Implementierung der

Stochastic Series Expansion

für Spin – 1/2 Heisenberg Systeme

Lukas B. Lentner

1. August 2011

Bachelorarbeit an der Fakultät für Physik der Ludwigs-Maximilians-Universität München

Vorwort

> *»Man versteht etwas nicht wirklich,*
> *wenn man nicht versucht, es zu implementieren.«*
>
> — von Donald Ervin Knuth [Knu02]

Sehr geehrter Leser,

das obige Zitat von Donald Knuth stellt eine der Grunderfahrung dar, welche ich bei der Erarbeitung dieses Dokuments ein weiteres Mal erleben durfte. Es behandelt den grundlegenden Unterschied, ob man über ein Thema plappert oder sich die Strenge auferlegt, das Verständnis in Quellcode „zu gießen“. Denn der Computer ist einer der unbarmherzigsten Zeitgenosse, welcher jeden Fehler, jede Unsicherheit bezüglich eines ihm bekannten Themas sofort enttarnt. So sah ich mich 2 Tag vor der Abgabe dieses Dokuments mit einem einfachen SEGMENTATION FAULT konfrontiert, der mich glatt an den Rand der Verzweiflung brachte, nur um 6 Stunden später einen einfachen Tippfehler zu beheben ...

Aber zum Glück empfindet der Mensch eben nicht nur die Last der Exaktheit, sondern auch deren Erfüllung, welche sich rasch nach einem Erfolg breit macht. Kann man diesen darüber hinaus auch noch mit seinem 2. Lieblingsfach verbinden, kann man tiefes Glück erfahren.

Nach dieser kleinen Achterbahn-der-Gefühle-Schilderung, kann ich Ihnen nur noch die selbe Freude beim Leser dieser Arbeit wünschen, wie ich sie (abschnittsweise) beim Schreiben innehatte!

Außerdem möchte ich mich bei meinen großen Unterstützern, meiner Freundin und meinem Bruder und meinen beiden Eltern vom Herzen für jegliche Unterstützung danken. Dank gebührt auch Prof. Dr. Ulrich Schollwoeck und Dr. Fabian Heidrich-Meisner welche mir die Möglichkeit gaben, in einer freundlichen Arbeitsatmosphäre meinen Hobbies, der Physik und der Informatik, nachzugehen!

Vielen Dank und Viel Spaß

Lukas B. Lentner München, 1. August 2011

`kontakt@lukaslentner.de`

Inhaltsverzeichnis

Kapitel 1

Einleitung

Ein Kind saußt auf Schlittschuhen auf einem zugefrorenen See umher.
Der Wind pfeift ihm um die Nase ...

Es ist eine selbstverständliche Naturerfahrung, dass physikalische Stoffe – wie in diesem Fall das Wasser (H2O) – in klar definierten Aggregatszuständen auftreten. Bei genauerem Hinsehen entpuppt sich dieser Sachverhalt jedoch als hochgradig nichttrivial.

Im Zentrum der Entwicklung der modernen Thermodynamik steht die statistische Interpretation von makroskopischen Größen wie Wärme, Druck, etc. zuerst als mikroskopische Zitterbewegungen der zugrundeliegenden Moleküle und schlussendlich als rein statistische Effekte eines abstrakten Zustandsraumes, wobei die Temperatur (in Form des Boltzmann-Faktors) eine Abwägung zwischen häufigen und niedrig-energetischen Zuständen trifft. Während im Grenzfall niedriger Temperaturen die Energieminimierung überwiegt und letztendlich einen **Grunzustand** ausprägt dominiert für hohe Temperaturen die statistische Gleichverteilung **(thermisches Chaos)**. Umso mehr muss es verwundern, dass ein derartiges System bei bestimmten, scheinbar willkürlichen Punkten plötzlich das sprunghafte Verhalten eines **Phasenübergangs** zeigt (siehe [BMB04]).

Die moderne Sicht auf dieses Phänomen geht zurück auf Ernst Ising, der 1924 auf Anregung seines Doktorvaters Wilhelm Lenz ein einfaches Modell linear gekoppelter Elementarmagneten untersuchte. Schon länger war bekannt, dass sich Ferromagneten oberhalb einer sog. Curie-Temperatur aprupt zu einem Paramagneten wandeln; ein Effekt analog zu oben beschriebenen Phasenübergängen, den Ising auf rein statistischem Wege (mit den Methoden der Thermodynamik) zu deuten suchte. Seine mathematische Auswertung ergab jedoch keinen Phasenübergang. Erst im Jahre 1944 gelang Lars Onsager der streng analytische Nachweis eines solchen Phasenübergangs im 2-dimensionalen (Gitter-) Ising-Modell (siehe [Ons44]).

Seither ist dieses Modell eines der am häufigsten untersuchten der statistischen Physik, und ähnliche Phänomene wurden alsbald in vielerlei anderen Kontexten bekannt; in jedem der Fälle zeigt ein lokal schwach gekoppeltes System, wenn es im **thermodynamischen Limes** sehr vieler Teilchen betrachtet wird, bei bestimmten **kritischen Temperaturen** diskontinuirliches Verhalten: Die **Korrelationslänge** ξ, welche die Reichweite der mittelbaren Wechselwirkung beschreibt, divergiert an dieser Stelle und es kommt zur charakteristischen Ausprägung makroskopischer Cluster, sogenannter **Weißscher Bezirke**, während die beteiligten Ordnungsparameter ein bemerkenswert universelles Verhalten zeigen.

Die Erforschung dererlei Effekte zieht sich heutzutage von verschiedensten Gebieten der Physik über rein geometrische (Perkolation) bis hin zu kritischem Verhalten etwa in Märkten, neuronalen-, und Gennetzwerken. Die wenigsten Anwendungen erlauben dabei allerdings eine geschlossene, analytische Lösung. Vielmehr haben sich derweil Techniken herauskristalisiert, diese (ähnlich gearteten) Probleme zuverlässig und schnell numerisch zu behandeln. Dies gilt im besonderen für Fragestellungen aus dem Bereich der Quantenmechanik (z.B. Spingläser, Gittereichtheorien), da die dort auftretenden, typischen Operatoren (anstelle normaler Zahlen) für zusätzliche Komplexität sorgen.

Eine sehr erfolgreiche Herangehensweise besteht in der von D.C Handscomb [Han62] 1962 vorgestellten Reihenentwicklung, mithilfe kombinatorischer Behandlung aller auftretender Operatorprodukte **(Operatorstring)**. Um 1991 gelang es Anders W. Sandvik, Olaf F. Syljuåsen und Juhani Kurkijärvi (siehe [San10]), diesen Ansatz zu einem praktikablen, numerischen Algorithmus weiterzuentwickeln, indem sie ihn mit der klassischen Monte-Carlo Methode zur stochastischen Auswertung a priori unbekannter Funktionen kombinierten **(Stochastic Series Expansion)** und einige wesentliche Neuerungen wie das Loop Update, welches eine systematische Analyse der Operatorstrings vollzieht, entwickelten (siehe z.B. [Gro04]).

Ziel der vorliegenden Bachelorarbeit war es nun zuersteinmal sich in das Themengebiet der Monte-Carlo Methode einzuarbeiten (Kapitel 2) und eine Verbindung zur Vorlesung der Statistischen Physik herzustellen, welche der Autor im Winter gehört hatte. Anschließend wurde ein Simulationsprogramm für das 2-dimensionale Ising-Modell erstellt (Kapitel 3), welches in den Folgemonaten für die Verwendung für SSE (Kapitel 4) stark ausgebaut wurde. Zusätzlich wurden dem System weitere Modellgeometrien wie die offene 1-dimensionale Kette hinzugefügt. Das Erfassen des theoretischen Sachverhalts (Abschnitt 3.1/4.1: Methode) sollte hierbei genauso Beachtung finden, wie eine saubere Implementierung und deren eingängige Beschreibung (Abschnitt 3.2/4.2). Die Hauptaufgabe bildete allerdings die Präsentation der Messdaten bzw. die fundierte Erläuterung der Zusammenhänge (Abschnitt 3.3/4.3), die man den Daten entnehmen kann. Um die Messwerte zu verifizieren wurde am Schluss der Bearbeitungszeit noch eine Erweiterung für die exakte Diagonalisierung (ED) implementiert. Für Systemgrößen, die außerhalb der möglichen Werte für ED lagen, konnte auf Literaturwerte zurückgegriffen werden.

Kapitel 2

Theorie der Monte-Carlo Simulation

2.1 Geschichte

1945 begann am Los Alamos Scientific Laboratory die erste ernsthafte Auseinandersetzung mit Strahlenschutz. Die Forscher suchten nach einer Möglichkeit, den Weg von Neutronen durch unterschiedliche Materialien vorherzusagen, was aufgrund der oft komplizierten Geometrie nur noch schwer analytisch geschehen konnte. Dabei erkannten die Physiker Stanislaw Ulam und John von Neumann schließlich (basierend auf den Ideen von Enrico Fermi um 1935) die Mächtigkeit eines **stochatischen Ansatzes**, bei dem durch Verfolgen einzelner Trajektorien von Teilchen in größerer Anzahl schnell auf eine gute Näherung der tatsächlichen Intensitäts-/Wahrscheinlichkeitsverteilung geschlossen werden kann [Met87].

Nicholas Metropolis, ebenfalls an dem Projekt beteiligt, gab dem Verfahren den Namen **Monte-Carlo Methode**, welcher sich auf die Spielbank Monte-Carlo, die im gleichnamigen Stadtteil des Stadtstaates Monaco liegt, bezieht. Anlass hierfür soll Ulams Onkel gegeben haben, der sich mehrmals von Verwandten Geld zum Spielen leihen wollte [Met87].

Heute findet die Methode zahlreiche Anwendungen in der Statistischen Physik, Numerik und Optimierung. In der Teilchenphysik beispielsweise werden Zerfallsbäume von Atomen auf ihre Häufigkeits-/Wahrscheinlichkeitsverteilung der Zwischenprodukte hin untersucht [Pyt], während mit dem selben Verfahren in der Thermodynamik die Wahrscheinlichkeitsverteilung der Mikroskopischen Zustände verfolgt wird [BMB04]. Weiterhin gibt es ganz klassische Beispiele als Anwendung, wie die Monte-Carlo Integration oder die Approximation von π.

2.2 Ziel

Das Ziel der **Monte-Carlo Simulation** (MCS) ist die Berechnung eines statistischen Mittels einer Größe A in einem n-dimensionalen Zustandsraum Ω, dessen Zustände a priori gewichtet sind (Wahrscheinlichkeitsraum):

$$\langle A \rangle = \sum_{\sigma \in \Omega} p_\sigma \cdot A(\sigma) \tag{2.1}$$

p_σ steht hier für die Wahrscheinlichkeit des Zustandes σ und $A(\sigma)$ ist der Wert der Größe A bei diesem Zustand. Für kontinuierliche Fälle ersetzt man die Summe durch ein Integral.

Beispiele hierfür reichen von der genannten Häufigkeitsverteilung von Zwischenprodukten über Boltzmann-gewichtete Mikrozustände (s. Kapitel 3) bis zum Operatorstring, welcher einer gegebenen, quantenmechanischen Amplitudenverteilung folgt (s. Kapitel 4).

2.3 Idee: Markov-Kette

Oft ist es nicht möglich, die oben angegebene Summe auszuwerten (z.B. wenn Ω sehr groß ist). In diesem Fall wollen wir den Zustandsraum quasidicht durch eine Markov-Kette von R Zuständen $\sigma_0, \sigma_1, \ldots \sigma_{R-1}$ ablaufen. Die Häufigkeit eines Zustandes σ in der Kette soll sich dabei im Grenzfall $R \to \infty$ genau der Wahrscheinlichkeit des Zustandes p_σ annähern (Importance Sampling). Der Mitterwert kann sodann erheblich leichter nach dem *Gesetz für große Zahlen* durch das arithmetische Mittel über die Kette approximiert werden:

$$\langle A \rangle \approx \overline{A} = \frac{1}{M} \sum_{m=0}^{M-1} A(\sigma_m) \tag{2.2}$$

Eine Markov-Kette beginnt mit einem beliebigen Anfangszustand σ_0. Von diesem aus werden mit einer Übergangswahrscheinlichkeit $(\boldsymbol{W})_{\sigma_0\sigma_1}$ Sprünge im Zustandsraum ausgeführt (MC-Schritte), welche die neuen Kettenglieder $\sigma_2\sigma_3\ldots$ definieren. Damit die Markov-Kette zur gewünschten Wahrscheinlichkeitsverteilung p_σ führt, muss bei der Bildung von $\boldsymbol{W}$ auf die zwei folgenden Bedingungen geachtet werden:

a) Die Bildung der Kette muss der **Ergodizität** folgen. D.h. sie muss theoretisch alle Zustände erreichen können (was sie in der Praxis natürlich nicht tut, da wir $R \ll |\Omega|$ wählen).

b) Die Übergangswahrscheinlichkeiten $\boldsymbol{W}$ müssen insofern im **Gleichgewicht** sein, als dass

$$\sum_{\sigma \in \Omega} p_\sigma \cdot W_{\sigma\nu} = p_\nu \ . \tag{2.3}$$

Wir fordern also, dass die gewünschte Wahrscheinlichkeitsverteilung einen Fixpunkt der Markov-Kette darstellt (Stationäre Lösung).

Eine deutlich stärkere Bedingung als b) stellt die sog. **Detailed Balance** (dt. detailiertes Gleichgewicht) dar:

$$p_\sigma \cdot W_{\sigma\nu} = p_\nu \cdot W_{\nu\sigma} \tag{2.4}$$

Anschaulich besagt sie, dass ein Sprung von einem Markov-Kettenglied zum Nachbar genauso wahrscheinlich ist, wie in Gegenrichtung. Die Kette besitzt also keine ausgezeichnete Richtung – es handelt sich um einen *reversiblen Prozess* im thermodynamischen Sinne (s. Kapitel 9 in [BMB04]). Die **Detailed Balance** stellt immer schon ein **Gleichgewicht** dar, da

$$\sum_{\sigma\in\Omega} p_\sigma \cdot W_{\sigma\nu} = \sum_{\sigma\in\Omega} p_\nu \cdot W_{\nu\sigma} = p_\nu \cdot \sum_{\sigma\in\Omega} W_{\nu\sigma} = p_\nu \ . \tag{2.5}$$

Hierbei verwendet man im letzten Schritt, dass der Zustand ν in jedem Fall in irgendeinen nächsten Zustand σ übergeht, die Zeilensummen der Matrix also jeweils 1 ergeben.

2.4 Metropolis Algorithmus

Die MCS erfordert also für eine gegebene Wahrscheinlichkeitsverteilung p_σ eine Wahl der Übergangswahrscheinlichkeiten $\boldsymbol{W}$, sodass p_σ **stationär** ist. 1953 stellte Nicholas Metropolis *et al.* eine solche Wahl vor:

$$W_{\nu\sigma} = \begin{cases} p_\sigma/p_\nu & p_\sigma < p_\nu \\ 1 & p_\sigma \geq p_\nu \end{cases} \tag{2.6}$$

Es kann leicht gezeigt werden, dass dieser Vorschlag sogar *Detailed Balance* (Gl. 2.4) erfüllt. Ein weiterer Algorithmus ist nach Roj J. Glauber benannt (*Glauber dynamics*) [LLP10].

2.5 Thermalisierung

Nachdem als Anfangszustand der Markov-Kette ein beliebig ausgewählter Zustand verwendet wird, werden die ersten Kettenglieder in ihrer Verteilung noch weit von der angestrebten Wahrscheinlichkeitsverteilung p_σ abweichen, auch weil die verlangte **Ergodizität** in den wenigen Schritten noch gar nicht zum Tragen kommen konnte.

Den Vorgang, bis zum ersten Mal eine hinreichend gute Übereinstimmung mit p_σ vorliegt, nennt man **Thermalisierung**: Vor der eigentlichen Messung (R_1 MC-Schritte) muss also eine genügend große Anzahl R_0 von Thermalisierungsschritten durchgeführt werden, während deren noch keine Messungen durchgeführt werden. Insgesamt sind dann $R = R_0 + R_1$ Monte-Carlo Schritte vonnöten.

In der Praxis, wie auch in dieser Arbeit, werden für R_0 oft Erfahrungswerte verwendet, die eine konstante, meist zu große Schrittanzahl erfordern (z.B. $R_0 = \frac{3}{2}R_1$). Alternativ können die einzelnen Messdaten auch vollständig gespeichert und in der Auswertung sortiert werden. Im Nachhinein kann dann über die tatsächliche Wahrscheinlichkeitsverteilung auf die Thermalisierungsphase geschlossen werden.

2.6 Autokorrelationsfunktion und Fehlerberechnung

Alle Messwerte der Größe A müssen nach der Termalisierung in der Auswertung statistisch interpretiert werden. Dabei ist zu beachten, dass die Daten von aufeinanderfolgenden Zuständen statistisch abhängig sind. Wie viele MC-Schritte zwischen zwei Messungen notwendig sind, um unabhängige Werte zu erhalten, gibt die **Autokorrelationszeit** τ_A an (im weiteren ist mit „Zeit“ immer die Simulationszeit gemessen in MC-Schritten gemeint). Zur Berechnung derselben wird die Autokorrelationsfunktion betrachtet:

$$\Theta_A(t) = \frac{\langle A(\sigma_{i+t}) \cdot A(\sigma_i)\rangle - \langle A\rangle^2}{\langle A^2\rangle - \langle A\rangle^2} \tag{2.7}$$

Hierbei läuft die Mittelwertbildung mit der Variable i über die gesamte ausgewertete Simulationszeit R_1. Θ_A ist in solch einer Weise normiert, dass $\Theta_A(0) = 1$ und $\Theta_A(t \to \infty) = 0$. Die Autokorrelationsfunktion hängt dabei negativ exponentiell mit der Autokorrelationszeit zusammen:

$$\Theta_A(t) \sim e^{-t/\tau_A} \tag{2.8}$$

Nachdem τ_A auf diese Art ermittelt wurde, können die Messwerte in Gruppen mit z.B. der Länge $3 \cdot \tau_A$ [BMB04] gebündelt und unabhängige Gruppenmittelwerte

$$\overline{A}_b = \frac{1}{3\tau_A} \sum_{i=0}^{3\tau_A - 1} A(\sigma_{b\cdot\tau_a + i}) \tag{2.9}$$

berechnet werden, wobei b hier der Gruppen-Index ist und die Anzahl der Gruppen:

$$B = \left\lfloor \frac{R_1}{3\tau_A} \right\rfloor \tag{2.10}$$

Nach dem *Zentralen Grenzwert-Satz* folgen diese Gruppenmittelwerte sodann einer Gauß-Verteilung, deren Erwartungswert dann der angestrebte **Mittelwert** ist:

$$\overline{A} = \frac{1}{B} \sum_{b=0}^{B-1} \overline{A}_b \tag{2.11}$$

Zur Abschätzung des Fehlers wird immer die Standardabweichung mit ausgewertet:

$$\sigma_A = \sqrt{\frac{1}{B(B-1)} \sum_{b=0}^{B-1} (\overline{A}_b - \overline{A})^2} \tag{2.12}$$

Kapitel 3

Klassische MCS
am Beispiel des Ising-Modells

Um die Grundlagen der Monte-Carlo Simulation (MCS) kennenzulernen, betrachten wir zuerst die Simulation des klassischen, 2-dimensionalen Ising-Modells mit periodischer Randbedingung. Als Messgrößen wählen wir die typischen thermodynamischen Größen: Den Mittelwert der Energie, Wärmekapazität, Magnetisierung und magnetischen Suszeptibilität. Außerdem betrachten wir die absolute Magnetisierung und die absolute magnetische Suszeptibilität, also den Mittelwert des Absolutbetrags der Spin-Summe und dessen Varianz (weil sich ohne äußeres Magnetfeld die Magnetisierung immer auf 0 mittelt). All diese Größen werden pro Spin gemessen.

Um ein Gegenüberstellen zu erleichtern, hat dieses und das nächste Kapitel eine analoge Struktur: Im ersten Abschnitt knüpfen wir an Kapitel 2 an, d.h. wir werden unseren Zustandsraum Ω und die Übergangswahrscheinlichkeiten $\boldsymbol{W}$ für dieses Szenario definieren. Danach betrachten wir die Implementierung der erstellten Anwendung detailiert. Im letzten Abschnitt werden die Ergebnisse verschiedener Simulationen vorgestellt und diskutiert.

3.1 Methode

3.1.1 Das Ising-Modell

Für das klassische, ferromagnetische Ising-Modell ist der Hamiltonian

$$H_{\text{Ising}} = -\sum_{\langle i,j \rangle} J_{ij} \cdot S_i^z S_j^z - h \sum_{i=0}^{N-1} \mu_i \cdot S_i^z \tag{3.1}$$

zusammengesetzt aus einer magnetischen z-Koppelung benachbarter Spins $\langle i, j \rangle$, die durch die Bindungsmatrix $\boldsymbol{J}$ gewichtet wird, und der Wechselwirkung eines externen Magnetfelds $\boldsymbol{h} = (0, 0, h)^T$ mit den magnetischen Momenten $\boldsymbol{\mu} = (0, 0, \mu)^T$. Für unser Beispiel setzten wir alle $J_{ij} = 1$ sowie $\mu_i = 1$ (Homogenität) und betrachten die Anordnung ohne Magnetfeld ($h = 0$)

– da uns nur die z-Richtung interessiert, setzen wir $S = S^z \in \{-1; 1\}$. Der Hamiltonian erhält dann die vereinfachte Struktur:

$$H = -\sum_{\langle i,j \rangle} S_i S_j \tag{3.2}$$

3.1.2 Sampling

Wegen der vorgegebenen Teilchenanzahl N und Temperatur T können wir für eine beliebige Größe A den Mittelwert

$$\langle A \rangle = \sum_{\sigma \in \Omega} \frac{e^{-\beta E_\sigma}}{Z} \cdot A(\sigma) \tag{3.3}$$

als kanonisches Ensamble ansetzen, wobei

$$Z = \sum_{\sigma \in \Omega} e^{-\beta E_\sigma} \tag{3.4}$$

die kanonische Zustandssumme, β die reduzierte Temperatur $1/T$ (wir setzten $k_B = 1$) und E_σ die Energie eines gewissen mikroskopischen Zustandes σ (Konfiguration) darstellt. Analog zum Abschnitt 2.4 wenden wir nun die **Monte-Carlo Methode** auf diese Konfigurationen $\in \{1, 2\}^N$ an. Im Vergleich zur Gl. 2.1 sieht man hierbei, dass die Wahrscheinlichkeit eines Zustandes Boltzmann-verteilt ist:

$$p_\sigma = \frac{e^{-\beta E_\sigma}}{Z} \tag{3.5}$$

Die Gewichte (speziell die Zustandssumme) sind allerdings schwer zu berechnen, da der Zustandsraum in solch einem Spin-System exponentiell mit der Spinanzahl anwächst ($|\Omega| \sim 2^N$) und eine numerische Berechnung von Z für große Systeme oft nicht mehr möglich ist. Für den **Metropolis Algorithmus** (siehe Gl. 2.6), benötigen wir allerdings diese einzelnen Gewichte gar nicht, sondern können uns mit deren Verhältnissen, die dann die Übergangswahrscheinlichkeiten $\boldsymbol{W}$ darstellen, begnügen:

$$W_{\nu\sigma} = \begin{cases} e^{-\beta(E_\sigma - E_\nu)} & E_\sigma > E_\nu \\ 1 & E_\sigma \leq E_\nu \end{cases} \tag{3.6}$$

3.2 Implementierung

Die Anwendung orientiert sich an [San10]. Sie gliedert sich grob in die Initialisierung des Systems, die Simulation des Modells sowie die Analyse der Messdaten. Um die gewünschten Größen abhängig von der Temperatur betrachten zu können, führen wir das Programm für mehrere Temperaturen aus.

3.2.1 Initialisierung

Generell müssen zuerst folgende Parameter festgelegt werden:

- Anzahl der Spins N,
- Anzahl der Messungen R_1 und
- Temperatur des Systems T.

Für den Status der Spins legen wir ein boolsches Array der Länge N an und initialisieren es mit zufälligen Werten (Anfangszustand). Da sich alle Messgrößen von der Energie und der Spin-Summe ($\approx$ Magnetisierung, siehe Abschnitt 3.3)

$$S = \sum_{i=0}^{N-1} S_i \tag{3.7}$$

ableiten lassen, speichern wir immer deren aktuelle Werte ab. Wie wir später sehen werden, können wir beide Werte in jedem MC-Schritt direkt angepassen (Update) und müssen diese nicht jedes Mal erneut berechnen (zu Beginn ist dies aber natürlich vonnöten).

3.2.2 Simulation

Um eine Markov-Kette der Länge R zu sampeln, verwenden wir eine Schleife, die jeweils einen MC-Schritt durchführt. Ab R_1 Durchläufen (Thermalisierung, siehe Abschnitt 2.5) legen wir jedes Mal zusätzlich die aktuelle Energie, die Magnetisierung ($M = S/N$) und die absolute Magnetisierung ($M' = |S|/N$) in einem geeigneten Array ab.

Monte-Carlo Schritt Wir erzeugen je das nächste Markov-Kettenglied, indem wir versuchen, jeden Spin des Systems umzudrehen (engl. flip). Das Umdrehen wird jeweils gestattet, wenn eine Zufallszahl zwischen 0 und 1 kleiner ist als das Boltzmanngewicht

$$e^{-\beta \Delta E} \ , \tag{3.8}$$

wobei ΔE der Energieunterschied zwischen der neuen, möglichen Konfiguration und der aktuellen ist. Damit decken wir bereits die Gl. 3.6 voll ab, da die Zufallszahl im zweiten Fall ($\Delta E < 0 \Rightarrow W_{\nu\sigma} = 1$) auf jeden Fall kleiner ist als das Boltzmanngewicht.

Updates Wird das Umdrehen eines Spins erlaubt, modifizieren wir das Spin-Array und addieren zur aktuellen Energie und Spin-Summe den berechneten Unterschied ΔE und ΔS:

- Zu ΔE tragen nur die Koppelungen zwischen dem Spin, den wir umdrehen wollen, und dessen Nachbaren bei. Diese sind im 2-dimensionalen Gitter die vier Spins über, unter sowie links und rechts von ihm.
- ΔS ergibt sich einfach aus dem alten Status des Spins (± 2).

3.2.3 Analyse

Die Mittelwerte folgender Größen wollen wir berechnen **(immer pro Spin)**:

$$\text{Energie}: \quad \left\langle \frac{E}{N} \right\rangle = \frac{-\partial_\beta \ln Z}{N} = \left\langle \frac{H}{N} \right\rangle \ , \tag{3.9}$$

$$\text{Wärmekapazität}: \quad \left\langle \frac{C}{N} \right\rangle = \frac{\partial_T H}{N} = \frac{N}{T^2} \left(\left\langle \left(\frac{H}{N} \right)^2 \right\rangle - \left\langle \frac{H}{N} \right\rangle^2 \right) \ , \tag{3.10}$$

$$\text{Magnetisierung}: \quad \left\langle \frac{M}{N} \right\rangle = \frac{T \partial_B \ln Z}{N} = \left\langle \frac{S_i}{N} \right\rangle \ , \tag{3.11}$$

$$\text{magnetische Suszeptibilität}: \quad \left\langle \frac{\chi}{N} \right\rangle = \frac{\partial_B M}{N} = \frac{N}{T} \left(\left\langle \left(\frac{S_i}{N} \right)^2 \right\rangle - \left\langle \frac{S_i}{N} \right\rangle^2 \right) \ , \tag{3.12}$$

$$\text{abs. Magnetisierung}: \quad \left\langle \frac{M'}{N} \right\rangle = \left\langle \left| \frac{S_i}{N} \right| \right\rangle \ , \tag{3.13}$$

$$\text{abs. mag. Suszeptibilität}: \quad \left\langle \frac{\chi'}{N} \right\rangle = \frac{N}{T} \left(\left\langle \left| \frac{S_i}{N} \right|^2 \right\rangle - \left\langle \left| \frac{S_i}{N} \right| \right\rangle^2 \right) \ . \tag{3.14}$$

Für jede dieser Größen werden – wie in Abschnitt 2.6 ausgeführt – zuerst die Autokorrelationszeit berechnet und anschließend die Messdaten gruppiert und schließlich gemittelt.

3.2.4 Quellcode

Der vom Author geschriebene C++ Quellcode ist im Anhang A zu finden. Folgende Dateien sind für diese, klassische Simulation relevant:

- A.1: Hauptprogramm SIM
- A.2: Abstrakte Gitterklasse
- A.5: 2D Gitter mit periodischen Randbedingungen
- A.6: Abstrakte Algorithmusklasse
- A.8: Ising Algorithmus
- A.10: Abstrakte Analyseklasse
- A.11: Analyse für die Energie (Ising)
- A.12: Analyse für die Wärmekapazität (Ising)
- A.13: Analyse für die Magnetisierung (Ising)
- A.14: Analyse für die magnetische Suszeptibilität (Ising)
- A.15: Analyse für die absolute Magnetisierung (Ising)
- A.16: Analyse für die absolute magnetische Suszeptibilität (Ising)

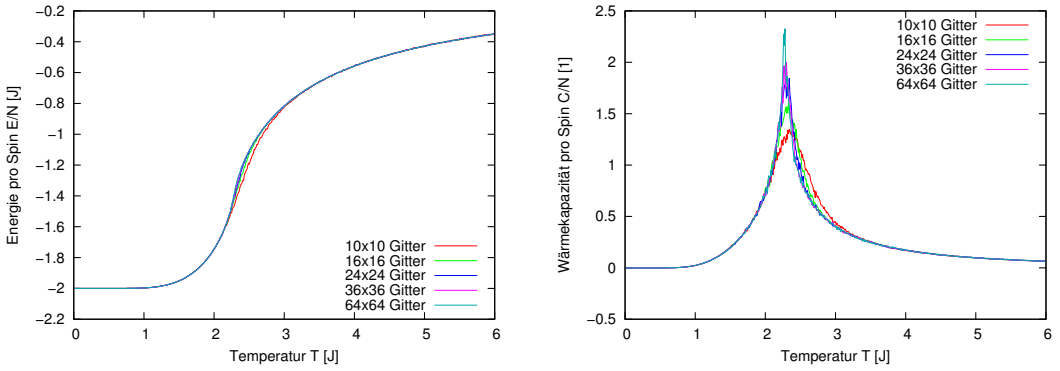
3.3 Ergebnisse und Diskussion

Bei der graphischen Aufbereitung wurde der Übersichtlichkeit wegen auf eine Angabe des Fehlers verzichtet (im Hauptteil werden wir sie gesondert darstellen). Für die Mittelwerte der Grundgrößen E, M und M' sind diese kleiner als graphisch darstellbar. Mittelwerte weiterführender Größen C, χ und χ' besitzen dagegen üblicherweise einen signifikanten Fehler in der Nähe des Phasenübergangs [Nol07]

$$T_c = \frac{2}{\ln(1+\sqrt{2})} \approx 2.269185 \ , \tag{3.15}$$

ansonsten gilt dasselbe wie bei den Grundgrößen.

3.3.1 Mittelwert der Energie und Wärmekapazität



(a) **Mittelwerte der Energie** (b) **Mittelwerte der Wärmekapazität**

Abbildung 3.1: Mittelwerte der Energie und Wärmekapazität für verschieden große Gitter mit periodischen Randbedingungen bei 10000 Messpunkten pro Temperaturpunkt; *Quelle:* Eigenwerk

Der Mittelwert der **Energie** (Gl. 3.9) in Abb. 3.1a verläuft erwartungsgemäß von -2 nach 0: Für kleine Temperaturen stehen alle Spins in die gleiche Richtung (*Grundzustand*), da die Wahrscheinlichkeit eines Flips (Gl. 3.8) eines einzelnen Spins gegen alle anderen verschwindend gering ist. Weil in einem 2-dimensionalen Gitter mit periodischen Randbedingungen die Anzahl der Koppelungen $N_b = 2N$ ist, hat der Mittelwert hier den Wert -2. Je höher jedoch die Temperatur steigt, je geringer wird der Einfluss des Boltzmann-Gewichts und führt letztendlich zu einer Gleichverteilung der Spins, die für $T \to \infty$ $\langle E/N \rangle \to 0$ liefert (*thermisches Chaos*).

Am Mittelwert der **Wärmekapazität** (Gl. 3.10) in Abb. 3.1b, also der Ableitung der mittleren Energie nach T, erkennt man deutlich den Phasenübergang, der sich durch den höchsten Wert für die Steigung der mittleren Energie erkenntlich macht. Dies erklärt sich durch die Bildung von gleichartig ausgerichteten (korrelierten) Spin-Clustern (Weißsche Bezirke) in der Größenordnung der Korrelationslänge ξ [Nol07]. Da die Größe außerdem die Varianz der Energie darstellt, erklärt sich der Verlauf ebenfalls aus der starken Reaktion der Energie auf geringste Temperaturänderungen; hingegen sind Grundzustand und thermisches Chaos weitgehend „stabil“.

Ein Vergleich **verschiedener Systemgrößen** zeigt eine Verschiebung des Peaks der mittleren Wärmekapazität als auch dessen Anwachsen, während abseits des Phasenübergangs kein Unterschied festzustellen ist. Der Grund hierfür kann wieder mit den Weißschen Bezirken plausibel gemacht werden (siehe Seite 33 in [San10]):

- $T \ll T_c$: Nahe am Grundzustand erwarten wir unabhängig von der Systemgröße einen ∞-größen Bezirk mit vereinzelten Störungen ($\xi = \infty$).
- $T \gg T_c$: Im thermischen Chaos (ξ klein) von größtenteils dekorrelierten Einzelspins spielt die makroskopische Systemgröße N keine Rolle.
- $T \approx T_c$: Nahe dem Phasenübergang nimmt jeder Bezirk einen makroskopischen Anteil des Systems ein, sodass sich das Verhalten bereits bei kleinen Änderungen massiv ändert.

Um schließlich die reale Übergangstemperatur T_c für $N \to \infty$ (thermodynamischer Limes) zu finden und die kritischen Exponenten bestimmen zu können, müssen wir einen polynominalen Fit über mehrere Systemgrößen hinweg verwenden (Finite Size Scaling) [BMB04]. Für die obige Simulation ergab sich:

	Eigene Werte	Exakter Wert	Wert durch Molekularfeldnäherung
T_c	2.26228	2.269185	*Kein Wert*
ν	1	1	0.5
γ	1.75958	1.75	1
β	0.125	0.125	0.5

Tabelle 3.1: Übergangstemperatur und kritische Exponenten

3.3.2 Autokorrelationszeit der Energie

Wie wir sehen, nimmt auch die **Autokorrelationszeit der Energie** τ_E (Gl. 2.8) in Abb. 3.2 um den Phasenübergang herum stark zu. Das bedeutet, dass die Konfigurationen über mehrere MC-Schritte hinweg korrelieren bzw. statistisch abhängig sind. Begründet liegt dies in der Tatsache, dass die Korrelationslänge ξ – wie schon mehrfach erwähnt – am Phasenübergang divergiert und die makroskopische Propagation von Information durch das System viel Zeit benötigt [BMB04]. τ_E ist neben der Temperatur auch vom System insbesondere dessen Größe abhängig.

3.3.3 Mittelwert der Magnetisierung und magnetischen Suszeptibilität

Die weiter oben angesprochene (dekorrelierte) Gleichverteilung der Spins bei hohen Temperaturen, drückt sich verständlicherweise in der verschwindenen, mittleren **Magnetisierung** (Gl. 3.11) in Abb. 3.3a für $t \gg T_c$ aus. Verringert man vom thermischen Chaos aus die Temperatur, so beginnen die Spins in der Nähe des kritischen Punktes, sich innerhalb der Weißschen Bezirke in eine Richtung auszurichten. Da hierbei keine der beiden Richtungen einen Vorzug erhält, wechselt der Mittelwert der Magnetisierung beliebig das Vorzeichen.

Hier zeigt sich die Simulation mit dem Metropolis Algorithmus (Gl. 3.6) fehlerbehaftet, da sich die positiven und negativen Beiträge dieser Bezirke aus Symetriegründen insgesamt aufheben

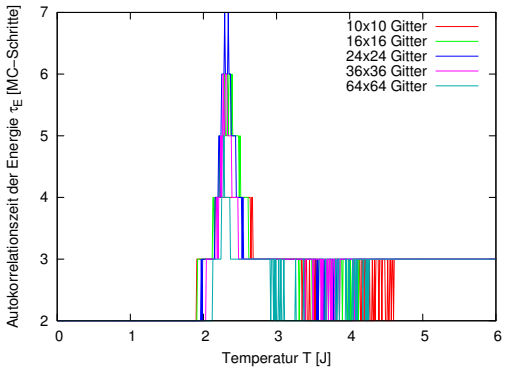


Abbildung 3.2: **Autokorrelationszeiten der Energie** für verschieden große Gitter mit periodischen Randbedingungen bei 10000 Messpunkten pro Temperaturpunkt; *Quelle:* Eigenwerk

müssten. Die Cluster kann der **Algorithmus** jedoch nur von den Rändern her umdrehen, denn das Flippen eine Spins in der Mitte eines solchen Clusters ist sehr unwahrscheinlich. Die Cluster bleiben also für längere Zeit bestehen und brechen die Symetrie der Verteilung – es entsteht eine **vermeintliche Magnetisierung**, die letztlich dem Verlust von Ergodizität (siehe Abschnitt a) der Markov-Kette geschuldet ist. Für kleine Temperaturen sieht man sogar, dass sich das Vorzeichen garnicht mehr verändert und mit der Zeit alle Spins ebenfalls passend geflippt werden.

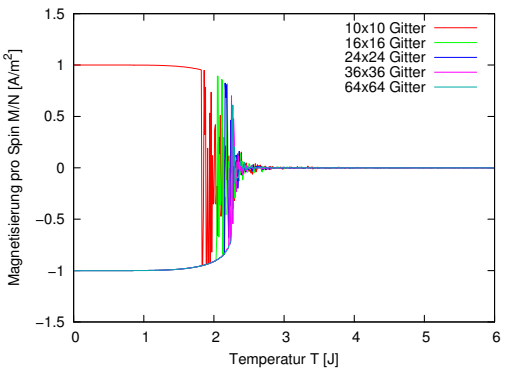
Ein Beispiel für eine Modifikation des Algorithmus', welcher diesem Sachverhalt Sorge trägt, ist der Cluster-Algorithmus, der von Swendson/Wang (1987) und Wolff (1989) erstmals vorgestell wurde. Die Cluster werden analysiert, im Ganzen gewichtet und eventuell geflippt. Für kleine Temperaturen werden die Cluster also korrekterweise ebenfalls bei fast jedem MC-Schritt geflippt und die Mittelung ergibt wie erwartet eine verschwindende Magnetisierung (vgl. [Gro04]).

Die mittlere magnetische Suszeptibilität (Gl. 3.12) in Abb. 3.3b zeigt ein ähnliches Verhalten, wie die Wärmekapazität. Auch sie formt zum Phasenübergang hin einen einen Peak aus. In der Realität ist allerdings auch sie immer 0, falls es kein äußeres Magnetfeld gibt.

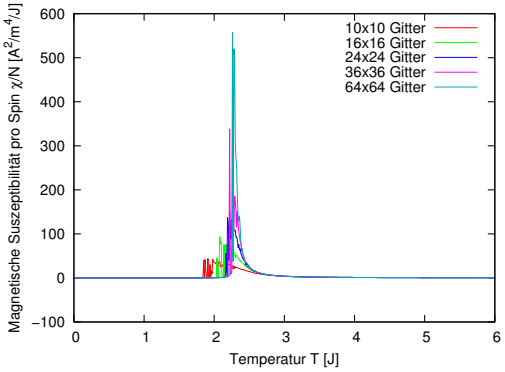
3.3.4 Mittelwert der abs. Magnetisierung und mag. Suszeptibilität

Nachdem die Magnetisierung zufällig das Vorzeichen wechselt und sich für Systeme ohne externes Magnetfeld wegmittelt, betrachtet man häufig nur die **absolute Magnetisierung** (Gl. 3.13) in Abb. 3.4a und deren Varianz (Gl. 3.14) in Abb. 3.4b. Beide Größen stellen sogenannte **Ordnungsparameter** dar; sie verdeutlichen gut das Ausbilden von Clustern. Insbesondere zeigen sie für **verschiedene Systemgrößen** N, dass der Prozess vom thermischen Chaos zur Ordnung hin im größeren System deutlich schneller vonstatten geht. Da sie Spin-Inversions-invariant sind, stellen sie trotz Clusterbildung Größen dar, welche der Metropolis Algorithmus zuverlässig berechnet.

Die mittlere absolute magnetische Suszeptibilität bietet sich neben der Wärmekapazität weiterhin an, T_c und kritische Exponenten zu finden 3.3.1.

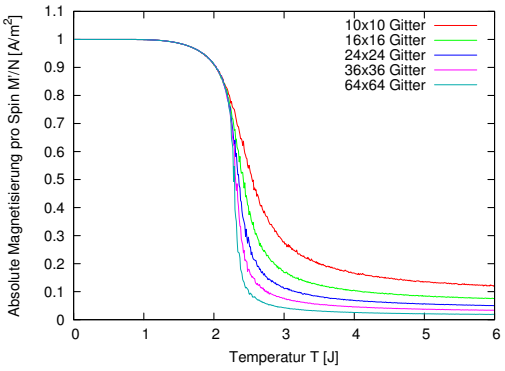


(a) **Mittelwerte der Magnetisierung**

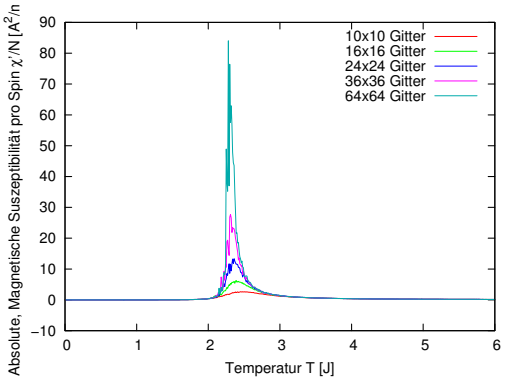


(b) **Mittelwerte der mag. Suszeptibilität**

Abbildung 3.3: Mittlere Magnetisierung und magnetischen Suszeptibilität für verschieden große Gitter mit periodischen Randbedingungen bei 10000 Messpunkten pro Temperaturpunkt; *Quelle:* Eigenwerk



(a) **Mittelwerte der abs. Magnetisierung**



(b) **Mittelwerte der abs. mag. Suszeptibilität**

Abbildung 3.4: Mittelwerte der absoluten Magnetisierung und magnetischen Suszeptibilität für verschieden große Gitter mit periodischen Randbedingungen bei 10000 Messpunkten pro Temperaturpunkt; *Quelle:* Eigenwerk

Kapitel 4

Quantenmechanische MCS
mit Hilfe der Stochastic Series Expansion

Im Folgenden wollen wir uns nun der Simulation des quantenmechanischen Spin-1/2 Heisenberg Systems mithilfe der SSE nach [BMB04] und [San10] zuwenden.

Unsere Beispielapplikation berechnet für verschiedene Systeme (*Offene Kette*, *Periodische Kette* und *Periodisches Gitter*) die Energie und die Wärmekapazität, wobei wir sämtliche Daten immer entweder im Vergleich zur *Exakten Diagonalisierung* des Hamiltonoperators oder entsprechenden Literaturwerten betrachten und einordnen.

4.1 Methode

4.1.1 Das Spin-1/2 Heisenberg System

Der Hamiltonian des Spin-1/2 Heisenberg Systems mit Zeemann-Term ist gegeben durch:

$$H_{\text{Heisenberg}} = \sum_{\langle i,j \rangle} J_{ij} \cdot \left(S_i^x S_j^x + S_i^y S_j^y + \Delta S_i^z S_j^z \right) - h \sum_{i=0}^{N-1} \mu_i \cdot S_i^z \tag{4.1}$$

Man betrachtet also eine magnetische Koppelung von 3-dimensionalen, benachbarten Spins (abweichend in Z-Richtung um den Faktor Δ) mit der Bindungsmatrix $\boldsymbol{J}$, an denen zusätzlich ein externes Magnetfeld $\boldsymbol{h} = (0, 0, h)^T$ angreift. Das magnetische Moment sei mit $\boldsymbol{\mu} = (0, 0, \mu)^T$ benannt.

Das Heisenberg System wird folgendermaßen durch das Δ klassifiziert:

- $\Delta < -1$: Ising-Phase
- $\Delta = -1$: Isotrope ferromagnetische Phase
- $|\Delta| < 1$: XY-Phase
- $\Delta = 1$: Isotrope antiferromagnetische Phase

- $\Delta > 1$: Néel-Phase

Wir wollen uns hier speziell mit der isotropen antiferromagnetischen Phase beschäftigen, wobei wir auch hier wieder alle $J_{ij} = 1$ sowie $\mu_i = 1$ setzen wollen (Homogenität). Die Anordnung betrachten wir weiterhin ohne Magnetfeld ($h = 0$). Daraus ergibt sich der vereinfachte Hamiltonian

$$H = \sum_{\langle i,j \rangle} S_i^x S_j^x + S_i^y S_j^y + S_i^z S_j^z = \sum_{\langle i,j \rangle} \boldsymbol{S}_i \cdot \boldsymbol{S}_j \tag{4.2}$$

welchen man mit $S_i^{\pm} = S_i^x \pm i S_i^y$ zu

$$H = \sum_{b=1}^{N_b} \frac{1}{2} \left(S_{i(b)}^+ S_{j(b)}^- + S_{i(b)}^- S_{j(b)}^+ \right) + S_{i(b)}^z S_{j(b)}^z \tag{4.3}$$

umformen kann, wobei wir die Pärchen $\langle i, j \rangle$ mit einem Index b durchnummerieren und für die Spin-Indizes Nachbar-Funktionen $i(b)$ und $j(b)$ einführen; diese ergeben sich aus der Geometrie des Systems. Betrachtet man die Spin Operatoren dann in der Standardbasis bezüglich S^z, so kann man den Hamiltonian pro Koppelung (engl. Bond) in einen diagonalen $H_{0,b}$ und einen off-diagonalen Teil $H_{1,b}$ aufspalten:

$$H = -\sum_{b=1}^{N_b} \left(\underbrace{\frac{1}{4} - S_{i(b)}^z S_{j(b)}^z}_{H_{0,b}} \underbrace{- \frac{1}{2} \left(S_{i(b)}^+ S_{j(b)}^- + S_{i(b)}^- S_{j(b)}^+ \right)}_{H_{1,b}} \right) + \left\{ \frac{N_b}{4} \right\} \tag{4.4}$$

Hierbei fügen wir $H_{0,b}$ eine zusätzliche Konstante $1/4$ ein, um die Eigenwerte des Produktoperators $-S_{i(b)}^z S_{j(b)}^z$ von $\pm 1/4$ auf $\{0,\ 1/2\}$, was für spätere Berechnungen bequemer ist.

4.1.2 Reihenentwicklung

Wie beim klassischen Ising Modell im Abschnitt 3.1.2 versuchen wir nun auch, die Zustandssumme durch einen Monte-Carlo Ansatz anzunähern, anstatt den gesamten Zustandsraum abtasten zu müssen. Der obere Ansatz ist hier jedoch nicht hilfreich, weil wir die Energie bzw. den Hamiltonian für den Boltzmannfaktor (in der Zustandssumme) nicht berechnen können. Deshalb schlagen wir einen anderen Weg ein:

Die quantenmechanische Zustandssumme

$$Z = \operatorname{tr} e^{-\beta H} \tag{4.5}$$

ist über Spur des „Boltzmann-Operators“ definiert. Schreiben wir die Spur mit der Basis $|\alpha\rangle$ und verwenden für die Exponentialfunktion die Reihendarstellung, ergibt sich

$$Z = \sum_{n=0}^{\infty} \frac{(-\beta)^n}{n!} \sum_{\alpha} \langle \alpha \mid H^n \mid \alpha \rangle \tag{4.6}$$

$$= \sum_{n=0}^{\infty} \frac{(-\beta)^n}{n!} \sum_{\alpha} \langle \alpha \mid \left(-\sum_{b=1}^{N_b} H_{0,b} - H_{1,b} \right)^n \mid \alpha \rangle \ , \tag{4.7}$$

wobei wir die obige Gl. 4.4 für den Hamiltonian einsetzen.

An dieser Stelle führen wir die Potenzierung der Hamiltonians explizit durch Ausmultiplizieren aus und erhalten dadurch Hamiltonoperatorketten der Länge n, sogenannte Operatorstrings. Jedes Kettenglied ist hierbei entweder diagonal oder off-diagonal und gehört zu einer bestimmten Koppelung $\langle i, j \rangle$ mit Index b. Da jede mögliche Kombination von n Operatoren vorkommt, haben wir insgesamt $(2N_b)^n$ Operatorstrings. Um diese geeignet zu verwalten, definieren wir die Menge aller Operatorstrings der Länge n als $\{S_n\}$ und ordnen jedem String je eine Funktion für die beiden Indizes $a(p) \in \{0, 1\}$ und $b(p) \in \{1, \ldots, N_b\}$ der Hamiltonoperatoren zu, welche den exakten Operator $H_{a(p),b(p)}$ an der Position p im String darstellen. Darüber hinaus hebt sich das erste Minuszeichen vor der Summe mit dem vor dem β weg und das Minuszeichen zwischen den Hamiltonoperatoren ziehen wir mit der Anzahl der off-diagonalen Operatoren n_1 vor das Matrixelement:

$$Z = \sum_{n=0}^{\infty} \frac{\beta^n}{n!} \sum_{\alpha} \sum_{\{S_n\}} (-1)^{n_1} \langle \alpha \mid \prod_{p=0}^{n-1} H_{a(p),b(p)} \mid \alpha \rangle \tag{4.8}$$

Nun betrachten wir noch die unendliche Summe über n. Diese schneiden wir bei $n = L$ ab (eine Fehlerrechnung folgt später) und bringen alle Operatorstrings mit $n < L$ auf die Länge L, indem wir $L - n$ Einheitsmatrizen in sie einfügen, die wir sinnvollerweise $H_{0,0}$ nennen. Dies führt zu nur noch **einer** Menge von Operatorstrings S_L, in die die kürzeren integriert wurden. Da es aber $\binom{L}{n}$ Möglichkeiten gibt die Einheitsmatrizen einzufügen, müssen wir zusätzlich durch diese Vielfachheit teilen, da ein ehemaliger S_n Operatorstring auch zukünftig nur einfach in die Zustandssumme eingehen soll:

$$Z = \sum_{\{S_L\}} \frac{\beta^n (-1)^{n_1} (L-n)!}{L!} \sum_{\alpha} \langle \alpha \mid \prod_{p=0}^{L-1} H_{a(p),b(p)} \mid \alpha \rangle \tag{4.9}$$

n ist nun nicht mehr die Stringlänge, sondern die Anzahl der Operatoren ungleich $H_{0,0}$.

Wir wollen nun die Wirkung des Operatorstrings auf die Basis $|\alpha\rangle$ betrachten: Diese *propagierten Zustände*

$$|\alpha(Q)\rangle = \prod_{p=0}^{Q-1} H_{a(p),b(p)} \mid \alpha \rangle \tag{4.10}$$

sind neue Basiszustände, sie ergeben sich also nicht aus der Superposition von anderen Zuständen. Explizit ist die Wirkung eines diagonalen Operators auf einen Basiszustand

$$H_{0,b} \mid \ldots \uparrow_{i(b)} \ldots \uparrow_{j(b)} \ldots\rangle = \frac{1}{4} - (\frac{1}{2} \cdot \frac{1}{2}) = 0 \ , \tag{4.11}$$

$$H_{0,b} \mid \ldots \downarrow_{i(b)} \ldots \downarrow_{j(b)} \ldots\rangle = \frac{1}{4} - (-\frac{1}{2} \cdot -\frac{1}{2}) = 0 \ , \tag{4.12}$$

$$\langle \ldots \uparrow_{i(b)} \ldots \downarrow_{j(b)} \ldots \mid H_{0,b} \mid \ldots \uparrow_{i(b)} \ldots \downarrow_{j(b)} \ldots\rangle = \frac{1}{4} - (\frac{1}{2} \cdot -\frac{1}{2}) = \frac{1}{2} \ , \tag{4.13}$$

$$\langle \ldots \downarrow_{i(b)} \ldots \uparrow_{j(b)} \ldots \mid H_{0,b} \mid \ldots \downarrow_{i(b)} \ldots \uparrow_{j(b)} \ldots\rangle = \frac{1}{4} - (-\frac{1}{2} \cdot \frac{1}{2}) = \frac{1}{2} \tag{4.14}$$

und die Wirkung eines off-diagonalen Operators auf einen Basiszustand

$$H_{1,b} \mid \ldots \uparrow_{i(b)} \ldots \uparrow_{j(b)} \ldots\rangle = \frac{1}{2}(0+0) = 0 \ , \tag{4.15}$$

$$H_{1,b} \mid \ldots \downarrow_{i(b)} \ldots \downarrow_{j(b)} \ldots\rangle = \frac{1}{2}(0+0) = 0 \ , \tag{4.16}$$

$$\langle \ldots \downarrow_{i(b)} \ldots \uparrow_{j(b)} \ldots \mid H_{0,b} \mid \ldots \uparrow_{i(b)} \ldots \downarrow_{j(b)} \ldots\rangle = \frac{1}{2} \ , \tag{4.17}$$

$$\langle \ldots \uparrow_{i(b)} \ldots \downarrow_{j(b)} \ldots \mid H_{0,b} \mid \ldots \downarrow_{i(b)} \ldots \uparrow_{j(b)} \ldots\rangle = \frac{1}{2} \ . \tag{4.18}$$

Dieser Sachverhalt vereinfacht den Algorithmus deutlich. Beim späteren Sampling tragen also nur solche Operatorstrings bei, deren Operatoren auf nicht-parallele Spins wirken. Das Matrixelement einer solchen Operation ist außerdem immer $\frac{1}{2}$ (dies war der Grund für die Konstante 1/4 in $H_{0,b}$ aus Gl. 4.4).

Das Gewicht für eine beitragende Konfiguration σ ist also

$$p(\sigma, S_L) = \left(\frac{\beta}{2}\right)^n \frac{(L-n)!}{L!} \tag{4.19}$$

wobei wir verwenden, dass n_1 auf Quadratgittern und Ketten immer gerade ist und deshalb wegfällt [San10].

4.1.3 Sampling

Wir wenden nun die **Monte-Carlo Methode** auf Operatorstrings anstatt Zuständen an: Zu Beginn der Simulation gehen wir von einem leeren Operatorstring und einer zufälligen Spinanordnung aus und führen wie in Kapitel 3 MC-Schritte aus. Diese sind unterteilt in ein Diagonal Update, welches diagonale Operatoren in den String ein- und ausbaut und in ein so genanntes off-diagonales Loop Update, welches diagonale zu off-diagonale Operatoren hin- und zurücktransformiert. Jedes Update muss allerdings darauf achten, dass die oben angegeben Beschränkungen nicht verletzt werden:

- Operatoren dürfen nicht auf parallele Spin-Paare wirken, ansonsten annihilieren sie den Zustand (lokale Bedingung).
- Die Periodizität $|\alpha\rangle = |\alpha(0)\rangle = |\alpha(L)\rangle$ des Algorithmus' muss gewahrt werden.

Das Sampling kann durch Graphiken wie 4.1 veranschaulicht werden. Die Anfangskonfiguration $|\alpha(0)\rangle$ steht hierbei unten und propagiert durch den Operatorstring in den Endzustand $|\alpha(L)\rangle = |\alpha(0)\rangle$. Operatoren sitzen jeweils auf zwei „Spin-Bahnen“ und lassen diese entweder unberührt (diagonale Operatoren, weiß dargestellt) oder vertauschen deren Ausrichtung (off-diagonale Operatoren, schwarz dargestellt). Reihen ohne Operatoren signalisieren einen $H_{0,0}$ Operator, also eine Einheitsmatrix.

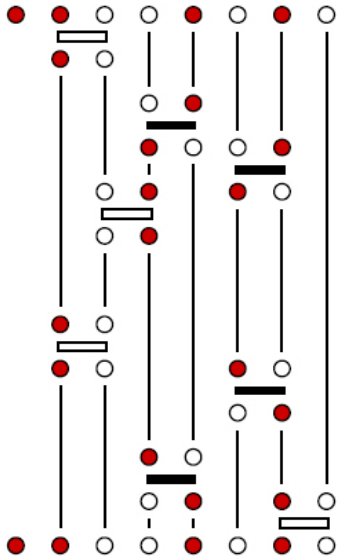


Abbildung 4.1: **Visualisierung des Operatorstrings:** Die Kreise ganz unten und ganz oben stellen die Spinkonfiguration $|\alpha(0)\rangle$ bzw. $|\alpha(L)\rangle$ dar. Dazwischen liegen diagonale (weiß) und off-diagonale Operatoren (schwarz). Reihen ohne Operatoren signalisieren einen $H_{0,0}$ Operator; *Quelle:* [San10]

Diagonales Update

Für das Einfügen von diagonalen Operatoren in den String muss darauf geachtet werden, dass dieser nicht an parallele Spins angelegt wird. Die Periodizität hingegen wird durch die Aktion nicht gestört, da diagonale Operatoren den Zustand nicht verändern. Das Entfernen von diagonalen Operatoren aus dem Operatorstring ist generell unproblematisch.

Wir legen nun unsere Übergangswahrscheinlichkeiten $\boldsymbol{W}$ für das Einfügen eines Operators an einem Platz p gemäß des **Metropolis Algorithmus'** fest (Gl. 2.6):

$$W_{\nu\sigma,\ \text{Einfügen}} = \begin{cases} \frac{p(\sigma,S_L)\cdot N_b}{p(\nu,S_L)} = \frac{\beta N_b}{2(L-n)} & p(\sigma,S_L)\cdot N_b < p(\nu,S_L) \\ 1 & p(\sigma,S_L)\cdot N_b \geq p(\nu,S_L) \end{cases} \tag{4.20}$$

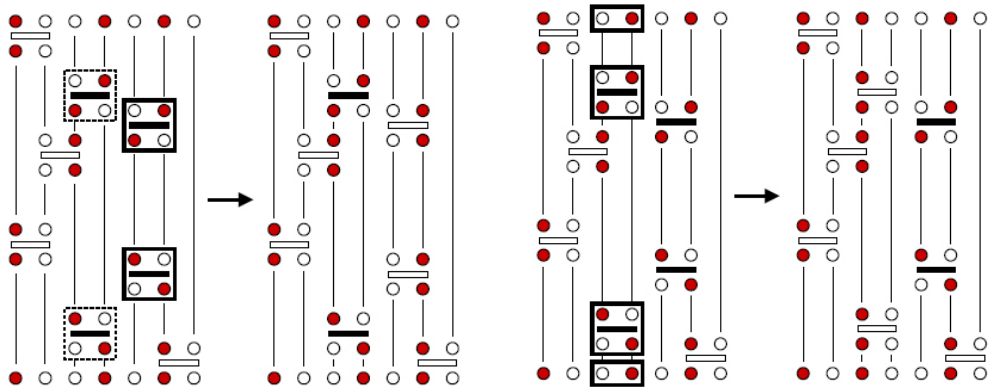
Analog erhält man beim Löschens eines Operators (ersetzen mit der Einheitsmatrix)

$$W_{\nu\sigma,\ \text{Entfernen}} = \begin{cases} \frac{p(\sigma,S_L)\cdot N_b}{p(\nu,S_L)} = \frac{2(L-n+1)}{\beta N_b} & p(\sigma,S_L)\cdot N_b < p(\nu,S_L) \\ 1 & p(\sigma,S_L)\cdot N_b \geq p(\nu,S_L)\ . \end{cases} \tag{4.21}$$

Die Eigenschaft *Detailed Balance* kann sofort durch Einsetzen in 2.4 verifiziert werden.

Off-Diagonales Loop Update

Beim Austausch von diagonalen und off-diagonalen Operatoren ist der Sachverhalt etwas komplizierter, denn dadurch wird ein Vertauschen von Spinrichtungen (durch den off-diagonalen Operator) hinzugefügt bzw. entfernt. Es ist selbstverständlich, dass unter der Berücksichtigung der Periodizität also mindestens zwei off-diagonale Operatoren in solch eine Aktion involviert sein müssen. Befinden sich zwischen diesem Operator-Paar allerdings weitere Operatoren, kann es zu einer Regelverletztung der lokalen Bedingung kommen, da ein Ändern der Zustände zwischen dem Operatorpaar den dazwischen liegenden beeinflusst (siehe Abb. 4.2a). In Abb. 4.2b wird eine mögliche Lösung dieses Problems dargestellt; hierbei wird nicht der Zustand zwischen dem Operatorenpaar verändert, sondern der außerhalb (inklusive dem Anfangs- und Endzustand).

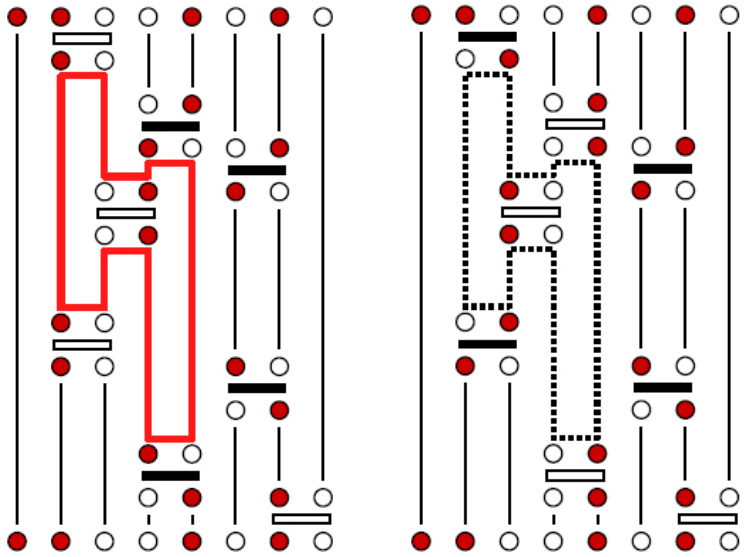


(a) **Problematischer (gestrichelt) und Unproblematischer Fall (durchgezogen)**

(b) **Ausweg über die Modifikation des Anfangszustand**

Abbildung 4.2: Mögliche Probleme bzgl. verletzte Bedingungen beim Off-Diagonalen Update. In (a) sehen wir, wie das durchgezogen marktierte Update unproblematisch durchgeführt, wobei das gestrichelte Update den Operator zwischen dem Paar so beeinflussen würde, dass dieser die lokale Bedingung nicht mehr erfüllt. In (b) sehen wir einen möglichen Ausweg, da hier Zustände zwischen den Operatoren nicht geändert werden, sondern der Anfangs- bzw. Endzustand; *Quelle:* [San10]

Eine andere Möglichkeit wäre natürlich, den zweiten (links) anliegenden Spin des involvierten mittleren Operators auch zu ändern, sodass die lokale Bedingung insgesamt wieder erfüllt ist. Dies beeinflusst aber wieder andere Operatoren, etc.. Im Endeffekt versucht man also, alle sogenannten Loops (siehe Abb. 4.3a), d.h. abgeschlossene Wege durch den Operatorstring, zu finden, da man diese dann wie in Abb. 4.3b unabhängig von einander flippen kann (Loop Update), wobei Operatoren die ganz in der Loop liegen hin- und sogleich wieder zurückgeflippt werden. Der Ausweg in Abb. 4.2b ist in dieser Lösung inbegriffen, da Loops auch über den periodischen Rand hinaus führen können.



(a) **Operatorstring mit eingezeichneter Loop**

(b) **Operatorstring nach dem Flippen der Loop**

Abbildung 4.3: Loop im Operatorstring. (a) bezieht sich noch auf den Ausgangszustand, (b) ergiebt sich nach dem flippen der in (a) angegeben Loop. Operatoren derer beider Seiten am selben Loop liegen werden nicht geflippt (hin und wieder zurückgeflippt); *Quelle:* [San10]

4.1.4 Formeln für die mittlere Energie und Wärmekapazität

Energie

Augehend von der Gl. 4.6 wollen wir nun eine Formel für die mittlere Energie pro Spin E/N herleiten,

$$Z = \sum_{n=0}^{\infty} \frac{(-\beta)^n}{n!} \sum_{\{\alpha\}_n} \langle \alpha_0 \mid H \mid \alpha_{n-1} \cdots \langle \alpha_1 \mid H \mid \alpha_0 \rangle \ , \tag{4.22}$$

wobei in die hintere Summe $n-1$ Summen über die Basis eingefügt wurden. Sodann ergiebt sich der Mittelwert von E/N mit:

$$\frac{E}{N} = \frac{1}{ZN} \sum_{n=0}^{\infty} \frac{(-\beta)^n}{n!} \sum_{\{\alpha\}_{n+1}} \langle \alpha_0 \mid H \mid \alpha_n \cdots \langle \alpha_1 \mid H \mid \alpha_0 \rangle \tag{4.23}$$

$$= \frac{1}{ZN} \sum_{n=1}^{\infty} \frac{(-\beta)^n}{n!} \frac{n}{-\beta} \sum_{\{\alpha\}_n} \langle \alpha_0 \mid H \mid \alpha_{n-1} \cdots \langle \alpha_1 \mid H \mid \alpha_0 \rangle \tag{4.24}$$

$$= \frac{1}{ZN} \sum_{n=0}^{\infty} \frac{(-\beta)^n}{n!} \frac{n}{-\beta} \sum_{\{\alpha\}_n} \langle \alpha_0 \mid H \mid \alpha_{n-1} \cdots \langle \alpha_1 \mid H \mid \alpha_0 \rangle \tag{4.25}$$

$$= -\frac{\langle n \rangle}{N\beta} \tag{4.26}$$

$$\left(\frac{E}{N}\right)_{\text{Real}} = -\frac{\langle n \rangle}{N\beta} + \left\{\frac{N_b}{4N}\right\} \tag{4.27}$$

Für die erste Gleichheit bemerken wir, dass die letzte Summe nun über ein H mehr läuft. Diese erste Umformung substituiert $n := n + 1$, die zweite fügt den $n = 0$-Term wieder ein, dies ist möglich, da er ohnehin 0 ergibt. Sodann kann erkannt werden, dass es sich bei dem vorliegenden Ausdruck um den Mittelwert von n handelt. Zu guter Letzt fügen wir noch die „verlorene“ Energie-Konstante von Gl. 4.4 hinzu.

Wärmekapazität

Die Wärmekapazität pro Spin erhält man dann über die Ableitung der mittleren Energie nach der Temperatur:

$$\frac{C}{N} = \frac{\partial_T E}{N} \tag{4.28}$$

$$= -\frac{1}{NT} \partial_T \langle n \rangle - \frac{\langle n \rangle}{N} \tag{4.29}$$

$$= \frac{\langle n^2 \rangle - \langle n \rangle^2 - \langle n \rangle}{N} \tag{4.30}$$

4.1.5 Cut-Off L

Da für $T \to \infty$ von $C \to 0$ ausgegangen werden kann, sehen wir, dass $\operatorname{var} n = \langle n \rangle$, das heißt, der T – n Graph fällt in beide Richtungen exponentiell ab. Nach [San10] kann für L also ein hinreichend höherer Wert als $\langle n \rangle$ verwendet werden (ca. $4/3 \cdot \langle n \rangle$). Dann erreicht n praktisch nie L.

Der Algorithmus muss L also für jeden MC-Schritt überprüfen und gegebenfalls anpassen.

4.2 Implementierung

Die Struktur der Anwendung ist ebenfalls wieder eng angelehnt an die Beschreibung in [San10]. Der Algorithmus ähnelt dem der klassischen MCS in Abschnitt 3.2 deutlich, verwendet allerdings andere Messformeln (Gl. 4.27 und 4.30) und einen anderen MC-Schritt, da er den Operatorstring inklusive Anfangskonfiguration sampled und nicht nur einzelne Konfigurationen.

4.2.1 Initialisierung

Wie bei der klassischen MCS benötigen wir zuerst die Eingabeparameter:

- Anzahl der Spins N
- Anzahl der Messungen R_1
- Temperatur des Systems T

Anschließend legen wir wieder ein boolsches Array der Länge N an, welches unseren Anfangszustand $|\alpha(0)\rangle$ enthält, und initialisieren es mit zufälligen Werten. Um den aktuellen Operatorstring abspeichern zu können, legen wir weiterhin ein Integer-Array s der Länge L an, welches für jeden Platz p im String die Art des Operators (diagonal, off-diagonal oder Einheitsmatrix) – also $a(p)$ und dessen Position im System $b(p)$ (Koppelung) speichert. Dies können wir gemeinsam in einer Ganzzahl speichern, wenn wir ausnutzen, dass $a \in \{0, 1\}$:

$$s(p) = a(p) + 2b(p) \tag{4.31}$$

Wir können die Informationen aus der Ganzzahl $s(p)$ wieder erhalten, wenn wir prüfen, ob

- $s(p) = 0 \Rightarrow$ Einheitsmatrix,
- even $s(p) \Rightarrow$ Diagonaler Operator,
- odd $s(p) \Rightarrow$ Off-diagonaler Operator.

Die Position $b(p)$ des Operators erhalten wir, wenn wir mittels einer ganzzahligen Division $b(p) = s(p)/2$. Außerdem speichern wir die Anzahl der Operatoren n, die nicht eine Einheitsmatrix darstellen, da diese Größe später zum Berechnen unserer Messgrößen verwendet wird.

4.2.2 Simulation

Für jeden MC-Schritt wird zuerst ein Diagonal Update durchgeführt und anschließend ein Off-Diagonal Loop Update. Abschließend wird der Cut-Off L nach Abschnitt 4.1.5 eventuell weiter noch oben gesetzt, um dem System genug Freiraum zum Einfügen weiterer Operatoren zu geben, d.h. wir verlängern den Operatorstring, um immer genug Einheitsmatrizen frei für diagonale Operatoren zu haben.

Diagonal Update

Bei jedem Diagonal Update wird für jeden Operatorplatz p, auf dem bereits ein diagonaler Operator sitzt, mittels einem Zufallstest entschieden, ob der Operator durch eine Einheitsmatrix ersetzt werden darf. Ist eine Zufallszahl kleiner als die Wahrscheinlichkeit aus Gl. 4.21, verringert man n um 1 und vermerkt im Operatorstring $s(p) = 0$.

Existiert noch kein Operator auf der Position, versucht man einen diagonalen Operator einzufügen: Die Koppelung b des Operators wird zufällig bestimmt. Wenn die beiden Spins an dieser Koppelung anti-parallel sind, wird mit einem ähnlichen Zufallstest wie oben mit der Wahrscheinlichkeit aus Gl. 4.20 entschieden, ob das Einfügen erfolgt. Als Konsquenz würden wir n um 1 erhöhen und $s(p) = 2b$ setzen. Um den bis zu p propagierten Zustand $|\alpha(p)\rangle$ schnell zu erhalten, schreiben wir ihn in jedem p-Schritt mit.

Trifft man auf einen Off-Diagonalen Operator, wird nichts am Operatorstring verändert, lediglich der propagierte Zustand wird angepasst (die beiden Spins tauschen ihren Zustand).

Off-Diagonal Loop Update

Um nun die durch das Diagonal Update eingefügten Operatoren auch in Off-Diagonale Operatoren zu transformieren (oder zurück), wird der Operatorstring nun systematisch nach Loops gescannt und für jeden Loop wird anschließend mit der Wahrscheinlichkeit von 50% entschieden, diesen gesamten Loop zu flippen.

Die Analyse des Operatorstrings Jeder Operator (der keine Einheitsmatrix ist) besitzt vier Vertizes, welche die Anknüpfungspunkte an die Spin-Bahnen (zwei oben, zwei unten) sind. Diese erhalten nach Abb. 4.4 je eine Typnummer $y \in \{0, 1, 2, 3\}$. Mithilfe dieses y und der Position des Operators im String p kann man jeden Vertex global mit der Zahl v indizieren:

$$v(y, p) = v + 4p \tag{4.32}$$

Von der Ganzzahl v kann man wie bei 4.31 wieder auf die speziellen Eigenschaften schließen.

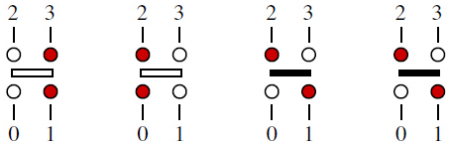


Abbildung 4.4: **Mögliche Verwendung der Operatoren mit den Typnummern der Vertizes**; *Quelle:* [San10]

Nun gehen wir jede Position p im String durch und verknüpfen Vertizes, die sich auf einer Spin-Bahn gegenüberliegen. Die Verknüpfungen stellen also die dicken Linien dar, z.B. wie in Abb. 4.3a, die die Operatoren verbinden. D.h. Vertizes stellen die Ecken eines Loop-Gebiets dar, diese Verknüpfungen und die Operatoren die Kanten. Die Verknüpfung wird in einem Verknüpfungs-Array x gespeichert. Zu beachten ist, dass Loops durchaus auch über den periodischen Rand hinweg möglich sind!

Flippen der Loops Dieses wird nun Stück für Stück durchgeführt: Für jede Loop wird mit einer Wahrscheinlichkeit 50% entschieden, ob diese geflippt werden soll. Ist dies der Fall, geht man auf den Kanten des Loop-Gebiets von Operator zu Operator und flippt einen jeden. Operatoren, die mitten in einer Loop liegen, werden also nicht verändert. Wird eine Loop geflippt, die sich über die periodischen Ränder hinaus erstreckt, muss anschließend der Anfangszustand demensprechend verändert werden.

4.2.3 Analyse

Die abgespeicherten Werte für n werden nach der Simulation verwendet, um sie – wie in Abschnitt 2.6 erklärt – zu analysieren. Ihre Anwendung zur Berechnung der Größen **Energie** und **Wärmekapazität** werden in den Gleichungen 4.27 und 4.30 beschrieben.

4.2.4 Quellcode

Der vom Author geschriebene C++ Quellcode ist im Anhang A zu finden. Folgende Dateien sind für diese quantenmechanische Simulation relevant:

- A.1: Hauptprogramm SIM
- A.2: Abstrakte Gitterklasse
- A.3: 1D Kette mit offenen Randbedingungen
- A.4: 1D Kette mit periodischen Randbedingungen
- A.5: 2D Gitter mit periodischen Randbedingungen
- A.6: Abstrakte Algorithmusklasse
- A.9: SSE Algorithmus
- A.10: Abstrakte Analyseklasse
- A.17: Analyse für die Energie (SSE)
- A.18: Analyse für die Wärmekapazität (SSE)

4.3 Ergebnisse und Diskussion

Im Folgenden sind die Messergebnisse der SSE-Simulationen aufgeführt. Bis auf die explizite Erwähnung im ersten Unterabschnitt wurde auf eine Fehlerdarstellung in den Diagrammen verzichtet, da die Fehler ihrer (kleinen) Größe wegen nicht darstellbar waren.

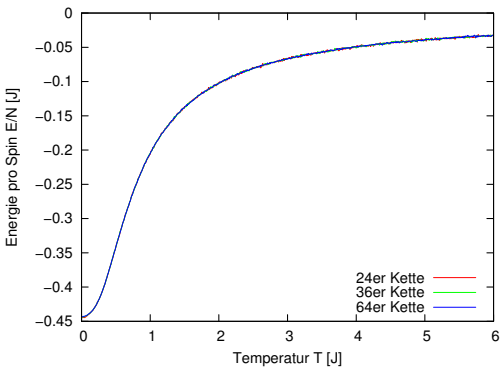
4.3.1 Heisenbergkette mit periodischen Randbedingungen

Auf den Abbildungen 4.5a und 4.5b sieht man die mittlere Energie und die mittlere Wärmekapazität für verschieden große Systeme (periodische Ketten) aufgetragen. Diese Kurven verlaufen stets übereinander, da im 1-dimensionalen Fall noch kein Phasenübergang auftritt. Das System befindet sich also entweder nahe dem Grundzustand oder besitzt bereits eine so geringe Korrelationslänge, dass für verschiedene Größen keine Effekte auftreten (vgl. Erörterung in 3.3.1). Die jeweils gleiche Grundzustandsenergie von -0.4432 stimmt sehr gut mit dem in [Gro04] auf Seite 58 angegeben Wert überein.

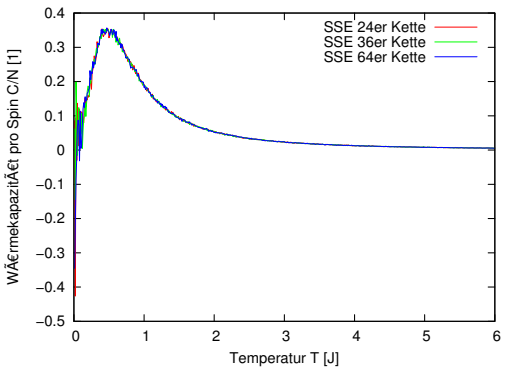
Wir betrachten nun noch explizit den absoluten Fehler (Standardabweichung) in Abb. 4.6: Mit steigender Systemgröße tritt also ein immer stärker Effekt zu Tage; bei hohen Temperaturen gibt es wesentlich mehr mögliche Konfigurationen. All diese Konfigurationen haben eine ähnliche Energie und müssen gesampelt werden. Bei $T = 0$ muss das System nur zum Grundzustand finden.

4.3.2 Heisenberggitter mit periodischen Randbedingungen

Nach Korrolar 1.11 aus [PS99] besitzt das 2-dimensionale Gitter im Gegensatz zum 1D-Fall einen Phasenübergang, d.h. zur Temperatur T_c hin divergiert die Korrelationslänge und damit



(a) **Mittelwerte der Energie**



(b) **Mittelwerte der Wärmekapazität**

Abbildung 4.5: Mittelwerte der Energie und Wärmekapazität für verschiedene Systemgrößen einer periodischen Kette bei 100000 Messpunkten pro Temperaturpunkt; *Quelle:* Eigenwerk

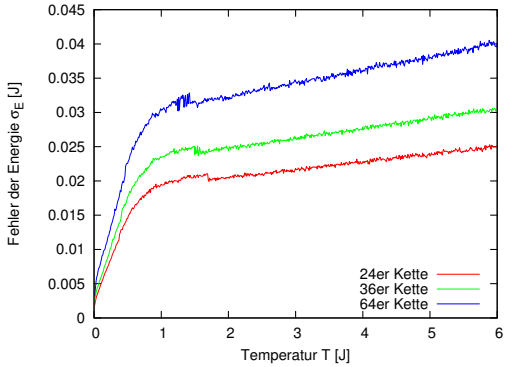


Abbildung 4.6: **Absoluter Fehler der Energie** für verschiedene Systemgrößen einer periodischen Kette bei 100000 Messpunkten pro Temperaturpunkt; *Quelle:* Eigenwerk

die Größe der Weißschen Bezirke (Cluster), so dass in der Umgebung um diesen kritischen Punkt die Größe des Systems eine gravierende Rolle spielt. Beide Grundzustandsenergien, -0.67887 und -0.702, werden durch [San97] bestätigt.

4.3.3 Vergleich verschiedener Modelle / Exakte Diagonalisierung

In den Abbildungen 4.8a und 4.8b sind die Energie und die Wärmekapazität für die drei betrachteten Modelle (Kette mit offenen, Kette mit periodischen und Gitter mit periodischen Randbedingungen) aufgetragen. Die exakte, dunklere Linie beschreibt die Werte der ED-Methode *(Exakte Diagonalisierung)*.

Die Energiekurven zeigen im 1-dimensionalen Fall keinen sichtbaren Unterschied zwischen den Methoden SSE und ED, wohingegen die SSE-Werte für das Gitter eine gut sichtbare höhere Ungenauigkeit aufweisen. Die Differenz der Grundzustandsenergien erklären sich wohl Hauptsächlich durch die größere Anzahl von Koppelungen pro Spin, qualitativ zeigen die Kurven allerdings ebenfalls kaum Unterschiede.

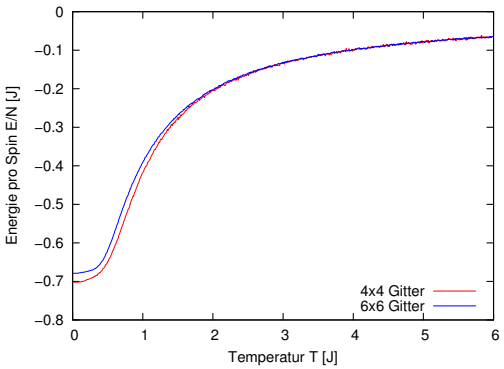
Die Ableitung der Energie nach der Temperatur, die Wärmekapazität, ist wie erwartet deutlich ungenauer. Besonders in den Temperaturen nahe 0 spielen arithmetische Fehler der Implementierung (Maschinengenauigkeit) eine große Rolle, da dort große Zahlen subtrahiert werden. Auffallend ist, dass die beiden periodischen Modelle diesselbe maximale Wärmekapazität besitzen.

An dem Beispiel der 4 Spin langen Heisenbergkette mit offenen Randbedingungen, wollen wir uns nun noch die Verwendung der Exakten Diagonalisierung veranschaulichen [Mab06]:

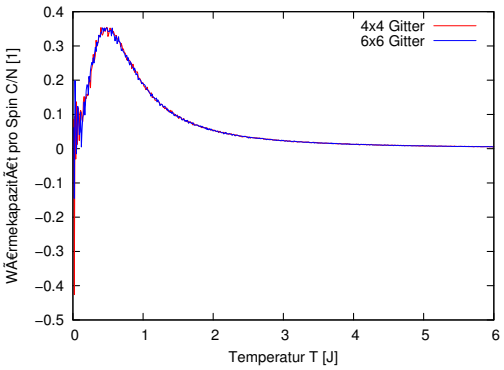
Da der Hamiltonian jeden Unter-Zustandsraum Ω_i aller Basiszustände diesselbe Anzahl i aufwärtsgerichteter Spins (siehe [Spi]) stabil lässt (Erhaltung der Summe aller S_j^z), stellt er in einer geeignet sortierten Basis eine Blockmatrix dar:

$$H = \begin{pmatrix} \left\{0.75\right\} & & & & 0 \\ & \left\{\begin{matrix} 0.25 & 0.5 & 0 & 0 \\ 0.5 & -0.25 & 0.5 & 0 \\ 0 & 0.5 & -0.25 & 0.5 \\ 0 & 0 & 0.5 & 0.25 \end{matrix}\right\} & & & \\ & & \left\{\begin{matrix} 0.25 & 0.5 & 0 & 0 & 0 & 0 \\ 0.5 & -0.75 & 0.5 & 0.5 & 0 & 0 \\ 0 & 0.5 & -0.25 & 0 & 0.5 & 0 \\ 0 & 0.5 & 0 & -0.25 & 0.5 & 0 \\ 0 & 0 & 0.5 & 0.5 & -0.75 & 0.5 \\ 0 & 0 & 0 & 0 & 0.5 & 0.25 \end{matrix}\right\} & & \\ & & & \left\{\begin{matrix} 0.25 & 0.5 & 0 & 0 \\ 0.5 & -0.25 & 0.5 & 0 \\ 0 & 0.5 & -0.25 & 0.5 \\ 0 & 0 & 0.5 & 0.25 \end{matrix}\right\} & \\ 0 & & & & \left\{0.75\right\} \end{pmatrix}$$

Die Blöcke können nun einzeln diagonalisiert werden, was einen deutlichen Performancegewinn gegenüber einer gesamten Diagonalisierung darstellt. Die Eigenwerte der obigen Matrix sind {0.75, 0.75, 0.75, 0.75, 0.75, 0.457107, 0.457107, 0.457107, 0.116025, -0.25, -0.25, -0.25, -0.957107, -0.957107, -0.957107, -1.61603}. Mithilfe dieser können nun z.B. die Boltzmanngewichte bzw. die Zustandssumme berechnet werden.

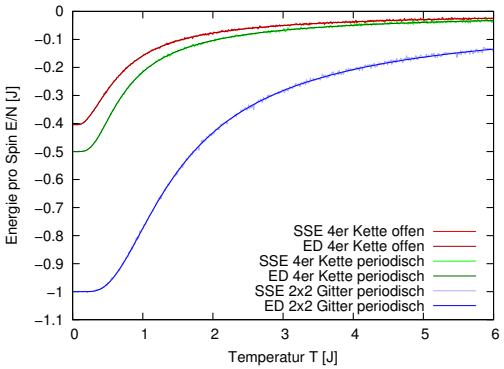


(a) **Mittelwerte der Energie**

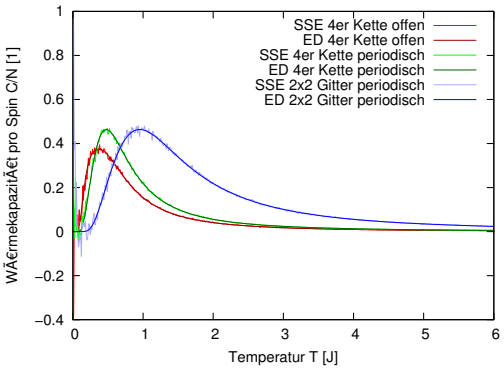


(b) **Mittelwerte der Wärmekapazität**

Abbildung 4.7: Mittelwerte der Energie und Wärmekapazität für verschiedene Systemgrößen eines periodischen Gitters bei 100000 Messpunkten pro Temperaturpunkt; *Quelle:* Eigenwerk



(a) **Mittelwerte der Energie**



(b) **Mittelwerte der Wärmekapazität**

Abbildung 4.8: Mittelwerte der Energie und Wärmekapazität für die verschiedenen Modelle 1D-Offen, 1-Periodisch und 2D-Periodisch mit der Spinanzahl 4 bei 100000 Messpunkten pro Temperaturpunkt; *Quelle:* Eigenwerk

Kapitel 5

Zusammenfassung

Die **Stochastic Series Expansion** stellt ein mächtiges Werkzeug für das Sampling von Operatorstrings innerhalb eines quantenmechanischen Lösungsansatzes dar. Sie ist für größere 2-dimensionale Systeme der schnellste Weg, um die typischen, thermodynamischen Größen zu messen und wird hierfür an mehreren Instituten erfolgreich eingesetzt. Der Algorithmus ist relativ leicht zu implementieren und arbeitet äußerst speichersparend.

Im Rahmen dieser Arbeit wurde ein Simulationsprogramm geschrieben, welches nicht nur ein bloßes SSE Modul enthält, sondern die Möglichkeit anbietet, die SSE-Daten für kleine Systeme mit ED **Exakt Diagonalisation** zu überprüfen, darüber hinaus kann ein Zusatzmodul für das numerische Lösen des klassischen Ising-Modells eingesetzt werden. Da das Sofware-Projekt objekt-orientiert ausgelegt ist, können beliebige Komponenten, wie mit einem Baukastensystem zusammengestellt werden. Es ist dadurch sehr flexibel.

Die durchgeführten Messungen bestätigten stets die Theorie und liefern besitzen nur einen sehr geringen Fehler.

Physikalische Sachverhalte werden an mehreren Beispielen/Bildern erklährt und die Abschnitte „Methode“ in den beiden Projektskapitel 3 und 4 führen ausführliche Beschreibungen der 3 Algorithmen an.

Anhang A

Quellcode

A.1 Hauptprogramm SIM

```
#include "Classes/Lattice/Open1DLattice.cpp"
#include "Classes/Lattice/Periodic1DLattice.cpp"
#include "Classes/Lattice/Periodic2DLattice.cpp"

#include "Classes/Algorithm/SSEAlgorithm.cpp"
#include "Classes/Algorithm/EDAlgorithm.cpp"
#include "Classes/Algorithm/ISINGAlgorithm.cpp"

#include <cstdlib>
#include <iostream>

int main(int argc, char *argv[]) {

  try {

    if(argc != 8) throw "[SIM] Error: Please specify 7 parameters (Size,
        Lattice-Index, Measure-Count, Algorithm-Index, Start-Temperature,
         End-Temperature, Temperature-Step)\n              - Size can be
        any positive integer\n              - Lattice-Index:\n
                        0: Open-1D\n                1: Periodic-1D\n
                        2: Periodic-2D\n             - Measure-Count
        should be a positive long integer - ED of course needs no Measure
        -Count (=0)\n             - Algorithm-Index:\n
        0: ED  - Exact Diagonalization\n                 1: ISING  -
        Classical Ising simulation\n                2: SSE - Stochastic
        Series Expansion\n             - Temperatures should be positive
         floats (use the dot as delimiter)";

    int size               = atoi(argv[1]);
    int latticeIndex       = atoi(argv[2]);
    long measureCount      = atoi(argv[3]);
    int algorithmIndex     = atoi(argv[4]);
    double startTemperature = atof(argv[5]);
    double endTemperature   = atof(argv[6]);
    double temperatureStep  = atof(argv[7]);

    if(size <= 0) throw "[SIM] Error: The Size has to be positive";

```

```
28          AbstractLattice* lattice;
29          const char* latticeLabel;
30
31          switch(latticeIndex) {
32
33            case 0:
34              lattice = new Open1DLattice(size);
35              latticeLabel = "Open-1D";
36              break;
37
38            case 1:
39              lattice = new Periodic1DLattice(size);
40              latticeLabel = "Periodic-1D";
41              break;
42
43            case 2:
44              lattice = new Periodic2DLattice(size);
45              latticeLabel = "Periodic-2D";
46              break;
47
48            default:
49              throw "[SIM] Error: The Lattice-Index has to be within [0;2]";
50
51          }
52
53          if(measureCount <= 0) throw "[SIM] Error: The Measure-Count has to be
                positive";
54
55          AbstractAlgorithm* algorithm;
56          const char* algorithmLabel;
57
58          switch(algorithmIndex) {
59
60            case 0:
61              algorithm = new EDAlgorithm(lattice, measureCount);
62              algorithmLabel = "ED";
63              break;
64
65            case 1:
66              algorithm = new ISINGAlgorithm(lattice, measureCount);
67              algorithmLabel = "ISING";
68              break;
69
70            case 2:
71              algorithm = new SSEAlgorithm(lattice, measureCount);
72              algorithmLabel = "SSE";
73              break;
74
75            default:
76              throw "[SIM] Error: The Algorithm-Index has to be within [0;2]";
77
78          }
79
80          if(startTemperature <  0) throw "[SIM] Error: The Start-Temperature
                has to be positive or zero";
81          if(endTemperature   <= 0) throw "[SIM] Error: The End-Temperature has
                to be positive";
82          if(temperatureStep  <= 0) throw "[SIM] Error: The Temperature-Step
                has to be positive";
83          if(startTemperature > endTemperature) std::swap(startTemperature,
                endTemperature);
84
```

```
85          printf("#\n");
86          printf("# SIM - DATA\n");
87          printf("#\n");
88          printf("# ----------------------------------------\n");
89          printf("#\n");
90          printf("# Size                  = %+29.29i\n", size);
91          printf("# Lattice               = %s\n", latticeLabel);
92          printf("# Measure-Count         = %+29.29li\n", measureCount);
93          printf("# Algorithm             = %s\n", algorithmLabel);
94          printf("# Start-Temperature     = %+30.23e\n", startTemperature);
95          printf("# End-Temperature       = %+30.23e\n", endTemperature);
96          printf("# Temperature-Step      = %+30.23e\n", temperatureStep);
97          printf("#\n");
98          printf("# %-27s | %-28s | %-28s | %-28s | %-28s | %-28s | %-28s |
                %-28s | %-28s | %-28s | %-28s | %-28s | %-28s | %-28s | %-28s |
                %-28s | %-28s | %-28s | %-28s | %-28s | %-28s | %-28s | %-28s\n",
99              "Size",
100             "Lattice-Index",
101             "Measure-Count",
102             "Algorithm-Index",
103             "Temperature",
104             "Average Energy",
105             "Error Of Energy",
106             "ACorTime Of Energy",
107             "Average Heat Capacity",
108             "Error Of Heat Capacity",
109             "ACorTime Of Heat Capacity",
110             "Average Magnetisation",
111             "Error Of Magnetisation",
112             "ACorTime Of Magnetisation",
113             "Average Susceptibility",
114             "Error Of Susceptibility",
115             "ACorTime Of Susceptibility",
116             "Average A Magnetisation",
117             "Error Of A Magnetisation",
118             "ACorTime Of A Magnetisation",
119             "Average A Susceptibility",
120             "Error Of A Susceptibility",
121             "ACorTime Of A Susceptibility");
122
123         for(double temperature = endTemperature; temperature >
                startTemperature + (temperatureStep / 2); temperature -=
                temperatureStep) {
124
125           algorithm->runTemperatureRound(temperature);
126
127           printf("%+29.29i|%+29.29i|%+29.29li|%+29.29i|%+30.23e|%+30.23e
                    |%+30.23e|%+29.29li|%+30.23e|%+30.23e|%+29.29li|%+30.23e
                    |%+30.23e|%+29.29li|%+30.23e|%+30.23e|%+29.29li|%+30.23e
                    |%+30.23e|%+29.29li|%+30.23e|%+30.23e|%+29.29li\n",
128                size,
129                latticeIndex,
130                measureCount,
131                algorithmIndex,
132                temperature,
133                algorithm->getAverageEnergy(),
134                algorithm->getErrorOfEnergy(),
135                algorithm->getAutoCorrelationTimeOfEnergy(),
136                algorithm->getAverageHeatCapacity(),
137                algorithm->getErrorOfHeatCapacity(),
138                algorithm->getAutoCorrelationTimeOfHeatCapacity(),
139                algorithm->getAverageMagnetisation(),
```

```
140                 algorithm->getErrorOfMagnetisation(),
141                 algorithm->getAutoCorrelationTimeOfMagnetisation(),
142                 algorithm->getAverageSusceptibility(),
143                 algorithm->getErrorOfSusceptibility(),
144                 algorithm->getAutoCorrelationTimeOfSusceptibility(),
145                 algorithm->getAverageAbsoluteMagnetisation(),
146                 algorithm->getErrorOfAbsoluteMagnetisation(),
147                 algorithm->getAutoCorrelationTimeOfAbsoluteMagnetisation(),
148                 algorithm->getAverageAbsoluteSusceptibility(),
149                 algorithm->getErrorOfAbsoluteSusceptibility(),
150                 algorithm->getAutoCorrelationTimeOfAbsoluteSusceptibility())
                        ;
151
152         std::cerr << "[SIM] Info: Finished, Size=" << size << ", Lattice="
                << latticeLabel << ", Measure-Count=" << measureCount << ",
                Algorithm=" << algorithmLabel << ", Temperature=" <<
                temperature << std::endl;
153
154       }
155
156       delete algorithm;
157       delete lattice;
158
159       return EXIT_SUCCESS;
160
161     } catch(const char *message) {
162
163       std::cerr << message << std::endl;
164       return EXIT_FAILURE;
165
166     }
167
168   };
```

Listing A.1: SIM.cpp

A.2 Gitter Klassen

A.2.1 Abstrakte Gitterklasse

```
1   #ifndef CLASS_ABSTRACTLATTICE
2   #define CLASS_ABSTRACTLATTICE
3
4   class AbstractLattice {
5
6     protected:
7
8       int n;
9
10    public:
11
12      AbstractLattice(int n_parameter) {
13
14        n = n_parameter;
15
16      };
17
```

```
18        int getN() {
19
20          return n;
21
22        };
23
24        virtual int *getNeighbours(int i) = 0;
25
26        virtual int getNb() = 0;
27        virtual int getI1(int b) = 0;
28        virtual int getI2(int b) = 0;
29
30    };
31
32    #endif
```

Listing A.2: Classes/Lattice/AbstractLattice.cpp

A.2.2 1D Gitter mit offenen Randbedingungen

```
1     #ifndef CLASS_OPEN1DLATTICE
2     #define CLASS_OPEN1DLATTICE
3
4     #include "AbstractLattice.cpp"
5
6     class Open1DLattice : public AbstractLattice {
7
8       public:
9
10        Open1DLattice(int n_parameter) : AbstractLattice(n_parameter) {};
11
12        int *getNeighbours(int i) {
13
14          int *neighbours;
15
16          if(i == 0) {
17
18            neighbours = new int(2);
19            neighbours[0] = 1;
20            neighbours[1] = i + 1;
21
22          } else if(i == n - 1) {
23
24            neighbours = new int(2);
25            neighbours[0] = 1;
26            neighbours[1] = i - 1;
27
28          } else {
29
30            neighbours = new int(3);
31            neighbours[0] = 2;
32            neighbours[1] = i - 1;
33            neighbours[2] = i + 1;
34
35          }
36
37          return neighbours;
38
39        };
```

```
40
41          int getNb() {
42
43            return n - 1;
44
45          };
46
47          int getI1(int b) {
48
49            return b - 1;
50
51          };
52
53          int getI2(int b) {
54
55            return b;
56
57          };
58
59      };
60
61      #endif
```

Listing A.3: Classes/Lattice/Open1DLattice.cpp

A.2.3 1D Gitter mit periodischen Randbedingungen

```
1   #ifndef CLASS_PERIODIC1DLATTICE
2   #define CLASS_PERIODIC1DLATTICE
3
4   #include "AbstractLattice.cpp"
5
6   class Periodic1DLattice : public AbstractLattice {
7
8     public:
9
10      Periodic1DLattice(int n_parameter) : AbstractLattice(n_parameter) {};
11
12      int getNb() {
13
14        return n;
15
16      };
17
18      int *getNeighbours(int i) {
19
20        int *neighbours = new int(3);
21        neighbours[0] = 2;
22        neighbours[1] = (i - 1 + n) % n;
23        neighbours[2] = (i + 1) % n;
24
25        return neighbours;
26
27      };
28
29      int getI1(int b) {
30
31        return b - 1;
32
```

```
33        };
34
35        int getI2(int b) {
36
37          return b % n;
38
39        };
40
41    };
42
43    #endif
```

Listing A.4: Classes/Lattice/Periodic1DLattice.cpp

A.2.4 2D Gitter mit periodischen Randbedingungen

```
1     #ifndef CLASS_PERIODIC2DLATTICE
2     #define CLASS_PERIODIC2DLATTICE
3
4     #include <cmath>
5
6     #include "AbstractLattice.cpp"
7
8     class Periodic2DLattice : public AbstractLattice {
9
10      protected:
11
12        int m;
13
14      public:
15
16        Periodic2DLattice(int n_parameter) : AbstractLattice(n_parameter) {
17
18          m = sqrt((double) n_parameter);
19          if(pow((int) m, 2) != n_parameter) throw "[Periodic2DLattice] Error
                : You choose a 2D lattice, please input a square number as n";
20
21        };
22
23        int *getNeighbours(int i) {
24
25          int *neighbours = new int[5];
26          neighbours[0] = 4;
27          neighbours[1] = i - m <  0     ? i - m + n : i - m;
28          neighbours[2] = i + m >= n     ? i + m - n : i + m;
29          neighbours[3] = i % m == 0     ? i + m - 1 : i - 1;
30          neighbours[4] = i % m == m - 1 ? i - m + 1 : i + 1;
31
32          return neighbours;
33
34        };
35
36        int getNb() {
37
38          return 2 * n;
39
40        };
41
42        int getI1(int b) {
```

```
43
44          return (b - 1) / 2;
45
46        };
47
48        int getI2(int b) {
49
50          if(b & 1) {
51
52            return (((b - 1) / 2) + m) % n;
53
54          } else {
55
56            return ((((b - 1) / 2) + 1) % m) + ((((b - 1) / 2) / m) * m);
57
58          }
59
60        };
61
62    };
63
64    #endif
```

Listing A.5: Classes/Lattice/Periodic2DLattice.cpp

A.3 Algorithmus Klassen

A.3.1 Abstrakte Algorithmusklasse

```
1     #ifndef CLASS_ABSTRACTALGORITHM
2     #define CLASS_ABSTRACTALGORITHM
3
4     #include <sstream>
5     #include <string>
6
7     #include "../Lattice/AbstractLattice.cpp"
8
9     class AbstractAlgorithm {
10
11      protected:
12
13        AbstractLattice* lattice;
14        long measureCount;
15        long runCount;
16
17        double *energyMeasurements;
18        double *magnetisationMeasurements;
19        double *absoluteMagnetisationMeasurements;
20
21        double t;
22
23        double averageEnergy;
24        double errorOfEnergy;
25        long autoCorrelationTimeOfEnergy;
26
27        double averageHeatCapacity;
28        double errorOfHeatCapacity;
```

```
29        long autoCorrelationTimeOfHeatCapacity;
30
31        double averageMagnetisation;
32        double errorOfMagnetisation;
33        long autoCorrelationTimeOfMagnetisation;
34
35        double averageSusceptibility;
36        double errorOfSusceptibility;
37        long autoCorrelationTimeOfSusceptibility;
38
39        double averageAbsoluteMagnetisation;
40        double errorOfAbsoluteMagnetisation;
41        long autoCorrelationTimeOfAbsoluteMagnetisation;
42
43        double averageAbsoluteSusceptibility;
44        double errorOfAbsoluteSusceptibility;
45        long autoCorrelationTimeOfAbsoluteSusceptibility;
46
47      public:
48
49        AbstractAlgorithm(AbstractLattice* lattice_parameter, int
              measureCount_parameter) {
50
51          lattice = lattice_parameter;
52          measureCount = measureCount_parameter;
53          runCount = measureCount * 3 / 2;
54
55          energyMeasurements                  = new double[measureCount];
56          magnetisationMeasurements           = new double[measureCount];
57          absoluteMagnetisationMeasurements   = new double[measureCount];
58
59        };
60
61        ~AbstractAlgorithm() {
62
63          delete[] absoluteMagnetisationMeasurements;
64          delete[] magnetisationMeasurements;
65          delete[] energyMeasurements;
66
67        };
68
69        long getMeasureCount() {
70
71          return measureCount;
72
73        };
74
75        long getRunCount() {
76
77          return runCount;
78
79        };
80
81        virtual void runTemperatureRound(double t_parameter) {
82
83          t = t_parameter;
84
85          averageEnergy = 0;
86          errorOfEnergy = 0;
87          autoCorrelationTimeOfEnergy = 0;
88
89          averageHeatCapacity = 0;
```

```
90            errorOfHeatCapacity = 0;
91            autoCorrelationTimeOfHeatCapacity = 0;
92
93            averageMagnetisation = 0;
94            errorOfMagnetisation = 0;
95            autoCorrelationTimeOfMagnetisation = 0;
96
97            averageSusceptibility = 0;
98            errorOfSusceptibility = 0;
99            autoCorrelationTimeOfSusceptibility = 0;
100
101           averageAbsoluteMagnetisation = 0;
102           errorOfAbsoluteMagnetisation = 0;
103           autoCorrelationTimeOfAbsoluteMagnetisation = 0;
104
105           averageAbsoluteSusceptibility = 0;
106           errorOfAbsoluteSusceptibility = 0;
107           autoCorrelationTimeOfAbsoluteSusceptibility = 0;
108
109       };
110
111       double getTemperature() {
112
113         return t;
114
115       };
116
117       double getEnergyMeasurement(long time) {
118
119         return energyMeasurements[time];
120
121       };
122
123       double getMagnetisationMeasurement(long time) {
124
125         return magnetisationMeasurements[time];
126
127       };
128
129       double getAbsoluteMagnetisationMeasurement(long time) {
130
131         return absoluteMagnetisationMeasurements[time];
132
133       };
134
135       double getAverageEnergy() {
136
137         return averageEnergy;
138
139       };
140
141       double getErrorOfEnergy() {
142
143         return errorOfEnergy;
144
145       };
146
147       long getAutoCorrelationTimeOfEnergy() {
148
149         return autoCorrelationTimeOfEnergy;
150
151       };
```

```
152
153         double getAverageHeatCapacity() {
154
155           return averageHeatCapacity;
156
157         };
158
159         double getErrorOfHeatCapacity() {
160
161           return errorOfHeatCapacity;
162
163         };
164
165         long getAutoCorrelationTimeOfHeatCapacity() {
166
167           return autoCorrelationTimeOfHeatCapacity;
168
169         };
170
171         double getAverageMagnetisation() {
172
173           return averageMagnetisation;
174
175         };
176
177         double getErrorOfMagnetisation() {
178
179           return errorOfMagnetisation;
180
181         };
182
183         long getAutoCorrelationTimeOfMagnetisation() {
184
185           return autoCorrelationTimeOfMagnetisation;
186
187         };
188
189         double getAverageSusceptibility() {
190
191           return averageSusceptibility;
192
193         };
194
195         double getErrorOfSusceptibility() {
196
197           return errorOfSusceptibility;
198
199         };
200
201         long getAutoCorrelationTimeOfSusceptibility() {
202
203           return autoCorrelationTimeOfSusceptibility;
204
205         };
206
207         double getAverageAbsoluteMagnetisation() {
208
209           return averageAbsoluteMagnetisation;
210
211         };
212
213         double getErrorOfAbsoluteMagnetisation() {
```

```
214
215         return errorOfAbsoluteMagnetisation;
216
217       };
218
219       long getAutoCorrelationTimeOfAbsoluteMagnetisation() {
220
221         return autoCorrelationTimeOfAbsoluteMagnetisation;
222
223       };
224
225       double getAverageAbsoluteSusceptibility() {
226
227         return averageAbsoluteSusceptibility;
228
229       };
230
231       double getErrorOfAbsoluteSusceptibility() {
232
233         return errorOfAbsoluteSusceptibility;
234
235       };
236
237       long getAutoCorrelationTimeOfAbsoluteSusceptibility() {
238
239         return autoCorrelationTimeOfAbsoluteSusceptibility;
240
241       };
242
243   };
244
245   #endif
```

Listing A.6: Classes/Algorithm/AbstractAlgorithm.cpp

A.3.2 ED Algorithmus

```
1   #ifndef CLASS_EDALGORITHM
2   #define CLASS_EDALGORITHM
3
4   #include <tnt/tnt_array2d.h>
5   #include <jama/jama_eig.h>
6
7   #include "AbstractAlgorithm.cpp"
8
9   class EDAlgorithm : public AbstractAlgorithm {
10
11    protected:
12
13      int twoPowN;
14      TNT::Array1D<double> *e;
15
16      void hAction(TNT::Array2D<double> hg, int s, TNT::Array1D<int> mapS)
            {
17
18        for(int b = 1; b <= lattice->getNb(); b++) {
19
20          bool i1Up = s & (int(1) << lattice->getI1(b));
21          bool i2Up = s & (int(1) << lattice->getI2(b));
```

```
22
23            hg[mapS[s]][mapS[s]] += i1Up == i2Up ? 0.25 : -0.25;
24
25            if(i1Up != i2Up) hg[mapS[s]][mapS[s ^ (int(1) << lattice->getI1(b
                  )) ^ (int(1) << lattice->getI2(b))]] += 0.5;
26
27          }
28
29        };
30
31        int getNumOf1Bits(int value) {
32
33          value = ((value & 0xaaaaaaaa) >> 1 ) + (value & 0x55555555);
34          value = ((value & 0xcccccccc) >> 2 ) + (value & 0x33333333);
35          value = ((value & 0xf0f0f0f0) >> 4 ) + (value & 0x0f0f0f0f);
36          value = ((value & 0xff00ff00) >> 8 ) + (value & 0x00ff00ff);
37          value = ((value & 0xffff0000) >> 16) + (value & 0x0000ffff);
38
39          return value;
40
41        };
42
43      public:
44
45        EDAlgorithm(AbstractLattice* lattice_parameter, int
              measureCount_parameter) : AbstractAlgorithm(lattice_parameter,
              measureCount_parameter) {
46
47          twoPowN = int(1) << lattice->getN();
48
49          TNT::Array2D<int> mapSIndex(lattice->getN() + 1, twoPowN, 0); // 2
                nd dimension needs to be minimum "choose n/2 from n", but I was
                 lazy ...
50          TNT::Array1D<int> mapS(twoPowN, 0);
51          TNT::Array1D<int> sIndexLength(lattice->getN() + 1, 0);
52
53          e = new TNT::Array1D<double>(twoPowN, 0.0);
54          int eIndex = 0;
55
56          for(int s = 0; s < twoPowN; s++) {
57            int g = getNumOf1Bits(s);
58            mapSIndex[g][sIndexLength[g]] = s;
59            mapS[s] = sIndexLength[g];
60            sIndexLength[g]++;
61          }
62
63          for(int g = 0; g <= lattice->getN() / 2; g++) {
64
65            TNT::Array2D<double> hg(sIndexLength[g], sIndexLength[g], 0.0);
66            TNT::Array1D<double> eg(sIndexLength[g], 0.0);
67
68            for(int sIndex = 0; sIndex < sIndexLength[g]; sIndex++) {
69              hAction(hg, mapSIndex[g][sIndex], mapS);
70            }
71
72            std::cerr << "Diagonalize matrix " << (g + 1) << " von " << ((
                  lattice->getN() / 2) + 1) << " [" << sIndexLength[g] << "^2]"
                   << std::endl;
73
74            JAMA::Eigenvalue<double> eFactory(hg);
75            eFactory.getRealEigenvalues(eg);
76
```

```
77            for(int sIndex = 0; sIndex < sIndexLength[g]; sIndex++) {
78
79              (*e)[eIndex] = eg[sIndex];
80              eIndex++;
81              if(lattice->getN() % 2 != 0 || g != lattice->getN() / 2) {
82                (*e)[eIndex] = eg[sIndex];
83                eIndex++;
84              }
85            }
86
87          }
88
89        };
90
91        ~EDAlgorithm() {
92
93          delete e;
94
95        };
96
97        void runTemperatureRound(double t_parameter) {
98
99          AbstractAlgorithm::runTemperatureRound(t_parameter);
100
101         double sumOfE = 0;
102         double sumOfESquared = 0;
103         double z = 0;
104
105         for(int s = 0; s < twoPowN; s++) {
106           double weight = exp(-(*e)[s] / t);
107           sumOfE += weight * (*e)[s];
108           sumOfESquared += weight * pow((*e)[s], 2);
109           z += weight;
110         }
111
112         averageEnergy = sumOfE / z / lattice->getN();
113         averageHeatCapacity = ((sumOfESquared / z) - pow(sumOfE / z, 2)) /
              pow(t, 2) / lattice->getN();
114
115       };
116
117   };
118
119   #endif
```

Listing A.7: Classes/Algorithm/EDAlgorithm.cpp

A.3.3 Ising Algorithmus

```
1   #ifndef CLASS_ISINGALGORITHM
2   #define CLASS_ISINGALGORITHM
3
4   #include <gsl/gsl_rng.h>
5
6   #include "AbstractAlgorithm.cpp"
7   #include "../Analyzer/IsingEnergyAnalyzer.cpp"
8   #include "../Analyzer/IsingHeatCapacityAnalyzer.cpp"
9   #include "../Analyzer/IsingMagnetisationAnalyzer.cpp"
10  #include "../Analyzer/IsingSusceptibilityAnalyzer.cpp"
```

```
11  #include "../Analyzer/IsingAbsoluteMagnetisationAnalyzer.cpp"
12  #include "../Analyzer/IsingAbsoluteSusceptibilityAnalyzer.cpp"
13
14  class ISINGAlgorithm : public AbstractAlgorithm {
15
16    protected:
17
18      const gsl_rng_type *generatorType;
19      gsl_rng *generator;
20
21      bool *spins;
22      double energy;
23      int spinSum;
24      bool start;
25
26      void doSweep() {
27
28        double energyDifference;
29        double weight;
30        double randomWeight;
31
32        for(int i = 0; i < lattice->getN(); i++) {
33
34          energyDifference = getEnergyDifferenceOfFlip(i);
35          weight = exp(-energyDifference / t);
36          randomWeight = gsl_rng_uniform(generator);
37
38          if(weight >= randomWeight) {
39            spinSum += getSpinSumDifferenceOfFlip(i);
40            energy  += energyDifference;
41            spins[i] = !spins[i];
42          }
43
44        }
45
46      };
47
48      double calculateEnergy() {
49
50        int sumOfBonds = 0;
51
52        for(int b = 1; b <= lattice->getNb(); b++) {
53          sumOfBonds += spins[lattice->getI1(b)] == spins[lattice->getI2(b)
              ] ? 1 : -1;
54        }
55
56        return -sumOfBonds;
57
58      };
59
60      int calculateSpinSum() {
61
62        int spinSum = 0;
63
64        for(int i = 0; i < lattice->getN(); i++) {
65          spinSum += spins[i] ? 1 : -1;
66        }
67
68        return spinSum;
69
70      };
71
```

```
72        double getEnergyDifferenceOfFlip(int i) {
73
74          int *neighbours = lattice->getNeighbours(i);
75          double energyDiffernce = 0;
76
77          for(int j = 1; j <= neighbours[0]; j++) {
78            energyDiffernce += 2 * (spins[i] == spins[neighbours[j]] ? 1 :
                  -1);
79          }
80
81          delete[] neighbours;
82
83          return energyDiffernce;
84
85        };
86
87        int getSpinSumDifferenceOfFlip(int i) {
88
89          return -2 * (spins[i] ? 1 : -1);
90
91        };
92
93      public:
94
95        ISINGAlgorithm(AbstractLattice* lattice_parameter, int
              measureCount_parameter) : AbstractAlgorithm(lattice_parameter,
              measureCount_parameter) {
96
97          gsl_rng_env_setup();
98          generatorType = gsl_rng_default;
99          generator = gsl_rng_alloc(generatorType);
100         gsl_rng_set(generator, time(NULL));
101
102         spins = new bool[lattice->getN()];
103         start = true;
104
105       };
106
107       ~ISINGAlgorithm() {
108
109         delete[] spins;
110
111         gsl_rng_free(generator);
112
113       };
114
115       void runTemperatureRound(double t_parameter) {
116
117         AbstractAlgorithm::runTemperatureRound(t_parameter);
118
119         if(start) {
120           for(int i = 0; i < lattice->getN(); i++) {
121             spins[i] = bool(gsl_rng_uniform_int(generator, 2));
122           }
123
124           energy  = calculateEnergy();
125           spinSum = calculateSpinSum();
126           start = false;
127         }
128
129         for(long i = 0; i < runCount; i++) {
130
```

```
131             if(i > runCount - measureCount) {
132               energyMeasurements[i - runCount + measureCount]
                                  = energy / lattice->getN();
133               magnetisationMeasurements[i - runCount + measureCount]
                          = double(spinSum) / lattice->getN();
134               absoluteMagnetisationMeasurements[i - runCount + measureCount]
                         = fabs(double(spinSum) / lattice->getN());
135             }
136
137             doSweep();
138
139           }
140
141           IsingEnergyAnalyzer *isingEnergyAnalyzer = new IsingEnergyAnalyzer(
                  this, lattice);
142           isingEnergyAnalyzer->analyze();
143           averageEnergy = isingEnergyAnalyzer->getAverage();
144           errorOfEnergy = isingEnergyAnalyzer->getError();
145           autoCorrelationTimeOfEnergy = isingEnergyAnalyzer->
                  getAutoCorrelationTime();
146           delete isingEnergyAnalyzer;
147
148           IsingHeatCapacityAnalyzer *isingHeatCapacityAnalyzer = new
                  IsingHeatCapacityAnalyzer(this, lattice);
149           isingHeatCapacityAnalyzer->analyze();
150           averageHeatCapacity = isingHeatCapacityAnalyzer->getAverage();
151           errorOfHeatCapacity = isingHeatCapacityAnalyzer->getError();
152           autoCorrelationTimeOfHeatCapacity = isingHeatCapacityAnalyzer->
                  getAutoCorrelationTime();
153           delete isingHeatCapacityAnalyzer;
154
155           IsingMagnetisationAnalyzer *isingMagnetisationAnalyzer = new
                  IsingMagnetisationAnalyzer(this, lattice);
156           isingMagnetisationAnalyzer->analyze();
157           averageMagnetisation = isingMagnetisationAnalyzer->getAverage();
158           errorOfMagnetisation = isingMagnetisationAnalyzer->getError();
159           autoCorrelationTimeOfMagnetisation = isingMagnetisationAnalyzer->
                  getAutoCorrelationTime();
160           delete isingMagnetisationAnalyzer;
161
162           IsingSusceptibilityAnalyzer *isingSusceptibilityAnalyzer = new
                  IsingSusceptibilityAnalyzer(this, lattice);
163           isingSusceptibilityAnalyzer->analyze();
164           averageSusceptibility = isingSusceptibilityAnalyzer->getAverage();
165           errorOfSusceptibility = isingSusceptibilityAnalyzer->getError();
166           autoCorrelationTimeOfSusceptibility = isingSusceptibilityAnalyzer->
                  getAutoCorrelationTime();
167           delete isingSusceptibilityAnalyzer;
168
169           IsingAbsoluteMagnetisationAnalyzer *
                  isingAbsoluteMagnetisationAnalyzer = new
                  IsingAbsoluteMagnetisationAnalyzer(this, lattice);
170           isingAbsoluteMagnetisationAnalyzer->analyze();
171           averageAbsoluteMagnetisation = isingAbsoluteMagnetisationAnalyzer->
                  getAverage();
172           errorOfAbsoluteMagnetisation = isingAbsoluteMagnetisationAnalyzer->
                  getError();
173           autoCorrelationTimeOfAbsoluteMagnetisation =
                  isingAbsoluteMagnetisationAnalyzer->getAutoCorrelationTime();
174           delete isingAbsoluteMagnetisationAnalyzer;
175
```

```
176           IsingAbsoluteSusceptibilityAnalyzer *
                  isingAbsoluteSusceptibilityAnalyzer = new
                  IsingAbsoluteSusceptibilityAnalyzer(this, lattice);
177           isingAbsoluteSusceptibilityAnalyzer->analyze();
178           averageAbsoluteSusceptibility = isingAbsoluteSusceptibilityAnalyzer
                  ->getAverage();
179           errorOfAbsoluteSusceptibility = isingAbsoluteSusceptibilityAnalyzer
                  ->getError();
180           autoCorrelationTimeOfAbsoluteSusceptibility =
                  isingAbsoluteSusceptibilityAnalyzer->getAutoCorrelationTime();
181           delete isingAbsoluteSusceptibilityAnalyzer;
182
183         };
184
185     };
186
187     #endif
```

Listing A.8: Classes/Algorithm/ISINGAlgorithm.cpp

A.3.4 SSE Algorithmus

```
1     #ifndef CLASS_SSEALGORITHM
2     #define CLASS_SSEALGORITHM
3
4     #include <gsl/gsl_rng.h>
5
6     #include "AbstractAlgorithm.cpp"
7     #include "../Analyzer/SseEnergyAnalyzer.cpp"
8     #include "../Analyzer/SseHeatCapacityAnalyzer.cpp"
9
10    class SSEAlgorithm : public AbstractAlgorithm {
11
12      protected:
13
14        const gsl_rng_type *generatorType;
15        gsl_rng *generator;
16
17        bool *spins;
18        long nr;
19        long lMax;
20        long l;
21        int *s;
22        long *x;
23
24        void doSweep() {
25
26          doDiagonalUpdate();
27
28          doOperatorLoopUpdate();
29
30          if(l < 4 * nr / 3) l = 4 * nr / 3;
31          if(l > lMax) throw "Reached lMax";
32
33        };
34
35        void doDiagonalUpdate() {
36
37          int b;
```

```
38
39          for(long p = 0; p < l; p++) {
40
41            if(s[p] == 0) { // No operator -> try to insert
42
43              b = gsl_rng_uniform_int(generator, lattice->getNb()) + 1;
44
45              if(spins[lattice->getI1(b)] == spins[lattice->getI2(b)])
                    continue; // if bond-neighbour spins are parallel -> go to
                    next p
46
47              if(gsl_rng_uniform(generator) < ((double) lattice->getNb()) / 2
                     / (l - nr) / t) {
48                s[p] = 2 * b;
49                nr++;
50              }
51
52            } else if(s[p] % 2 == 0) { // Diagonal operator -> try to remove
53
54              if(gsl_rng_uniform(generator) < (double) 2 * (l - nr + 1) * t /
                     lattice->getNb()) {
55                s[p] = 0;
56                nr--;
57              }
58
59            } else { // Off-Diagonal operator -> just flip the bond-neighbour
                   spins
60
61              b = s[p] / 2;
62
63              spins[lattice->getI1(b)] = !spins[lattice->getI1(b)];
64              spins[lattice->getI2(b)] = !spins[lattice->getI2(b)];
65
66            }
67
68          }
69
70        };
71
72        void doOperatorLoopUpdate() {
73
74          // Construct link list x
75          long v0;
76          int b;
77          int i1;
78          int i2;
79
80          long *vFirst = new long[lattice->getN()];
81          for(int i = 0; i < lattice->getN(); i++) vFirst[i] = -1;
82          long *vLast = new long[lattice->getN()];
83          for(int i = 0; i < lattice->getN(); i++) vLast[i]  = -1;
84
85          for(long v = 0; v < 4 * l; v++) x[v] = -1;
86
87          for(long p = 0; p < l; p++) {
88
89            if(s[p] == 0) continue; // No operator -> go to next p
90
91            v0 = 4 * p;
92            b = s[p] / 2;
93            i1 = lattice->getI1(b);
94            i2 = lattice->getI2(b);
```

```
95
96            // Link the last 2-vertex on this spin to new 0-vertex
97            if(vLast[i1] == -1) {
98              vFirst[i1] = v0;
99            } else {
100             x[vLast[i1]] = v0;
101             x[v0] = vLast[i1];
102           }
103
104           // Link the last 3-vertex on this spin to new 1-vertex
105           if(vLast[i2] == -1) {
106             vFirst[i2] = v0 + 1;
107           } else {
108             x[vLast[i2]] = v0 + 1;
109             x[v0 + 1] = vLast[i2];
110           }
111
112           // Set the 2 and 3-vertex as last vertex on this spin
113           vLast[i1] = v0 + 2;
114           vLast[i2] = v0 + 3;
115
116         }
117
118         // Link vertexes periodicly if they interacted with operators
119         for(int i = 0; i < lattice->getN(); i++) {
120
121           if(vFirst[i] != -1) {
122             x[vFirst[i]] = vLast[i];
123             x[vLast[i]] = vFirst[i];
124           }
125
126         }
127
128         // Do the actual update
129         for(long v = 0; v < 4 * l; v += 2) {
130
131           if(x[v] < 0) continue;
132
133           long vT = v;
134           bool flipping = gsl_rng_uniform(generator) < 0.5;
135
136           while(x[vT] >= 0) {
137
138             // Flip
139             if(flipping) {
140               long p = vT / 4;
141               s[p] = s[p] ^ 1;
142             }
143
144             // Walk to other operator
145             vT = x[vT];
146
147             // Delete the way I went and the way back
148             x[vT] = x[x[vT]] = flipping ? -2 : -1;
149
150             // Walk to neighbour on operator
151             vT = vT ^ 1;
152
153           }
154
155         }
156
```

```
157           // Adjust spins
158           for(int i = 0; i < lattice->getN(); i++) {
159
160             if(vFirst[i] == -1) {
161               if(gsl_rng_uniform(generator) < 0.5) spins[i] = !spins[i];
162             } else {
163               if(x[vFirst[i]] == -2) spins[i] = !spins[i];
164             }
165
166           }
167
168           delete[] vLast;
169           delete[] vFirst;
170
171         };
172
173       public:
174
175         SSEAlgorithm(AbstractLattice* lattice_parameter, int
                measureCount_parameter) : AbstractAlgorithm(lattice_parameter,
                measureCount_parameter) {
176
177           gsl_rng_env_setup();
178           generatorType = gsl_rng_default;
179           generator = gsl_rng_alloc(generatorType);
180           gsl_rng_set(generator, time(NULL));
181
182           spins = new bool[lattice->getN()];
183           for(int i = 0; i < lattice->getN(); i++) spins[i] =
                gsl_rng_uniform_int(generator, 2);
184           nr = 0;
185           lMax = 10000000;
186           l = 10;
187           s = new int[lMax];
188           for(long p = 0; p < lMax; p++) s[p] = 0;
189           x = new long[4 * lMax];
190
191         };
192
193         ~SSEAlgorithm() {
194
195           delete x;
196           delete s;
197           delete spins;
198
199           gsl_rng_free(generator);
200
201         };
202
203         void runTemperatureRound(double t_parameter) {
204
205           AbstractAlgorithm::runTemperatureRound(t_parameter);
206
207           for(long i = 0; i < runCount; i++) {
208
209             if(i > runCount - measureCount) {
210               energyMeasurements[i - runCount + measureCount] = -double(nr) *
                      t / lattice->getN();
211               magnetisationMeasurements[i - runCount + measureCount] = nr;
212             }
213             doSweep();
214
```

```
215         }
216
217         SseEnergyAnalyzer *sseEnergyAnalyzer = new SseEnergyAnalyzer(this,
                lattice);
218         sseEnergyAnalyzer->analyze();
219         averageEnergy = sseEnergyAnalyzer->getAverage() + (double(lattice->
                getNb()) / lattice->getN() / 4);
220         errorOfEnergy = sseEnergyAnalyzer->getError();
221         autoCorrelationTimeOfEnergy = sseEnergyAnalyzer->
                getAutoCorrelationTime();
222         delete sseEnergyAnalyzer;
223
224         SseHeatCapacityAnalyzer *sseHeatCapacityAnalyzer = new
                SseHeatCapacityAnalyzer(this, lattice);
225         sseHeatCapacityAnalyzer->analyze();
226         averageHeatCapacity = sseHeatCapacityAnalyzer->getAverage();
227         errorOfHeatCapacity = sseHeatCapacityAnalyzer->getError();
228         autoCorrelationTimeOfHeatCapacity = sseHeatCapacityAnalyzer->
                getAutoCorrelationTime();
229         delete sseHeatCapacityAnalyzer;
230
231       };
232
233   };
234
235   #endif
```

Listing A.9: Classes/Algorithm/SSEAlgorithm.cpp

A.4 Analysemodule

A.4.1 Abstrakte Analyseklasse

```
1   #ifndef CLASS_ABSTRACTANALYZER
2   #define CLASS_ABSTRACTANALYZER
3
4   class AbstractAnalyzer {
5
6     protected:
7
8       AbstractAlgorithm* algorithm;
9       AbstractLattice* lattice;
10
11      double average;
12      double error;
13      long autoCorrelationTime;
14
15    public:
16
17      AbstractAnalyzer(AbstractAlgorithm* algorithm_parameter,
            AbstractLattice* lattice_parameter) {
18
19        algorithm = algorithm_parameter;
20        lattice = lattice_parameter;
21
22        average = 0;
23        error = 0;
```

```
24          autoCorrelationTime = 0;
25
26        };
27
28        ~AbstractAnalyzer() {};
29
30        virtual const char* getQuantityName() = 0;
31        virtual double getQuantity(long time) = 0;
32
33        void analyze() {
34
35          double *sum        = new double[algorithm->getMeasureCount()];
36          double *sumSquared = new double[algorithm->getMeasureCount()];
37
38          for(long time = 0; time < algorithm->getMeasureCount(); time++) {
39            sum[time]        = (time == 0 ? 0 : sum[time - 1]) + getQuantity(
                  time);
40            sumSquared[time] = (time == 0 ? 0 : sumSquared[time - 1]) + pow(
                  getQuantity(time), 2);
41          }
42
43          for(autoCorrelationTime = 0; autoCorrelationTime < algorithm->
                getMeasureCount(); autoCorrelationTime++) {
44
45            double sumP = 0;
46
47            for(long time = 0; time < algorithm->getMeasureCount() -
                  autoCorrelationTime; time++) {
48              sumP += getQuantity(time) * getQuantity(time +
                    autoCorrelationTime);
49            }
50
51            long timeToAverage = algorithm->getMeasureCount() -
                  autoCorrelationTime;
52            double autoCorrelation = ((sumP / timeToAverage) - pow(sum[
                  algorithm->getMeasureCount() - 1] / timeToAverage, 2)) / ((
                  sumSquared[algorithm->getMeasureCount() - 1] / timeToAverage)
                   - pow(sum[algorithm->getMeasureCount() - 1] / timeToAverage,
                   2));
53
54            if(isnan(autoCorrelation)) {
55              printf("# %s at temperature=%+20.13e was not measured, due to
                    the unknown relaxation time\n", getQuantityName(),
                    algorithm->getTemperature());
56              autoCorrelationTime = -1;
57              break;
58            }
59
60            printf("# Auto-Correlation of %s=%+20.13e\n", getQuantityName(),
                  autoCorrelation);
61
62            if(autoCorrelation < exp(-1)) break;
63
64          }
65
66          autoCorrelationTime++;
67
68          if(autoCorrelationTime != 0) {
69
70            long binsCount = algorithm->getMeasureCount() / 3 /
                  autoCorrelationTime;
71            double *binAverage = new double[binsCount];
```

```
72
73            for(long bin = 0; bin < binsCount; bin++) {
74              binAverage[bin] = (sum[(bin + 1) * 3 * autoCorrelationTime] -
                    sum[bin * 3 * autoCorrelationTime]) / 3 /
                    autoCorrelationTime;
75              average += binAverage[bin] / binsCount;
76            }
77
78            double deviationSum = 0;
79            for(long bin = 0; bin < binsCount; bin++) {
80              deviationSum += pow(binAverage[bin] - average, 2);
81            }
82            error = sqrt(deviationSum / binsCount / (binsCount - 1));
83
84            delete[] binAverage;
85
86          }
87
88          delete[] sum;
89          delete[] sumSquared;
90
91        };
92
93        double getAverage() {
94
95          return average;
96
97        };
98
99        double getError() {
100
101         return error;
102
103       };
104
105       long getAutoCorrelationTime() {
106
107         return autoCorrelationTime;
108
109       };
110
111   };
112
113   #endif
```

Listing A.10: Classes/Analyzer/AbstractAnalyzer.cpp

A.4.2 Analyse für die Energie (Ising)

```
1   #ifndef CLASS_ISINGENERGYANALYZER
2   #define CLASS_ISINGENERGYANALYZER
3
4   #include "AbstractAnalyzer.cpp"
5
6   class IsingEnergyAnalyzer : public AbstractAnalyzer {
7
8     public:
9
```

```
10      IsingEnergyAnalyzer(AbstractAlgorithm* algorithm_parameter,
            AbstractLattice* lattice_parameter) : AbstractAnalyzer(
            algorithm_parameter, lattice_parameter) {};
11
12      ~IsingEnergyAnalyzer() {};
13
14      const char* getQuantityName() {
15
16        return "Energy";
17
18      };
19
20      double getQuantity(long time) {
21
22        return algorithm->getEnergyMeasurement(time);
23
24      };
25
26  };
27
28  #endif
```

Listing A.11: Classes/Analyzer/IsingEnergyAnalyzer.cpp

A.4.3 Analyse für die Wärmekapazität (Ising)

```
1   #ifndef CLASS_ISINGHEATCAPACITYANALYZER
2   #define CLASS_ISINGHEATCAPACITYANALYZER
3
4   #include "AbstractAnalyzer.cpp"
5
6   class IsingHeatCapacityAnalyzer : public AbstractAnalyzer {
7
8     public:
9
10      IsingHeatCapacityAnalyzer(AbstractAlgorithm* algorithm_parameter,
            AbstractLattice* lattice_parameter) : AbstractAnalyzer(
            algorithm_parameter, lattice_parameter) {};
11
12      ~IsingHeatCapacityAnalyzer() {};
13
14      const char* getQuantityName() {
15
16        return "Heat Capacity";
17
18      };
19
20      double getQuantity(long time) {
21
22        return (pow(algorithm->getEnergyMeasurement(time), 2) - pow(
              algorithm->getAverageEnergy(), 2)) * lattice->getN() / pow(
              algorithm->getTemperature(), 2);
23
24      };
25
26  };
27
28  #endif
```

Listing A.12: Classes/Analyzer/IsingHeatCapacityAnalyzer.cpp

A.4.4 Analyse für die Magnetisierung (Ising)

```
#ifndef CLASS_ISINGMAGNETISATIONANALYZER
#define CLASS_ISINGMAGNETISATIONANALYZER

#include "AbstractAnalyzer.cpp"

class IsingMagnetisationAnalyzer : public AbstractAnalyzer {

  public:

    IsingMagnetisationAnalyzer(AbstractAlgorithm* algorithm_parameter,
        AbstractLattice* lattice_parameter) : AbstractAnalyzer(
        algorithm_parameter, lattice_parameter) {};

    ~IsingMagnetisationAnalyzer() {};

    const char* getQuantityName() {

      return "Magnetization";

    };

    double getQuantity(long time) {

      return algorithm->getMagnetisationMeasurement(time);

    };

};

#endif
```

Listing A.13: Classes/Analyzer/IsingMagnetisationAnalyzer.cpp

A.4.5 Analyse für die magnetische Suszeptibilität (Ising)

```
#ifndef CLASS_ISINGSUSCEPTIBILITYANALYZER
#define CLASS_ISINGSUSCEPTIBILITYANALYZER

#include "AbstractAnalyzer.cpp"

class IsingSusceptibilityAnalyzer : public AbstractAnalyzer {

  public:

    IsingSusceptibilityAnalyzer(AbstractAlgorithm* algorithm_parameter,
        AbstractLattice* lattice_parameter) : AbstractAnalyzer(
        algorithm_parameter, lattice_parameter) {};

    ~IsingSusceptibilityAnalyzer() {};

```

```
14        const char* getQuantityName() {
15
16          return "Susceptibility";
17
18        };
19
20        double getQuantity(long time) {
21
22          return (pow(algorithm->getMagnetisationMeasurement(time), 2) - pow(
                algorithm->getAverageMagnetisation(), 2)) * lattice->getN() /
                algorithm->getTemperature();
23
24        };
25
26    };
27
28    #endif
```

Listing A.14: Classes/Analyzer/IsingSusceptibilityAnalyzer.cpp

A.4.6 Analyse für die abs. Magnetisierung (Ising)

```
1     #ifndef CLASS_ISINGABSOLUTEMAGNETISATION
2     #define CLASS_ISINGABSOLUTEMAGNETISATION
3
4     #include "AbstractAnalyzer.cpp"
5
6     class IsingAbsoluteMagnetisationAnalyzer : public AbstractAnalyzer {
7
8       public:
9
10        IsingAbsoluteMagnetisationAnalyzer(AbstractAlgorithm*
              algorithm_parameter, AbstractLattice* lattice_parameter) :
              AbstractAnalyzer(algorithm_parameter, lattice_parameter) {};
11
12        ~IsingAbsoluteMagnetisationAnalyzer() {};
13
14        const char* getQuantityName() {
15
16          return "Absolute Magnetization";
17
18        };
19
20        double getQuantity(long time) {
21
22          return algorithm->getAbsoluteMagnetisationMeasurement(time);
23
24        };
25
26    };
27
28    #endif
```

Listing A.15: Classes/Analyzer/IsingAbsoluteMagnetisationAnalyzer.cpp

A.4.7 Analyse für die abs. mag. Suszeptibilität (Ising)

```
#ifndef CLASS_ISINGABSOLUTESUSCEPTIBILITYANALYZER
#define CLASS_ISINGABSOLUTESUSCEPTIBILITYANALYZER

#include "AbstractAnalyzer.cpp"

class IsingAbsoluteSusceptibilityAnalyzer : public AbstractAnalyzer {

  public:

    IsingAbsoluteSusceptibilityAnalyzer(AbstractAlgorithm*
        algorithm_parameter, AbstractLattice* lattice_parameter) :
        AbstractAnalyzer(algorithm_parameter, lattice_parameter) {};

    ~IsingAbsoluteSusceptibilityAnalyzer() {};

    const char* getQuantityName() {

      return "Absolute Susceptibility";

    };

    double getQuantity(long time) {

      return (pow(algorithm->getAbsoluteMagnetisationMeasurement(time),
          2) - pow(algorithm->getAverageAbsoluteMagnetisation(), 2)) *
          lattice->getN() / algorithm->getTemperature();

    };

};

#endif
```

Listing A.16: Classes/Analyzer/IsingAbsoluteSusceptibilityAnalyzer.cpp

A.4.8 Analyse für die Energie (SSE)

```
#ifndef CLASS_SSEENERGYANALYZER
#define CLASS_SSEENERGYANALYZER

#include "AbstractAnalyzer.cpp"

class SseEnergyAnalyzer : public AbstractAnalyzer {

  public:

    SseEnergyAnalyzer(AbstractAlgorithm* algorithm_parameter,
        AbstractLattice* lattice_parameter) : AbstractAnalyzer(
        algorithm_parameter, lattice_parameter) {};

    ~SseEnergyAnalyzer() {};

    const char* getQuantityName() {

      return "Energy";

    };

```

```
20        double getQuantity(long time) {
21
22          return algorithm->getEnergyMeasurement(time);
23
24        };
25
26    };
27
28    #endif
```

Listing A.17: Classes/Analyzer/SseEnergyAnalyzer.cpp

A.4.9 Analyse für die Wärmekapazität (SSE)

```
1     #ifndef CLASS_SSEHEATCAPACITYANALYZER
2     #define CLASS_SSEHEATCAPACITYANALYZER
3
4     #include "AbstractAnalyzer.cpp"
5
6     class SseHeatCapacityAnalyzer : public AbstractAnalyzer {
7
8       public:
9
10        SseHeatCapacityAnalyzer(AbstractAlgorithm* algorithm_parameter,
              AbstractLattice* lattice_parameter) : AbstractAnalyzer(
              algorithm_parameter, lattice_parameter) {};
11
12        ~SseHeatCapacityAnalyzer() {};
13
14        const char* getQuantityName() {
15
16          return "Heat Capacity";
17
18        };
19
20        double getQuantity(long time) {
21
22          double averageNr = ((algorithm->getAverageEnergy() - (double(
                lattice->getNb()) / lattice->getN() / 4)) * -lattice->getN() /
                algorithm->getTemperature());
23
24          return (pow(-algorithm->getEnergyMeasurement(time) * lattice->getN
                () / algorithm->getTemperature(), 2) - pow(averageNr, 2) -
                averageNr)/lattice->getN();
25
26        };
27
28    };
29
30    #endif
```

Listing A.18: Classes/Analyzer/SseHeatCapacityAnalyzer.cpp

Abbildungsverzeichnis

Quellcodeverzeichnis

Literaturverzeichnis

[BMB04] Ballac, Michel L. ; Mortessagne, Fabrice ; Batrouni, G. G.: *Equilibrium and Non-Equilibrium Statistical Thermodynamics.* Cambridge University Press, 2004

[Gro04] Grossjohann, Simon-Nils: *Stochastic Series Expansion an niedrigdimensionalen Quanten-Spin-Systemen*, Technische Universität Braunschweig, Diplomarbeit, 2004

[Han62] Handscomb, D. C.: The Monte Carlo method in quantum statistical mechanics. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 58 (1962), S. 594–598

[Knu02] Knuth, Donald E.: Der Perfektionist. In: *heise.de - c't* (2002), Mai, S. 190

[LLP10] Levin, David ; Luczak, Malwina ; Peres, Yuval: Glauber dynamics for the mean-field Ising model: cut-off, critical power law, and metastability. In: *Probability Theory and Related Fields* 146 (2010), S. 223–265

[Mab06] Mabiala, Justin: Exact Diagonalisation of Heisenberg Cluster. (2006)

[Met87] Metropolis, Nicholas: The beginning of the Monte Carlo Method. In: *Los Alamos Science* 15 (1987), S. 125–130

[Nol07] Nolting, Wolfgang: *Grundkurs Theoretische Physik 6.* Springer Verlag, 2007

[Ons44] Onsager, Lars: Crystal Statistics. I. A Two-Dimensional Model with an Order-Disorder Transition. In: *Phys. Rev.* 65 (1944), Feb, Nr. 3-4, S. 117–149

[PS99] Pemantle, Robin ; Steif, Jeffrey E.: Robust Phase Transitions for Heisenberg and other Models on General Trees. In: *Annals Of Probabillity* (1999), Nr. 2, S. 876–912

[Pyt] *Homepage des Projekts PYTHIA. Ein Programm zur Generation von Hochenergie-Physik Events.* `http://projects.hepforge.org/pythia6/`

[San97] Sandvik, Anders W.: Finite-size scaling of the ground-state parameters of the two-dimensional Heisenberg model. (1997)

[San10] Sandvik, Anders W.: Computational Studies of Quantum Spin Systems. In: *AIP Conference Proceedings* 1297 (2010), Nr. 1, S. 135–338

[Spi] *Beitrag in techinterviewz.blogspot.com. Wie findet man die Anzahl der 1er Bits einer Zahl?* `http://techinterviewz.blogspot.com/2010/03/finding-number-of-on-bits-1-bits-given.html`

Erklärung zur Selbstständigkeit

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

_____ , _____ _____

Ort Datum Unterschrift