Cq.matrix+3;
        for (i = 0; i < Gammaq.size; i+=4, Gammaq.ptr+=4, ptr11+=4, ptr12+=4, pt
r21+=4, ptr22+=4) {
                z1 = 1.0/(1.0-(*ptr11+*ptr12**ptr21*pb)*ps+*ptr22*pb*(-1.0+*ptr1
1*ps));
                *(Gammaq.ptr) = z1*(*ptr12**ptr21*pb+*ptr11*(*ptr11+*ptr12**ptr2
1*pb-*ptr11**ptr22*pb)*ps);
                *(Gammaq.ptr+1) = *ptr12*(-1.0+z1);
                *(Gammaq.ptr+2) = *ptr21*(-1.0+z1);
                *(Gammaq.ptr+3) = z1*(*ptr12**ptr21*ps+*ptr22*(*ptr22+*ptr12**pt
r21*ps-*ptr11**ptr22*ps)*pb);
        }
}

void FBT2 (t_fbt f) {
        unsigned i, m;

        double *ptr11_r, *ptr12_r, *ptr21_r, *ptr22_r;
        double *ptr11_q, *ptr12_q, *ptr21_q, *ptr22_q;

        for (i = 0, f.bessel.ptr = f.bessel.matrix, ptr11_r = f.mr.matrix, ptr12
_r = ptr11_r+1, ptr21_r = ptr12_r+1, ptr22_r = ptr21_r+1;
                                i < f.gitter; i++, ptr11_r+=4, ptr12_r+=4, ptr21_r+=4, p
tr22_r+=4) {
                *ptr11_r = *ptr12_r = *ptr21_r = *ptr22_r = 0.0;
                for (m = 0, ptr11_q = f.mq.matrix, ptr12_q = ptr11_q+1, ptr21_q
= ptr12_q+1, ptr22_q = ptr21_q+1;
                                                m < f.gitter; m++, f.bessel.ptr++, ptr11_q+=4, p
tr12_q+=4, ptr21_q+=4, ptr22_q+=4) {
                                *ptr11_r += *ptr11_q**f.bessel.ptr;
                                *ptr12_r += *ptr12_q**f.bessel.ptr;
#ifndef SYMMETRIC_FT
                                *ptr21_r += *ptr21_q**f.bessel.ptr;
#endif
                                *ptr22_r += *ptr22_q**f.bessel.ptr;
                }
                *ptr11_r *= f.cons;
                *ptr12_r *= f.cons;
#ifdef SYMMETRIC_FT
                *ptr21_r = *ptr12_r;
#else
                *ptr21_r *= f.cons;
#endif
                *ptr22_r *= f.cons;
        }
}

void symmetry (t_matrix matrix, t_matrix symmetric_matrix) {
        unsigned int i;

        matrix.ptr = matrix.matrix;
        symmetric_matrix.ptr = symmetric_matrix.matrix;
        for (i = 0; i < symmetric_matrix.size; i+=4, matrix.ptr+=4, symmetric_ma
trix.ptr+=4) {
                *(symmetric_matrix.ptr) = *matrix.ptr;
                *(symmetric_matrix.ptr+1) = 0.5*(*(matrix.ptr+1)+*(matrix.ptr+2)
);
                *(symmetric_matrix.ptr+2) = *(symmetric_matrix.ptr+1);
                *(symmetric_matrix.ptr+3) = *(matrix.ptr+3);
        }
}
```

lm_bin.c

```
double *alloc_matrix_1 (int anz_1) {
        double *m;
        unsigned int i;

        m = (double *) malloc ((size_t) (anz_1 * sizeof (double)));
        if (!m)
                die ("allocation failure in alloc_matrix_1");

        for (i = 0; i < anz_1; i++)
                m[i] = 0.0;

        return m;
}

double **alloc_matrix_2(int anz_1,int anz_2) {
        unsigned int i;
        double **m;

        m = (double **) malloc((size_t) (anz_1 * sizeof(double *)));
        if (!m)
                die ("allocation failure in alloc_matrix_2");
        for (i = 0; i < anz_1; i++)
                m[i] = alloc_matrix_1(anz_2);
        return m;
}

void free_matrix_1(double *m) {
        free(m);
}

void free_matrix_2(double **m,int anz_1) {
        unsigned int i;

        for (i = 0; i < anz_1; i++)
                free_matrix_1(m[i]);
        free(m);
}

unsigned int find_i (double *r_, double X, unsigned int gitter) {
        unsigned int i = gitter-1, j;
        double minimum = r_[i];

        for (j = 0; j < gitter; j++) {
#ifdef __FIND_I__
                out ("\ni:%d j:%d r_[j]:%lf X:%lf min:%lf, fabs:%lf", i, j, r_[j], X, minimum
, fabs(r_[j]-X));
#endif
                if (fabs(r_[j]-X) <= minimum) {
                        minimum = fabs(r_[j]-X);
                        i = j;
                }
        }
#ifdef __FIND_I__
        out ("\n[find_i: r%Lf X%Lf min%Lf i%d]", r_[i], X, minimum, i);
#endif
        return i;
}

void invert (double **matrix, double **inverse, unsigned int dimension) {
        int k, i, j;
        double min_value = 0.0000000001;

        for (i = 0; i < dimension; i++)
                if (matrix[i][i] < min_value)
                        die ("Invertierfehler: (Diagonalelemet %d) = %.64lf", i, matrix[i][i]
);

        double **unity = alloc_matrix_2 (dimension, dimension);
        for (i = 0; i < dimension; i++) for (j = 0; j < dimension; j++)
```

lm_bin.c

```
                        unity[i][j] = (double)(!(i-j));

        double xmult;
        for (i = 0; i < dimension-1; i++) {
                for (j = i+1; j < dimension; j++) {
                        xmult = matrix[j][i]/matrix[i][i];
                        for (k = i+1; k < dimension; k++) {
                                matrix[j][k] -= xmult*matrix[i][k];
                        }
                        matrix[j][i] = xmult;
                        for (k = 0; k < dimension; k++) {
                                unity[j][k] -= xmult*unity[i][k];
                        }
                }
        }
        double sum;
        for (i = 0; i < dimension; i++) {
                inverse[dimension-1][i] = unity[dimension-1][i]/matrix[dimension
-1][dimension-1];
                for (j = dimension-2; j >= 0; j--) {
                        sum = unity[j][i];
                        for (k = j+1; k < dimension; k++) {
                                sum -= matrix[j][k]*inverse[k][i];
                        }
                        inverse[j][i] = sum/matrix[j][j];
                }
        }

        free_matrix_2 (unity, dimension);
}

void pq_init (double r_i_a[], unsigned int nu, t_matrix r_, t_matrix P_, t_matri
x Q_) {
#ifdef __PQ_INIT__
        out ("\n=> PQ generieren");
#endif
        unsigned int i, m, n;
        unsigned int gitter = r_.size;

        int i_[nu];
        for (n = 0; n < nu; n++) {
                i_[n] = find_i (r_.matrix, r_i_a[n], gitter);
#ifdef __PQ_INIT__
                out ("[%d %lf]", i_[n], r_i_a[n]);
#endif
                        }

#ifdef __PQ_INIT__
        out ("P_n^i ");
#endif
        double **P__ = alloc_matrix_2 (nu, gitter);
        for (n = 0; n < nu; n++)
                if (n == 0) {
                        for (i = 0; i < gitter; i++) {
                                P__[n][i] = 0.0;
                                if (i <= i_[1]) P__[n][i] = (double)(i_[1]-i)/(d
ouble)(i_[1]);
                        }
                } else if (n == 1) {
                        for (i = 0; i < gitter; i++) {
                                P__[n][i] = 0.0;
                                if (0 <= i && i <= i_[1]) P__[n][i] = (double)(i
)/(double)(i_[1]);
                                else if (i_[1] <= i && i <= i_[2]) P__[n][i] = (
double)(i_[2]-i)/(double)(i_[2]-i_[1]);
                        }
                } else {
                        for (i = 0; i < gitter; i++) {
                                P__[n][i] = 0.0;
```

lm_bin.c

```
390                                 if (i_[n-1] <= i && i <= i_[n]) P__[n][i] = (dou
    ble)(i-i_[n-1])/(-i_[n-1]+i_[n]);
                                    else if (i_[n] <= i && i <= i_[n+1]) P__[n][i] =
     (double)(i_[n+1]-i)/(i_[n+1]-i_[n]);
                                }
                        }

395     #ifdef __PQ_INIT__
                out ("Q_n^i ");
        #endif
                double **Q__ = alloc_matrix_2 (nu, gitter);

400             double sum;
                double **PP = alloc_matrix_2 (nu, nu);
                for (m = 0; m < nu; m++) for (n = 0; n < nu; n++) {
                        sum = 0.0;
                        for (i = 0; i < gitter; i++)
405                             sum += P__[m][i]*P__[n][i];
                        PP[m][n] = sum;
                }

                double **RR = alloc_matrix_2 (nu, nu);
410             invert (PP, RR, nu);

                for (m = 0; m < nu; m++)
                        for (i = 0; i < gitter; i++) {
                                Q__[m][i] = 0.0;
415                             for (n = 0; n < nu; n++)
                                        Q__[m][i] += RR[m][n]*P__[n][i];
                        }

                for (i = 0, P_.ptr = P_.matrix, Q_.ptr = Q_.matrix; i < gitter; i++)
420                     for (n = 0; n < nu; n++, P_.ptr++, Q_.ptr++) {
                                *P_.ptr = (double)P__[n][i];
                                *Q_.ptr = (double)Q__[n][i];
        #ifdef __PQ_INIT__
                                out ("\n(p%lf q%lf)", P__[n][i], Q__[n][i]);
425     #endif
                        }


        }

430     void decomposite (t_composite d) {
                unsigned int i, m;

        #ifdef __DECOMPOSITE_DEBUG
                printf ("\n=> Zerlegung: Grobanteil 1"); fflush (stdout);
435     #endif
                for (i = 0, d.coarse.ptr = d.coarse.matrix; i < d.coarse.size; i++, d.co
    arse.ptr++)
                        *d.coarse.ptr = 0.0;

        #ifdef __DECOMPOSITE_DEBUG
440             printf (" 2"); fflush (stdout);
        #endif
                for (i = 0, d.function.ptr = d.function.matrix, d.Q_.ptr = d.Q_.matrix;
    i < d.gitter; i++, d.function.ptr+=4)
                        for (m = 0, d.coarse.ptr = d.coarse.matrix; m < d.nu; m++, d.Q_.
    ptr++, d.coarse.ptr+=4) {
        #ifdef __DECOMPOSITE_DEBUG
445                             printf (" (%f*%f)", *(d.function.ptr), *d.Q_.ptr); fflush
     (stdout);
        #endif
                                *(d.coarse.ptr) += *(d.function.ptr)**d.Q_.ptr;
                                *(d.coarse.ptr+1) += *(d.function.ptr+1)**d.Q_.ptr;
                                *(d.coarse.ptr+2) += *(d.function.ptr+2)**d.Q_.ptr;
450                             *(d.coarse.ptr+3) += *(d.function.ptr+3)**d.Q_.ptr;
                        }
```

lm_bin.c

```
        #ifdef __DECOMPOSITE_DEBUG
                printf (" Feinanteil 1"); fflush (stdout);
455     #endif
                for (i = 0, d.fine.ptr = d.fine.matrix, d.function.ptr = d.function.matr
    ix; i < d.fine.size; i++, d.fine.ptr++, d.function.ptr++)
                        *d.fine.ptr = *d.function.ptr;

        #ifdef __DECOMPOSITE_DEBUG
460             printf (" 2"); fflush (stdout);
        #endif
                for (i=0, d.fine.ptr=d.fine.matrix, d.P_.ptr=d.P_.matrix; i < d.gitter;
    i++, d.fine.ptr+=4)
                        for (m = 0, d.coarse.ptr = d.coarse.matrix; m < d.nu; m++, d.coa
    rse.ptr+=4, d.P_.ptr++) {
                                *(d.fine.ptr) -= *(d.coarse.ptr)**d.P_.ptr;
465                             *(d.fine.ptr+1) -= *(d.coarse.ptr+1)**d.P_.ptr;
                                *(d.fine.ptr+2) -= *(d.coarse.ptr+2)**d.P_.ptr;
                                *(d.fine.ptr+3) -= *(d.coarse.ptr+3)**d.P_.ptr;
                        }
        }
470
        void composite (t_composite c) {
                unsigned int i, m;

                for (i = 0, c.function.ptr = c.function.matrix, c.P_.ptr = c.P_.matrix,
    c.fine.ptr = c.fine.matrix; i < c.gitter; i++, c.fine.ptr+=4, c.function.ptr+=4)
    {
475     #ifdef _COMPOSITE_
                        if (i <= 3)
                                out ("\ni:%d fine:%lf", i, *c.function.ptr);
        #endif
                        *(c.function.ptr) = *(c.fine.ptr);
480                     *(c.function.ptr+1) = *(c.fine.ptr+1);
                        *(c.function.ptr+2) = *(c.fine.ptr+2);
                        *(c.function.ptr+3) = *(c.fine.ptr+3);
                        for (m = 0, c.coarse.ptr = c.coarse.matrix; m < c.nu; m++, c.coa
    rse.ptr+=4, c.P_.ptr++) {
        #ifdef _COMPOSITE_
485                             if (i <= 3)
                                        out (" cp:%lf", *c.coarse.ptr**c.P_.ptr);
        #endif
                                *(c.function.ptr) += *(c.coarse.ptr)**c.P_.ptr;
                                *(c.function.ptr+1) += *(c.coarse.ptr+1)**c.P_.ptr;
490                             *(c.function.ptr+2) += *(c.coarse.ptr+2)**c.P_.ptr;
                                *(c.function.ptr+3) += *(c.coarse.ptr+3)**c.P_.ptr;
                        }
        #ifdef _COMPOSITE_
                        if (i <= 3)
495                             out (" -> %lf", *c.function.ptr);
        #endif
                }
        }

500     #ifdef _BASIS_
        void basis_out (t_basis_out b) {
                char filename[255];
                FILE *basis_file;
                unsigned int n, i;
505
                for (n = 0; n < b.nu; n++) {
                        sprintf (filename, "P_%d.dat", n);
                        basis_file = fopen (filename, "w");
                        for (i = 0; i < b.gitter; i++)
510                             fprintf (basis_file, "\n%f %f", b.r_.matrix[i], b.P_.matr
    ix[i*b.nu+n]);
                        fclose (basis_file);
                }
        }
        #endif
```

lm_bin.c

```
515

    void lm_evaluate_default (double* coarse, int nu_, double* diff, void *data, int
     *info) {
            unsigned int i;
            lm_data_type *d = (lm_data_type*) data;
520
            for (i = 0, d->comp.coarse.ptr = d->comp.coarse.matrix; i < nu_; i++, d-
    >comp.coarse.ptr++)
                    *d->comp.coarse.ptr = coarse[i];

            composite (d->comp);
525
    #if APPROXIMATION == 1
            PY (d->Cr, d->Gammar, d->mayer);
    #elif APPROXIMATION == 2
            HNC (d->Cr, d-> Gammar, d->Vr);
530 #endif

            d->fbt1.mr = d->Cr; d->fbt1.mq = d->Cq;
            FBT1 (d->fbt1);
            OZ (d->Cq, d->Gammaq, d->p_s, d->p_b);
535         d->fbt2.mq = d->Gammaq; d->fbt2.mr = d->Gammar;
            FBT2 (d->fbt2);
            decomposite (d->decomp);


540         for (i = 0, d->comp.coarse.ptr = d->comp.coarse.matrix, d->decomp.coarse
    .ptr = d->decomp.coarse.matrix; i < nu_; i+=4, d->comp.coarse.ptr+=4, d->decomp.
    coarse.ptr+=4) {
                    diff[i] = *d->comp.coarse.ptr-*d->decomp.coarse.ptr;
                    diff[i+1] = *(d->comp.coarse.ptr+1)-*(d->decomp.coarse.ptr+1);
                    diff[i+2] = *(d->comp.coarse.ptr+2)-*(d->decomp.coarse.ptr+2);
                    diff[i+3] = *(d->comp.coarse.ptr+3)-*(d->decomp.coarse.ptr+3);
545                 if (isnan (diff[i]) || isnan (diff[i+1]) || isnan (diff[i+2]) ||
     isnan (diff[i+3]))
                            *info = -1;
            }

            *info = *info;
550 }



    void write_matrix (t_matrix matrix, t_matrix vector, char *name, int pic) {
555         char filename[255];
            sprintf (filename, "%s__pic%d.dat", name, pic);
            FILE *matrix_file = fopen (filename, "w");

            if (!matrix_file)
560                 die ("\nwrite_matrix: %s %d", name, pic);

            unsigned int i;

            for (i = 0, matrix.ptr = matrix.matrix, vector.ptr = vector.matrix; i <
    matrix.size; i+=4, matrix.ptr+=4, vector.ptr++)
565                 fprintf (matrix_file, "%.16f %.16f %.16f %.16f %.16f\n", *vector.ptr, *
    (matrix.ptr), *(matrix.ptr+1), *(matrix.ptr+2), *(matrix.ptr+3));

            fclose (matrix_file);
    }

570 void coarse_fine_out (t_coarse_fine cf) {
            char filename_coarse[255], filename_fine[255];
            sprintf (filename_coarse, "coarse__pic%d", *cf.pic);
            sprintf (filename_fine, "fine__pic%d", *cf.pic);
            FILE *coarse_file = fopen (filename_coarse, "w"), *fine_file = fopen (f
    ilename_fine, "w");
575
```

lm_bin.c

```
            unsigned int i, m;

            for (i = 0; i < cf.gitter; i++)
                    fprintf (fine_file, "\n%.16f %.16f %.16f %.16f %.16f", cf.r_.matrix[i],

580                                         cf.fine.matrix[i*4+0*2+0], cf.fine.matrix[i*4+0*
    2+1], cf.fine.matrix[i*4+1*2+0], cf.fine.matrix[i*4+1*2+1]);

            for (m = 0; m < cf.nu; m++)
                    fprintf (coarse_file, "\n%.16f %.16f %.16f %.16f %.16f", cf.r_.matrix[i
    ],
                                                cf.coarse.matrix[i*4+0*2+0], cf.coarse.matrix[i*
    4+0*2+1], cf.coarse.matrix[i*4+1*2+0], cf.coarse.matrix[i*4+1*2+1]);
585
            fclose (coarse_file);
            fclose (fine_file);
    }

590 void matrix_out (t_matrices m) {
            unsigned int i;
            t_matrix tmp = init_matrix (m.Gammar.size);

            for (i = 0, tmp.ptr = tmp.matrix, m.Gammar.ptr = m.Gammar.matrix, m.Cr.p
    tr = m.Cr.matrix; i < m.Cr.size; i++, m.Cr.ptr++, m.Gammar.ptr++, tmp.ptr++)
595                 *tmp.ptr = *m.Gammar.ptr + *m.Cr.ptr;

            write_matrix (tmp, m.r_, "Hr", *m.pic);
            for (i = 0, tmp.ptr = tmp.matrix, m.Gammaq.ptr = m.Gammaq.matrix, m.Cq.p
    tr = m.Cq.matrix; i < m.Cq.size; i++, m.Cq.ptr++, m.Gammaq.ptr++, tmp.ptr++)
                    *tmp.ptr = *m.Gammaq.ptr + *m.Cq.ptr;
600         write_matrix (tmp, m.q_, "Hq", *m.pic);

            write_matrix (m.Gammar, m.r_, "Gammar", *m.pic);
            write_matrix (m.Gammaq, m.q_, "Gammaq", *m.pic);

605         write_matrix (m.Cr, m.r_, "Cr", *m.pic);
            write_matrix (m.Cq, m.q_, "Cq", *m.pic);

            for (i = 0, tmp.ptr = tmp.matrix, m.Gammaq.ptr = m.Gammaq.matrix, m.Cq.p
    tr = m.Cq.matrix; i < m.Cq.size; i+=4, m.Cq.ptr+=4, m.Gammaq.ptr+=4, tmp.ptr+=4)
     {
                    *(tmp.ptr) = (*m.x_s) + (*m.x_s)*(*m.x_b)*(*m.p)*(*m.Gammaq.ptr+
    *m.Cq.ptr);
610                 *(tmp.ptr+1) = (*m.x_s)*(*m.x_b)*(*m.p)*(*(m.Gammaq.ptr+1)+*(m.C
    q.ptr+1));
                    *(tmp.ptr+2) = (*m.x_s)*(*m.x_b)*(*m.p)*(*(m.Gammaq.ptr+2)+*(m.C
    q.ptr+2));
                    *(tmp.ptr+3) = (*m.x_b) + (*m.x_s)*(*m.x_b)*(*m.p)*(*(m.Gammaq.p
    tr+3)+*(m.Cq.ptr+3));
            }
            write_matrix (tmp, m.q_, "Sq", *m.pic);
615
            cleanup_matrix (tmp);
    }



620
    void spline (t_matrix x, t_matrix y, t_matrix y2) {
            unsigned int i, k;
            double p, qn, sig, un;
    #ifdef __SPLINE__
625         printf ("=> Init"); fflush (stdout);
    #endif
            t_matrix u = init_matrix (x.size-1);


            u.matrix[0] = 0.0;
630
    #ifdef __SPLINE__
            printf (" for"); fflush (stdout);
```

lm_bin.c

```
    #endif
            for (i = 1; i < x.size-1; i++) {
635                 sig = (x.matrix[i]-x.matrix[i-1])/(x.matrix[i+1]-x.matrix[i-1]);
                    p = sig*y2.matrix[i-1]+2.0;
                    y2.matrix[i] = (sig-1.0)/p;
                    u.matrix[i] = (y.matrix[i+1]-y.matrix[i])/(x.matrix[i+1]-x.matri
    x[i])-(y.matrix[i]-y.matrix[i-1])/(x.matrix[i]-x.matrix[i-1]);
                    u.matrix[i] = (6.0*u.matrix[i]/(x.matrix[i+1]-x.matrix[i-1])-sig
    *u.matrix[i-1])/p;
640         }

            qn = un = 0.0;

    #ifdef __SPLINE__
645         printf (" for"); fflush (stdout);
    #endif
            y2.matrix[y2.size-1] = (un-qn*u.matrix[y2.size-2])/(qn*y2.matrix[y2.size
    -2]+1.0);
            for (k = y2.size-2; k > 0; k--)
                    y2.matrix[k] = y2.matrix[k]*y2.matrix[k+1]+u.matrix[k];
650
    #ifdef __SPLINE__
            printf (" Cleanup"); fflush (stdout);
    #endif
            cleanup_matrix (u);
655 }

    double splint (t_matrix x, t_matrix y, t_matrix y2, double x_) {
            int klo = 0, khi = x.size-1, k;
            double h, b, a;
660
            while (khi-klo > 1) {
                    k = (khi+klo) >> 1;
                    if (x.matrix[k] > x_)
                            khi = k;
665                 else
                            klo = k;
            }

            h = x.matrix[khi]-x.matrix[klo];
670         if (h == 0.0)
                    return (double)(0.0);
            a = (x.matrix[khi]-x_)/h;
            b = (x_-x.matrix[klo])/h;
            return (double)(a*y.matrix[klo]+b*y.matrix[khi]+((a*a*a-a)*y2.matrix[klo
    ]+(b*b*b-b)*y2.matrix[khi])*(h*h)/6.0);
675 }

    void splie2 (t_matrix x1a, t_matrix x2a, t_matrix ya, t_matrix y2a) {
            unsigned int j;
            t_matrix ya_tmp;
680         t_matrix y2a_tmp;
            unsigned int m = x1a.size, n = x2a.size;
            ya_tmp.size = y2a_tmp.size = n;

    #ifdef __SPLIE2__
685         printf ("\n"); fflush (stdout);
    #endif

            for (j = 0, ya.ptr = ya.matrix, y2a.ptr = y2a.matrix; j < m; j++, ya.ptr
    +=n, y2a.ptr+=n) {
                    ya_tmp.matrix = ya.ptr;
690                 y2a_tmp.matrix = y2a.ptr;
    #ifdef __SPLIE2__
                    printf ("\nsplie2"); fflush (stdout);
    #endif
                    spline (x2a, ya_tmp, y2a_tmp);
695 #ifdef __SPLIE2__
                    printf ("ok"); fflush (stdout);
```

lm_bin.c

```
    #endif
            }
    }
700
    double splin2 (t_matrix x1a, t_matrix x2a, t_matrix ya, t_matrix y2a, double x1,
     double x2) {
            double y;
            unsigned int j;

705 #ifdef __SPLIN2__
            printf ("\ninit"); fflush (stdout);
    #endif
            unsigned int m = x1a.size;
            unsigned int n = x2a.size;
710         t_matrix ytmp = init_matrix (m);
            t_matrix yytmp = init_matrix (m);
            t_matrix ya_tmp, y2a_tmp;
            ya_tmp.size = y2a_tmp.size = n;

715 #ifdef __SPLIN2__
            printf (" for"); fflush (stdout);
    #endif
            for (j = 0, ya.ptr = ya.matrix, y2a.ptr = y2a.matrix, yytmp.ptr = yytmp.
    matrix; j < m; j++, ya.ptr+=n, y2a.ptr+=n, yytmp.ptr++) {
    #ifdef __SPLIN2__
720                 printf ("."); fflush (stdout);
    #endif
                    ya_tmp.matrix = ya.ptr;
                    y2a_tmp.matrix = y2a.ptr;
                    *yytmp.ptr = splint (x2a, ya_tmp, y2a_tmp, x2);
725         }
    #ifdef __SPLIN2__
            printf (" spline"); fflush (stdout);
    #endif
            spline (x1a, yytmp, ytmp);
730 #ifdef __SPLIN2__
            printf (" splint"); fflush (stdout);
    #endif
            y = splint (x1a, yytmp, ytmp, x1);

735 #ifdef __SPLIN2__
            printf (" cleanup"); fflush (stdout);
    #endif
            cleanup_matrix (ytmp);
            cleanup_matrix (yytmp);
740 #ifdef __SPLIN2__
            printf (" ok"); fflush (stdout);
    #endif

            return (y);
745 }



    t_read_matrix read_matrix (t_matrix matrix, char *name) {
750         FILE *matrix_file = fopen (name, "r");
            unsigned int i = 1;
            double rmax = 0;
            t_read_matrix result;

755         matrix.ptr = matrix.matrix;

            result.stepsize = rmax/i;
            result.rows = i;
            fclose (matrix_file);
760
            return result;
    }
```

lm_bin.c

```
    void interpol_matrix (t_matrix matrix, char *fileprefix, t_matrix q_, t_matrix n
    , double n_target) {
765 #define _INTERPOL_MATRIX_
            out ("\n-> S(q) extrapolieren");

            unsigned int i, j;
            unsigned int lines_in = 0;
770         FILE *infile;
            char filename[255];

            out (" lines_in:");
            sprintf (filename, "%s_%lf", fileprefix, n.matrix[0]);
775         infile = fopen (filename, "r");
            if (!infile)
                    die ("startgamma: infile");
            double tmp;
            do {
780                 fscanf (infile, "\n%lf %lf %lf %lf %lf", &tmp, &tmp, &tmp, &tmp, &tmp)
    ;
                    lines_in++;
            } while (!feof (infile));
            lines_in--;
            out ("%d", lines_in);
785         fclose (infile);

            out (" Init");
            t_matrix sq_in_11 = init_matrix (lines_in*n.size),
                     sq_in_12 = init_matrix (lines_in*n.size),
790                  sq_in_21 = init_matrix (lines_in*n.size),
                     sq_in_22 = init_matrix (lines_in*n.size);
            t_matrix q_in = init_matrix (lines_in);
            t_matrix deriv_11 = init_matrix (lines_in*n.size),
                     deriv_12 = init_matrix (lines_in*n.size),
795                  deriv_21 = init_matrix (lines_in*n.size),
                     deriv_22 = init_matrix (lines_in*n.size);
            t_matrix sqn_11 = init_matrix (q_.size),
                     sqn_12 = init_matrix (q_.size),
                     sqn_21 = init_matrix (q_.size),
800                  sqn_22 = init_matrix (q_.size);

            out (" Matrix einlesen");
            for (i = 0, n.ptr = n.matrix; i < n.size; i++, n.ptr++) {
                    out (" %s_%lf", fileprefix, *n.ptr);
805                 sprintf (filename, "%s_%lf", fileprefix, *n.ptr);
                    infile = fopen (filename, "r");
                    if (!infile)
                            die ("\nstartgamma: infile\n");
                    for (j = 0, sq_in_11.ptr = sq_in_11.matrix+i*lines_in, sq_in_12.
    ptr = sq_in_12.matrix+i*lines_in,
810                                 sq_in_21.ptr = sq_in_21.matrix+i*lines_in, sq_in
    _22.ptr = sq_in_22.matrix+i*lines_in, q_in.ptr = q_in.matrix;
                                    j < lines_in; j++, sq_in_11.ptr++, sq_in_12.ptr+
    +, sq_in_21.ptr++, sq_in_22.ptr++, q_in.ptr++)
                            fscanf (infile, "\n%lf %lf %lf %lf %lf", q_in.ptr, sq_in_11.pt
    r, sq_in_12.ptr, sq_in_21.ptr, sq_in_22.ptr);
                    fclose (infile);
            }
815
            out (" Ableitung");
            splie2 (n, q_in, sq_in_11, deriv_11);
            splie2 (n, q_in, sq_in_12, deriv_12);
            splie2 (n, q_in, sq_in_21, deriv_21);
820         splie2 (n, q_in, sq_in_22, deriv_22);

            out (" Interpolieren: n=%lf", n_target);
            for (i = 0, sqn_11.ptr = sqn_11.matrix, sqn_12.ptr = sqn_12.matrix, sqn_
    21.ptr = sqn_21.matrix, sqn_22.ptr = sqn_22.matrix, q_.ptr = q_.matrix;
                            i < sqn_11.size; i++, sqn_11.ptr++, sqn_12.ptr++, sqn_21
    .ptr++, sqn_22.ptr++, q_.ptr++)
```

lm_bin.c

```
825                 if (*q_.ptr < q_in.matrix[lines_in-1]) {
                            *sqn_11.ptr = splin2 (n, q_in, sq_in_11, deriv_11, n_tar
    get, *q_.ptr);
                            *sqn_12.ptr = splin2 (n, q_in, sq_in_12, deriv_12, n_tar
    get, *q_.ptr);
                            *sqn_21.ptr = splin2 (n, q_in, sq_in_21, deriv_21, n_tar
    get, *q_.ptr);
                            *sqn_22.ptr = splin2 (n, q_in, sq_in_22, deriv_22, n_tar
    get, *q_.ptr);
830                 } else
                            *sqn_11.ptr = *sqn_12.ptr = *sqn_21.ptr = *sqn_22.ptr =
    sqn_11.matrix[lines_in-2];

    #ifdef _INTERPOL_MATRIX_
            out (" out");
835         FILE *outfile = fopen ("interpol_matrix", "w");
            if (!outfile)
                    die ("\nstartgamma: outfile\n");
            for (i = 0, q_.ptr = q_.matrix, sqn_11.ptr = sqn_11.matrix, sqn_12.ptr =
     sqn_12.matrix, sqn_21.ptr = sqn_21.matrix, sqn_22.ptr = sqn_22.matrix;
                            i < q_.size; i++, q_.ptr++, sqn_11.ptr++, sqn_12.ptr++,
    sqn_21.ptr++, sqn_22.ptr++)
840                 fprintf (outfile, "\n%lf %lf %lf %lf %lf", *q_.ptr, *sqn_11.ptr, *sqn_
    12.ptr, *sqn_21.ptr, *sqn_22.ptr);
            fclose (outfile);
    #endif


            out (" Cleanup");
845         cleanup_matrix (sq_in_11);
            cleanup_matrix (sq_in_12);
            cleanup_matrix (sq_in_21);
            cleanup_matrix (sq_in_22);
            cleanup_matrix (q_in);
850         cleanup_matrix (deriv_11);
            cleanup_matrix (deriv_12);
            cleanup_matrix (deriv_21);
            cleanup_matrix (deriv_22);
            cleanup_matrix (sqn_11);
855         cleanup_matrix (sqn_12);
            cleanup_matrix (sqn_21);
            cleanup_matrix (sqn_22);
    }

860 void interpol_sq (t_matrix Sq, t_matrix q_, t_matrix n, double n_target) {
    #define _INTERPOL_SQ_
            out ("\n-> S(q) extrapolieren");

            unsigned int i, j;
865         unsigned int lines_in = 0;
            FILE *infile;
            char filename[255];

            out (" lines_in:");
870         sprintf (filename, "sq_%lf", n.matrix[0]);
            infile = fopen (filename, "r");
            if (!infile)
                    die ("startgamma: infile");
            double tmp;
875         do {
                    fscanf (infile, "\n%lf %lf %lf %lf %lf", &tmp, &tmp, &tmp, &tmp, &tmp)
    ;
                    lines_in++;
            } while (!feof (infile));
            lines_in--;
880         out ("%d", lines_in);
            fclose (infile);

            out (" Init");
            t_matrix sq_in_11 = init_matrix (lines_in*n.size),
```

lm_bin.c

```
                    sq_in_12 = init_matrix (lines_in*n.size),
                    sq_in_21 = init_matrix (lines_in*n.size),
                    sq_in_22 = init_matrix (lines_in*n.size);
            t_matrix q_in = init_matrix (lines_in);
            t_matrix deriv_11 = init_matrix (lines_in*n.size),
                    deriv_12 = init_matrix (lines_in*n.size),
                    deriv_21 = init_matrix (lines_in*n.size),
                    deriv_22 = init_matrix (lines_in*n.size);
            t_matrix sqn_11 = init_matrix (q_.size),
                    sqn_12 = init_matrix (q_.size),
                    sqn_21 = init_matrix (q_.size),
                    sqn_22 = init_matrix (q_.size);

            out (" S(q) einlesen");
            for (i = 0, n.ptr = n.matrix; i < n.size; i++, n.ptr++) {
                    out (" sq_%lf", *n.ptr);
                    sprintf (filename, "sq_%lf", *n.ptr);
                    infile = fopen (filename, "r");
                    if (!infile)
                            die ("\nstartgamma: infile\n");
                    for (j = 0, sq_in_11.ptr = sq_in_11.matrix+i*lines_in, sq_in_12.
    ptr = sq_in_12.matrix+i*lines_in,
                                    sq_in_21.ptr = sq_in_21.matrix+i*lines_in, sq_in
    _22.ptr = sq_in_22.matrix+i*lines_in, q_in.ptr = q_in.matrix;
                                    j < lines_in; j++, sq_in_11.ptr++, sq_in_12.ptr+
    +, sq_in_21.ptr++, sq_in_22.ptr++, q_in.ptr++)
                            fscanf (infile, "\n%lf %lf %lf %lf %lf", q_in.ptr, sq_in_11.pt
    r, sq_in_12.ptr, sq_in_21.ptr, sq_in_22.ptr);
                    fclose (infile);
            }

            out (" Ableitung");
            splie2 (n, q_in, sq_in_11, deriv_11);
            splie2 (n, q_in, sq_in_12, deriv_12);
            splie2 (n, q_in, sq_in_21, deriv_21);
            splie2 (n, q_in, sq_in_22, deriv_22);

            out (" Interpolieren: n=%lf", n_target);
            for (i = 0, sqn_11.ptr = sqn_11.matrix, sqn_12.ptr = sqn_12.matrix, sqn_
    21.ptr = sqn_21.matrix, sqn_22.ptr = sqn_22.matrix, q_.ptr = q_.matrix;
                    i < sqn_11.size; i++, sqn_11.ptr++, sqn_12.ptr++, sqn_21
    .ptr++, sqn_22.ptr++, q_.ptr++)
                    if (*q_.ptr < q_in.matrix[lines_in-1]) {
                            *sqn_11.ptr = splin2 (n, q_in, sq_in_11, deriv_11, n_tar
    get, *q_.ptr);
                            *sqn_12.ptr = splin2 (n, q_in, sq_in_12, deriv_12, n_tar
    get, *q_.ptr);
                            *sqn_21.ptr = splin2 (n, q_in, sq_in_21, deriv_21, n_tar
    get, *q_.ptr);
                            *sqn_22.ptr = splin2 (n, q_in, sq_in_22, deriv_22, n_tar
    get, *q_.ptr);
                    } else
                            *sqn_11.ptr = *sqn_12.ptr = *sqn_21.ptr = *sqn_22.ptr =
    1.0;

    #ifdef _INTERPOL_SQ_
            out (" out");
            FILE *outfile = fopen ("interpol_sq", "w");
            if (!outfile)
                    die ("\nstartgamma: outfile\n");
            for (i = 0, q_.ptr = q_.matrix, sqn_11.ptr = sqn_11.matrix, sqn_12.ptr =
     sqn_12.matrix, sqn_21.ptr = sqn_21.matrix, sqn_22.ptr = sqn_22.matrix;
                    i < q_.size; i++, q_.ptr++, sqn_11.ptr++, sqn_12.ptr++,
    sqn_21.ptr++, sqn_22.ptr++)
                    fprintf (outfile, "\n%lf %lf %lf %lf %lf", *q_.ptr, *sqn_11.ptr, *sqn_
    12.ptr, *sqn_21.ptr, *sqn_22.ptr);
            fclose (outfile);
    #endif
```

lm_bin.c

```
            out (" Cleanup");
            cleanup_matrix (sq_in_11);
            cleanup_matrix (sq_in_12);
            cleanup_matrix (sq_in_21);
            cleanup_matrix (sq_in_22);
            cleanup_matrix (q_in);
            cleanup_matrix (deriv_11);
            cleanup_matrix (deriv_12);
            cleanup_matrix (deriv_21);
            cleanup_matrix (deriv_22);
            cleanup_matrix (sqn_11);
            cleanup_matrix (sqn_12);
            cleanup_matrix (sqn_21);
            cleanup_matrix (sqn_22);
    }

    void sq_to_cq (t_matrix sq, t_matrix cq, double n, double x_s, double x_b) {
            unsigned int i;

            double *sq11, *sq12, *sq21, *sq22;
            double *cq11, *cq12, *cq21, *cq22;
            double p1 = x_s*n, p2 = x_b*n,
                    x11 = x_s*x_s*n, x12 = x_s*x_b*n, x21 = x_b*x_s*n, x22 = x_b*x_b*
    n;

            for (i = 0, sq11 = sq.matrix, sq12 = sq11+1, sq21 = sq12+1, sq22 = sq21+
    1, cq11 = cq.matrix, cq12 = cq11+1, cq21 = cq12+1, cq22 = cq21+1;
                            i < sq.size; i+=4, sq11+=4, sq12+=4, sq21+=4, sq22+=4, c
    q11+=4, cq12+=4, cq21+=4, cq22+=4) {
                    *cq11 = (-p2*(-1.0+*sq11+*sq12**sq21)+p2*(-1.0+*sq11)**sq22-*sq2
    1*x12+(-1.0+*sq11)*x22)/(p2*(-1.0+*sq22)*x11-p2**sq12*x21-x12*x21+x11*x22+
                                                    p1*(-p2*(-1.0+*sq11+*sq12**sq21)+p2*(-1.0+*sq11)
    **sq22-*sq21*x12+(-1.0+*sq11)*x22));
                    *cq12 = ((-1.0+*sq22)*x12-*sq12*x22)/(x12*x21+p2*(x11-*sq22*x11+
    *sq12*x21)-x11*x22+
                                                    p1*(p2*(-1.0+*sq11+*sq12**sq21+*sq22-*sq11**sq22
    )+*sq21*x12+x22-*sq11*x22));
                    *cq21 = (-*sq21*x11+(-1.0+*sq11)*x21)/(x12*x21+p2*(x11-*sq22*x11
    +*sq12*x21)-x11*x22+p1*(p2*(-1.0+*sq11+*sq12**sq21+*sq22-*sq11**sq22)+*sq21*x12+
    x22-*sq11*x22));
                    *cq22 = (-p1*(-1.0+*sq11+*sq12**sq21)+p1*(-1.0+*sq11)**sq22+(-1.
    0+*sq22)*x11-*sq12*x21)/(p2*(-1.0+*sq22)*x11-p2**sq12*x21-x12*x21+x11*x22+p1*(-p
    2*(-1.0+*sq11
                                                            +*sq12**sq21)+p2*(-1.0+*sq11)**s
    q22-*sq21*x12+(-1.0+*sq11)*x22));

            }
    }

    double delta_hs;
    double delta_chi;

    void return_mayer (t_matrix mayer, t_matrix r_, t_matrix Vr, double density, dou
    ble temperature, double b_field) {
            unsigned int i;

    #if SYSTEM == 1
            double r_s = 1.4e-6, r_b = 2.35e-6;
            double sigma_ss = r_s+r_s, sigma_bs = r_s+r_b, sigma_bb = r_b+r_b;
            for (i = 0, mayer.ptr = mayer.matrix, r_.ptr = r_.matrix; i < mayer.size
    ; i+=4, mayer.ptr+=4, r_.ptr++) {
                    *(mayer.ptr) = (*(r_.ptr) < 1.0*sigma_ss/sigma_bs) ? -1.0 : 0.0;
                    *(mayer.ptr+1) = (*(r_.ptr) < 1.0) ? -1.0 : 0.0;
                    *(mayer.ptr+2) = (*(r_.ptr) < 1.0) ? -1.0 : 0.0;
                    *(mayer.ptr+3) = (*(r_.ptr) < 1.0*sigma_bb/sigma_bs) ? -1.0 : 0.
    0;
            }

            for (i = 0, Vr.ptr = Vr.matrix, r_.ptr = r_.matrix; i < Vr.size; i+=4, V
```

lm_bin.c

```
    r.ptr+=4, r_.ptr++) {
                    *(Vr.ptr) = (*(r_.ptr) < 1.0*sigma_ss/sigma_bs) ? -1.0/0.0 : 0.0
;
                    *(Vr.ptr+1) = (*(r_.ptr) < 1.0) ? -1.0/0.0 : 0.0;
                    *(Vr.ptr+2) = (*(r_.ptr) < 1.0) ? -1.0/0.0 : 0.0;
                    *(Vr.ptr+3) = (*(r_.ptr) < 1.0*sigma_bb/sigma_bs) ? -1.0/0.0 : 0
.0;
            }
    #elif SYSTEM == 2
            double r_s = 1.4e-6, r_b = 2.35e-6;
            double delta = delta_hs;
            r_b = 2.35 * pow (10.0, -6.0);
            r_s = r_b*delta;
            double sigma_ss = r_s+r_s, sigma_bs = r_s+r_b, sigma_bb = r_b+r_b;

            for (i = 0, mayer.ptr = mayer.matrix, r_.ptr = r_.matrix; i < mayer.size
    ; i+=4, mayer.ptr+=4, r_.ptr++) {
                    *(mayer.ptr) = (*(r_.ptr) < 1.0*sigma_ss/sigma_bs) ? -1.0 : 0.0;
                    *(mayer.ptr+1) = (*(r_.ptr) < 1.0) ? -1.0 : 0.0;
                    *(mayer.ptr+2) = (*(r_.ptr) < 1.0) ? -1.0 : 0.0;
                    *(mayer.ptr+3) = (*(r_.ptr) < 1.0*sigma_bb/sigma_bs) ? -1.0 : 0.
    0;
            }
    #elif SYSTEM == 3
            double r_s = 1.4e-6, r_b = 2.35e-6;

            double kb = 1.3806505e-23;
            double mu_ = pow (10.0, -7.0);
            double chi_s = 6.6e-12, chi_b = 6.2e-11;
            double B = b_field;
            double T = temperature;

    #ifdef SYSTEM_3
            r_s = r_b = 1.4e-6;
            chi_s = chi_b = 6.2e-12;
    #endif

            double sigma_ss = r_s+r_s, sigma_bs = r_s+r_b, sigma_bb = r_b+r_b;
            double norm = pow (sigma_bs, -3.0);

            double V_ss = (-mu_*B*B/(kb*T) * chi_s*chi_s * norm);
            double V_sb = (-mu_*B*B/(kb*T) * chi_s*chi_b * norm);
            double V_bs = (-mu_*B*B/(kb*T) * chi_b*chi_s * norm);
            double V_bb = (-mu_*B*B/(kb*T) * chi_b*chi_b * norm);

    #ifdef NEW_NORM
            norm = pow (density/(sigma_bs*sigma_bs), -3.0/2.0);
            V_ss = (-mu_*B*B/(kb*T) * chi_s*chi_s);
            V_sb = (-mu_*B*B/(kb*T) * chi_s*chi_b);
            V_bs = (-mu_*B*B/(kb*T) * chi_b*chi_s);
            V_bb = (-mu_*B*B/(kb*T) * chi_b*chi_b);
    #endif

            out ("\n-> reduziertes Potential: V_ss:%lf V_sb:%lf V_bs:%lf V_bb:%lf", V_ss, V_sb, V_bs,
    V_bb);

            for (i = 0, mayer.ptr = mayer.matrix, r_.ptr = r_.matrix; i < mayer.size
    ; i+=4, mayer.ptr+=4, r_.ptr++) {
                    *(mayer.ptr) = (*(r_.ptr) < 1.0*sigma_ss/sigma_bs) ? -1.0 : exp
    (V_ss*pow(*r_.ptr,-3.0))-1.0;
                    *(mayer.ptr+1) = (*(r_.ptr) < 1.0) ? -1.0 : exp (V_bs*pow(*r_.pt
    r,-3.0))-1.0;
                    *(mayer.ptr+2) = (*(r_.ptr) < 1.0) ? -1.0 : exp (V_sb*pow(*r_.pt
    r,-3.0))-1.0;
                    *(mayer.ptr+3) = (*(r_.ptr) < 1.0*sigma_bb/sigma_bs) ? -1.0 : ex
    p (V_bb*pow(*r_.ptr,-3.0))-1.0;
    #ifdef NEW_NORM
                    *(mayer.ptr) = (*(r_.ptr) < 1.0*sigma_ss/sigma_bs) ? -1.0 : exp
    (V_ss*pow(*r_.ptr/norm,-3.0))-1.0;
```

lm_bin.c

```
                    *(mayer.ptr+1) = (*(r_.ptr) < 1.0) ? -1.0 : exp (V_bs*pow(*r_.pt
    r/norm,-3.0))-1.0;
                    *(mayer.ptr+2) = (*(r_.ptr) < 1.0) ? -1.0 : exp (V_sb*pow(*r_.pt
    r/norm,-3.0))-1.0;
                    *(mayer.ptr+3) = (*(r_.ptr) < 1.0*sigma_bb/sigma_bs) ? -1.0 : ex
    p (V_bb*pow(*r_.ptr/norm,-3.0))-1.0;
    #endif
            }

            for (i = 0, Vr.ptr = Vr.matrix, r_.ptr = r_.matrix; i < Vr.size; i+=4, V
    r.ptr+=4, r_.ptr++) {
                    *(Vr.ptr) = (*(r_.ptr) < 1.0*sigma_ss/sigma_bs) ? -1.0/0.0 : V_s
    s*pow(*r_.ptr,-3.0);
                    *(Vr.ptr+1) = (*(r_.ptr) < 1.0) ? -1.0/0.0 : V_sb*pow(*r_.ptr,-3
    .0);
                    *(Vr.ptr+2) = (*(r_.ptr) < 1.0) ? -1.0/0.0 : V_bs*pow(*r_.ptr,-3
    .0);
                    *(Vr.ptr+3) = (*(r_.ptr) < 1.0*sigma_bb/sigma_bs) ? -1.0/0.0 : V
    _bb*pow(*r_.ptr,-3.0);
    #ifdef NEW_NORM
                    *(Vr.ptr) = (*(r_.ptr) < 1.0*sigma_ss/sigma_bs) ? -1.0/0.0 : V_s
    s*pow(*r_.ptr/norm,-3.0);
                    *(Vr.ptr+1) = (*(r_.ptr) < 1.0) ? -1.0/0.0 : V_sb*pow(*r_.ptr/no
    rm,-3.0);
                    *(Vr.ptr+2) = (*(r_.ptr) < 1.0) ? -1.0/0.0 : V_bs*pow(*r_.ptr/no
    rm,-3.0);
                    *(Vr.ptr+3) = (*(r_.ptr) < 1.0*sigma_bb/sigma_bs) ? -1.0/0.0 : V
    _bb*pow(*r_.ptr/norm,-3.0);
    #endif
            }
    #elif SYSTEM == 4
            double r_s = 1.4e-6, r_b = 2.35e-6;
            double sigma_ss = r_s+r_s, sigma_bs = r_s+r_b, sigma_bb = r_b+r_b;
            double kb = 1.3806505e-23;
            double mu_ = 10e-7;
            double T = temperature;

            double norm = 1.0;

            double V_ss = (-mu_/(kb*T) * norm);
            double V_sb = (-mu_/(kb*T) * norm);
            double V_bs = (-mu_/(kb*T) * norm);
            double V_bb = (-mu_/(kb*T) * norm);

            out ("\n-> reduziertes Potential: V_ss:%lf V_sb:%lf V_bs:%lf V_bb:%lf", V_ss, V_sb, V_bs,
    V_bb);

            for (i = 0, mayer.ptr = mayer.matrix, r_.ptr = r_.matrix; i < mayer.size
    ; i+=4, mayer.ptr+=4, r_.ptr++) {
                    *(mayer.ptr) = (*(r_.ptr) < 1.0*sigma_ss/sigma_bs) ? -1.0 : exp
    (V_ss*(pow(*r_.ptr*sigma_bs,-6.0)-pow(*r_.ptr*sigma_bs,-12.0)))-1.0;
                    *(mayer.ptr+1) = (*(r_.ptr) < 1.0) ? -1.0 : exp (V_bs*(pow(*r_.p
    tr*sigma_bs,-3.0)-pow(*r_.ptr*sigma_bs,-12.0)))-1.0;
                    *(mayer.ptr+2) = (*(r_.ptr) < 1.0) ? -1.0 : exp (V_sb*(pow(*r_.p
    tr*sigma_bs,-3.0)-pow(*r_.ptr*sigma_bs,-12.0)))-1.0;
                    *(mayer.ptr+3) = (*(r_.ptr) < 1.0*sigma_bb/sigma_bs) ? -1.0 : ex
    p (V_bb*(pow(*r_.ptr*sigma_bs,-3.0)-pow(*r_.ptr*sigma_bs,-12.0)))-1.0;
            }
            for (i = 0, Vr.ptr = Vr.matrix, r_.ptr = r_.matrix; i < Vr.size; i+=4, V
    r.ptr+=4, r_.ptr++) {
                    *(Vr.ptr) = (*(r_.ptr) < 1.0*sigma_ss/sigma_bs) ? -1.0/0.0 : V_s
    s*(pow(*r_.ptr*sigma_bs,-3.0)-pow(*r_.ptr*sigma_bs,-12.0));
                    *(Vr.ptr+1) = (*(r_.ptr) < 1.0) ? -1.0/0.0 : V_sb*(pow(*r_.ptr*s
    igma_bs,-3.0)-pow(*r_.ptr*sigma_bs,-12.0));
                    *(Vr.ptr+2) = (*(r_.ptr) < 1.0) ? -1.0/0.0 : V_bs*(pow(*r_.ptr*s
    igma_bs,-3.0)-pow(*r_.ptr*sigma_bs,-12.0));
                    *(Vr.ptr+3) = (*(r_.ptr) < 1.0*sigma_bb/sigma_bs) ? -1.0/0.0 : V
    _bb*(pow(*r_.ptr*sigma_bs,-3.0)-pow(*r_.ptr*sigma_bs,-12.0));
            }
```

lm_bin.c

```
    #elif SYSTEM == 5
            double V_ss = -50.0;
            double V_sb = -50.0;
            double V_bs = -50.0;
            double V_bb = -50.0;

            #define POTENTIAL ( exp (0.3/ *r_.ptr)-1.0 )
            out ("\n-> reduziertes Potential: V_ss:%lf V_sb:%lf V_bs:%lf V_bb:%lf", V_ss, V_sb, V_bs,
    V_bb);

            for (i = 0, mayer.ptr = mayer.matrix, r_.ptr = r_.matrix; i < mayer.size
    ; i+=4, mayer.ptr+=4, r_.ptr++) {
                    *(mayer.ptr) = (*(r_.ptr) < 1.5) ? -1.0 : exp (V_ss*POTENTIAL)-1
    .0;
                    *(mayer.ptr+1) = (*(r_.ptr) < 1.0) ? -1.0 : exp (V_bs*POTENTIAL)
    -1.0;
                    *(mayer.ptr+2) = (*(r_.ptr) < 1.0) ? -1.0 : exp (V_sb*POTENTIAL)
    -1.0;
                    *(mayer.ptr+3) = (*(r_.ptr) < 0.5) ? -1.0 : exp (V_bb*POTENTIAL)
    -1.0;
            }

            for (i = 0, Vr.ptr = Vr.matrix, r_.ptr = r_.matrix; i < Vr.size; i+=4, V
    r.ptr+=4, r_.ptr++) {
                    *(Vr.ptr) = (*(r_.ptr) < 1.5) ? -1.0/0.0 : V_ss*POTENTIAL;
                    *(Vr.ptr+1) = (*(r_.ptr) < 1.0) ? -1.0/0.0 : V_bs*POTENTIAL;
                    *(Vr.ptr+2) = (*(r_.ptr) < 1.0) ? -1.0/0.0 : V_sb*POTENTIAL;
                    *(Vr.ptr+3) = (*(r_.ptr) < 0.5) ? -1.0/0.0 : V_bb*POTENTIAL;
            }
    #undef POTENTIAL
    #elif SYSTEM == 6
            double r_s = 1.4e-6, r_b = 2.35e-6;

            double kb = 1.3806505e-23;
            double mu_ = pow (10.0, -7.0);
            double chi_s = 6.6e-12, chi_b = 6.2e-11;
            double B = b_field;
            double T = temperature;

    #ifdef SYSTEM_3
            r_s = r_b = 1.4e-6;
            chi_s = chi_b = 6.2e-12;
    #endif

            r_s = r_b*delta_hs;
            chi_s = chi_b*delta_chi;

            out (" MAYER: rs:%g rb:%g cs:%g cb:%g",r_s, r_b, chi_s, chi_b);

            double sigma_ss = r_s+r_s, sigma_bs = r_s+r_b, sigma_bb = r_b+r_b;
            double norm = pow (sigma_bs, -3.0);

            double V_ss = (-mu_*B*B/(kb*T) * chi_s*chi_s * norm);
            double V_sb = (-mu_*B*B/(kb*T) * chi_s*chi_b * norm);
            double V_bs = (-mu_*B*B/(kb*T) * chi_b*chi_s * norm);
            double V_bb = (-mu_*B*B/(kb*T) * chi_b*chi_b * norm);

    #ifdef NEW_NORM
            norm = pow (density/(sigma_bs*sigma_bs), -3.0/2.0);
            V_ss = (-mu_*B*B/(kb*T) * chi_s*chi_s);
            V_sb = (-mu_*B*B/(kb*T) * chi_s*chi_b);
            V_bs = (-mu_*B*B/(kb*T) * chi_b*chi_s);
            V_bb = (-mu_*B*B/(kb*T) * chi_b*chi_b);
    #endif

            out ("\n-> reduziertes Potential: V_ss:%lf V_sb:%lf V_bs:%lf V_bb:%lf", V_ss, V_sb, V_bs,
    V_bb);

            for (i = 0, mayer.ptr = mayer.matrix, r_.ptr = r_.matrix; i < mayer.size
```

lm_bin.c

```
    ; i+=4, mayer.ptr+=4, r_.ptr++) {
                    *(mayer.ptr) = (*(r_.ptr) < 1.0*sigma_ss/sigma_bs) ? -1.0 : exp
    (V_ss*pow(*r_.ptr,-3.0))-1.0;
                    *(mayer.ptr+1) = (*(r_.ptr) < 1.0) ? -1.0 : exp (V_bs*pow(*r_.pt
    r,-3.0))-1.0;
                    *(mayer.ptr+2) = (*(r_.ptr) < 1.0) ? -1.0 : exp (V_sb*pow(*r_.pt
    r,-3.0))-1.0;
                    *(mayer.ptr+3) = (*(r_.ptr) < 1.0*sigma_bb/sigma_bs) ? -1.0 : ex
    p (V_bb*pow(*r_.ptr,-3.0))-1.0;
    #ifdef NEW_NORM
                    *(mayer.ptr) = (*(r_.ptr) < 1.0*sigma_ss/sigma_bs) ? -1.0 : exp
    (V_ss*pow(*r_.ptr/norm,-3.0))-1.0;
                    *(mayer.ptr+1) = (*(r_.ptr) < 1.0) ? -1.0 : exp (V_bs*pow(*r_.pt
    r/norm,-3.0))-1.0;
                    *(mayer.ptr+2) = (*(r_.ptr) < 1.0) ? -1.0 : exp (V_sb*pow(*r_.pt
    r/norm,-3.0))-1.0;
                    *(mayer.ptr+3) = (*(r_.ptr) < 1.0*sigma_bb/sigma_bs) ? -1.0 : ex
    p (V_bb*pow(*r_.ptr/norm,-3.0))-1.0;
    #endif
            }

            for (i = 0, Vr.ptr = Vr.matrix, r_.ptr = r_.matrix; i < Vr.size; i+=4, V
    r.ptr+=4, r_.ptr++) {
                    *(Vr.ptr) = (*(r_.ptr) < 1.0*sigma_ss/sigma_bs) ? -1.0/0.0 : V_s
    s*pow(*r_.ptr,-3.0);
                    *(Vr.ptr+1) = (*(r_.ptr) < 1.0) ? -1.0/0.0 : V_sb*pow(*r_.ptr,-3
    .0);
                    *(Vr.ptr+2) = (*(r_.ptr) < 1.0) ? -1.0/0.0 : V_bs*pow(*r_.ptr,-3
    .0);
                    *(Vr.ptr+3) = (*(r_.ptr) < 1.0*sigma_bb/sigma_bs) ? -1.0/0.0 : V
    _bb*pow(*r_.ptr,-3.0);
    #ifdef NEW_NORM
                    *(Vr.ptr) = (*(r_.ptr) < 1.0*sigma_ss/sigma_bs) ? -1.0/0.0 : V_s
    s*pow(*r_.ptr/norm,-3.0);
                    *(Vr.ptr+1) = (*(r_.ptr) < 1.0) ? -1.0/0.0 : V_sb*pow(*r_.ptr/no
    rm,-3.0);
                    *(Vr.ptr+2) = (*(r_.ptr) < 1.0) ? -1.0/0.0 : V_bs*pow(*r_.ptr/no
    rm,-3.0);
                    *(Vr.ptr+3) = (*(r_.ptr) < 1.0*sigma_bb/sigma_bs) ? -1.0/0.0 : V
    _bb*pow(*r_.ptr/norm,-3.0);
    #endif
            }

    #endif
    }


    int main (int argc, char* argv[]) {
            int a, b;
            int i, j;
            int q;
            int m, n;


            double p = 5e-2;;
            unsigned int gitter = 1*1024;
            double Rmax = 30.0;;
            double x_s = 0.3, x_b = 1.0-x_s;
    #if SYSTEM == 1
            out ("\n-> Harte Scheibchen");

    #ifndef BASIS_LINE
            Rmax = 20.0;
            #define NU 20
            int nu = NU;
            double r_i_a[NU] = {.1, .2, .3, .4, .5, .65, .95, 1.2, 1.3, 1.5, 1.7, 2.
    0, 2.3, 2.5, 2.7, 3.0, 3.5, 4.0, 5.0, 6.0};
```

lm_bin.c

```
        if (argc != 1+4)
                die ("\n<cmd> <dichte> <x_s> <rmax> <m>\n");
#else
        if ((int) argc <= 1+4)
                die ("\n<cmd> <dichte> <x_s> <rmax> <m> [basen...]\n");

#endif
        p = (double) atof (argv[1]);
        x_s = (double) atof (argv[2]);
        x_b = 1.0-x_s;
        Rmax = (double)(atof(argv[3]));
        gitter = (unsigned int)(atoi(argv[4]));

        double delta = 1.4/2.35;

        double temperature = 0.0;
        double b_field = 0.0;

#ifdef BASIS_LINE
        int NU = (int)argc - (1+4);
        int nu = NU;
        double *r_i_a = (double *) calloc ((size_t)nu,(size_t)(sizeof(double)));
        for (i = 0; i < NU; i++)
                r_i_a[i] = (double)(atof(argv[1+4+i]));
#endif

        out ("\n-> Parameter: p:%lf x_s:%lf x_b:%lf Rmax:%lf Pkte:%d", p, x_s, x_b, Rmax, gitt
er);
#elif SYSTEM == 2
        out ("\n-> Mix harter Scheibchen");

#ifndef BASIS_LINE
        Rmax = 30.0;
        #define NU 20
        int nu = NU;
        double r_i_a[NU] = {.1, .2, .3, .4, .5, .65, .95, 1.2, 1.3, 1.5, 1.7, 2.
0, 2.3, 2.5, 2.7, 3.0, 3.5, 4.0, 5.0, 6.0};

        if (argc != 1+5)
                die ("\n<cmd> <dichte> <x1> <delta> <rmax> <m>\n");
#else
        if (argc <= 1+5)
                die ("\n<cmd> <dichte> <x1> <delta> <rmax> <m> [basen...]\n");

        int NU = (int)argc - (1+5);
        int nu = NU;
        double *r_i_a = (double *) calloc ((size_t)nu,(size_t)(sizeof(double)));
        for (i = 0; i < NU; i++)
                r_i_a[i] = (double)(atof(argv[1+5+i]));
#endif
        p = (double) atof (argv[1]);
        double xi = 0;
        delta_hs =(double) atof (argv[3]);
        double delta = delta_hs;
        x_s = (double)atof(argv[2]);
        x_b = 1.0-x_s;
        gitter = (unsigned int)(atoi(argv[5]));
        Rmax = (double)(atof(argv[4]));

        double temperature = 0.0;
        double b_field = 0.0;

        out ("\n-> Parameter: p:%lf xi=x_s/x_b:%lf delta=r_s/r_b:%lf Rmax:%lf Pkte:%d", p, xi, delt
a, Rmax, gitter);
#elif SYSTEM == 3
        out ("\n-> Dipolare harte Scheibchen");

#ifndef BASIS_LINE
        Rmax = 100.0;
```

lm_bin.c

```
        #define NU 45
        int nu = NU;
        double r_i_a[NU] = {.2, .65, .9, 1.4, 1.7, 2.0, 2.3, 2.7, 3.0, 3.3, 3.9,
 4.2, 4.5, 4.8, 5.1, 5.4,
        5.7, 6.0, 6.3, 6.6, 6.9, 7.2, 7.5,
        7.8, 8.1, 8.4, 8.7, 9.0, 9.5, 10.0, 10.5, 11.0,
        11.5, 12.0, 12.5, 13.0, 13.5, 14.0, 14.5, 15.0, 15.6, 16.0,
        17.0, 18.0,
        20.0};

        gitter = .75*1024;
#endif

#ifndef BASIS_LINE
        if (argc != 1+6)
                die ("\n<cmd> <dichte [s^2]> <x_s> <temperatur [K]> <b [mT]> <rmax> <m>\n");
#else
        if ((int)argc <= 1+6)
                die ("\n<cmd> <dichte [s^2]> <x_s> <temperatur [K]> <b [mT]> <rmax> <m> [basen..]\n"
);
#endif
        p = (double) atof (argv[1]);
        x_s = (double) atof (argv[2]);
        x_b = 1.0-x_s;
        double temperature = (double) atof (argv[3]);
        double b_field = (double) (atof (argv[4])) * pow (10.0, -3.0);
        Rmax = (int)(atoi(argv[5]));
        gitter = (int)(atoi(argv[6]));

#ifdef BASIS_LINE
        int NU = (int)argc - (1+6);
        int nu = NU;
        double *r_i_a = (double *) calloc ((size_t)nu,(size_t)(sizeof(double)));
        for (i = 0; i < NU; i++)
                r_i_a[i] = (double)(atof(argv[1+6+i]));
#endif

        out ("\n-> Parameter: p:%lf x_s:%lf x_b:%lf T:%lf B:%lf Rmax:%lf Pkte:%d", p, x_s, x_b, t
emperature, b_field, Rmax, gitter);
#elif SYSTEM == 4
        out ("\n-> Binäres LJ");

        Rmax = 20.0;
        #define NU 20
        int nu = NU;
        double r_i_a[NU] = {.1, .2, .3, .4, .5, .65, .95, 1.2, 1.3, 1.5, 1.7, 2.
0, 2.3, 2.5, 2.7, 3.0, 3.5, 4.0, 5.0, 6.0};
        gitter = 1*1024;

        if (argc != 1+3)
                die ("\n<cmd> <dichte> <x_s> <temperature>\n");
        p = (double) atof (argv[1]);
        x_s = (double) atof (argv[2]);
        x_b = 1.0-x_s;
        double temperature = (double) atof (argv[3]);

        double b_field = 0.0;

        out ("\n-> Parameter: p:%lf x_s:%lf x_b:%lf Rmax:%lf Pkte:%d", p, x_s, x_b, Rmax, gitt
er);
#elif SYSTEM == 5
        out ("\n-> Exponentieller Abfall");

        Rmax = 50.0;
        #define NU 20
        int nu = NU;
        double r_i_a[NU] = {.1, .2, .3, .4, .5, .65, .95, 1.2, 1.3, 1.5, 1.7, 2.
0, 2.3, 2.5, 2.7, 3.0, 3.5, 4.0, 5.0, 6.0};
        gitter = 1024;
```

lm_bin.c

```
            if (argc != 1+2)
                    die ("\n<cmd> <dichte> <x_s>\n");
1340        p = (double) atof (argv[1]);
            x_s = (double) atof (argv[2]);
            x_b = 1.0-x_s;

            double temperature = 0.0;
1345        double b_field = 0.0;

            out ("\n-> Parameter: p:%lf x_s:%lf x_b:%lf Rmax:%lf Pkte:%d", p, x_s, x_b, Rmax, gitt
    er);
    #elif SYSTEM == 6
            out ("\n-> Dipolare harte Scheibchen (mix)");
1350
    #ifndef BASIS_LINE
            Rmax = 100.0;
            #define NU 45
            int nu = NU;
1355        double r_i_a[NU] = {.2, .65, .9, 1.4, 1.7, 2.0, 2.3, 2.7, 3.0, 3.3, 3.9,
     4.2, 4.5, 4.8, 5.1, 5.4,
            5.7, 6.0, 6.3, 6.6, 6.9, 7.2, 7.5,
            7.8, 8.1, 8.4, 8.7, 9.0, 9.5, 10.0, 10.5, 11.0,
            11.5, 12.0, 12.5, 13.0, 13.5, 14.0, 14.5, 15.0, 15.6, 16.0,
            17.0, 18.0,
1360        20.0};

            gitter = .75*1024;
    #endif

1365 #ifndef BASIS_LINE
            if (argc != 1+6)
                    die ("\n<cmd> <dichte [s^2]> <x_s> <temperatur [K]> <b [mT]> <rmax> <m>\n");
    #else
            if ((int)argc <= 1+8)
1370                die ("\n<cmd> <dichte [s^2]> <x_s> <delta_r> <delta_chi> <temperatur [K]> <b [mT]> <r
    max> <m> [basen..]\n");
    #endif
            p = (double) atof (argv[1]);
            x_s = (double) atof (argv[2]);
            x_b = 1.0-x_s;
1375        delta_hs = (double) atof(argv[3]);
            delta_chi = (double)atof(argv[4]);
            double delta = delta_hs;
            double temperature = (double) atof (argv[5]);
            double b_field = (double) (atof (argv[6])) * pow (10.0, -3.0);
1380        Rmax = (int)(atoi(argv[7]));
            gitter = (int)(atoi(argv[8]));

    #ifdef BASIS_LINE
            int NU = (int)argc - (1+8);
1385        int nu = NU;
            double *r_i_a = (double *) calloc ((size_t)nu,(size_t)(sizeof(double)));
            for (i = 0; i < NU; i++)
                    r_i_a[i] = (double)(atof(argv[1+8+i]));
    #endif
1390
            out ("\n-> Parameter: p:%lf x_s:%lf x_b:%lf T:%lf B:%lf Rmax:%lf Pkte:%d delta_r:%lf delta_chi:%lf
    ", p, x_s, x_b, temperature, b_field, Rmax, gitter, delta_hs, delta_chi);

    #endif


1395
    #ifdef PACKING_FRACTION
            out ("\n-> Packingfraction: %lf", p);
            double p_new = (p/(double)(M_PIl))*pow((delta+1.0),2.0)/(x_s*delta*delta
    +x_b);
            out (" -> ns_12^2=%lf", p_new);
1400        p = p_new;
```

lm_bin.c

```
    #endif


    #if APPROXIMATION == 1
1405        out ("\n-> PY");
    #elif APPROXIMATION == 2
            out ("\n-> HNC");
    #endif

1410 #ifdef SYM
            printf ("\n-> Symmetriesiere Gamma(q) nach OZ"); fflush (stdout);
    #if SYM == 1
            printf (" - alle 4 Matrixelemente gleich machen"); fflush (stdout);
    #endif
1415 #endif
    #ifdef ASYMPTOTE
            printf ("\n-> Asymptote Gamma(r)=-1-C(r) r<sigma"); fflush (stdout);
    #endif

1420        out ("\n-> %d Basen:", nu);
            for (i = 0; i < nu; i++)
                    out (" %.2lf", r_i_a[i]);


1425        printf ("\n-> Matrizen initialisieren");
            t_matrix Cr = init_matrix (gitter*2*2);
            t_matrix Cq = init_matrix (gitter*2*2);
            t_matrix Gammar = init_matrix (gitter*2*2);
            t_matrix Gammaq = init_matrix (gitter*2*2);
1430 #ifdef SYM
            t_matrix sym_Gammaq = init_matrix (gitter*2*2);
    #endif
            t_matrix Vr = init_matrix (gitter*2*2);

1435

            printf ("\n-> Besselfunktionen"); fflush (stdout);
            printf (" Fkt."); fflush (stdout);
            double Qmax = (double)(gsl_sf_bessel_zero_J0 (gitter+1)/Rmax);
            t_matrix r_ = init_matrix (gitter);
1440        for (i = 0, r_.ptr = r_.matrix; i < r_.size; i++, r_.ptr++)
                    *r_.ptr = (double)(gsl_sf_bessel_zero_J0 (i+1)/Qmax);
            t_matrix q_ = init_matrix (gitter);
            for (m = 0, q_.ptr = q_.matrix; m < q_.size; m++, q_.ptr++)
                    *q_.ptr = (double)(gsl_sf_bessel_zero_J0 (m+1)/Rmax);
1445
            printf (" Fbt-Konstanten"); fflush (stdout);
            printf (" 1"); fflush (stdout);
            t_fbt fbt1 = {
                    .cons = 4.0*(double)(M_PIl)/(Qmax*Qmax),
1450                .gitter = gitter,
                    .bessel = init_matrix (gitter*gitter),
                    .mr = Cr,
                    .mq = Cq
            };
1455        for (m = 0, fbt1.bessel.ptr = fbt1.bessel.matrix, q_.ptr = q_.matrix; m
    < q_.size; m++, q_.ptr++)
                    for (i = 0, r_.ptr = r_.matrix; i < r_.size; i++, fbt1.bessel.pt
    r++, r_.ptr++)
                            *fbt1.bessel.ptr = j0(*q_.ptr**r_.ptr) / ( j1(*r_.ptr*Qm
    ax)*j1(*r_.ptr*Qmax) );

            printf (" 2"); fflush (stdout);
1460        t_fbt fbt2 = {
                    .cons = 1.0/((double)(M_PIl)*Rmax*Rmax),
                    .gitter = gitter,
                    .bessel = init_matrix (gitter*gitter),
                    .mq = Gammaq,
1465                .mr = Gammar
            };
```

lm_bin.c

```
        for (i = 0, fbt2.bessel.ptr = fbt2.bessel.matrix, r_.ptr = r_.matrix; i
< r_.size; i++, r_.ptr++)
                for (m = 0, q_.ptr = q_.matrix; m < q_.size; m++, fbt2.bessel.pt
r++, q_.ptr++)
                        *fbt2.bessel.ptr = j0(*q_.ptr**r_.ptr) / ( j1(*q_.ptr*Rm
ax)*j1(*q_.ptr*Rmax) );

        printf ("\n-> Mayerfunktion"); fflush (stdout);
        t_matrix mayer = init_matrix (gitter*2*2);
        return_mayer (mayer, r_, Vr, p, temperature, b_field);

#if STARTVALUE == 1
        Gammar.ptr = Gammar.matrix;
        r_.ptr = r_.matrix;
        for (i = 0; i < Gammar.size; i+=4, Gammar.ptr+=4, r_.ptr++)
                *(Gammar.ptr) = *(Gammar.ptr+1) = *(Gammar.ptr+2) = *(Gammar.ptr
+3) = 20.0*exp(-powl(*r_.ptr, 2.0)/4.0);
        printf (" Gamma(r) = 20*exp(-x^2/4)");
#elif STARTVALUE == 2
        for (i = 0, Gammar.ptr = Gammar.matrix; i < Gammar.size; i++, Gammar.ptr
++)
                *Gammar.ptr = 0.0;
        printf (" Gamma(r) = 0");
#elif STARTVALUE == 3
        printf (" lese h(r) aus 'startvalue'"); fflush (stdout);
        t_read_matrix returnvalue;
        returnvalue = read_matrix (Gamma_q, gitter, "startvalue");
        printf (" %d Zeilen gelesen", returnvalue.rows);
        printf (" - konvertiere zu Gamma(r)"); fflush (stdout);
        for (i = 0; i < gitter; i++) for (a = 0; a < 2; a++) for (b = 0; b < 2;
b++)
                Gamma_r[i][a][b] = Gamma_q[i][a][b];
#elif STARTVALUE == 4
        for (i = 0, Gammar.ptr = Gammar.matrix; i < Gammar.size; i++, Gammar.ptr
++)
                *Gammar.ptr = 0.1;
#elif STARTVALUE == 5
        out (" Gamma(r) einlesen: ");
        read_matrix (Gammaq, "start_gamma");
        for (i = 0, Gammaq.ptr = Gammaq.matrix, Gammar.ptr = Gammar.matrix; i <
Gammar.size; i++, Gammaq.ptr++, Gammar.ptr++)
                *Gammar.ptr = *Gammaq.ptr;
#endif

        out ("\n-> Zerlegungsbasis");
        t_matrix P_ = init_matrix (gitter*nu),
                 Q_ = init_matrix (gitter*nu);
        pq_init (r_i_a, nu, r_, P_, Q_);
#ifdef __BASIS_OUT__
        t_basis_out basis = {
                .P_ = P_,
                .gitter = gitter,
                .nu = nu,
                .r_ = r_
        };
        basis_out (basis);
#endif
#ifdef __TEST_PERP__
        double sum;
        for (m = 0; m < nu; m++) {
                out ("\nm:%d  ", m);
                for (n = 0; n < nu; n++) {
                        sum = 0.0;
                        for (i = 0; i < gitter; i++) {
                                sum += P_.matrix[i*nu+n]*Q_.matrix[i*nu+m];
                        }
```

lm_bin.c

```
                        out (" %lf", sum);
                }
        }
#endif

        printf (" Strukturen"); fflush (stdout);
        t_composite comp = {
                .function = Gammar,
                .coarse = init_matrix (nu*2*2),
                .fine = init_matrix (gitter*2*2),
                .P_ = P_,
                .Q_ = Q_,
                .gitter = gitter,
                .nu = nu
        };

        t_composite decomp = {
                .function = Gammar,
                .coarse = init_matrix (nu*2*2),
                .fine = init_matrix (gitter*2*2),
                .P_ = P_,
                .Q_ = Q_,
                .gitter = gitter,
                .nu = nu
        };

#ifdef MARQUARDT
        out ("\n-> Marquardt initialisieren");

        lm_control_type control;
        lm_initialize_control (&control);

        lm_data_type data = {
                .Cr = Cr, .Cq = Cq, .Gammar = Gammar, .Gammaq = Gammaq,
                .p_s = x_s*p, .p_b = x_b*p,
                .mayer = mayer, .Vr = Vr,
                .fbt1 = fbt1, .fbt2 = fbt2,
                .comp = {
                        .function = Gammar,
                        .coarse = init_matrix (nu*2*2),
                        .fine = init_matrix (gitter*2*2),
                        .P_ = P_,
                        .Q_ = Q_,
                        .gitter = gitter,
                        .nu = nu
                },
                .decomp = {
                        .function = Gammar,
                        .coarse = init_matrix (nu*2*2),
                        .fine = init_matrix (gitter*2*2),
                        .P_ = P_,
                        .Q_ = Q_,
                        .gitter = gitter,
                        .nu = nu
                }
        };
#endif

        out ("\n-> Hauptschleife");
        unsigned int pic = 0;
        unsigned int pic_max = 10000;
        double norm_fine = 0.0;
        double norm_fine_max = 1e-12;

        time_t start, now;
        time (&start);
```

lm_bin.c

```
        out (" Ausgabe");
        printf (" 1"); fflush (stdout);
        t_matrices matrices = {
                .Gammar = Gammar,
                .Gammaq = Gammaq,
                .Cr = Cr,
                .Cq = Cq,
                .r_ = r_,
                .q_ = q_,
                .pic = &pic,
                .x_s = &x_s,
                .x_b = &x_b,
                .p = &p
        };

        printf (" 2"); fflush (stdout);
#ifdef _STARTVALUE_
        matrix_out (matrices);
#endif

#ifdef _COARSE_FINE_
        printf (" 3"); fflush (stdout);
        t_coarse_fine coarse_fine = {
                .coarse = comp.coarse,
                .fine = comp.fine,
                .gitter = gitter,
                .nu = nu,
                .nr = &nr,
                .pic = &pic,
                .r_ = r_,
                .P_ = P_
        };
#endif



        double out_period_norm = 1e-1;
        unsigned int out_period = 1;

#ifdef DYN_COARSE
        out ("\n--> Grobanteilgenauigkeit adaptiv anpassen");
        double dyn_coarse_step = 1e-1,
                dyn_coarse = 1e-3,
                dyn_coarse_max = 1e-12;
        double norm_fine_max_dyn_coarse = norm_fine_max*10;
        control.epsilon = dyn_coarse;
        control.ftol = dyn_coarse;
        control.xtol = dyn_coarse;
        control.gtol = dyn_coarse;
        norm_fine_max = dyn_coarse_max;
#else
        control.epsilon = control.ftol = control.xtol = control.gtol = 1e-15;
#endif

#ifdef FINE_RESCALE
        double alpha = 1.0;
#endif

        printf ("\n--> Erste Zerlegung:"); fflush (stdout);
        decomposite (decomp);
        for (i = 0, decomp.fine.ptr = decomp.fine.matrix, comp.fine.ptr = comp.f
ine.matrix; i < decomp.fine.size; i++, decomp.fine.ptr++, comp.fine.ptr++)
                *comp.fine.ptr = *decomp.fine.ptr;
        for (i = 0, decomp.coarse.ptr = decomp.coarse.matrix, comp.coarse.ptr =
 comp.coarse.matrix; i < decomp.coarse.size; i++, decomp.coarse.ptr++, comp.coars
e.ptr++)
                *comp.coarse.ptr = *decomp.coarse.ptr;
```

lm_bin.c

```
#ifdef T_PATH
        double t_step = 0.001;
        double t_max = 10000.0, t_min = temperature;
        temperature = t_max;
        do {
        out ("\n--> *** T=%lf ***", temperature);
        return_mayer (mayer, r_, Vr, p, temperature, b_field);
#endif
        double norm_fine_old = norm_fine;

        do {
                pic++;
                out ("\n%d ", pic);

                for (i = 0, data.comp.fine.ptr = data.comp.fine.matrix, comp.fin
e.ptr = comp.fine.matrix; i < comp.fine.size; i++, data.comp.fine.ptr++, comp.fi
ne.ptr++)
                        *data.comp.fine.ptr = *comp.fine.ptr;
                lm_minimize (4*nu, 4*nu, comp.coarse.matrix, lm_evaluate_default
, lm_print_default, &data, &control);

                norm_fine_old = norm_fine;
                norm_fine = 0.0;
                for (i = 0, comp.fine.ptr = comp.fine.matrix, data.decomp.fine.p
tr = data.decomp.fine.matrix; i < comp.fine.size; i++, comp.fine.ptr++, data.dec
omp.fine.ptr++) {
                        norm_fine += pow ((*comp.fine.ptr-*data.decomp.fine.ptr)
, 2.0);
                        *comp.fine.ptr = *data.decomp.fine.ptr;
                }

                if (norm_fine != 0)
                        norm_fine = sqrt (norm_fine);
#ifdef DEBUG
                matrix_out (matrices);
#endif

                if (norm_fine < out_period_norm) {
                        matrix_out (matrices);
                        out_period_norm = .1*norm_fine;
                        out_period *= 10;
                }
                if ((pic % out_period) == 0)
                        matrix_out (matrices);

                out (" fine: %g", norm_fine);

#ifdef DYN_COARSE
                if (norm_fine < dyn_coarse && dyn_coarse > dyn_coarse_max) {
                        dyn_coarse *= dyn_coarse_step;
                        control.epsilon = dyn_coarse;
                        control.ftol = dyn_coarse;
                        control.xtol = dyn_coarse;
                        control.gtol = dyn_coarse;
                        out ("\n--> Grobanteilgenauigkeit anpassen %g\n", dyn_coarse);
                }
#endif
                time (&now);
                out (" time: %.0lf", difftime (now, start));
#ifdef DYN_COARSE
                } while (norm_fine > norm_fine_max && pic < pic_max);
#else
                } while (norm_fine > norm_fine_max && pic < pic_max);
#endif


#ifdef T_PATH
        matrix_out (matrices);
        temperature -= t_step;
```

lm_bin.c

```
        } while (temperature > t_min);
#endif


        a = b = j = q = n = 1;
#ifdef DYN_COARSE
        norm_fine_max_dyn_coarse = 0.0;
#endif

        goto clean;
clean:
        out ("\n--> Aufräumen");
        matrix_out (matrices);

        out (" Korrelatoren");
        cleanup_matrix (Cr); out (".");
        cleanup_matrix (Cq); out (".");
#ifdef SYM
        cleanup_matrix (sym_Gammaq);
#endif
        cleanup_matrix (mayer); out (".");
        cleanup_matrix (Gammar); out (".");
        cleanup_matrix (Gammaq); out (".");

        cleanup_matrix (Vr);

        out (" Bessel");
        cleanup_matrix (r_);
        cleanup_matrix (q_);
        cleanup_matrix (fbt1.bessel);
        cleanup_matrix (fbt2.bessel);

        out (" Basis");
        cleanup_matrix (P_);
        cleanup_matrix (Q_);

        out (" Grob/Fein");
        cleanup_matrix (comp.fine);
        cleanup_matrix (comp.coarse);
        cleanup_matrix (decomp.fine);
        cleanup_matrix (decomp.coarse);

        out (" Marquardt");
        cleanup_matrix (data.comp.fine);
        cleanup_matrix (data.comp.coarse);
        cleanup_matrix (data.decomp.fine);
        cleanup_matrix (data.decomp.coarse);

#ifdef BASIS_LINE
        free (r_i_a);
#endif


        printf ("\n--> Fertig\n");
        return 0;
}
```

B.3. Levenberg-Marquardt Routine

marquardt.c

```
/*
 * Slightly modified marquardt main-routines
 * original source at: www.sourceforge.net/projects/lmfit
 */
// parameters for calling the high-level interface lmfit
//   ( lmfit.c provides lm_initialize_control which sets default values ):
typedef struct {
        double ftol;       // relative error desired in the sum of squares.
        double xtol;       // relative error between last two approximations.
        double gtol;       // orthogonality desired between fvec and its derivs.
        double epsilon;    // step used to calculate the jacobian.
        double stepbound;  // initial bound to steps in the outer loop.
        double fnorm;      // norm of the residue vector fvec.
        int maxcall;       // maximum number of iterations.
        int nfev;          // actual number of iterations.
        int info;          // status of minimization.
} lm_control_type;

// the subroutine that calculates fvec:
typedef void (lm_evaluate_ftype) (
    double* par, int m_dat, double* fvec, void *data, int *info );
// default implementation therof, provided by lm_eval.c:
void lm_evaluate_default (
    double* par, int m_dat, double* fvec, void *data, int *info );

// the subroutine that informs about fit progress:
typedef void (lm_print_ftype) (
    int n_par, double* par, int m_dat, double* fvec, void *data,
    int iflag, int iter, int nfev );
// default implementation therof, provided by lm_eval.c:
void lm_print_default (
    int n_par, double* par, int m_dat, double* fvec, void *data,
    int iflag, int iter, int nfev );

// compact high-level interface:
void lm_initialize_control( lm_control_type *control );
void lm_minimize ( int m_dat, int n_par, double* par,
                   lm_evaluate_ftype *evaluate, lm_print_ftype *printout,
                   void *data, lm_control_type *control );
double lm_enorm( int, double* );

// low-level interface for full control:
void lm_lmdif( int m, int n, double* x, double* fvec, double ftol, double xtol,
               double gtol, int maxfev, double epsfcn, double* diag, int mode,
               double factor, int *info, int *nfev,
               double* fjac, int* ipvt, double* qtf,
               double* wa1, double* wa2, double* wa3, double* wa4,
               lm_evaluate_ftype *evaluate, lm_print_ftype *printout,
               void *data );



#ifndef _LMDIF
extern char *lm_infmsg[];
extern char *lm_shortmsg[];
#endif


void lm_print_default( int n_par, double* par, int m_dat, double* fvec,
                       void *data, int iflag, int iter, int nfev )
/*
 *       data  : for soft control of printout behaviour, add control
 *               variables to the data struct
 *       iflag : 0 (init) 1 (outer loop) 2(inner loop) -1(terminated)
 *       iter  : outer loop counter
 *       nfev  : number of calls to *evaluate
 */
{
```

marquardt.c

```
#ifdef __OLD_MARQUARDT__
    double f, y, t;
    int i;
    lm_data_type *mydata;
    mydata = (lm_data_type*)data;
#endif
#define QUIET
#ifdef QUIET
#ifdef DEBUG
    out (".");
#endif
#else
    if (iflag==2) {
        printf ("trying step in gradient direction\n");
    } else if (iflag==1) {
        printf ("determining gradient (iteration %d)\n", iter);
    } else if (iflag==0) {
        printf ("starting minimization\n");
    } else if (iflag==-1) {
        printf ("terminated after %d evaluations\n", nfev);
    }

    printf( " par: " );
    for( i=0; i<n_par; ++i )
        printf( " %12g", par[i] );
    printf ( " => norm: %12g\n", lm_enorm( m_dat, fvec ) );
#endif
#ifdef __OLD_MARQUARDT__
    if ( iflag == -1 ) {
        printf( " fitting data as follows:\n" );
        for( i=0; i<m_dat; ++i ) {
            t = (mydata->user_t)[i];
            y = (mydata->user_y)[i];
            f = mydata->user_func( t, par );
            printf( "  t[%2d]=%12g y=%12g fit=%12g residue=%12g\n",
                    i, t, y, f, y-f );
        }
    }
#endif
}

/*
 * lmfit
 *
 * Solves or minimizes the sum of squares of m nonlinear
 * functions of n variables.
 *
 * From public domain Fortran version
 * of Argonne National Laboratories MINPACK
 *     argonne national laboratory. minpack project. march 1980.
 *     burton s. garbow, kenneth e. hillstrom, jorge j. more
 * C translation by Steve Moshier
 * Joachim Wuttke converted the source into C++ compatible ANSI style
 * and provided a simplified interface
 */

#include <stdlib.h>
#include <math.h>
#define _LMDIF

/* ********************** high-level interface *************************** */


void lm_initialize_control( lm_control_type *control )
{
    control->maxcall = 100;
    control->epsilon = 1.e-10;//14;
    control->stepbound = 100.;
```

marquardt.c

```
    control->ftol = 1.e-14;
    control->xtol = 1.e-14;
    control->gtol = 1.e-14;
}

void lm_minimize( int m_dat, int n_par, double* par,
                  lm_evaluate_ftype *evaluate, lm_print_ftype *printout,
                  void *data, lm_control_type *control )
{

// *** allocate work space.

    double *fvec, *diag, *fjac, *qtf, *wa1, *wa2, *wa3, *wa4;
    int *ipvt;

    int n = n_par;
    int m = m_dat;

    if (!(fvec = (double*) malloc(  m*sizeof(double))) ||
        !(diag = (double*) malloc(n*  sizeof(double))) ||
        !(qtf =  (double*) malloc(n*  sizeof(double))) ||
        !(fjac = (double*) malloc(n*m*sizeof(double))) ||
        !(wa1 =  (double*) malloc(n*  sizeof(double))) ||
        !(wa2 =  (double*) malloc(n*  sizeof(double))) ||
        !(wa3 =  (double*) malloc(n*  sizeof(double))) ||
        !(wa4 =  (double*) malloc(  m*sizeof(double))) ||
        !(ipvt = (int*)    malloc(n*  sizeof(int)))) {
            control->info = 9;
            return;
    }

// *** perform fit.

    control->info = 0;
    control->nfev = 0;

    // this goes through the modified legacy interface:
    lm_lmdif( m, n, par, fvec, control->ftol, control->xtol, control->gtol,
              control->maxcall*(n+1), control->epsilon, diag, 1,
              control->stepbound, &(control->info),
              &(control->nfev), fjac, ipvt, qtf, wa1, wa2, wa3, wa4,
              evaluate, printout, data );

    (*printout)( n, par, m, fvec, data, -1, 0, control->nfev );
    control->fnorm = lm_enorm(m, fvec);
    if (control->info < 0 ) control->info = 10;

// *** clean up.

    free(fvec);
    free(diag);
    free(qtf);
    free(fjac);
    free(wa1);
    free(wa2);
    free(wa3 );
    free(wa4);
    free(ipvt);
}


// ***** the following messages are referenced by the variable info.

char *lm_infmsg[] = {
    "improper input parameters",
    "the relative error in the sum of squares is at most tol",
    "the relative error between x and the solution is at most tol",
    "both errors are at most tol",
    "fvec is orthogonal to the columns of the jacobian to machine precision",
```

marquardt.c

```
    "number of calls to fcn has reached or exceeded 200*(n+1)",
    "ftol is too small. no further reduction in the sum of squares is possible",
    "xtol too small. no further improvement in approximate solution x possible",
    "gtol too small. no further improvement in approximate solution x possible",
    "not enough memory",
    "break requested within function evaluation"
};

char *lm_shortmsg[] = {
        "invalid input",
        "success (f)",
        "success (p)",
        "success (f,p)",
        "degenerate",
        "call limit",
        "failed (f)",
        "failed (p)",
        "failed (o)",
        "no memory",
        "user break"
};


/* ************************* implementation ***************************** */


#define BUG 0
#if BUG
#include <stdio.h>
#endif

// the following values seem good for an x86:
#define LM_MACHEP .555e-16 /* resolution of arithmetic */
#define LM_DWARF  9.9e-324 /* smallest nonzero number */
// the follwoing values should work on any machine:
// #define LM_MACHEP 1.2e-16
// #define LM_DWARF 1.0e-38

// the squares of the following constants shall not under/overflow:
// these values seem good for an x86:
#define LM_SQRT_DWARF 1.e-160
#define LM_SQRT_GIANT 1.e150
// the following values should work on any machine:
// #define LM_SQRT_DWARF 3.834e-20
// #define LM_SQRT_GIANT 1.304e19


void lm_qrfac( int m, int n, double* a, int pivot, int* ipvt,
               double* rdiag, double* acnorm, double* wa);
void lm_qrsolv(int n, double* r, int ldr, int* ipvt, double* diag,
               double* qtb, double* x, double* sdiag, double* wa);
void lm_lmpar( int n, double* r, int ldr, int* ipvt, double* diag, double* qtb,
               double delta, double* par, double* x, double* sdiag,
               double* wa1, double* wa2);

#define MIN(a,b) (((a)<=(b)) ? (a) : (b))
#define MAX(a,b) (((a)>=(b)) ? (a) : (b))
#define SQR(x)   (x)*(x)


// ***** the low-level legacy interface for full control.

void lm_lmdif( int m, int n, double* x, double* fvec, double ftol, double xtol,
               double gtol, int maxfev, double epsfcn, double* diag, int mode,
               double factor, int *info, int *nfev,
               double* fjac, int* ipvt, double* qtf,
               double* wa1, double* wa2, double* wa3, double* wa4,
               lm_evaluate_ftype *evaluate, lm_print_ftype *printout,
               void *data )
```

marquardt.c

```
    {
    /*
     *   the purpose of lmdif is to minimize the sum of the squares of
     *   m nonlinear functions in n variables by a modification of
     *   the levenberg-marquardt algorithm. the user must provide a
     *   subroutine evaluate which calculates the functions. the jacobian
     *   is then calculated by a forward-difference approximation.
     *
     *   the multi-parameter interface lm_lmdif is for users who want
     *   full control and flexibility. most users will be better off using
     *   the simpler interface lmfit provided above.
     *
     *   the parameters are the same as in the legacy FORTRAN implementation,
     *   with the following exceptions:
     *      the old parameter ldfjac which gave leading dimension of fjac has
     *        been deleted because this C translation makes no use of two-
     *        dimensional arrays;
     *      the old parameter nprint has been deleted; printout is now controlled
     *        by the user-supplied routine *printout;
     *      the parameter field *data and the function parameters *evaluate and
     *        *printout have been added; they help avoiding global variables.
     *
     *   parameters:
     *
     *      m is a positive integer input variable set to the number
     *        of functions.
     *
     *      n is a positive integer input variable set to the number
     *        of variables. n must not exceed m.
     *
     *      x is an array of length n. on input x must contain
     *        an initial estimate of the solution vector. on output x
     *        contains the final estimate of the solution vector.
     *
     *      fvec is an output array of length m which contains
     *        the functions evaluated at the output x.
     *
     *      ftol is a nonnegative input variable. termination
     *        occurs when both the actual and predicted relative
     *        reductions in the sum of squares are at most ftol.
     *        therefore, ftol measures the relative error desired
     *        in the sum of squares.
     *
     *      xtol is a nonnegative input variable. termination
     *        occurs when the relative error between two consecutive
     *        iterates is at most xtol. therefore, xtol measures the
     *        relative error desired in the approximate solution.
     *
     *      gtol is a nonnegative input variable. termination
     *        occurs when the cosine of the angle between fvec and
     *        any column of the jacobian is at most gtol in absolute
     *        value. therefore, gtol measures the orthogonality
     *        desired between the function vector and the columns
     *        of the jacobian.
     *
     *      maxfev is a positive integer input variable. termination
     *        occurs when the number of calls to lm_fcn is at least
     *        maxfev by the end of an iteration.
     *
     *      epsfcn is an input variable used in determining a suitable
     *        step length for the forward-difference approximation. this
     *        approximation assumes that the relative errors in the
     *        functions are of the order of epsfcn. if epsfcn is less
     *        than the machine precision, it is assumed that the relative
     *        errors in the functions are of the order of the machine
     *        precision.
     *
     *      diag is an array of length n. if mode = 1 (see below), diag is
     *        internally set. if mode = 2, diag must contain positive entries
```

marquardt.c

```
     *        that serve as multiplicative scale factors for the variables.
     *
     *      mode is an integer input variable. if mode = 1, the
     *        variables will be scaled internally. if mode = 2,
     *        the scaling is specified by the input diag. other
     *        values of mode are equivalent to mode = 1.
     *
     *      factor is a positive input variable used in determining the
     *        initial step bound. this bound is set to the product of
     *        factor and the euclidean norm of diag*x if nonzero, or else
     *        to factor itself. in most cases factor should lie in the
     *        interval (.1,100.). 100. is a generally recommended value.
     *
     *      info is an integer output variable that indicates the termination
     *        status of lm_lmdif as follows:
     *
     *        info < 0  termination requested by user-supplied routine *evaluate;
     *
     *        info = 0  improper input parameters;
     *
     *        info = 1  both actual and predicted relative reductions
     *                  in the sum of squares are at most ftol;
     *
     *        info = 2  relative error between two consecutive iterates
     *                  is at most xtol;
     *
     *        info = 3  conditions for info = 1 and info = 2 both hold;
     *
     *        info = 4  the cosine of the angle between fvec and any
     *                  column of the jacobian is at most gtol in
     *                  absolute value;
     *
     *        info = 5  number of calls to lm_fcn has reached or
     *                  exceeded maxfev;
     *
     *        info = 6  ftol is too small. no further reduction in
     *                  the sum of squares is possible;
     *
     *        info = 7  xtol is too small. no further improvement in
     *                  the approximate solution x is possible;
     *
     *        info = 8  gtol is too small. fvec is orthogonal to the
     *                  columns of the jacobian to machine precision;
     *
     *      nfev is an output variable set to the number of calls to the
     *        user-supplied routine *evaluate.
     *
     *      fjac is an output m by n array. the upper n by n submatrix
     *        of fjac contains an upper triangular matrix r with
     *        diagonal elements of nonincreasing magnitude such that
     *
     *               t     t           t
     *              p *(jac *jac)*p = r *r,
     *
     *        where p is a permutation matrix and jac is the final
     *        calculated jacobian. column j of p is column ipvt(j)
     *        (see below) of the identity matrix. the lower trapezoidal
     *        part of fjac contains information generated during
     *        the computation of r.
     *
     *      ipvt is an integer output array of length n. ipvt
     *        defines a permutation matrix p such that jac*p = q*r,
     *        where jac is the final calculated jacobian, q is
     *        orthogonal (not stored), and r is upper triangular
     *        with diagonal elements of nonincreasing magnitude.
     *        column j of p is column ipvt(j) of the identity matrix.
     *
     *      qtf is an output array of length n which contains
     *        the first n elements of the vector (q transpose)*fvec.
```

marquardt.c

```
415  *
     *      wa1, wa2, and wa3 are work arrays of length n.
     *
     *      wa4 is a work array of length m.
     *
420  *   the following parameters are newly introduced in this C translation:
     *
     *      evaluate is the name of the subroutine which calculates the functions.
     *        a default implementation lm_evaluate_default is provided in lm_eval.c;
     *        alternatively, evaluate can be provided by a user calling program.
425  *        it should be written as follows:
     *
     *        void evaluate ( double* par, int m_dat, double* fvec,
     *                        void *data, int *info )
     *        {
430  *           // for ( i=0; i<m_dat; ++i )
     *           //     calculate fvec[i] for given parameters par;
     *           // to stop the minimization,
     *           //     set *info to a negative integer.
     *        }
435  *
     *      printout is the name of the subroutine which nforms about fit progress.
     *        a default implementation lm_print_default is provided in lm_eval.c;
     *        alternatively, printout can be provided by a user calling program.
     *        it should be written as follows:
440  *
     *        void printout ( int n_par, double* par, int m_dat, double* fvec,
     *                        void *data, int iflag, int iter, int nfev )
     *        {
     *           // iflag : 0 (init) 1 (outer loop) 2(inner loop) -1(terminated)
445  *           // iter  : outer loop counter
     *           // nfev  : number of calls to *evaluate
     *        }
     *
     *      data is an input pointer to an arbitrary structure that is passed to
450  *        evaluate. typically, it contains experimental data to be fitted.
     *
     */
        int i, iter, j;
        double actred, delta, dirder, eps, fnorm, fnorm1, gnorm, par, pnorm,
455         prered, ratio, step, sum, temp, temp1, temp2, temp3, xnorm;
        static double p1 = 0.1;
        static double p5 = 0.5;
        static double p25 = 0.25;
        static double p75 = 0.75;
460     static double p0001 = 1.0e-4;

        *nfev = 0; // function evaluation counter
        iter = 1;  // outer loop counter
        par = 0;   // levenberg-marquardt parameter
465     delta = 0; // just to prevent a warning (initialization within if-clause)
        xnorm = 0; // dito

        temp = MAX(epsfcn,LM_MACHEP);
        eps = sqrt(temp); // used in calculating the Jacobian by forward differences
470
    // *** check the input parameters for errors.

        if ( (n <= 0) || (m < n) || (ftol < 0.)
            || (xtol < 0.) || (gtol < 0.) || (maxfev <= 0) || (factor <= 0.) )
475     {
            *info = 0; // invalid parameter
            return;
        }
        if ( mode == 2 )  /* scaling by diag[] */
480     {
            for ( j=0; j<n; j++ )  /* check for nonpositive elements */
            {
                if ( diag[j] <= 0.0 )
```

marquardt.c

```
                {
485                 *info = 0; // invalid parameter
                    return;
                }
            }
        }
490 #if BUG
        printf( "lmdif\n" );
    #endif

    // *** evaluate the function at the starting point and calculate its norm.
495
        *info = 0;
        (*evaluate)( x, m, fvec, data, info );
        (*printout)( n, x, m, fvec, data, 0, 0, ++(*nfev) );
        if ( *info < 0 ) return;
500     fnorm = lm_enorm(m,fvec);

    // *** the outer loop.

        do {
505 #if BUG
            printf( "lmdif/ outer loop iter=%d nfev=%d fnorm=%.10e\n",
                    iter, *nfev, fnorm );
    #endif

510 // O** calculate the jacobian matrix.

            for ( j=0; j<n; j++ )
            {
                temp = x[j];
515             step = eps * fabs(temp);
                if (step == 0.) step = eps;
                x[j] = temp + step;
                *info = 0;
                (*evaluate)( x, m, wa4, data, info );
520             (*printout)( n, x, m, wa4, data, 1, iter, ++(*nfev) );
                if ( *info < 0 ) return;  // user requested break
                x[j] = temp;
                for ( i=0; i<m; i++ )
                    fjac[j*m+i] = (wa4[i] - fvec[i]) / step;
525         }
    #if BUG>1
            // DEBUG: print the entire matrix
            for ( i=0; i<m; i++ )
            {
530             for ( j=0; j<n; j++ )
                    printf( "%.5e ", y[j*m+i] );
                printf( "\n" );
            }
    #endif
535
    // O** compute the qr factorization of the jacobian.

            lm_qrfac( m, n, fjac, 1, ipvt, wa1, wa2, wa3);

540 // O** on the first iteration ...

            if (iter == 1)
            {
                if (mode != 2)
545 //      ... scale according to the norms of the columns of the initial jacobian.
                {
                    for ( j=0; j<n; j++ )
                    {
                        diag[j] = wa2[j];
550                     if ( wa2[j] == 0. )
                            diag[j] = 1.;
                    }
```

marquardt.c

```
            }

//      ... calculate the norm of the scaled x and
//          initialize the step bound delta.

            for ( j=0; j<n; j++ )
                wa3[j] = diag[j] * x[j];

            xnorm = lm_enorm( n, wa3 );
            delta = factor*xnorm;
            if (delta == 0.)
                delta = factor;
        }

// O** form (q transpose)*fvec and store the first n components in qtf.

        for ( i=0; i<m; i++ )
            wa4[i] = fvec[i];

        for ( j=0; j<n; j++ )
        {
            temp3 = fjac[j*m+j];
            if (temp3 != 0.)
            {
                sum = 0;
                for ( i=j; i<m; i++ )
                    sum += fjac[j*m+i] * wa4[i];
                temp = -sum / temp3;
                for ( i=j; i<m; i++ )
                    wa4[i] += fjac[j*m+i] * temp;
            }
            fjac[j*m+j] = wa1[j];
            qtf[j] = wa4[j];
        }

// O** compute the norm of the scaled gradient and test for convergence.

        gnorm = 0;
        if ( fnorm != 0 )
        {
            for ( j=0; j<n; j++ )
            {
                if ( wa2[ ipvt[j] ] == 0 ) continue;

                sum = 0.;
                for ( i=0; i<=j; i++ )
                    sum += fjac[j*m+i] * qtf[i] / fnorm;
                gnorm = MAX( gnorm, fabs(sum/wa2[ ipvt[j] ]) );
            }
        }

        if ( gnorm <= gtol )
        {
            *info = 4;
            return;
        }

// O** rescale if necessary.

        if ( mode != 2 )
        {
            for ( j=0; j<n; j++ )
                diag[j] = MAX(diag[j],wa2[j]);
        }

// O** the inner loop.

        do {
#if BUG
```

marquardt.c

```
            printf( "lmdif/ inner loop iter=%d nfev=%d\n", iter, *nfev );
#endif

// OI* determine the levenberg-marquardt parameter.

            lm_lmpar( n,fjac,m,ipvt,diag,qtf,delta,&par,wa1,wa2,wa3,wa4 );

// OI* store the direction p and x + p. calculate the norm of p.

            for ( j=0; j<n; j++ )
            {
                wa1[j] = -wa1[j];
                wa2[j] = x[j] + wa1[j];
                wa3[j] = diag[j]*wa1[j];
            }
            pnorm = lm_enorm(n,wa3);

// OI* on the first iteration, adjust the initial step bound.

            if ( *nfev<= 1+n ) // bug corrected by J. Wuttke in 2004
                delta = MIN(delta,pnorm);

// OI* evaluate the function at x + p and calculate its norm.

            *info = 0;
            (*evaluate)( wa2, m, wa4, data, info );
            (*printout)( n, x, m, wa4, data, 2, iter, ++(*nfev) );
            if ( *info < 0 ) return;  // user requested break

            fnorm1 = lm_enorm(m,wa4);
#if BUG
            printf( "lmdif/ pnorm %.10e  fnorm1 %.10e  fnorm %.10e"
                    " delta=%.10e par=%.10e\n",
                    pnorm, fnorm1, fnorm, delta, par );
#endif

// OI* compute the scaled actual reduction.

            if ( p1*fnorm1 < fnorm )
                actred = 1 - SQR( fnorm1/fnorm );
            else
                actred = -1;

// OI* compute the scaled predicted reduction and
//     the scaled directional derivative.

            for ( j=0; j<n; j++ )
            {
                wa3[j] = 0;
                for ( i=0; i<=j; i++ )
                    wa3[i] += fjac[j*m+i]*wa1[ ipvt[j] ];
            }
            temp1 = lm_enorm(n,wa3) / fnorm;
            temp2 = sqrt(par) * pnorm / fnorm;
            prered = SQR(temp1) + 2 * SQR(temp2);
            dirder = - ( SQR(temp1) + SQR(temp2) );

// OI* compute the ratio of the actual to the predicted reduction.

            ratio = prered!=0 ? actred/prered : 0;
#if BUG
            printf( "lmdif/ actred=%.10e prered=%.10e ratio=%.10e"
                    " sq(1)=%.10e sq(2)=%.10e dd=%.10e\n",
                    actred, prered, prered!=0 ? ratio : 0.,
                    SQR(temp1), SQR(temp2), dirder );
#endif

// OI* update the step bound.
```

marquardt.c

```
            if (ratio <= p25)
            {
                if (actred >= 0.)
                    temp = p5;
695             else
                    temp = p5*dirder/(dirder + p5*actred);
                if ( p1*fnorm1 >= fnorm || temp < p1 )
                    temp = p1;
                delta = temp * MIN(delta,pnorm/p1);
700             par /= temp;
            }
            else if ( par == 0. || ratio >= p75 )
            {
                delta = pnorm/p5;
705             par *= p5;
            }

    // OI* test for successful iteration...

710         if (ratio >= p0001)
            {

    //      ... successful iteration. update x, fvec, and their norms.

715             for ( j=0; j<n; j++ )
                {
                    x[j] = wa2[j];
                    wa2[j] = diag[j]*x[j];
                }
720             for ( i=0; i<m; i++ )
                    fvec[i] = wa4[i];
                xnorm = lm_enorm(n,wa2);
                fnorm = fnorm1;
                iter++;
725         }
    #if BUG
            else {
                printf( "ATTN: iteration considered unsuccessful\n" );
            }
730 #endif

    // OI* tests for convergence ( otherwise *info = 1, 2, or 3 )

            *info = 0; // do not terminate (unless overwritten by nonzero value)
735         if ( fabs(actred) <= ftol && prered <= ftol && p5*ratio <= 1 )
                *info = 1;
            if (delta <= xtol*xnorm)
                *info += 2;
            if ( *info != 0)
740             return;

    // OI* tests for termination and stringent tolerances.

            if ( *nfev >= maxfev)
745             *info = 5;
            if ( fabs(actred) <= LM_MACHEP &&
                 prered <= LM_MACHEP && p5*ratio <= 1 )
                *info = 6;
            if (delta <= LM_MACHEP*xnorm)
750             *info = 7;
            if (gnorm <= LM_MACHEP)
                *info = 8;
            if ( *info != 0)
                return;
755
    // OI* end of the inner loop. repeat if iteration unsuccessful.

        } while (ratio < p0001);
```

marquardt.c

```
760 // O** end of the outer loop.

    } while (1);

}
765



void lm_lmpar(int n, double* r, int ldr, int* ipvt, double* diag, double* qtb,
              double delta, double* par, double* x, double* sdiag,
770           double* wa1, double* wa2)
{
/*     given an m by n matrix a, an n by n nonsingular diagonal
 *     matrix d, an m-vector b, and a positive number delta,
 *     the problem is to determine a value for the parameter
775  *     par such that if x solves the system
 *
 *          a*x = b ,     sqrt(par)*d*x = 0 ,
 *
 *     in the least squares sense, and dxnorm is the euclidean
780  *     norm of d*x, then either par is 0. and
 *
 *          (dxnorm-delta) .le. 0.1*delta ,
 *
 *     or par is positive and
785  *
 *          abs(dxnorm-delta) .le. 0.1*delta .
 *
 *     this subroutine completes the solution of the problem
 *     if it is provided with the necessary information from the
790  *     qr factorization, with column pivoting, of a. that is, if
 *     a*p = q*r, where p is a permutation matrix, q has orthogonal
 *     columns, and r is an upper triangular matrix with diagonal
 *     elements of nonincreasing magnitude, then lmpar expects
 *     the full upper triangle of r, the permutation matrix p,
795  *     and the first n components of (q transpose)*b. on output
 *     lmpar also provides an upper triangular matrix s such that
 *
 *           t   t                   t
 *          p *(a *a + par*d*d)*p = s *s .
800  *
 *     s is employed within lmpar and may be of separate interest.
 *
 *     only a few iterations are generally needed for convergence
 *     of the algorithm. if, however, the limit of 10 iterations
805  *     is reached, then the output par will contain the best
 *     value obtained so far.
 *
 *     parameters:
 *
810  *      n is a positive integer input variable set to the order of r.
 *
 *      r is an n by n array. on input the full upper triangle
 *        must contain the full upper triangle of the matrix r.
 *        on output the full upper triangle is unaltered, and the
815  *        strict lower triangle contains the strict upper triangle
 *        (transposed) of the upper triangular matrix s.
 *
 *      ldr is a positive integer input variable not less than n
 *        which specifies the leading dimension of the array r.
820  *
 *      ipvt is an integer input array of length n which defines the
 *        permutation matrix p such that a*p = q*r. column j of p
 *        is column ipvt(j) of the identity matrix.
 *
825  *      diag is an input array of length n which must contain the
 *        diagonal elements of the matrix d.
 *
 *      qtb is an input array of length n which must contain the first
```

marquardt.c

```
 *        n elements of the vector (q transpose)*b.
 *
 *      delta is a positive input variable which specifies an upper
 *        bound on the euclidean norm of d*x.
 *
 *      par is a nonnegative variable. on input par contains an
 *        initial estimate of the levenberg-marquardt parameter.
 *        on output par contains the final estimate.
 *
 *      x is an output array of length n which contains the least
 *        squares solution of the system a*x = b, sqrt(par)*d*x = 0,
 *        for the output par.
 *
 *      sdiag is an output array of length n which contains the
 *        diagonal elements of the upper triangular matrix s.
 *
 *      wa1 and wa2 are work arrays of length n.
 *
 */
    int i, iter, j, nsing;
    double dxnorm, fp, fp_old, gnorm, parc, parl, paru;
    double sum, temp;
    static double p1 = 0.1;
    static double p001 = 0.001;

#if BUG
    printf( "lmpar\n" );
#endif

// *** compute and store in x the gauss-newton direction. if the
//     jacobian is rank-deficient, obtain a least squares solution.

    nsing = n;
    for ( j=0; j<n; j++ )
    {
        wa1[j] = qtb[j];
        if ( r[j*ldr+j] == 0 && nsing == n )
            nsing = j;
        if (nsing < n)
            wa1[j] = 0;
    }
#if BUG
    printf( "nsing %d ", nsing );
#endif
    for ( j=nsing-1; j>=0; j-- )
    {
        wa1[j] = wa1[j]/r[j+ldr*j];
        temp = wa1[j];
      }

    for ( j=0; j<n; j++ )
        x[ ipvt[j] ] = wa1[j];

// *** initialize the iteration counter.
//     evaluate the function at the origin, and test
//     for acceptance of the gauss-newton direction.

    iter = 0;
    for ( j=0; j<n; j++ )
        wa2[j] = diag[j]*x[j];
    dxnorm = lm_enorm(n,wa2);
    fp = dxnorm - delta;
    if (fp <= p1*delta)
    {
#if BUG
        printf( "lmpar/ terminate (fp<delta/10\n" );
#endif
        *par = 0;
        return;
```

marquardt.c

```
    }

// *** if the jacobian is not rank deficient, the newton
//     step provides a lower bound, parl, for the 0. of
//     the function. otherwise set this bound to 0..

    parl = 0;
    if (nsing >= n)
    {
        for ( j=0; j<n; j++ )
            wa1[j] = diag[ ipvt[j] ] * wa2[ ipvt[j] ] / dxnorm;

        for ( j=0; j<n; j++ )
        {
            sum = 0.;
            for ( i=0; i<j; i++ )
                sum += r[j*ldr+i]*wa1[i];
            wa1[j] = (wa1[j] - sum)/r[j+ldr*j];
        }
        temp = lm_enorm(n,wa1);
        parl = fp/delta/temp/temp;
    }

// *** calculate an upper bound, paru, for the 0. of the function.

    for ( j=0; j<n; j++ )
    {
        sum = 0;
        for ( i=0; i<=j; i++ )
            sum += r[j*ldr+i]*qtb[i];
        wa1[j] = sum/diag[ ipvt[j] ];
    }
    gnorm = lm_enorm(n,wa1);
    paru = gnorm/delta;
    if (paru == 0.)
        paru = LM_DWARF/MIN(delta,p1);

// *** if the input par lies outside of the interval (parl,paru),
//     set par to the closer endpoint.

    *par = MAX( *par,parl);
    *par = MIN( *par,paru);
    if ( *par == 0.)
        *par = gnorm/dxnorm;
#if BUG
    printf( "lmpar/ parl %.4e  par %.4e  paru %.4e\n", parl, *par, paru );
#endif

// *** iterate.

    for ( ; ; iter++ ) {

// *** evaluate the function at the current value of par.

        if ( *par == 0.)
            *par = MAX(LM_DWARF,p001*paru);
        temp = sqrt( *par );
        for ( j=0; j<n; j++ )
            wa1[j] = temp*diag[j];
        lm_qrsolv( n, r, ldr, ipvt, wa1, qtb, x, sdiag, wa2);
        for ( j=0; j<n; j++ )
            wa2[j] = diag[j]*x[j];
        dxnorm = lm_enorm(n,wa2);
        fp_old = fp;
        fp = dxnorm - delta;

// ***   if the function is small enough, accept the current value
//       of par. also test for the exceptional cases where parl
//       is 0. or the number of iterations has reached 10.
```

marquardt.c

```
            if ( fabs(fp) <= p1*delta
                 || (parl == 0. && fp <= fp_old && fp_old < 0.)
                 || iter == 10 )
                break; // the only exit from this loop

        // *** compute the Newton correction.

            for ( j=0; j<n; j++ )
                wa1[j] = diag[ ipvt[j] ] * wa2[ ipvt[j] ] / dxnorm;

            for ( j=0; j<n; j++ )
            {
                wa1[j] = wa1[j]/sdiag[j];
                for ( i=j+1; i<n; i++ )
                    wa1[i] -= r[j*ldr+i]*wa1[j];
            }
            temp = lm_enorm( n, wa1);
            parc = fp/delta/temp/temp;

        // *** depending on the sign of the function, update parl or paru.

            if (fp > 0)
                parl = MAX(parl, *par);
            else if (fp < 0)
                paru = MIN(paru, *par);
            // the case fp==0 is precluded by the break condition

        // *** compute an improved estimate for par.

            *par = MAX(parl, *par + parc);

        }

    }



    void lm_qrfac(int m, int n, double* a, int pivot, int* ipvt,
                  double* rdiag, double* acnorm, double* wa)
    {
    /*
     *     this subroutine uses householder transformations with column
     *     pivoting (optional) to compute a qr factorization of the
     *     m by n matrix a. that is, qrfac determines an orthogonal
     *     matrix q, a permutation matrix p, and an upper trapezoidal
     *     matrix r with diagonal elements of nonincreasing magnitude,
     *     such that a*p = q*r. the householder transformation for
     *     column k, k = 1,2,...,min(m,n), is of the form
     *
     *                            t
     *           i - (1/u(k))*u*u
     *
     *     where u has 0.s in the first k-1 positions. the form of
     *     this transformation and the method of pivoting first
     *     appeared in the corresponding linpack subroutine.
     *
     *     parameters:
     *
     *      m is a positive integer input variable set to the number
     *        of rows of a.
     *
     *      n is a positive integer input variable set to the number
     *        of columns of a.
     *
     *      a is an m by n array. on input a contains the matrix for
     *        which the qr factorization is to be computed. on output
     *        the strict upper trapezoidal part of a contains the strict
     *        upper trapezoidal part of r, and the lower trapezoidal
```

marquardt.c

```
     *        part of a contains a factored form of q (the non-trivial
     *        elements of the u vectors described above).
     *
     *      pivot is a logical input variable. if pivot is set true,
     *        then column pivoting is enforced. if pivot is set false,
     *        then no column pivoting is done.
     *
     *      ipvt is an integer output array of length lipvt. ipvt
     *        defines the permutation matrix p such that a*p = q*r.
     *        column j of p is column ipvt(j) of the identity matrix.
     *        if pivot is false, ipvt is not referenced.
     *
     *      rdiag is an output array of length n which contains the
     *        diagonal elements of r.
     *
     *      acnorm is an output array of length n which contains the
     *        norms of the corresponding columns of the input matrix a.
     *        if this information is not needed, then acnorm can coincide
     *        with rdiag.
     *
     *      wa is a work array of length n. if pivot is false, then wa
     *        can coincide with rdiag.
     *
     */
        int i, j, k, kmax, minmn;
        double ajnorm, sum, temp;
        static double p05 = 0.05;

    // *** compute the initial column norms and initialize several arrays.

        for ( j=0; j<n; j++ )
        {
            acnorm[j] = lm_enorm(m, &a[j*m]);
            rdiag[j] = acnorm[j];
            wa[j] = rdiag[j];
            if ( pivot )
                ipvt[j] = j;
        }
    #if BUG
        printf( "qrfac\n" );
    #endif

    // *** reduce a to r with householder transformations.

        minmn = MIN(m,n);
        for ( j=0; j<minmn; j++ )
        {
            if ( !pivot ) goto pivot_ok;

    // *** bring the column of largest norm into the pivot position.

            kmax = j;
            for ( k=j+1; k<n; k++ )
                if (rdiag[k] > rdiag[kmax])
                    kmax = k;
            if (kmax == j) goto pivot_ok; // bug fixed in rel 2.1

            for ( i=0; i<m; i++ )
            {
                temp        = a[j*m+i];
                a[j*m+i]    = a[kmax*m+i];
                a[kmax*m+i] = temp;
            }
            rdiag[kmax] = rdiag[j];
            wa[kmax] = wa[j];
            k = ipvt[j];
            ipvt[j] = ipvt[kmax];
            ipvt[kmax] = k;
```

marquardt.c

```
    pivot_ok:

// *** compute the Householder transformation to reduce the
//     j-th column of a to a multiple of the j-th unit vector.

        ajnorm = lm_enorm( m-j, &a[j*m+j] );
        if (ajnorm == 0.)
        {
            rdiag[j] = 0;
            continue;
        }

        if (a[j*m+j] < 0.)
            ajnorm = -ajnorm;
        for ( i=j; i<m; i++ )
            a[j*m+i] /= ajnorm;
        a[j*m+j] += 1;

// *** apply the transformation to the remaining columns
//     and update the norms.

        for ( k=j+1; k<n; k++ )
        {
            sum = 0;

            for ( i=j; i<m; i++ )
                sum += a[j*m+i]*a[k*m+i];

            temp = sum/a[j+m*j];

            for ( i=j; i<m; i++ )
                a[k*m+i] -= temp * a[j*m+i];

            if ( pivot && rdiag[k] != 0. )
            {
                temp = a[m*k+j]/rdiag[k];
                temp = MAX( 0., 1-temp*temp );
                rdiag[k] *= sqrt(temp);
                temp = rdiag[k]/wa[k];
                if ( p05*SQR(temp) <= LM_MACHEP )
                {
                    rdiag[k] = lm_enorm( m-j-1, &a[m*k+j+1]);
                    wa[k] = rdiag[k];
                }
            }
        }

        rdiag[j] = -ajnorm;
    }
}



void lm_qrsolv(int n, double* r, int ldr, int* ipvt, double* diag,
               double* qtb, double* x, double* sdiag, double* wa)
{
/*
 *     given an m by n matrix a, an n by n diagonal matrix d,
 *     and an m-vector b, the problem is to determine an x which
 *     solves the system
 *
 *          a*x = b ,     d*x = 0 ,
 *
 *     in the least squares sense.
 *
 *     this subroutine completes the solution of the problem
 *     if it is provided with the necessary information from the
 *     qr factorization, with column pivoting, of a. that is, if
 *     a*p = q*r, where p is a permutation matrix, q has orthogonal
```

marquardt.c

```
 *     columns, and r is an upper triangular matrix with diagonal
 *     elements of nonincreasing magnitude, then qrsolv expects
 *     the full upper triangle of r, the permutation matrix p,
 *     and the first n components of (q transpose)*b. the system
 *     a*x = b, d*x = 0, is then equivalent to
 *
 *                  t       t
 *          r*z = q *b ,  p *d*p*z = 0 ,
 *
 *     where x = p*z. if this system does not have full rank,
 *     then a least squares solution is obtained. on output qrsolv
 *     also provides an upper triangular matrix s such that
 *
 *           t   t               t
 *          p *(a *a + d*d)*p = s *s .
 *
 *     s is computed within qrsolv and may be of separate interest.
 *
 *     parameters
 *
 *      n is a positive integer input variable set to the order of r.
 *
 *      r is an n by n array. on input the full upper triangle
 *        must contain the full upper triangle of the matrix r.
 *        on output the full upper triangle is unaltered, and the
 *        strict lower triangle contains the strict upper triangle
 *        (transposed) of the upper triangular matrix s.
 *
 *      ldr is a positive integer input variable not less than n
 *        which specifies the leading dimension of the array r.
 *
 *      ipvt is an integer input array of length n which defines the
 *        permutation matrix p such that a*p = q*r. column j of p
 *        is column ipvt(j) of the identity matrix.
 *
 *      diag is an input array of length n which must contain the
 *        diagonal elements of the matrix d.
 *
 *      qtb is an input array of length n which must contain the first
 *        n elements of the vector (q transpose)*b.
 *
 *      x is an output array of length n which contains the least
 *        squares solution of the system a*x = b, d*x = 0.
 *
 *      sdiag is an output array of length n which contains the
 *        diagonal elements of the upper triangular matrix s.
 *
 *      wa is a work array of length n.
 *
 */
    int i, kk, j, k, nsing;
    double qtbpj, sum, temp;
    double sin, cos, tan, cotan; // these are local variables, not functions
    static double p25 = 0.25;
    static double p5 = 0.5;

// *** copy r and (q transpose)*b to preserve input and initialize s.
//     in particular, save the diagonal elements of r in x.

    for ( j=0; j<n; j++ )
    {
        for ( i=j; i<n; i++ )
            r[j*ldr+i] = r[i*ldr+j];
        x[j] = r[j*ldr+j];
        wa[j] = qtb[j];
    }
#if BUG
    printf( "qrsolv\n" );
#endif
```

marquardt.c

```
    // *** eliminate the diagonal matrix d using a givens rotation.
1245
        for ( j=0; j<n; j++ )
        {

    // ***   prepare the row of d to be eliminated, locating the
1250 //       diagonal element using p from the qr factorization.

            if (diag[ ipvt[j] ] == 0.)
                goto L90;
            for ( k=j; k<n; k++ )
1255            sdiag[k] = 0.;
            sdiag[j] = diag[ ipvt[j] ];

    // ***   the transformations to eliminate the row of d
    //       modify only a single element of (q transpose)*b
1260 //       beyond the first n, which is initially 0..

            qtbpj = 0.;
            for ( k=j; k<n; k++ )
            {
1265
    //          determine a givens rotation which eliminates the
    //          appropriate element in the current row of d.

                if (sdiag[k] == 0.)
1270                continue;
                kk = k + ldr * k; // <! keep this shorthand !>
                if ( fabs(r[kk]) < fabs(sdiag[k]) )
                {
                    cotan = r[kk]/sdiag[k];
1275                sin = p5/sqrt(p25+p25*SQR(cotan));
                    cos = sin*cotan;
                }
                else
                {
1280                tan = sdiag[k]/r[kk];
                    cos = p5/sqrt(p25+p25*SQR(tan));
                    sin = cos*tan;
                }

1285 // ***      compute the modified diagonal element of r and
    //          the modified element of ((q transpose)*b,0).

                r[kk] = cos*r[kk] + sin*sdiag[k];
                temp = cos*wa[k] + sin*qtbpj;
1290            qtbpj = -sin*wa[k] + cos*qtbpj;
                wa[k] = temp;

    // *** accumulate the tranformation in the row of s.

1295            for ( i=k+1; i<n; i++ )
                {
                    temp = cos*r[k*ldr+i] + sin*sdiag[i];
                    sdiag[i] = -sin*r[k*ldr+i] + cos*sdiag[i];
                    r[k*ldr+i] = temp;
1300            }
            }
        L90:

    // *** store the diagonal element of s and restore
1305 //     the corresponding diagonal element of r.

            sdiag[j] = r[j*ldr+j];
            r[j*ldr+j] = x[j];
        }
1310
    // *** solve the triangular system for z. if the system is
```

marquardt.c

```
    //     singular, then obtain a least squares solution.

        nsing = n;
1315    for ( j=0; j<n; j++ )
        {
            if ( sdiag[j] == 0. && nsing == n )
                nsing = j;
            if (nsing < n)
1320            wa[j] = 0;
        }

        for ( j=nsing-1; j>=0; j-- )
        {
1325        sum = 0;
            for ( i=j+1; i<nsing; i++ )
                sum += r[j*ldr+i]*wa[i];
            wa[j] = (wa[j] - sum)/sdiag[j];
        }
1330
    // *** permute the components of z back to components of x.

        for ( j=0; j<n; j++ )
            x[ ipvt[j] ] = wa[j];
1335 }



    double lm_enorm( int n, double *x )
1340 {
    /*     given an n-vector x, this function calculates the
     *     euclidean norm of x.
     *
     *     the euclidean norm is computed by accumulating the sum of
1345 *     squares in three different sums. the sums of squares for the
     *     small and large components are scaled so that no overflows
     *     occur. non-destructive underflows are permitted. underflows
     *     and overflows do not occur in the computation of the unscaled
     *     sum of squares for the intermediate components.
1350 *     the definitions of small, intermediate and large components
     *     depend on two constants, LM_SQRT_DWARF and LM_SQRT_GIANT. the main
     *     restrictions on these constants are that LM_SQRT_DWARF**2 not
     *     underflow and LM_SQRT_GIANT**2 not overflow.
     *
1355 *     parameters
     *
     *      n is a positive integer input variable.
     *
     *      x is an input array of length n.
1360 */
        int i;
        double agiant, s1, s2, s3, xabs, x1max, x3max, temp;

        s1 = 0;
1365    s2 = 0;
        s3 = 0;
        x1max = 0;
        x3max = 0;
        agiant = LM_SQRT_GIANT/( (double) n);
1370
        for ( i=0; i<n; i++ )
        {
            xabs = fabs(x[i]);
            if ( xabs > LM_SQRT_DWARF && xabs < agiant )
1375        {
    // **  sum for intermediate components.
                s2 += xabs*xabs;
                continue;
            }
1380
```

marquardt.c

```
            if ( xabs >  LM_SQRT_DWARF )
            {
    // **  sum for large components.
                if (xabs > x1max)
1385            {
                    temp = x1max/xabs;
                    s1 = 1 + s1*SQR(temp);
                    x1max = xabs;
                }
1390            else
                {
                    temp = xabs/x1max;
                    s1 += SQR(temp);
                }
1395            continue;
            }
    // **  sum for small components.
            if (xabs > x3max)
            {
1400            temp = x3max/xabs;
                s3 = 1 + s3*SQR(temp);
                x3max = xabs;
            }
            else
1405        {
                if (xabs != 0.)
                {
                    temp = xabs/x3max;
                    s3 += SQR(temp);
1410            }
            }
        }

    // *** calculation of norm.
1415
        if (s1 != 0)
            return x1max*sqrt(s1 + (s2/x1max)/x1max);
        if (s2 != 0)
        {
1420        if (s2 >= x3max)
                return sqrt( s2*(1+(x3max/s2)*(x3max*s3)) );
            else
                return sqrt( x3max*((s2/x3max)+(x3max*s3)) );
        }
1425
        return x3max*sqrt(s3);
    }
```

B.4. Iteration der monodispersen NEP

nep_mono.c

```
// compile with: gcc -Wall -pedantic -funroll-loops -std=c99 -lgsl -lgslcblas -l
m -O3
#include <stdio.h>
#include <stdlib.h>
#define __USE_GNU
#define __USE_ISOC99
#include <math.h>
#include <string.h>
#include <stdarg.h>
#include <time.h>

//#define __DEBUG__

#define STARTVALUE 2



typedef struct {
        double *matrix;
        double *ptr, *ptr1;
        unsigned int size;
} t_matrix;

t_matrix init_matrix (unsigned int size) {
        t_matrix matrix = {
                .size = size,
                .matrix = (double *) calloc ((size_t)(size), (size_t)(sizeof (do
uble))),
                .ptr = NULL
        };
        if (!matrix.matrix)
                exit (-1);
        return matrix;
}

void cleanup_matrix (t_matrix matrix) {
        free (matrix.matrix);
}



int out (const char *format, ...) {
        va_list arg;
        int done;

        va_start (arg, format);
        done = vfprintf (stdout, format, arg);
        fflush (stdout);
        va_end (arg);

        return done;
}

void die (const char *format, ...) {
        va_list arg;
        int done;

        va_start (arg, format);
        done = vfprintf (stdout, format, arg);
        fflush (stdout);
        va_end (arg);

        exit (-1);
}

void matrix_out (t_matrix x, t_matrix y, char *filename) {
        FILE *out_file = fopen (filename, "w");
        unsigned int i;
```

nep_mono.c

```
        if (!out_file)
                die ("matrix_out");

        for (i = 0, x.ptr = x.matrix, y.ptr = y.matrix; i < x.size; i++, x.ptr++
, y.ptr++)
                fprintf (out_file, "\n%lf %lf", *x.ptr, *y.ptr);

        fflush (out_file);
        fclose (out_file);
}

void read_function (t_matrix x, t_matrix function, char *name) {
        unsigned int i;
        FILE *function_file = fopen (name, "r");
        if (!function_file)
                die ("\nread_function: %s\n", name);

        for (i = 0, x.ptr = x.matrix, function.ptr = function.matrix; i < functi
on.size; i++, x.ptr++, function.ptr++)
                fscanf (function_file, "\n%lf %lf", x.ptr, function.ptr);

        fclose (function_file);
}



void spline (t_matrix x, t_matrix y, t_matrix y2) {
        unsigned int i, k;
        double p, qn, sig, un;
        t_matrix u = init_matrix (x.size-1);

        u.matrix[0] = 0.0;

        for (i = 1; i < x.size-1; i++) {
                sig = (x.matrix[i]-x.matrix[i-1])/(x.matrix[i+1]-x.matrix[i-1]);
                p = sig*y2.matrix[i-1]+2.0;
                y2.matrix[i] = (sig-1.0)/p;
                u.matrix[i] = (y.matrix[i+1]-y.matrix[i])/(x.matrix[i+1]-x.matri
x[i])-(y.matrix[i]-y.matrix[i-1])/(x.matrix[i]-x.matrix[i-1]);
                u.matrix[i] = (6.0*u.matrix[i]/(x.matrix[i+1]-x.matrix[i-1])-sig
*u.matrix[i-1])/p;
        }

        qn = un = 0.0;

        y2.matrix[y2.size-1] = (un-qn*u.matrix[y2.size-2])/(qn*y2.matrix[y2.size
-2]+1.0);
        for (k = y2.size-2; k > 0; k--)
                y2.matrix[k] = y2.matrix[k]*y2.matrix[k+1]+u.matrix[k];

        cleanup_matrix (u);
}

double splint (t_matrix x, t_matrix y, t_matrix y2, double x_) {
        int klo = 0, khi = x.size-1, k;
        double h, b, a;

        while (khi-klo > 1) {
                k = (khi+klo) >> 1;
                if (x.matrix[k] > x_)
                        khi = k;
                else
                        klo = k;
        }

        h = x.matrix[khi]-x.matrix[klo];
        if (h == 0.0)
                return (double)(0.0);
        a = (x.matrix[khi]-x_)/h;
```

nep_mono.c

```
        b = (x_-x.matrix[klo])/h;
        return (double)(a*y.matrix[klo]+b*y.matrix[khi]+((a*a*a-a)*y2.matrix[klo
]+(b*b*b-b)*y2.matrix[khi])*(h*h)/6.0);
}
135



int main (int argc, char *argv[]) {
        out ("\n *** NEP monodispers ***");
140
        out ("\n--> Init");
        unsigned int l,
                     i, j;
        unsigned long int itera = 0;
145
        unsigned int size = 300, in_size = 1020;
        double n = 0.6;
        double qmin = 0.5, qmax = 50.0;
        char *sq_infilename = NULL;
150     out ("\nNur N, qmax angeben ( Sperls Diskretisierung )");
        if (argc != 1+5)
                die ("\ncmd <points> <density [s^2]> <qmax [s]> <sq_infilename> <insize>\n");
        else {
                size = (unsigned int)(atoi (argv[1]));
155             n = (double)(atof (argv[2]));
                qmax = (double)(atof (argv[3]));
                sq_infilename = argv[4];
                in_size = (unsigned int)(atoi (argv[5]));
                qmin = .5*qmax/(double)(size);
160     }

        out ("\n\t* size=%d n=%g qmin=%g qmax=%g sq_infilename:%s in_size:%d", size, n, qmin,
 qmax, sq_infilename, in_size);

        t_matrix sq_bessel = init_matrix (in_size);
165     t_matrix cq_bessel = init_matrix (in_size);
        t_matrix q_bessel = init_matrix (in_size);
        t_matrix sq_deriv = init_matrix (in_size);
        t_matrix cq_deriv = init_matrix (in_size);

170     t_matrix sq = init_matrix (size);
        t_matrix cq = init_matrix (size);

        t_matrix q = init_matrix (size);
        t_matrix fq_new = init_matrix (size);
175     t_matrix fq_old = init_matrix (size);
        t_matrix k = init_matrix (size);
        t_matrix p = init_matrix (size);
        t_matrix mct = init_matrix (size);


180
        out ("\n\t* qkp");
        double dq = qmax/(double)(size);
        for (i = 0, q.ptr = q.matrix, p.ptr = p.matrix, k.ptr = k.matrix; i < q.
size; i++, q.ptr++, p.ptr++, k.ptr++)
                *q.ptr = *p.ptr = *k.ptr = qmin+(double)(i)*dq;
185
        out ("\n\t* S(q)");
        FILE *sq_infile = fopen (sq_infilename, "r");
        if (!sq_infile)
                die ("sq_infile");
190     for (i = 0, sq_bessel.ptr = sq_bessel.matrix, q_bessel.ptr = q_bessel.ma
trix; i < sq_bessel.size; i++, sq_bessel.ptr++, q_bessel.ptr++)
                fscanf (sq_infile, "\n%lf %lf", q_bessel.ptr, sq_bessel.ptr);
        fclose (sq_infile);
        if (qmin < q_bessel.matrix[0] || qmax > q_bessel.matrix[q_bessel.size-1]
)
                out ("\nqmin:%lf %lf qmax: %lf %lf\n", qmin, q_bessel.matrix[0], qmax, q
_bessel.matrix[q_bessel.size-1]);
```

nep_mono.c

```
195     spline (q_bessel, sq_bessel, sq_deriv);
        for (i = 0, sq.ptr = sq.matrix, q.ptr = q.matrix; i < sq.size; i++, sq.p
tr++, q.ptr++)
                *sq.ptr = splint (q_bessel, sq_bessel, sq_deriv, *q.ptr);

        out (" c(q)");
200     for (i = 0, cq.ptr = cq.matrix, sq.ptr = sq.matrix; i < cq.size; i++, cq
.ptr++, sq.ptr++)
                *cq.ptr = (*sq.ptr-1.0)/ *sq.ptr / n;


        out ("\n\t* Iterationskonstanten");
205     double iconst = n/8.0*(double)(powl(2.0*M_PIl,-2.0)*dq*dq)*2.0;
        out ("\niconst:%lf", iconst);

        unsigned int jmax = 0;
        unsigned int jmin = 0;
210     int j_ = 0;
        const unsigned int j_const = (unsigned int)(q.matrix[0]/dq);

        double q2, k2, p2;
        double q5;
215     double abs_sq;
        double sq_rt;
        const double renorm1 = 3.0/8.0, renorm2 = 7.0/6.0, renorm3 = 23.0/24.0;
        double sqa;
        double summand_inner = 0.0, summand_outer = 0.0;
220     double step = 0.5;
        const double step_multiplicator = 1.1;

        double *sp, *sk;

225     out ("\n--> Startwert f(q) ");
#if STARTVALUE == 1
        out ("f(q) = S(q)");
        for (i = 0, fq_new.ptr = fq_new.matrix, sq.ptr = sq.matrix; i < fq_new.s
ize; i++, fq_new.ptr++, sq.ptr++)
                *fq_new.ptr = *sq.ptr;
230 #elif STARTVALUE == 2
        out ("einlesen (muß gleiche Skalierung der qs haben!) 'fq_infile'");

        t_matrix q_tmp = init_matrix (q.size);
        read_function (q_tmp, fq_new, "fq_infile");
235     cleanup_matrix (q_tmp);
        step = 0.9;
        out ("  passe auch Schrittweite an: %lf", step);
#endif

240     out ("\n--> Beginne Iteration");
        double norm_fq_new = 0.0, norm_fq_old = 0.0, norm_fq_adap1 = 0.0;
        double norm_fq_max = 1e-8;

        time_t start, now;
245     time (&start);

        do {
                norm_fq_adap1 = norm_fq_old;
                norm_fq_old = norm_fq_new;
250             norm_fq_new = 0.0;
                for (i = 0, fq_new.ptr = fq_new.matrix, fq_old.ptr = fq_old.matr
ix; i < fq_new.size; i++, fq_new.ptr++, fq_old.ptr++) {
                        *fq_old.ptr = *fq_new.ptr;
                        norm_fq_new += *fq_new.ptr**fq_new.ptr;
                }
255             norm_fq_new = sqrt (norm_fq_new);
                out ("\n(i:%ld) (n:%.8f) (d:%.8f) (s:%f)", ++itera, norm_fq_new, fabs(norm_
fq_old-norm_fq_new), step);

                out (" m");
```

nep_mono.c

```
                for (l = 0, mct.ptr = mct.matrix, q.ptr = q.matrix, sq.ptr = sq.
matrix; l < mct.size; l++, mct.ptr++, q.ptr++, sq.ptr++) {
                        *mct.ptr = 0.0;
                        q5 = 1.0/(*q.ptr**q.ptr**q.ptr**q.ptr**q.ptr);
                        for (i = 0, k.ptr = k.matrix, sk = sq.matrix, fq_new.ptr
 = fq_new.matrix, cq.ptr = cq.matrix; i < sq.size; i++, k.ptr++, sk++, fq_new.pt
r++, cq.ptr++) {
                                summand_outer = 0.0;
                                j_ = abs(l-i)-(int)(j_const);
                                jmin = (j_ > 0) ? (unsigned int)(j_) : 0;
                                j_ = i+l+(int)(j_const);
                                jmax = ((unsigned int)(j_) < size) ? j_ : (unsig
ned int)(size);
                                for (j = jmin, p.ptr = p.matrix+jmin, sp = sq.ma
trix+jmin, fq_new.ptr1 = fq_new.matrix+jmin, cq.ptr1 = cq.matrix+jmin; j < jmax;
 j++, p.ptr++, sp++, fq_new.ptr1++, cq.ptr1++) {
                                        q2 = *q.ptr**q.ptr;
                                        p2 = *p.ptr**p.ptr;
                                        k2 = *k.ptr**k.ptr;
                                        abs_sq = (q2-p2+k2)**cq.ptr+(q2-k2+p2)**
cq.ptr1;
                                        abs_sq *= abs_sq;
                                        sq_rt = (q2-p2+k2)/(2.0**q.ptr);
                                        sq_rt *= sq_rt;
                                        sqa = pow(k2-sq_rt,-0.5);


                                        if (isnan (sqa) || isinf (sqa))
                                                die ("nan/inf: inner");

                                        summand_inner = *p.ptr**sp*sqa*abs_sq**f
q_new.ptr1;

                                        if (j == 0 || j == jmax-1)
                                                summand_inner *= renorm1;
                                        else if (j == 1 || j == jmax-2)
                                                summand_inner *= renorm2;
                                        else if (j == 2 || j == jmax-3)
                                                summand_inner *= renorm3;

                                        summand_outer += summand_inner;
                                }
                                summand_outer *= *k.ptr**sk**fq_new.ptr*q5;

                                if (isnan (summand_outer) || isinf (summand_oute
r))
                                        die ("nan/inf: outer");

                                if (i == 0 || i == sq.size-1)
                                        summand_outer *= renorm1;
                                else if (i == 1 || i == sq.size-2)
                                        summand_outer *= renorm2;
                                else if (i == 2 || i == sq.size-3)
                                        summand_outer *= renorm3;

                                *mct.ptr += summand_outer;
                        }
                        *mct.ptr *= iconst**sq.ptr;

                        if (isnan (*mct.ptr) || isinf (*mct.ptr))
                                die ("nan/inf: mct");
                }

                if (fabs(norm_fq_adap1-norm_fq_old) > fabs(norm_fq_old-norm_fq_n
ew) && step*step_multiplicator < 0.999)
                        step *= step_multiplicator;

                out ("n");
                for (i = 0, fq_new.ptr = fq_new.matrix, fq_old.ptr = fq_old.matr
```

nep_mono.c

```
    ix, mct.ptr = mct.matrix; i < fq_new.size; i++, fq_new.ptr++, fq_old.ptr++, mct.
    ptr++)
                                *fq_new.ptr = step*(*mct.ptr/(1.0+*mct.ptr)) + (1.0-step
    )**fq_old.ptr;

                time (&now);
                out (" time: %lf", difftime (now, start));
        } while (fabs(norm_fq_old-norm_fq_new) > norm_fq_max || step < 0.9);


        goto out;
    out:
        out ("\n-> f(q) ausgeben");
        char *tmp = (char *) &(*(sq_infilename+1));
        char fq_outfilename[255];
        sprintf (fq_outfilename, "f%s", tmp);

        FILE *fq_outfile = fopen (fq_outfilename, "w");
        for (i = 0, fq_new.ptr = fq_new.matrix, q.ptr = q.matrix; i < fq_new.siz
    e; i++, fq_new.ptr++, q.ptr++)
                fprintf (fq_outfile, "\n%.32lf %.32lf", *q.ptr, *fq_new.ptr);
        fclose (fq_outfile);


        goto clean;
    clean:
        out ("\n-> Cleanup");

        cleanup_matrix (sq);
        cleanup_matrix (cq);

        cleanup_matrix (sq_bessel);
        cleanup_matrix (cq_bessel);
        cleanup_matrix (sq_deriv);
        cleanup_matrix (cq_deriv);
        cleanup_matrix (q_bessel);

        cleanup_matrix (q);
        cleanup_matrix (fq_new);
        cleanup_matrix (fq_old);
        cleanup_matrix (k);
        cleanup_matrix (p);
        cleanup_matrix (mct);

        out ("\nok.");
        if (norm_fq_new < 1.0) {
                exit (-1);
        }
        return (0);
    }
```

Abbildungsverzeichnis

List of Algorithms

Literaturverzeichnis

[Bra06] J. Brader. *Structure factor for monodisperse hard spheres with MHNC*. 2006.

[CS03] S.-H. Chong and F. Sciortino. Structural relaxation in a supercooled molecular liquid. *Europhys. Lett*, 64(2):197–203, 2003.

[For83] D. Forster. *Hydrodynamic Fluctuations, Broken Symmetry, and Correlation Functions*. The Benjamin/Cummings Publishing Company, Inc., W.A. Benjamin, Inc., Advanced Book Program, Reading, Massachusetts 01867, USA, 1983.

[Fuc06] M. Fuchs. *Private Comm.* 2006.

[GFV04] F. Sciortino P. Tartaglia G. Foffi, W. Götze and T. Voigtmann. α-relaxation processes in binary hard-sphere mixtures. *Phys. Rev. E*, 69, 2004.

[Gil79] M. J. Gillan. A new method of solving the liquid structure integral equations. *Mol. Phys.*, 38(6):1781–1794, 1979.

[GR80] I. S. Gradshteyn and I. M. Ryzhik. *Table of Integrals, Series, and Products*. Academic Press, INC., 1250 Sixth Avenue, San Diege, California 92101, 1980.

[GSL05] *GNU Scientific Library*. www.gsl.org, 2005.

[GV03] W. Götze and T. Voigtmann. Effect of composition on the structural relaxation of a binary mixtures. *Phys. Rev. E*, 67, 2003.

[Göt89] W. Götze. *liquids, freezing and glass transition (Part 1)*. North Holland, Elsevier Science Publishers B.V., P.O. Box 211, 1000 AE Amsterdam, The Netherlands, 1989.

[HKM05] K. Zahn H. König, R. Hund and G. Maret. Experimental realization of a model glass former in 2d. *Europhys. J. E*, 18(3):287–293, 2005.

[HM86] J.-P. Hansen and I.R. McDonald. *Theory of simple liquids (2nd edition)*. Academic Press, 32 Jamestown Road, London NW1 7BY, UK, 1986.

[Hun01] R. Hund. *Videomikroskopische Untersuchung des 2D Glasüberganges magnetischer Kolloide*. Diplomarbeit, Institut für Physik Universität Konstanz, 2001.

[JLBL89] W. Götze J. L. Barrat and A. Latz. *J. Phys.*, page 7163, 1989.

[Lad68] F. Lado. *J. Chem. Phys.*, 49(7):3092–3096, 1968.

[Lad73] F. Lado. Perturbation correction for the free energy and structure of simple fluids. *Phys. Rev. A*, 8(5):2548–2552, 1973.

[Lad78] F. Lado. Hypernetted-chain solutions for the two-dimensional classical electron gas. *Phys. Rev. B*, 17(7):2827–3832, 1978.

[Leb63] J. L. Lebowitz. Exact solution of generalized percus-yevick equation for a mixture of hard spheres. *Phys. Rev.*, 133(4a):895–899, 1963.

[MB87] J. L. Colot M. Baus. *Phy. Rev. A*, 36(8):3912–3925, 1987.

[MBvG03] U. Herz M. Brunner, C. Bechinger and H.H. von Grünberg. Measuring the equation of state of a hard-disc fluid. *EPL*, 63(6):791–797, 2003.

[Olv74] F. W. J. Olver. *Asymptotics and Special Functions*. Academic Press, INC., 1250 Sixth Avenue, San Diego, California 92101, 1974.

[RS05] M. Ricker and R. Schilling. Microscopic theory of glassy dynamics and glass transition for molecular crystals. *Phys. Rev. E*, 72, 2005.

[Sch94] R. Schilling. *Mode Coupling Approach to the Glass Transition*. Springer, www.springeronline.com, 1994.

[SH70] W.R. Smith and D. Henderson. Analytical representation of the percus-yevick hard-sphere radial distribution function. *Molec. Phys.*, 19:411–415, 1970.

[Spe00] M. Sperl. *Glass Transition in Colloids with Attractive Interaction*. Diplomarbeit, 2000.

[Spe06] M. Sperl. *Private Comm.* 2006.

[SS97] R. Schilling and T. Scheidsteger. Mode coupling approach to the ideal glass transition of molecular liquids: Linear molecules. *Phys. Rev. E*, 56(3):2932–2949, 1997.

[SS98] T. Scheidsteger and R. Schilling. Glass transition for dipolar hard spheres: a mode-coupling approach. *Philosophical Magazine B*, 77(2):305–311, 1998.

[SSB94] P. Nielaba S. Sengupta, D. Marx and K. Binder. *Phys. Rev. E*, 49(2):1468–1477, 1994.

[Ste] G. Stell. *Physica*, 29(517).

[TTL01] A. Latz T. Theenhaus, R. Schilling and M. Letz. Microscopic dynamics of molecular liquids and glasses: Role of orientations and translation-rotation coupling. *Phys. Rev. E*, 64(5), 2001.

[UBS84] W. Götze U. Bengtzelius and A. Sjörlander. *J. Phys. C: Solid State Phys.*, 17:5915, 1984.

[Voi03] T. Voigtmann. *Mode Coupling Theory of the Glass Transition in Binary Mixtures*. Dissertation, Institut für Physik der Technischen Universität München, 2003.

[Wer63] M. S. Wertheim. Exact solution of the percus-yevick integral equation for hard spheres. *Phys. Rev. Lett*, 10(8):321–323, 1963.

[WHP92] W. T. Vetterling B. P. Flannery W. H. Press, S. A. Teuklosky. *Numerical Recipes in C - The Art of Scientific Computing (Second Edition)*. Cambridge University Press, 40 West 20th Street, New York, NY 10011-4211, USA, 1992.

[Wut05] J. Wuttke. *lmfit*. sourceforge, www.sourceforge.net/lmfit, 2005.

[WvM01] S. R. Williams and W. van Megen. *Phys. Rev. E*, 64, 2001.