Das Ising Modell mit Monte Carlo Methoden

Bachelorarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Marcel Wirz

2007

Leiter der Arbeit

Prof. Uwe-Jens Wiese
Institut für Theoretische Physik, Universität Bern

Zusammenfassung

Der erste Teil dieser Arbeit umfasste die Erarbeitung des theoretischen Hintergrunds des Ising Modells. Im zweiten Teil ging es zuerst darum, das ein- und zweidimensionale Modell mit dem Metropolisalgorithmus auf einem Computer zu implementieren. Anschliessend wurde für das zweidimensionale Modell der viel effizientere Swendsen-Wang Cluster Algorithmus verwendet.

Inhaltsverzeichnis

Kapitel 1

Theoretische Grundlagen[1]

1.1 Physikalischer Hintergrund: Phasenübergänge und kritische Phänomene

Wenn Wasser gekühlt wird, wird es bei 0 Grad Celsius zu Eis, wird es erhitzt, wird es bei 100 Grad Celsius zu Dampf. Dieselbe Materie hat in diesen verschiedenen Aggregatszuständen ganz verschiedene physikalische Eigenschaften. Wechselt ein Material den Aggregatszustand, findet ein Phasenübergang statt. Solche Phasenübergänge findet man bei allen Materialien. Ein schematisches Phasendiagramm ist in Abbildung 1.1 gegeben. Dieses zeigt unter anderem

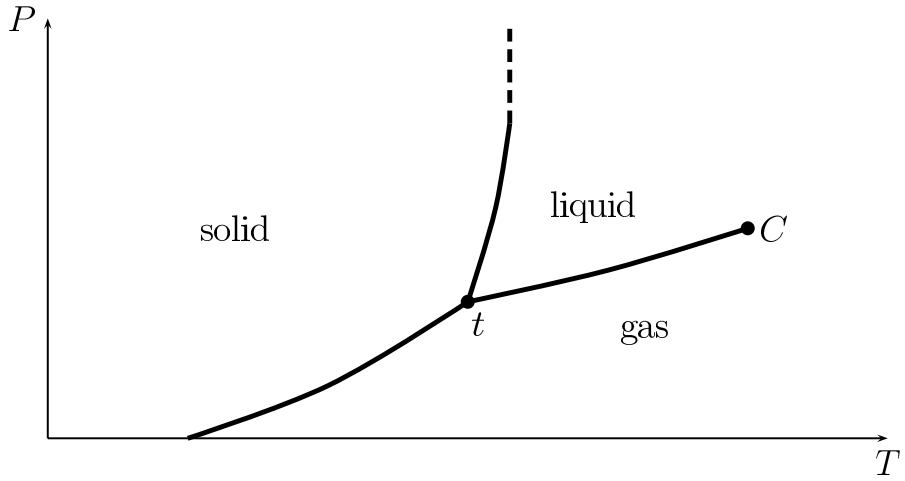


Abbildung 1.1: Schematisches Phasendiagramm. T: Temperatur, P: Druck, t: Tripelpunkt, C: kritischer Punkt. Bei den Linien finden die Phasenübergänge statt. (Abbildung aus [3])

den für uns interessanten kritischen Punkt (C) bei der Temperatur T_c und dem Druck P_c. Die beiden Phasen unterhalb des kritischen Punkts, flüssig und gasförmig, unterscheiden sich in Dichte, Wärmekapazität, etc. Nähert man sich dem kritischen Punkt, werden diese Unterschiede immer kleiner, bis sie beim kritischen Punkt ganz verschwinden. Beispielsweise zwischen Wasser und Dampf gibt es beim und oberhalb des kritischen Punkts keinen Unterschied mehr. Beim Wasser sind fernab von kritischen Punkt die Regionen gleicher Dichte klein, vergleichbar

[1] Die folgenden Ausführungen basieren hauptsächlich auf [1], [2] und [3].

zur durchschnittlichen Distanz zwischen den Teilchen. Man charakterisiert die Grösse der Regionen, über welche die Teilchen miteinander korreliert sind über die *Korrelationslänge*. Nahe beim kritischen Punkt ist die Korrelationslänge gross und sie geht beim kritischen Punkt gegen unendlich (vorausgesetzt, das System ist unendlich gross).
Kritische Phänomene gibt es nicht nur bei Wasser, diese sind im Gegenteil ganz generelle Phänomene der statistischen Physik. Sie treten beispielsweise auch in magnetischen Materialien auf. Diese sind etwas einfacher, weshalb wir uns im Folgenden auf den Magnetismus beschränken.
Wir wissen, dass ein Ferromagnet nach einer Magnetisierung durch ein äusseres Magnetfeld eine Restmagnetisierung behält. Dies passiert jedoch nur unterhalb der kritischen Temperatur, der sogenannten Curie-Temperatur. Oberhalb der Curie-Temperatur hat ein Ferromagnet ohne äusseres Magnetfeld keine Magnetisierung. Unterhalb der kritischen Temperatur tritt auch ohne ein äusseres Magnetfeld eine *spontane Magnetisierung* auf. Das heisst, dass sich die kleinsten magnetischen Einheiten ganz spontan ausrichten. In der Nähe des kritischen Punkts finden Phänomene statt, die ganz ähnlich sind zu denen bei Wasser. Verschiedene Systeme lassen sich in sogenannte *Universalitätsklassen* einteilen, wobei die Systeme innerhalb einer Klasse universelle kritische Eigenschaften aufweisen. Dies ist die Motivation, mit einem ganz einfachen System, dem Ising Modell, das viel einfacher zu untersuchen und zu simulieren ist, andere Systeme bezüglich des kritischen Verhaltens zu untersuchen.

1.2 Das Ising Modell

Das Ising Modell ist eines der einfachsten Modelle der klassischen statistischen Mechanik. Dennoch enthält es genug, um es auf viele verschiedene physikalische Systeme anzuwenden. Es kann idealisierte Magnete, die Mischung von Flüssigkeiten, kritische Opaleszenz in kochendem Wasser bei hohen Temperaturen und sogar spezielle Eigenschaften des Quark-Gluon-Plasmas, das das frühe Universum füllte, beschreiben. Wir verwenden es im Folgenden, um den Ferromagnetismus zu untersuchen.

1.2.1 Definition und Eigenschaften

Das Ising Modell ist das einfachste Spinmodell. Dabei handelt es sich nicht um quantenmechanische, sondern lediglich um klassische Spins, welche im Falle des Ising Modells nur die Werte ± 1 annehmen können. An jedem Gitterplatz eines kubischen, d dimensionalen, räumlichen Gitters befindet sich ein Spin. Das Ising Modell ist charakterisiert durch seine klassische Hamiltonfunktion, welche die Energie der Spinkonfigurationen angibt,

$$\mathcal{H}[s] = -J \sum_{\langle xy \rangle} s_x s_y - \mu B \sum_x s_x. \tag{1.1}$$

Dabei bedeutet $[s]$ eine bestimmte Spinkonfiguration, die Summe über $\langle xy \rangle$ die Summe über die nächsten Nachbarn, J die Kopplungskonstante (wie stark die Spins gekoppelt sind), s_i einen individuellen Spin, μ die Permeabilität und B die magnetische Flussdichte. Der erste Term stellt die Wechselwirkung der Spins und der zweite Term den Einfluss des Magnetfeldes dar. Ein positiver Wert für J bedeutet, da der Term ein negatives Vorzeichen hat, dass der erste Teil der Energie minimal ist, wenn die Spins parallel sind und maximal, wenn die Spins

antiparallel sind.[2] Da auch der zweite Term ein negatives Vorzeichen hat, wird dieser Teil der Energie minimal, wenn die Spins parallel zum äusseren Magnetfeld sind und maximal, wenn diese antiparallel zum äusseren Magnetfeld sind. Das System befindet sich in einem Wärmebad. Bei tiefen Temperaturen werden die Spins also über gewisse Bereiche gleichgerichtet und mit zunehmendem Magnetfeld bevorzugt parallel zu diesem sein. Bei höheren Temperaturen steht genug thermische Energie zur Verfügung, dass die Spins die Möglichkeit haben, untereinander und auch zum äusseren Magnetfeld antiparallel zu stehen.
Die klassische Zustandssumme lautet

$$Z = \sum_{[s]} \exp(-\beta \mathcal{H}[s]) = \prod_x \sum_{s_x = \pm 1} \exp(-\beta \mathcal{H}[s]). \tag{1.2}$$

Die Summe über alle Spinkonfigurationen entspricht einer unabhängigen Summation über alle möglichen Richtungen der individuellen Spins. Thermische Mittelwerte werden durch das Einsetzen von entsprechenden Observablen $\mathcal{O}[s]$ berechnet,

$$\langle \mathcal{O} \rangle = \frac{1}{Z} \sum_{[s]} \mathcal{O}[s] \exp(-\beta \mathcal{H}[s]). \tag{1.3}$$

Beispielsweise ist die Magnetisierung gegeben durch

$$\mathcal{M}[s] = \sum_x s_x, \tag{1.4}$$

mit dem entsprechenden thermischen Mittelwert

$$\langle \mathcal{M} \rangle = \frac{1}{Z} \sum_{[s]} \mathcal{M}[s] \exp(-\beta \mathcal{H}[s]) = \frac{\partial \log Z}{\partial(\beta \mu B)}. \tag{1.5}$$

Die magnetische Suszeptibilität ist definiert als

$$\chi = \frac{1}{L^d} (\langle \mathcal{M}^2 \rangle - \langle \mathcal{M} \rangle^2) = \frac{1}{L^d} \frac{\partial^2 \log Z}{\partial(\beta \mu B)^2}, \tag{1.6}$$

wobei L^d das räumliche Volumen bedeutet.

1.2.2 Periodische Randbedingungen

Bei Simulationen auf dem Computer können natürlich nur endliche Systeme berechnet werden. Um auch den Spins an den Rändern gleich viele nächste Nachbarn zu geben, werden periodische Randbedingungen eingeführt. Wie das funktioniert, sieht man am besten an der Abbildung 1.2 für ein zweidimensionales System.

1.2.3 Universalität und kritische Exponenten

Beim Ising Modell ist die *Korrelationslänge* die Länge, über welche die Spins korreliert, d.h. im Wesentlichen gleichgerichtet sind. Bei Temperaturen oberhalb der kritischen Temperatur

[2] Dies stellt den ferromagnetischen Fall dar. Ein negativer Wert für J stellt den antiferromagnetischen Fall dar, welcher hier nicht diskutiert wird.

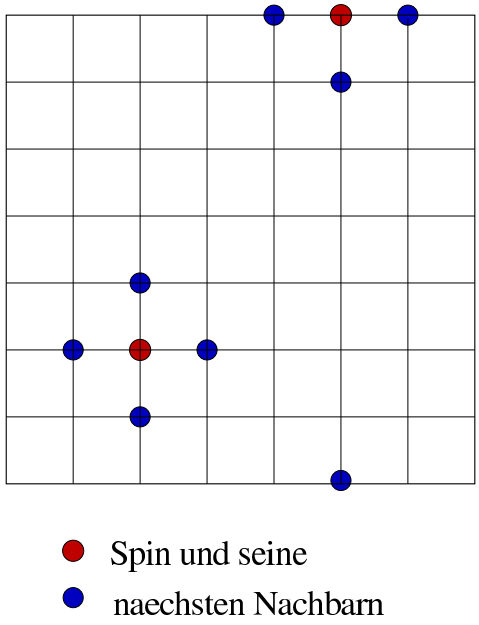


Abbildung 1.2: Definition der nächsten Nachbarn bei einem zweidimensionalen System. (Abbildung von [9])

beträgt die Korrelationslänge nur wenige Gitterabstände. Wenn man bei diesen Temperaturen ein reales Material simulieren möchte, wäre das Ising Modell viel zu simpel, weil zum Beispiel ein realer Ferromagnet nicht nur Wechselwirkung zwischen den nächsten Spinnachbarn hat. Jedoch sind die Details der Hamiltonfunktion auf der Skala der Gitterabstände nicht immer relevant. Bei der kritischen Temperatur divergiert die Korrelationslänge und universelles Verhalten tritt auf. Dabei werden die Details des Modells auf den Grössenskalen der Gitterabstände irrelevant für die Physik, welche auf Grössenskalen der Korrelationslänge stattfindet. In der Tat verhalten sich reale Materialien bei ihren kritischen Temperaturen wie das simple Ising Modell. Dies macht das Ising Modell so interessant, man kann damit indirekt viel kompliziertere Systeme am kritischen Punkt untersuchen, weil man das Ising Modell simulieren kann. Beim zweidimensionalen Ising Modell findet ein sogenannter Phasenübergang zweiter Ordnung statt und es ist ein sehr einfaches Mitglied einer grossen Universalitätsklasse von verschiedenen Modellen, welche alle dasselbe kritische Verhalten zeigen. Damit ist keineswegs gemeint, dass diese dieselbe kritische Temperatur haben. Jedoch divergiert bei ihnen die Korrelationslänge mit demselben kritischen Exponenten ν

$$\xi \propto |T - T_c|^{-\nu}, \tag{1.7}$$

ihre mittleren Magnetisierungen gehen bei der kritischen Temperatur mit demselben Exponenten β gegen null

$$\langle \mathcal{M} \rangle \propto |T - T_c|^{\beta}; \qquad T \leq T_c, \tag{1.8}$$

und ihre Suszeptibilitäten divergieren mit demselben kritischen Exponenten γ

$$\chi \propto |T - T_c|^{-\gamma}. \tag{1.9}$$

Die kritischen Exponenten ν, β und γ sind also identisch für verschiedene Systeme in der gleichen Universalitätsklasse. Dies ist jedoch nur ein Beispiel für die Universalität, das Konzept

ist wesentlich breiter. Für das Auftreten von kritischen Phänomenen spielt die Tatsache, dass die Teilchen miteinander wechselwirken, eine essentielle Rolle, aber die Art der Wechselwirkung ist nicht wesentlich. Deshalb müssen wir Systeme betrachten, in welchen die Teilchen miteinander wechselwirken. In welcher Universalitätsklasse sich ein System befindet, hängt von generellen Eigenschaften wie der Anzahl Dimensionen und seinen Symmetrien ab. So tritt zum Beispiel beim eindimensionalen Ising Modell kein Phasenübergang auf, während man in zwei Dimensionen einen Phasenübergang zweiter Ordnung beobachtet.

1.2.4 Analytische Resultate

Eine Dimension

Das eindimensionale Ising Modell kann analytisch gelöst werden. Eine Herleitung für ein System mit periodische Randbedingungen und ohne äusseres Magnetfeld findet man in [1]. Für die Zustandssumme ergibt sich

$$Z = \{[2\cosh(\beta J)]^L + [2\sinh(\beta J)]^L\}, \tag{1.10}$$

wobei L die Anzahl Spins bedeutet. Die Suszeptibilität ist gegeben durch

$$\chi = \frac{1 - \tanh^L(\beta J)}{1 + \tanh^L(\beta J)} \exp(2\beta J). \tag{1.11}$$

Das heisst, die Suszeptibilität hängt auch von der Grösse des Systems ab, allerdings variiert sie bei grossen Systemen nur schwach.

Zwei Dimensionen

Das zweidimensionale Ising Modell wurde 1944 von Onsager gelöst [5]. Eine Herleitung für die inverse kritische Temperatur β_c für ein System mit periodische Randbedingungen und ohne äusseres Magnetfeld ist in [1] gezeigt. Man findet die Gleichung

$$\tanh(\beta_c J) = \exp(-2\beta_c J), \tag{1.12}$$

und somit

$$\beta_c = \frac{\log(\sqrt{2}+1)}{2J}. \tag{1.13}$$

Mehr als zwei Dimensionen

Für Systeme mit mehr als zwei Dimensionen gibt es keine analytischen Lösungen, jedoch gute Näherungen und numerische Verfahren.

1.2.5 Mögliche Wege zu Resultaten

Die Untersuchung von kritischem Verhalten ist kompliziert. Es gibt beispielsweise keine generelle analytische Methode, um die kritischen Exponenten zu bestimmen. Wie bereits gesagt, stellen ein und zweidimensionale Systeme Ausnahmen dar. Eine Möglichkeit, um das kritische Verhalten zu illustrieren, stellt die *mean field approximation* dar. Sie liefert aber nur qualitative Resultate und zum Beispiel keine exakten kritischen Exponenten. Die mean field

approximation wird erst im Limes von unendlich vielen Dimensionen exakt. Die Methode soll hier aber nicht näher erläutert werden, genaueres findet man zum Beispiel in [1] oder [3]. Monte Carlo Methoden sind numerische Verfahren und können verwendet werden, um das kritische Verhalten zu untersuchen. Sie stellen den Mittelpunkt dieser Arbeit dar.

1.3 Importance sampling[3]

Unser System befindet sich in einem Wärmebad der Temperatur T. Die möglichen Spinkonfigurationen des Systems seien durch $[s^{(i)}]$ mit $i = 1,2,...$ gekennzeichnet. Nach genügend langer Zeit befindet sich das System mit dem Wärmebad im Gleichgewicht. Gemäss der statistischen Thermodynamik befindet sich dann das System mit der Boltzmannwahrscheinlichkeit $p[s^{(i)}]$ im Zustand i

$$p[s^{(i)}] = \frac{1}{Z}\exp(-\beta\mathcal{H}[s^{(i)}]) \qquad \text{mit} \qquad \beta = \frac{1}{k_B T}. \tag{1.14}$$

Die Zustandssumme ist dabei gegeben durch

$$Z = \sum_i \exp(-\beta\mathcal{H}[s^{(i)}]). \tag{1.15}$$

Man erhält wichtige Informationen über ein System bei einer bestimmten Temperatur durch das Mitteln über ein Ensemble (oder über eine grosse Zeitspanne eines einzelnen Systems)

$$\langle\mathcal{O}\rangle = \sum_i p[s^{(i)}]\mathcal{O}[s^{(i)}]. \tag{1.16}$$

Dabei ist $\mathcal{O}[s^{(i)}]$ eine physikalische Observable des Systems im Zustand i. In der Zustandssumme wird über alle möglichen Zustände summiert. Für sehr kleine Systeme kann man sie und damit $\langle\mathcal{O}\rangle$ berechnen, doch wächst die Anzahl der möglichen Zustände Q im Ising Modell nach der Formel

$$Q = 2^{L^d}, \tag{1.17}$$

wobei L die Anzahl Spins in einer Dimension und d die Anzahl Dimensionen bedeuten. Man müsste also schon bei nicht besonders grossen Systemen unvorstellbar grosse Summen ausführen, wozu auch der schnellste Computer nicht in der Lage ist. Wenn man das System nicht analytisch berechnen kann, wie das bei mehreren Dimensionen der Fall ist, stellt sich also die Frage nach einer anderen Methode.
Es ist klar, dass sehr viele Zustände nur mit sehr kleiner Wahrscheinlichkeit beitragen, zum Beispiel alle Zustände hoher Energie, wenn die Temperatur des Systems klein ist. Eine Möglichkeit ist nun, eine Art Experiment auf dem Computer zu machen (das System zu simulieren) und dabei die „typischen“ Zustände zu „messen“ und diese zu mitteln. Genauer gesagt, misst man N zufällige Zustände. Ein bestimmter Zustand wird in dieser Auswahl ungefähr $Np[s^{(i)}]$ mal vertreten sein. Dann ist

$$\overline{\mathcal{O}} \equiv \frac{1}{N}\sum_{i=1}^{N}\mathcal{O}[s^{(i)}] \tag{1.18}$$

ein erwartungstreuer Schätzer für $\langle\mathcal{O}\rangle$ d.h. $\langle\overline{\mathcal{O}}\rangle = \langle\mathcal{O}\rangle$. In einem grossen System treten die meisten Summanden in Formel (1.16) mit vernachlässigbarer Wahrscheinlichkeit auf, wobei

[3] Dieses und auch nachfolgende Themen sind recht allgemein, werden hier aber am Beispiel des Ising Modells erläutert.

die in Formel (1.18) auftretenden Zustände *typisch* sind. Daher muss N nicht extrem gross sein, ein typischer Wert bei einer Simulation ist eine Million, wobei dies vom System und von der geforderten Genauigkeit abhängt (siehe Abschnitt 1.9). Wichtig ist, dass die Stichprobe hinreichend gross ist, damit die Wahrscheinlichkeit, dass ein Element der Stichprobe im Zustand $s^{(i)}$ ist, durch $p[s^{(i)}]$ gegeben ist. Der eben beschriebene Trick nennt sich *importance sampling*. Anstatt dem exakten Mittelwert (1.16), wird also der Schätzer (1.18) durch eine Simulation bestimmt. Dies geschieht mit Monte Carlo Methoden.

1.4 Monte Carlo Methoden

Monte Carlo Methoden bzw. Simulationen sind sehr allgemeine Begriffe. Werden bei einer Simulation gewisse Grössen zufällig ausgewählt, spricht man von einer Monte Carlo Simulation. Monte Carlo Methoden werden beispielsweise auch in der Wahrscheinlichkeitstheorie oder zur näherungsweisen Berechnung von Integralen verwendet. Sie stellen aber auch eine starke numerische Technik dar, die in der statistischen Physik verwendet wird. In dieser Arbeit wurden zwei Monte Carlo Algorithmen, der Metropolis- und der Swendsen-Wang Clusteralgorithmus, verwendet, deren Eigenschaften aus dem allgemeineren Konzept der Markovketten folgen.

1.5 Markovketten

Betrachten wir ein System mit einer diskreten Anzahl von möglichen Zuständen und ein unendlich grosses Ensemble von solchen Systemen. (In den folgenden Erläuterungen ist es wichtig, zwischen der Wahrscheinlichkeitsverteilung im unendlich grossen Ensemble und einer konkreten Realisierung eines Systems mit bestimmten Zuständen zu unterscheiden.)[4] Das System evolviert in diskreten Zeitschritten und die Evolution ist zufällig. Der Prozess wird durch Übergangswahrscheinlichkeiten $w(i \to j)$ beschrieben, wobei

$$0 \leq w(i \to j) \leq 1, \qquad \sum_j w(i \to j) = 1 \tag{1.19}$$

gilt. Die Wahrscheinlichkeitsverteilung im Ensemble zur Zeit $T = k$ hängt nur von der Verteilung zur Zeit $T = k - 1$ und von den Übergangswahrscheinlichkeiten $w(i \to j)$ ab, welche zeitunabhängig sind. Man erhält also für ein System eine Markovkette von Zuständen

$$[s^{(1)}] \to [s^{(2)}] \to \ldots \to [s^{(N)}], \tag{1.20}$$

durch die Anwendung eines Algorithmus, welcher aus dem Zustand $[s^{(i)}]$ den Zustand $[s^{(i+1)}]$ generiert. Analog evolviert die Wahrscheinlichkeitsverteilung des Ensembles in einer Markovkette. Die Wahrscheinlichkeitsverteilung zur Zeit $t = k$ ist gegeben durch die *master equation*

$$p_j^{(k)} = \sum_i p_i^{(k-1)} w(i \to j). \tag{1.21}$$

Um dies als Matrix-Vektorprodukt zu schreiben, wird die Matrix w mit den Elementen

$$w_{ji} \equiv w(i \to j) \tag{1.22}$$

[4] Die Argumente in den runden Klammern bedeuten dabei Wahrscheinlichkeitsverteilungen im Ensemble, die in den Eckigen konkrete Zustände.

eingeführt. (Man beachte die Reihenfolge der Indizes.) Nun kann man die Gleichung (1.21) schreiben als

$$p^{(k)} = wp^{(k-1)}, \tag{1.23}$$

wobei die Wahrscheinlichkeitsverteilung $\{p_i\}$ als Vektor p betrachtet wird. Die Verteilung zur Zeit $t = n$ erhält man aus dem Anfangszustand durch das n-malige Anwenden der Übergangsmatrix auf die Anfangsverteilung

$$p^{(n)} = w^n p^{(0)}. \tag{1.24}$$

Nun sind wir an folgenden Dingen interessiert:

- Existiert der Limes $n \to \infty$ und wenn ja, ist dies eine stationäre Verteilung d.h. $P = \lim_{n\to\infty} p^{(n)}$ und $wP = P$?
- Ist die stationäre Verteilung eindeutig, d.h., unabhängig von der Anfangsverteilung $p^{(0)}$?

In [2] wird gezeigt, dass für eine ergodische Markovkette der Limes existiert, die Verteilung stationär und eindeutig ist. Man findet auch, dass die Abhängigkeit von der Anfangsverteilung exponentiell abnimmt. Es kann also eine beliebige Anfangskonfiguration gewählt werden. Da wir nur an den Gleichgewichtszuständen interessiert sind, müssen wir einfach so lange warten, bis der Einfluss der Anfangskonfiguration vernachlässigbar klein ist und alle vorherigen Zustände von der Betrachtung ausschliessen.

1.6 Ergodizität und detailliertes Gleichgewicht

Ein Algorithmus ist *ergodisch*, wenn man von einem beliebigen Zustand aus alle anderen Zustände mit einer endlichen Anzahl Schritte erreichen kann. Die Bedingung für detailliertes Gleichgewicht lautet

$$p[s]w[s \to s'] = p[s']w[s' \to s], \tag{1.25}$$

wobei sich die beiden Zustände $[s]$ und $[s']$ nur durch einen Algorithmusschritt unterscheiden, $p[s]$ und $p[s']$ die Wahrscheinlichkeiten der jeweiligen Zustände und $w[s \to s']$ bzw. $w[s' \to s]$ die Übergangswahrscheinlichkeiten bedeuten. Im Folgenden zeigen wir, dass die Boltzmannverteilung

$$p[s] = \frac{1}{Z}\exp(-\beta\mathcal{H}[s]) \tag{1.26}$$

ein Eigenvektor (pro Zeile ein Zustand) der Matrix $w[s', s]$[5] ist, falls der Algorithmus detailliertes Gleichgeweicht erfüllt. Aus der Bedingung für detailliertes Gleichgewicht und durch Kürzen des Faktors $1/Z$ folgt

$$\begin{aligned}\sum_{[s]} \exp(-\beta\mathcal{H}[s])w[s', s] &= \sum_{[s]} \exp(-\beta\mathcal{H}[s'])w[s, s'] \\ &= \exp(-\beta\mathcal{H}[s'])\sum_{[s]} w[s, s'] \\ &= \exp(-\beta\mathcal{H}[s']). \end{aligned} \tag{1.27}$$

Am Schluss wurde verwendet, dass die Summe über alle Wahrscheinlichkeiten eins ergibt. Um zu zeigen, dass ein Algorithmus zur Boltzmannverteilung konvergiert, reicht es somit zu

[5] Hier soll entsprechend zu vorhin $w[s \to s'] \equiv w[s', s]$ bedeuten.

zeigen, dass dieser Ergodizität und detailliertes Gleichgewicht erfüllt. Denn aus der Ergodizität folgt, dass eine stationäre Verteilung existiert und eindeutig (und somit unabhängig vom Anfangszustand) ist. Aus dem detaillierten Gleichgewicht folgt, dass die Boltzmannverteilung eine stationäre Verteilung ist. Zusammen resultiert also die Boltzmannverteilung.

1.7 Metropolisalgorithmus

Dies ist ein relativ einfacher Algorithmus. Die neue Spinkonfiguration $[s']$ wird aus der alten $[s]$ durch das folgende Vorgehen erzeugt:

- Ein Spin wird per Zufall ausgewählt und es wird „vorgeschlagen“, diesen zu flipen (d.h. sein Vorzeichen zu wechseln).
- Es wird berechnet, wie gross die Energiedifferenz $\mathcal{H}[s'] - \mathcal{H}[s]$ bei einem Wechsel zu dieser neuen Konfiguration wäre.
- Falls die Energie der neuen Konfiguration kleiner oder gleich ist wie die der alten, wird die neue Konfiguration mit der Wahrscheinlichkeit 1 angenommen. Ist die Energie der neuen Konfiguration dagegen grösser, wird die neue Konfiguration nur mit der Wahrscheinlichkeit $\exp(-\beta(\mathcal{H}[s']-\mathcal{H}[s]))$ angenommen, ansonsten wird die alte Konfiguration beibehalten, jedoch trotzdem als neue gezählt.

Die neue Konfiguration unterscheidet sich von der alten also höchstens um eine Spinrichtung, es kommt sogar oft vor, dass die neue Konfiguration gleich der alten ist. Die Übergangswahrscheinlichkeit $w[s', s]$ ist also das Produkt aus $1/L^d$, da jeder Spin mit der gleichen Wahrscheinlichkeit ausgewählt wird und der jeweiligen Annahmewahrscheinlichkeit. Man erhält

$$w[s', s] = \frac{1}{L^d}\min\{1, \exp(-\beta(\mathcal{H}[s'] - \mathcal{H}[s])\}. \tag{1.28}$$

Eine wichtige Beobachtung ist, dass während die Energie des Systems von allen Spins abhängt, für die Berechnung der Energiedifferenz hingegen nur der zu flipende Spin und seine nächsten Nachbarn zu berücksichtigen sind, alle anderen kürzen sich heraus. Dies kommt dadurch zustande, dass beim Ising Modell nur die nächsten Nachbarn miteinander koppeln. Ein wichtiger Begriff ist der *Sweep*. Ein Sweep entspricht L^d möglichen Flips. Erst nach mindestens einem ganzen Sweep werden Messungen vorgenommen.
Der Metropolisalgorithmus ist ergodisch, denn da die Spins per Zufall ausgewählt werden, ist klar, dass, zumindest im Prinzip, jede Spinkonfiguration erreicht werden kann. Um zu zeigen, dass er auch detailliertes Gleichgewicht erfüllt, betrachten wir zwei beliebige Konfigurationen $[s]$ und $[s']$. Man kann ohne Beschränkung der Allgemeinheit annehmen, dass $\mathcal{H}[s'] \leq \mathcal{H}[s]$ und somit $w[s', s] = 1$ gilt. Dann ist $w[s, s'] = \exp(-\beta(\mathcal{H}[s] - \mathcal{H}[s']))$. Nun gilt

$$\begin{aligned} \exp(-\beta\mathcal{H}[s])w[s', s] &= \exp(-\beta\mathcal{H}[s]) \\ &= \exp(-\beta\mathcal{H}[s'])\exp(-\beta(\mathcal{H}[s] - \mathcal{H}[s'])) \\ &= \exp(-\beta\mathcal{H}[s'])w[s, s'], \end{aligned} \tag{1.29}$$

was der Bedingung für detailliertes Gleichgewicht entspricht.

1.8 Swendsen-Wang Clusteralgorithmus

In diesem Algorithmus ist kein äusseres Magnetfeld vorgesehen. Entsprechend der aktuellen Spinkonfiguration werden zwischen jeweils zwei Spins sogenannte *Bonds* mit den folgenden Wahrscheinlichkeiten gesetzt bzw. nicht gesetzt.

Parallele Spins:

$$p_p = 1 - \exp(-2\beta J), \qquad \overline{p_p} = \exp(-2\beta J) \tag{1.30}$$

Antiparallele Spins:

$$p_a = 0, \qquad \overline{p_a} = 1 \tag{1.31}$$

Zwischen antiparlallelen Spins wird also nie ein Bond gesetzt. Alle Spins, die danach durch Bonds verbunden sind, gehören zu ein und demselben *Cluster*. Abbildung 1.3 veranschaulicht die Clusterbildung durch das Bondsetzen. Alle Spins in einem Cluster sind somit parallel,

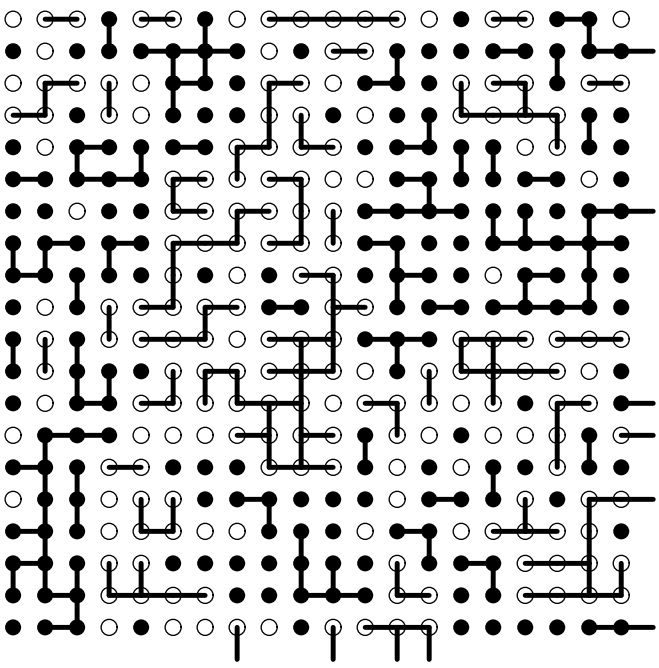


Abbildung 1.3: Clusterbildung durch das Setzen von Bonds. Weisse Kreisflächen bedeuten Spin = 1, schwarze Spin = -1.

in der Regel sind die Cluster jedoch kleiner als die ganzen Regionen gleichgerichteter Spins. Spins, die durch keinen Bond mit anderen Spins verbunden sind, bilden ihren eigenen Cluster. Somit gehört jeder Spin zu genau einem Cluster. Anschliessend wird jeder Cluster mit einer Wahrscheinlichkeit von 50% geflipt. Dies schliesst einen Sweep ab und es können neue Bonds gesetzt werden. Spins im gleichen Cluster sind also korreliert, Spins in unterschiedlichen Clusters hingegen nicht.
Auch dieser Algorithmus ist ergodisch. Mit einer (eventuell sehr kleinen) Wahrscheinlichkeit werden gar keine Bonds gesetzt. Dann bildet jeder Spin seinen eigenen Cluster. Durch das Flipen dieser individuellen Spins kann jede beliebige Spinkonfiguration erreicht werden. Für

das detaillierte Gleichgewicht soll nur eine Beweisskizze gegeben werden, da es sich eher um eine Buchhaltungsübung handelt. Es reicht, ein nachbarliches Spinpaar in einer alten und neuen Spinkonfiguration zu betrachten. Es existieren $2^4 = 16$ Möglichkeiten, wobei man nur 8 betrachten muss, da bei jeweils zwei nur die Orientierung vertauscht ist. Anschliessend kann man für alle Möglichkeiten die Bedingung für detailliertes Gleichgewicht nach (1.25) aufstellen und mit den Wahrscheinlichkeiten für das Bondsetzen bzw. nicht Setzen nach (1.30) und (1.31) und der Wahrscheinlichkeit $1/2$, dass ein Cluster geflipt wird, durchspielen.
Mit dem Clusteralgorithmus kann die Clusterrepräsentation der Suszeptibilität verwendet werden, welche nun hergeleitet wird. Die totale Magnetisierung einer Konfiguration ist die Summe von Clustermagnetisierungen

$$\mathcal{M}[s] = \sum_x s_x = \sum_C \mathcal{M}_C, \tag{1.32}$$

wobei die Clustermagnetisierung gegeben ist durch

$$\mathcal{M}_C = \sum_{x \in C} s_x. \tag{1.33}$$

In einem endlichen Volumen verschwindet die mittlere Magnetisierung immer

$$\langle \mathcal{M} \rangle = \langle \sum_C \mathcal{M}_C \rangle = 0. \tag{1.34}$$

Damit kann die Suszeptibilität nach der Definition (1.6) ausgedrückt werden durch

$$\begin{aligned} \chi &= \frac{1}{L^d} \langle \mathcal{M}^2 \rangle \\ &= \frac{1}{L^d} \langle (\sum_C \mathcal{M}_C)^2 \rangle \\ &= \frac{1}{L^d} \langle \sum_C \mathcal{M}_C^2 \rangle. \end{aligned} \tag{1.35}$$

Im letzten Schritt wurde verwendet, dass zwei verschiedene Cluster C_i und C_j unabhängig sind und daher $\langle \mathcal{M}_{C_i} \mathcal{M}_{C_j} \rangle = \langle \mathcal{M}_{C_i} \rangle \langle \mathcal{M}_{C_i} \rangle$ gilt und diese Summanden dadurch verschwinden. Da alle Spins in einem Cluster parallel sind, ist die Clustermagnetisierung bis auf ein Vorzeichen durch die Clustergrösse $|C|$ gegeben

$$\mathcal{M}_C = \pm |C| = \pm \sum_{x \in C} 1. \tag{1.36}$$

Somit kann die Suszeptibilität ausgedrückt werden durch

$$\chi = \frac{1}{L^d} \langle \sum_C |C|^2 \rangle. \tag{1.37}$$

Dies zeigt, dass die Clustergrösse direkt mit einer physikalischen Grösse verknüpft ist und die Cluster in diesem Sinne physikalische Objekte sind. Die Gleichung zeigt auch, dass $|C|$ mit χ wächst und so am kritischen Punkt divergiert. Anstatt wie beim Metropolisalgorithmus nur die Magnetisierung einer Konfiguration zu messen, werden alle Quadrate der Clustergrössen gemessen und summiert. Dadurch wird die Statistik um den Faktor 2^{N_c} vergrössert, wobei N_c die Anzahl der Cluster der Konfiguration ist. Dies macht den Schätzer (1.37) zu einem sogenannten *improved estimator*, welcher deutlich genauere Werte liefert.

1.9 Fehleranalyse[6]

Da jede Monte Carlo Simulation eine endliche Länge hat, was in unserem Fall eine endliche Anzahl Sweeps bedeutet, sind die Resultate mit statistischen Fehlern behaftet. Deshalb ist die Fehleranalyse ein wichtiger Teil jeder Monte Carlo Simulation. Wenn die Monte Carlo Daten für eine Observable Gauss-verteilt sind, was in unserem Fall so ist, ist der Schätzer für die Standardabweichung

$$\triangle\mathcal{O} = \sqrt{\frac{1}{N-1}(\langle\mathcal{O}^2\rangle - \langle\mathcal{O}\rangle^2)}, \tag{1.38}$$

wobei N die Anzahl verwendeter Gleichgewichtssweeps ist. Es ist ersichtlich, dass, um den Fehler um eine Nachkommastelle zu reduzieren, N um den Faktor hundert erhöht werden muss, wodurch sich die benötigte Computerzeit um denselben Faktor erhöht.
In unseren Simulationen sind wir hauptsächlich an der Suszeptibilität interessiert, welche nach der Definition (1.6) von $\langle\mathcal{M}\rangle$ und $\langle\mathcal{M}^2\rangle$ abhängt. Somit berechnet sich der Fehler der Suszeptibilität nach dem Gaussschen Fehlerfortpflanzungsgesetz. Nun stimmt die Berechnung des Fehlers nach (1.38) jedoch nur für unabhängige Messungen. Ein idealer Monte Carlo Algorithmus würde eine Markovkette von unabhängigen Konfigurationen produzieren. Kein realer Monte Carlo Algorithmus ist jedoch ideal. Die neue Konfiguration wird aus der alten generiert, wodurch die Konfigurationen immer mehr oder weniger korreliert sind. Dass die Messungen korreliert sind, bedeutet im Prinzip, dass man weniger Daten hat, da diese ja zum Teil fast identisch sind, und damit eine kleinere Statistik und so einen grösseren Fehler (siehe auch Abschnitt 1.10). Eine Möglichkeit, den realen Fehler einer Observablen zu berechnen, ist, die Daten zu Bins zusammenzufassen.

$$\underbrace{\underbrace{\mathcal{O}_1 \quad \mathcal{O}_2}_{\mathcal{O}_1} \quad \underbrace{\mathcal{O}_3 \quad \mathcal{O}_4}_{\mathcal{O}_2}}_{\mathcal{O}_1} \quad \underbrace{\underbrace{\mathcal{O}_5 \quad \mathcal{O}_6}_{\mathcal{O}_3} \quad \underbrace{\mathcal{O}_7 \quad \mathcal{O}_8}_{\mathcal{O}_4}}_{\mathcal{O}_2} \quad \dots$$

$$\vdots$$

Jeweils zwei aufeinanderfolgende Messwerte werden zu einem neuen gemittelt. Mit den erhaltenen Daten wird erneut der Fehler nach Formel (1.38) berechnet. Normalerweise ist dieser Fehler etwas grösser und näher am realen Fehler. In der Regel werden die neuen Daten jedoch immer noch korreliert sein und müssen erneut einem Binning unterzogen und der Fehler neu berechnet werden. Dadurch wächst der Fehler mit jedem Binning, bis die Daten nicht mehr korreliert sind und der Fehler ein Plateau erreicht und sich so dem realen Fehler annähert. Der Fehler der Suszeptibilität hat sein Plateau erreicht, wenn sowohl der Fehler von $\langle\mathcal{M}\rangle$ als auch der von $\langle\mathcal{M}^2\rangle$ ihre Plateaus erreicht haben. Abbildung 1.4 zeigt die Entwicklung eines Fehlers. Es ist zu beachten, dass (1.38) ein Schätzer ist und somit nach erreichen des Plateaus fluktuiert. Mit dem Fehler auf dem Plateau hat man jedoch eine gute Abschätzung des realen Fehlers. Auf der Abbildung ist auch klar zu sehen, dass für eine verlässliche Berechnung des Schätzers (1.38) die Anzahl Messwerte deutlich grösser als eins sein muss, sonst kann diese statistische Methode nicht richtig funktionieren.

[6] Die hier erläuterte Fehleranalyse ist eine mögliche Variante von vielen.

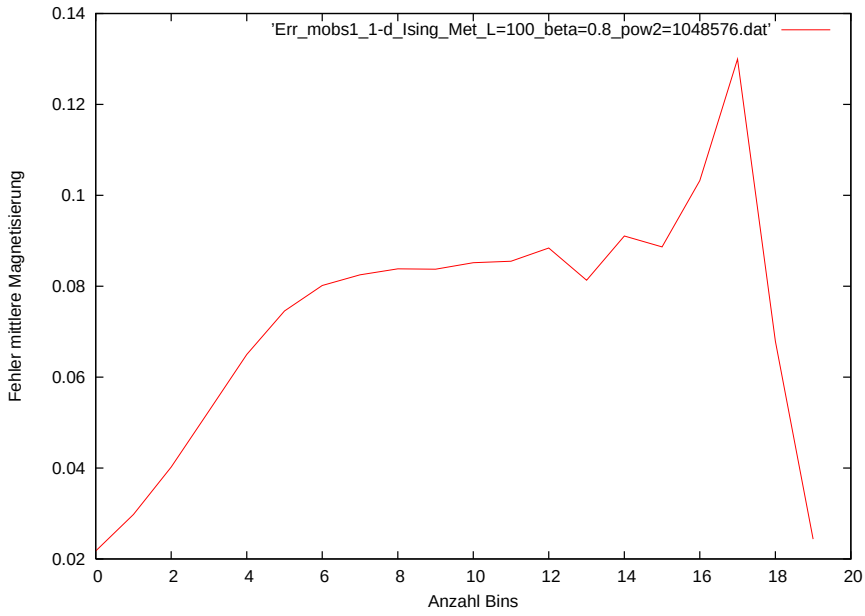


Abbildung 1.4: Entwicklung eines Fehlers während des Binningverfahrens.

1.10 Autokorrelationsfunktion und critical slowing down

Um abzuschätzen, wie viele Monte Carlo Schritte zwischen statistisch unabhängigen Konfigurationen liegen, kann man die Autokorrelationsfunktion einer Observablen

$$\langle \mathcal{O}^{(i)} \mathcal{O}^{(i+t)} \rangle = \lim_{N\to\infty} \frac{1}{N-t} \sum_{i=1}^{N-t} \mathcal{O}[s^{(i)}]\mathcal{O}[s^{(i+t)}] - \langle \mathcal{O} \rangle^2 \propto \exp(-t/\tau) \tag{1.39}$$

berechnen. Dabei bedeutet N wieder die Anzahl Sweeps im Gleichgewicht und τ die *Autokorrelationszeit*, wobei $s^{(k)}$ und $s^{(k+\tau)}$ als mehr oder weniger unabhängig betrachtet werden können. Somit hat man von N Sweeps nur N/τ unabhängige Konfigurationen. Als Konsequenz ist der reale Fehler ungefähr um den Faktor $\sqrt{\tau}$ grösser als der nach (1.38) berechnete Fehler der Rohdaten.
Wenn man sich einem Phasenübergang zweiter Ordnung nähert, divergiert die Korrelationslänge ξ und auch die Korrelationszeit. Man findet das sogenannte *critical slowing down*

$$\tau \propto \xi^z, \tag{1.40}$$

wobei z ein dynamischer kritischer Exponent ist, der die Effizienz eines Monte Carlo Algorithmus' charakterisiert. In [4] findet man für die in dieser Arbeit implementierten Algorithmen die folgenden, empirisch ermittelten Werte für den Exponenten z:

Dimension d	Metropolis	Swendsen-Wang
2	2.167± 0.001	0.25± 0.01
3	2.02± 0.02	0.54± 0.02

Dies stellt einen wesentlichen Unterschied bezüglich der Effizienz der beiden Algorithmen dar. Bei einer typischen Systemgrösse von $L = 100$ erreicht die Korrelationslänge etwa eine Grösse von 100. Somit benötigt der Metropolisalgorithmus in einem zweidimensionalen System

ungefähr 10000 Iterationen, um eine neue Konfiguration zu generieren. Stochastische lokale Updatingmethoden wie der Metropolisalgorithmus, haben in der Regel $z \approx 2$ (Vergleiche mit dem random walk, wo $\triangle x \propto \sqrt{t}$ ist). Dagegen ist der Clusteralgorithmus ein nichtlokaler stochastischer Updatingalgorithmus, welcher Regionen der linearen Grösse $O(\xi)$ updatet. Somit wächst die typische Clustergrösse, wenn die Korrelationslänge wächst (und divergiert am kritischen Punkt).

Kapitel 2

Implementierung auf dem Computer

Für meine Simulationen habe ich die Programmiersprache C++ verwendet und je ein Programm mit dem Metropolisalgorithmus für das ein- und zweidimensionale Modell, sowie eines mit dem Clusteralgorithmus für das zweidimensionale Modell geschrieben. Dazu kommt ein separates Fehleranalyseprogramm und ein Programm für die Autokorrelationsfunktion. Hier soll nur auf einige Aspekte der Implementierung und ein paar Besonderheiten zur Programmierung eingegangen werden. Für Details und programmiertechnische Dinge sind die Programmcodes im Anhang zu konsultieren, welche alle recht ausführlich kommentiert sind (grau = Kommentar). In den Simulationen wurde die Kopplungskonstante $J = 1$ und die Permeabilität $\mu = 1$ gesetzt. Damit ergibt sich für β_c mit (1.13)

$$\beta_c \approx 0.447. \tag{2.1}$$

Dem äusseren Magnetfeld B wurden beim Experimentieren auch Werte ungleich null zugeordnet, bei den folgenden Resultaten ist es jedoch stets auf null gesetzt.
Wie im Theorieteil erläutert, hat die Wahl der Anfangskonfiguration keinen Einfluss auf die Gleichgewichtsverteilung. Ich habe jeweils die Konfiguration mit allen Spins nach oben (+1) gewählt. Bei den Simulationen werden sehr viele Zufallszahlen benötigt und es ist wichtig, einen qualitativ hochstehenden Pseudozufallszahlengenerator zu verwenden, da sich sonst die Zahlenfolgen wiederholen können. Es wurde der bereits bestehende Pseudozufallszahlengenerator „RanLux“ (Version 3.0) von Martin Lüscher verwendet, welcher in C++ geschrieben ist und gleichverteilte Zufallszahlen im Intervall $]0;1[$ erzeugt. Dieser musste den Programmierprojekten nur noch hinzugefügt und entsprechend aufgerufen werden. Es handelt sich um einen *Pseudo*zufallszahlengenerator, d.h. er ist deterministisch. Für eine bestimmte Initialisierung liefert er also immer die gleiche Zahlenfolge.
Wichtig ist bei allen Programmen für das Ising Modell die Definition der nächsten Nachbarn. Dabei wird am Anfang des Programms jedem Spin in jeder Dimension ein forward und ein backward Nachbar zugeordnet. Diese können dann an jeder Stelle des Programms aufgerufen werden und man braucht sich nicht mehr darum zu kümmern. In zwei Dimensionen ist diese Definition für beliebige Gittergrössen etwas trickreich, man kann dazu die Modulofunktion verwenden (siehe Programmcode B.2). Die Spins kennzeichnet man am besten mit einer durchgehenden Nummerierung nach dem folgenden Schema am Beispiel eines 3 mal 3 Gitters:

$$\begin{array}{ccc} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{array}$$

Wie bereits erwähnt, ist es beim Metropolisalgorithmus von der Effizienz her wichtig, dass bei der Berechnung der Energiedifferenz nicht immer über alle nächsten Nachbarn summiert wird, denn alle, ausser der zu flipende Spin und seine nächsten Nachbarn, kürzen sich heraus.
Nach jedem Sweep wird die Magnetisierung gemessen und am Ende des jeweiligen Programms werden alle Werte in ein File geschrieben, um auch nachträgliche Auswertungen vornehmen zu können. Beim Clusteralgorithmus passiert das gleiche zusätzlich mit den $\sum_C |C|^2$-Werten. Die Programme werten die Daten jedoch gleich aus machen auch eine Fehleranalyse.
Da die Konfigurationen am Anfang nicht Gleichgewichtskonfigurationen sind, dürfen diese nicht verwendet werden. Um herauszufinden, nach wie vielen Sweeps sich das System im Gleichgewicht befindet, kann die Magnetisierung verwendet werden. Dazu betrachtet man am besten eine Visualisierung der Magnetisierungsdaten wie in Abbildung 2.1. Im Gleichgewicht

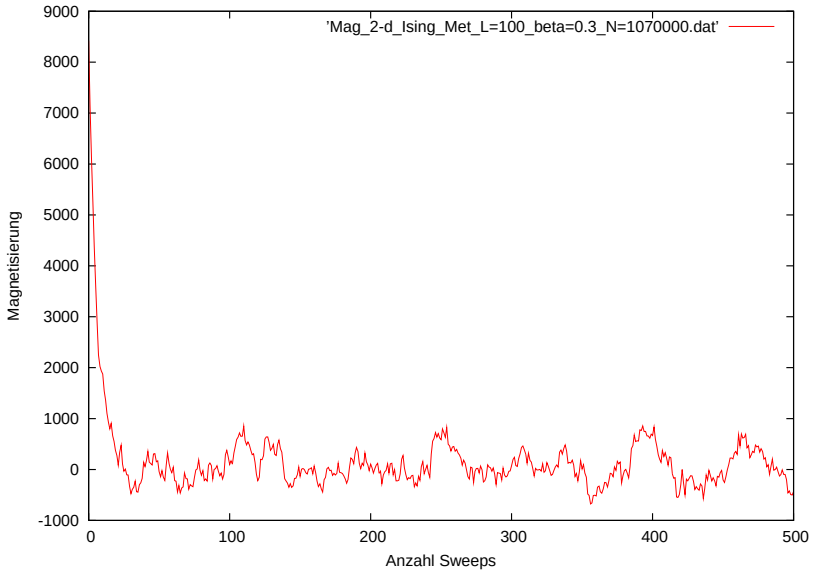


Abbildung 2.1: Visualisierung von Magnetisierungsdaten.

fluktuiert die Magnetisierung um null. Es ist zu erkennen, dass etwa die ersten 25 Magnetisierungswerte nicht Gleichgewichtswerte sind. Um sicher zu gehen, schneidet man noch um ein paar Faktoren mehr Werte ab. Im diesem Zusammenhang ist zu sagen, dass bei der Fehleranalyse ja das Binningverfahren verwendet wird, wo immer zwei Werte zusammen gemittelt und diese wieder gemittelt werden usw. Daher ist es praktisch, eine Anzahl von Daten zu haben, die eine Potenz von zwei ist. Deshalb ermittelt das Programm, wenn man ihm sagt, es solle N Sweeps machen und davon die ersten M abschneiden, die grösste Potenz von zwei, nennen wir sie R, welche kleiner oder gleich $N - M$ ist. Danach werden alle $N - R$ ersten Werte weggeworfen. Bei den Simulationen habe ich oft $N = 1070000$ und $M = 0$ gewählt. Somit werden die ersten $1070000 - 2^{20} = 21424$ Werte abgeschnitten, in der Regel also um Faktoren mehr als unbedingt nötig.
Die Simulation eines Systems von 100^2 Spins mit 1070000 Sweeps dauert auf meinem verwendeten Laptop etwa eine Stunde. Da wir an Suszeptibilitätswerten von verschiedenen Betawerten interessiert sind, befindet sich der Hauptteil des Programms in einer for-Schlaufe, wodurch gleich mehrere Durchläufe für verschiedene Betawerte gemacht werden. So ist auch sichergestellt, dass nicht immer wieder die gleichen Zufallszahlen des Zufallszahlengenerators

verwendet werden. Um auch bei verschiedenen Simulationen verschiedene Zufallszahlen zu verwenden, wird der Zufallszahlengenerator mittels CPU-Uhr initialisiert. Die Funktion time() liefert die vergangenen Sekunden seit dem 1.1.1970.

2.1 Clusteralgorithmus

Der Clusteralgorithmus ist etwas komplizierter zu implementieren als der Metropolisalgorithmus, deshalb soll seine Implementierung hier kurz erklärt werden. Für Details sei auch hier auf den Programmcode verwiesen. Der Clusteralgorithmus wurde nur für das zweidimensionale Modell implementiert. Zuerst werden von jedem Spin aus in zwei Dimensionsrichtungen mit den erwähnten Wahrscheinlichkeiten die Bonds gesetzt. Dann erhält jeder Spin die Markierung -1, was bedeutet, dass er noch keinem Cluster zugeordnet ist. Später erhalten die Spins eine Zahl, entsprechend dem Cluster, zu dem sie gehören. Anschliessend wird das Gitter sequentiell durchgegangen. Findet das Programm einen Spin, der noch keinem entdeckten Cluster zugeordnet ist, hat es einen neuen Cluster gefunden und untersucht in der Folge, welche Spins zu diesem gehören. Vom ersten Spin des entdeckten Clusters aus „schaut“ das Programm in alle vier Richtungen, ob der Spin durch einen Bond mit dem jeweiligen nächsten Nachbarn verbunden ist. Ist dies der Fall, kommt dieser auf eine Liste. Von allen Spins auf dieser Liste aus schaut das Programm wieder in alle Richtungen, ob weitere, neue Spins zu diesem Cluster gehören. So werden alle Spins auf der Liste abgearbeitet, bis der ganze Cluster detektiert ist. Dann geht das Programm weiter durch das Gitter, bis es den nächsten Spin findet, der noch in keinem entdeckten Cluster ist. So werden alle Cluster identifiziert. Anschliessend wird jeder Cluster mit der Wahrscheinlichkeit $1/2$ geflipt, wodurch man eine neue Konfiguration erhält. Damit ist ein Sweep abgeschlossen und das Bondsetzen beginnt erneut.

2.2 Fehleranalyse

Empirisch habe ich herausgefunden, dass das Binningverfahren in der Regel funktioniert (d.h. vernünftige Werte liefert), solange mindestens 512 Datenpunkte vorhanden sind. In den Simulationsprogrammen wird der Fehler jeweils mit starken Sicherheitsvorkehrungen automatisch ermittelt, damit man nicht für jeden Betawert die Fehler von Auge analysieren muss. Das Binningverfahren wird also so lange durchgeführt, wie der neu berechnete Fehler multipliziert mit 0.95 noch grösser ist als der zuvor berechnete und auch mindestens 512 Datenpunkte vorhanden sind. Sollte der Fehler einmal deutlich kleiner werden (zuvor berechneter Fehler multipliziert mit 0.95 noch grösser ist als der neu berechnete), was eventuell passieren könnte, wenn nur noch wenige Datenpunkte vorhanden sind, wird eine Fehlermeldung ausgegeben und der Fehler auf null gesetzt, nach dem Motto: Besser keine Fehlerangabe als eine falsche. Auch wenn nur noch 512 Datenpunkte übrig sind und der Fehler immer noch wächst, wird eine Fehlermeldung ausgegeben und der Fehler auf null gesetzt. Um dann die Fehleranalyse trotzdem noch von Auge vornehmen zu können und das Verhalten eines Fehlers zu untersuchen, habe ich noch ein separates Fehleranalyseprogramm geschrieben. In diesem Programm kann man auch angeben, ob die Fehleranalyse abgebrochen werden soll, sobald der Fehler das Plateau erreicht hat oder ob das Binning so lange fortgeführt werden soll, bis nur noch zwei Datenpunkte übrig sind. Daraus kann man Graphen wie in Abbildung 1.4 generieren und so die Fehlerentwicklung genauer untersuchen. Es kann die Fehleranalyse sowohl wie in den Programmen mit dem Metropolisalgorithmus als auch wie im Programm mit dem Clusteralgorithmus ausführen.

2.3 Autokorrelationsfunktion

Ein separates Autokorrelationsfunktionsprogramm kann ein File mit Magnetisierungsdaten (oder auch sonst einer Observablen) einlesen, die Autokorrelationsfunktion berechnen und die Werte in ein File schreiben. Damit ist es nach Formel (1.39) möglich, die Autokorrelationszeit zu bestimmen.

Kapitel 3

Resultate

Es soll nochmals darauf hingewiesen sein, dass die Betrachtungen kein äusseres Magnetfeld einschliessen. Daher ist, sofern der verwendete Algorithmus effizient funktionierte, die mittlere Magnetisierung jeweils ungefähr null. Bei den Plots ist zu beachten, dass die inverse Temperatur β aufgetragen ist. Im Anhang finden sich die Resultate noch in tabellarischer Form.

3.1 Thermodynamischer Limes

Unter dem thermodynamischen Limes versteht man folgendes Vorgehen: Man wählt ein kleines Magnetfeld h, lässt das Volumen nach unendlich gehen und schaltet dann das Magnetfeld ab. Mathematisch gesprochen: $\lim_{h\to 0} \lim_{V\to\infty}$. Streng genommen kommen Phasenübergänge erst bei unendlich grossem Volumen vor. Unsere Simulationen sind also nur Näherungen und es ist das Ziel, möglichst grosse Systeme zu simulieren, um physikalische Sachverhalte zu untersuchen.

3.2 Eindimensionales Ising Modell mit Metropolisalgorithmus

Bei nur einer Dimension tritt kein Phasenübergang auf. Somit passiert hier aus physikalischer Sicht nicht viel. Als Beispiel werden die Resultate eines kleinen und eines grossen Systems gezeigt (Abbildung 3.1 und 3.2). Die Resultate hängen von der Grösse des Systems ab, was in 1.2.4 bereits erwähnt wurde. Bei nur einer Dimension bestand auch die Möglichkeit, die erhaltenen Resultate mit den analytischen Werten zu vergleichen. Die gemessenen Werte stimmen gut mit den analytischen überein, die Fehlerbalken sind so klein, dass diese in den Grafiken kaum sichtbar sind. Für grössere Werte von β werden die relativen Fehler grösser, da die Messungen über grössere Zeiten korreliert sind (siehe dazu auch Abschnitt 3.3.1).

3.3 Zweidimensionales Ising Modell

Wie bereits erwähnt, tritt beim zweidimensionalen Ising Modell ein Phasenübergang zweiter Ordnung auf. Oberhalb der kritischen Temperatur ist das System in der sogenannten symmetrischen (oder ungebrochenen) Phase mit $\langle \mathcal{M} \rangle = 0$, während sich das System für $T < T_c$ in der gebrochenen (oder geordneten) Phase, in welcher eine spontane Magnetisierung auftritt, befindet und $\langle \mathcal{M} \rangle \neq 0$ ist. Die beiden Phasen sind durch einen Phasenübergang getrennt. Der

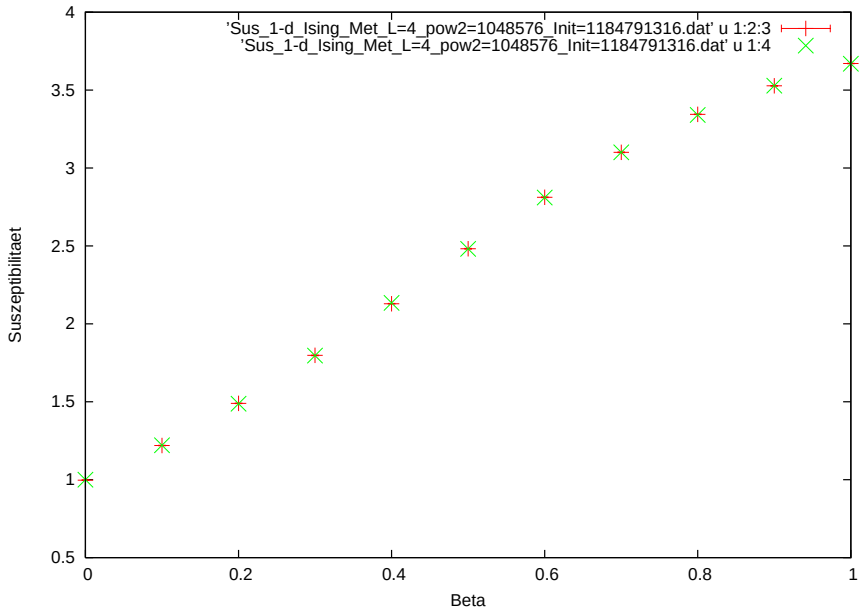


Abbildung 3.1: Eindimensionales Ising Modell mit Metropolisalgorithmus, 4 Spins. Rot: Messresultate, grün: analytische Resultate.

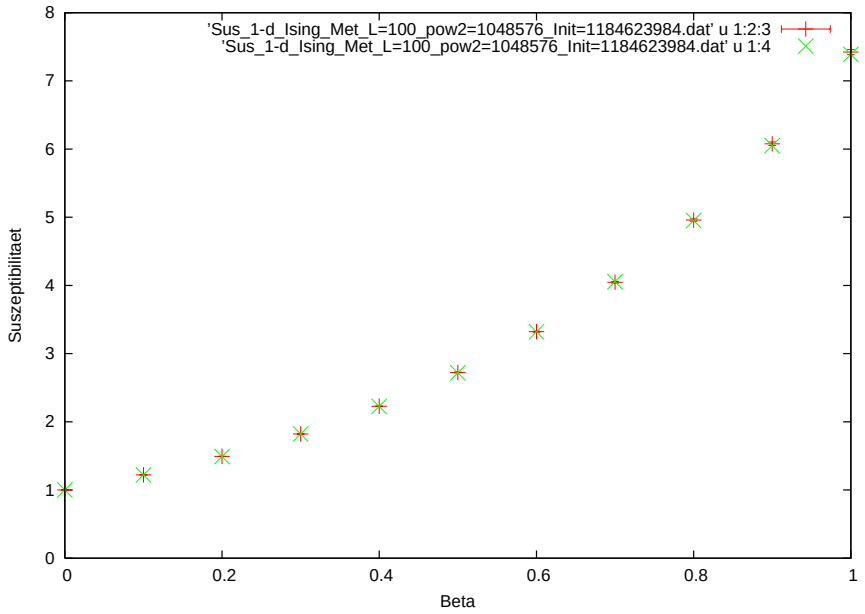


Abbildung 3.2: Eindimensionales Ising Modell mit Metropolisalgorithmus, 100 Spins. Rot: Messresultate, grün: analytische Resultate.

Wert von $\langle \mathcal{M} \rangle$ definiert, ob sich das System in der ungebrochenen oder in der geordneten Phase befindet. Deshalb nennt man $\langle \mathcal{M} \rangle$ einen *order parameter*. Bei effizienten Algorithmen wie dem Clusteralgorithmus ist $\langle \mathcal{M} \rangle$ jedoch trotzdem null, da er zwischen den beiden möglichen Magnetisierungen hin und her wechseln kann.
Weil beim zweidimensionalen Ising Modell der order parameter $\langle \mathcal{M} \rangle$ bei T_c stetig gegen null geht, handelt es sich um einen Phasenübergang zweiter Ordnung. Bei einem Phasenübergang erster Ordnung würde dort ein order parameter eine Unstetigkeit aufweisen.
Abbildung 3.3 zeigt die Zustände des Systems. Nach Formel (1.8) geht die Magnetisierung

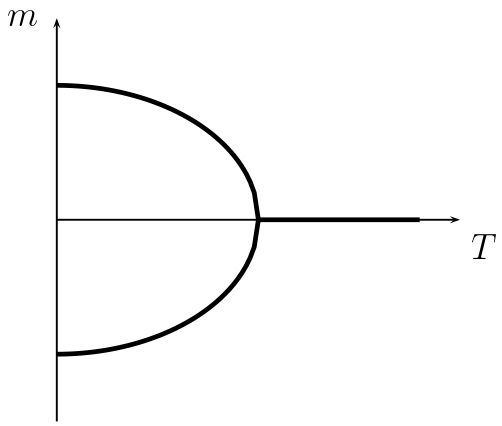


Abbildung 3.3: Qualitatives Bild der Zustände des Systems bei verschiedenen Temperaturen. T: Temperatur, m mittlere Magnetisierung des Systems oder eines einzelnen Spins. (Abbildung von [3])

wurzelartig gegen null, wobei beim zweidimensionalen Ising Modell der kritische Exponent $\beta = 1/8$ beträgt. Unterhalb der kritischen Temperatur tritt eine spontane Magnetisierung auf, was einer spontanen Symmetriebrechung entspricht. Betrachtet man nämlich die Hamiltonfunktion (1.1) ohne äusseres Magnetfeld, stellt man fest, dass die Energie einer bestimmten Spinkonfiguration genau gleich bleibt, wenn man alle Spins gleichzeitig flipt, also $\mathcal{H}[s] = \mathcal{H}[-s]$. Die Hamiltonfunktion ist somit symmetrisch bezüglich der Umkehrung aller Spins. Die Spins hätten also alle ebenso gut in die entgegengesetzte Richtung zeigen können. Deshalb spricht man von einer *spontanen* Symmetriebrechung, im Gegensatz zur *expliziten* Symmetriebrechung durch ein äusseres Magnetfeld, wodurch sich die Spins bevorzugt parallel zum Magnetfeld ausrichten und somit eine Spinrichtung ausgezeichnet wird und nicht mehr $\mathcal{H}[s] = \mathcal{H}[-s]$ gilt.

3.3.1 Zweidimensionales Ising Modell mit Metropolisalgorithmus

Die Resultate sind in den Abbildungen 3.4 und 3.5 für ein kleines und ein grosses System dargestellt. Deutlich zu erkennen ist der Phasenübergang bei $\beta_c \approx 0.447$ (theoretischer Wert), wo die Suszeptibilität stark ansteigt. Beim System mit 4^2 Spins steigt die Suszeptibilität viel langsamer an, dies ist ein Effekt des kleinen Volumens. Wie bereits erwähnt, tritt der Phasenübergang streng genommen erst bei unendlich grossem Volumen auf. Natürlich kann die Suszeptibilität in einem endlichen Volumen auch nicht wirklich divergieren, sie kann nach der Definition (1.6) höchstens $\chi = 1/L^d(\langle \mathcal{M}^2 \rangle - \langle \mathcal{M} \rangle^2) = L^d$ betragen (wenn $\langle \mathcal{M} \rangle = 0$ ist). Auffallend ist das Verhalten für Betawerte grösser als β_c. Beim grossen System fluktuiert

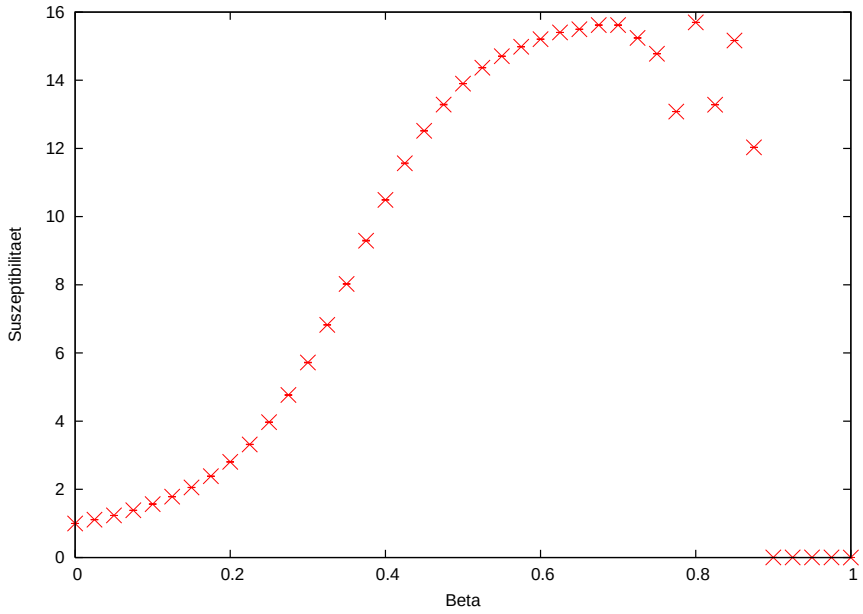


Abbildung 3.4: Zweidimensionales Ising Modell mit Metropolisalgorithmus, 4^2 Spins.

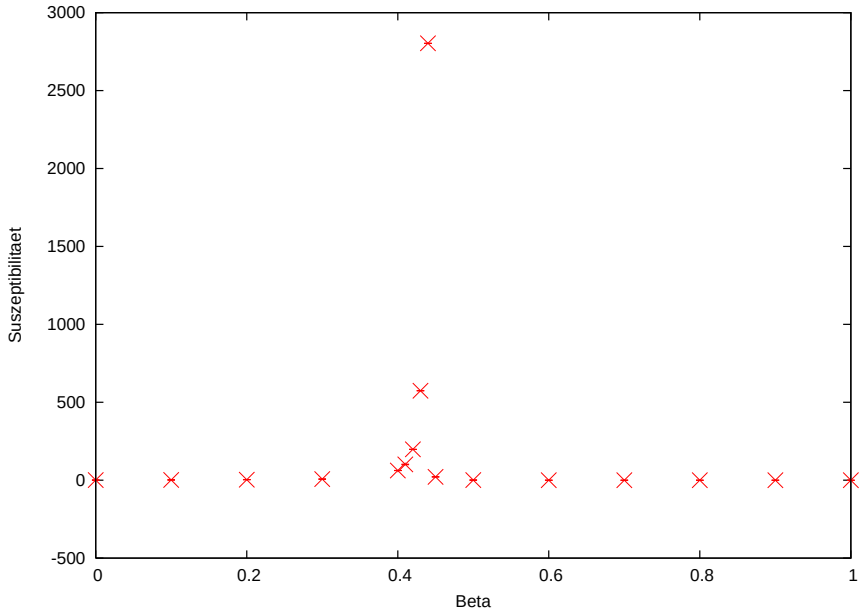


Abbildung 3.5: Zweidimensionales Ising Modell mit Metropolisalgorithmus, 100^2 Spins.

dort die Suszeptibilität um null herum. Dies lässt sich dadurch erklären, dass das System magnetisiert ist, also fast alle Spins in dieselbe Richtung zeigen. Im Prinzip wäre die andere Richtung energetisch genau gleichwertig. Der Metropolisalgorithmus versucht jedoch immer einzelne Spin zu flipen, was energetisch ungünstig ist. Meist wird der Vorschlag, einen Spin zu flipen abgelehnt, weil es zu viel Energie kostet, oder häufig wird er bei einem folgenden Sweep wieder zurückgeflipt. Mit dem Metropolisalgorithmus dauert es sehr lange, bis er es einmal schafft, in den anderen Magnetisierungszustand zu kommen. Somit ist die mittlere Magnetisierung $\langle M \rangle \approx L^d$ anstatt null und kompensiert $\langle M^2 \rangle$, wodurch die Suszeptibilität ungefähr null beträgt. Beim kleineren System ist der Metropolisalgorithmus ab und zu noch in der Lage, das ganze System zu flipen, was erklärt, dass einige Suszeptibilitätswerte nahe von β_c nicht null betragen. Dies ist in Abbildung 3.6 zu erkennen. Dabei ist zu beachten, wie

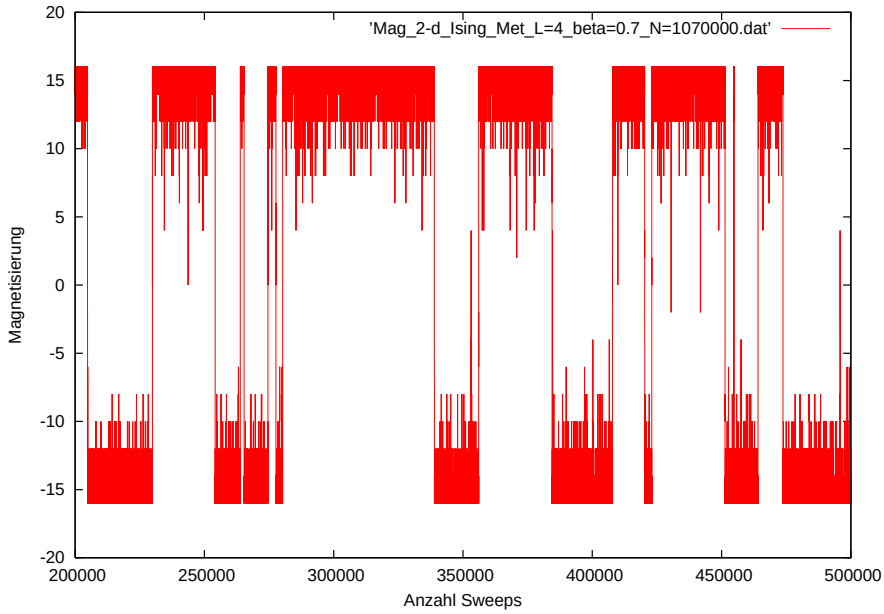


Abbildung 3.6: Ausschnitt aus den Magnetisierungsdaten eines Systems mit 4^2 Spins bei $\beta = 0.7$, generiert durch den Metropolisalgorithmus.

viele Sweeps es selbst bei diesem kleinen System benötigt, bis der Metropolisalgorithmus es schafft, das System in den anderen Magnetisierungszustand zu bringen. Bei noch grösseren Werten von β oder grösseren Systemen dauert es im Durchschnitt noch sehr viel länger.
In einiger Distanz vom kritischen Punkt sind die Fehler der Suszeptibilität sehr klein. Um den kritischen Punkt herum geschieht jedoch das, was im Kapitel über das critical slowing down (1.10) bereits erklärt wurde. Die Konfigurationen sind über sehr lange Zeiten korreliert. Das Binningverfahren wird so oft ausgeführt, bis nur noch 512 Datenpunkte übrig sind, aber die Fehler haben noch kein Plateau erreicht. Daher werden die Fehlerbalken auf null gesetzt. Dies sind gute Gründe, um zum viel effizienteren Clusteralgorithmus zu wechseln.

3.3.2 Zweidimensionales Ising Modell mit Clusteralgorithmus

Die Resultate sind wieder für ein kleines und ein grosses System in den Abbildungen 3.7 und 3.8 gezeigt. Nun bleiben die Suszeptibilitätswerte nach dem Phasenübergang gross (wie es

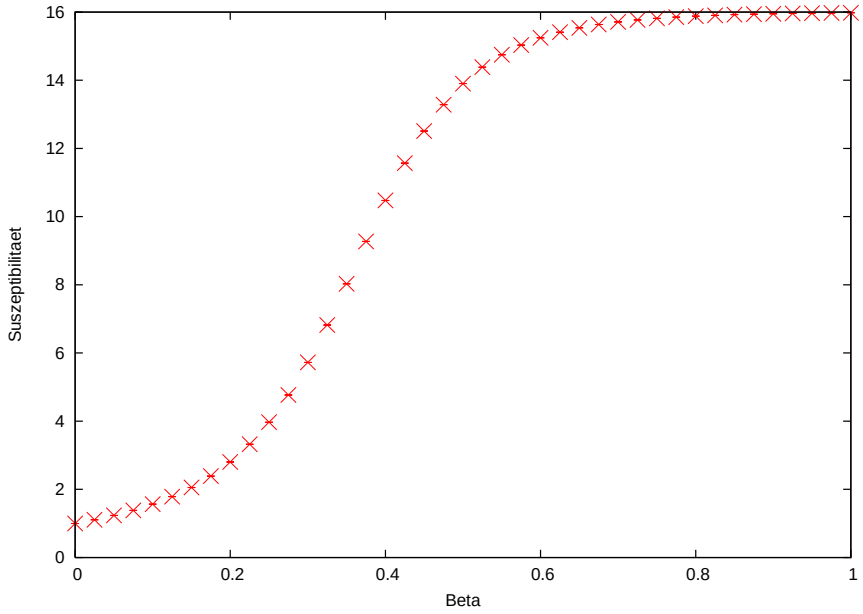


Abbildung 3.7: Zweidimensionales Ising Modell mit Clusteralgorithmus, 4^2 Spins.

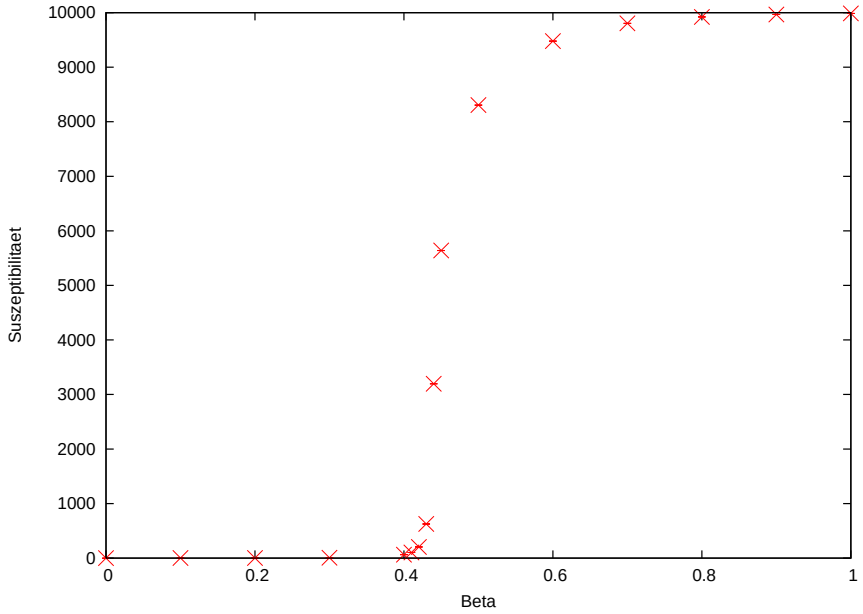


Abbildung 3.8: Zweidimensionales Ising Modell mit Clusteralgorithmus, 100^2 Spins.

auch sein sollte) und konvergieren mit grösser werdendem β gegen L^d. Die Fehler sind jetzt auch im Bereich des Phasenübergangs berechenbar und allgemein kleiner als beim Metropolisalgorithmus. Der Clusteralgorithmus liefert viel weniger korrelierte Konfigurationen.
Die Magnetisierung fluktuiert oberhalb der kritischen Temperatur um null, darunter springt der Clusteralgorithmus zwischen den beiden Magnetisierungen hin und her (siehe Abbildungen 3.9 und 3.10). Dies ist auch in den folgenden Abbildungen zu den typischen Spinkonfigurationen bei verschiedenen Temperaturen zu sehen (Abbildung 3.11). Oberhalb der kritischen Temperatur sieht das Bild ziemlich homogen aus, es hat ungefähr gleich viele Spins mit +1 wie solche mit -1 und die Regionen gleicher Spinrichtung sind sehr klein. Diese werden gegen die kritische Temperatur hin etwas grösser und werden beim Phasenübergang sehr gross (hier divergiert die Korrelationslänge theoretisch). Das endliche Volumen hat zur Folge, dass der Phasenübergang nicht schlagartig stattfindet. Unterhalb der kritischen Temperatur findet man immer eine Magnetisierung und je tiefer die Temperatur ist, desto grösser werden die Regionen gleicher Spinausrichtung. Wenn β noch grösser ist als 0.6, findet man praktisch nur noch ein paar einzelne Spins, welche antiparallel zur Magnetisierungsrichtung stehen.
In der Abbildung 3.12 ist zu sehen, wie mit wachsendem β auch die Cluster wachsen. Bei kleinen Betawerten bilden neben ein paar kleinen Clustern die meisten Spins ihren eigenen Cluster, bei $\beta = 0.6$ ist hingegen fast das ganze Gitter ein Cluster. Der Clusteralgorithmus „passt sich somit der Situation an“.

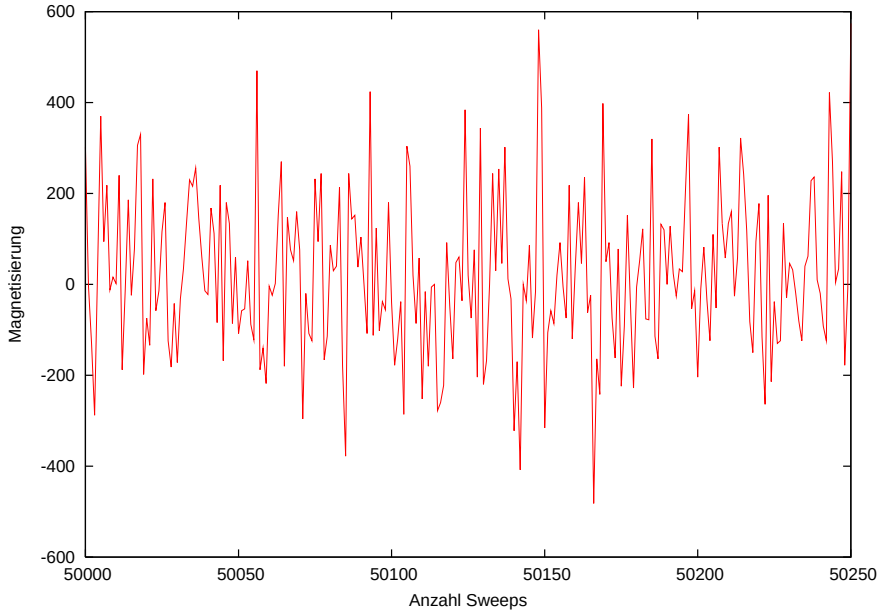


Abbildung 3.9: Ausschnitt aus den Magnetisierungsdaten eines Systems mit 100^2 Spins bei $\beta = 0.2$, generiert durch den Clusteralgorithmus.

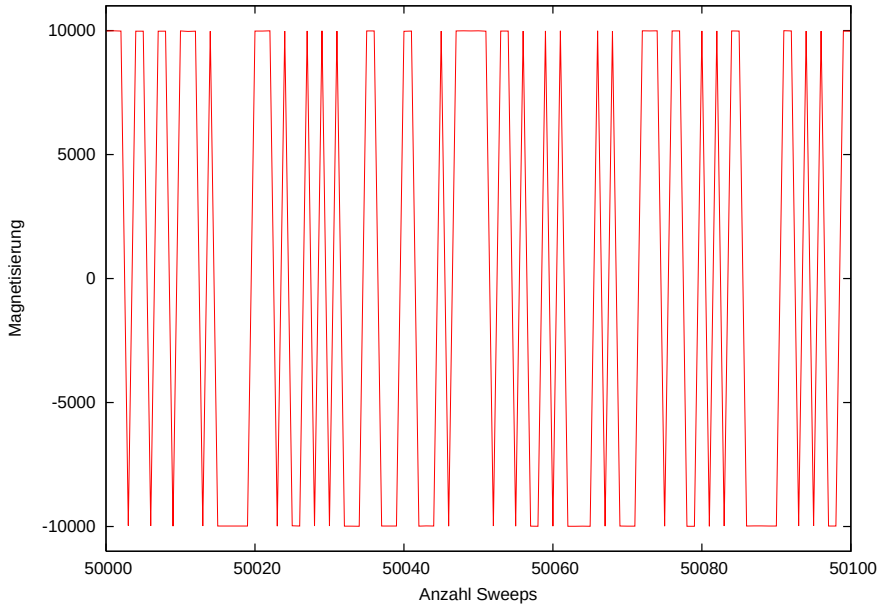
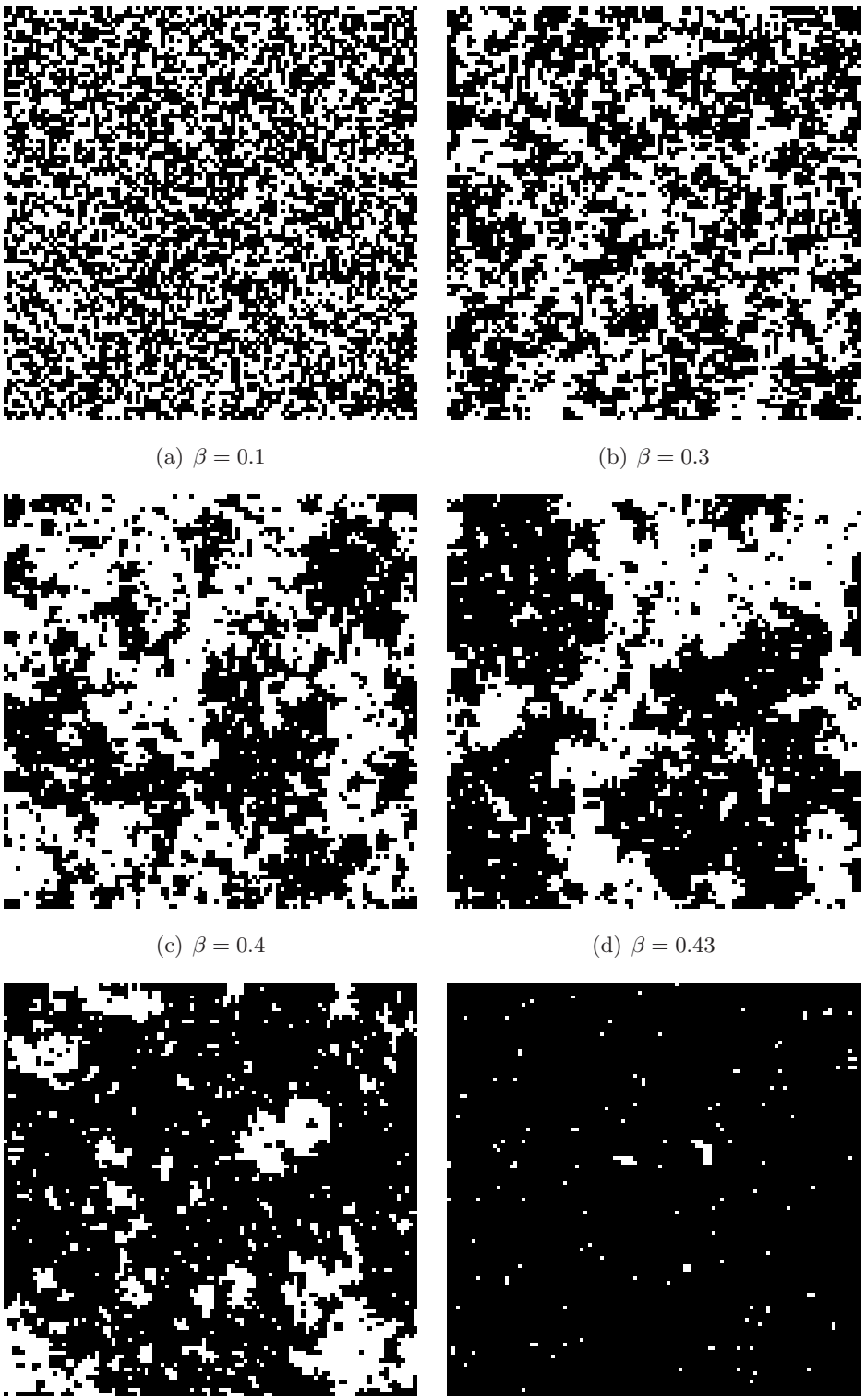


Abbildung 3.10: Ausschnitt aus den Magnetisierungsdaten eines Systems mit 100^2 Spins bei $\beta = 0.9$, generiert durch den Clusteralgorithmus.

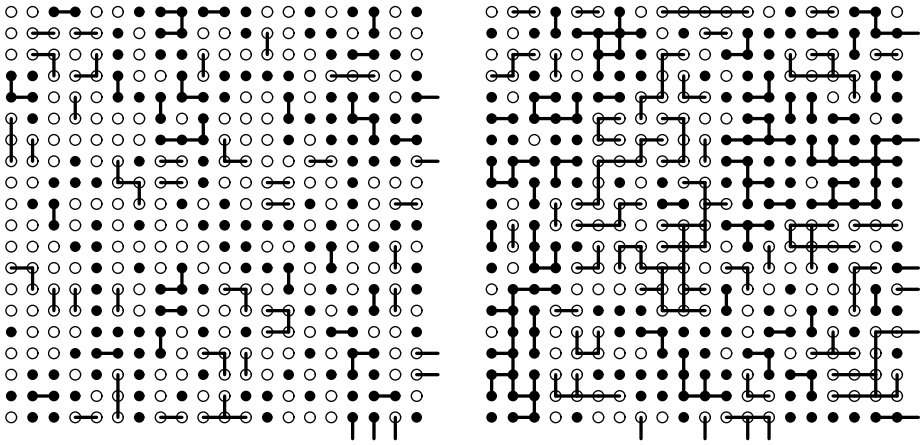


(a) $\beta = 0.1$ (b) $\beta = 0.3$

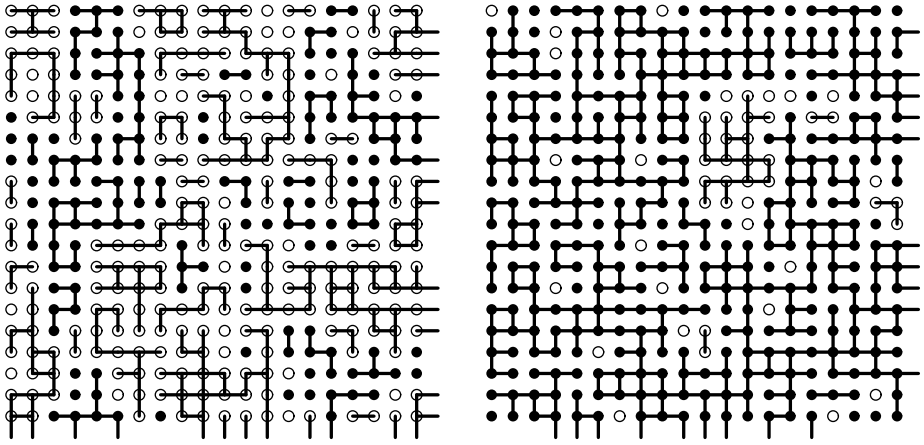
(c) $\beta = 0.4$ (d) $\beta = 0.43$

(e) $\beta = 0.44$ (f) $\beta = 0.6$

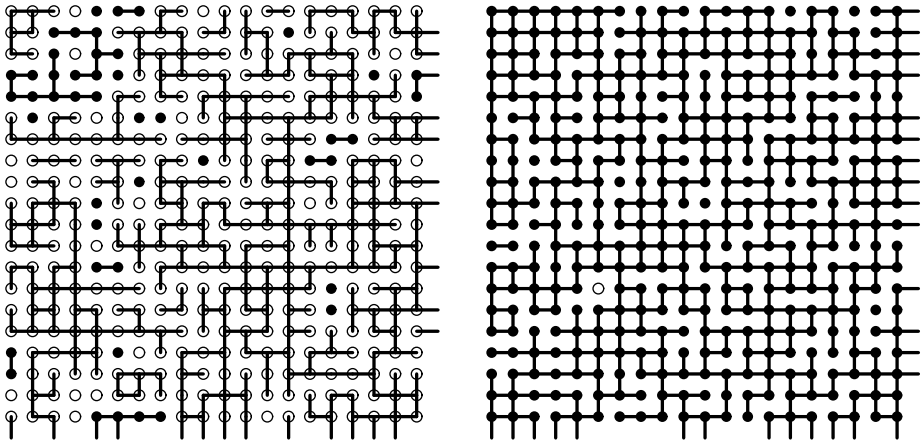
Abbildung 3.11: Konfigurationen bei verschiedenen Werten von β, System mit 100^2 Spins. Generiert mit dem Clusteralgorithmus.



(a) $\beta = 0.1$ (b) $\beta = 0.3$



(c) $\beta = 0.4$ (d) $\beta = 0.43$



(e) $\beta = 0.44$ (f) $\beta = 0.6$

Abbildung 3.12: Bondsetzung bei verschiedenen Werten von β, System mit 20^2 Spins. Generiert mit dem Clusteralgorithmus.

3.4 Autokorrelationsfunktion

Die Autokorrelationsfunktion ist definiert nach (1.39). Zwei Beispiele des zweidimensionalen Ising Modells mit dem Metropolisalgorithmus sind in Abbildung 3.13 gegeben. Nach (1.39)

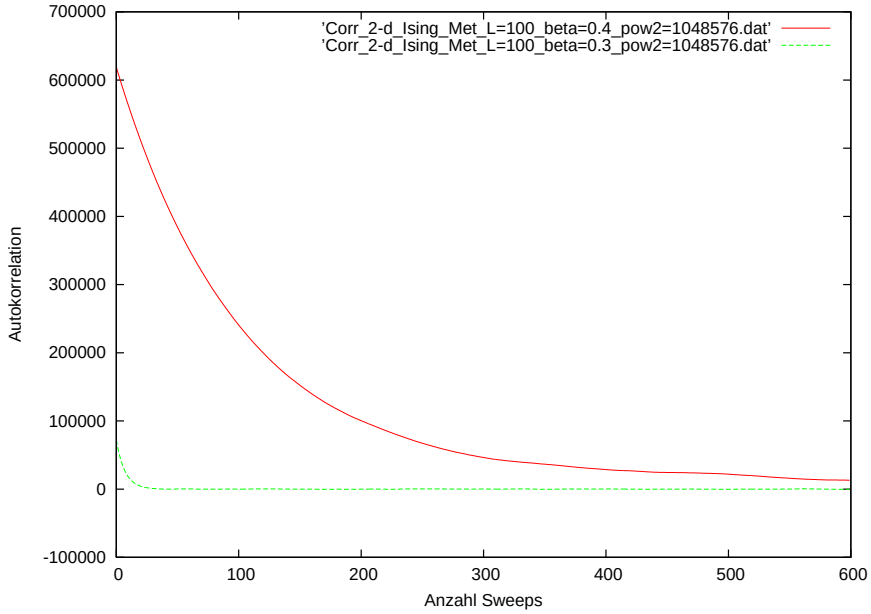


Abbildung 3.13: Autokorrelationsfunktionen beim zweidimensionalen Ising Modell mit Metropolisalgorithmus, 100^2 Spins. Grün: $\beta = 0.3$, rot: $\beta = 0.4$.

ist die Autokorrelationsfunktion proportional zu $\exp(-t/\tau)$. Damit erkennt man, dass bei $\beta = 0.4$ und somit nahe am Phasenübergang, die Autokorrelationszeit viel grösser ist als bei $\beta = 0.3$. Bei $\beta = 0.44$ fällt die Autokorrelationsfunktion noch sehr viel langsamer ab. Beim Clusteralgorithmus fällt sie hingegen immer sehr schnell ab.

Anhang A

Resultate in tabellarischer Form

Hier sind die numerischen Rohdaten, auf welchen die Graphiken basieren, tabellarisch aufgelistet. Es ist zu beachten, dass C++ standardmässig sechs signifikante Stellen ausgibt. Für eine bessere Übersichtlichkeit haben die folgenden Tabellen jedoch spaltenweise gleich viele Nachkommastellen. Nachfolgende Nullstellen sind also zum Teil gerundet. Dabei ist auch zu beachten, dass es sich bei den Fehlern lediglich um Schätzer handelt und daher nur deren Grössenordnung relevant ist und die numerischen Werte der Observablen stimmen natürlich nur innerhalb der Fehlerschranken. Das Symbol „$\triangle$“ steht im Folgenden für den absoluten Fehler.

β	χ	$\triangle\chi$	χ analyt	$\langle\mathcal{M}\rangle$	$\triangle\langle\mathcal{M}\rangle$
0.0	0.997217	0.00169011	1.00000	-0.00256729	0.00200843
0.1	1.219280	0.00135870	1.22116	0.00325966	0.00243227
0.2	1.490050	0.00152894	1.48730	0.00252151	0.00309243
0.3	1.797600	0.00174510	1.79606	0.00158310	0.00406741
0.4	2.129260	0.00196576	2.13467	0.00031853	0.00533232
0.5	2.481800	0.00210321	2.48116	0.00416756	0.00696134
0.6	2.812220	0.00218393	2.81016	-0.01031880	0.00904220
0.7	3.100470	0.00213951	3.10052	0.01628490	0.01175700
0.8	3.344400	0.00199030	3.34050	0.03201290	0.01470340
0.9	3.527210	0.00180142	3.52823	0.01480100	0.01869450
1.0	3.670400	0.00155530	3.66885	-0.00065231	0.02261690

Tabelle A.1: Eindimensionales Ising Modell mit Metropolisalgorithmus; System mit 4 Spins, 1048576 Gleichgewichtssweeps.

β	χ	$\triangle\chi$	χ analyt	$\langle\mathcal{M}\rangle$	$\triangle\langle\mathcal{M}\rangle$
0.0	1.00019	0.00138604	1.00000	-0.01180650	0.0107682
0.1	1.22271	0.00172112	1.22140	-0.00897026	0.0132504
0.2	1.49367	0.00222729	1.49182	-0.02534680	0.0166299
0.3	1.82153	0.00299424	1.82212	-0.03437610	0.0209378
0.4	2.22877	0.00417078	2.22554	-0.03159710	0.0275479
0.5	2.72398	0.00586040	2.71828	0.03917120	0.0354990
0.6	3.32185	0.00845716	3.32012	0.06525230	0.0477065
0.7	4.04634	0.01205010	4.05520	0.04253200	0.0625710
0.8	4.95908	0.01776380	4.95303	0.05093190	0.0825166
0.9	6.07815	0.02551630	6.04965	-0.10448100	0.1081820
1.0	7.42246	0.03709930	7.38906	0.18388000	0.1475270

Tabelle A.2: Eindimensionales Ising Modell mit Metropolisalgorithmus; System mit 100 Spins, 1048576 Gleichgewichtssweeps.

β	χ	$\triangle\chi$	$\langle\mathcal{M}\rangle$	$\triangle\langle\mathcal{M}\rangle$
0.0	0.99873000	0.00139370	0.201210	0.10763200
0.1	1.56570000	0.00235824	-0.117422	0.16943300
0.2	2.86024000	0.00577115	0.131525	0.33181600
0.3	7.22283000	0.02602870	-1.398280	0.96112200
0.4	61.91560000	0.83639900	9.929320	11.30710000
0.41	101.15000000	1.86220000	-9.954070	18.35560000
0.42	198.31700000	0.00000000	59.639900	36.72810000
0.43	573.59500000	0.00000000	-41.753200	87.96140000
0.44	2803.59000000	0.00000000	1666.990000	230.19300000
0.45	21.06390000	0.00000000	7494.610000	10.73710000
0.5	0.77362400	0.95954200	9112.960000	0.37302200
0.6	0.09838220	0.17142600	9736.060000	0.06226720
0.7	0.02753370	0.07581370	9901.630000	0.02707370
0.8	0.00975799	0.04244910	9960.170000	0.01506920
0.9	0.00379218	0.02564360	9983.230000	0.00908220
1.0	0.00156872	0.01617070	9992.750000	0.00572167

Tabelle A.3: Zweidimensionales Ising Modell mit Metropolisalgorithmus; System mit 100^2 Spins, 1048576 Gleichgewichtssweeps.

β	χ	$\triangle\chi$	$\langle\mathcal{M}\rangle$	$\triangle\langle\mathcal{M}\rangle$
0.000	1.00034000	0.00134511	0.00260544	0.00426698
0.025	1.11067000	0.00148881	-0.00606728	0.00463691
0.050	1.23604000	0.00165462	-0.00311470	0.00521776
0.075	1.38626000	0.00185712	-0.00150108	0.00582111
0.100	1.56818000	0.00210524	-0.00640869	0.00674532
0.125	1.78499000	0.00250056	0.02049830	0.00768664
0.150	2.05545000	0.00301787	-0.00173187	0.00892862
0.175	2.38642000	0.00360631	-0.00087357	0.01083690
0.200	2.80552000	0.00447240	-0.00200653	0.01301020
0.225	3.31753000	0.00541098	-0.01617430	0.01578780
0.250	3.97097000	0.00661691	-0.01477240	0.01996920
0.275	4.76746000	0.00801828	0.00547791	0.02495740
0.300	5.72099000	0.00934322	0.05297090	0.03253610
0.325	6.82362000	0.01062800	0.05976870	0.04121340
0.350	8.02579000	0.01149340	-0.02061270	0.05422760
0.375	9.29525000	0.01189710	-0.01147270	0.07006200
0.400	10.49110000	0.01169530	0.01623150	0.09111850
0.425	11.56530000	0.01102250	0.00613022	0.11625500
0.450	12.51840000	0.00997217	-0.05897140	0.15432900
0.475	13.28600000	0.01163840	-0.29026400	0.20928200
0.500	13.90150000	0.00000000	-0.16107600	0.25914700
0.525	14.37010000	0.00000000	0.25365600	0.32966300
0.550	14.70750000	0.00000000	-0.75870900	0.40672800
0.575	14.98150000	0.00000000	-0.87509300	0.47151200
0.600	15.20560000	0.00000000	-0.83709700	0.56204200
0.625	15.40370000	0.00000000	-0.36472100	0.61114300
0.650	15.49620000	0.00000000	-0.82107700	0.62967100
0.675	15.62080000	0.00000000	0.47625700	0.65825500
0.700	15.62060000	0.00000000	1.18810000	0.68177900
0.725	15.24140000	0.00000000	2.90042000	0.68240500
0.750	14.77970000	0.00000000	4.07475000	0.67056800
0.775	13.08330000	0.00000000	-6.65832000	0.63499700
0.800	15.69660000	0.00000000	-1.72376000	0.69861900
0.825	13.28130000	0.00000000	-6.47628000	0.64220400
0.850	15.16700000	0.00000000	-3.47726000	0.68850800
0.875	12.03110000	0.00000000	7.90552000	0.61260900
0.900	0.00381866	0.00098139	15.97280000	0.00036177
0.925	0.00303825	0.00087416	15.97830000	0.00032234
0.950	0.00246807	0.00077151	15.98230000	0.00028384
0.975	0.00200634	0.00071019	15.98530000	0.00026085
1.000	0.00160571	0.00062283	15.98810000	0.00022866

Tabelle A.4: Zweidimensionales Ising Modell mit Metropolisalgorithmus; System mit 4^2 Spins, 1048576 Gleichgewichtssweeps.

β	χ	$\triangle\chi$	$\langle\mathcal{M}\rangle$	$\triangle\langle\mathcal{M}\rangle$	$\sum_C\langle\|C\|^2\rangle$	$\triangle\sum_C\langle\|C\|^2\rangle$
0.000	1.00000	0.00000004	0.00190544	0.00138166	16.0000	0.00000067
0.025	1.10804	0.00004673	0.00102067	0.00145343	17.7286	0.00074770
0.050	1.23498	0.00008080	-0.00030804	0.00153571	19.7596	0.00129280
0.075	1.38531	0.00012151	-0.00140429	0.00162555	22.1650	0.00194420
0.100	1.56584	0.00017389	0.00153613	0.00172837	25.0535	0.00278236
0.125	1.78545	0.00025031	0.00144482	0.00184547	28.5672	0.00400501
0.150	2.05418	0.00034574	0.00195074	0.00197964	32.8670	0.00553185
0.175	2.38850	0.00048602	0.00154066	0.00213446	38.2160	0.00777646
0.200	2.80251	0.00065760	0.00514436	0.00231173	44.8402	0.01052160
0.225	3.32526	0.00088081	-0.00214624	0.00251816	53.2042	0.01409300
0.250	3.97026	0.00118407	0.00235558	0.00275142	63.5241	0.01894510
0.275	4.76821	0.00151715	-0.00359273	0.00301469	76.2914	0.02427440
0.300	5.72349	0.00187315	-0.00008965	0.00330369	91.5759	0.02997030
0.325	6.82135	0.00220327	-0.00034118	0.00360509	109.1420	0.03525230
0.350	8.02855	0.00252982	-0.00028968	0.00391264	128.4570	0.04047710
0.375	9.27392	0.00268514	0.00377083	0.00420593	148.3830	0.04296230
0.400	10.47510	0.00269365	-0.00280881	0.00446992	167.6020	0.04309840
0.425	11.57030	0.00257551	0.00471520	0.00469889	185.1250	0.04120810
0.450	12.51070	0.00236712	0.00064230	0.00488423	200.1720	0.03787380
0.475	13.28420	0.00209817	0.00259471	0.00503156	212.5480	0.03357070
0.500	13.90160	0.00181827	0.00800681	0.00515082	222.4260	0.02909240
0.525	14.38530	0.00155629	0.00466084	0.00523731	230.1640	0.02490060
0.550	14.74880	0.00132970	-0.00407052	0.00530328	235.9820	0.02127510
0.575	15.03230	0.00112458	0.00241113	0.00535523	240.5170	0.01799320
0.600	15.24610	0.00095591	-0.00413442	0.00538957	243.9380	0.01529460
0.625	15.41130	0.00081835	0.00395799	0.00542226	246.5810	0.01309370
0.650	15.53780	0.00070116	-0.00019145	0.00544356	248.6050	0.01121860
0.675	15.63540	0.00060195	0.00319195	0.00546107	250.1670	0.00963127
0.700	15.71070	0.00052370	-0.00173664	0.00547615	251.3710	0.00837924
0.725	15.76950	0.00045640	-0.00440073	0.00548159	252.3120	0.00730254
0.750	15.81550	0.00038966	-0.00135827	0.00549057	253.0480	0.00623470
0.775	15.85270	0.00034278	0.00149202	0.00549998	253.6440	0.00548450
0.800	15.88160	0.00030215	0.00002861	0.00550439	254.1060	0.00483441
0.825	15.90460	0.00026771	0.00557375	0.00550898	254.4740	0.00428346
0.850	15.92360	0.00023667	-0.00064182	0.00551138	254.7770	0.00378681
0.875	15.93790	0.00021089	0.00837898	0.00551343	255.0070	0.00337427
0.900	15.94970	0.00018795	-0.00721717	0.00551710	255.1950	0.00300727
0.925	15.95970	0.00016699	-0.00278759	0.00551753	255.3560	0.00267189
0.950	15.96780	0.00014772	0.01232360	0.00551724	255.4840	0.00236352
0.975	15.97350	0.00013410	0.00047541	0.00552046	255.5750	0.00214562
1.000	15.97830	0.00011997	0.00247550	0.00552102	255.6530	0.00191955

Tabelle A.5: Zweidimensionales Ising Modell mit Clusteralgorithmus; System mit 4^2 Spins, 1048576 Gleichgewichtssweeps.

β	χ	$\triangle\chi$	$\langle\mathcal{M}\rangle$	$\triangle\langle\mathcal{M}\rangle$	$\sum_C\langle\|C\|^2\rangle$	$\triangle\sum_C\langle\|C\|^2\rangle$
0.0	1.00000	0.00000000	-0.136856	0.097647	10000.0	0.000047
0.1	1.56711	0.00002000	0.168736	0.122283	15671.1	0.200031
0.2	2.85949	0.00008529	0.160320	0.165263	28594.9	0.852947
0.3	7.24193	0.00054075	0.287273	0.262974	72419.3	5.407540
0.4	62.66810	0.02405720	-1.172610	0.772933	626681.0	240.572000
0.41	102.54000	0.05621890	1.155140	0.989051	1025400.0	562.189000
0.42	204.28900	0.17755900	-0.803434	1.395980	2042890.0	1775.590000
0.43	626.06400	0.97989400	-1.024980	2.443160	6260640.0	9798.940000
0.44	3195.85000	3.74373000	-0.107674	5.519810	31958500.0	37437.300000
0.45	5639.43000	1.39562000	6.651560	7.334730	56394300.0	13956.200000
0.5	8305.38000	0.25144600	-1.786510	8.902460	83053800.0	2514.460000
0.6	9479.29000	0.08074580	1.939470	9.500830	94792900.0	807.458000
0.7	9804.18000	0.04009210	8.275870	9.673330	98041800.0	400.921000
0.8	9920.54000	0.02327780	24.382300	9.724160	99205400.0	232.778000
0.9	9966.46000	0.01425190	-9.890900	9.744380	99664600.0	142.519000
1.0	9985.53000	0.00913566	-6.265080	9.759770	99855300.0	91.356600

Tabelle A.6: Zweidimensionales Ising Modell mit Clusteralgorithmus; System mit 100^2 Spins, 1048576 Gleichgewichtssweeps.

Anhang B

Programmcodes

B.1 Eindimensionales Ising Modell mit Metropolisalgorithmus

```
/****************************************
 *  1-d Ising Model                     *
 *  with Metropolis algorithm           *
 *  Bachelor Thesis                     *
 *                                      *
 *  Marcel Wirz, summer semester 2007   *
 *                                      *
 ****************************************/

/****************************************
 *  used libraries                      *
 ****************************************/

#include <cstdlib>                                          //standard library
#include "random.h"                                 //used for the random generator
#include <cmath>                                   //used for mathematic operations
#include<fstream>                                    //used to write data in a file
#include <ctime>          //used to initialize random generator with CPU time
#include <iostream>                                         //used to print out data
using std::cout;                                 //used with "#include <iostream>"

using namespace std;
```

```
/*************************************************************************
 *  Because of bining, the program uses only the last 2^n<=N-M           *
 *  measurements. So N-M should be equal or a little bigger than a       *
 *  powers of two:                                                       *
 *                                                                       *
 *  2^1       2^2       2^3       2^4       2^5       2^6       2^7      *
 *  2         4         8         16        32        64        128      *
 *                                                                       *
 *  2^8       2^9       2^10      2^11      2^12      2^13      2^14     *
 *  256       512       1024      2048      4096      8192      16384    *
 *                                                                       *
 *  2^15      2^16      2^17      2^18      2^19      2^20      2^21     *
 *  32768     65536     131072    262144    524288    1048576   2097152  *
 *                                                                       *
 *  2^22      2^23      2^24      2^25      2^26      2^27      2^28     *
 *  4194304   8388608   16777216  33554432  67108864  134217728 268435456*
 *                                                                       *
 *************************************************************************/

/*****************************************
 *  definition of global variables       *
 *****************************************/

const int L = 100;                    //L = number of sites; limited to 2^31-1
const int N = 1070000;        //number of metropolis sweeps; limited to 2^31-1
const int M = 0;//numb of ignor sweeps (system at equil.); sweep 0 counted too
const double beta_start = 0; //1/(Boltzman konstant*temperature) (start value)
const double delta_beta = 0.1;                                  //beta steps
const int nr = 11;         //number of calculations with different beta values
const double B = 0;                                      //magnetic flux density
const double J = 1;                          //ferromacnetic coupling constant
const double mu = 1;                                          //permeability
double beta;                                                  //variing beta
short s[L];                                                  //array of spins
int l[L],r[L];                                              //spin neighbours
double Mag[N];                                //measurements of magnetization
double MM[N];//array for magnetization/squared magnetization (used for bining)
double MM2[N];//array with squared magnetizations (used for error calculation)
double Sus[N];                                //measurements of susceptibility
int mag;                                                     //magnetization
double a_sus;                               //analytic result for susceptibility
int count1;    //counting variable in the for loop of the Metropolis algorithm
//used until count4 (main and error)
double sus;                                                  //susceptibility
int pow2;                        //number of used measurements (power of two)
double mm;                                                  //mean magnetization
double mm2;                                    //mean squared magnetization
```

```
double mm4;                                          //mean (magnetization to four)
double err_mm,err_mm2;                                  //error of mm and mm2
double err; //error calculated in error subrutine (becomes error of mm or mm2)
bool err_squared;//true if error of mean squared mag is calculated, flase else
bool err_ofl;                                                  //error overflow
//true=error is still growing after as many as possible binings are done

//used in Metropolis method
double dE;          //energy difference (E_new - E_old), generated by one flip
int pos;               //position of eventually fliped spin (randomly chosen)
double tprob;                                          //transition probability
double rprob;                   //random probability for compatison with tprob

/*****************************************
 *  definition of methods                *
 *****************************************/

void start();
void metropolis();
void analytic();
void error();

/*****************************************
 *  main method                          *
 *****************************************/

int main(int argc, char *argv[]){
    char file1[100];                     //array for magnetization datafilename
    char file2[100];                    //array for susceptibility datafilename
    double err_sus;                                  //error of susceptibility
    time_t t;                          //time variable declaration (like int t)
    time(&t);                                                  //set time to t
    //t is used to initialize the random generator always different
    InitRandomNumberGenerator(t);          //initialize random number geerator
    int count3=-1;//find gratest power of two which is smaller or equal to N-M
    do {
        count3++;
    } while (pow(2.,count3+1)<=N-M);
    pow2 = int(pow(2.,count3));
    //create filename
    sprintf(file2,"Sus_1-d_Ising_Met_L=%d_pow2=%d_Init=%d.dat",L,pow2,t);
    ofstream out_sus(file2,ios::out);         //create susceptibility datafile
    for (int count2=0; count2<nr; count2++){            //run program nr times
        beta = beta_start+count2*delta_beta;                       //set beta
        start();
        for (count1=0; count1<N; count1++){
             metropolis();
        }
```

```
//create filename
sprintf(file1,"Mag_1-d_Ising_Met_L=%d_beta=%g_N=%d.dat",L,beta,N);
ofstream out_mag(file1,ios::out);      //create magnetization datafile
for (int i=0; i<N; i++){            //write magnetizations in datafile
    out_mag<<Mag[i]<<"\n";
}
out_mag.close();                          //close magnetization datafile
for (int i=0; i<pow2; i++){
    Mag[i] = Mag[N-pow2+i];//overwrite the first N-pow2 magnetizations
    MM[i] = Mag[i];                              //create MM[] and MM2[]
    MM2[i] =pow(Mag[i],2);
}
err_squared = false;
err_ofl = false;
error();               //calculate the error of the mean magnetization
err_mm = err;
for (int i=0; i<pow2; i++){//recreate MM[] with squared magnetizations
    MM[i] = MM2[i];
}
err_squared = true;
//use the same subrutine to calculate the error of the mean squared
//magnetization but now with squared magnetizations
error();
err_mm2 = err;
mm = 0;
mm2 = 0;
for (int i=0; i<pow2; i++){             //calculate mean magnetization
    mm = mm+Mag[i];                    //and mean squared magnetization
    mm2 = mm2+pow(Mag[i],2);
}
mm = mm/pow2;
mm2 = mm2/pow2;
sus = 1./L*(mm2-pow(mm,2));                 //calculate susceptibility
err_sus =1./L*sqrt(pow(err_mm2,2)+pow(2*mm*err_mm,2));
if (err_ofl==true){       //if one of the errors didn't reach a plateau
    err_sus = 0;                  //set the whole error to zero because
}                                          //there's no reasonable error
analytic();
//write results in datafile
out_sus<<beta<<"\t"<<sus<<"\t"<<err_sus<<"\t"<<a_sus<<"\t"
<<mm<<"\t"<<err_mm<<"\t"<<mm2<<"\t"<<err_mm2<<"\n";
cout<<"Beta:  "<<beta<<"\n"<<"\n";                //print out data
cout<<"Analytic result for susceptibility (B=0):"<<"\n";
cout<<a_sus<<"\n";
cout<<"Susceptibility:"<<"\n";
cout<<sus<<"+-"<<err_sus<<"\n";
cout<<"lower value:  "<<sus-err_sus<<"\n";
```

```
        cout<<"upper value:  "<<sus+err_sus<<"\n"<<"\n"<<"\n";
    }
    out_sus.close();                          //close susceptibility datafile
    system("PAUSE");              //console application shoud stay at the end
    return 0;                                                 //no error occured
}

/*****************************************
 *  start method                         *
 *  Generates an initial spin            *
 *  configuration with all spins up.     *
 *  Defines next neighbours and          *
 *  implements the periodic boundary     *
 *  condition.                           *
 *****************************************/

void start(){
    for (int i=0; i<L; i++){          //generate an initial spin configuration
        s[i] = 1;
    }
    for (int i=1; i<L-1; i++){                        //define index for each
        r[i] = i+1;                                       //l and r neighbour
        l[i] = i-1;
    }
    r[0]   = 1;   //the same for the first and last site (boundary conditions)
    l[0]   = L-1;                                    //Arrays go from 0 to L-1!
    r[L-1] = 0;
    l[L-1] = L-2;
    mag = L;                                  //magnetization at the beginning
}

/*****************************************
 *  Metropolis method                    *
 *  Chooses L times a lattice site       *
 *  randomly and makes a Metropolis      *
 *  step.                                *
 *****************************************/

void metropolis(){
    for (int j=0; j<L; j++){
    //generate rand val in int. [0;L[; i int => cut off digits after dez point
        pos = int(FloatRandomValue()*(L));
        dE = 2*(J*s[pos]*(s[l[pos]]+s[r[pos]])+mu*B*s[pos]);
        if (dE <= 0){                                          //then accept flip
            s[pos] = -s[pos];
            mag = mag+2*s[pos];
        }else{
```

```
            tprob = exp(-beta*dE);              //define transition probability
            rprob = FloatRandomValue();          //generate random probability
            if (rprob <= tprob){     //then flip the spin, otherwise let it be
                s[pos] = -s[pos];
                mag = mag+2*s[pos];
            }
        }
    }
    Mag[count1] = mag;         //write magnetization afer a sweep in Mag array
}


/****************************************
 *  error method                        *
 *  Calculates the error                *
 *  with bining method.                 *
 ****************************************/

void error(){
    char file3[100];                            //array for error datafilename
    double old_err;          //old error of mean magnetization for comparison
    int count4 = 1;                                          //number of bins
    mm = 0;
    mm2 = 0;
    for (int i=0; i<pow2; i++){                 //calculate mean magnetization
        mm = mm+MM[i];                         //and mean squared magnetization
        mm2 = mm2+pow(MM[i],2);
    }
    mm = mm/pow2;
    mm2 = mm2/pow2;
    err = sqrt((mm2-pow(mm,2))/(pow2-1));  //calculate first estimate of error
    if (err_squared == false){
    //create filename depending on wihch error is calculated
    sprintf(file3,"Err_mm_1-d_Ising_Met_L=%d_beta=%g_pow2=%d.dat",
    L,beta,pow2);
    }
    else{
    sprintf(file3,"Err_mm2_1-d_Ising_Met_L=%d_beta=%g_pow2=%d.dat",
    L,beta,pow2);
    }
    ofstream out_err(file3,ios::out);                 //create error datafile
    out_err<<err<<"\n";                      //write error in error datafile
    do {
        count4 = 2*count4;
        old_err = err;                      //save new to old for comparison
        mm = 0;                                          //set again to zero
        mm2 = 0;
```

```
        for (int i=0; i<pow2/count4; i++){
            MM[i] = (MM[2*i]+MM[2*i+1])/2;                    //bin two values
            mm = mm+MM[i];                          //calculate mean magnetization
            mm2 = mm2+pow(MM[i],2);               //and mean squared magnetization
        }
        mm = mm/(pow2/count4);
        mm2 = mm2/(pow2/count4);
        err = sqrt((mm2-pow(mm,2))/(pow2/count4-1));//calc new estimate of err
        out_err<<err<<"\n";                      //write error in error datafile
//if error got much smaller (happens sometimes, if there's not much data left)
        if (0.95*old_err>err){
            cout<<"Error got smaller."<<"\n";
            err_ofl = true;
        }
    //do bining until the error has reached a plateau or until there's not
    //enough data to bin again and calculate a reasonable error
    } while ((old_err<0.95*err)&&(pow2/count4>512));
    //if there are only 512 datapoints left and error is still growing
    if (old_err<0.95*err){
        cout<<"Not enough data to calculate error or error is too big."
        <<"\n";
        err_ofl = true;
    }
    out_err.close();                                    //close error datafile
}

/*****************************************
 *  analytic method                      *
 *  Calculates the exact result of the   *
 *  susceptibility for B=0.              *
 *****************************************/

void analytic(){
    a_sus = (1-pow(tanh(beta*J),L))*exp(2*beta*J)/(1+pow(tanh(beta*J),L));
}
```

B.2 Zweidimensionales Ising Modell mit Metropolisalgorithmus

```
/***************************************
 *  2-d Ising Model                    *
 *  with Metropolis algorithm          *
 *  Bachelor Thesis                    *
 *                                     *
 *  Marcel Wirz, summer semester 2007  *
 *                                     *
 ***************************************/

/***************************************
 *  used libraries                     *
 ***************************************/

#include <cstdlib>                                    //standard library
#include "random.h"                          //used for the random generator
#include <cmath>                            //used for mathematic operations
#include<fstream>                            //used to write data in a file
#include <ctime>          //used to initialize random generator with CPU time
#include <iostream>                                   //used to print out data
using std::cout;                            //used with "#include <iostream>"

using namespace std;

/*************************************************************************
 *  Because of bining, the program uses only the last 2^n<=N-M           *
 *  measurements. So N-M should be equal or a little bigger than a       *
 *  powers of two:                                                       *
 *                                                                       *
 *  2^1       2^2       2^3       2^4       2^5       2^6       2^7      *
 *  2         4         8         16        32        64        128      *
 *                                                                       *
 *  2^8       2^9       2^10      2^11      2^12      2^13      2^14     *
 *  256       512       1024      2048      4096      8192      16384    *
 *                                                                       *
 *  2^15      2^16      2^17      2^18      2^19      2^20      2^21     *
 *  32768     65536     131072    262144    524288    1048576   2097152  *
 *                                                                       *
 *  2^22      2^23      2^24      2^25      2^26      2^27      2^28     *
 *  4194304   8388608   16777216  33554432  67108864  134217728 268435456 *
 *                                                                       *
 *************************************************************************/
```

```
/*****************************************
 *  definition of global variables       *
 *****************************************/

const int L = 100;                    //L = number of sites; limited to 2^31-1
const int L2 = 10000;                                              //L squared
const int N = 1070000;        //number of metropolis sweeps; limited to 2^31-1
const int M = 0;//numb of ignor sweeps (system at equil.); sweep 0 counted too
const double beta_start = 0; //1/(Boltzman konstant*temperature) (start value)
const double delta_beta = 0.025;                                   //beta steps
const int nr = 41;         //number of calculations with different beta values
const double B = 0;                                      //magnetic flux density
const double J = 1;                           //ferromacnetic coupling constant
const double mu = 1;                                               //permeability
const int d = 2;                                                     //dimension
double beta;                                                       //variing beta
short s[L2];                                                     //array of spins
int b0[L2],f0[L2];       //spin neighbours back and forward of first dimension
int b1[L2],f1[L2];      //spin neighbours back and forward of second dimension
double Mag[N];                                  //measurements of magnetization
double MM[N];//array for magnetization/squared magnetization (used for bining)
double MM2[N];//array with squared magnetizations (used for error calculation)
double Sus[N];                                 //measurements of susceptibility
int mag;                                                        //magnetization
int count1 = 0;//counting variable in the for loop of the Metropolis algorithm
//used until count4 (main and error)
double sus;                                                      //susceptibility
int pow2;                          //number of used measurements (power of two)
double mm;                                                     //mean magnetization
double mm2;                                          //mean squared magnetization
double mm4;                                        //mean (magnetization to four)
double err_mm,err_mm2;                                    //error of mm and mm2
double err; //error calculated in error subrutine (becomes error of mm or mm2)
bool err_squared;//true if error of mean squared mag is calculated, flase else
bool err_ofl;                                                  //error overflow
//true=error is still growing after as many as possible binings are done

//used in Metropolis method
double dE;          //energy difference (E_new - E_old), generated by one flip
int pos;               //position of eventually fliped spin (randomly chosen)
double tprob;                                          //transition probability
double rprob;                  //random probability for compatison with tprob
```

```
/****************************************
 *  definition of methods               *
 ****************************************/

void start();
void metropolis();
void error();
void configuration();

/****************************************
 *  main method                         *
 ****************************************/

int main(int argc, char *argv[]){
    char file1[100];                    //array for magnetization datafilename
    char file2[100];                   //array for susceptibility datafilename
    double err_sus;                                  //error of susceptibility
    time_t t;                          //time variable declaration (like int t)
    time(&t);                                                  //set time to t
    //t is used to initialize the random generator always different
    InitRandomNumberGenerator(t);          //initialize random number geerator
    int count3 = -1;//find gratest pow of two which is smaller or equal to N-M
    do {
        count3++;
    } while (pow(2.,count3+1)<=N-M);
    pow2 = int(pow(2.,count3));
    //create filename
    sprintf(file2,"Sus_2-d_Ising_Met_L=%d_pow2=%d_Init=%d.dat",L,pow2,t);
    ofstream out_sus(file2,ios::out);         //create susceptibility datafile
    for (int count2=0; count2<nr; count2++){            //run program nr times
        beta = beta_start+count2*delta_beta;                        //set beta
        start();
        for (count1=0; count1<N; count1++){
            metropolis();
        }
        //create filename
        sprintf(file1,"Mag_2-d_Ising_Met_L=%d_beta=%g_N=%d.dat",L,beta,N);
        ofstream out_mag(file1,ios::out);      //create magnetization datafile
        for (int i=0; i<N; i++){            //write magnetizations in datafile
            out_mag<<Mag[i]<<"\n";
        }
        out_mag.close();                         //close magnetization datafile
        for (int i=0; i<pow2; i++){
            Mag[i] = Mag[N-pow2+i];//overwrite the first N-pow2 magnetizations
            MM[i] = Mag[i];                              //create MM[] and MM2[]
            MM2[i] =pow(Mag[i],2);
        }
```

```
        err_squared = false;
        err_ofl = false;
        error();                //calculate the error of the mean magnetization
        err_mm = err;
        for (int i=0; i<pow2; i++){//recreate MM[] with squared magnetizations
            MM[i] = MM2[i];
        }
        err_squared = true;
        //use the same subrutine to calculate the error of the mean squared
        //magnetization but now with squared magnetizations
        error();
        err_mm2 = err;
        mm = 0;
        mm2 = 0;
        for (int i=0; i<pow2; i++){             //calculate mean magnetization
            mm = mm+Mag[i];                    //and mean squared magnetization
            mm2 = mm2+pow(Mag[i],2);
        }
        mm = mm/pow2;
        mm2 = mm2/pow2;
        sus = 1/pow(float(L),d)*(mm2-pow(mm,2));    //calculate susceptibility
        err_sus = 1/pow(float(L),d)*sqrt(pow(err_mm2,2)
        +pow(2*mm*err_mm,2));
        if (err_ofl==true){       //if one of the errors didn't reach a plateau
            err_sus = 0;                  //set the whole error to zero because
        }                                          //there's no reasonable error
        out_sus<<beta<<"\t"<<sus<<"\t"<<err_sus<<"\t"//write res in file
        <<mm<<"\t"<<err_mm<<"\t"<<mm2<<"\t"<<err_mm2<<"\n";
        cout<<"Beta: "<<beta<<"\n"<<"\n";                //print out data
        cout<<"Susceptibility:"<<"\n";
        cout<<sus<<"+-"<<err_sus<<"\n";
        cout<<"lower value:  "<<sus-err_sus<<"\n";
        cout<<"upper value:  "<<sus+err_sus<<"\n"<<"\n"<<"\n";
        count1 = N-1;  //in the for loop it counts till N but does it till N-1
        configuration();
    }
    out_sus.close();                           //close susceptibility datafile
    system("PAUSE");              //console application shoud stay at the end
    return 0;                                              //no error occured
}
```

```
/****************************************
 *  start method                        *
 *  Generates an initial spin           *
 *  configuration with all spins up.    *
 *  Defines next neighbours and         *
 *  implements the periodic boundary    *
 *  condition.                          *
 ****************************************/

void start(){
    for (int i=0; i<L2; i++){
        s[i] = 1;                      //generate an initial spin configuration
        f0[i] = (i+1)%L+(i/L)*L;             //define index for each neighbour
        b0[i] = (i-1)%L+(i/L)*L;            //% stands for the modulo function
        f1[i] = int((i-L+pow(float(L),2)))%int(pow(float(L),2));
        b1[i] = int((i+L+pow(float(L),2)))%int(pow(float(L),2));
    }
    b0[0] = L-1;                                                 //correct b0[0]
    mag = L2;                                  //magnetization at the beginning
}

/****************************************
 *  Metropolis method                   *
 *  Chooses L^2 times a lattice site    *
 *  randomly and makes a Metropolis     *
 *  step.                               *
 ****************************************/

void metropolis(){
    for (int j=0; j<L2; j++){
//generate random val in int. [0;L^2[; i int => cut off digits after dez point
        pos = int(FloatRandomValue()*(L2));
        dE = 2*(J*s[pos]*(s[f0[pos]]+s[b0[pos]]+s[f1[pos]]+s[b1[pos]])+
        mu*B*s[pos]);
        if (dE <= 0){                                          //then accept flip
            s[pos] = -s[pos];
            mag = mag+2*s[pos];
        }else{
            tprob = exp(-beta*dE);              //define transition probability
            rprob = FloatRandomValue();          //generate random probability
            if (rprob <= tprob){     //then flip the spin, otherwise let it be
                s[pos] = -s[pos];
                mag = mag+2*s[pos];
            }
        }
    }
    Mag[count1] = mag;         //write magnetization afer a sweep in Mag array
}
```

```
/*****************************************
 *  error method                         *
 *  Calculates the error                 *
 *  with bining method.                  *
 *****************************************/

void error(){
    char file3[100];                              //array for error datafilename
    double old_err;          //old error of mean magnetization for comparison
    int count4 = 1;                                          //number of bins
    mm = 0;
    mm2 = 0;
    for (int i=0; i<pow2; i++){                 //calculate mean magnetization
        mm = mm+MM[i];                         //and mean squared magnetization
        mm2 = mm2+pow(MM[i],2);
    }
    mm = mm/pow2;
    mm2 = mm2/pow2;
    err = sqrt((mm2-pow(mm,2))/(pow2-1));  //calculate first estimate of error
    if (err_squared == false){
        //create filename depending on wihch error is calculated
        sprintf(file3,"Err_mm_2-d_Ising_Met_L=%d_beta=%g_pow2=%d.dat",
        L,beta,pow2);
    }
    else{
        sprintf(file3,"Err_mm2_2-d_Ising_Met_L=%d_beta=%g_pow2=%d.dat",
        L,beta,pow2);
    }
    ofstream out_err(file3,ios::out);                  //create error datafile
    out_err<<err<<"\n";                       //write error in error datafile
    do {
        count4 = 2*count4;
        old_err = err;                        //save new to old for comparison
        mm = 0;                                          //set again to zero
        mm2 = 0;
        for (int i=0; i<pow2/count4; i++){
            MM[i] = (MM[2*i]+MM[2*i+1])/2;                    //bin two values
            mm = mm+MM[i];                      //calculate mean magnetization
            mm2 = mm2+pow(MM[i],2);           //and mean squared magnetization
        }
        mm = mm/(pow2/count4);
        mm2 = mm2/(pow2/count4);
        err = sqrt((mm2-pow(mm,2))/(pow2/count4-1));//calc new estimate of err
        out_err<<err<<"\n";                  //write error in error datafile
//if error got much smaller (happens sometimes, if there's not much data left)
        if (0.95*old_err>err){
            cout<<"Error got smaller."<<"\n";
```

```
            err_ofl = true;
        }
    //do bining until the error has reached a plateau or until there's not
    //enough data to bin again and calculate a reasonable error
    } while ((old_err<0.95*err)&&(pow2/count4>512));
    //if there are only 512 datapoints left and error is still growing
    if (old_err<0.95*err){
        cout<<"Not enough data to calculate error or error is too big."
        <<"\n";
        err_ofl = true;
    }
    out_err.close();                                    //close error datafile
}

/*****************************************
 *  configuration method                 *
 *  Writes the spin confguration         *
 *  in a file.        .                  *
 *****************************************/

void configuration(){
    char file4[100];                    //array for configuration datafilename
    //create filename
    sprintf(file4,"Conf_2-d_Ising_Met_L=%d_beta=%g_pow2=%d_sweep=%d.dat"
    ,L,beta,pow2,count1);
    ofstream out_con(file4,ios::out);          //create configuration datafile
    for (int i=0; i<L2; i++){                //write spin configuration in file
        out_con<<s[i]<<"\t";
        if ((i+1)%L==0){
            out_con<<"\n";
        }
    }
    out_con.close();                            //close configuration datafile
}
```

B.3 Zweidimensionales Ising Modell mit Clusteralgorithmus

```
/***************************************
 *  2-d Ising Model                    *
 *  with Swendsen-Wang Cluster Algorithm *
 *  Bachelor Thesis                    *
 *                                     *
 *  Marcel Wirz, summer semester 2007  *
 *                                     *
 ***************************************/

/***************************************
 *  used libraries                     *
 ***************************************/

#include <cstdlib>                                    //standard library
#include "random.h"                          //used for the random generator
#include <cmath>                            //used for mathematic operations
#include<fstream>                            //used to write data in a file
#include <ctime>          //used to initialize random generator with CPU time
#include <iostream>                                  //used to print out data
using std::cout;                            //used with "#include <iostream>"

using namespace std;

/*************************************************************************
 *  Because of bining, the program uses only the last 2^n<=N-M           *
 *  measurements. So N-M should be equal or a little bigger than a       *
 *  powers of two:                                                       *
 *                                                                       *
 *  2^1       2^2       2^3       2^4       2^5       2^6       2^7      *
 *  2         4         8         16        32        64        128      *
 *                                                                       *
 *  2^8       2^9       2^10      2^11      2^12      2^13      2^14     *
 *  256       512       1024      2048      4096      8192      16384    *
 *                                                                       *
 *  2^15      2^16      2^17      2^18      2^19      2^20      2^21     *
 *  32768     65536     131072    262144    524288    1048576   2097152  *
 *                                                                       *
 *  2^22      2^23      2^24      2^25      2^26      2^27      2^28     *
 *  4194304   8388608   16777216  33554432  67108864  134217728 268435456 *
 *                                                                       *
 *************************************************************************/
```

```
/*****************************************
 *  definition of global variables       *
 *****************************************/

const int L = 100;                    //L = number of sites; limited to 2^31-1
const int L2 = 10000;                                              //L squared
const int N = 1070000;           //number of cluster sweeps; limited to 2^31-1
const int M = 0;//numb of ignor sweeps (system at equil.); sweep 0 counted too
const double beta_start = 0; //1/(Boltzman konstant*temperature) (start value)
const double delta_beta = 0.1;                                     //beta steps
const int nr = 11;         //number of calculations with different beta values
const double J = 1;                          //ferromacnetic coupling constant
const double mu = 1;                                              //permeability
const int d = 2;                                                    //dimension
double beta;                                                      //variing beta
short s[L2];                                                     //array of spins
int b0[L2],f0[L2];       //spin neighbours back and forward of first dimension
int b1[L2],f1[L2];      //spin neighbours back and forward of second dimension
double Mag[N];                                  //measurements of magnetization
double MOBS1[N];                       //array for observable (used for bining)
double MOBS2[N]; //array with squared observables (used for error calculation)
double Sus[N];                                  //measurements of susceptibility
int mag;                                                          //magnetization
int count1 = 0;   //counting variable in the for loop of the cluster algorithm
//used until count7 (main, error and cluster)
double sus;                                                      //susceptibility
int pow2;                         //number of used measurements (power of two)
double mobs1;                               //mean observable1 (mean mag or mscss)
double mobs2;                     //mean observable2 (mean squared mag or mscss)
double err_mobs1,err_mscss;                //error of mean observable and mscss
double err; //error calculated in err subrutine (becomes err of mobs or mscss)
bool bool_err_mscss; //true=err of scss is calc, flase=err of mean mag is calc
bool err_ofl;                                                   //error overflow
//true=error is still growing after as many as possible binings are done
double mscss;                                  //mean summed cluster size squared

//used in cluster method
double prob;                //probability to set a bond between parallel spins
//(prob=1-exp(-2*Beta*J) used in cluster algorithm)
short b[L2][d];
//bonds b[lattice site][to right or lower neighbour] = 1 or 0
//d=0 means to right neighbour d=1 to the lower (lower because of plotting)
double rprob;                    //random probability used in cluster algorithm
int list[L2];                             //list of spins in the current cluster
int mark[L2];  //every spin gets a mark depending on the cluster he belongs to
//-1 = not yet in a cluster
int count5;           //counting variable (write position in list) starts at 0
```

```
//count5 = number of spins in current cluster
int count6;           //counting variable (read position in list) starts at -1
int count7;                  //counting variable (cluster number) starts at -1
unsigned long long CSS[L2];                  //array for cluster sizes squared
unsigned long long SCSS[N];          //array for sum of cluster sizes squared
unsigned long long scss;                        //summed cluster sizes squared

/****************************************
 *  definition of methods               *
 ****************************************/

void start();
void cluster();
void error();
void configuration();
void bonds();

/****************************************
 *  main method                         *
 ****************************************/

int main(int argc, char *argv[]){
    char file1[100];                    //array for mag and scss datafilename
    char file2[100];                   //array for susceptibility datafilename
    double err_sus;                                 //error of susceptibility
    time_t t;                          //time variable declaration (like int t)
    time(&t);                                                 //set time to t
    //t is used to initialize the random generator always different
    InitRandomNumberGenerator(t);          //initialize random number geerator
    int count3 = -1;//find gratest pow of two which is smaller or equal to N-M
    do {
        count3++;
    } while (pow(2.,count3+1)<=N-M);
    pow2 = int(pow(2.,count3));
    //create filename
    sprintf(file2,"Sus_2-d_Ising_Cluster_L=%d_pow2=%d_Init=%d.dat",L,pow2,t);
    ofstream out_sus(file2,ios::out);         //create susceptibility datafile
    for (int count2 = 0; count2<nr; count2++){          //run program nr times
        beta = beta_start+count2*delta_beta;                        //set beta
        prob = 1-exp(-2*beta*J);  //calc probability used in cluster algorithm
        start();
        for (count1 = 0; count1<N; count1++){
            cluster();
        }
        //create filename
        sprintf(file1,"Mag_2-d_Ising_Cluster_L=%d_beta=%g_N=%d.dat",L,beta,N);
        ofstream out_mag(file1,ios::out);                    //create datafile
```

```
for (int i=0; i<N; i++){                                //write in datafile
    out_mag<<Mag[i]<<"\n";
}
out_mag.close();                                          //close datafile
sprintf(file1,"SCSS_2-d_Ising_Cluster_L=%d_beta=%g_N=%d.dat",
L,beta,N);
ofstream out_scss(file1,ios::out);                      //create datafile
for (int i=0; i<N; i++){                                //write in datafile
    out_scss<<SCSS[i]<<"\n";
}
out_scss.close();                                         //close datafile
for (int i=0; i<pow2; i++){
    Mag[i] = Mag[N-pow2+i];//overwrite the first N-pow2 magnetizations
    SCSS[i] = SCSS[N-pow2+i];        //overwrite the first N-pow2 scss
    MOBS1[i] = Mag[i];                        //create MOBS[] and MOBS2[]
    MOBS2[i] =pow(Mag[i],2);
}
bool_err_mscss = false;
err_ofl = false;
error();               //calculate the error of the mean magnetization
err_mobs1 = err;
for (int i=0; i<pow2; i++){                  //recreate MOBS[] with scss
    MOBS1[i] = SCSS[i];
}
bool_err_mscss = true;   //use same subrutine to calc the err of mscss
error();
err_mscss = err;
mobs1 = 0;
for (int i=0; i<pow2; i++){              //calculate mean magnetization
    mobs1 = mobs1+Mag[i];
}
mobs1 = mobs1/pow2;
mscss=0;
for (int i=0; i<pow2; i++){                        //calculate mean scss
    mscss = mscss+SCSS[i];
}
mscss = mscss/pow2;
sus = 1/pow(float(L),d)*mscss;                //calculate susceptibility
err_sus = 1/pow(float(L),d)*err_mscss;
if (err_ofl==true){       //if one of the errors didn't reach a plateau
    err_sus = 0;                    //set the whole error to zero because
}                                            //there's no reasonable error
//write results in datafile
out_sus<<beta<<"\t"<<sus<<"\t"<<err_sus<<"\t"
<<mobs1<<"\t"<<err_mobs1<<"\t"<<mscss<<"\t"<<err_mscss<<"\n";
cout<<"Beta: "<<beta<<"\n"<<"\n";                  //print out data
cout<<"Susceptibility:"<<"\n";
```

```
        cout<<sus<<"+-"<<err_sus<<"\n";
        cout<<"lower value:  "<<sus-err_sus<<"\n";
        cout<<"upper value:  "<<sus+err_sus<<"\n"<<"\n"<<"\n";
        count1 = N-1;  //in the for loop it counts till N but does it till N-1
        configuration();
        bonds();
    }
    out_sus.close();                          //close susceptibility datafile
    system("PAUSE");              //console application shoud stay at the end
    return 0;                                                 //no error occured
}

/*****************************************
 *  start method                         *
 *  Generates an initial spin            *
 *  configuration with all spins up.     *
 *  Defines next neighbours and          *
 *  implements the periodic boundary     *
 *  condition.                           *
 *****************************************/

void start(){
    for (int i=0; i<L2; i++){
        s[i] = 1;                      //generate an initial spin configuration
        f0[i] = (i+1)%L+(i/L)*L;             //define index for each neighbour
        b0[i] = (i-1)%L+(i/L)*L;            //% stands for the modulo function
        f1[i] = int((i-L+pow(float(L),2)))%int(pow(float(L),2));
        b1[i] = int((i+L+pow(float(L),2)))%int(pow(float(L),2));
    }
    b0[0] = L-1;                                                 //correct b0[0]
    mag = L2;                                  //magnetization at the beginning
}

/*****************************************
 *  Cluster method                       *
 *  Sets bonds and flips the created     *
 *  clusters with probatility 0.5.       *
 *****************************************/

void cluster(){
    for (int i=0; i<L2; i++){              //set bonds for every lattice point
    //right neighbours
        if (s[i]==s[f0[i]]){  //if spin[i] and his right neighbour are parallel
            rprob = FloatRandomValue();          //generate random probability
            if (rprob<=prob){                             //then put bond to 1
                b[i][0]=1;
            }else{                                                //else to zero
```

```
                b[i][0]=0;
            }
        }else{                                          //if they are antiparallel
            b[i][0]=0;                                          //put bond to zero
        }
    //lower neighbours
        if (s[i]==s[b1[i]]){  //if spin[i] and his lower neighbour are parallel
            rprob = FloatRandomValue();          //generate random probability
            if (rprob<=prob){                                   //then put bond to 1
                b[i][1]=1;
            }else{                                                      //else to zero
                b[i][1]=0;
            }
        }else{                                          //if they are antiparallel
            b[i][1]=0;                                          //put bond to zero
        }
    }
    for (int i=0; i<L2; i++){     //set all marks to -1 (not yet in a cluster)
        mark[i] = -1;
    }
    count7 = -1;                            //cluster number zero is not yet found
    for (int i=0; i<L2; i++){            //go sequentially through the lattice
        if (mark[i]==-1){ //if spin i is not yet in a cluster (new clust found)
            count7 = count7+1;                  //increase counter of clusters
            count6 = -1;                    //reading position is not yet zero
            count5 = 0;                  //put writing position in list to zero
            mark[i] = count7;                                   //set cluster number
            list[count5] = i;                                   //put spin on list
            count5 = count5+1;
            while (count6<count5-1){      //work through all spins on the list
                count6 = count6+1;
//check all neighbours of spin list[count6], if they have activated bonds and
//if they are not yet on the list
                if ((b[list[count6]][0]==1)&&(mark[f0[list[count6]]]==-1)){
                    mark[f0[list[count6]]] = count7;
                    list[count5] = f0[list[count6]];        //put spin on list
                    count5 = count5+1;     //increase writing position in list
                }
                if ((b[f1[list[count6]]][1]==1)&&(mark[f1[list[count6]]]==-1))
                    {
                    mark[f1[list[count6]]] = count7;
                    list[count5] = f1[list[count6]];
                    count5 = count5+1;
                }
                if ((b[b0[list[count6]]][0]==1)&&(mark[b0[list[count6]]]==-1))
                    {
                    mark[b0[list[count6]]] = count7;
```

```
                    list[count5] = b0[list[count6]];
                    count5 = count5+1;
                }
                if ((b[list[count6]][1]==1)&&(mark[b1[list[count6]]]==-1)){
                    mark[b1[list[count6]]] = count7;
                    list[count5] = b1[list[count6]];
                    count5 = count5+1;
                }
            }
            CSS[count7] = count5*count5;
            rprob = FloatRandomValue();          //generate random probability
            if (rprob<=0.5){ //flip all spins in current cluster with prob 0.5
                for (int i=0; i<count5; i++){
                    s[list[i]]=-s[list[i]];
                }
                mag = mag+2*count5*s[list[0]];   //calculate new magnetization
            }
        }
    }
    scss = 0;
    for (int i=0; i<count7+1; i++){             //sum up cluster sizes squared
        scss = scss+CSS[i];
    }
    SCSS[count1] = scss;         //write sum of cluster sizes squared in array
    Mag[count1] = mag;          //write magnetization afer a sweep in Mag array
}

/*****************************************
 *  error method                         *
 *  Calculates the error                 *
 *  with bining method.                  *
 *****************************************/

void error(){
    char file3[100];                            //array for error datafilename
    double old_err;                                 //old error for comparison
    int count4 = 1;                                           //number of bins
    mobs1 = 0;
    mobs2 = 0;
    for (int i=0; i<pow2; i++){                   //calculate mean observable
        mobs1 = mobs1+MOBS1[i];                  //and mean squared observable
        mobs2 = mobs2+pow(MOBS1[i],2);
    }
    mobs1 = mobs1/pow2;
    mobs2 = mobs2/pow2;
    err = sqrt((mobs2-pow(mobs1,2))/(pow2-1));  //calc first estimate of error
    if (bool_err_mscss == false){
```

```
        //create filename depending on wihch error is calculated
        sprintf(file3,"Err_mm_2-d_Ising_Cluster_L=%d_beta=%g_pow2=%d.dat",
        L,beta,pow2);
    }
    else{
        sprintf(file3,"Err_scss_2-d_Ising_Cluster_L=%d_beta=%g_pow2=%d.dat",
        L,beta,pow2);
    }
    ofstream out_err(file3,ios::out);                    //create error datafile
    out_err<<err<<"\n";                          //write error in error datafile
    do {
        count4=2*count4;
        old_err = err;                           //save new to old for comparison
        mobs1 = 0;                                          //set again to zero
        mobs2 = 0;
        for (int i=0; i<pow2/count4; i++){
            MOBS1[i] = (MOBS1[2*i]+MOBS1[2*i+1])/2;            //bin two values
            mobs1 = mobs1+MOBS1[i];                  //calculate mean observable
            mobs2 = mobs2+pow(MOBS1[i],2);         //and mean squared observable
        }
        mobs1 = mobs1/(pow2/count4);
        mobs2 = mobs2/(pow2/count4);
        //calc new estimate of error
        err = sqrt((mobs2-pow(mobs1,2))/(pow2/count4-1));
        out_err<<err<<"\n";                      //write error in error datafile
//if error got much smaller (happens sometimes, if there's not much data left)
        if (0.95*old_err>err){
            cout<<"Error got smaller."<<"\n";
            err_ofl = true;
        }
    //do bining until the error has reached a plateau or until there's not
    //enough data to bin again and calculate a reasonable error
    } while ((old_err<0.95*err)&&(pow2/count4>512));
    //if there are only 512 datapoints left and error is still growing
    if (old_err<0.95*err){
        cout<<"Not enough data to calculate error or error is too big."
        <<"\n";
        err_ofl = true;
    }
    out_err.close();                                     //close error datafile
}
```

```
/*****************************************
 *  configuration method                 *
 *  Writes the spin confguration         *
 *  in a file. (used for visualisation)  *
 *****************************************/

void configuration(){
    char file4[100];                    //array for configuration datafilename
    //create filename
    sprintf(file4,"Conf_2-d_Ising_Cluster_L=%d_beta=%g_pow2=%d_sweep=%d.dat"
    ,L,beta,pow2,count1);
    ofstream out_con(file4,ios::out);          //create configuration datafile
    for (int i=0; i<L2; i++){                //write spin configuration in file
        out_con<<s[i]<<"\t";
        if ((i+1)%L==0){
            out_con<<"\n";
        }
    }
    out_con.close();                             //close configuration datafile
}

/*****************************************
 *  bond configuration method            *
 *  Writes the right and the lower bonds *
 *  each in a file.                      *
 *  (used for visualisation)             *
 *****************************************/

void bonds(){
    char file4[100];                    //array for configuration datafilename
    //create filename
    sprintf(file4,
    "RightBonds_2-d_Ising_Cluster_L=%d_beta=%g_pow2=%d_sweep=%d.dat"
    ,L,beta,pow2,count1);
    ofstream out_BondRight(file4,ios::out);    //create configuration datafile
    for (int i=0; i<L2; i++){         //write right bond configuration in file
        out_BondRight<<b[i][0]<<"\t";
        if ((i+1)%L==0){
            out_BondRight<<"\n";
        }
    }
    out_BondRight.close();                       //close configuration datafile
    //create filename
    sprintf(file4,
    "LowerBonds_2-d_Ising_Cluster_L=%d_beta=%g_pow2=%d_sweep=%d.dat"
    ,L,beta,pow2,count1);
    ofstream out_BondLow(file4,ios::out);      //create configuration datafile
```

```
    for (int i=0; i<L2; i++){          //write lower bond configuration in file
        out_BondLow<<b[i][1]<<"\t";
        if ((i+1)%L==0){
            out_BondLow<<"\n";
        }
    }
    out_BondLow.close();                          //close configuration datafile
}
```

B.4 Separates Fehleranalyseprogramm

```
/*************************************************************************
 *  Seperate error analysis for the Ising Model in one and two dimensions *
 *  with bining method. The error can be calculated as it is done in the  *
 *  Ising Model programs ore until only two data points are left (for     *
 *  investigation of the behaviour of the error).                         *
 *  When the Metropolis Algorithm was used, it reads in a                 *
 *  magnetization data file, when the Cluster algorithm was used, it      *
 *  reads in a magnetization data file and a scss data file.              *
 *  Bachelor Thesis                                                       *
 *                                                                        *
 *  Marcel Wirz, summer semester 2007                                     *
 *                                                                        *
 *************************************************************************/

/*****************************************
 *  used libraries                       *
 *****************************************/

#include <cstdlib>                                        //standard library
#include <cmath>                                 //used for mathematic operations
#include<fstream>                                  //used to write data in a file
#include <iostream>                                      //used to print out data
#include <iomanip>                  //used to manipulate console output format
using std::cout;                             //used with "#include <iostream>"

using namespace std;
```

```
/*************************************************************************
 *  Because of bining, the program uses only the last 2^n<=N-M           *
 *  measurements. So N-M should be equal or a little bigger than a       *
 *  powers of two:                                                       *
 *                                                                       *
 *  2^1       2^2       2^3       2^4       2^5       2^6       2^7      *
 *  2         4         8         16        32        64        128      *
 *                                                                       *
 *  2^8       2^9       2^10      2^11      2^12      2^13      2^14     *
 *  256       512       1024      2048      4096      8192      16384    *
 *                                                                       *
 *  2^15      2^16      2^17      2^18      2^19      2^20      2^21     *
 *  32768     65536     131072    262144    524288    1048576   2097152  *
 *                                                                       *
 *  2^22      2^23      2^24      2^25      2^26      2^27      2^28     *
 *  4194304   8388608   16777216  33554432  67108864  134217728 268435456 *
 *                                                                       *
 *************************************************************************/

/*****************************************
 *  definition of global variables       *
 *****************************************/

//The following data is needed; also the name of the datafile is needed

const int L = 4;                        //L = number of sites; limited to 2^31-1
const int N = 1070000;         //number of metropolis sweeps; limited to 2^31-1
const int M = 0;//numb of ignor sweeps (system at equil.); sweep 0 counted too
const double beta = 0.825;   //1/(Boltzman konstant*temperature) (start value)
const int d = 2;                                    //dimension of Ising Model
const double J = 1;                           //ferromacnetic coupling constant
bool Cluster = false; //true=error analysis for Cluster alg, false=for Met alg
bool full_err = true;//false=standard err is calc as it is done in the simula-
//tion programs; true=the err are calc until there are only 2 data points left

double Mag[N];                                  //measurements of magnetization
unsigned long long SCSS[N];           //array for sum of cluster sizes squared
int pow2;                           //number of used measurements (power of two)
double MOBS1[N];                              //array for mobs1 (used for bining)
double MOBS2[N];                //array for mobs2 (used for error calculation)
double mobs1;                          //mean observable1 (mean mag or mscss)
double mobs2;                  //mean observable2 (mean squared mag or mscss)
double mobs4;                                 //mean (magnetization to four)
double a_sus;                           //analytic result for susceptibility
double sus;                                                 //susceptibility
double err_mobs1,err_mobs2;                          //error of mobs1 and mobs2
double* pERR_MOBS1;              //pointers for ERR_MOBS1 and ERR_MOBS2 arrays
```

```
double* pERR_MOBS2;
double* pERR_SUS;
double err;  //error calc in error subrutine (becomes error of mobs1 or mobs2)
double err_sus;                                        //error of susceptibility
bool err_second;   //true=error of mobs1 is calc, flase=error of mobs2 is calc
bool err_ofl;                                                  //error overflow
//true=error is still growing after as many as possible binings are done
char file1[100];                                       //array for datafilenames

/*****************************************
 *  definition of methods                *
 *****************************************/

void error();
void analytic();

/*****************************************
 *  main method                          *
 *****************************************/

int main(int argc, char *argv[]){
    //read in data from magnetization datafile
    ifstream in("Mag_2-d_Ising_Met_L=4_beta=0.825_N=1070000.dat");
    for (int i=0; i<N; i++){                            //write it in Mag array
        in>>Mag[i];
    }
    if (Cluster==true){
        //read in data from scss datafile
        ifstream in("SCSS_2-d_Ising_Cluster_L=3_beta=0.3_N=16.dat");
        for (int i=0; i<N; i++){                       //write it in SCSS array
            in>>SCSS[i];
        }
    }
    int count3 = -1;//find gratest pow of two which is smaller or equal to N-M
    do {
        count3++;
    } while (pow(2.,count3+1)<=N-M);
    pow2 = int(pow(2.,count3));
    if (full_err==true){
        pERR_MOBS1  = new double[count3];//bining will be done count3-1 times,
        pERR_MOBS2 = new double[count3]; //so count3 (one more for err without
        pERR_SUS = new double[count3];   //bining) places are needed in arrays
    }
    for (int i=0; i<pow2; i++){
        Mag[i] = Mag[N-pow2+i];    //overwrite the first N-pow2 magnetizations
        if (Cluster==true){
            SCSS[i] = SCSS[N-pow2+i];        //overwrite the first N-pow2 scss
```

```
    }
    MOBS1[i] = Mag[i];                          //create OBS1[] and OBS2[]
    MOBS2[i] = pow(Mag[i],2);
}
err_second = false;
err_ofl = false;
error();                    //calculate the error of the mean magnetization
err_mobs1 = err;
if (Cluster==true){
    for (int i=0; i<pow2; i++){                //recreate MOBS[] with scss
        MOBS1[i] = SCSS[i];
    }
}else{
    for (int i=0; i<pow2; i++){        //recreate MOBS1[] with observable2
        MOBS1[i] = MOBS2[i];
    }
}
err_second = true;//use the same subrutine to calculate the error of mobs2
error();
err_mobs2 = err;
if (full_err==false){
    if (Cluster==true){
        mobs1 = 0;
        for (int i=0; i<pow2; i++){         //calculate mean magnetization
            mobs1 = mobs1+Mag[i];
        }
        mobs1 = mobs1/pow2;
        mobs2=0;
        for (int i=0; i<pow2; i++){                  //calculate mean scss
            mobs2 = mobs2+SCSS[i];
        }
        mobs2 = mobs2/pow2;
        sus = 1/pow(float(L),d)*mobs2;          //calculate susceptibility
        err_sus = 1/pow(float(L),d)*err_mobs2;
    }else{
        mobs1 = 0;
        mobs2 = 0;
        for (int i=0; i<pow2; i++){         //calculate mean magnetization
            mobs1 = mobs1+Mag[i];         //and mean squared magnetization
            mobs2 = mobs2+pow(Mag[i],2);
        }
        mobs1 = mobs1/pow2;
        mobs2 = mobs2/pow2;
        sus = 1/pow(float(L),d)*(mobs2-pow(mobs1,2));//calc susceptibility
        err_sus = 1/pow(float(L),d)*sqrt(pow(err_mobs2,2)
        +pow(2*mobs1*err_mobs1,2));
    }
```

```
    if (err_ofl==true){       //if one of the errors didn't reach a plateau
        err_sus = 0;                  //set the whole error to zero because
    }                                         //there's no reasonable error
    if (d==1){
        analytic();
    }
    cout<<"Beta:  "<<beta<<"\n"<<"\n";                //print out data
    if (d==1){
        cout<<"Analytic result for susceptibility (d=1 and B=0):"<<"\n";
        cout<<a_sus<<"\n";
    }
    cout<<"Susceptibility:"<<"\n";
    cout<<sus<<"+-"<<err_sus<<"\n";
    cout<<"lower value:  "<<sus-err_sus<<"\n";
    cout<<"upper value:  "<<sus+err_sus<<"\n"<<"\n"<<"\n";
}else{
    if (Cluster==true){
        mobs1 = 0;
        for (int i=0; i<pow2; i++){         //calculate mean magnetization
            mobs1 = mobs1+Mag[i];
        }
        mobs1 = mobs1/pow2;
        mobs2=0;
        for (int i=0; i<pow2; i++){                 //calculate mean scss
            mobs2 = mobs2+SCSS[i];
        }
        mobs2 = mobs2/pow2;
        //calculate the error of the susceptibility after each bining step
        for (int i=0; i<count3; i++){
            pERR_SUS[i] = 1/pow(float(L),d)*pERR_MOBS2[i];
        }
        //create filename
        sprintf(file1,
        "Err_mobs1_%d-d_Ising_Cluster_L=%d_beta=%g_pow2=%d.dat",
        d,L,beta,pow2);
        ofstream out_err_mobs1(file1,ios::out);          //create datafile
        for (int i=0; i<count3; i++){
            out_err_mobs1<<pERR_MOBS1[i]<<"\n";      //write in datafile
        }
        out_err_mobs1.close();                            //close datafile
        sprintf(file1,
        "Err_mscss_%d-d_Ising_Cluster_L=%d_beta=%g_pow2=%d.dat",
        d,L,beta,pow2);
        ofstream out_err_mscss(file1,ios::out);
        for (int i=0; i<count3; i++){
            out_err_mscss<<pERR_MOBS2[i]<<"\n";
        }
```

```
        out_err_mscss.close();
        sprintf(file1,"Err_sus_%d-d_Ising_Cluster_
        L=%d_beta=%g_pow2=%d.dat",d,L,beta,pow2);
        ofstream out_err_sus(file1,ios::out);
        for (int i=0; i<count3; i++){
            out_err_sus<<pERR_SUS[i]<<"\n";
        }
        out_err_sus.close();
        cout<<"Errors:"<<"\n\n"<<setw(8)<<left<<"Sus:"<<"\t\t"
        <<setw(8)<<left<<"mobs1:"<<"\t\t"<<setw(8)<<left<<"mobs2:"
        <<"\n";
        for (int i=0; i<count3; i++){                    //print out data
            cout<<fixed<<pERR_SUS[i]<<"\t\t"<<pERR_MOBS1[i]<<"\t\t"
            <<pERR_MOBS2[i]<<"\n";
        }
    }else{
        mobs1 = 0;
        for (int i=0; i<pow2; i++){          //calculate mean magnetization
            mobs1 = mobs1+Mag[i];
        }
        mobs1 = mobs1/pow2;
        //calculate the error of the susceptibility after each bining step
        for (int i=0; i<count3; i++){
            pERR_SUS[i] = 1/pow(float(L),d)*sqrt(pow(pERR_MOBS2[i],2)
            +pow(2*mobs1*pERR_MOBS1[i],2));
        }
        //create filename
        sprintf(file1,"Err_mobs1_%d-d_Ising_Met_L=%d_beta=%g_pow2=%d.dat",
        d,L,beta,pow2);
        ofstream out_err_mobs1(file1,ios::out);          //create datafile
        for (int i=0; i<count3; i++){
            out_err_mobs1<<pERR_MOBS1[i]<<"\n";      //write in datafile
        }
        out_err_mobs1.close();                            //close datafile
        sprintf(file1,"Err_mobs2_%d-d_Ising_Met_L=%d_beta=%g_pow2=%d.dat",
        d,L,beta,pow2);
        ofstream out_err_mobs2(file1,ios::out);
        for (int i=0; i<count3; i++){
            out_err_mobs2<<pERR_MOBS2[i]<<"\n";
        }
        out_err_mobs2.close();
        sprintf(file1,"Err_sus_%d-d_Ising_Met_L=%d_beta=%g_pow2=%d.dat",
        d,L,beta,pow2);
        ofstream out_err_sus(file1,ios::out);
        for (int i=0; i<count3; i++){
            out_err_sus<<pERR_SUS[i]<<"\n";
        }
```

```
            out_err_sus.close();
            cout<<"Errors:"<<"\n\n"<<setw(8)<<left<<"Sus:"<<"\t\t"
            <<setw(8)<<left<<"mobs1:"<<"\t\t"<<setw(8)<<left<<"mobs2:"
            <<"\n";
            for (int i=0; i<count3; i++){                    //print out data
                cout<<fixed<<pERR_SUS[i]<<"\t\t"<<pERR_MOBS1[i]<<"\t\t"
                <<pERR_MOBS2[i]<<"\n";
            }
        }
    }
    system("PAUSE");
    return 0;
}

/*****************************************
 *  error method                         *
 *  Calculates the error                 *
 *  with bining method.                  *
 *****************************************/

void error(){
    char file3[100];                              //array for error datafilename
    double old_err;                                  //old error for comparison
    int count4 = 1;                                                //number of bins
//counts how many bins are done (used for writing pos in ERR_MM/ERR_MM2 array)
    int count5 = 0;
    mobs1 = 0;
    mobs2 = 0;
    for (int i=0; i<pow2; i++){                                  //calculate mobs1
        mobs1 = mobs1+MOBS1[i];                                        //and mobs2
        mobs2 = mobs2+pow(MOBS1[i],2);
    }
    mobs1 = mobs1/pow2;
    mobs2 = mobs2/pow2;
    err = sqrt((mobs2-pow(mobs1,2))/(pow2-1)); //calcu first estimate of error
    if (full_err==false){
        if (err_second == false){
            //create filename depending on wihch error is calculated
            sprintf(file3,"Err_mobs1_%d-d_Ising_Met_L=%d_beta=%g_pow2=%d.dat",
            d,L,beta,pow2);
        }
        else{
            sprintf(file3,"Err_mobs2_%d-d_Ising_Met_L=%d_beta=%g_pow2=%d.dat",
            d,L,beta,pow2);
        }
        ofstream out_err(file3,ios::out);                //create error datafile
        out_err<<err<<"\n";                        //write error in error datafile
```

```
        do {
            count4 = 2*count4;
            old_err = err;                    //save new to old for comparison
            mobs1 = 0;                                  //set again to zero
            mobs2 = 0;
            for (int i=0; i<pow2/count4; i++){
                MOBS1[i] = (MOBS1[2*i]+MOBS1[2*i+1])/2;       //bin two values
                mobs1 = mobs1+MOBS1[i];          //calculate mean observable 1
                mobs2 = mobs2+pow(MOBS1[i],2); //and mean squared observable 2
            }
            mobs1 = mobs1/(pow2/count4);
            mobs2 = mobs2/(pow2/count4);
            //calc new estimate of error
            err = sqrt((mobs2-pow(mobs1,2))/(pow2/count4-1));
            out_err<<err<<"\n";              //write error in error datafile
//if error got much smaller (happens sometimes, if there's not much data left)
            if (0.95*old_err>err){
                cout<<"Error got smaller."<<"\n";
                err_ofl = true;
            }
        //do bining until the error has reached a plateau or until there's not
        //enough data to bin again and calculate a reasonable error
        } while ((old_err<0.95*err)&&(pow2/count4>512));
        //if there are only 512 datapoints left and error is still growing
        if (old_err<0.95*err){
            cout<<"Not enough data to calculate error or error is too big."
            <<"\n";
            err_ofl = true;
        }
        out_err.close();                                //close error datafile
    }else{
        if (err_second == false){
            pERR_MOBS1[count5] = err;
        }
        else{
            pERR_MOBS2[count5] = err;
        }
        count5 = 1;
        do {
            count4 = 2*count4;
            old_err = err;                    //save new to old for comparison
            mobs1 = 0;                                  //set again to zero
            mobs2 = 0;
            for (int i=0; i<pow2/count4; i++){
                MOBS1[i] = (MOBS1[2*i]+MOBS1[2*i+1])/2;       //bin two values
                mobs1 = mobs1+MOBS1[i];          //calculate mean magnetization
                mobs2 = mobs2+pow(MOBS1[i],2);//and mean squared magnetization
            }
```

```
            mobs1 = mobs1/(pow2/count4);
            mobs2 = mobs2/(pow2/count4);
            //calc new estimate of error
            err = sqrt((mobs2-pow(mobs1,2))/(pow2/count4-1));
            if (err_second == false){
                pERR_MOBS1[count5] = err;
            }
            else{
                pERR_MOBS2[count5] = err;
            }
            count5 = count5+1;
        }while (pow2/count4>2);        //do bining until there are only 2 data
                                       //points left if error is still growing
    }
}

/****************************************
 *  analytic method                     *
 *  Calculates the exact result of the  *
 *  susceptibility for B=0.             *
 ****************************************/

void analytic(){
    a_sus = (1-pow(tanh(beta*J),L))*exp(2*beta*J)/(1+pow(tanh(beta*J),L));
}
```

B.5 Autokorrelationsfunktion

```
/*************************************************************************
 *  Autocorrelation function                                             *
 *  Reads in a magnetization data file and calculates the discrete       *
 *  autocorrelation function.                                            *
 *  C_O(t) = <O(s(k)*O(s(k+t))> - <O>^2                                  *
 *  Can also be used for another observable.                             *
 *  Bachelor Thesis                                                      *
 *                                                                       *
 *  Marcel Wirz, summer semester 2007                                    *
 *                                                                       *
 *************************************************************************/

/*****************************************
 *  used libraries                       *
 *****************************************/

#include <cstdlib>                                          //standard library
#include <cmath>                                  //used for mathematic operations
#include<fstream>                                   //used to write data in a file
#include <iostream>                                       //used to print out data
using std::cout;                                 //used with "#include <iostream>"

using namespace std;

/*****************************************
 *  definition of global variables       *
 *****************************************/

//The following data is needed; also the name of the datafile is needed

const int N = 1070000;        //number of metropolis sweeps; limited to 2^31-1
const int M = 0; //numb of ignored sweeps (system at equil); sweep 0 count too
const int nr_v = 600;           //number of values to be calculated; has to be
//smaller or equal to the gratest power of 2 which is smaller or equal to N-M
const int d = 2;                      //number of dimensions (used for filename)
const int L = 100;                     //L = number of sites (used for filename)
const double beta = 0.7;                                  //(used for filename)
bool cluster = true;                        //if Cluster or Metropolis was used


double Mag[N];                                    //measurements of magnetization
double Corr[nr_v];                  //array of the correlation function values
int pow2;                          //number of used measurements (power of two)
double mm;                                                  //mean magnetization
double corr;                              //a value of the correlation function
```

```
/*****************************************
 *  main method                          *
 *****************************************/

int main(int argc, char *argv[]){
    char file1[100];                                    //array for datafilename
    //read in data from datafile
    ifstream in("Mag_2-d_Ising_Cluster_L=100_beta=0.7_N=1070000.dat");
    for (int i=0; i<N; i++){                             //write it in Mag array
        in>>Mag[i];
    }
    int count3 = -1; //find gratest pow. of 2 which is smaller or equal to N-M
    do {
        count3++;
    } while (pow(2.,count3+1)<=N-M);
    pow2 = int(pow(2.,count3));
    for (int i=0; i<pow2; i++){
        Mag[i] = Mag[N-pow2+i];    //overwrite the first N-pow2 magnetizations
    }
    mm = 0;
    for (int i=0; i<pow2; i++){                 //calculate mean magnetization
            mm = mm+Mag[i];
    }
    mm = mm/pow2;
    for (int i=0; i<nr_v; i++){     //calc autocorrelation fkt for nr_v values
        corr = 0;                               //calculate autocorrelation fkt
        for (int j=0; j<(pow2-i); j++){
             corr = corr+Mag[j]*Mag[j+i];
        }
        Corr[i] = corr/(pow2-i)-pow(mm,2);
    }
    //create filename
    if (cluster==false){
        sprintf(file1,"Corr_%d-d_Ising_Met_L=%d_beta=%g_pow2=%d.dat",
        d,L,beta,pow2);
    }else{
        sprintf(file1,"Corr_%d-d_Ising_Cluster_L=%d_beta=%g_pow2=%d.dat",
        d,L,beta,pow2);
    }
    ofstream out_corr(file1,ios::out);                      //create datafile
    for (int i=0; i<nr_v; i++){
        out_corr<<Corr[i]<<"\n";                //write results in datafile
    }
    out_corr.close();                                       //close datafile
    system("PAUSE");              //console application shoud stay at the end
     return 0;                                             //no error occured
}
```

B.6 Mathematicacodes

Mathematica wurde verwendet, um Konfigurationsbilder wie 3.11 und Visualisierungsbilder der Bondsetzung wie 3.12 zu generieren. Es folgen die entsprechenden Codes, wobei konkrete Filenamen eingesetzt sind.

B.6.1 Code für Konfigurationsbilder

```
In[1]:= L := 100
In[2]:= conf :=
         Import["Conf_2-d_Ising_Cluster_L=100_beta=0.3_pow2=
            1048576_sweep=1069999.dat"]
In[3]:= pict :=
         Show[
          Graphics[
           Table[If[conf[[i, j]] == -1, Rectangle[{j - 1, -i + 1}, {j, -i}]],
            {i, 1, L}, {j, 1, L} ]], AspectRatio -> Automatic]
In[4]:= Export["2d_Cluster_L=100_Config_Beta=0.3.eps", pict]
```

B.6.2 Code für Bondvisualisierung

```
In[1]:= L := 20
In[2]:= conf :=
         Import["Conf_2-d_Ising_Cluster_L=20_beta=0.1_pow2=1048576
            _sweep=1069999.dat"]
In[3]:= BondsR :=
         Import["RightBonds_2-d_Ising_Cluster_L=20_beta=0.1_pow2=
            1048576_sweep=1069999.dat"]
In[4]:= BondsL :=
         Import["LowerBonds_2-d_Ising_Cluster_L=20_beta=0.1_pow2=
            1048576_sweep=1069999.dat"]
In[5]:= Bonds :=
         Show[
          Graphics[
           {Table[If[conf[[i, j]] == 1, Circle[{j, -i}, 0.25],
              Disk[{j, -i}, 0.25]], {i, 1, L}, {j, 1, L} ],
            Table[If[BondsR[[i, j]] == 1,
              {Thickness[0.007], Line[{{j, -i}, {j + 1, -i}}]}], {i, 1, L},
             {j, 1, L} ],
            Table[If[BondsL[[i, j]] == 1,
              {Thickness[0.007], Line[{{j, -i}, {j, -i - 1}}]}],
             {i, 1, L},{j, 1, L} ]}], AspectRatio -> Automatic]
In[6]:= Export["2d_Cluster_L=20_Bonds_Beta=0.1.eps", Bonds]
```

Danksagung

Ich möchte Prof. Uwe-Jens Wiese für die sehr gute Betreuung und dass er sich immer sehr viel Zeit genommen hat, alle meine Fragen zu beantworten, herzlich danken. Dieser Dank geht auch an den Assistenten Matthias Nyfeler für all seine fachkundigen Auskünfte. Nicht zuletzt möchte ich mich auch bei meinem Studienkollegen Bänz Bessire, der seine Bachelorarbeit zum gleichen Thema geschrieben hat, für die super Zusammenarbeit bedanken und bei all den vielen anderen, die mir in irgend einer Art und Weise mit Rat und Tipps geholfen haben.

Literaturverzeichnis

[1] Uwe-Jens Wiese: *Physics with the Computer: The Ising Model with Monte Carlo Methods*, Institut für theoretische Physik, Universität Bern, September, 2005.

[2] F. Niedermayer: *Notes on Monte Carlo Simulations of Systems in Thermal Equilibrium*, Institut für theoretische Physik, Universität Bern, September, 2005.

[3] F. Niedermayer: *Statistical Physics II*, 2005.

[4] M. E. J. Newman und G. T. Barkema: *Monte Carlo Methods in Statistical Physics,* Clarendon Press, Oxford, 1999.

[5] L. Onsager:*Crystal statistics. 1. a two-dimensional model with an order disorder transition*, Phys. Rev. Lett., 65:117-149, 1944.

[6] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller: *Equation of state calculations by fast computing machines*, J. Chem. Phys., 21:1087-1092, 1953.

[7] R. H. Swendsen and J.-S. Wang: *Nonuniversal critical dynamics in monte carlo simulations*, Phys. Rev. Lett., 58:86-88, 1987.

[8] M. Nyfeler: *Einführung in C/C++ für die Statistische Thermodynamik I*, August, 2006.

[9] K. Langfeld: *Ising Modell: Monte Carlo Sampling auf dem Gitter, Phasenübergänge*, http://www.tat.physik.uni-tuebingen.de/∼kley/lehre/cp-prakt/projekte/projekt2.pdf, Stand 09.07.2007.