Statistische Mechanik Bonusblatt

Jun Wei Tan*

Julius-Maximilians-Universität Würzburg

(Dated: 23. Dezember 2024)

CONTENTS

I. AUFGABE 1

Betrachten Sie eine Kette aus N Monomeren der Größe a in zwei Dimensionen. D.h., jedem Glied i kann der Vektor $\vec{r}_i$ zugeordnet werden, der in eine beliebige Richtung zeigen kann und die Länge $|\vec{r}_i| = a$ hat.

a) Berechnen Sie für dieses System die mittlere quadratische Distanz zwischen Anfangs- und Endpunkt:

$$\langle \vec{R}^2 \rangle = \left\langle \left(\sum_{i=1}^{N} \vec{r}_i \right)^2 \right\rangle$$

Hinweis: Hier lässt sich mit Symmetrie argumentieren. Was ist $\langle \vec{r}_i \cdot \vec{r}_j \rangle$ für $i = j$ bzw. für $i \neq j$?

* jun-wei.tan@stud-mail.uni-wuerzburg.de

b) Das Polymer befindet sich jetzt in einem Wärmebad mit der Temperatur T. Die beiden Enden des Polymers werden mit einer Kraft $\vec{F} = (0, 0, F)$ auseinandergezogen, so dass (1) gilt. Berechnen Sie die Korrelationsfunktionen

$$\langle \vec{R} \rangle = \left\langle \sum_{i=1}^{N} \vec{r}_i \right\rangle,$$

$$\langle \vec{R}^2 \rangle = \left\langle \left(\sum_{i=1}^{N} \vec{r}_i \right)^2 \right\rangle$$

als Funktion der Kraft und der Temperatur. Wie verhalten sich diese in den Grenzfällen $aF \ll k_B T$ und $aF \gg k_B T$?

Beweis. a) Für $i \neq j$ sind $\vec{r}_i$ und $\vec{r}_j$ unabhängig. Damit ist

$$\langle \vec{r}_i \cdot \vec{r}_j \rangle = \langle \vec{r}_i \rangle \cdot \langle \vec{r}_j \rangle = \vec{0} \cdot \vec{0} = 0.$$

Für $i = j$ ist $\vec{r}_i \cdot \vec{r}_j = |\vec{r}_i|^2 = a^2$, was überhaupt nicht stochastisch ist und damit ist $\langle \vec{r}_i \cdot \vec{r}_i \rangle = a^2$. Damit ist

$$\begin{aligned}
\langle \vec{R}^2 \rangle &= \left\langle \left(\sum_{i=1}^{N} \vec{r}_i \right)^2 \right\rangle \\
&= \left\langle \sum_{i=1}^{N} \sum_{j=1}^{N} \vec{r}_i \cdot \vec{r}_j \right\rangle \\
&= \sum_{i=1}^{N} \sum_{j=1}^{N} \langle \vec{r}_i \cdot \vec{r}_j \rangle \\
&= \sum_{i=1}^{N} \sum_{j=1}^{N} a^2 \delta_{ij} \\
&= N a^2.
\end{aligned}$$

Die kanonische Zustandssumme ist

$$Z = \int_{S^2} \cdots \int_{S^2} \exp(-\beta H) \, \mathrm{d}\vec{r}_i \ldots \mathrm{d}\vec{r}_n$$

wobei S^2 der Einheitskreis in 3-Dim und

$$H = -\sum_{i=1}^{n} \vec{F} \cdot \vec{r}_i$$

ist. Damit gilt

$$Z = \int_{S^2} \cdots \int_{S^2} \prod_{k=1}^{n} e^{\beta \vec{F} \cdot \vec{r}_k} \, \mathrm{d}\vec{r}_1 \ldots \mathrm{d}\vec{r}_n \, .$$

Ein inneres Integral ist

$$\begin{aligned}
\int_{S^2} e^{\beta(\vec{F} \cdot \vec{r})} \, \mathrm{d}\vec{r} &= \int_0^{2\pi} \int_0^{\pi} e^{\beta F a \cos\theta} \sin\theta \, \mathrm{d}\theta \, \mathrm{d}\varphi \\
&= -2\pi \int_1^{-1} e^{\beta F a u} \, \mathrm{d}u \\
&= \frac{4\pi \sinh(aF\beta)}{aF\beta}
\end{aligned}$$

und damit ist

$$Z = \left[\frac{4\pi \sinh(aF\beta)}{aF\beta} \right]^N .$$

Nun bestimmen wir die 3 Komponente des Vektors $\langle \vec{R} \rangle$ $\langle R_i \rangle$. Es gilt

$$\begin{aligned}
\langle R_i \rangle &= \frac{1}{Z} \int_{(S^2)^n} R_i \exp\left(\beta \sum_{k=1}^{n} \sum_{p \in \{x,y,z\}} F_p (r_k)_p \right) \mathrm{d}\vec{r}_1 \ldots \mathrm{d}\vec{r}_n \\
&= \frac{1}{Z} \int_{(S^2)^n} \left(\sum_{l=1}^{n} (r_l)_i \right) \exp\left(\beta \sum_{k=1}^{n} \sum_{p \in \{x,y,z\}} F_p (r_k)_p \right) \mathrm{d}\vec{r}_1 \ldots \mathrm{d}\vec{r}_n \\
&= \frac{1}{\beta} \frac{1}{Z} \int_{(S^2)^n} \frac{\partial}{\partial F_i} \exp\left(\beta \sum_{k=1}^{n} \sum_{p \in \{x,y,z\}} F_p (r_k)_p \right) \mathrm{d}\vec{r}_1 \ldots \mathrm{d}\vec{r}_n \\
&= \frac{1}{\beta} \frac{\partial}{\partial F_i} \ln Z
\end{aligned}$$

In diesem Fall hat F nur eine z-Komponente. Damit verschwinden die x und y Ableitungen:

$$\langle R_x \rangle = \langle R_y \rangle = 0.$$

Die z-Komponente ist

$$\begin{aligned}
\langle R_z \rangle &= \frac{1}{\beta} \frac{\partial}{\partial F} N \ln \frac{4\pi \sinh(aF\beta)}{aF\beta} \\
&= \frac{N}{\beta} \left[a\beta \coth(a\beta F) - \frac{1}{F} \right] \\
&= Na \coth(a\beta F) - \frac{N}{\beta F}.
\end{aligned}$$

Zur Bestimmung von $\langle R^2 \rangle$ betrachten wir

$$\langle R^2 \rangle = \frac{1}{Z} \int_{(S^2)^N} R^2 \exp\left(\beta \sum_{k=1}^{N} \sum_{p \in \{x,y,z\}} F_p (r_k)_p \right) \mathrm{d}\vec{r}_1 \ldots \mathrm{d}\vec{r}_N$$

$$= \frac{1}{Z} \int_{(S^2)^N} \sum_{i\in\{x,y,z\}} \left(\sum_{l=1}^{N} (r_l)_i\right)^2 \exp\left(\beta \sum_{k=1}^{N} \sum_{p\in\{x,y,z\}} F_p (r_k)_p\right) \mathrm{d}\vec{r}_1 \ldots \mathrm{d}\vec{r}_N$$
$$= \frac{1}{Z} \int_{(S^2)^N} \sum_{i\in\{x,y,z\}} \frac{1}{\beta^2} \frac{\partial^2}{\partial F_i^2} \exp\left(\beta \sum_{k=1}^{N} \sum_{p\in\{x,y,z\}} F_p (r_k)_p\right) \mathrm{d}\vec{r}_1 \ldots \mathrm{d}\vec{r}_N$$
$$= \frac{1}{Z} \frac{1}{\beta^2} \sum_{i\in\{x,y,z\}} \frac{\partial^2 Z}{\partial F_i^2}.$$

Die Ableitungen nach F_x und F_y verschwinden wieder und damit bleibt nur

$$\begin{aligned} \langle R^2 \rangle &= \frac{1}{\beta^2 Z} \frac{\partial^2 Z}{\partial F^2} \\ &= \frac{N(4\pi)^N \left(a^2\beta^2F^2 + a^2\beta^2F^2(N-1)\coth^2(a\beta F) - 2a\beta F N \coth(a\beta F) + N + 1\right)}{F^2\beta^2}. \square \end{aligned}$$

II. AUFGABE 2

Die Codes befinden sich im Anhang A. Die sind für den Random Walk und den Self Avoiding Random Walk zwar sehr ähnlich, können aber nicht vertauscht werden, da die Methode `generateWalk()` unterschiedlich implementiert wird. Beim SARW unterbricht die Methode, sobald die Trajektorie sich schneidet. Damit spart das Programm $O(1)$ Zeit, was die Zeitkomplexität jedoch nicht ändert. Mit der Flagge -O3 läuft das Programm für den RW in 5.146s und das Programm für den SARW in 141.988s.

Abbildung 1. Ich, nachdem mein Code 0.01% schneller läuft.

20 Trajektorien für den RW und den SARW sind in Abb. 2 dargestellt. Ein Best-Fit-Gerade wurde zu den Daten im Anhang B bestimmt. Für den Random-Walk ist es aus den Daten klar, dass eine lineare Anpassung sehr gut geeignet ist. Damit führen wir eine lineare

Regression durch. Für den SARW führen wir stattdessen eine lineare Regression in log-log Skala bzw. eine polynomiale Anpassung durch.

Damit ist die lineare Anpassung zum RW (geplottet in Abb. 3(a)):

$$y = a + bx$$

mit Parameter

$$a = 0.052 \pm 0.010$$

$$b = 1.000241 \pm 0.000088$$

Weil der RW noch symmetrisch ist, können wir dasselbe Argument wie in Aufgabe 1a anwende. Es ist also zu erwarten, dass $\langle R^2(t) \rangle = t$. Dies stimmt nicht im Rahmen des Fehlers mit der Regressionsgerade überein. Es liegt wahrscheinlich daran, dass die $\langle R(t)^2 \rangle$ zu unterschiedlichen Zeitpunkten korreliert sind. Dies sieht man in Abb. 3(b). Wenn die Werte unabhängig wären, wäre die Streuung symmetrisch oberhalb und unterhalb der $x = y$ Gerade. Stattdessen sieht man eine kontinuierliche Kurve. Die Korrelation sieht man durch ein Beispiel: Wenn eine Kette weiter weg vom Ursprung als erwartet wandert, werden die folgenden Abstände auch größer als erwartet. Damit führt eine kleine statistische Abweichung am Anfang zu einer fortgesetzten Abweichung in der gleichen Richtung, was die Parameter der Regression ändern kann.

Für den SARW machen wir stattdessen eine polynomiale Anpassung (Abb. 3(c))

$$y = Ax^b$$

mit Parameter

$$A = 0.9820 \pm 0.0059$$

$$b = 1.4603 \pm 0.0024$$

Es ist auffällig, dass die letzten 3 Punkte nicht an der Gerade liegen. Es liegt vermutlich daran, dass die Verteilung von der Länge abhängig ist. Es kann sein, dass manche Pfade sich verschließen. Dann wäre dieser Pfad abgelehnt, wenn er fortgesetzt wäre. Daher gibt es auch den Begriff des Non-Trapped Self Avoiding Walks. Da unser Programm bei einer endlichen Anzahl von Schritte haltet, sind diese Pfade aber nicht abgelehnt. Dies führt zu einer geringen Abstand vom Ursprung als erwartet, was wir im Ergebnis sehen.

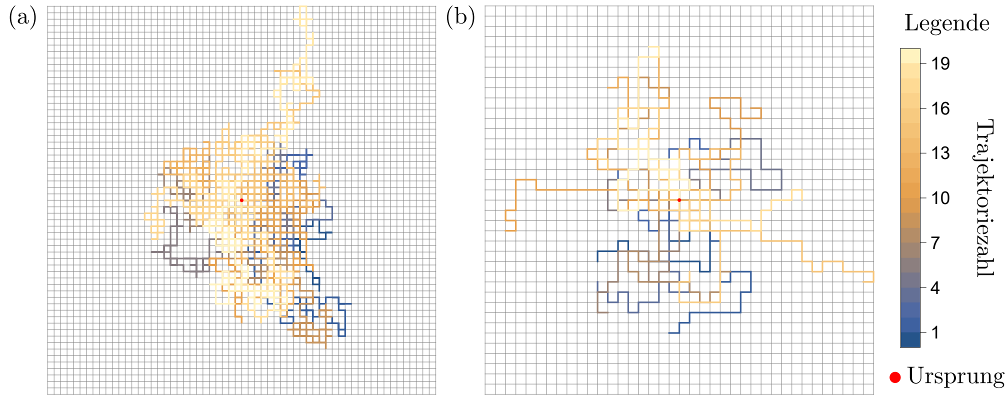

Abbildung 2. 20 Trajektorien sind dargestellt. **(a)** Trajektorien des Random Walks mit Länge 200. **(b)** Trajektorien des Self Avoiding Random Walks mit Länge 30.

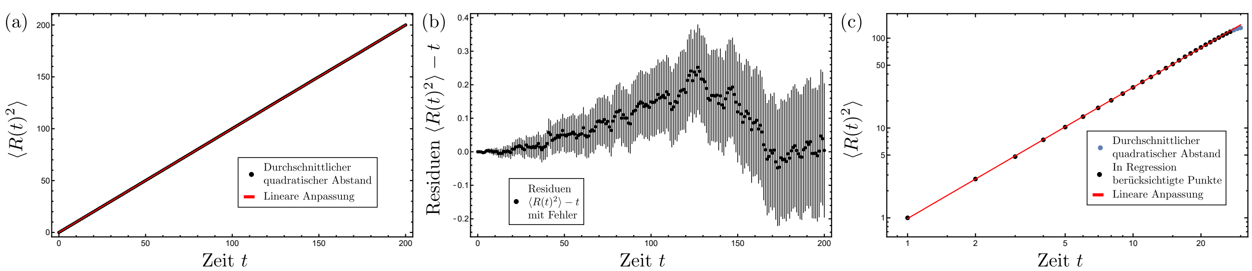

Abbildung 3. **(a)** Durchschnittlicher quadratischer Abstand des RWs vom Ursprung in Abhängigkeit von der Zeit bzw. Anzahl der Schritte. Die lineare Anpassung $y = 0.052 + 1.000241x$ stimmt sehr gut mit den Daten überein. **(b)** Residuen bezüglich der Gerade $y = x$. Die Residuen zu unterschiedlichen Zeitpunkten sind korreliert. **(c)** Durchschnittlicher quadratischer Abstand des SARWs in Abhängigkeit der Zeit in log-log Skala. Eine Gerade wird zu den ersten 27 Punkten angepasst.

Der Abstand vom Ursprung steigt schneller für den SARW als für den RW. Weil die Pfade nicht selbst schneiden können, gibt es immer ein Bereich um Ursprung, der „verboten“ ist. Damit ist die Wahrscheinlichkeit größer, dass der Pfad weg vom Ursprung wandert.

Anhang A: Code

1. Code für den Random Walk

```
#include <cstdio>
#include <vector>
#include <random>
#include <math.h>
#include <fstream>
#include <iostream>

inline int sqDist(std::pair<int,int> coord){
    return coord.first*coord.first + coord.second*coord.second;
}

class RandomWalk {
    public:
        int N; //Length
        std::vector<std::pair<int,int>> coords; //list of positions
        std::pair<int,int> currentPos;

        //random device
        static std::random_device rd;
        /* This may not be a random seed!
         * https://en.cppreference.com/w/cpp/numeric/random/random_device
         * On my computer, it generated the same numbers at every run
         * Maybe it works better on yours lmao
         */
        static std::mt19937 gen; //static so different class instances generate different numbers
          static std::discrete_distribution<> dist;

        RandomWalk(int Np){
                this -> N = Np;
```

```
    }

    int generateWalk(){
    //reset variables
    currentPos = std::make_pair(0,0);
    coords = std::vector<std::pair<int,int>>();
    coords.push_back(currentPos);

    for (int i = 0; i < N; ++i){
        int dir = dist(gen); //0 for right, 1 for up, 2 for left, 3 for down
        switch (dir){
            case 0:
                currentPos.first++;
                break;
            case 1:
                currentPos.second++;
                break;
            case 2:
                currentPos.first--;
                break;
            case 3:
                currentPos.second--;
        }
        coords.push_back(currentPos);
    }
    return 0; //no errors; this will be different in the SARW
    }
};

```

```
//Initialize static random members
std::random_device RandomWalk::rd;
std::mt19937 RandomWalk::gen(rd());
std::discrete_distribution<> RandomWalk::dist({1,1,1,1});


int main(){
   int numExperiments = 1000000;
   int L = 200; //length of a single walk
   std::vector<std::vector<int>> distances(numExperiments, std::vector<int>(L+1,0));
   RandomWalk rw = RandomWalk(L);
   //we export the first 20 trajectories here, 20 is hardcoded
   std::ofstream outfile;
   outfile.open("trajectories.csv");
   for (int exp = 0; exp < numExperiments; ++exp){
      rw.generateWalk();
      for (int i = 1; i <= L; ++i) distances[exp][i] = sqDist(rw.coords[i]);
      if (exp < 20) for (int i = 1; i <= L; ++i) outfile << rw.coords[i].first << "," << rw.coords[i].second << "\n";
   }
   outfile.close();

   /*for (int exp = 0; exp < numExperiments; ++exp){
for (int i = 0; i <= L; ++i) printf("%d ", distances[exp][i]); printf("\n");
   }*/

   /* Be careful when taking the average/stddev for large enough numbers
    * Because we compute the sum before dividing, we may have an integer overflow
```

```
     * To make this as high as possible, we use a long double
     */

    std::vector<double> averageDistances(L+1,0);
    std::vector<double> standardDeviations(L+1,0);
    for (int i = 1; i <= L; ++i){
        long double sum = 0.0;
        for (int exp = 0; exp < numExperiments; ++exp) sum += distances[exp][i];
        averageDistances[i] = sum / numExperiments;
        sum = 0.0;
        for (int exp = 0; exp < numExperiments; ++exp) sum += pow(distances[exp][i] - averageDistances[i], 2);
        sum /= numExperiments - 1;
        sum = sqrt(sum);
        standardDeviations[i] = sum;
    }

    outfile.open("randomWalk.csv");
    for (int i = 0; i <= L; ++i) outfile << averageDistances[i] << ",";
    outfile << "\n";
    for (int i = 0; i <= L; ++i) outfile << standardDeviations[i] << ",";
    outfile.close();
        return 0;
}
```

2. Code für den Self Avoiding Random Walk

```
#include <cstdio>
```

```
#include <vector>
#include <random>
#include <math.h>
#include <fstream>
#include <iostream>
#include <set>

inline int sqDist(std::pair<int,int> coord){
    return coord.first*coord.first + coord.second*coord.second;
}

class RandomWalk {
    public:
        int N; //Length
        std::vector<std::pair<int,int>> coords; //list of positions
        std::pair<int,int> currentPos;
        std::set<std::pair<int,int>> visited;

        //random device
        static std::random_device rd;
        /* This may not be a random seed!
         * https://en.cppreference.com/w/cpp/numeric/random/random_device
         * On my computer, it generated the same numbers at every run
         * Maybe it works better on yours lmao
         */
        static std::mt19937 gen; //static so different class instances generate different numbers
          static std::discrete_distribution<> dist;

        RandomWalk(int Np){
```

```
            this -> N = Np;
    }

    int generateWalk(){
    //reset variables
    currentPos = std::make_pair(0,0);
    coords = std::vector<std::pair<int,int>>();
    visited = std::set<std::pair<int,int>>();

    coords.push_back(currentPos);
    visited.insert(currentPos);

    for (int i = 0; i < N; ++i){
        int dir = dist(gen); //0 for right, 1 for up, 2 for left, 3 for down
        switch (dir){
            case 0:
                currentPos.first++;
                break;
            case 1:
                currentPos.second++;
                break;
            case 2:
                currentPos.first--;
                break;
            case 3:
                currentPos.second--;
        }
        coords.push_back(currentPos);
        if (visited.count(currentPos)) return 1;
```

```
            visited.insert(currentPos);
        }
        return 0;
        }

        int generateAvoidingWalk(){
        while (true){
            int x = generateWalk();
            if (not x) return 0;
        }
        }
};

//Initialize static random members
std::random_device RandomWalk::rd;
std::mt19937 RandomWalk::gen(rd());
std::discrete_distribution<> RandomWalk::dist({1,1,1,1});


int main(){
    int numExperiments = 10000;
    int L = 30; //length of a single walk
    std::vector<std::vector<int>> distances(numExperiments, std::vector<int>(L+1,0));
    RandomWalk rw = RandomWalk(L);
    //here we also export the first 20 walks, 20 is hardcoded
    std::ofstream outfile;
    outfile.open("trajectoriesAvoiding.csv");
    for (int exp = 0; exp < numExperiments; ++exp){
        rw.generateAvoidingWalk();
```

```
        for (int i = 1; i <= L; ++i) distances[exp][i] = sqDist(rw.coords[i]);
        if (exp < 20) for (int i = 1; i <= L; ++i) outfile << rw.coords[i].first << "," << rw.coords[i].second << "\n"; //here is the exporting of the trajectory
    }
    outfile.close();

    /*for (int exp = 0; exp < numExperiments; ++exp){
for (int i = 0; i <= L; ++i) printf("%d ", distances[exp][i]); printf("\n");
    }*/

    /* Be careful when taking the average/stddev for large enough numbers
     * Because we compute the sum before dividing, we may have an integer overflow
     * To make this as high as possible, we use a long double
     */

    std::vector<double> averageDistances(L+1,0);
    std::vector<double> standardDeviations(L+1,0);
    for (int i = 1; i <= L; ++i){
        long double sum = 0.0;
        for (int exp = 0; exp < numExperiments; ++exp) sum += distances[exp][i];
        averageDistances[i] = sum / numExperiments;
        sum = 0.0;
        for (int exp = 0; exp < numExperiments; ++exp) sum += pow(distances[exp][i] - averageDistances[i], 2);
        sum /= numExperiments - 1;
        sum = sqrt(sum);
        standardDeviations[i] = sum;
```

```
    }

    outfile.open("randomAvoidingWalk.csv");
    for (int i = 0; i <= L; ++i) outfile << averageDistances[i] << ",";
    outfile << "\n";
    for (int i = 0; i <= L; ++i) outfile << standardDeviations[i] << ",";
    outfile.close();
        return 0;
}
```

Anhang B: Simulationsdaten

1. Daten für den Random Walk

Zeit (t)	$\langle R(t)^2 \rangle$
0	0 ± 0
1	1 ± 0
2	1.9995 ± 0.0014
3	2.9973 ± 0.0024
4	3.9984 ± 0.0035
5	5.0028 ± 0.0045
6	6.0034 ± 0.0055
7	7.0019 ± 0.0065
8	8.0022 ± 0.0075
9	8.9991 ± 0.0085
10	10.0009 ± 0.0095
11	11.001 ± 0.010

12	11.995 ± 0.011
13	12.996 ± 0.012
14	13.995 ± 0.013
15	15.001 ± 0.014
16	16.000 ± 0.015
17	17.002 ± 0.016
18	18.001 ± 0.017
19	18.998 ± 0.019
20	20.008 ± 0.020
21	21.008 ± 0.021
22	22.015 ± 0.022
23	23.015 ± 0.023
24	24.018 ± 0.024
25	25.015 ± 0.025
26	26.009 ± 0.026
27	27.006 ± 0.027
28	28.020 ± 0.028
29	29.015 ± 0.029
30	30.014 ± 0.030
31	31.012 ± 0.030
32	32.016 ± 0.031
33	33.012 ± 0.032
34	34.017 ± 0.033

35	35.025 ± 0.034
36	36.014 ± 0.035
37	37.006 ± 0.036
38	38.013 ± 0.037
39	39.016 ± 0.038
40	40.025 ± 0.039
41	41.057 ± 0.041
42	42.051 ± 0.042
43	43.034 ± 0.043
44	44.038 ± 0.044
45	45.041 ± 0.045
46	46.049 ± 0.046
47	47.043 ± 0.047
48	48.050 ± 0.048
49	49.063 ± 0.049
50	50.050 ± 0.050
51	51.047 ± 0.051
52	52.051 ± 0.052
53	53.052 ± 0.053
54	54.054 ± 0.054
55	55.059 ± 0.055
56	56.053 ± 0.056
57	57.041 ± 0.057
58	58.049 ± 0.058

59	59.052 ± 0.059
60	60.054 ± 0.060
61	61.072 ± 0.061
62	62.049 ± 0.062
63	63.044 ± 0.063
64	64.047 ± 0.064
65	65.061 ± 0.065
66	66.065 ± 0.066
67	67.062 ± 0.067
68	68.075 ± 0.068
69	69.090 ± 0.069
70	70.092 ± 0.070
71	71.089 ± 0.071
72	72.089 ± 0.072
73	73.096 ± 0.073
74	74.084 ± 0.074
75	75.091 ± 0.075
76	76.074 ± 0.076
77	77.064 ± 0.077
78	78.091 ± 0.078
79	79.105 ± 0.079
80	80.119 ± 0.080
81	81.120 ± 0.081
82	82.124 ± 0.082

83	83.100 ± 0.083
84	84.111 ± 0.084
85	85.100 ± 0.085
86	86.100 ± 0.086
87	87.086 ± 0.087
88	88.083 ± 0.088
89	89.113 ± 0.089
90	90.121 ± 0.090
91	91.124 ± 0.091
92	92.135 ± 0.092
93	93.143 ± 0.093
94	94.149 ± 0.094
95	95.145 ± 0.095
96	96.135 ± 0.096
97	97.141 ± 0.097
98	98.137 ± 0.098
99	99.132 ± 0.099
100	100.149 ± 0.10
101	101.15 ± 0.10
102	102.15 ± 0.10
103	103.15 ± 0.10
104	104.17 ± 0.10
105	105.16 ± 0.10
106	106.16 ± 0.11

107	107.17 ± 0.11
108	108.16 ± 0.11
109	109.18 ± 0.11
110	110.17 ± 0.11
111	111.14 ± 0.11
112	112.13 ± 0.11
113	113.13 ± 0.11
114	114.15 ± 0.11
115	115.15 ± 0.11
116	116.17 ± 0.12
117	117.18 ± 0.12
118	118.18 ± 0.12
119	119.19 ± 0.12
120	120.18 ± 0.12
121	121.21 ± 0.12
122	122.23 ± 0.12
123	123.24 ± 0.12
124	124.21 ± 0.12
125	125.24 ± 0.12
126	126.23 ± 0.13
127	127.25 ± 0.13
128	128.24 ± 0.13
129	129.22 ± 0.13
130	130.22 ± 0.13

131	131.21 ± 0.13
132	132.19 ± 0.13
133	133.16 ± 0.13
134	134.15 ± 0.13
135	135.17 ± 0.13
136	136.17 ± 0.14
137	137.17 ± 0.14
138	138.17 ± 0.14
139	139.14 ± 0.14
140	140.17 ± 0.14
141	141.14 ± 0.14
142	142.13 ± 0.14
143	143.16 ± 0.14
144	144.18 ± 0.14
145	145.19 ± 0.14
146	146.19 ± 0.15
147	147.19 ± 0.15
148	148.17 ± 0.15
149	149.15 ± 0.15
150	150.12 ± 0.15
151	151.14 ± 0.15
152	152.11 ± 0.15
153	153.09 ± 0.15
154	154.10 ± 0.15

155	155.12 ± 0.15
156	156.11 ± 0.16
157	157.10 ± 0.16
158	158.07 ± 0.16
159	159.09 ± 0.16
160	160.09 ± 0.16
161	161.09 ± 0.16
162	162.07 ± 0.16
163	163.08 ± 0.16
164	164.06 ± 0.16
165	165.02 ± 0.16
166	166.01 ± 0.17
167	166.98 ± 0.17
168	167.99 ± 0.17
169	169.00 ± 0.17
170	170.00 ± 0.17
171	171.00 ± 0.17
172	171.97 ± 0.17
173	172.95 ± 0.17
174	173.95 ± 0.17
175	174.98 ± 0.17
176	175.98 ± 0.18
177	176.98 ± 0.18
178	177.97 ± 0.18

179	178.99 ± 0.18
180	179.98 ± 0.18
181	181.00 ± 0.18
182	182.01 ± 0.18
183	183.02 ± 0.18
184	184.00 ± 0.18
185	184.99 ± 0.18
186	185.99 ± 0.19
187	186.98 ± 0.19
188	188.01 ± 0.19
189	189.01 ± 0.19
190	190.03 ± 0.19
191	191.02 ± 0.19
192	192.03 ± 0.19
193	193.00 ± 0.19
194	194.00 ± 0.19
195	195.00 ± 0.19
196	195.99 ± 0.20
197	197.00 ± 0.20
198	198.05 ± 0.20
199	199.04 ± 0.20
200	200.00 ± 0.20

2. Daten für den Self Avoiding Random Walk

Zeit (t)	$\langle R(t)^2 \rangle$
0	0 ± 0
1	1 ± 0
2	2.7046 ± 0.0096
3	4.798 ± 0.022
4	7.388 ± 0.035
5	10.232 ± 0.051
6	13.376 ± 0.069
7	16.752 ± 0.088
8	20.33 ± 0.11
9	24.12 ± 0.13
10	28.30 ± 0.16
11	32.61 ± 0.18
12	36.96 ± 0.21
13	41.62 ± 0.23
14	46.45 ± 0.26
15	51.49 ± 0.29
16	56.75 ± 0.33
17	62.18 ± 0.36
18	67.72 ± 0.39
19	73.24 ± 0.42
20	78.87 ± 0.46
21	84.67 ± 0.49
22	90.22 ± 0.53

23	95.88 ± 0.56
24	101.55 ± 0.60
25	107.18 ± 0.64
26	112.70 ± 0.68
27	118.05 ± 0.72
28	122.96 ± 0.76
29	127.14 ± 0.80
30	130.25 ± 0.84