Jamming und glasartige Dynamik von Ellipsoiden

Bachelorarbeit aus der Materialphysik

Vorgelegt von
Moritz Villmow
10.01.2018

Institut für Theoretische Physik I
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Prof. Dr. M. Schmiedeberg

Eigenständigkeitserklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer, als von mir angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 9.1.2018

Inhaltsverzeichnis

1 Einleitung

Das Wort Jamming stammt vom englischen Wort “to jam”was “sich stauen” oder auch “blockieren” bedeutet. In der Umwelt tritt Jamming bzw. jam in vielen Bereichen auf, beispielsweise im Verkehrsstau (engl. traffic jam) oder bei Marmelade (engl. jam). Die Gemeinsamkeit liegt hier immer in der eigentlichen Bewegung, die durch das System selbst blockiert wird und sich das System somit in einem starren Zustand befindet. Wie, um beim obigen Beispiel zu bleiben, die Autos, die sich gegenseitig blockieren und somit einen Stau bilden.
In der Materialphysik beschreibt Jamming ebenfalls den Verlust der Dynamik, also den Übergang von einer flüssigen oder fließenden Phase in einen starren, festen Zustand. Dies kann beispielsweise in einem Granulat, einer molekularen Flüssigkeit oder einem Schaum auftreten [1]. In dieser Arbeit wird ein theoretisches System von Kugeln betrachtet um den kritischen Jammingpunkt zu untersuchen. Für viele reale Materialien, z. B. Granulate wie Sand, sind Kugeln jedoch eine sehr idealisierte Näherung. Als erste Verallgemeinerung von Kugeln wird das Jammingverhalten von rotationssymetrischen Ellipsoiden untersucht. Außerdem wird betrachtet, wie sich der Einfluss einer geringen Temperatur auf den Jammingpunkt sowohl bei Kugeln, als auch bei Ellipsoiden auswirkt.

Abbildungsverzeichnis

2 Grundlagen

2.1 Jamming

Jamming bezeichnet einen Phasenübergang eines Systems von einer flüssigen Phase in eine statische, geblockte Phase, der durch Veränderung bestimmter Kontrollparameter auftritt [1]. Bei diesem Übergang verlangsamt sich die Dynamik des Systems erheblich, während sich die Viskosität und Relaxationszeit stark erhöhen. Es entsteht eine starre Phase, effektiv ein Feststoff.
Die andere gewöhnliche Art eines Übergangs von einer flüssigen Phase zu einem Feststoff ist die Kristallisation. Im Gegensatz dazu ist beim Jamming jedoch sowohl die flüssige als auch die feste Phase ungeordnet und das System befindet sich nicht im thermodynamischen Gleichgewicht [2]. Vielmehr ist das System in einem Bereich des Phasenraums "gefangen" und hat keine Möglichkeit das globale Energieminimum zu erreichen. Somit lässt sich nicht eindeutig sagen, ob das System dauerhaft in dieser gejammten Phase bleibt oder sich wieder ohne äußere Einwirkung löst und wieder eine Möglichkeit hat, ein globales Energieminimum zu erreichen. Also ob eine endliche Relaxationszeit existiert. Aus diesem Grund wird bei der Definition von Jamming von einer Phase mit einer Relaxationszeit oberhalb einer experimentellen Zeitskala gesprochen [1].
Der Jamming-Übergang hat somit eine große Ähnlichkeit zum Glasübergang. Hier wird die Temperatur als Kontrollparameter erniedrigt, bis die Dynamik des Systems zum Erliegen kommt und eine feste Phase entsteht. Auch hier ist diese ungeordnet. Es wird diskutiert, ob es sich beim Glasübergang und Jamming um das selbe Phänomen handelt und sogar ein vereinheitlichtes Phasendiagramm vorgeschlagen [3]. Dieses ist jedoch umstritten. So wird von Ikeda, Berthier und Sollich [4] etwa gezeigt, dass der Glasübergang für Temperaturen, die gegen null extrapoliert werden, der Jammingpunkt nicht erreicht wird.
In dieser Arbeit wird als Kontrollparameter die Packungsdichte Φ, also das Verhältnis des Volumen der Teilchen zum Gesamtvolumen, verwendet (Glei-

chung 1).

$$\Phi = \frac{V_{Teilchen}}{V_{total}} \tag{1}$$

Es wird zunächst athermisches Jamming untersucht. Anschließend wird eine niedrige Temperatur durch zufällige Teilchenbewegungen simuliert.

2.2 Packungsdichte als Kontrollparameter

Oberhalb einer kritischen Jamming-Packungsdichte wird die Bewegung der Teilchen durch die anderen Teilchen eingeschränkt. Es liegen sozusagen immer Teilchen im Weg. Durch die größtenteils blockierten Bewegungen kann das System keine Kristallstruktur bilden.
Hierbei spielt der Begriff des "random closest packing"(RCP) eine Rolle. So gibt es ähnlich zum Kristallgitter auch für ungeordnete zufällige Teilchen eine größtmögliche Packungsdichte. Diese liegt unterhalb der dichtesten möglichen Packungsdichte. Kugeln lassen sich z. B. am dichtesten in einem kubisch flächenzentrierten (engl. face centered cubic / fcc) oder hexagonal dichtest gepackten Gitter (engl. hexagonal close-packed / hcp) packen. Hier liegt die Packungsdichte bei $\frac{\pi}{3\sqrt{2}}$, also $\approx 0,74$. Der RCP-Wert liegt hingegen nur bei $\approx 0,64$. Dieser Wert entspricht der kritischen Packungsdichte für den athermischen Jamming-Übergang in einem einfachen System, das heißt für reibungslose Kugeln mit einem abstoßenden Potential bei Überlappung und keinen sonstigen Wechselwirkungen [5]. Ein solches System wird in Abschnitt 3 simuliert. Der Jamming-Phasenübergang tritt also dann auf, wenn die Packungsdichte des Systems den RCP-Wert überschreitet. Das System ist somit dichtest gepackt, ohne einen geordneten Zustand zu erreichen.

2.3 Ellipsoide

In vielen Bereichen reicht es nicht aus nur Kugeln zu betrachten, da auch Teilchen mit komplexeren Formen vorkommen. Die einfachste Verallgemeinerung der Kugeln stellen die Ellipsoide dar, insbesondere die Rotationsellipsoide (engl. spheroids). Ein Ellipsoid ist allgemein durch seine Halbachsen a, b und c gegeben, siehe Gleichung 2. Wenn a = b = c liegt der Spezial-

fall der Kugel vor. Hierbei entspricht die Länge der Halbachsen dem Radius. Im Falle von Rotationsellipsoiden gilt $a = b \neq c$. Hierbei wird noch zwischen $c < a = b$ (engl. oblate) und $c > a = b$ (engl. prolate) unterschieden. Für solche Rotationsellipsoide steigt der RCP-Wert und damit die kritische Packungsdichte für den Phasenübergang bis zu einem gewissen Verhältnis der Achsen zueinander an (siehe Abbildung 1. Von Donev u. a. wird als Erklärung vorgeschlagen, dass durch die Rotation zusätzlichen Freiheitsgrade hinzukommen [6]. Dadurch braucht ein Ellipsoid mehr Nachbarn um diese alle zu blockieren und das Teilchen im lokalen Energieminimum zu halten. Das wird durch eine erhöhte Packungsdichte erreicht. Ab einem Achsenverhältnis von $\approx 1,5$ sinkt die kritische Packungsdichte wieder.

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1 \tag{2}$$

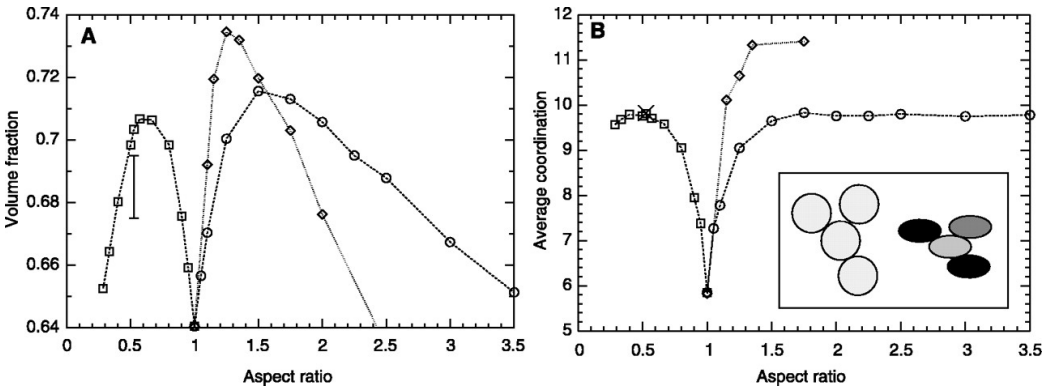


Abbildung 1: Kritische Jammingpackungsdichte (A) und Anzahl der Kontaktpunkte pro Teilchen (B) in Abhängigkeit des Achsenverhältnis der Rotationsellipsoide [6]

.

2.4 Thermisches Jamming

Für das thermische Jamming bei niedrigen Temperaturen wird ein vereinfachtes Modell benutzt [7]. Anstatt alle Teilchenfluktuationen zu simulieren,

beispielsweise über Brownsche Bewegung oder mithilfe einer Monte-Carlo-Simulation werden nur Bewegungen ausgeführt, die tatsächlich eine Energiebarriere überqueren. Alle Fluktuationen innerhalb eines Potentialtals werden vernachlässigt, da sich das angestrebte Energieminimum des Teilchens nicht ändern würde. Somit ist für eine stabiles System, also nach Erreichen des lokalen bzw. globalen Energieminimums, kein anderes Ergebnis zu erwarten. Für das in Abschnitt 2.2 angesprochene System von reibungslosen, repulsiven Kugeln, diesmal allerdings mit einer Temperatur $T > 0$, liegt die kritische Packungsdichte beim thermischen Jamming unterhalb der, des athermischen Jammings. Für $p \leq 10^{-4}$ wurde eine Packungsdichte zwischen 0,53 und 0,555 gefunden [7], wobei p der Wahrscheinlichkeit entspricht, mit welcher ein Teilchen eine Bewegung über eine Energiebarriere ausführt. In dieser Arbeit wird ein solches System ebenfalls simuliert und anschließend untersucht, wie sich der Jammingpunkt für Ellipsoide im Vergleich zu Kugeln verändert.

3 Simulation

Im Folgenden wird der Aufbau einer Simulation beschrieben um die kritische Packungsdichte eines Jamming-Phasenübergangs für Kugeln und Rotationsellipsoide zu ermitteln. Hierzu werden N Ellipsoide für verschiedene Packungsdichten Φ in einer entsprechend großen Box erstellt. Für jedes Teilchen werden zufällige Koordinaten innerhalb der Box, sowie eine zufällige Rotation, gewählt. Anschließend wird die Energie des Systems mithilfe der Methode der konjugierten Gradienten minimiert. Es werden also keine echten Kräfte oder Drehmomente simuliert, sondern reibunglose Teilchen angenommen, die nur über ein abstoßendes Potential interagieren (siehe Abschnitt 3.2 und Gleichung 6). Die Minimierung endet, wenn die Gesamtenergie des Systems null wird oder sich durch wiederholte Schritte nicht mehr verändert.

3.1 Generieren der Ellipsoide

Als Erstes werden die Rotationsellipsoide erstellt. Die Durchmesser in Richtung der Halbachsen a und b werden als 1,0 gewählt. Der Durchmesser in Richtung von Halbachse c wird entsprechend des gewünschten Achsenverhältnisses angepasst. Simuliert wird im Bereich $2c = 1$ bis $2c = 1.5$ in Schritten von 0.1. Der erste Fall entspricht hierbei Kugeln. Die Position und Rotation eines Ellipsoids wird über fünf Koordinaten festgelegt. Für die drei Ortskoordinaten x, y und z werden Zufallswerte im Intervall $-L/2$ bis $L/2$ gezogen. Hierbei entspricht L der Länge der Box, welche über Gleichung 3 ermittelt wird. Hierzu wird das Volumen aller Teilchen benötigt, welches sich mit der Volumenformel für Ellipsoide berechnen lässt (Gleichung 4).

$$L^3 = \frac{V_{Teilchen}}{\Phi} \tag{3}$$

$$V_{Teilchen} = \sum_{i=1}^{N} \frac{1}{3} \cdot \pi \cdot a \cdot b \cdot c \tag{4}$$

Die Rotation wird durch zwei Winkel festgelegt. Einem Azimutwinkel ϕ zwischen 0 und $2 \cdot \pi$ und einem Polarwinkel θ zwischen 0 und π. Der Polarwinkel

wird dabei nach Gleichung 5 so gezogen, dass die Rotationen der Ellipsoide gleichverteilt sind, wobei k einer Zufallszahl zwischen 0 und 1 entspricht.

$$\arccos\left(1 - 2 \cdot k\right) \tag{5}$$

3.2 Potential

$$V(r_{ij}, \sigma_{ij}) = \begin{cases} (1 - \frac{r_{ij}}{\sigma_{ij}})^2 & \text{für } r_{ij} < \sigma_{ij} \\ 0 & \text{für } r_{ij} \geq \sigma_{ij} \end{cases} \tag{6}$$

In der Simulation wird ein harmonisches Potential verwendet (siehe Gleichung 6). Es wird paarweise zwischen zwei Teilchen i und j berechnet. r_{ij} ist der Abstand der beiden Ellipsoide, während σ_{ij} der Summe der Radien der beiden Teilchen entspricht. Im Spezialfall von Kugeln hat σ_{ij} einen Wert, der nur vom Radius der beiden Kugeln abhängt und konstant bezüglich der Position und Rotation der Teilchen ist. Dann hängt das Potential nur von sechs Koordinaten ab, nämlich den Ortskoordinaten beider Teilchen.
Im Falle von allgemeineren Ellipsoiden hängt σ_{ij} jedoch sowohl von der Rotation als auch der Position beider Teilchen ab. Konkret ist σ_{ij} aus σ_i und σ_j zusammengesetzt, wobei diese Werte den Abstand vom Mittelpunkt bis zur Oberfläche des entsprechenden Ellipsoids in Richtung des anderen Teilchens entsprechen.
Insgesamt ist das Potential also 0, wenn sich die Ellipsoide nicht berühren und steigt quadratisch mit der Überlappung an. Um die Gesamtenergie des Systems zu berechnen werden die paarweisen Potentiale addiert.

3.3 Minimierung der Energie

Zur Energieminimierung wird die Methode der konjugierten Gradienten eingesetzt. Hierbei werden alle fünf Koordinaten jedes Ellipsoids verändert um ein lokales Energieminimum zu finden. Hierfür werden periodische Randbedingungen genutzt. Das bedeutet, ein Teilchen welches die vorgegebene Box auf einer Seite verlässt, wird auf der gegenüberliegenden Seite wieder hinzugefügt. Dadurch sind schon für wenige simulierte Teilchen sinnvolle Er-

gebnisse zu erwarten.
Für die Initialrichtung h_1 der Minimierung wird der steilste Abstieg des Potentials gesucht (engl. steepest descent). Dazu wird für jedes Teilchen die Kraft entlang der Koordinatenrichtungen, also die Ableitungen des Potentials bezüglich aller fünf Koordinaten, berechnet. Mit einer Schrittweite von 0,01 werden alle Teilchen nun in die entsprechende Richtung bewegt. Dies geschieht solange, bis in diese Richtung keine Verringerung der Energie mehr erreicht wird. Sollte das Minimum überschritten werden, wird die Schrittweite angepasst, sodass ein möglichst genaues Ergebnis erreicht wird. Allgemein darf Richtung in diesem Zusammenhang nicht nur entlang der Ortskoordinaten gesehen werden. Auch die Rotationswinkel werden verändert.
Für die folgenden Iterationsschritte wird nicht wieder der steilste Abstieg verwendet, da dadurch das in der vorherigen Suchrichtung gefundene Minimum wieder verlassen wird. Stattdessen wird eine zu h_n konjugierte Richtung h_{n+1} ermittelt (engl. conjugate gradient) [8]. Hierzu wird zur Richtung des steilsten Abstiegs noch ein Term hinzugerechnet, welcher von der vorangegangenen Minimierungsrichtung abhängt. Wie diese berechnet wird, ist in Gleichung 7 dargestellt, wobei f_n den Gradienten des Potentials im Iterationsschritt n der Minimierung bezeichnet. Dadurch wird die Minimierung vor allem in der Nähe des Minimums viel schneller, da ein Verfahren mit steilsten Abstiegen oft in einer Art "Zick-Zack-Kurs" auf das Minimum zuläuft, und somit mehr Iterationen braucht.

$$h_{n+1} = f_n + \beta \cdot h_n \tag{7}$$

$$\beta = \frac{(f_n + f_{n+1}) \cdot f_n}{f_n^2}$$

Die Minimierung wird beendet, wenn das Potential 0 ist oder sich nicht mehr signifikant verringert. Aufgrund der Maschinengenauigkeit wird hierfür ein Toleranzbereich von 10^{-16} verwendet.

3.4 Thermisches Jamming

Zur Simulation einer kleinen Temperatur wird eine zufällige Teilchenbewegung implementiert, siehe Abschnitt 2.4. Mit einer kleinen Wahrscheinlichkeit von $p = \frac{1}{N}$ wird ein Teilchen nicht wie in Abschnitt 3.3 beschrieben bewegt. Die Richtung für die Minimierung wird manuell auf null gesetzt. Falls diese Ellipsoide noch mit anderen überlappen wird stattdessen eine zufällige Richtung bestimmt. Sollte diese ein Anstieg im Potential darstellen, werden die Teilchen über das nächste Maximum hinweg bewegt. Dies geschieht nachdem die Energie für alle anderen Teilchen wie beim athermischen Jamming minimiert wurde.
In der nächsten Iteration der Minimierung werden alle zuvor zufällig bewegten Teilchen in Richtung des steilsten Abstieges bewegt, während für alle anderen die Methode der konjugierten Gradienten weiterhin zum Einsatz kommt.

3.5 Ausgabe der Ergebnisse

Als Ergebnis der Simulation wird sowohl das endgültige Potential nach allen Minimierungsdurchläufen, als auch die Anzahl der Kontaktpunkte pro Teilchen gespeichert. Für Letzteres wird eine Toleranz von $10^{-6} \cdot \sigma_{ij}$ verwendet. Dies ist notwendig, da aufgrund der Toleranz in der Energieminimierung auch bei einem Absenken des Potentials auf fast 0 noch Berührungspunkte vorhanden sind. Die Toleranz wird so gewählt, da die Überlappung der Teilchen quadratisch in das Potential einfließt. Somit tragen zwei Teichen unterhalb dieser höchstens in einer Größenordnung 10^{-12} zur Energie bei. Da die Energietoleranz bei 10^{-16} liegt, können so 1000 Teichen simuliert werden.

Für das thermische Jamming wird nach jedem Minimierungsdurchlauf die Anzahl der Ellipsoide berechnet, welche sich überlappen, also mindestens einen Kontaktpunkt haben. Dies wird ins Verhältnis zur Gesamtteilchenzahl gesetzt (8).

$$f_{ov} = \frac{N_{ov}}{N} \tag{8}$$

4 Auswertung

Mit dem in Abschnitt 3 beschriebenen Programm, wird die kritische Packungsdichte für den Jammingphasenübergang gesucht. Hierzu werden 100 Teilchen simuliert. Der Durchmesser in Richtung der Halbachse c und somit das Achsenverhältnis wird von 1,0 bis 1,5 variiert. Für jedes Verhältnis wird nun für verschiedene Packungsdichten simuliert. Hierbei wird ein Bereich um den aus der Literatur bekannten Wert von $\approx 0,64$ abgedeckt. Oberhalb der kritischen Packungsdichte lässt sich die Energie des Systems nicht auf 0 minimieren, wodurch die Teilchen noch Berührungspunkte haben.

4.1 Athermisches Jamming von Kugeln

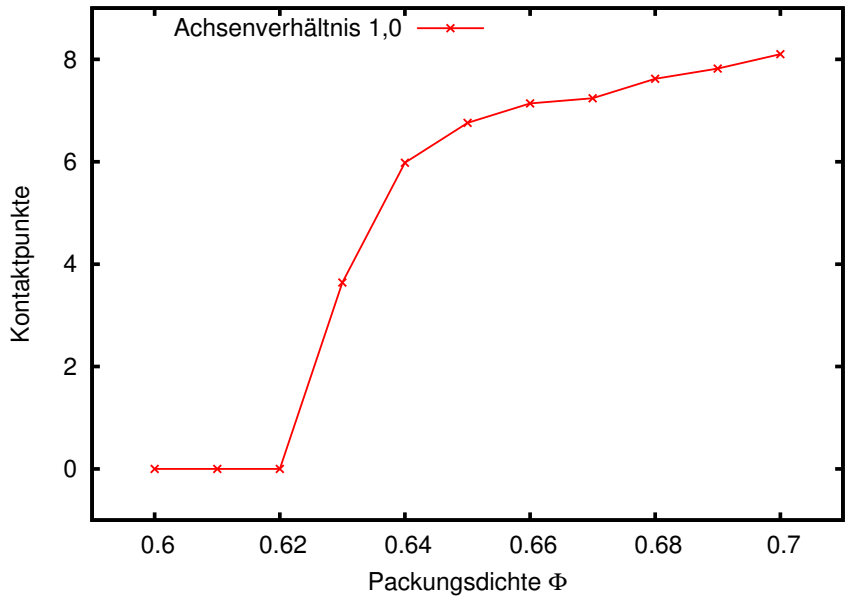


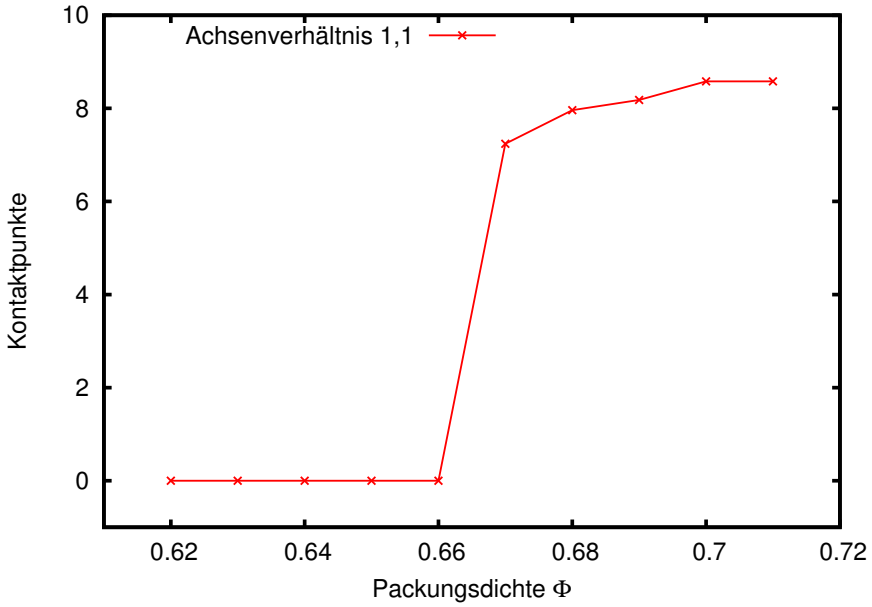
Abbildung 2: Bestimmung der kritischen Packungsdichte für 100 Kugeln

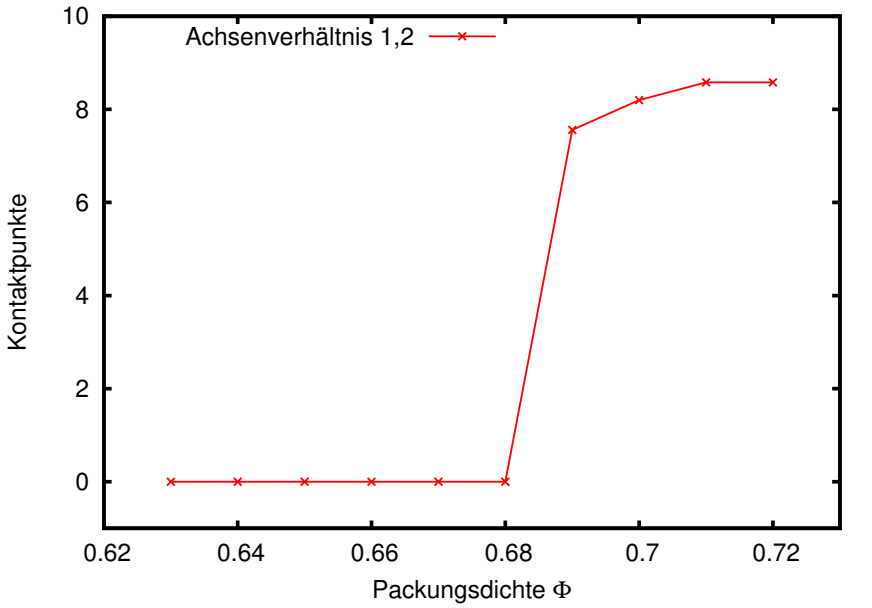
In Abbildung 2 sind diese Kontaktpunkte pro Teilchen über die verschiedenen Packungsdichten aufgetragen. Hier für ein Achsenverhältnis von 1, also

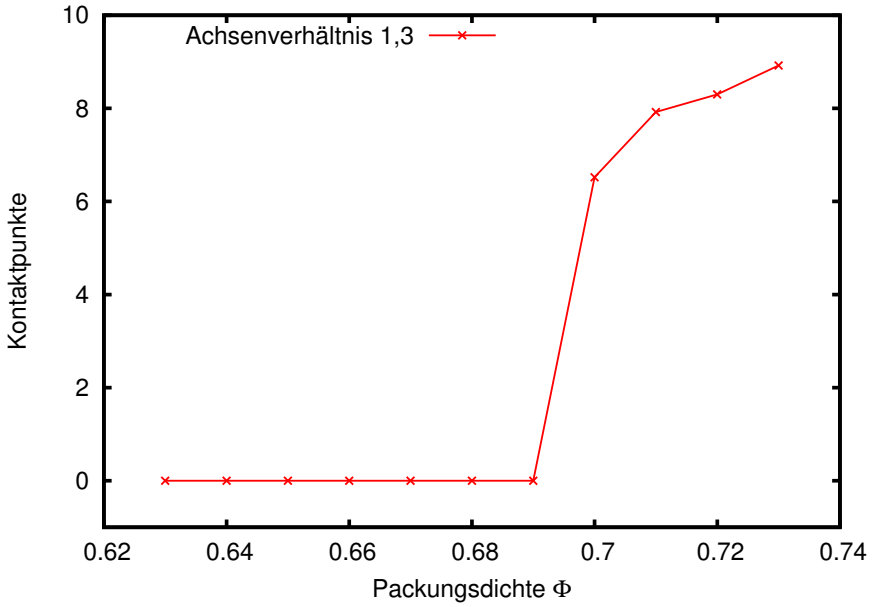
Kugeln. Bis zu einer Packungsdichte von 0,62 ist die Zahl der Kontaktpunkte 0, d. h. das System lässt sich durch Energieminimierung in einen Zustand bringen, in dem sich keine Teilchen mehr überlappen.
Zwischen 0,62 und 0,64 steigt die Zahl Kontakte pro Teilchen sprunghaft auf circa 6 pro Teilchen an. Dies entspricht in drei Dimensionen der Mindestzahl an Nachbarn um alle Bewegungen zu blockieren [2]. Der in 2.1 beschriebene RCP-Wert für Kugeln ist überschritten und die Kugeln können sich nicht mehr ungeordnet ohne Überlappung anordnen. Theoretisch wäre in einer geordneten Struktur eine noch höhere Packungsdichte möglich. Das diese nicht erreicht wird, verdeutlicht noch einmal, dass das System sich in einem ungeordneten und geblockten Zustand befindet. Aufgrund einer durchschnittlichen Nachbarzahl > 6 ist das Teilchen in diesem Bereich "gefangen" und kann das globale Energieminimum nicht mehr erreichen. Somit liegt eine gejammte Phase und keine Kristallisation vor. Der gefundene Wert von 0,63 bis 0,64 stimmt also ungefähr mit dem Literaturwert von $\approx 0,64$ überein. Um einen genaueren Wert zu erhalten kann ein System mit mehr Kugeln simuliert werden, sogenannte "finite size effects", also Effekte die nur aufgrund der Systemgröße auftreten, zu vermeiden.

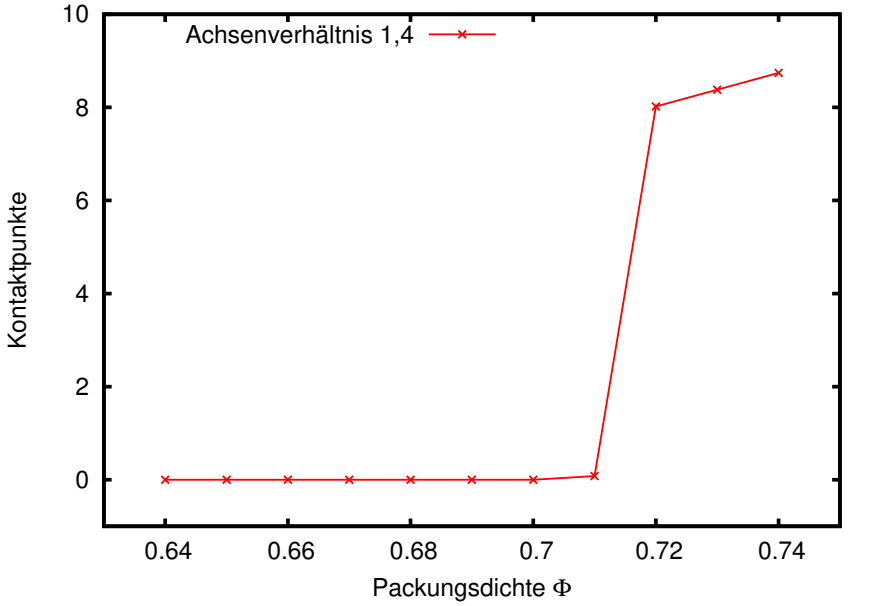
4.2 Athermisches Jamming von Ellipsoiden

Nun wird die Simulation auch für Rotationsellipsoide durchgeführt. Hierbei werden die Achsenverhältnisse 1,1 bis 1,5 in Schritten von 0,1 verwendet. Die Ergebnisse werden wie in Abschnitt 4.1 aufgetragen. Auch hier ist die kritische Packungsdichte gut aus dem sprunghaften Anstieg der Kontaktpunkte abzulesen (siehe Abbildung 3).
In Abbildung 4 wird diese gegen das Achsenverhältnis aufgetragen. So lässt sich erkennen, das sich die kritische Packungsdichte erhöht, je unähnlicher die Ellipsoide zu Kugeln werden. Dies entspricht auch den Ergebnissen in Donev u. a. [6], welche schon in Abschnitt 2.3 aufgeführt wurden. In dieser Veröffentlichung wurde jedoch eine Molekulardynamiksimulation verwendet. Bei dieser bewegen sich die einzelnen Teilchen mit einer gegebenen Geschwindigkeit und kollidieren elastisch mit anderen Teilchen.


Achsenverhältnis 1,1
Kontaktpunkte
Packungsdichte Φ
0
2
4
6
8
10
0.62
0.64
0.66
0.68
0.7
0.72


Achsenverhältnis 1,2
Kontaktpunkte
Packungsdichte Φ
0
2
4
6
8
10
0.62
0.64
0.66
0.68
0.7
0.72


Achsenverhältnis 1,3
Kontaktpunkte
Packungsdichte Φ
0
2
4
6
8
10
0.62
0.64
0.66
0.68
0.7
0.72
0.74


Achsenverhältnis 1,4
Kontaktpunkte
Packungsdichte Φ
0
2
4
6
8
10
0.64
0.66
0.68
0.7
0.72
0.74

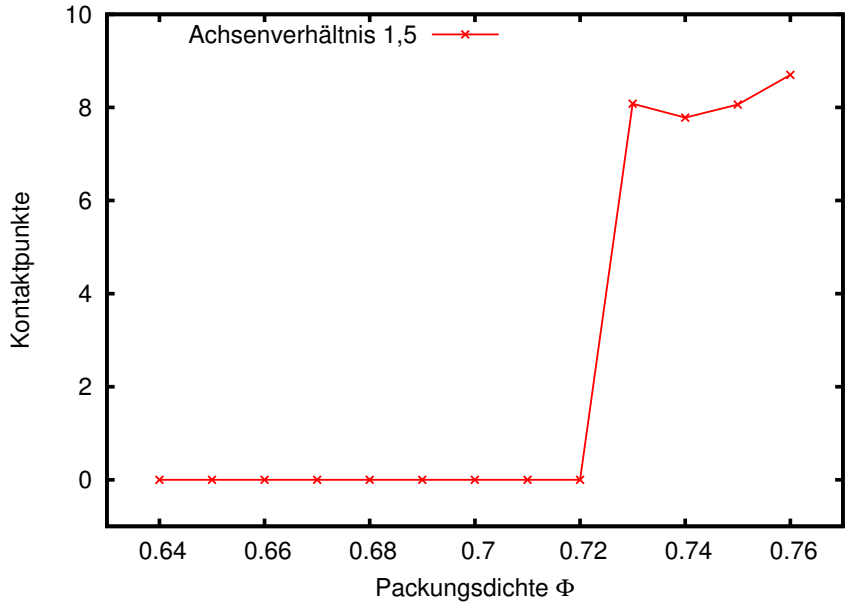


Abbildung 3: Bestimmung der kritischen Packungsdichte für 100 Rotationsellipsoide mit Achsenverhältnis 1,1 bis 1,5

In Abbildung 3 ist außerdem zu sehen, dass auch die Anzahl der durchschnittlichen Kontaktpunkte pro Teilchen in einem System mit kritischer Packungsdichte mit dem Achsenverhältnis ansteigt. Während bei Kugeln noch ≈ 6 Nachbarn pro Teilchen nötig sind um weitere Bewegungen zu blockieren und somit die Energie weiter zu minimieren, sind bei Rotationsellipsoiden mit Achsenverhältnis 1,5 schon mehr als 8 durchschnittliche Nachbarn notwendig. Das liegt an den zusätzlichen Freiheitsgraden der Rotation, die Ellipsoide gegenüber Kugeln besitzen. Dies erklärt auch, warum die kritische Packungsdichte für Ellipsoide höher ist als für Kugeln.
Ein System aus Rotationsellipsoiden ist somit aufgrund der zusätzlichen Freiheitsgrade der Rotation noch in der fließenden Phase, während ein System aus Kugeln bei der gleichen Packungsdichte schon gejammt ist.

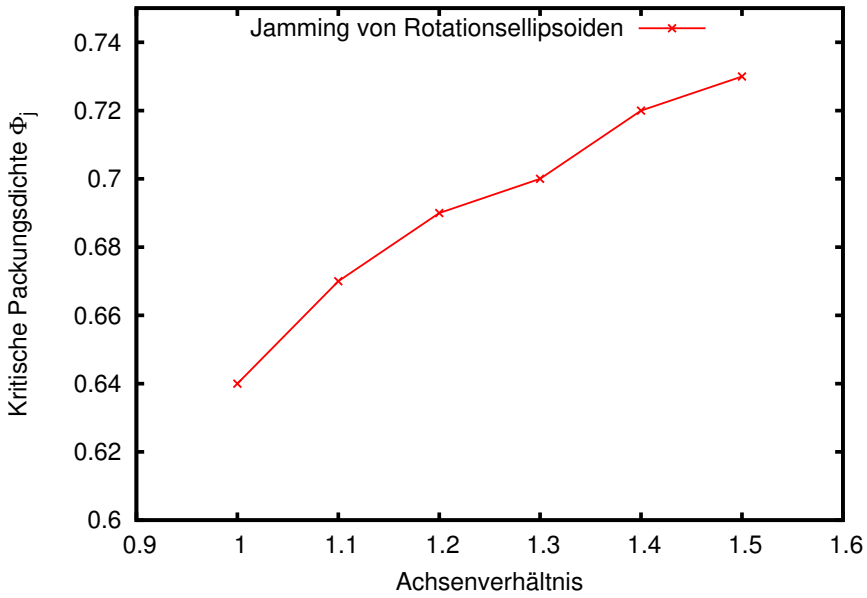


Abbildung 4: Kritische Packungsdichte des athermischen Jamming-Übergangs für Rotationsellipsoide abhängig vom Achsenverhältnis

4.3 Thermisches Jamming von Kugeln und Ellispoiden

Als Kriterium für thermisches Jamming wird, wie beim athermischen Jamming, geschaut, ob es durch Energieminimierung gelingt, alle Überlappungen der Teilchen aufzulösen. Hierfür wird, wie in Abschnitt 3.4 beschrieben, der Anteil der Teilchen, welche noch Kontaktpunkte mit anderen Teilchen haben, an der Gesamtzahl berechnet.
Durch die zufälligen Bewegungen können jedoch auch in Bereichen des Systems wieder Überlappungen auftreten, in welchen diese schon eliminiert wurden und ein globales Energieminimum gefunden wurde. Um dies zu verhindern, werden zum einen nur noch Teilchen bewegt, welche sich noch überlappen, also sich in einem Potential $V > 0$ befinden. Dadurch ist klar zu erkennen, ob ein System den Grundzustand erreicht und eventuell nur durch die zufälligen Bewegungen künstlich am Leben gehalten wurde. Allerdings besteht die Gefahr, dass ein System, das eigentlich einen Jamming-Übergang hat, zufällig in ein globales Minimum kommt und dieses aufgrund der nun fehlenden Temperatur nicht mehr verlässt. Dies muss bei der Auswertung der Daten berücksichtigt werden.
Zum anderen wird das Verhältnis der überlappenden Teilchen f_{ov} für jeden Minimierungsschritt aufgetragen und nicht nur das Endergebnis betrachtet.
Das athermische Jamming wird für 1000 Teilchen simuliert. Somit beträgt die Wahrscheinlichkeit $p = \frac{1}{N}$ für die zufällige Bewegung eines Teilchens 0.1%. Dies ist ein Kompromiss aus einem kleinen System und dadurch einer schnellen Simulation und einer kleinen Wahrscheinlichkeit, bei welcher eine konstante kritische Packungsdichte unabhängig von p gefunden wurde [7].
Der Anteil der überlappenden Teilchen wird über der Zeit für verschiedene Packungsdichten aufgetragen, wobei die Zeit den Minimierungsdurchläufen entspricht. In Abbildung 5 ist zu erkennen, dass für Kugeln bis zu einer Packungsdichte von $\Phi = 0,46$ sich das System vollständig lösen und in einen Grundzustand versetzten lässt. Erst ab einer Packungsdichte von $\Phi = 0,47$ gelingt es nicht in 10000 Minimierungsschritten alle Überlappungen zu beseitigen. Es findet also ein Jamming-Übergang bei einer Packungsdichte wesentlich unterhalb des athermischen Jammings statt. Allerdings ist der Wert

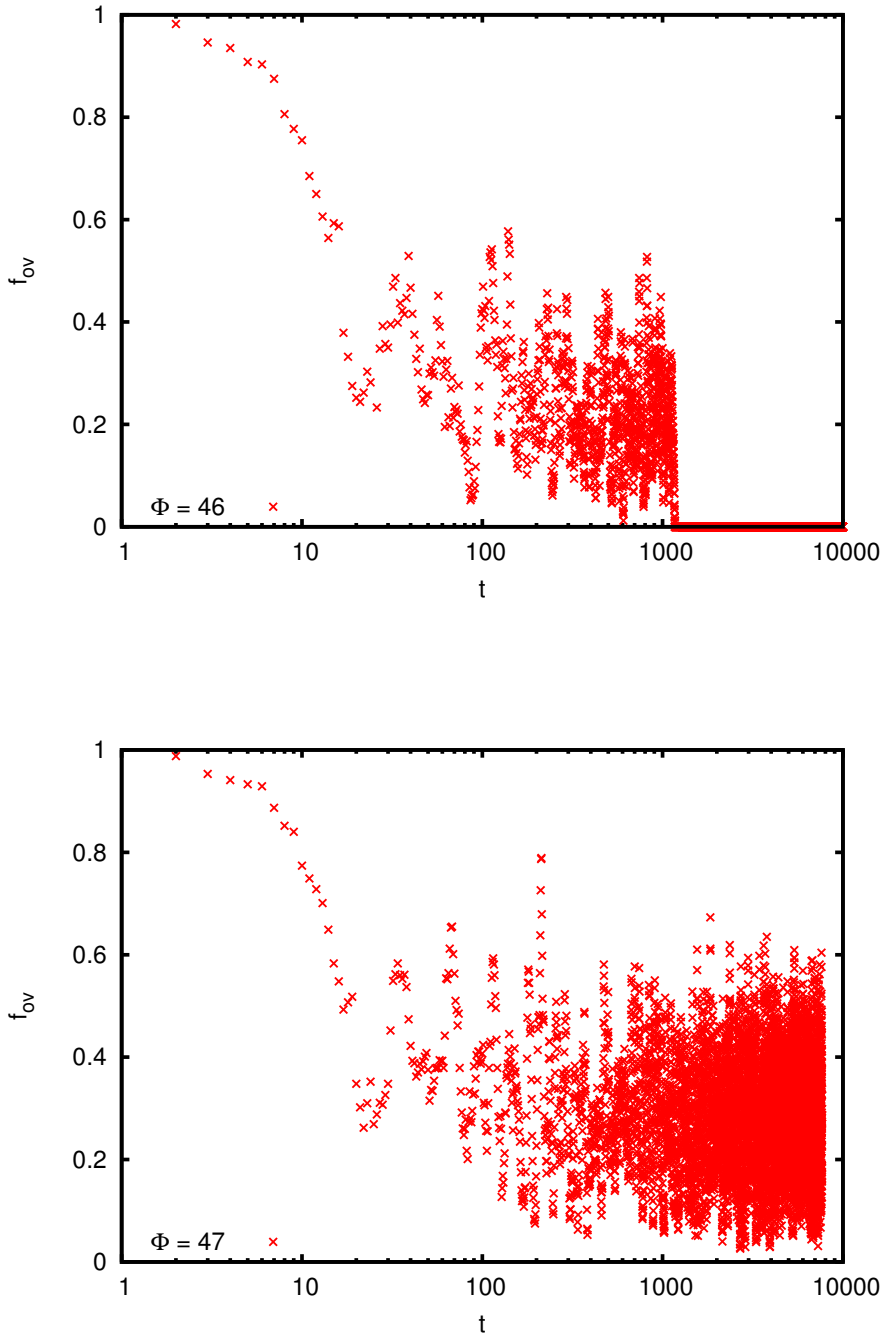


Abbildung 5: Thermischer Jamming-Übergang für Kugeln bei einer Packungsdichte zwischen 0,46 und 0,47

auch niedriger als vergleichbare Ergebnisse von Maiti und Schmiedeberg [7]. Dort wurde für $p = 10^{-3}$ eine kritische Packungsdichte von $\Phi \approx 0,5$ gefunden. Dies kann darauf zurückgeführt werden, dass in dieser Arbeit nicht die Methode aus dem Hauptartikel von Maiti und Schmiedeberg [7] angewandt wird, sondern eine, die in der Veröffentlichung in den Ergänzungen unter Methode C aufgelistet ist. Außerdem werden nur 1000 Teilchen simuliert, sodass "finite size effects" auftreten.

Die selbe Simulation wird nun für Ellipsoide mit einem Achsenverhältnis von 1,5 ausgeführt (siehe Abbildung 6). Die Ergebnisse werden genauso wie bei Kugeln aufgetragen. Hier ist der Übergang von einer vollständigen Minimierung zu einem verbleibenden Potential zwischen den Packungsdichten 0,5 und 0,51 zu beobachten. Der Jammingpunkt liegt also auch beim thermischen Jamming für Ellipsoide bei einer höheren Packungsdichte als für Kugeln. Jedoch liegt der Wert für Ellipsoide nicht soweit oberhalb dem für Kugeln wie beim athermischen Jamming (siehe Abbildung 3).

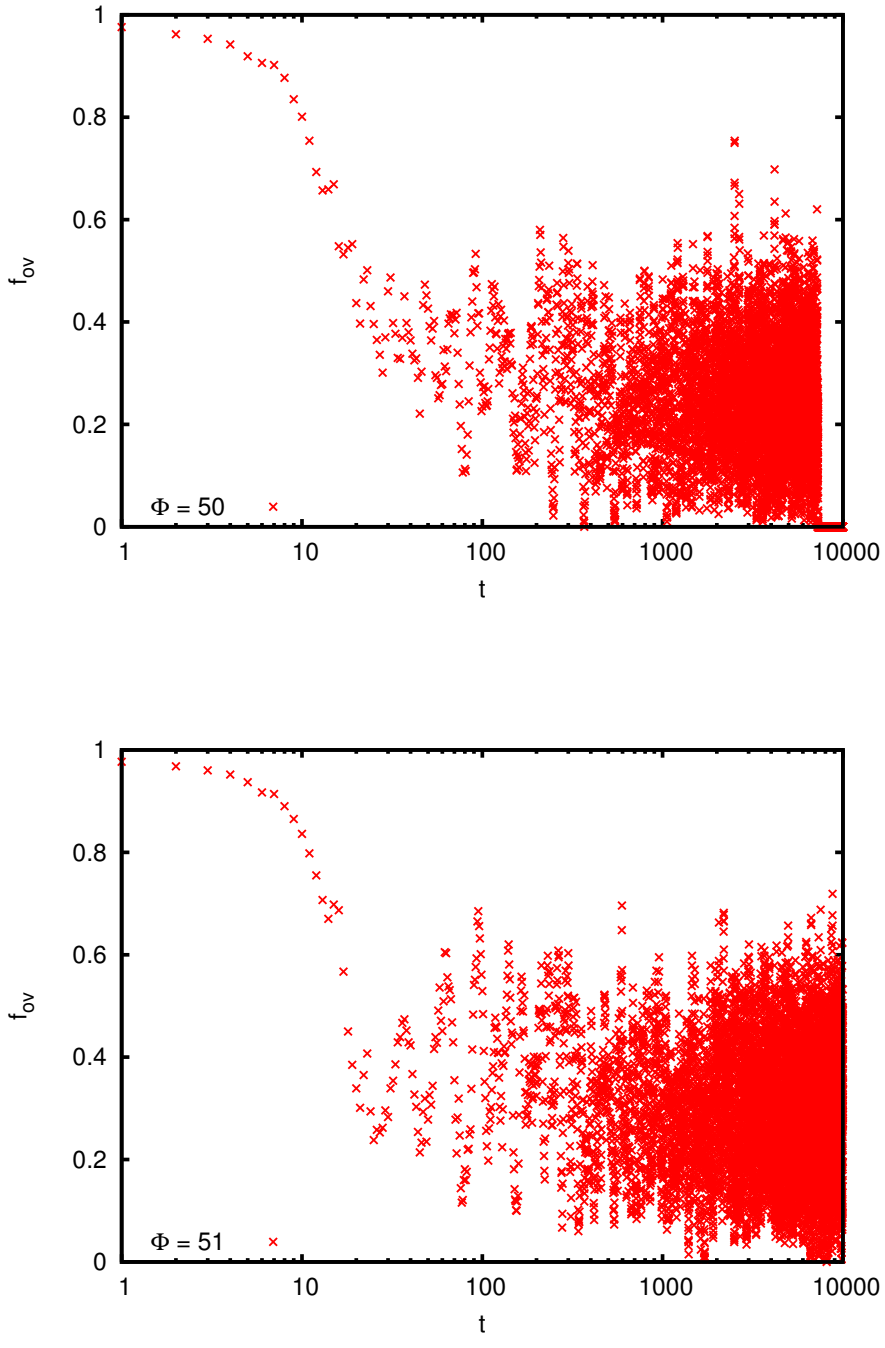


Abbildung 6: Thermischer Jamming-Übergang für Rotationsellipsoide (Achsenverhältnis 1,5) bei einer Packungsdichte zwischen 0,46 und 0,47

5 Fazit

Im Rahmen dieser Arbeit wurde eine Simulation erstellt, mit welcher die kritische Packungsdichte für den Jamming-Übergang für reibungslose Kugeln und Ellipsoide ermittelt werden kann. Mithilfe dieser Simulation wurde zunächst der kritische Jammingpunkt von $\approx 0,64$ für Kugeln bestätigt. Des Weiteren wurde gezeigt, dass sich dieser Punkt für Rotationsellipsoide erhöht, je kugelunähnlicher diese werden, bis zu einem Achsenverhältnis von 1,5.
Außerdem wurde thermisches Jamming bei kleinen Temperaturen untersucht. Hierbei liegt die kritische Packungsdichte für Kugeln weit unter der des athermischen Jammings bei ca. 0,46. Für Rotationsellipsoide lässt sich auch hier ein Ansteigen dieser auf 0,51 feststellen.
Jedoch wurden in dieser Arbeit nur relativ kleine Systeme von 100 Teilchen beim athermischen bzw. 1000 beim thermischen Jamming simuliert. Insbesondere beim thermischen Jamming kann die Simulation von größeren Systemen zu aussagekräftigeren Ergebnissen führen. Unter Anderem kann, ähnlich wie in [7], die Wahrscheinlichkeit p einer zufälligen Bewegung verändert und somit unterschiedliche Temperaturen simuliert werden. Auch bezüglich der Achsenverhältnisse können noch genauere Ergebnisse erzielt werden. So ist etwa eine Simulation von allgemeinen Ellipsoiden im Vergleich zu den hier behandelten Rotationsellipsoiden vorstellbar.

Literatur

[1] Raffaele Pastore. „Jamming transition in thermal and non-thermal systems“. Diss. Universiá degli Studi di Napoli Federico II, 29. Nov. 2011.

[2] Andrea J. Liu und Sidney R. Nagel. „The Jamming Transition and the Marginally Jammed Solid“. In: *Annual Review of Condensed Matter Physics* 1 (Aug. 2010), S. 347–369. DOI: 10.1146/annurev-conmatphys-070909-104045.

[3] Corey S. O'Hern u. a. „Jamming at zero temperature and zero applied stress: The epitome of disorder“. In: *Physical Review E* 68 (17. Juli 2003). DOI: 10.1103/physreve.68.011306.

[4] Atsushi Ikeda, Ludovic Berthier und Peter Sollich. „Unified study of glass and jamming rheology in soft particle systems“. In: *Physical Review Letters* 109.1 (Juli 2012). DOI: 10.1103/physrevlett.109.018301.

[5] L. E. Silbert. „Jamming of frictional spheres and random loose packing“. In: *Soft Matter* 6 (2010), S. 2918. DOI: 10.1039/c001973a.

[6] A. Donev u. a. „Improving the Density of Jammed Disordered Packings Using Ellipsoids“. In: *Science* 303 (13. Feb. 2004), S. 990–993. DOI: 10.1126/science.1093010.

[7] Moumita Maiti und Michael Schmiedeberg. „Ergodicity breaking transition in a glassy soft sphere system at small but non-zero temperatures“. In: (11. Mai 2017). arXiv: 1705.04095v1 [cond-mat.soft].

[8] W. H. Press u. a. *Numerical Recipes. The Art of Scientific Computing.* 2007. Kap. 2.6.7 and 10.7.1. ISBN: ISBN 0-521-88068-8.

6 Danksagung

Ich danke Prof. Dr. Michael Schmiedeberg für die Chance, in seiner Gruppe zu arbeiten und die gute Betreuung während dieser Zeit. Insbesondere bei Problemen mit dem Code. Des Weiteren möchte ich mich bei Moumita Maiti für die Hilfe, besonders zu Beginn der Arbeit, und dem Bereitstellen des Codes für das Jamming von Kugeln bedanken. Außerdem danke ich Sonja und meiner Familie für die Unterstützung während der Arbeit.

7 Anhang

Im Folgenden ist der Programmcode der Simulation aufgelistet. Das Programm ist in C++ geschrieben. *JammingEllipsoids.cpp* beinhaltet die main-Methode. Alle globalen Parameter werden in *Global.h* gespeichert. *input.cpp* dient dem Einlesen der Eingabeparameter der Simulation aus *input.dat*. Dementsprechend werden in *generate_ellipsoid.cpp* die Teilchen Koordinaten erstellt. In *pair_harm* wird das Potential, sowie die Kräfte in alle Richtungen berechnet. Eine Nachbarliste wird in *verlet_list.cpp* erstellt. Die Schleife für die Energieminimierung befindet sich in *minimization.cpp*, wo auch die neue Richtung für die Methode der konjugierten Gradienten berechnet wird. Die eigentlichen Teilchenbewegungen werden allerdings in *linemin.cpp* ausgeführt. In *result.cpp* werden anschließend die Ergebnisse ausgegeben.

Listing 1: JammingEllipsoids.cpp

```
1  #include "Global.h"
2  #include "input.h"
3  #include "pair_harm.h"
4  #include "verlet_list.h"
5  #include "minimization.h"
6  #include "results.h"
7
8  int main() {
9    input myinput;
10   myinput.input_params();
11   minimization conjugate_gradient;
12   results result;
13   pair_harm potential;
14   potential.initForce();
15   result.init();
16   if (verlet == 1) {
17     verlet_list myneigh;
18     myneigh.init();
19     myneigh.vlist();
20     conjugate_gradient.cg();
21     result.ContactPoints();
22   }
23   result.Print();
24 }
```

Listing 2: Global.h

```
#ifndef GLOBAL_H
#define GLOBAL_H
extern double density;
extern int N, no_neigh, D;
extern int verlet, cell;
extern double cutoff, skin, skinsqr;
extern double sigma1, sigma2, sigma3;
extern double ds_start, L;
extern double dtheta, dphi;
extern int runtime;
extern double **x, **xs, **xu;

extern int *inum, *ilist;//Verlet neighbour list variables
extern double **force;
extern double u;
extern double initialX;
extern int* randomParticles;
extern bool* hasContacts; //for thermal
extern int thermal, iter; //

extern double **xi;
#endif
```

Listing 3: input.cpp

```
#include "Global.h"
#include "input.h"
#include "generate_ellipsoid.h"
#include <iostream>
using namespace std;

int N, verlet, cell;
int no_neigh, D;
double **x, **xs;
double sigma1, sigma2, sigma3;
double cutoff, skin, skinsqr, ds_start, L, density;
int thermal, iter;
int runtime;
bool printContacts;
void input::input_params() {
  ifstream infile("input.dat");
  string mystr, configfile, line;

```

```
19    while (getline(infile, line)) {
20      istringstream ss(line);
21      ss >> mystr >> a;
22      if (mystr == "Dimension")D = a;
23      if (mystr == "Particles") N = a;
24      if (mystr == "Diameter_1")sigma1 = a;
25      if (mystr == "Diameter_2")sigma2 = a;
26      if (mystr == "Diameter_3")sigma3 = a;
27      if (mystr == "Cutoff") cutoff = a;
28      if (mystr == "Skin") skin = a;
29      if (mystr == "Density") density = a;
30      if (mystr == "Verlet_list") verlet = a;
31      if (mystr == "Cell_list") cell = a;
32      if (mystr == "Neigh")no_neigh = a;
33      if (mystr == "Initialisation") initial = a;
34      if (mystr == "Runtime") runtime = a;
35      if (mystr == "ds_start")ds_start = a;
36      if (mystr == "Dtheta")dtheta = a;
37      if (mystr == "Dphi")dphi = a;
38      if (mystr == "Thermal")thermal = a;
39    }
40
41    //calculating boxlength
42    double vol;
43    double pi = 3.141592653589793238;
44    vol = (N*pi*sigma1*sigma2*sigma3) / (6.0*density);
45    L = pow(vol, (1.0 / 3.0));
46
47    //allocating coordinates variable x
48    x = new double*[D];
49    xs = new double*[D - 2];
50
51    for (int j = 0; j<D; j++)x[j] = new double[N];
52    for (int j = 0; j<D - 2; j++)xs[j] = new double[N];
53
54    //reading coordinates from a file
55    if (initial == 1) {
56      cout << "what is the name of configfile?" << endl;
57      getline(cin, configfile);
58      ifstream coordinates(configfile);
59      int i = 0;
60      while (coordinates >> a >> b >> c >> e >> f) {
61        x[0][i] = a;
```

```
62        x[1][i] = b;
63        x[2][i] = c;
64        x[3][i] = e;
65        x[4][i] = f;
66        i++;
67      }
68      infile.close();
69    }
70    //loop to generate random coordinates
71    else if (initial == 0) {
72      cout << "generating random configuration" << endl;
73      generate_ellipsoid myellipsoids;
74      myellipsoids.ellipsoids();
75    }
76  }
```

Listing 4: generate_ellipsoid.cpp

```
1  #include "Global.h"
2  #include "generate_ellipsoid.h"
3  #include <stdlib.h>
4  #include "math.h"
5  #include <random>
6
7  std::default_random_engine newGenerator;
8  std::uniform_real_distribution<double> randomNum(0, 1.0);
9  double initialX;
10
11 void generate_ellipsoid::ellipsoids() {
12   double pi = 3.141592653589793238;
13
14   for (int i = 0; i < N; i++) {
15     //coordinates
16     x[0][i] = (1 - 2 * randomNum(newGenerator))*L*.5;
17     x[1][i] = (1 - 2 * randomNum(newGenerator))*L*.5;
18     x[2][i] = (1 - 2 * randomNum(newGenerator))*L*.5;
19     //angles
20     x[3][i] = randomNum(newGenerator)*2.0*pi;
21     x[4][i] = acos(1 - (2 * randomNum(newGenerator)));
22   }
23   initialX = [0][2];
24 }
```

Listing 5: pair_harm.cpp

```
#include "Global.h"
#include "pair_harm.h"
#include "anint.h"
#include "math.h"
#include <fstream>
using namespace std;

double **force;
double u;
void pair_harm::initForce() {
  force = new double*[D];

  for (int i = 0; i < D; i++)
    force[i] = new double[N];
}

void pair_harm::ellipsoid() {
  contacts = 0;
  contacts6 = 0;
  anint myround;
  double Linv = 1.0 / L;
  double xtemp, ytemp, ztemp;
  double phi_i, phi_k, theta_i, theta_k;
  double delx, dely, delz;
  double sigma, sigmai, sigmak, sigmasq;
  double xdir_i, ydir_i, zdir_i, ydir_k, xdir_k, zdir_k;
  double wd, r, rsq, rinv, fpair;
  int i, j, jb, je, k;
  u = 0.0;
  for (i = 0; i < N; i++) {
    force[0][i] = 0.0;
    force[1][i] = 0.0;
    force[2][i] = 0.0;
    force[3][i] = 0.0;
    force[4][i] = 0.0;
  }
  for (i = 0; i < N; i++) {

    xtemp = xu[0][i];
    ytemp = xu[1][i];
    ztemp = xu[2][i];

```

```
43      jb = inum[i];
44      je = inum[i + 1];
45
46      for (j = jb; j < je; j++) {
47        k = ilist[j];
48        if (k < i)continue;
49        delx = xtemp - xu[0][k];
50        delx = delx - L*myround.round(delx*Linv);
51        dely = ytemp - xu[1][k];
52        dely = dely - L*myround.round(dely*Linv);
53        delz = ztemp - xu[2][k];
54        delz = delz - L*myround.round(delz*Linv);
55        rsq = delx*delx + dely*dely + delz*delz;
56        r = sqrt(rsq);
57
58        //compute Rotation for Ellipsoid i
59        phi_i = xu[3][i];
60        theta_i = xu[4][i];
61
62        xdir_i = (cos(phi_i)*cos(theta_i)*delx - sin(phi_i)*
             dely + cos(phi_i)*sin(theta_i)*delz) / r;
63        ydir_i = (cos(theta_i)*sin(phi_i)*delx + cos(phi_i)*
             dely + sin(phi_i)*sin(theta_i)*delz) / r;
64        zdir_i = (-sin(theta_i)*delx + cos(theta_i)*delz) / r
             ;
65
66        double tempi = (xdir_i * xdir_i / sigma1sq) + (ydir_i
              * ydir_i / sigma2sq) + (zdir_i * zdir_i /
             sigma3sq);
67        sigmai = sqrt(1.0 / tempi);
68
69        //compute Rotation for Ellipsoid k
70        phi_k = xu[3][k];
71        theta_k = xu[4][k];
72
73        xdir_k = -(cos(phi_k)*cos(theta_k)*delx - sin(phi_k)*
             dely + cos(phi_k)*sin(theta_k)*delz) / r;
74        ydir_k = -(cos(theta_k)*sin(phi_k)*delx + cos(phi_k)*
             dely + sin(phi_k)*sin(theta_k)*delz) / r;
75        zdir_k = -(-sin(theta_k)*delx + cos(theta_k)*delz) /
             r;
76
77        double tempk = (xdir_k * xdir_k / sigma1sq) + (ydir_k
```

```
        * ydir_k / sigma2sq) + (zdir_k * zdir_k /
        sigma3sq);
78      sigmak = sqrt(1.0 / tempk);
79
80      sigma = sigmai + sigmak;
81      sigmasq = sigma*sigma;
82
83      //compute u and force
84      if (rsq < sigmasq) {
85        wd = 1.0 - r / sigma;
86        rinv = 1.0 / r;
87        fpair = 2.0*wd;
88
89        double rOverSigmasq = r / sigmasq;
90        double overRSigma = rinv / sigma;
91
92        double dx_1OverR = -delx / (r*rsq);
93        double dx_xOverR = rinv + delx*dx_1OverR;
94        double sqrtPow3_i = pow(tempi, -1.5);
95        double sqrtPow3_k = pow(tempk, -1.5);
96
97        //compute force 0
98        double dx_xdir, dx_ydir, dx_zdir;
99
100       //sigma i derivatives
101       dx_xdir = cos(phi_i)*cos(theta_i)*dx_xOverR - sin(
              phi_i)*dx_1OverR*dely + cos(phi_i)*sin(theta_i)*
              dx_1OverR*delz;
102       dx_ydir = cos(theta_i)*sin(phi_i)*dx_xOverR + cos(
              phi_i)*dx_1OverR*dely + sin(phi_i)*sin(theta_i)*
              dx_1OverR*delz;
103       dx_zdir = -sin(theta_i)*dx_xOverR + cos(theta_i)*
              dx_1OverR*delz;
104       double dx_sigma = -(xdir_i*dx_xdir / sigma1sq +
              ydir_i*dx_ydir / sigma2sq + zdir_i *dx_zdir /
              sigma3sq)*sqrtPow3_i;
105
106       //sigma k derivatives
107       dx_xdir = cos(phi_k)*cos(theta_k)*dx_xOverR - sin(
              phi_k)*dely*dx_1OverR + cos(phi_k)*sin(theta_k)*
              delz*dx_1OverR;
108       dx_ydir = cos(theta_k)*sin(phi_k)*dx_xOverR + cos(
              phi_k)*dely*dx_1OverR + sin(phi_k)*sin(theta_k)*
```

```
            delz*dx_1OverR;
109         dx_zdir = -sin(theta_k)*dx_xOverR + cos(theta_k)*
            delz*dx_1OverR;
110         dx_sigma -= (xdir_k*dx_xdir / sigma1sq + ydir_k*
            dx_ydir / sigma2sq + zdir_k*dx_zdir / sigma3sq)*
            sqrtPow3_k;
111         double force0 = fpair*(rOverSigmasq*dx_sigma - delx
            *overRSigma);
112
113         //compute force 1
114         double dy_1OverR = -dely / (r*rsq);
115         double dy_yOverR = rinv + dy_1OverR*dely;
116         double dy_xdir, dy_ydir, dy_zdir;
117
118         //sigma i derivatives
119         dy_xdir = cos(phi_i)*cos(theta_i)*dy_1OverR*delx -
            sin(phi_i)*dy_yOverR + cos(phi_i)*sin(theta_i)*
            dy_1OverR*delz;
120         dy_ydir = cos(theta_i)*sin(phi_i)*dy_1OverR*delx +
            cos(phi_i)*dy_yOverR + sin(phi_i)*sin(theta_i)*
            dy_1OverR*delz;
121         dy_zdir = -sin(theta_i)*dy_1OverR*delx + cos(
            theta_i)*dy_1OverR*delz;
122         double dy_sigma = -(xdir_i*dy_xdir / sigma1sq +
            ydir_i*dy_ydir / sigma2sq + zdir_i*dy_zdir /
            sigma3sq)*sqrtPow3_i;
123         //sigma k derivatives
124         dy_xdir = cos(phi_k)*cos(theta_k)*dy_1OverR*delx -
            sin(phi_k)*dy_yOverR + cos(phi_k)*sin(theta_k)*
            dy_1OverR*delz;
125         dy_ydir = cos(theta_k)*sin(phi_k)*dy_1OverR*delx +
            cos(phi_k)*dy_yOverR + sin(phi_k)*sin(theta_k)*
            dy_1OverR*delz;
126         dy_zdir = -sin(theta_k)*dy_1OverR*delx + cos(
            theta_k)*dy_1OverR*delz;
127         dy_sigma -= (xdir_k*dy_xdir / sigma1sq + ydir_k*
            dy_ydir / sigma2sq + zdir_k*dy_zdir / sigma3sq)*
            sqrtPow3_k;
128         double force1 = fpair*(rOverSigmasq*dy_sigma - dely
            *overRSigma);
129
130         //compute force 2
131         double dz_1OverR = -delz / (r*rsq);
```

```
132         double dz_zOverR = rinv + dz_1OverR*delz;
133         double dz_xdir, dz_ydir, dz_zdir;
134         //sigma i derivatives
135         dz_xdir = cos(phi_i)*cos(theta_i)*dz_1OverR*delx -
               sin(phi_i)*dz_1OverR*dely + cos(phi_i)*sin(
               theta_i)*dz_zOverR;
136         dz_ydir = cos(theta_i)*sin(phi_i)*dz_1OverR*delx +
               cos(phi_i)*dz_1OverR*dely + sin(phi_i)*sin(
               theta_i)*dz_zOverR;
137         dz_zdir = -sin(theta_i)*dz_1OverR*delx + cos(
               theta_i)*dz_zOverR;
138         double dz_sigma = -(xdir_i*dz_xdir / sigma1sq +
               ydir_i*dz_ydir / sigma2sq + zdir_i*dz_zdir /
               sigma3sq)*sqrtPow3_i;
139
140         //sigma k derivatives
141         dz_xdir = cos(phi_k)*cos(theta_k)*dz_1OverR *delx -
                sin(phi_k)*dz_1OverR *dely + cos(phi_k)*sin(
               theta_k)*dz_zOverR;
142         dz_ydir = cos(theta_k)*sin(phi_k)*dz_1OverR *delx +
                cos(phi_k)*dz_1OverR *dely + sin(phi_k)*sin(
               theta_k)*dz_zOverR;
143         dz_zdir = -sin(theta_k)*dz_1OverR*delx + cos(
               theta_k)*dz_zOverR;
144         dz_sigma -= (xdir_k*dz_xdir / sigma1sq + ydir_k*
               dz_ydir / sigma2sq + zdir_k*dz_zdir / sigma3sq)
               * sqrtPow3_k;
145         double force2 = fpair*(rOverSigmasq*dz_sigma - delz
               *overRSigma);
146
147         //compute force 3
148         double dphi_xdir = (-sin(phi_i)*cos(theta_i)*delx -
                cos(phi_i)*dely - sin(phi_i)*sin(theta_i)*delz)
                / r;
149         double dphi_ydir = (cos(theta_i)*cos(phi_i)*delx -
               sin(phi_i)*dely + cos(phi_i)*sin(theta_i)*delz)
               / r;
150         double force3_i = -fpair*sqrtPow3_i*(xdir_i*
               dphi_xdir / sigma1sq + ydir_i*dphi_ydir /
               sigma2sq);
151
152         dphi_xdir = (-sin(phi_k)*cos(theta_k)*delx - cos(
               phi_k)*dely - sin(phi_k)*sin(theta_k)*delz) / r;
```

```
153             dphi_ydir = (cos(theta_k)*cos(phi_k)*delx - sin(
                    phi_k)*dely + cos(phi_k)*sin(theta_k)*delz) / r;
154             double force3_k = -fpair*sqrtPow3_k*(xdir_k*
                    dphi_xdir / sigma1sq + ydir_k*dphi_ydir /
                    sigma2sq);
155
156             //compute force 4
157             double dtheta_xdir = (-cos(phi_i)*sin(theta_i)*delx
                     + cos(phi_i)*cos(theta_i)*delz) / r;
158             double dtheta_ydir = (-sin(theta_i)*sin(phi_i)*delx
                     + sin(phi_i)*cos(theta_i)*delz) / r;
159             double dtheta_zdir = (-cos(theta_i)*delx - sin(
                    theta_i)*delz) / r;
160             double force4_i = -fpair*sqrtPow3_i*(xdir_i*
                    dtheta_xdir / sigma1sq + ydir_i*dtheta_ydir /
                    sigma2sq + zdir_i*dtheta_zdir / sigma3sq);
161
162             dtheta_xdir = (-cos(phi_k)*sin(theta_k)*delx + cos(
                    phi_k)*cos(theta_k)*delz) / r;
163             dtheta_ydir = (-sin(theta_k)*sin(phi_k)*delx + sin(
                    phi_k)*cos(theta_k)*delz) / r;
164             dtheta_zdir = (-cos(theta_k)*delx - sin(theta_k)*
                    delz) / r;
165             double force4_k = -fpair*sqrtPow3_k*(xdir_k*
                    dtheta_xdir / sigma1sq + ydir_k*dtheta_ydir /
                    sigma2sq + zdir_k*dtheta_zdir / sigma3sq);
166
167             force[0][i] -= force0;
168             force[1][i] -= force1;
169             force[2][i] -= force2;
170             force[0][k] += force0;
171             force[1][k] += force1;
172             force[2][k] += force2;
173             force[3][i] -= force3_i;
174             force[4][i] -= force4_i;
175             force[3][k] += force3_k;
176             force[4][k] += force4_k;
177             u += wd*wd;
178           }
179         }
180       }
181     }
```

Listing 6: verlet_list.cpp

```
#include "stdafx.h"
#include "Global.h"
#include "math.h"
#include "verlet_list.h"
#include "anint.h"
#include <iostream>
using namespace std;

int *inum;
int *ilist;
void verlet_list::init() {
  inum = new int[N + 1];
   ilist = new int[N*no_neigh];
}
//making the neigborlist
void verlet_list::vlist() {
  double dx, dy, dz, dr;
  double x0, x1, x2;
  double rnc;
  int i, j, k;
  rnc = (cutoff + skin)*sigma1;
  k = 0;
  inum[0] = 0;
  anint myround;
  for (i = 0; i<N; i++) {
    x0 = x[0][i];
    x1 = x[1][i];
    x2 = x[2][i];
    for (j = 0; j<N; j++) {
      if (i == j)continue;
      dx = x0 - x[0][j];
      dx = dx - L*myround.round(dx / L);
      dy = x1 - x[1][j];
      dy = dy - L*myround.round(dy / L);
      dz = x2 - x[2][j];
      dz = dz - L*myround.round(dz / L);
      dr = sqrt(dx*dx + dy*dy + dz*dz);
      if (dr <= rnc) {
        ilist[k] = j;
        k++;
      }
    }
```

```
43      inum[i + 1] = k;
44    }
45    for (i = 0; i<N; i++)
46      for (j = 0; j<D - 2; j++)xs[j][i] = x[j][i];
47  }
```

Listing 7: minimization.cpp

```
1  #include "Global.h"
2  #include "pair_harm.h"
3  #include "minimization.h"
4  #include "linemin.h"
5  #include "math.h"
6  #include <iostream>
7  #include <fstream>
8  #include <random>
9  #include <stdlib.h>
10 using namespace std;
11 default_random_engine generatorMin;
12 uniform_real_distribution<double> probability(0, 1.0);
13
14 int *randomParticles;
15 double **xi, **xu;
16 double dtheta, dphi;
17
18 void minimization::cg() {
19   double ftol = 1.0e-12;
20   double eps = 1.0e-10;
21   double dutol = 1.0e-08;
22   double grad, umag, gg, dgg, gam;
23   double **g, **h;
24   int i, j;
25   pair_harm potential;
26   g = new double*[D];
27   h = new double*[D];
28   xi = new double*[D];
29   xu = new double*[D];
30   randomParticles = new int[30];
31   randomParticles[0] = -1;
32   for (i = 0; i < D; i++) {
33     g[i] = new double[N];
34     h[i] = new double[N];
35   }
36
```

```
37    for (i = 0; i < D; i++) {
38      xi[i] = new double[N];
39      xu[i] = new double[N];
40    }
41
42    for (i = 0; i < N; i++)
43      for (j = 0; j < D; j++)
44        xu[j][i] = x[j][i];
45    potential.ellipsoid();
46    printf("u=%lf\n", u);
47
48    //set gradient to be initial direction
49    grad = 0.0;
50    umag = 0.0;
51
52    for (i = 0; i < N; i++) {
53      for (j = 0; j < D; j++) {
54        grad += force[j][i] * force[j][i];
55        xi[j][i] = -force[j][i];
56        g[j][i] = -xi[j][i];
57        h[j][i] = g[j][i];
58        xi[j][i] = h[j][i];
59        umag += xi[j][i] * xi[j][i];
60      }
61    }
62
63    umag = sqrt(umag);
64    for (i = 0; i < N; i++) {
65      for (j = 0; j < D; j++) {
66        xi[j][i] = xi[j][i] / umag;
67      }
68    }
69    iter = 0;
70    linemin min;
71    double ukeep;
72    ofstream myfile;
73    myfile.open("grad_info.dat");
74    double steepestDescent;
75    while (iter < runtime) {
76      ukeep = u;
77      steepestDescent = min.linemin_method(dutol);
78      cout << u << "\t" << ukeep << endl;
79      //checking minimization criterion
```

```
80      if (2.0*fabs(ukeep - u) < ftol*(fabs(ukeep) + fabs(u) +
            eps) || u < 1.0e-16) {
81        potential.ellipsoid();
82        if (!steepestDescent && thermal == 0)
83          return;
84      }
85      //use steepest descent instead of conjugate gradient
86      if (steepestDescent) {
87        cout << "steepestDescent" << endl;
88        for (i = 0; i < N; i++) {
89          for (j = 0; j < D; j++) {
90            grad += force[j][i] * force[j][i];
91            xi[j][i] = -force[j][i];
92            g[j][i] = -xi[j][i];
93            h[j][i] = g[j][i];
94            xi[j][i] = h[j][i];
95            umag += xi[j][i] * xi[j][i];
96          }
97        }
98      }
99      else {
100       umag = sqrt(umag);
101       for (i = 0; i < N; i++) {
102         for (j = 0; j < D; j++) {
103           xi[j][i] = xi[j][i] / umag;
104         }
105       }
106       //Calculating new conjugate direction h_i+1
107       gg = 0.0;
108       dgg = 0.0;
109       grad = 0.0;
110       for (i = 0; i < N; i++) {
111         for (j = 0; j < D; j++) {
112           grad += force[j][i] * force[j][i];
113           xi[j][i] = -force[j][i];
114           gg += g[j][i] * g[j][i];
115           dgg += (g[j][i] + xi[j][i])*xi[j][i];
116         }
117       }
118       //(thermal Jamming) with propability p set direction
              0 for particle i
119       if (thermal == 1) {
120         double p;
```

```
121
122           int k = 0;
123           for (i = 0; i < N; i++) {
124             p = probability(generatorMin);
125             if (p < 1.0 / N) {
126               for (j = 0; j < D; j++)
127                 xi[j][i] = 0;
128               randomParticles[k] = i;
129               k++;
130             }
131           }
132           randomParticles[k] = -1;
133         }
134
135         gam = dgg / gg;
136         umag = 0.0;
137         for (i = 0; i < N; i++) {
138           for (j = 0; j < D; j++) {
139             g[j][i] = -xi[j][i];
140             h[j][i] = g[j][i] + gam*h[j][i];
141             xi[j][i] = h[j][i];
142             umag += xi[j][i] * xi[j][i];
143           }
144         }
145         //(thermal Jamming)
146         if (thermal == 1) {
147           int k = 0;
148           while (randomParticles[k] != -1) {
149             i = randomParticles[k];
150             for (j = 0; j < D; j++) {
151               xi[j][i] = 0;
152               h[j][i] = 0;
153             }
154             k++;
155           }
156         }
157         umag = sqrt(umag);
158         int k = 0;
159         for (i = 0; i < N; i++) {
160           for (j = 0; j < D; j++) {
161             xi[j][i] = xi[j][i] / umag;
162           }
163         }
```

```
164
165       }
166       iter++;
167       myfile << iter << "\t" << u << "\t" << grad << "\n";
168     }
169   }
```

Listing 8: linemin.cpp

```
1  #include "Global.h"
2  #include "linemin.h"
3  #include "pair_harm.h"
4  #include "math.h"
5  #include "anint.h"
6  #include "verlet_list.h"
7  #include <algorithm>
8  #include <random>
9  #include "results.h"
10
11 using namespace std;
12 default_random_engine generatorLinemin;
13 uniform_real_distribution<double> randomDir(-1.0, 1.0);
14
15 bool linemin::linemin_method(double dutol) {
16   results printThermal;
17   anint myround;
18   double Linv = 1.0 / L;
19   int step = 0;
20   int return_index = 1;
21   double uprev;
22   int i, j, index;
23
24   x0 = 0.0;
25   x1 = ds_start;
26   u0 = u;
27   ukeep = u;
28
29   f1d0 = 0.0;
30   for (i = 0; i < N; i++) {
31     for (j = 0; j < D; j++) {
32       f1d0 += force[j][i] * xi[j][i];
33     }
34   }
35   f1d0 = -f1d0;
```

```
36    update(x1);
37    u1 = u;
38    f1d1 = f1d;
39    uprev = std::min(u0, u1);
40    double breakLoop = false;
41    int count = 0;
42
43    while (return_index == 1) {
44      dx = x1 - x0;
45      dx2 = dx*dx;
46      dx3 = dx*dx2;
47      if (f1d0*f1d1 < 0.0) {
48        bracket();
49        index = -1;
50      }
51      else {
52        breakLoop = oneside();
53        index = 1;
54      }
55      if ((fabs((uprev - u2) / u2) < dutol) || u2 == 0.0 ||
           abs(f1ds) < 1.0e-14) {
56        for (i = 0; i < N; i++) {
57          for (j = 0; j < D; j++) {
58            x[j][i] += xsol*xi[j][i];
59            if (j < D - 2)
60              x[j][i] = x[j][i] - L*myround.round(x[j][i] *
                    Linv);
61            xu[j][i] = x[j][i];
62          }
63        }
64        pair_harm potential;
65        potential.ellipsoid();
66        return_index = 0;
67      }
68      if (index == -1) {
69        if (f1d0*f1ds < 0.0) {
70          x1 = xsol;
71          u1 = u2;
72          f1d1 = f1ds;
73        }
74        else {
75          x0 = xsol;
76          u0 = u2;
```

```
77            f1d0 = f1ds;
78          }
79          uprev = std::min(u0, u1);
80        }
81        else if (index == 1) {
82          if (u1 < u0) {
83            x0 = x1;
84            u0 = u1;
85            f1d0 = f1d1;
86          }
87          x1 = xsol;
88          u1 = u2;
89          f1d1 = f1ds;
90          uprev = std::min(u0, u1);
91        }
92        if (breakLoop)
93          return true;
94      }
95      printThermal.EllipsoidsInContact();
96      if (thermal == 1)
97        randomDirection();
98      return false;
99    }
100   void linemin::bracket() {
101     double uppcub, u3pcub, x1sol, x2sol, curv1, curv2;
102
103     uppcub = (2.0 / dx2)*(3.0*(u1 - u0) - (f1d1 + 2.0*f1d0)*
          dx);
104
105     u3pcub = (-12.0 / (dx3))*((u1 - u0) - f1d1 + f1d0)*(dx /
          2.0);
106
107     x1sol = -uppcub + sqrt(uppcub*uppcub - 2.0*u3pcub*f1d0);
108     x1sol = x1sol / u3pcub;
109
110     x2sol = -uppcub - sqrt(uppcub*uppcub - 2.0*u3pcub*f1d0);
111     x2sol = x2sol / u3pcub;
112
113     curv1 = uppcub + u3pcub*x1sol;
114     curv2 = uppcub + u3pcub*x2sol;
115
116     if (curv1 > 0.0) xsol = x1sol + x0;
117     if (curv2 > 0.0) xsol = x2sol + x0;
```

```
118     update(xsol);
119     u2 = u;
120     f1ds = f1d;
121 
122     while (u2 > ukeep) {
123       xsol = x0 + (xsol - x0) / 2;
124       update(xsol);
125       u2 = u;
126       f1ds = f1d;
127     }
128   }
129   bool linemin::oneside() {
130     double u2pdis, adx;
131     u2pdis = (f1d1 - f1d0) / (x1 - x0);
132     adx = fabs(x1 - x0);
133     if (u2pdis > 0.0) {
134       xsol = x0 - f1d0 / u2pdis;
135       if (fabs(xsol - x0) > 3.0*adx)
136         xsol = x0 + 3.0*adx;
137     }
138     else
139       xsol = x0 + 1.5*adx;
140     update(xsol);
141     u2 = u;
142     f1ds = f1d;
143 
144     while (u2 > ukeep) {
145       xsol = x1 + (xsol - x1) / 2;
146       if (fabs(xsol - x1) < 1e-16){
147         return true;
148       }
149       update(xsol);
150       u2 = u;
151       f1ds = f1d;
152     }
153     return false;
154   }
155   void linemin::update(double ds) {
156     double dr, dr2, displ, Linv;
157     int i, j;
158     Linv = 1.0 / L;
159     anint myround;
160 
```

```
161     for (i = 0; i < N; i++) {
162       for (j = 0; j < D; j++) {
163         xu[j][i] = x[j][i] + ds*xi[j][i];
164         if (j < D - 2)
165           xu[j][i] = xu[j][i] - L*myround.round(xu[j][i] *
                Linv);
166       }
167     }
168     //checking criterion for updating neighborlist
169     displ = 0.0;
170     for (i = 0; i < N; i++) {
171       for (j = 0; j < D - 2; j++) {
172         dr2 = 0.0;
173         dr = xs[j][i] - x[j][i];
174         dr = dr - L*myround.round(dr*Linv);
175         dr2 += dr*dr;
176       }
177       if (displ < dr2)displ = dr2;
178     }
179
180     if (displ > skinsqr) {
181       verlet_list myneigh;
182       myneigh.vlist();
183     }
184     //calling potential
185     pair_harm potential;
186     potential.ellipsoid();
187     f1d = 0.0;
188     for (i = 0; i < N; i++) {
189       for (j = 0; j < D; j++) {
190         f1d += force[j][i] * xi[j][i];
191       }
192     }
193     f1d = -f1d;
194   }
195   void linemin::randomDirection() {
196     int j, i;
197     double Linv = 1.0 / L;
198     double step = ds_start * 10;
199     int k = 0;
200     while (randomParticles[k] != -1) {
201       //check if particle already has no contacts
202       if (randomParticles[k] == -2) {
```

```
203         k++;
204         continue;
205       }
206       i = randomParticles[k];
207       ukeep = u;
208       for (j = 0; j < D; j++)
209         xr.at(j) = randomDir(generatorLinemin);
210       randomUpdate(i, ds_start);
211       //check if random direction is an increase of u
212       if (u < ukeep) {
213         k++;
214         continue;
215       }
216       for (j = 0; j < D; j++)
217         x[j][i] = xu[j][i];
218 
219       while (u > ukeep) {
220         ukeep = u;
221         randomUpdate(i, step);
222         for (j = 0; j < D; j++)
223           x[j][i] = xu[j][i];
224 
225       }
226       k++;
227     }
228   }
229 
230   void linemin::randomUpdate(int particle, double step)
231   {
232     double dr, dr2, displ, Linv;
233     int j;
234     int i = particle;
235     Linv = 1.0 / L;
236     anint myround;
237     for (j = 0; j < D; j++) {
238       xu[j][i] = x[j][i] + step*xr.at(j);
239       if (j < D - 2)
240         xu[j][i] = xu[j][i] - L*myround.round(xu[j][i] * Linv
                );
241     }
242     //checking criterion for updating neighborlist
243     displ = 0.0;
244 
```

```
245     for (j = 0; j < D - 2; j++) {
246       dr2 = 0.0;
247       dr = xs[j][i] - x[j][i];
248       dr = dr - L*myround.round(dr*Linv);
249       dr2 += dr*dr;
250     }
251     if (displ < dr2)
252       displ = dr2;
253     if (displ > skinsqr) {
254       verlet_list myneigh;
255       myneigh.vlist();
256     }
257     //calling potential
258     pair_harm potential;
259     potential.ellipsoid();
260   }
```

Listing 9: results.cpp

```
1  #include "results.h"
2  #include "Global.h"
3  #include <cmath>
4  #include "anint.h"
5  #include <iostream>
6  #include <fstream>
7  using namespace std;
8  bool *hasContacts;
9
10 void results::init() {
11   hasContacts = new bool[N];
12 }
13
14 void results::EllipsoidsInContact() {
15   sigma1sq = sigma1*sigma1 / 4;
16   sigma2sq = sigma2*sigma2 / 4;
17   anint myround;
18   double Linv = 1.0 / L;
19   double xtemp, ytemp, ztemp;
20   double delx, dely, delz;
21   double r, rsq;
22   int i, j, jb, je, k;
23   double sigma, sigmai, sigmak, sigmasq;
24
25   for (i = 0; i < N; i++)
```

```
26      hasContacts[i] = false;
27
28    for (i = 0; i < N; i++) {
29      xtemp = xu[0][i];
30      ytemp = xu[1][i];
31      ztemp = xu[2][i];
32
33      jb = inum[i];
34      je = inum[i + 1];
35
36      for (j = jb; j < je; j++) {
37        k = ilist[j];
38        if (k < i)continue;
39        delx = xtemp - xu[0][k];
40        delx = delx - L*myround.round(delx*Linv);
41        dely = ytemp - xu[1][k];
42        dely = dely - L*myround.round(dely*Linv);
43        delz = ztemp - xu[2][k];
44        delz = delz - L*myround.round(delz*Linv);
45        rsq = delx*delx + dely*dely + delz*delz;
46        r = sqrt(rsq);
47        sigmai = computeSigma(delx, dely, delz, i);
48        sigmak = computeSigma(delx, dely, delz, k);
49
50        sigma = sigmai + sigmak;
51        sigmasq = sigma*sigma;
52        if (r < sigma - sigma*1e-8) {
53          contacts_1e8++;
54          hasContacts[i] = true;
55          hasContacts[k] = true;
56        }
57      }
58    }
59    f_ov = 0;
60    for (i = 0; i < N; i++) {
61      if (hasContacts[i])
62        f_ov++;
63    }
64    int n = 0;
65    while (randomParticles[n] != -1) {
66      if (hasContacts[randomParticles[n]])
67        randomParticles[n] = -2;
68      n++;
```

```
69    }
70    std::ofstream out;
71    out.open("thermalOut.dat", std::ios::app);
72    out << iter << "\t" << sigma1 << "\t" << N << "\t" <<
          density << "\t" << u
73      << "\t" << contacts_1e8 << "\t" << f_ov << "\t" <<
          initialX << std::endl;
74  }
75  void results::ContactPoints()
76  {
77    contacts = 0;
78    contacts_1e2 = 0;
79    contacts_1e4 = 0;
80    contacts_1e8 = 0;
81    anint myround;
82    double Linv = 1.0 / L;
83    double xtemp, ytemp, ztemp;
84    double delx, dely, delz;
85    double r, rsq;
86
87    int i, j, jb, je, k;
88
89    double sigma, sigmai, sigmak, sigmasq;
90
91    for (i = 0; i < N; i++) {
92      xtemp = xu[0][i];
93      ytemp = xu[1][i];
94      ztemp = xu[2][i];
95
96      jb = inum[i];
97      je = inum[i + 1];
98
99      for (j = jb; j < je; j++) {
100       k = ilist[j];
101       if (k < i)continue;
102       delx = xtemp - xu[0][k];
103       delx = delx - L*myround.round(delx*Linv);
104       dely = ytemp - xu[1][k];
105       dely = dely - L*myround.round(dely*Linv);
106       delz = ztemp - xu[2][k];
107       delz = delz - L*myround.round(delz*Linv);
108       rsq = delx*delx + dely*dely + delz*delz;
109       r = sqrt(rsq);
```

```
110         sigmai = computeSigma(delx, dely, delz, i);
111         sigmak = computeSigma(delx, dely, delz, k);
112         sigma = sigmai + sigmak;
113         sigmasq = sigma*sigma;
114 
115         if (rsq < sigmasq) {
116           contacts++;
117         }
118         if (r < sigma - sigma*1e-2)
119           contacts_1e2++;
120         if (r < sigma - sigma*1e-4)
121           contacts_1e4++;
122         if (r < sigma - sigma*1e-6) {
123           contacts_1e8++;
124         }
125       }
126     }
127 }
128 void results::Print()
129 {
130   std::ofstream out;
131   out.open("out.dat", std::ios::app);
132   out << sigma1 << "\t" << N << "\t" << density << "\t" <<
        u
133     << "\t" << contacts << "\t" << contacts_1e2 << "\t" <<
         contacts_1e4 << "\t" << contacts_1e8 << "\t" <<
         initialX << std::endl;
134 }
135 double results::computeSigma(double delx, double dely,
     double delz, int atomID) {
136   double A, B, C, xdir, ydir, zdir;
137   double phi = xu[3][atomID];
138   double theta = xu[4][atomID];
139 
140   double norm = sqrt(delx*delx + dely*dely + delz*delz);
141   delx = delx / norm;
142   dely = dely / norm;
143   delz = delz / norm;
144   xdir = cos(phi)*cos(theta)*delx - sin(phi)*dely + cos(phi
        )*sin(theta)*delz;
145   ydir = cos(theta)*sin(phi)*delx + cos(phi)*dely + sin(phi
        )*sin(theta)*delz;
146   zdir = -sin(theta)*delx + cos(theta)*delz;
```

```
147
148   A = xdir * xdir / (sigma1 / 2.0 * sigma1 / 2.0);
149   B = ydir * ydir / (sigma2 / 2.0 * sigma2 / 2.0);
150   C = zdir * zdir / (sigma3 / 2.0 * sigma3 / 2.0);
151   return sqrt(1.0 / (A + B + C));
152 }
```

Listing 10: anint.cpp

```
1  #include "anint.h"
2  #include <cmath>
3  int anint::round(double v) {
4    int i = 4;
5
6    if (abs(v)>0.5) {
7      if (v>0.0)
8        i = 1;
9      else i = -1;
10   }
11   if (abs(v) <= 0.5)
12     i = 0;
13   return i;
14 }
```