Implementierung der

Stochastic Series Expansion

für Spin – 1/2 Heisenberg Systeme

29. Juli 2011

Lukas B. Lentner

Vorwort

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

Lukas B. Lentner
kontakt@lukaslentner.de

München, 29. Juli 2011

Inhaltsverzeichnis

Kapitel 1

Einleitung

Kapitel 2

Theorie der Monte Carlo Simulation

2.1 Geschichte

Basierend auf den Ideen von Enrico Fermi (um 1935) verwendete zum ersten Mal Stanislaw Ulam und John von Neumann um 1945 das Prinzip der Monte Carlo Methode während ihrer Arbeit am Los Alamos Scientific Laboratory.

Der von Nicholas Metropolis gewählte Name bezieht sich auf die Spielbank Monte Carlo, die im gleichnamigen Stadtteil des Stadtstaates Monaco liegt. Anlass hierfür soll Ulams Onkel gegeben haben, der sich mehrmals von Verwandten Geld zum spielen leihen wollte [Met87].

Heute findet die Methode zahlreiche Anwendungen in der Statistischen Physik, Numerik und Optimierung.

2.2 Ziel

Die Idee der Monte Carlo Simulation (MCS) lässt sich beschreiben als ein gewichteter Weg durch einen n-dimensionalen Zustandsraum Ω. Hierbei interessiert man sich speziell für den statistischen Mittelwert einer Größe A,

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

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

2.3 Markov-Kette

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

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

genähert werden.

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

a) Die Bildung der Kette muss **ergodisch** sein. D.h. sie muss theoretisch alle Zustände enthalten können, was sie in der Praxis natürlich nicht tut, da wir $M \ll |\Omega|$ wählen.

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

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

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

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

In Worten besagt sie, dass ein Sprung von einem Markov-Kettenglied zum Nachbar genauso wahrscheinlich ist, wie andersherum. Die Kette besitzt also keine ausgezeichnete Richtung.

Gleichung 2.4 erfüllt automatisch Gl. 2.3, da

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

Hierbei verwendet man im letzten Schritt, dass der Zustand ν in jedem Fall in irgendeinen nächsten Zustand σ übergeht. Diese zunächst starke Einschränkung wird häufig verwendet, um der Bedingung b) zu genügen. Später werden wir sehen, dass sie in unserem Fall auch die Berechnung von $\boldsymbol{W}$ deutlich vereinfacht.

2.4 Metropolis Algorithmus

Ein möglicher Algorithmus zur Bestimmung der Übergangswahrscheinlichkeiten $\boldsymbol{W}$ wurde 1953 von Nicholas Metropolis et al. vorgestellt [MRR$^+$53],

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

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

2.5 Thermalisierung

Nachdem als Anfangszustand der Markov-Kette ein beliebig ausgewählter Zustand verwendet wird, ist es ziemlich unwahrscheinlich, dass dieser Zustand ein hohes Wahrscheinlichkeitsgewicht P_σ besitzt. Es wird sich also nicht um einen Zustand im Gleichgewicht handeln. Aus diesem Grund sollte vor der eigentlichen Messung eine genügend große Anzahl von Thermalisierungsschritten (MC-Schritte) durchgeführt werden.

In der Praxis werden entweder Erfahrungswerte verwendet, die eine konstante, meist zu große Schrittanzahl erfordern oder die Daten werden vollständig gespeichert und in der Auswertung sortiert. Im Nachhinein kann über tatsächliche Wahrscheinlichkeitsverteilung auf die Thermalisierungsphase geschlossen werden. Diese Daten werden dann für die anschließende Analyse nicht verwendet.

2.6 Autokorrelationsfunktion und Fehlerberechnung

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

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

betrachtet. Hierbei läuft die Mittelwertbildung mit der Variable i über die gesamte ausgewertete Simulationszeit $\widetilde{M}$ (Die gesamte Simulationszeit beträgt M). Sie ist in solch einer Weise normiert, dass $\Theta_A(\sigma_0) = 1$ und $\Theta_A(\sigma_{t\to\infty}) = 0$. Die Autokorrelationsfunktion hängt sodann mit der Autokorrelationszeit negativ exponentiell zusammen,

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

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

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

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

$$B = \left\lfloor \frac{\widetilde{M}}{3\tau_A} \right\rfloor \ . \tag{2.10}$$

Nach dem zentralen Grenzwert Satz folgen diese Gruppenmittelwerte sodann einer Gauß-Verteilung, welche den exakten Wert

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

in der Mitte hält. Als Fehler kann eine Standardabweichung, also

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

angegeben werden.

Kapitel 3

Klassische MCS
am Beispiel des Ising Modells

Um die Grundlagen einer Monte Carlo Simulation (MCS) kennenzulernen, wurde im Vorfeld eine Anwendung zur Simulation des 2-dimensionalen Ising-Modells mit periodischer Randbedingung erstellt. Die gewonnene Erfahrung erwies sich für das Studium der quantenmechanischen MCS als äußerst hilfreich.

3.1 Methode

3.1.1 Das Ising-Modell

Für das klassische, ferromagnetische Ising-Modell ist der Hamiltonian

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

zusammengesetzt aus einer z-Koppelung benachbarter Spins $\langle i, j \rangle$ (wird später über das Gittermodell definiert), die durch die Bindungsmatrix $\boldsymbol{J}$ gewichtet wird, und einer magnetischen Wechselwirkung, in die das externe Magnetfeld $\boldsymbol{B} = (0, 0, B)^T$ und das Magnetische Moment $\boldsymbol{\mu} = (0, 0, \mu)^T$ eingeht. Für unser Beispiel setzten wir jedes $J_{ij} = 1$ und betrachten die Anordnung ohne Magnetfeld – da uns nur die z-Richtung interessiert setzen wir $S = S^z$. Der Hamiltonian erhält dann die vereinfachte Struktur,

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

3.1.2 Kanonische Übergangswahrscheinlichkeiten

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

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

kanonisch an, wobei

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

die kanonische Zustandssumme, β die reduzierte Temperatur $1/T$ und E_σ die Energie eines gewissen mikroskopischen Zustandes σ (Konfiguration) darstellt. Im Vergleich zur Gl. 2.1 sieht man, dass die Wahrscheinlichkeit eines Zustandes σ

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

boltzmannverteilt ist. Diese Gewichte sind schwer zu berechnen, da der Zustandsraum in solch einem Spin-System exponentiell mit der Spinanzahl anwächst ($|\Omega| \sim 2^N$). Machen wir allerdings vom Metropolis Algorithmus Gebrauch (siehe Gl. 2.6), benötigen wir die einzelnen Gewichte gar nicht, sondern können uns mit deren Verhältnissen, die dann die Übergangswahrscheinlichkeiten $\boldsymbol{W}$ darstellen, genügen,

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

3.1.3 Messgrößen

Folgende typischen, thermodynamischen Größen wollen wir in unserer Beispielanwendung messen:

$$\text{Energie}: \quad E = \langle H \rangle \ , \tag{3.7}$$

$$\text{Magnetisierung pro Spin}: \quad M = \langle S_i \rangle \ , \tag{3.8}$$

$$\text{Spezifische Wärme}: \quad C_V = \frac{1}{NkT^2} \left(\langle H^2 \rangle - \langle H \rangle^2 \right) \ , \tag{3.9}$$

$$\text{Magnetische Suszeptibilität}: \quad \chi_m = \frac{N}{kT} \left(\langle S_i^2 \rangle - \langle S_i \rangle^2 \right) \ . \tag{3.10}$$

3.2 Implementierung

Hallo

3.3 Ergebnisse

relaxation time goes up for transition

aussicht cluster alg

finite size scaling critical exp

Vergleich mit Meanfield

Kapitel 4

Quantenmechanische MCS
mit Hilfe der Stochastic Series Expansion

Kapitel 5

Zusammenfassung

Anhang A

Quellcode

A.1 Hauptprogramm SIM

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

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

#include <cstdlib>
#include <iostream>

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

  try {

    if(argc != 8) throw "[SIM] Error: Please specify 7 parameters (Size, Lattice-Index, Measure

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

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

    AbstractLattice* lattice;
    const char* latticeLabel;

    switch(latticeIndex) {

      case 0:
        lattice = new Open1DLattice(size);
        latticeLabel = "Open-1D";
        break;

      case 1:
```

```
39            lattice = new Periodic1DLattice(size);
40            latticeLabel = "Periodic-1D";
41            break;
42
43          case 2:
44            lattice = new Periodic2DLattice(size);
45            latticeLabel = "Periodic-2D";
46            break;
47
48          default:
49            throw "[SIM] Error: The Lattice-Index has to be within [0;2]";
50
51        }
52
53        if(measureCount <= 0) throw "[SIM] Error: The Measure-Count has to be positive";
54
55        AbstractAlgorithm* algorithm;
56        const char* algorithmLabel;
57
58        switch(algorithmIndex) {
59
60          case 0:
61            algorithm = new EDAlgorithm(lattice, measureCount);
62            algorithmLabel = "ED";
63            break;
64
65          case 1:
66            algorithm = new ISINGAlgorithm(lattice, measureCount);
67            algorithmLabel = "ISING";
68            break;
69
70          case 2:
71            algorithm = new SSEAlgorithm(lattice, measureCount);
72            algorithmLabel = "SSE";
73            break;
74
75          default:
76            throw "[SIM] Error: The Algorithm-Index has to be within [0;2]";
77
78        }
79
80        if(startTemperature <  0) throw "[SIM] Error: The Start-Temperature has to be positi
81        if(endTemperature   <= 0) throw "[SIM] Error: The End-Temperature has to be positive
82        if(temperatureStep  <= 0) throw "[SIM] Error: The Temperature-Step has to be positiv
83        if(startTemperature > endTemperature) std::swap(startTemperature, endTemperature);
84
85        printf("#\n");
86        printf("# SIM - DATA\n");
87        printf("#\n");
88        printf("# ----------------------------------------\n");
89        printf("#\n");
90        printf("# Size                  = %+19.19i\n", size);
91        printf("# Lattice               = %s\n", latticeLabel);
92        printf("# Measure-Count         = %+19.19li\n", measureCount);
93        printf("# Algorithm             = %s\n", algorithmLabel);
94        printf("# Start-Temperature     = %+20.13e\n", startTemperature);
95        printf("# End-Temperature       = %+20.13e\n", endTemperature);
96        printf("# Temperature-Step      = %+20.13e\n", temperatureStep);
97        printf("#\n");
98        printf("# %-17s | %-18s | %-18s | %-18s | %-18s | %-18s | %-18s | %-18s | %-18s | %-
99               "Size",
100              "Lattice-Index",
```

```
101                 "Measure-Count",
102                 "Algorithm-Index",
103                 "Temperature",
104                 "Average␣Energy",
105                 "StdDv␣Of␣Energy",
106                 "RelTi␣Of␣Energy",
107                 "Average␣Heat",
108                 "StdDv␣Of␣Heat",
109                 "RelTi␣Of␣Heat",
110                 "Average␣Mag",
111                 "StdDv␣Of␣Mag",
112                 "RelTi␣Of␣Mag",
113                 "Average␣Suscept",
114                 "StdDv␣Of␣Suscept",
115                 "RelTi␣Of␣Suscept");
116         printf("#␣-------------------------------------------------------------------------------
117
118         for(double temperature = endTemperature; temperature > startTemperature + (temperatureStep
119
120           algorithm->setTemperature(temperature);
121           algorithm->runTemperatureRound();
122
123           printf("%+19.19i|%+19.19i|%+19.19li|%+19.19i|%+20.13e|%+20.13e|%+20.13e|%+19.19li|%+20.13
124                  size,
125                  latticeIndex,
126                  measureCount,
127                  algorithmIndex,
128                  temperature,
129                  algorithm->getAverageEnergy(),
130                  algorithm->getErrorOfEnergy(),
131                  algorithm->getRelTiOfEnergy(),
132                  algorithm->getAverageHeat(),
133                  algorithm->getErrorOfHeat(),
134                  algorithm->getRelTiOfHeat(),
135                  algorithm->getAverageMag(),
136                  algorithm->getErrorOfMag(),
137                  algorithm->getRelTiOfMag(),
138                  algorithm->getAverageSuscept(),
139                  algorithm->getErrorOfSuscept(),
140                  algorithm->getRelTiOfSuscept());
141
142           std::cerr << "[SIM]␣Info:␣Finished,␣Size=" << size << ",␣Lattice=" << latticeLabel << ",␣
143
144         }
145
146         delete algorithm;
147         delete lattice;
148
149         return EXIT_SUCCESS;
150
151       } catch(const char *message) {
152
153         std::cerr << message << std::endl;
154         return EXIT_FAILURE;
155
156       }
157
158     };
```

Listing A.1: SIM.cpp

A.2 Gitter Klassen

A.2.1 Abstrakte Gitter

```
#ifndef CLASS_ABSTRACTLATTICE
#define CLASS_ABSTRACTLATTICE

class AbstractLattice {

  protected:

    int n;

  public:

    AbstractLattice(int n_parameter) {

      n = n_parameter;

    };

    int getN() {

      return n;

    };

    virtual int *getNeighbours(int i) = 0;

    virtual int getNb() = 0;
    virtual int getI1(int b) = 0;
    virtual int getI2(int b) = 0;

};

#endif
```

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

A.2.2 1D Gitter mit offenen Randbedingungen

```
#ifndef CLASS_OPEN1DLATTICE
#define CLASS_OPEN1DLATTICE

#include "AbstractLattice.cpp"

class Open1DLattice : public AbstractLattice {

  public:

    Open1DLattice(int n_parameter) : AbstractLattice(n_parameter) {};

    int *getNeighbours(int i) {

      int *neighbours;

      if(i == 0) {
```

```
17
18            neighbours = new int(2);
19            neighbours[0] = 1;
20            neighbours[1] = i + 1;
21
22          } else if(i == n - 1) {
23
24            neighbours = new int(2);
25            neighbours[0] = 1;
26            neighbours[1] = i - 1;
27
28          } else {
29
30            neighbours = new int(3);
31            neighbours[0] = 2;
32            neighbours[1] = i - 1;
33            neighbours[2] = i + 1;
34
35          }
36
37          return neighbours;
38
39        };
40
41        int getNb() {
42
43          return n - 1;
44
45        };
46
47        int getI1(int b) {
48
49          return b - 1;
50
51        };
52
53        int getI2(int b) {
54
55          return b;
56
57        };
58
59    };
60
61    #endif
```

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

A.2.3 1D Gitter mit periodischen Randbedingungen

```
1   #ifndef CLASS_PERIODIC1DLATTICE
2   #define CLASS_PERIODIC1DLATTICE
3
4   #include "AbstractLattice.cpp"
5
6   class Periodic1DLattice : public AbstractLattice {
7
8     public:
9
```

```
10        Periodic1DLattice(int n_parameter) : AbstractLattice(n_parameter) {};
11
12        int getNb() {
13
14          return n;
15
16        };
17
18        int *getNeighbours(int i) {
19
20          int *neighbours = new int(3);
21          neighbours[0] = 2;
22          neighbours[1] = (i - 1 + n) % n;
23          neighbours[2] = (i + 1) % n;
24
25          return neighbours;
26
27        };
28
29        int getI1(int b) {
30
31          return b - 1;
32
33        };
34
35        int getI2(int b) {
36
37          return b % n;
38
39        };
40
41    };
42
43    #endif
```

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

A.2.4 2D Gitter mit periodischen Randbedingungen

```
1     #ifndef CLASS_PERIODIC2DLATTICE
2     #define CLASS_PERIODIC2DLATTICE
3
4     #include <cmath>
5
6     #include "AbstractLattice.cpp"
7
8     class Periodic2DLattice : public AbstractLattice {
9
10      protected:
11
12        int m;
13
14      public:
15
16        Periodic2DLattice(int n_parameter) : AbstractLattice(n_parameter) {
17
18          m = sqrt((double) n_parameter);
19          if(pow((int) m, 2) != n_parameter) throw "[Periodic2DLattice] Error: You choose a
20
```

```
21        };
22
23        int *getNeighbours(int i) {
24
25          int *neighbours = new int[5];
26          neighbours[0] = 4;
27          neighbours[1] = i - m <  0     ? i - m + n : i - m;
28          neighbours[2] = i + m >= n     ? i + m - n : i + m;
29          neighbours[3] = i % m == 0     ? i + m - 1 : i - 1;
30          neighbours[4] = i % m == m - 1 ? i - m + 1 : i + 1;
31
32          return neighbours;
33
34        };
35
36        int getNb() {
37
38          return 2 * n;
39
40        };
41
42        int getI1(int b) {
43
44          return (b - 1) / 2;
45
46        };
47
48        int getI2(int b) {
49
50          if(b & 1) {
51
52            return (((b - 1) / 2) + m) % n;
53
54          } else {
55
56            return ((((b - 1) / 2) + 1) % m) + ((((b - 1) / 2) / m) * m);
57
58          }
59
60        };
61
62    };
63
64    #endif
```

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

A.3 Algorithmus Klassen

A.3.1 Abstrakter Algorithmus

```
1   #ifndef CLASS_ABSTRACTALGORITHM
2   #define CLASS_ABSTRACTALGORITHM
3
4   #include <sstream>
5   #include <string>
6
```

```
7   #include "../Lattice/AbstractLattice.cpp"
8
9   class AbstractAlgorithm {
10
11    protected:
12
13      AbstractLattice* lattice;
14      long measureCount;
15      long runCount;
16      double *energyMeasurements;
17      double *magMeasurements;
18      double t;
19      double avE;
20      double erE;
21      long rtE;
22      double avH;
23      double erH;
24      long rtH;
25      double avM;
26      double erM;
27      long rtM;
28      double avS;
29      double erS;
30      long rtS;
31
32    public:
33
34      AbstractAlgorithm(AbstractLattice* lattice_parameter, int measureCount_parameter) {
35
36        lattice = lattice_parameter;
37        measureCount = measureCount_parameter;
38        runCount = measureCount * 3 / 2;
39
40        energyMeasurements = new double[measureCount];
41        magMeasurements    = new double[measureCount];
42
43        t = 0;
44
45        avE = 0;
46        erE = 0;
47        rtE = 0;
48
49        avH = 0;
50        erH = 0;
51        rtH = 0;
52
53        avM = 0;
54        erM = 0;
55        rtM = 0;
56
57        avS = 0;
58        erS = 0;
59        rtS = 0;
60
61      };
62
63      ~AbstractAlgorithm() {
64
65        delete[] magMeasurements;
66        delete[] energyMeasurements;
67
68      };
```

```
69
70          long getMeasureCount() {
71
72            return measureCount;
73
74          };
75
76          long getRunCount() {
77
78            return runCount;
79
80          };
81
82          virtual void runTemperatureRound() = 0;
83
84          virtual void setTemperature(double t_parameter) {
85
86            t = t_parameter;
87
88            avE = 0;
89            erE = 0;
90            rtE = 0;
91
92            avH = 0;
93            erH = 0;
94            rtH = 0;
95
96            avM = 0;
97            erM = 0;
98            rtM = 0;
99
100           avS = 0;
101           erS = 0;
102           rtS = 0;
103
104         };
105
106         double getTemperature() {
107
108           return t;
109
110         };
111
112         double getEnergyMeasurement(long time) {
113
114           return energyMeasurements[time];
115
116         };
117
118         double getMagMeasurement(long time) {
119
120           return magMeasurements[time];
121
122         };
123
124         double getAverageEnergy() {
125
126           return avE;
127
128         };
129
130         double getErrorOfEnergy() {
```

```
131
132           return erE;
133
134         };
135
136         long getRelTiOfEnergy() {
137
138           return rtE;
139
140         };
141
142         double getAverageHeat() {
143
144           return avH;
145
146         };
147
148         double getErrorOfHeat() {
149
150           return erH;
151
152         };
153
154         long getRelTiOfHeat() {
155
156           return rtH;
157
158         };
159
160         double getAverageMag() {
161
162           return avM;
163
164         };
165
166         double getErrorOfMag() {
167
168           return erM;
169
170         };
171
172         long getRelTiOfMag() {
173
174           return rtM;
175
176         };
177
178         double getAverageSuscept() {
179
180           return avS;
181
182         };
183
184         double getErrorOfSuscept() {
185
186           return erS;
187
188         };
189
190         long getRelTiOfSuscept() {
191
192           return rtS;
```

```
193
194         };
195
196     };
197
198     #endif
```

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

A.3.2 ED

```
1     #ifndef CLASS_EDALGORITHM
2     #define CLASS_EDALGORITHM
3
4     #include <tnt/tnt_array2d.h>
5     #include <jama/jama_eig.h>
6
7     #include "AbstractAlgorithm.cpp"
8
9     class EDAlgorithm : public AbstractAlgorithm {
10
11      protected:
12
13        int twoPowN;
14        TNT::Array1D<double> *e;
15
16        void hAction(TNT::Array2D<double> hg, int s, TNT::Array1D<int> mapS) {
17
18          for(int b = 1; b <= lattice->getNb(); b++) {
19
20            bool i1Up = s & (int(1) << lattice->getI1(b));
21            bool i2Up = s & (int(1) << lattice->getI2(b));
22
23            hg[mapS[s]][mapS[s]] += i1Up == i2Up ? 0.25 : -0.25;
24
25            if(i1Up != i2Up) hg[mapS[s]][mapS[s ^ (int(1) << lattice->getI1(b)) ^ (int(1) << lattic
26
27          }
28
29        };
30
31        int getNumOf1Bits(int value) {
32
33          value = ((value & 0xaaaaaaaa) >> 1 ) + (value & 0x55555555);
34          value = ((value & 0xcccccccc) >> 2 ) + (value & 0x33333333);
35          value = ((value & 0xf0f0f0f0) >> 4 ) + (value & 0x0f0f0f0f);
36          value = ((value & 0xff00ff00) >> 8 ) + (value & 0x00ff00ff);
37          value = ((value & 0xffff0000) >> 16) + (value & 0x0000ffff);
38
39          return value;
40
41        };
42
43      public:
44
45        EDAlgorithm(AbstractLattice* lattice_parameter, int measureCount_parameter) : AbstractAlgor
46
47          twoPowN = int(1) << lattice->getN();
48
```

```
49          TNT::Array2D<int> mapSIndex(lattice->getN() + 1, twoPowN, 0); // 2nd dimension nee
50          TNT::Array1D<int> mapS(twoPowN, 0);
51          TNT::Array1D<int> sIndexLength(lattice->getN() + 1, 0);
52
53          e = new TNT::Array1D<double>(twoPowN, 0.0);
54          int eIndex = 0;
55
56          for(int s = 0; s < twoPowN; s++) {
57            int g = getNumOf1Bits(s);
58            mapSIndex[g][sIndexLength[g]] = s;
59            mapS[s] = sIndexLength[g];
60            sIndexLength[g]++;
61          }
62
63          for(int g = 0; g <= lattice->getN() / 2; g++) {
64
65            TNT::Array2D<double> hg(sIndexLength[g], sIndexLength[g], 0.0);
66            TNT::Array1D<double> eg(sIndexLength[g], 0.0);
67
68            for(int sIndex = 0; sIndex < sIndexLength[g]; sIndex++) {
69              hAction(hg, mapSIndex[g][sIndex], mapS);
70            }
71
72            std::cerr << "Diagonalize␣matrix␣" << (g + 1) << "␣von␣" << ((lattice->getN() /
73
74            JAMA::Eigenvalue<double> eFactory(hg);
75            eFactory.getRealEigenvalues(eg);
76
77            for(int sIndex = 0; sIndex < sIndexLength[g]; sIndex++) {
78
79              (*e)[eIndex] = eg[sIndex];
80              eIndex++;
81              if(lattice->getN() % 2 != 0 || g != lattice->getN() / 2) {
82                (*e)[eIndex] = eg[sIndex];
83                eIndex++;
84              }
85            }
86
87          }
88
89        };
90
91        ~EDAlgorithm() {
92
93          delete e;
94
95        };
96
97        void runTemperatureRound() {
98
99          double sumOfE = 0;
100         double sumOfESquared = 0;
101         double z = 0;
102
103         for(int s = 0; s < twoPowN; s++) {
104           double weight = exp(-(*e)[s] / t);
105           sumOfE += weight * (*e)[s];
106           sumOfESquared += weight * pow((*e)[s], 2);
107           z += weight;
108         }
109
110         avE = sumOfE / z;
```

```
111         avH = ((sumOfESquared / z) - pow(sumOfE / z, 2)) / pow(t, 2);
112         //avS = ((sumOfESquared / z) - pow(sumOfE / z, 2)) / t / lattice->getN();
113
114       };
115
116   };
117
118   #endif
```

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

A.3.3 ISING

```
1   #ifndef CLASS_ISINGALGORITHM
2   #define CLASS_ISINGALGORITHM
3
4   #include <gsl/gsl_rng.h>
5
6   #include "AbstractAlgorithm.cpp"
7   #include "../Analyzer/EAnalyzer.cpp"
8   #include "../Analyzer/HAnalyzer.cpp"
9   #include "../Analyzer/MAnalyzer.cpp"
10  #include "../Analyzer/SAnalyzer.cpp"
11
12  class ISINGAlgorithm : public AbstractAlgorithm {
13
14    protected:
15
16      const gsl_rng_type *generatorType;
17      gsl_rng *generator;
18
19      bool *spins;
20      double energy;
21      int spinSum;
22      bool start;
23
24      void doSweep() {
25
26        double energyDifference;
27        double weight;
28        double randomWeight;
29
30        for(int i = 0; i < lattice->getN(); i++) {
31
32          energyDifference = getEnergyDifferenceOfFlip(i);
33          weight = exp(-energyDifference / t);
34          randomWeight = gsl_rng_uniform(generator);
35
36          if(weight >= randomWeight) {
37            spinSum += getSpinSumDifferenceOfFlip(i);
38            energy  += energyDifference;
39            spins[i] = !spins[i];
40          }
41
42        }
43
44      };
45
46      double calculateEnergy() {
```

```
47
48          int sumOfBonds = 0;
49
50          for(int b = 1; b <= lattice->getNb(); b++) {
51            sumOfBonds += spins[lattice->getI1(b)] == spins[lattice->getI2(b)] ? 1 : -1;
52          }
53
54          return -sumOfBonds;
55
56        };
57
58        int calculateSpinSum() {
59
60          int spinSum = 0;
61
62          for(int i = 0; i < lattice->getN(); i++) {
63            spinSum += spins[i] ? 1 : -1;
64          }
65
66          return spinSum;
67
68        };
69
70        double getEnergyDifferenceOfFlip(int i) {
71
72          int *neighbours = lattice->getNeighbours(i);
73          double energyDiffernce = 0;
74
75          for(int j = 1; j <= neighbours[0]; j++) {
76            energyDiffernce += 2 * (spins[i] == spins[neighbours[j]] ? 1 : -1);
77          }
78
79          delete[] neighbours;
80
81          return energyDiffernce;
82
83        };
84
85        int getSpinSumDifferenceOfFlip(int i) {
86
87          return -2 * (spins[i] ? 1 : -1);
88
89        };
90
91      public:
92
93        ISINGAlgorithm(AbstractLattice* lattice_parameter, int measureCount_parameter) : Abs
94
95          gsl_rng_env_setup();
96          generatorType = gsl_rng_default;
97          generator = gsl_rng_alloc(generatorType);
98          gsl_rng_set(generator, time(NULL));
99
100         spins = new bool[lattice->getN()];
101         start = true;
102
103       };
104
105       ~ISINGAlgorithm() {
106
107         delete[] spins;
108
```

```
109           gsl_rng_free(generator);
110
111         };
112
113         void setTemperature(double t_parameter) {
114
115           AbstractAlgorithm::setTemperature(t_parameter);
116
117           if(start) {
118             for(int i = 0; i < lattice->getN(); i++) {
119               spins[i] = bool(gsl_rng_uniform_int(generator, 2));
120             }
121
122             energy  = calculateEnergy();
123             spinSum = calculateSpinSum();
124             start = false;
125           }
126
127         };
128
129         void runTemperatureRound() {
130
131           for(long i = 0; i < runCount; i++) {
132
133             if(i > runCount - measureCount) {
134               energyMeasurements[i - runCount + measureCount] = energy;
135               magMeasurements[i - runCount + measureCount]    = fabs(double(spinSum) / lattice->get
136             }
137
138             doSweep();
139
140           }
141
142           EAnalyzer *eAnalyzer = new EAnalyzer(this, lattice);
143           eAnalyzer->analyze();
144           avE = eAnalyzer->getAverage();
145           erE = eAnalyzer->getError();
146           rtE = eAnalyzer->getRelaxationTime();
147           delete eAnalyzer;
148
149           HAnalyzer *hAnalyzer = new HAnalyzer(this, lattice);
150           hAnalyzer->analyze();
151           avH = hAnalyzer->getAverage();
152           erH = hAnalyzer->getError();
153           rtH = hAnalyzer->getRelaxationTime();
154           delete hAnalyzer;
155
156           MAnalyzer *mAnalyzer = new MAnalyzer(this, lattice);
157           mAnalyzer->analyze();
158           avM = mAnalyzer->getAverage();
159           erM = mAnalyzer->getError();
160           rtM = mAnalyzer->getRelaxationTime();
161           delete mAnalyzer;
162
163           SAnalyzer *sAnalyzer = new SAnalyzer(this, lattice);
164           sAnalyzer->analyze();
165           avS = sAnalyzer->getAverage();
166           erS = sAnalyzer->getError();
167           rtS = sAnalyzer->getRelaxationTime();
168           delete sAnalyzer;
169
170         };
```

```
171
172   };
173
174   #endif
```

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

A.3.4 SSE

```
1   #ifndef CLASS_SSEALGORITHM
2   #define CLASS_SSEALGORITHM
3
4   #include <gsl/gsl_rng.h>
5
6   #include "AbstractAlgorithm.cpp"
7   #include "../Analyzer/EAnalyzer.cpp"
8   #include "../Analyzer/HAnalyzer.cpp"
9   #include "../Analyzer/MAnalyzer.cpp"
10  #include "../Analyzer/SAnalyzer.cpp"
11
12  class SSEAlgorithm : public AbstractAlgorithm {
13
14    protected:
15
16      const gsl_rng_type *generatorType;
17      gsl_rng *generator;
18
19      bool *spins;
20      long nr;
21      long lMax;
22      long l;
23      int *s;
24      long *x;
25
26      void doSweep() {
27
28        doDiagonalUpdate();
29
30        doOperatorLoopUpdate();
31
32        if(l < 4 * nr / 3) l = 4 * nr / 3;
33        if(l > lMax) throw "Reached␣lMax";
34
35      };
36
37      void doDiagonalUpdate() {
38
39        int b;
40
41        for(long p = 0; p < l; p++) {
42
43          if(s[p] == 0) { // No operator -> try to insert
44
45            b = gsl_rng_uniform_int(generator, lattice->getNb()) + 1;
46
47            if(spins[lattice->getI1(b)] == spins[lattice->getI2(b)]) continue; // if bond-
48
49            if(gsl_rng_uniform(generator) < ((double) lattice->getNb()) / 2 / (l - nr) / t
50              s[p] = 2 * b;
```

```
51              nr++;
52            }
53
54          } else if(s[p] % 2 == 0) { // Diagonal operator -> try to remove
55
56            if(gsl_rng_uniform(generator) < (double) 2 * (l - nr + 1) * t / lattice->getNb()) {
57              s[p] = 0;
58              nr--;
59            }
60
61          } else { // Off-Diagonal operator -> just flip the bond-neighbour spins
62
63            b = s[p] / 2;
64
65            spins[lattice->getI1(b)] = !spins[lattice->getI1(b)];
66            spins[lattice->getI2(b)] = !spins[lattice->getI2(b)];
67
68          }
69
70        }
71
72      };
73
74      void doOperatorLoopUpdate() {
75
76        // Construct link list x
77        long v0;
78        int b;
79        int i1;
80        int i2;
81
82        long *vFirst = new long[lattice->getN()];
83        for(int i = 0; i < lattice->getN(); i++) vFirst[i] = -1;
84        long *vLast = new long[lattice->getN()];
85        for(int i = 0; i < lattice->getN(); i++) vLast[i]  = -1;
86
87        for(long v = 0; v < 4 * l; v++) x[v] = -1;
88
89        for(long p = 0; p < l; p++) {
90
91          if(s[p] == 0) continue; // No operator -> go to next p
92
93          v0 = 4 * p;
94          b = s[p] / 2;
95          i1 = lattice->getI1(b);
96          i2 = lattice->getI2(b);
97
98          // Link the last 2-vertex on this spin to new 0-vertex
99          if(vLast[i1] == -1) {
100           vFirst[i1] = v0;
101         } else {
102           x[vLast[i1]] = v0;
103           x[v0] = vLast[i1];
104         }
105
106         // Link the last 3-vertex on this spin to new 1-vertex
107         if(vLast[i2] == -1) {
108           vFirst[i2] = v0 + 1;
109         } else {
110           x[vLast[i2]] = v0 + 1;
111           x[v0 + 1] = vLast[i2];
112         }
```

```
113
114             // Set the 2 and 3-vertex as last vertex on this spin
115             vLast[i1] = v0 + 2;
116             vLast[i2] = v0 + 3;
117
118           }
119
120           // Link vertexes periodicly if they interacted with operators
121           for(int i = 0; i < lattice->getN(); i++) {
122
123             if(vFirst[i] != -1) {
124               x[vFirst[i]] = vLast[i];
125               x[vLast[i]] = vFirst[i];
126             }
127
128           }
129
130           // Do the actual update
131           for(long v = 0; v < 4 * l; v += 2) {
132
133             if(x[v] < 0) continue;
134
135             long vT = v;
136             bool flipping = gsl_rng_uniform(generator) < 0.5;
137
138             while(x[vT] >= 0) {
139
140               // Flip
141               if(flipping) {
142                 long p = vT / 4;
143                 s[p] = s[p] ^ 1;
144               }
145
146               // Walk to other operator
147               vT = x[vT];
148
149               // Delete the way I went and the way back
150               x[vT] = x[x[vT]] = flipping ? -2 : -1;
151
152               // Walk to neighbour on operator
153               vT = vT ^ 1;
154
155             }
156
157           }
158
159           // Adjust spins
160           for(int i = 0; i < lattice->getN(); i++) {
161
162             if(vFirst[i] == -1) {
163               if(gsl_rng_uniform(generator) < 0.5) spins[i] = !spins[i];
164             } else {
165               if(x[vFirst[i]] == -2) spins[i] = !spins[i];
166             }
167
168           }
169
170           delete[] vLast;
171           delete[] vFirst;
172
173         };
174
```

```
175         int getSpinSum() {
176
177           int spinSum = 0;
178
179           for(int i = 0; i < lattice->getN(); i++) {
180             spinSum += spins[i] ? 0.5 : -0.5;
181           }
182
183           return spinSum;
184
185         };
186
187       public:
188
189         SSEAlgorithm(AbstractLattice* lattice_parameter, int measureCount_parameter) : AbstractAlgo
190
191           gsl_rng_env_setup();
192           generatorType = gsl_rng_default;
193           generator = gsl_rng_alloc(generatorType);
194           gsl_rng_set(generator, time(NULL));
195
196           spins = new bool[lattice->getN()];
197           for(int i = 0; i < lattice->getN(); i++) spins[i] = gsl_rng_uniform_int(generator, 2);
198           nr = 0;
199           lMax = 10000000;
200           l = 10;
201           s = new int[lMax];
202           for(long p = 0; p < lMax; p++) s[p] = 0;
203           x = new long[4 * lMax];
204
205         };
206
207         ~SSEAlgorithm() {
208
209           delete x;
210           delete s;
211           delete spins;
212
213           gsl_rng_free(generator);
214
215         };
216
217         void runTemperatureRound() {
218
219           for(long i = 0; i < runCount; i++) {
220
221             if(i > runCount - measureCount) {
222               energyMeasurements[i - runCount + measureCount] = -nr * t;
223               magMeasurements[i - runCount + measureCount]    = fabs(double(getSpinSum()) / lattice
224             }
225
226             doSweep();
227
228           }
229
230           EAnalyzer *eAnalyzer = new EAnalyzer(this, lattice);
231           eAnalyzer->analyze();
232           avE = eAnalyzer->getAverage();
233           erE = eAnalyzer->getError();
234           rtE = eAnalyzer->getRelaxationTime();
235           delete eAnalyzer;
236
```

```
237         HAnalyzer *hAnalyzer = new HAnalyzer(this, lattice);
238         hAnalyzer->analyze();
239         avH = hAnalyzer->getAverage();
240         erH = hAnalyzer->getError();
241         rtH = hAnalyzer->getRelaxationTime();
242         delete hAnalyzer;
243 
244         MAnalyzer *mAnalyzer = new MAnalyzer(this, lattice);
245         mAnalyzer->analyze();
246         avM = mAnalyzer->getAverage();
247         erM = mAnalyzer->getError();
248         rtM = mAnalyzer->getRelaxationTime();
249         delete mAnalyzer;
250 
251         SAnalyzer *sAnalyzer = new SAnalyzer(this, lattice);
252         sAnalyzer->analyze();
253         avS = sAnalyzer->getAverage();
254         erS = sAnalyzer->getError();
255         rtS = sAnalyzer->getRelaxationTime();
256         delete sAnalyzer;
257 
258       };
259 
260   };
261 
262   #endif
```

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

A.4 Analysemodule

A.4.1 Abstraktes Analysemodul

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

```
25
26        };
27
28        ~AbstractAnalyzer() {};
29
30        virtual const char* getQuantityName() = 0;
31        virtual double getQuantity(long time) = 0;
32
33        void analyze() {
34
35          double *sum        = new double[algorithm->getMeasureCount()];
36          double *sumSquared = new double[algorithm->getMeasureCount()];
37
38          for(long time = 0; time < algorithm->getMeasureCount(); time++) {
39            sum[time]        = (time == 0 ? 0 : sum[time - 1]) + getQuantity(time);
40            sumSquared[time] = (time == 0 ? 0 : sumSquared[time - 1]) + pow(getQuantity(time), 2);
41          }
42
43          for(relaxationTime = 0; relaxationTime < algorithm->getMeasureCount(); relaxationTime++)
44
45            double sumP = 0;
46
47            for(long time = 0; time < algorithm->getMeasureCount() - relaxationTime; time++) {
48              sumP += getQuantity(time) * getQuantity(time + relaxationTime);
49            }
50
51            long timeToAverage = algorithm->getMeasureCount() - relaxationTime;
52            double autoCorrelation = ((sumP / timeToAverage) - pow(sum[algorithm->getMeasureCount()
53
54            if(isnan(autoCorrelation)) {
55              printf("#␣%s␣at␣temperature=%+20.13e␣was␣not␣measured,␣due␣to␣the␣unknown␣relaxation␣
56              relaxationTime = -1;
57              break;
58            }
59
60            printf("#␣Auto-Correlation␣of␣%s=%+20.13e\n", getQuantityName(), autoCorrelation);
61
62            if(autoCorrelation < exp(-1)) break;
63
64          }
65
66          relaxationTime++;
67
68          if(relaxationTime != 0) {
69
70            long binsCount = algorithm->getMeasureCount() / 3 / relaxationTime;
71            double *binAverage = new double[binsCount];
72
73            for(long bin = 0; bin < binsCount; bin++) {
74              binAverage[bin] = (sum[(bin + 1) * 3 * relaxationTime] - sum[bin * 3 * relaxationTime
75              average += binAverage[bin] / binsCount;
76            }
77
78            double deviationSum = 0;
79            for(long bin = 0; bin < binsCount; bin++) {
80              deviationSum += pow(binAverage[bin] - average, 2);
81            }
82            error = sqrt(deviationSum / binsCount / (binsCount - 1));
83
84            delete[] binAverage;
85
86          }
```

```
87
88          delete[] sum;
89          delete[] sumSquared;
90
91        };
92
93        double getAverage() {
94
95          return average;
96
97        };
98
99        double getError() {
100
101         return error;
102
103       };
104
105       long getRelaxationTime() {
106
107         return relaxationTime;
108
109       };
110
111   };
112
113   #endif
```

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

A.4.2 Analysemodul für die Energie

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

```
28  #endif
```

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

A.4.3 Analysemodul für die Spezifische Wärme

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

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

A.4.4 Analysemodul für die Magnetisierung pro Spin

```
1   #ifndef CLASS_MANALYZER
2   #define CLASS_MANALYZER
3
4   #include "AbstractAnalyzer.cpp"
5
6   class MAnalyzer : public AbstractAnalyzer {
7
8     public:
9
10      MAnalyzer(AbstractAlgorithm* algorithm_parameter, AbstractLattice* lattice_parameter) : Abs
11
12      ~MAnalyzer() {};
13
14      const char* getQuantityName() {
15
16        return "Magnetization";
```

```
17
18          };
19
20          double getQuantity(long time) {
21
22            return algorithm->getMagMeasurement(time);
23
24          };
25
26      };
27
28      #endif
```

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

A.4.5 Analysemodul für die Magnetische Suszeptibilität

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

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

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Literaturverzeichnis

[Knu02] Donald E. Knuth. Der perfektionist. *heise.de - c't*, page 190, May 2002.

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

[Met87] Nicholas Metropolis. The beginning of the monte carlo method. *Los Alamos Science*, 15:125–130, 1987.

[MRR+53] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equations of state calculations by fast computing machines. *The Journal Of Chemical Physics*, 21(6):1087–1092, 1953.