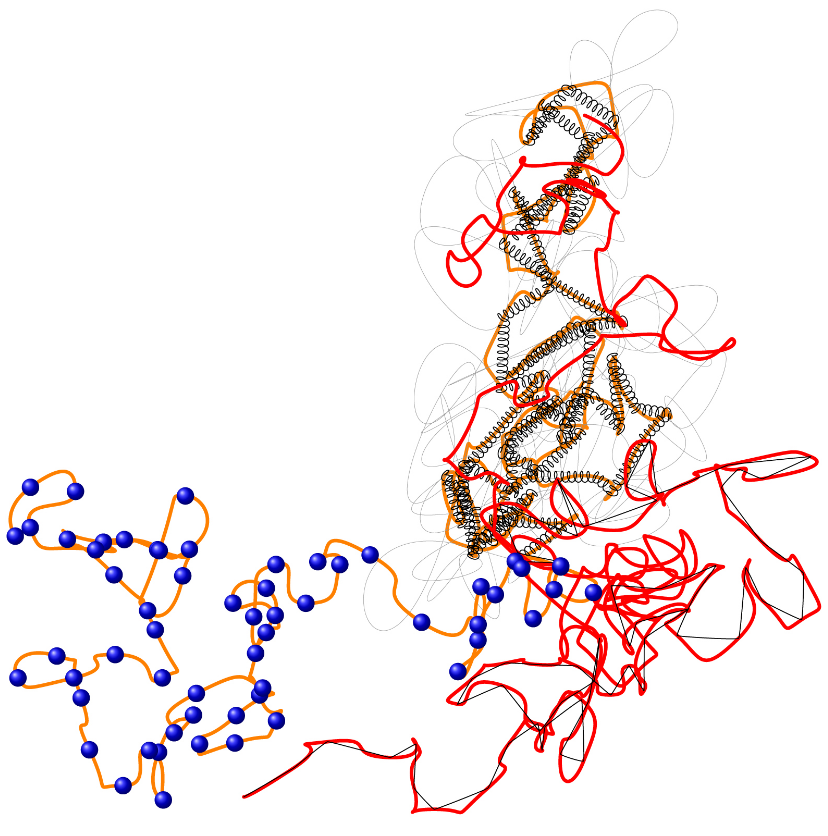
Suppressed Finite-Size Corrections in Self Avoiding Walks

Q.E. Krol

Master Thesis, June 2013

Supervisor: Prof. Gerard T. Barkema, Utrecht University



Abstract

Polymers are simulated with a bead-spring model, with a short-ranged repulsive interaction between the beads. For a large number N of beads, the mean end-to-end length, $\left\langle \vec{r}_e^2(N) \right\rangle$ scales as $N^{2\nu}$, For a finite number beads, the range of our simulations, there are corrections to this scaling law of the form $N^{2\nu_{\text{eff}}} = N^{2\nu}(a_0 + a_1/N^{\Delta})$. Determining the growth exponent by simulations is troubled by these corrections. In this thesis simulations with various interactions between the beads are done in order to minimize these finite-size corrections. This is done by tuning the parameters of the potentials such that the effective growth exponent converges towards a region of $[\nu - 0.02, \nu + 0.02]$. The potentials that are used are the hard sphere, Lennard Jones, and the Gaussian potential. The simulations are done in both two and three dimensions. The hard sphere potential, with diameters $\sigma_{2D} = 2.86$ and $\sigma_{3D} = 2.05$, was the best potential according to our criterion. For the Gaussian potential we found parameters that give approximately the same results as the hard sphere potential in both two and three dimensions, but it did not significantly improve upon them.

Contents

1 Introduction

In daily life we are used to a variety of substances that consist of very long molecules with high molar masses, such as plastics, PVC, proteins, DNA, etc. These so-called macromolecules consist of a chain of many repeating monomers connected by flexible bonds. Usually the number of monomers is very large and lies in range of $10^4 - 10^5$. These polymers occur naturally but are also produced industrially by polymerization. In this thesis unbranched polymers are considered. The key property of these polymers is their size. In a solution with a high density of polymers the size of their end-to-end length scales with the square root of the number of monomers, just like brownian motion. The random walk model is often used to describe their size by its mean squared displacement.

A physical property of the monomers is that they cannot overlap. In strong diluted solutions, where polymers can be singled out, this property results in a swelling of the polymer's size. For very large polymers the squared end-to-end length scales as $N^{2\nu}$, with N the number of monomers and ν the growth exponent. For smaller polymers (i.e. finite size) there are corrections to this scaling law. To describe these polymers on a physical level, the self-avoiding walk (SAW) model is used. Both the random walk and the SAW model will be treated in the second section of this thesis.

The growth exponent of the size of a single polymer, or the mean squared displacement of the self-avoiding walk, is of great interest because its value is universal. This means that it does not depend on the microscopic details of our model but only on the dimension the self-avoiding walk lives in. However, the finite-size corrections do depend on the microscopic details of the model. To determine the value of ν exactly one needs simulations of (infinitely) large self-avoiding walks, but this is practically impossible. The most extensive simulations still suffer from the finite-size corrections to this scaling law. Therefore it is important to get insight in these finite-size corrections and by tuning the microscopic details of the model to minimize them.

For simulations of the SAWs on a cubic lattice this has been done by Grassberger [1]. He used the Domb Joyce model to tune the leading correction to zero. For simulations of SAWs in continuous space this has not been done before. Our goal with this master thesis project is to find a interaction potential between the beads aiming to reduce the finite-size corrections.

2 Polymers and Self-Avoiding Walks

2.1 Random Walks

A lot is known about random walks (RWs), which are often referred to as Brownian motion or ideal chains. The basic idea behind a RW is that it is a Markovian process where in a fixed time interval Δt_i, a step with length $\Delta\vec{r}_i$ is taken into a random direction in a d-dimensional space. Each step is uncorrelated with all others and its direction is uniformly distributed, whereas the length of the steps has a normal distribution with mean b. Mathematically, we assign a d-dimensional vector $\vec{R}_i$ (i running from 1 to N) to each of the N time steps. The distance of the ith step, $\Delta\vec{r}_i$ and the end to end distance, $\vec{r}_e$, are given by

$$\begin{aligned}\Delta\vec{r}_i =&\vec{r}_i - \vec{r}_{i-1},\\ \vec{r}_e(N) =& \sum_{i=1}^{N} \Delta\vec{r}_i.\end{aligned}$$

Because of the Markovian character of the steps and normal distribution of their length we know that

$$\begin{aligned}\langle\Delta\vec{r}_i\cdot\Delta\vec{r}_j\rangle =&\delta_{i,j}b^2,\\ \langle\Delta\vec{r}_i\rangle =&0,\end{aligned}$$

with b^2 the average squared length of the steps. The mean squared end to end length $\left\langle \vec{r}_e{}^2\right\rangle$ is calculated by

$$\begin{aligned}\langle\vec{r}_e(N)^2\rangle =&\langle\sum_{i=1}^{N}\Delta\vec{r}_i\cdot\sum_{j=1}^{N}\Delta\vec{r}_j\rangle\\ =&\sum_{i,j=1}^{N}\langle\Delta\vec{r}_i\cdot\Delta\vec{r}_j\rangle\\ =&Nb^2 \qquad (1)\end{aligned}$$

which is independent of the dimension of the space that the RW is living in.
The probability distribution of a RW with length 1 from x_1 to x_2 in one dimension is given by

$$P_x(x_1,x_2;1) = \sqrt{\frac{3}{2\pi b^2}}\exp\left\{-\frac{3(x_2-x_1)^2}{2b^2}\right\},$$

where the constants are chosen such that in three dimensions the normalized probability distribution is given by

$$\begin{aligned}P_3(\vec{r}_1,\vec{r}_2;1) =&P_x(x_1,x_2;1)P_y(y_1,y_2;1)P_z(z_1,z_2;1)\\ =&(2\pi b^2/3)^{-\frac{3}{2}}\exp\left\{-\frac{3|\vec{r}_2-\vec{r}_1|^2}{2b^2}\right\}. \qquad (2)\end{aligned}$$

This expression ensures that in three dimensions, the mean of the squared step length is given by

$$\begin{aligned}\langle \vec{r}_e(1)^2\rangle =&\int_{-\infty}^{\infty}\vec{r}^{\,2}P_3(0,\vec{r},1))\mathrm{d}\vec{r}\\ =&\int_{-\infty}^{\infty}\vec{r}^{\,2}\sqrt{\frac{3}{2\pi b^2}}\exp\left\{-\frac{3\vec{r}^{\,2}}{2b^2}\right\}\mathrm{d}\vec{r}\\ =&b^2.\end{aligned}$$

To generalize eq.(2) to arbitrary length N, we write

$$P_3(\vec{r}_1,\vec{r}_2;N)=(2\pi Nb^2/3)^{-\frac{3}{2}}\exp\left\{-\frac{3|\vec{r}_2-\vec{r}_1|^2}{2Nb^2}\right\}. \tag{3}$$

We can prove this expression by induction: for $N=1$ it is true by eq.(2). Assuming that eq.(3) is true for N, then for $N+1$, we have

$$\begin{aligned}P_3(\vec{r}_1,\vec{r}_2;N+1)=&\int_{-\infty}^{\infty}P_3(\vec{r}_1,\vec{r},N)P_3(\vec{r},\vec{r}_2,1)\mathrm{d}\vec{r}\\ =&(2\pi b^2/3)^{-3}N^{-\frac{3}{2}}\int_{-\infty}^{\infty}\exp\left\{-\frac{3|\vec{r}-\vec{r}_1|^2}{2Nb^2}\right\}\exp\left\{-\frac{3|\vec{r}_2-\vec{r}|^2}{2b^2}\right\}\mathrm{d}\vec{r}\\ =&(2\pi(N+1)b^2/3)^{-\frac{3}{2}}\exp\left\{-\frac{3|\vec{r}_2-\vec{r}_1|^2}{2(N+1)b^2}\right\}.\end{aligned}$$

This calculation is explicitly done in Appendix 5.1. From this we can conclude that this expression is true for all values of N. This means that depending on the number of steps, the end to end distribution can be seen as a single step process with a squared average step length of Nb^2.

Physically, we could describe this RW as a number of connected springs with a spring energy $E_s(\Delta\vec{r})=\frac{1}{2}k_s(\Delta\vec{r})^2$. The spring constant is now temperature-dependent and can be written as $k_s=\frac{3}{Nb^2}k_bT$, with the corresponding Boltzmann distribution for the connected springs given by

$$\exp\left\{-\frac{E(N,\vec{r}_e)}{k_bT}\right\}=\exp\left\{-\frac{\sum_{i=1}^{N}E_s(\Delta\vec{r}_i)}{k_bT}\right\},$$

which is mathematically the same as the probability distribution of the RW.

From eq.(3), we can write down the probability distribution of a RW, starting in the origin and having length r, after N steps

$$P_3(r,N)\sim 4\pi r^2\exp\left\{-\frac{3r^2}{2Nb^2}\right\}.$$

The entropy of this system is therefore equal to

$$\begin{aligned}S_3(r,N)=&-k_b\log(P_3(r,N)),\\ =&S_{3,0}-2k_b\log(r)+k_b\frac{3r^2}{2Nb^2},\end{aligned}$$

where S_0 is a constant independent of r and N. Since there is no energy related to these RWs, we consider the free energy F at temperature T to be

$$\begin{aligned}F_3(r,N)=&E(r,N)-TS_3(r,N),\\ =&F_{3,0}+2k_bT\log(r)-k_bT\frac{3r^2}{2Nb^2}.\end{aligned}$$

Physically, this means that for minimizing the free energy there is a battle between a stretching volume term and contracting springs. For the equilibrium length, r_e, we must minimize the free energy. Solving $\frac{\partial}{\partial r}F(r,N) = 0$ with respect to r yields $\langle \vec{r}_e(N)^2 \rangle \sim Nb^2$, which is in agreement with eq.(1). These results are valid for a three-dimensional RW. In the rest of this thesis, part of the work will be done in two spacial dimensions and therefore the results for that case are listed below:

$$\begin{aligned} P_2(r,N) \sim& 2\pi r \exp\left\{-\frac{r^2}{Nb^2}\right\}, \\ S_2(r,N) =& S_{2,0} - 2k_b \log(r) + k_b \frac{r^2}{Nb^2}, \\ F_2(r,N) =& F_{2,0} + k_b T \log(r) - k_b T \frac{r^2}{Nb^2}, \end{aligned}$$

where again, $\langle \vec{r}_e(N)^2 \rangle \sim Nb^2$.

To relate the RW model to polymers, we have to assign to every ith step the ith bond, and to every location after the jth step the jth monomer. Having done so, the RW model can be used to predict the end-to-end length for highly dense polymers in a solution.

2.2 Self-avoiding walks

In real life, the RW description is not sufficient to give a correct prediction for the mean end-to-end length of a polymer in a strongly diluted solution. The main discrepancy with the RW predictions is due to the fact that interactions of different origins play an important role in the swelling of the polymer. These bead-bead interactions are mainly expected to be excluded volume (hard sphere) and attractive van der Waals interactions. An example for such a combined interaction is the Lennard Jones (LJ) potential, see Fig.3, where also two other examples are given. For high enough temperatures ($T \gg \epsilon$), the attractive part is negligible, and only excluded volume interactions play an important role for the end to end length of the polymer. Therefore a more accurate description of the size of the polymer is the size of a self-avoiding walk (SAW). These were first introduced by Flory [2], to describe polymers and are essentially a RW with an added rule: not visiting the same place more than once. An everyday example of this kind of problem is the computer game called “snake”, which is lost when an intersection occurs. There are numerous options to incorporate this feature in existing models for the RW and a few of them will be described in section 2.4. Although few analytical results exist, numerous numerical studies have been carried out towards a better understanding the SAW.

2.3 The Growth exponent ν

The mean squared end-to-end length $\langle \vec{r_e}^2 \rangle$ of the SAW for a very high number of steps can be written as

$$\lim_{N\to\infty} \langle \vec{r_e}^2(N) \rangle \sim N^{2\nu}, \tag{4}$$

where ν is called the growth exponent. For the RW (Brownian motion), this exponent is 1/2 regardless of the dimension. The growth exponent ν is a so-called critical exponent, appearing in various continuous phase transitions. This exponent is believed to be universal [3], which means that, for classical systems, ν only depends on the spacial

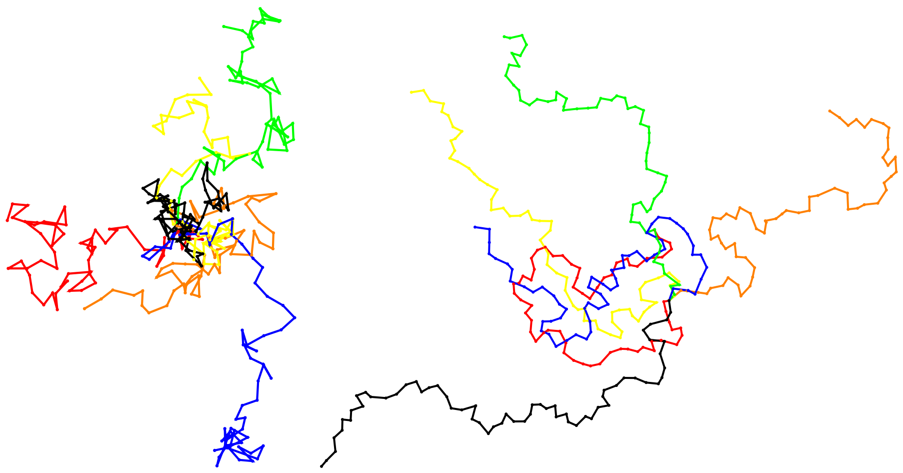


Figure 1: *six two-dimensional examples of random walks (left) and self-avoiding walks (right).*

dimension and the range of interaction, therefore it does not depend on the microscopic details of the model. The following are the known values for ν

$$\nu = \begin{cases} 1 & \text{if } d = 1, \\ 3/4 & \text{if } d = 2, \\ \approx 0.587597(7) & \text{if } d = 3, \\ 1/2 & \text{if } d \geqslant 4, \end{cases}$$

where d is the dimension. The values for dimensions $d = 1$, $d = 2$ and $d > 4$ are analytical. In three dimensions the best numerical value was found by Clisby [4] and is given in the previous list for the values of ν. In two dimensions the value $\frac{3}{4}$ is found by Nienhuis [5]. In dimensions larger than four, de Gennes [6] found, with a method called renormalization, that $\nu = 1/2$ so there is simply too much space for a SAW to run into itself, and it therefore behaves like a RW. In four dimensions a logarithmic correction, $\langle \vec{r}_e(N) \rangle \sim N(\log N)^{\frac{1}{4}}$, was found by the same author. A large group of physicists is trying to obtain the numerical value for ν in three dimensions as accurately as possible, but as we will see in section 3, finite-size corrections and computational limitations are troubling these efforts.

2.4 Implementation of the self avoiding walk in the random walk model

A physical interpretation for the RW, introduced in section 2.1, is the bead-spring model, where each coordinate $\vec{r}_i$ represents a bead and each step $\Delta\vec{r}_i$ a spring (see Figure 2 for an illustration). The Hamiltonian for this system is

$$\mathcal{H}_0 = \frac{1}{2} k_s \sum_{i=1}^{N} (1 - |\vec{r}_i - \vec{r}_{i-1}|)^2,$$

where k_s is the spring constant. The Hamiltonian is written in this form to ensure that, for high values of k_s or low temperatures (when the bead-spring model becomes

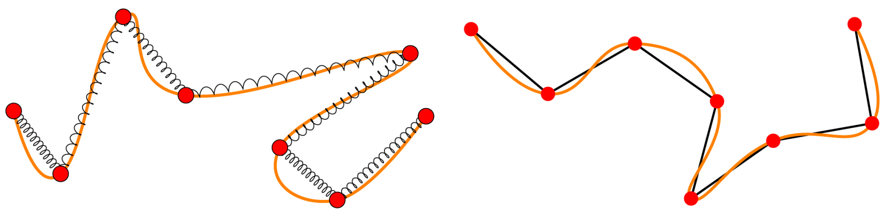


Figure 2: *the bead-spring model (left) and the bead-stick model (right).*

a bead-stick model), the springs have a non-zero length. To include the self-avoiding feature into this bead-spring model we need to add a short-range repulsive interaction term to the original Hamiltonian

$$\mathcal{H} = \mathcal{H}_0 + \sum_{i<j+1}^{N} \mathcal{U}(|\vec{r}_i - \vec{r}_j|).$$

However when an interaction potential is added to the Hamiltonian the model becomes analytically unsolvable, a feature which leaves us no choice but to search for approximate methods (such as mean field approximations) and simulations. A natural choice for this potential would be a hard-sphere interaction. The beads are assumed to be impenetrable with a diameter σ,

$$\mathcal{U}_{hs}(|\vec{r}_i - \vec{r}_j|) = \begin{cases} 0 & \text{if } |\vec{r}_i - \vec{r}_j| > \sigma, \\ \infty & \text{if } |\vec{r}_i - \vec{r}_j| < \sigma. \end{cases}$$

The repulsive interaction $\mathcal{U}$ does not necessarily have to be hard or become infinitely large when r approaches zero. A model that incorporates a finite energy penalty ϵ every time the bonds overlap, would describe a SAW with the same growth exponent ν as the SAW with a hard-sphere interaction. In the one-dimensional case this is proven [7]. This allows us to play around with different potentials as well, as long as they are short-ranged. Examples of potentials that we could use are a Gaussian repelling potential, or a simple positive square well potential, see Fig. 3.

2.5 The Flory calculation of the exponent ν

Paul J. Flory was the first to make an estimate for ν beyond the ideal chain configurational approach. Documented in [2], using a surprisingly accurate mean field approximation, he shows how a hard sphere interaction of the monomers leads to a nontrivial growth exponent $\nu = \frac{3}{d+2}$, in d dimensions.The argument is briefly described below. We start with the distribution function of a gaussian chain in d dimensions with end-to-end length R_F

$$P_d(R_F, N) \sim R_F^{d-1} \exp\left\{-\frac{1}{2} k_s R_F^2\right\}.$$

To find $P_{d,SAW}(R_F, N)$, we have to multiply this distribution function with the fraction of non-overlapping RWs:

$$P_{d,SAW}(R_F, N) = P_d(R_F, N) \cdot (1 - P_{overlap})^{N(N-1)/2},$$

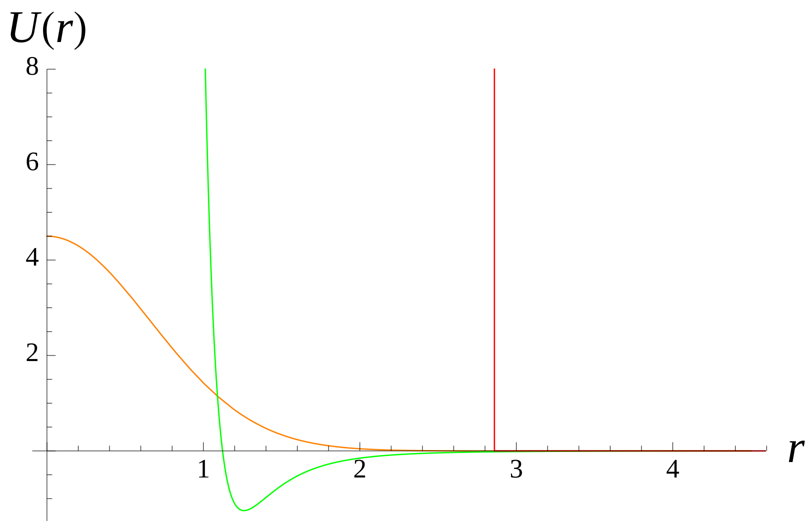


Figure 3: *a plot of a Gaussian potential, hard sphere potential and the Lennard Jones potential.*

where

$$P_{overlap} \sim \frac{v_e}{V},$$

with v_e the excluded volume of the monomer and $V \sim R_F^d$, roughly the volume of the SAW. The excluded volume v_e for any repulsive interaction $\phi(\vec{r})$ is simply given by $v_e = 2B_2$, with B_2 the second Virial coefficient

$$B_2 = -\frac{1}{2}\int_V (\exp\left\{-\frac{1}{k_bT}\phi(\vec{r})\right\} - 1)\mathrm{d}\vec{r}.$$

The entropy of the SAW is now written as

$$\begin{aligned} S_{SAW} &= -\log\left(P_{d,SAW}\right) \\ &= -\frac{1}{2}k_sR_F^2 + \log\left(R_F\right)^{(d-1)} - \frac{N(N-1)}{2}\log\left(1 - \frac{v_e}{R_F^d}\right) \\ &\approx -\frac{1}{2}k_sR_F^2 + (d-1)\log\left(R_F\right) - \frac{N^2}{2}\frac{v_e}{R_F^d}. \end{aligned}$$

Maximizing the entropy gives us the following equation for R_F:

$$-k_sR_F^2 + d - 1 + \frac{dN^2v_e}{2R_F^d} = 0.$$

If we solve this equation for $v_e = 0$, we get the same result as in section 2.1, namely

$$R_F^2 = \frac{d-1}{k_s}.$$

Since $k_s \sim \frac{1}{N}$ we conclude that $R_F^2 \sim N$. For a v_e significantly large (i.e. larger than some reference v_e^*), and using the limit

$$\lim_{N\to\infty}\frac{N^2}{(R_F^2)^{\frac{d}{2}}} \gg (d+1), \tag{5}$$

we can neglect the second term of the eq.(5) and solve it for R_F^2, which results in

$$R_F^2 = \left(\frac{dv_e}{k_s}\right)^{\frac{2}{d+2}} N^{\frac{4}{d+2}} \sim N^{2\left(\frac{3}{d+2}\right)}.$$

This implies that

$$\nu = \frac{3}{d+2}, \tag{6}$$

which yields the correct results in one and two dimensions, being also remarkably close to the three-dimensional value found by Clisby. If we put the result for ν back into eq.(5), we can see that the limit is still divergent for large N in one and two dimensions. If we investigate the limit of eq.(5) for the three-dimensional case, we see that it is not divergent, but constant. Therefore, eq.(6) is only 'valid' when $v_e \gg v_e^*$ with

$$v_e^* = \frac{4R_F^{3/2}}{3N^2}. \tag{7}$$

When $v_e \ll v_e^*$, the polymer behaves as a RW, with v_e* vanishing in the $N \to \infty$ limit. For a long time, Flory's estimate for ν in three dimensions was considered to be exact, but convincing numerical evidence has shown that this is not the case. We have to ask ourselves why this result is this accurate. It seems that the estimate for v_e is overestimated, but so is the spring constant: its contribution is not $1/N$ but $1/N^{2\nu}$ (since it is related to the mean squared displacement of the spring). It is due to these two errors mostly canceling each other out that such good estimates for ν have been obtained and attempts of independently correcting these errors have produced poor estimations for ν as the cancellation does not happen anymore [6].

2.6 Finite-size corrections in self-avoiding walks

Because of the universal nature of ν, it is interesting to know this value as accurately as possible. Therefore we would ideally simulate a polymer with the right behavior in the appropriate limit: $N \to \infty$. Unfortunately, computers have a finite amount of system memory and time is always expensive, which means we can only simulate the SAW for a finite number of steps and the higher we can push the number of steps, the less important the finite-size corrections will become. However, instead of pushing the limits of the number of steps, we can also try to make these finite-size corrections smaller.

In order to do so, we have to find an interaction potential which minimizes the finite-size corrections. To investigate the SAWs in the regime of finite N, we have to extend eq.(4) to include finite-size corrections:

$$\langle \vec{r}_e^{\,2} \rangle = N^{2\nu}\left(a_{0,e} + \frac{a_{1,e}}{N} + \frac{a_{2,e}}{N^2} + \ldots + \frac{b_{1,e}}{N^{\Delta_1}} + \frac{b_{2,e}}{N^{\Delta_1+1}} + \ldots + \frac{c_{1,e}}{N^{\Delta_2}} + \frac{c_{2,e}}{N^{\Delta_2+1}} \ldots\right), \tag{8}$$

where Δ_i are universal scaling exponents, independent of the choice of model, and $a_{i,e}$, $b_{i,e}$, $c_{i,e} \ldots$ are model dependent constants [8].

It is not a trivial job to numerically estimate the values of Δ_i. In two dimensions, there are not only two different theoretical predictions but also numerical results for Δ_1. The former are $3/2$ and $11/16$ while the latter varies from ≈ 0.5 to ≈ 1.5. Convincing numerical evidence is found by both Caracciolo [9] and Jensen [10] that the first scaling exponent is $\Delta_1 = 3/2$. We will focus on this value of Δ_1, but keep in mind that other values can still be candidates. In three dimensions, Dayantis and Palierne[11] worked out two different possibilities: $\Delta_i = 1/2$ and $\Delta_1 = 0.47$, $\Delta_2 = 1.05 \pm 0.02$ and $\Delta_3 = 2.2 \pm 0.2$. With their precision they were not able to distinguish these two choices, but it will remain an important question which of the predictions in the end will survive the always improving verdict of computer simulations. For sake of simplicity, we will work with only one option, namely $\Delta_i = 1/2$, to fit our our data in the next chapter.

For our SAWs in continuous space it is not guaranteed that the correction exponents Δ_i are the same as for the SAWs that live on a lattice. Although they are universal, the prefactors determine the relative importance of the corresponding Δ_i, and for this case it is not known which correction exponents are important. There can be other, not necessary analytical, correction exponents that we are not aware of yet. It is not our goal to specify these so we will use the prescription of the lattice models to provide fits for our results. This said, we have to remember that the fits may not represent the theoretical predictions.

3 Simulations of self-avoiding walks

Numerous problems in physics cannot be solved by exact equations. To find accurate solutions to these problems one needs to look for numerical methods such as modeling and simulation. The main goal of this thesis is to simulate the SAW with various interaction potentials between the beads, for a small number of steps, and compare their finite-size corrections. In this way we can find the a potential that minimizes these corrections. In the next sections we will discuss different simulation techniques. At first we will discuss exact enumeration, where we place a SAW of lengths up to $N = 32$ on a square lattice and let the computer calculate all the possible configurations in two dimensions. From these configurations we can extract the mean square displacement and get an estimate of its finite-size corrections. The next step will be generating a representative subsection of the total number of SAWs (statistical enumeration), which will allow us to push the maximal length up to $N = 250$. We will briefly explain this method and show some results in three dimensions. After that, we will investigate the finite-size corrections of the SAW in continuous space, by introducing a Metropolis algorithm, and discus the results in the last section.

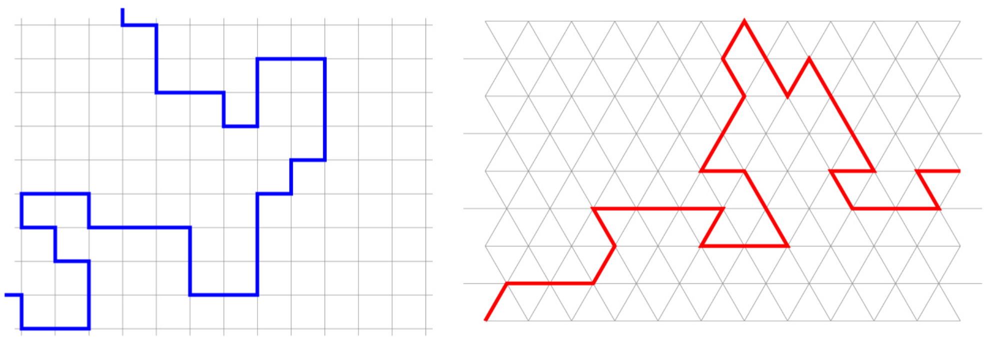


Figure 4: *two examples of a SAW, on a square lattice (left) and a triangular lattice (right).*

3.1 Simulations on the lattice model

In this section we will address simulations of SAWs in lattices. We begin with exact enumeration in two dimensions.

3.1.1 Exact Enumeration in two dimensions

The goal of exact enumeration is to generate all possible SAWs with N steps on a lattice. The most used lattices are the triangular and square ones, but others could be used as well. This can be done by a recursive program on a square lattice, an example of which is listed in section 6.1. The basic idea behind this algorithm is that it invokes a recursive function that produces a step in all possible directions. By calling itself, it creates an iterative structure from which all possible combinations are rendered. During the process it keeps track of the coordinates it already visited and the number of steps it has taken. If the chain of steps hits an occupied coordinate, the algorithm will stop this particular trial and move on from the previous step with a different move instead.

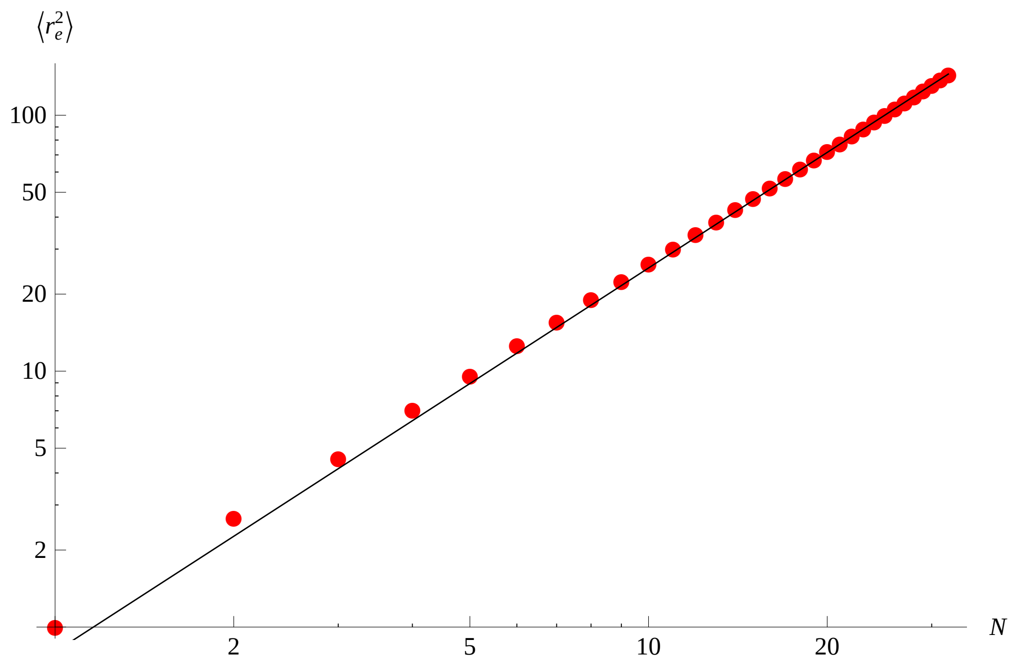


Figure 5: *A* log -log *plot of the mean displacement* $\langle \vec{r_e}^2 \rangle$ *as a function of* N.

If it keeps on hitting occupied coordinates, it will start subtracting steps until there is an allowed move. If the SAW reaches the desired the length, the algorithm gives back the coordinates of the nth step, and continues as if it has hit an occupied coordinate, this way generating all possible end points of the SAWs (including its degeneracy), from which we can calculate the mean squared end-tot-end length. The results of this simulation are shown in Fig.5 and took the program approximatively 30 hours of CPU time to get to $N = 32$. Once we have the results for the mean squared displacement, we want to address the finite-size corrections of this model. Let us recall eq.(8) for the finite-size corrections in two dimensions and take the three most important scaling exponents

$$\langle \vec{r_e}^2 \rangle = N^{2\nu}(a_{0,e} + \frac{a_{1,e}}{N} + \frac{b_{1,e}}{N^{3/2}} + \frac{a_{2,e}}{N^2} \ldots). \tag{9}$$

If we examine the results shown in Fig.5, we can see that in the small N regime the fit is too small. The discrepancies with the slope are then considered to be due to finite-size corrections. To make the finite-size corrections more visible, we take the derivative of $\log \langle \vec{r_e}^2 \rangle$ with respect to $\log N$ finding

$$\frac{\partial \log \langle \vec{r_e}^2 \rangle}{\partial \log N} = 2\nu_{\text{eff}} = 2\nu - \frac{a_{1,e}}{N} - \frac{3b_{1,e}}{2N^{3/2}} - \frac{2a_{2,e}}{N^2} \ldots \tag{10}$$

This means we have to take the numerical differential of the mean squared displacement, given by

$$\frac{\partial \log \langle \vec{r_e}^2 \rangle}{\partial \log N} \approx \frac{\log \langle \vec{r_e}^2(N + \Delta N) \rangle - \log \langle \vec{r_e}^2(N - \Delta N) \rangle}{\log (N + \Delta N) - \log (N - \Delta N)}, \tag{11}$$

with different step sizes $\Delta N \in \{1, 2, 3 \ldots\}$. The results of this method are shown in Fig.6. Depending on the numerical step size one takes, the results will oscillate differently around the fit that is depicted in the plot. For this reason, the average of step sizes until four was taken. The best fit comes with the values $a_{1,e} = 1.36$ $b_{1,e} = -2.70$ and $a_{2,e} = 2.01$. The main problem with exact enumeration is that the

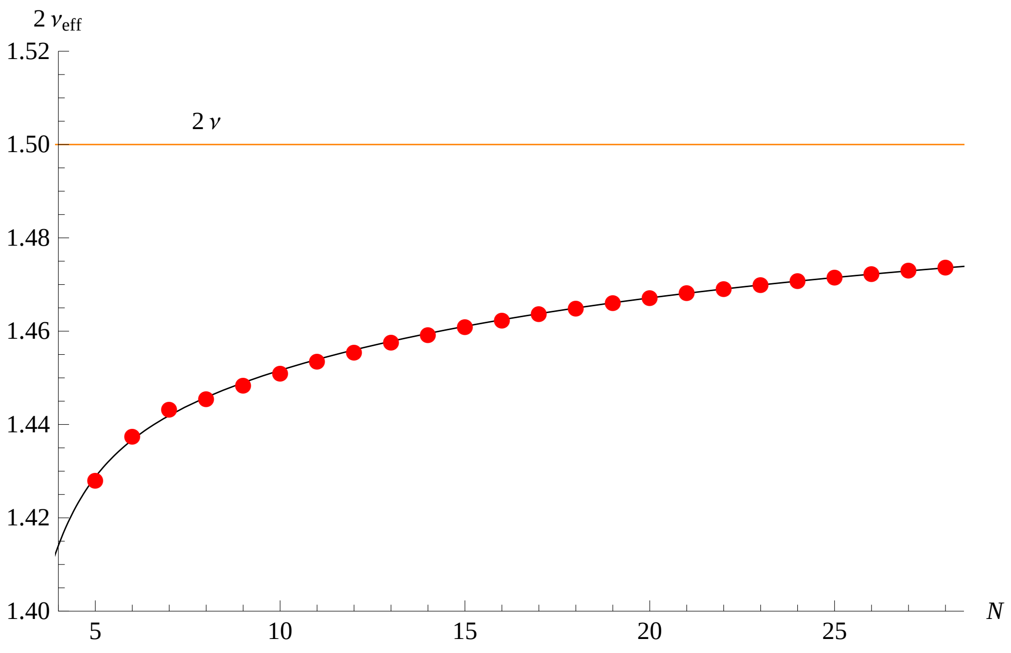


Figure 6: *A plot of the effective exponent $2\nu_{eff}$ as a function of N.*

CPU time will grow exponentially with N, being even worse for the three-dimensional case. Due to excessive use of computer power (50.000 CPU hours) and a smart trick to double the step length, Schram and al.[12] were able to generate all configurations for the three-dimensional case till $N = 36$. We will treat the three-dimensional case with a different method, called statistical enumeration, in the next section.

3.1.2 Statistical enumeration in three dimensions, including pruning and enrichment

As mentioned above, we now treat the three-dimensional case with a different method. For higher N it is not necessary to generate all possible SAWs, as we did with exact enumeration, but it is sufficient to get a representative sample of the SAWs. With statistical enumeration we will grow a number of SAWs on a cubic lattice, in the meanwhile keeping track, at every ith step, of the number of possibilities m_i the SAW has. The probability that we end up in such a SAW would then be $1/w_i$, with

$$w_i = \prod_{j=1}^{i} m_j, \tag{12}$$

the so called Rosenbluth weight [13]. However, in this procedure not all SAWs are generated with the same probability and therefore, if we want to properly compute $\langle \vec{r}_e^{\,2} \rangle$, we need to compensate each SAW by giving it a weight equal to the inverse of its probability

$$\langle \vec{r_e}^2 \rangle = \frac{\sum_k w_k \langle \vec{r_e}^2 \rangle_k}{\sum_k w_k}. \tag{13}$$

There are a few problems with this method. Because of the exponential behavior of the Rosenbluth weight the SAWs that have a large w_i dominate the results. This is shown by Barkema[14] by finding the results for the partition function (the total number of SAWs) to be very non-gaussian for $N > 150$. This makes the results badly reproducible.

To get rid of the dominance of one single SAW with a relatively high w_i, we follow the method called pruning and enrichment. It is statistically allowed, at any point in the SAW generating process, to stop the process with chance p, or continue the process (with chance $1 - p$) and multiply the Rosenbluth weight with $1/p$. This is called pruning, as it cuts some branches branches in the tree of SAWs. Pruning decreases the computational effort but increases the statistical errors. If, instead, the process is not stopped with chance p but continues twice with half the Rosenbluth weight it is called enrichment and has the opposite effect of pruning, increasing computing time but decreasing statistical errors.

Neither process introduces a bias in the results, as a SAW twice with half the weight will not change eq(13). If we apply pruning for SAWs with a low Rosenbluth weight

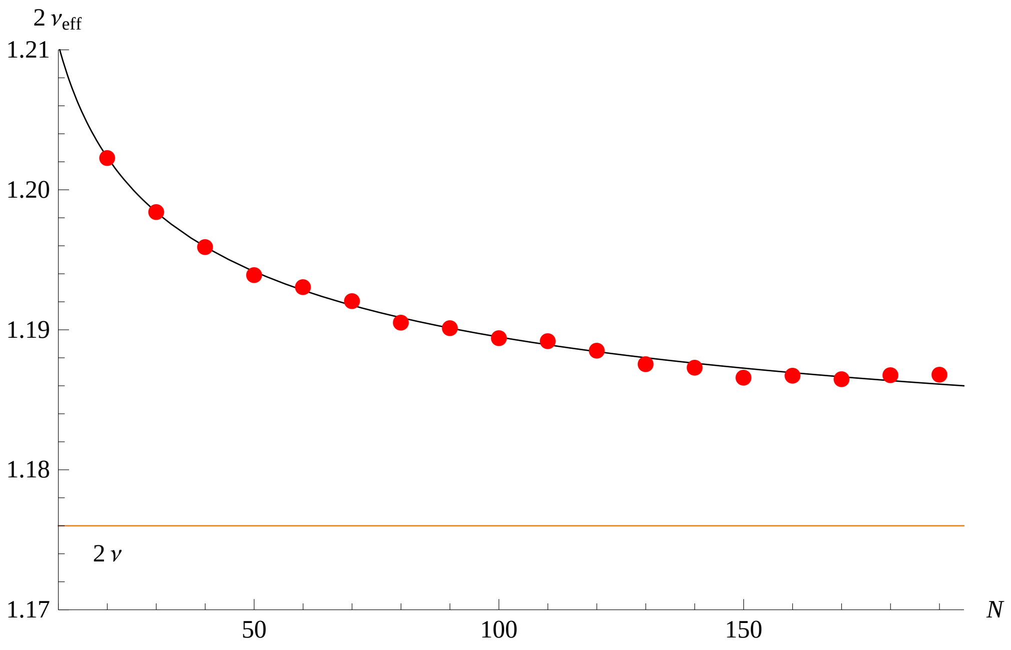


Figure 7: *A plot of the effective exponent $2\nu_{eff}$ in three dimensions, as a function of N.*

(compared to the average), it will cut branches of SAWs that will not contribute much to the measurements, whereas if we apply enrichment for SAW's with a high Rosenbluth weight, it will ensure us that there is not just one, or a few leading SAWs, but multiple ones, therefore yielding better statistics overall. It is common to choose $p = \frac{1}{2}$ and apply pruning when w_i is smaller than $\frac{1}{2}$ times it's average, and enrichment if w_i is 5 times bigger. In our program (see 6.2) we chose these values and let it generate 10^9 SAWs. The results for the finite-size corrections are shown in the figure 7, and for that data, $\Delta_1 = 1/2$ is chosen and fitted to

$$2\nu_{\text{eff}} = 2\nu - \frac{b_{1,e}}{2N^{1/2}} - \frac{a_{1,e}}{N} - \frac{3b_{2,e}}{N^{3/2}} \dots, \tag{14}$$

with the corresponding values $b_{1,e} = -0.31$, $a_{1,e} = 0.20$, and $b_{2,e} = -0.12$. These outcomes suggests that if the scaling exponent $\Delta_i = 1/2$ is absent in two dimensions, the signs of $a_{i,e}$ and $b_{i,e}$ are the same in two and three dimensions.

3.2 Monte Carlo simulations

We have seen in the previous section that when we cannot simulate all possible SAWs, it is important to get a representative sample of them. A Monte Carlo simulation does exactly that, it produces a suitable set of states such that, when measuring our desired quantities, we end up with good statistics. In that sense, the previous method, statistical enumeration including pruning and enrichment, is an example of a Monte Carlo simulation. In the following section we will discuss an effective method that ensures a sample of states equivalent to the Boltzmann distribution, closely following the theory described by Newman and Barkema[14].

The goal of the simulation is to calculate the mean squared displacement of the SAW which for one particular configuration $\vec{r}_i$ has a Hamiltonian of the form

$$\mathcal{H}_{\text{SAW}}(\vec{r}_i) = \mathcal{H}_0(\vec{r}_i) + \mathcal{H}_i(\vec{r}_i), \tag{15}$$

as defined in section 2.4. Ideally we would get the estimator $\langle \vec{r_e}^2 \rangle$ by integrating over all possible states

$$\langle \vec{r_e}^2(N) \rangle = \frac{1}{\mathcal{Z}} \prod_{i=1}^{N} \int d\vec{r}_i |\vec{r}_N - \vec{r}_0|^2 e^{-\beta \mathcal{H}_{\text{SAW}}(\vec{r}_i)}, \tag{16}$$

with the partition function

$$\mathcal{Z} = \prod_{i=1}^{N} \int d\vec{r}_i e^{-\beta \mathcal{H}_{\text{SAW}}(\vec{r}_i)}. \tag{17}$$

In these equations, β is the inverse temperature and $\vec{r}_0$ is placed in the origin. If we take a subset of M configurations, with a certain probability distribution p_i to measure $\langle \vec{r_e}^2 \rangle$, then these equations change into

$$\langle \vec{r_e}^2(N) \rangle = \frac{\sum_j^M \vec{r}_{e,j}^{\,2}(r_i) p_j^{-1} e^{-\beta \mathcal{H}_{\text{SAW}}(\vec{r}_i)}}{\sum_j^M p_j^{-1} e^{-\beta \mathcal{H}_{\text{SAW}}(\vec{r}_i)}}. \tag{18}$$

If we give all chosen configurations the same weight, which means

$$\langle \vec{r_e}^2(N) \rangle = \frac{\sum_j^M \vec{r}_{e,j}^{\,2}(r_i) e^{-\beta \mathcal{H}_{\text{SAW}}(\vec{r}_i)}}{\sum_j^M e^{-\beta \mathcal{H}_{\text{SAW}}(\vec{r}_i)}}, \tag{19}$$

then our sample would be dominated by a few configurations in the low energy domain and since our sample is always a very low fraction of the available configurations, this would again lead to very poor estimates. If we choose $p_j = e^{+\beta \mathcal{H}_{\text{SAW}}(\vec{r}_i)}/\mathcal{Z}$, the simulation spends the right amount of time in the chosen states. Our estimator, in this case, is then simply written as

$$\langle \vec{r}_e^{\,2}(N) \rangle = \frac{1}{M} \sum_j^M \vec{r}_{e,j}^{\,2}(r_j). \tag{20}$$

The problem now is how to choose the states that have this distribution. If we would generate states at random, we would not obtain a good sample as the higher energetic states will be rejected exponentially fast. To provide the simulation with the appropriate states we then make use of the so called Markov processes.

A Markov process starts with a certain state μ, and from this state it will generate a new state ν with a transition probability $P_{\mu\to\nu}$. Since a Markov process is independent of time and previously visited states, $P_{\mu\to\nu}$ depends on states μ and ν only. Moreover the probability of staying in the same state should be non-zero. If we let the simulation run for a longer time it will generate a Markov chain of states, and to ensure it resembles the Boltzmann distribution after a certain amount of running time, and not another, we have to add two extra conditions, called *ergodicity* and *detailed balance* which we will briefly describe in the following section.

3.2.1 Ergodicity and detailed balance

Ergodicity means that for any given state, all other states that should be available according to the Boltzmann distribution, can actually be reached after applying a finite number of steps in the Markov chain. Since we want to impose this on our simulations, we must always prove that this condition is met before using a specific algorithm.

The condition "detailed balance" is a little less straightforward and deserves more attention. This condition says that, if our system is in equilibrium (it has reached a certain stable probability distribution p_μ), the sum over outgoing rates times the probability distribution is the same as the sum over the incoming rates times their probability distributions. We can write this condition as

$$\sum_\nu p_\mu P_{\mu\to\nu} = \sum_\nu p_\nu P_{\nu\to\mu}, \tag{21}$$

and using the normalization condition,

$$p_\mu = \sum_\nu p_\nu P_{\nu\to\mu}. \tag{22}$$

If we look closely to this equation, we see that it is possible to create cycles of probability currents. To avoid these, we need to strengthen our condition by demanding that not only the sum of incoming and outgoing rates are the same, but that the rates between any two states must be equal

$$p_\mu P_{\mu\to\nu} = p_\nu P_{\nu\to\mu}. \tag{23}$$

This condition is called the detailed balance condition. By eliminating cycles, the matrix $P_{\mu\to\nu}$ has the eigenvector p_μ with eigenvalue one. A proof that if we start with any distribution v_μ, it will finally end up in in distribution p_μ can be found in appendix 5.2. If we want our probability distribution p_μ to be the Boltzmann distribution, we have to tune our transition rates $P_{\mu\to\nu}$ such that

$$\frac{P_{\mu\to\nu}}{P_{\nu\to\mu}} = \frac{p_\nu}{p_\mu} = e^{-\beta(E_\nu - E_\mu)}, \tag{24}$$

with E_ν the energy of the corresponding states.

In a Monte Carlo algorithm the transition amplitude $P_{\mu\to\nu}$ is defined by two processes. Starting from state μ we have to design a move that picks a new state ν with a certain probability $g_{\mu\to\nu}$. After picking this new state, we have to accept it with acceptance ratio $A_{\mu\to\nu}$. Inserting this in the previous equation, we obtain

$$\frac{P_{\mu\to\nu}}{P_{\nu\to\mu}} = \frac{g_{\mu\to\nu}A_{\mu\to\nu}}{g_{\mu\to\nu}A_{\nu\to\mu}}. \tag{25}$$

Depending on the structure of the program, we can choose $g_{\mu\to\nu}$ to be symmetrical. In this case we can define acceptance ratios to be

$$A_{\mu\to\nu} = \begin{cases} e^{-\beta(E_\nu - E_\mu)} & \text{if } E_\nu - E_\mu > 0 \\ 1 & \text{if } E_\nu - E_\mu < 0. \end{cases} \tag{26}$$

These choices make the algorithm a Metropolis algorithm, named after Nicolas Metropolis who was the first in making these choices for the transition rates with a simulation on the hard-sphere model for gases. One could make other choices, but this one has proven itself to be one of the most efficient and widely used algorithms.

3.2.2 Measurements

Before we start measuring observables as the mean squared displacement from our simulation, we have to make sure that the simulation has thermalized, by which we mean that the distribution of the states has converged to the Boltzmann distribution. Therefore we have run the program for a certain amount of time (the thermalization time) to ensure that our simulation has thermalized. It is not that easy to determine whether one has reached it or not because the simulation could be in a local minimum of the free energy, and stay there for a long time. Therefore it is useful to compare the measurements for different starting configurations. Another check could be that if we found a thermalization time, to double it and check whether it affects the results or not.
After we ensured thermalization we can start making measurements. How often we can effectively measure a new independent state depends on how fast the states in the Markov chain are changing. A very useful quantity that describes this rate is the correlation time τ. If we measure the mean squared displacement, then the correlation time τ is defined by the autocorrelation function

$$\chi(\Delta t) = \frac{1}{T-\Delta t}\sum_{t'}^{T}\left[\vec{r}_e^{\,2}(t') - \langle\vec{r}_e^{\,2}\rangle\right]\left[\vec{r}_e^{\,2}(t'+\Delta t) - \langle\vec{r}_e^{\,2}\rangle\right] \sim e^{-\Delta t/\tau}. \tag{27}$$

The autocorrelation function falls of exponentially, the bigger Δt is, the smaller the states will be correlated to each other. A proof of this can be found on page 66 of Newton and Barkema [14].

3.3 Simulations on continuous models

3.3.1 Our algorithm

In the previous section, we constructed a set of requirements that a Monte Carlo simulation has to meet in order to produce a set of states that represents the Boltzmann distribution. In the following section we will describe a Metropolis algorithm that simulates the SAW in continuous space and discuss the requirements of detailed balance and ergodicity. Our algorithm consists of two Monte Carlo moves, one tick move and one pivot move, which produce new states in the Markov chain. The tick move is a simple move which takes a random state μ, with a bead configuration $\{\vec{r}_0, \vec{r}_1 \ldots \vec{r}_N\}$to a new state of the SAW, ν, by taking a random bead coordinate $\vec{r}_i$ to a nearby coordinate $\vec{r}_i'$. A visualization of the tick move is shown in Fig.8. The probability of selecting the new coordinates is uniformly distributed within a circle (a sphere, if in three dimensions) of radius R centered in $\vec{r}_i$. This means that the selection probability is symmetric under interchanging states μ and ν, i.e. $g_{\mu\to\nu}/g_{\nu\to\mu} = 1$. Detailed balance is then automatically satisfied if we choose our transition rates according to the Metropolis algorithm, i.e. eq.(26). If the radius R is chosen large compared to the average step size, the acceptance probability will be small (the energy of the springs becomes very large), where on the other hand, if a small R is chosen, the acceptance probability might be high but the correlation time will grow. We can then tune R in order to keep the program fast and efficient. With this move it is trivial that all configurations can be reached within a final amount of steps and therefore the requirement of ergodicity is satisfied.

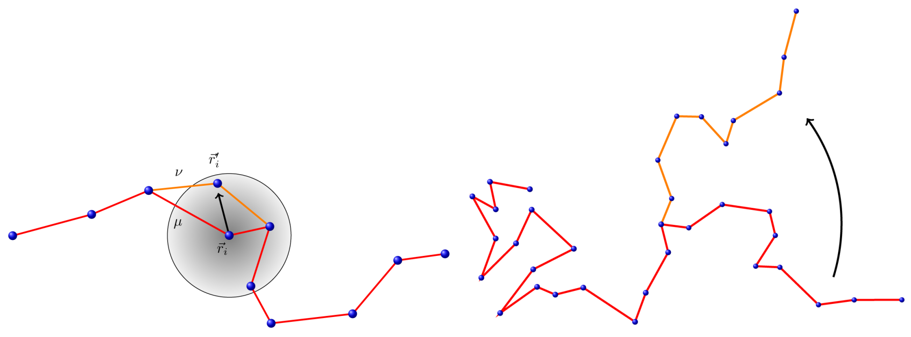


Figure 8: *visualization of the tick(left) move and the pivot move(right).*

The pivot move starts in the same way with certain a bead configuration from which a random bead $\vec{r}_p$ is picked. Instead of moving it, it rotates all beads $\vec{r}_j$, with $j > p$, around $\vec{r}_p$, with an angle α chosen in the interval $[-\theta, \theta]$. A visualization of this move is given in Fig.8. Again we can tune θ to get a more efficient program. The probability distribution within this interval is uniform, and therefore the move is symmetric under interchanging the old and the new state and rotate with $-\alpha$ instead. For this reason, we can satisfy detailed balance if we define the acceptance ratios to satisfy eq.(26). This pivot move itself does not satisfy ergodicity, as it preserves the lengths of the steps.

If we consider the transition towards a new state to be a combination of both moves, the requirements of detailed balance and ergodicity are satisfied. I have chosen to consider one Monte Carlo step (one move towards a new state) to be a combination of twice the tick move and ten times the pivot move.

3.3.2 Thermalization and the Autocorrelation Function

To obtain the thermalization time for our algorithm, we have to feed it different initial states. We have chosen to pick a fully stretched SAW with average step length $b = 1,2$

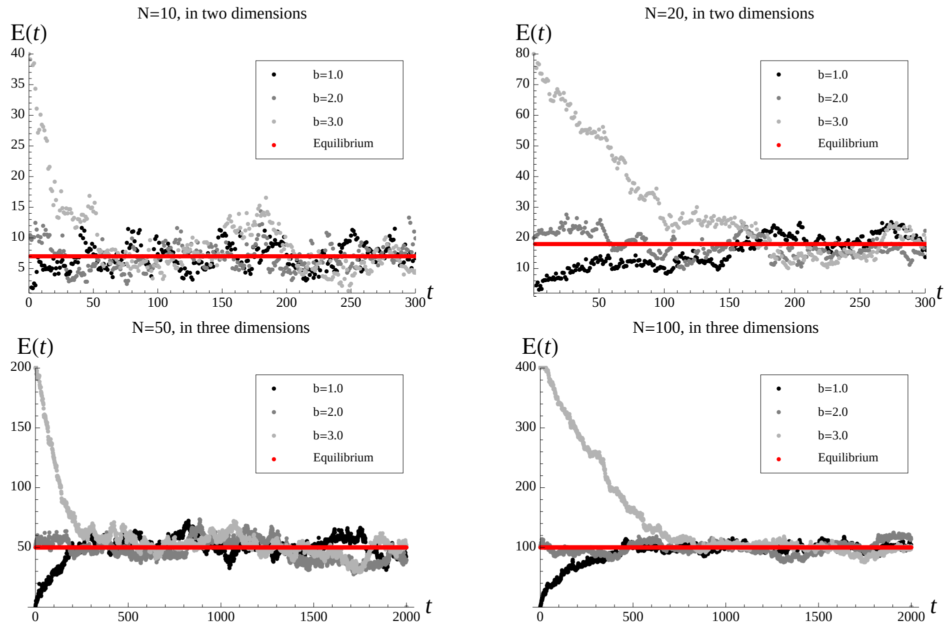


Figure 9: *plot of the energy $E(t)$ as a function of Monte Carlo steps t, for three different starting configurations.*

and 3. The repulsive potential is temporarily set to be a Gaussian one $\mathcal{U}(\vec{r_i}-\vec{r_j}) = 4.5\exp\{-2(\vec{r_i}-\vec{r_j})^2\}$. To obtain a good estimate of the thermalization it is necessary to check when these different starting states converge towards an equilibrium. We have chosen to measure the thermalization time with the energy instead of the mean squared displacement, since the variance in the measurement of the latter is too large, making it unsuitable to do so. The results are shown in Fig.9.

As we can see, the longer the SAWs are the longer their thermalization time is. We also see that in two and three dimensions the simulation with the starting configuration associated with $b = 2$ are almost immediately in equilibrium. Although not strictly necessary in all simulations, the thermalisation time has been chosen to be a thousand Monte Carlo steps as a matter of caution.

The autocorrelation functions for this algorithm are depicted in Fig.10. If we attempt to fit an exponentially decaying function as suggested by eq.(27) we find this is not possible. The data suggests that it actually falls off with a stretched exponential function written as

$$-\log\left[\chi(\Delta t)\right] \sim \left(\frac{\Delta t}{\tau}\right)^{\alpha},$$

with an exponent α smaller than one. Albeit an interesting point for further research, it goes beyond the scope of this work. What is interesting to us is how fast the autocorrelation function is decreasing with a factor of e^{-1}. With this method we estimated $\tau \approx 1.8$ Monte Carlo steps, regardless of the chosen repulsive interaction. Note that the acceptance ratio is approximately 50%, which means that after approximately every four Monte Carlo steps we can measure a relevant new state.

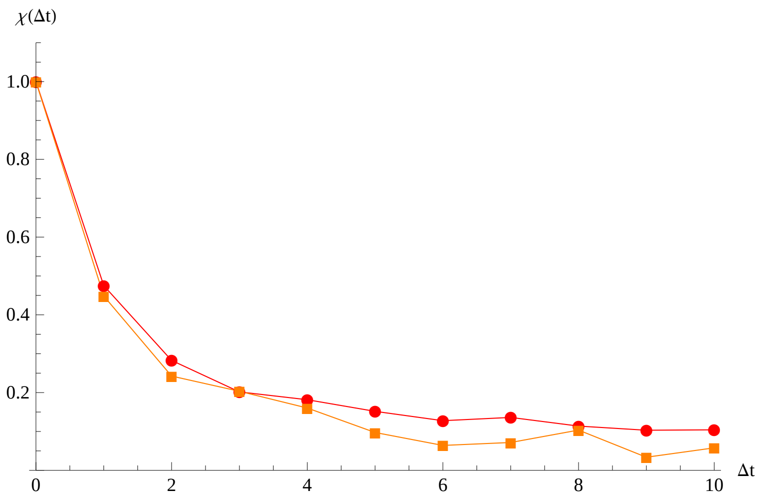


Figure 10: *plot of the autocorrelation function $\chi(\Delta t/\tau)$. The time Δt is measured in Monte Carlo steps. The dots represent the autocorrelation function in two dimensions and the squares the one in three dimensions.*

3.4 Specific goal of our simulations

To specify what we mean by minimizing the finite-size corrections, we must define a more specific goal. Simply tuning our chosen interactions such that all prefactors turn zero is not always possible and judging by our first trial runs of our program, it is suggested that this is not the case. Therefore we defined a criterion that our data has to meet by the following,

Find a bead-bead interaction potential for which the effective growth exponent

$$2\nu_{\mathit{eff}} = \frac{\partial \log \langle \vec{r}_e^{\,2} \rangle}{\partial \log N} \tag{28}$$

converges to a range of $[\nu - \delta, \nu + \delta]$, with $\delta = 0.02$, for minimal N.

The Hamiltonian for the Metropolis algorithm, described in the previous section, is given by

$$\mathcal{H} = \mathcal{H}_0 + \sum_{i<j+1}^{N} \mathcal{U}(|\vec{r}_i - \vec{r}_j|).$$

Note that the sum of the interaction is over all pairs of the beads except nearest neighbors. The potentials we are going to use in our two-dimensional simulations are the Gaussian, the Lennard Jones and the hard sphere,

$$\begin{aligned} \mathcal{U}_G(\vec{r}_i - \vec{r}_j) =& a e^{-b|\vec{r}_i - \vec{r}_j|^2}, \\ \mathcal{U}_{LJ}(\vec{r}_i - \vec{r}_j) =& \frac{a}{r^{12}} - \frac{b}{r^6}, \\ \mathcal{U}_{HS}(|\vec{r}_i - \vec{r}_j|) =& \begin{cases} 0 & \text{if } |\vec{r}_i - \vec{r}_j| > \sigma, \\ \infty & \text{if } |\vec{r}_i - \vec{r}_j| < \sigma. \end{cases} \end{aligned}$$

In three dimensions, we restrict ourselves to the hard sphere and Gaussian potentials. Note that only positive values for a and b are to be used, since the potential should be repulsive.

3.5 Results in two dimensions

We start the analysis of the two-dimensional case by taking a look at the hard sphere potential. The results of running simulations with three different diameters σ are shown in Fig.11.

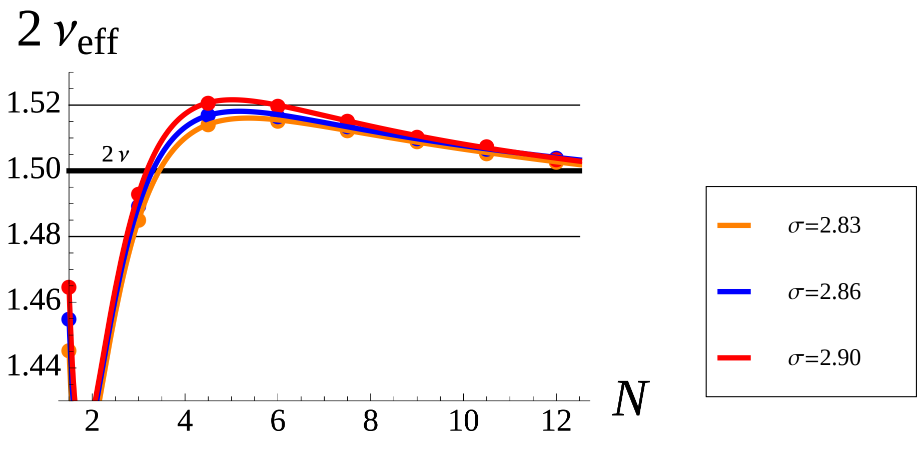


Figure 11: *a plot of the effective growth exponent $2\nu_{\mathit{eff}}$, as a function of the number of steps N, subject to the hard sphere potential with a diameter $\sigma = 2.83$, $\sigma = 2.86$ and $\sigma = 2.90$.*

From the figure, we see that the larger the diameter is, the higher the effective exponent grows when compared to ν. The best value for the diameter according to our criterion is then $\sigma = 2.86$. This value is larger than the average step size and therefore nearest neighbors do overlap, but as has already been pointed out, nearest neighbor interactions are not included in the Hamiltonian.

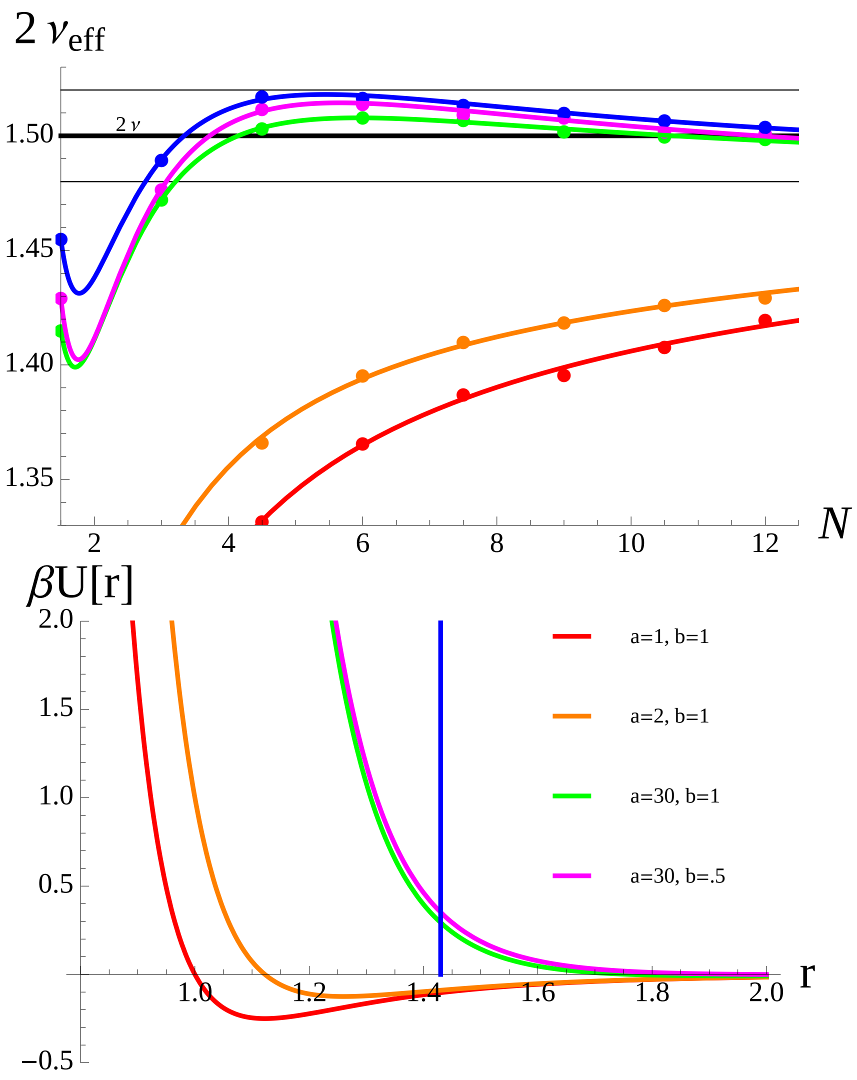


Figure 12: *a plot of the effective growth exponent $2\nu_{eff}$ (top) in two dimensions, as a function of the number of steps N, subject to the Lennard Jones potential $U(r) = a/r^{12} - b/r^{6}$. The colors represent the potential parameters corresponding to the ones that are declared in the plot of potentials(down).*

For the next simulation we have chosen to simulate the SAW with the Lennard Jones potential. The results and the corresponding potentials are shown in Fig.12. The best hard sphere result, in blue, is shown as a reference. For a standard Lennard Jones potential i.e. $a = 1$ and $b = 1$, we see that the effective growth exponent is much lower than our reference hard sphere case. The more we tuned the parameters a and b to get the data to move to the region $[\nu - 0.02, \nu + 0.02]$, the more the Lennard Jones potential approaches the hard sphere potential.

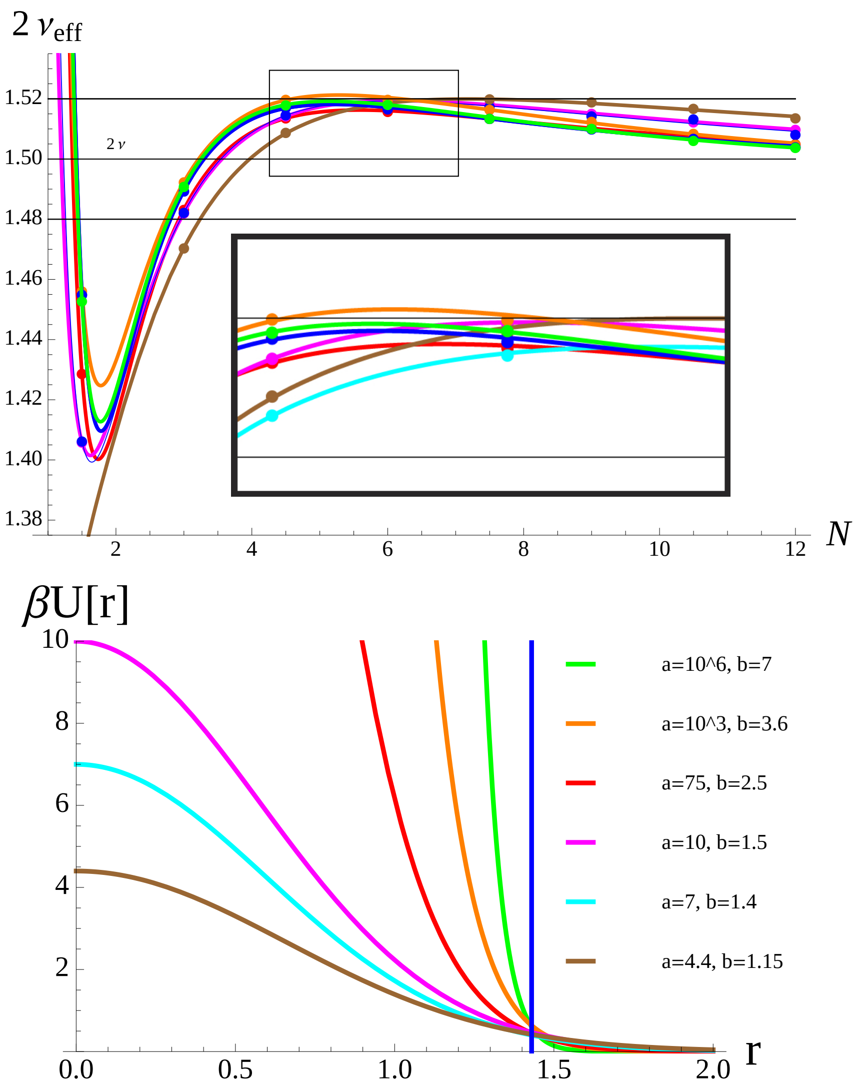


Figure 13: *a plot of the effective growth exponent $2\nu_{eff}$ (top) in two dimensions, including a zoom of the rectangular area, as a function of the number of steps N, subject to the Gaussian potential given by $U(r) = a\exp(-br^2)$. The colors represent the potential parameters corresponding to the ones that are declared in the plot of the potentials (down).*

For the simulations with the Gaussian potential, we have chosen to vary the parameters a and b such that it approaches the blue reference hard sphere case. The results of the simulations including a plot of the corresponding potentials are shown in Fig.13. We see that, according to our criterion, we get the best results for the values $a = 10^7$ and $b = 7$. We have to conclude that with these values it produces better results then our best hard sphere potential, but it is a close call between these two. Furthermore, we can see that there might be a better hard sphere potential if we pick a value for the diameter between $\sigma = 2.90$ and $\sigma = 2.86$. The more our Gaussian approaches the hard sphere, the closer our measurements become.

3.6 Results in three dimensions

In three dimensions, we have restricted ourselves to the simulations with interactions of the hard sphere potential and the Gaussian potential. The results of the former are shown in Fig.14.

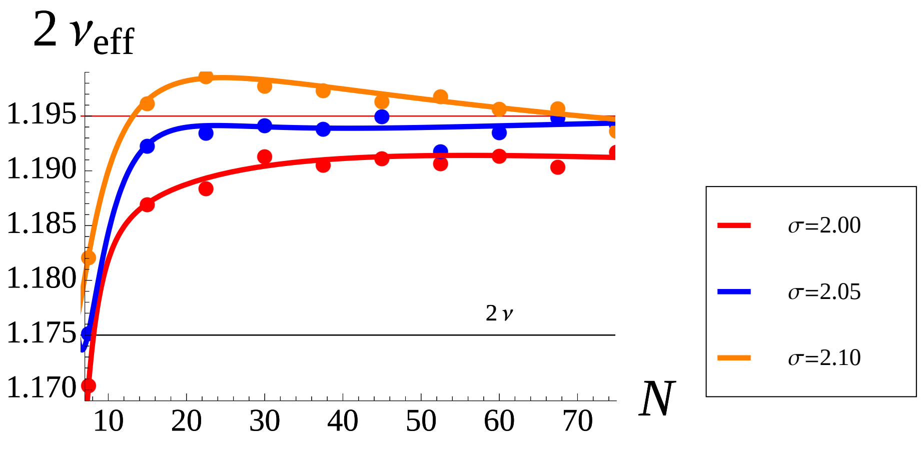


Figure 14: *a plot of the effective growth exponent $2\nu_{\text{eff}}$, as a function of the number of steps N, subject to the hard sphere potential.*

From these results we see that the finite-size corrections are more persistent then in two dimensions, therefore suggesting an enlargement of the step size. For these simulations we spend within the same order of CPU time as for the simulations that are done in two dimensions. As a consequence our measurements are less precise. The best value for the diameter that we found, is $\sigma = 2.05$, according to our criterion.

For the Gaussian potential, we adopted the same method as in the two-dimensional case. We found values for a and b such that the hard sphere potential is gradually approached. The results and their corresponding potentials are shown in Fig.15. We see that the two gaussians with values $a = 10^7$, $b = 16$, and $a = 100$, $b = 5$, give approximately the same results as our best hard sphere potential, even though the latter is not one of the closer ones to the hard sphere. We therefore conclude there is a close call between the Gaussian potential and the hard sphere potential as we had in the two dimensions.

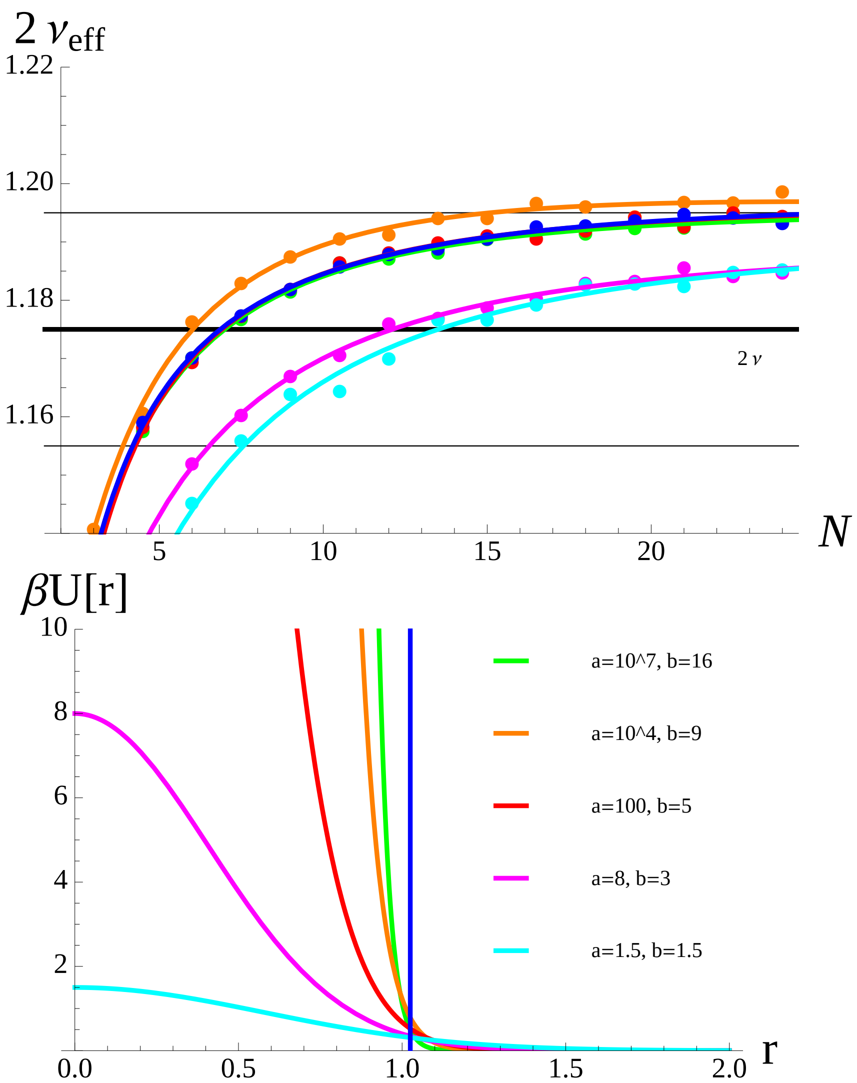


Figure 15: *a plot of the effective growth exponent $2\nu_{eff}$ (top) in three dimensions, as a function of the number of steps N, subject to the Gaussian potential given by $U(r) = a\exp(-br^2)$. The colors represent the potential parameters corresponding to the ones that are declared in the plot of the potentials (down).*

4 Conclusion, discussion and further research

4.1 Conclusion

Various simulations of the self-avoiding walk in two-and three-dimensional space have been performed in order to obtain insight about the finite-size corrections of the mean squared displacement characterizing the self-avoiding walk. Subject to various bead-bead interactions, these corrections could be measured and tuned in order to get the effective growth exponent ν_{eff} in the desired region of $[\nu - 0.02, \nu + 0.02]$ for the smallest number of steps. In both dimensions, the hard sphere interactions with $\sigma_{2D} = 2.86$ and $\sigma_{3D} = 2.05$ gave the best results that we have found. In two dimensions these results were approached by both the Lennard Jones and the Gaussian potential. In particularly, the Gaussian potential with values $a = 10^6$ and $b = 7$ produced a slightly better result, remaining a close call between these two. In three dimensions the results of the hard sphere potential can be approached by the Gaussian parameters $a = 10^7$, $b = 16$ and $a = 100$, $b = 5$, but not improved.

4.2 Discussion

In our simulations we have found that the hard-sphere potential and the Gaussian potential can produce results that are very close to each other. It is therefore hard to tell which of the two potentials truly minimizes the finite-size corrections according to our criterion. In our three-dimensional simulations, we have found Gaussian potentials that approximately have the same results as the best hard sphere potential. It is noteworthy that by tuning the parameters of the Gaussian potential we were able to move the maximum of ν_{eff} to lower values of N, without altering its value. The gaussian potential with values $a = 100$ and $b = 5$ produced the same result as the best hard-sphere potential, but is not the steepest one we have tested. This suggests that if we tune a and b further there is maybe a better potential to be found that could improve on the hard sphere result. However as far as we can be sure, no potential is performing significantly better than a simple hard-sphere potential with a fine-tuned diameter.

4.3 Further Research

In this master thesis project we considered only a two-particle interaction. Since by nature the beads are relatively close to their nearest and next-nearest neighbors, it might be interesting to include a three-particle interaction to see how the finite-size corrections are depending on these and if they can be tuned to minimize the corrections further.

For the hard sphere potential, there seems to be a clear relation between the diameter and the maximum value of the effective growth exponent; the bigger the diameter the higher the maximum value of the effective growth exponent. Similarly for the Gaussian potential, the bigger the amplitude a, the higher the maximum value of the effective growth exponent, while higher values of b yielded a smaller maximum. A combination of both parameters moved the maximum value for the growth exponent towards a lower value of N, suggesting that there might be a relation between the excluded volume v_e, defined by the second Virial coefficient, and the prefactors of the correction exponents. This might be very interesting to explore in further research.

5 Appendices

5.1 Appendix A

To generalize (2) to arbitrary length N, we write:

$$P_3(\vec{r}_1, \vec{r}_2; N) = (2\pi N b^2/3)^{-\frac{3}{2}} \exp\left\{-\frac{3|\vec{r}_2 - \vec{r}_1|^2}{2Nb^2}\right\}. \tag{29}$$

We can prove this by Induction: For N=1 it is true by (2). Assuming that $P_3(\vec{r}_1, \vec{r}_2; N)$ is true then for $N+1$ we have:

$$\begin{aligned}
P_3(\vec{r}_1, \vec{r}_2; N+1) &= \int_{-\infty}^{\infty} P_3(\vec{r}_1, \vec{r}, N) P_3(\vec{r}, \vec{r}_2, 1) \mathrm{d}\vec{r}, \\
&= (2\pi b^2/3)^{-3} N^{-\frac{3}{2}} \int_{-\infty}^{\infty} \exp\left\{-\frac{3|\vec{r} - \vec{r}_1|^2}{2Nb^2}\right\} \exp\left\{-\frac{3|\vec{r}_2 - \vec{r}|^2}{2b^2}\right\} \mathrm{d}\vec{r}, \\
&= (2\pi b^2/3)^{-3} N^{-\frac{3}{2}} \int_{-\infty}^{\infty} \exp\left\{-\frac{3|\vec{r}\,'|^2}{2Nb^2}\right\} \exp\left\{-\frac{3|\vec{r}\,' - \Delta\vec{r}|^2}{2b^2}\right\} \mathrm{d}\vec{r}\,', \\
&= (2\pi b^2/3)^{-3} N^{-\frac{3}{2}} \int_{-\infty}^{\infty} \exp\left\{-\frac{3}{2Nb^2}((N+1)|\vec{r}\,'|^2 - 2N\vec{r}\,' \cdot \Delta\vec{r} + N|\Delta\vec{r}|^2)\right\} \mathrm{d}\vec{r}\,', \\
&= \frac{(2\pi b^2/3)^{-3} N^{-\frac{3}{2}}}{\sqrt{(N+1)^3}} \int_{-\infty}^{\infty} \exp\left\{-\frac{3}{2Nb^2}(|\vec{q}|^2 - \frac{2N\Delta\vec{r} \cdot \vec{q}}{\sqrt{N+1}} + N|\Delta\vec{r}\,|^2)\right\} \mathrm{d}\vec{q}, \\
&= \frac{(2\pi b^2/3)^{-3} N^{-\frac{3}{2}}}{\sqrt{(N+1)^3}} \int_{-\infty}^{\infty} \exp\left\{-\frac{3}{2Nb^2}(\left|\left(\vec{r} + \frac{N\Delta\vec{r}}{\sqrt{N+1}}\right)\right|^2 - \frac{2N\Delta\vec{r} \cdot \vec{r}}{\sqrt{N+1}}\right. \\
&\qquad \left. - \frac{4N^2|\Delta\vec{r}|^2}{N+1} + N|\Delta\vec{r}|^2)\right\} \mathrm{d}\vec{r} \\
&= \frac{(2\pi b^2/3)^{-3} N^{-\frac{3}{2}}}{\sqrt{(N+1)^3}} \int_{-\infty}^{\infty} \exp\left\{-\frac{3}{2Nb^2}(|\vec{r}|^2 - \frac{N^2|\Delta\vec{r}|^2}{N+1} + N|\Delta\vec{r}|^2)\right\} \mathrm{d}\vec{r}, \\
&= \frac{(2\pi b^2/3)^{-3} N^{-\frac{3}{2}}}{\sqrt{(N+1)^3}} \int_{-\infty}^{\infty} \exp\left\{-\frac{3}{2Nb^2}|\vec{r}|^2\right\} \exp\left\{-\frac{3}{2(N+1)b^2}|\Delta\vec{r}|^2\right\} \mathrm{d}\vec{r}, \\
&= (2\pi(N+1)b^2/3)^{-\frac{3}{2}} \exp\left\{-\frac{3|\vec{r}_2 - \vec{r}_1|^2}{2(N+1)b^2}\right\},
\end{aligned}$$

where $\Delta\vec{r} = \vec{r}_2 - \vec{r}_1$. The last expression completes the proof, from which we deduce that (3) is true for all N. In this calculation I have used a change of variables several times, completing the square once, and finished it by a Gaussian integral.

5.2 Appendix B

In this appendix I will proof that when we start with a with a probability distribution $w_\mu(0)$ and perform a Monte Carlo simulation with a long Markov chain, that we end up in the desired probability distribution p_μ defined by

$$p_\mu = \sum_\nu p_\nu P_{\nu \to \mu}. \tag{30}$$

Starting with $w_\mu(0)$ we end up in $w_\mu(t)$ after t steps in the Markov chain, where

$$w_\mu(t) = \sum_\nu P^t_{\mu \to \nu} w_\nu(0).$$

If we write $w^\nu(0)$ in terms of eigenvectors v_i^ν of matrix $P_{\nu\to\mu}$ we obtain

$$\begin{aligned}
w_\mu(t) &= \sum_\nu P^t_{\mu\to\nu} \sum_i a_i v_i^\nu(0) \\
&= \sum_\nu \sum_i a_i \lambda_i^t v_{i,\mu}(0),
\end{aligned}$$

with λ_i the eigenvalues corresponding to the eigenvectors v_i. In the limit of large t, this equation will be dominated by the largest eigenvalue λ_0 of the Markov matrix and therefore $w_\mu(t)$ becomes proportional to v_μ^0. If we combine the fact that there is at least one eigenvalue of a Markov matrix that is one, with eq.(30) we can conclude that

$$\lim_{t\to\infty} w_\mu(t) = p_\mu. \tag{31}$$

If there are more eigenvalues that are one, then this would lead to a violation of the condition of ergodicity, which we excluded by designing the Monte Carlo algorithm.

References

[1] P. Grassberger, P. Sutter and L. Schäfer. Field theoretic and Monte Carlo analysis of the Domb-Joyce model. *J. Phys. A*, **30**:7039, 1997.

[2] P. Flory. *Principles of polymer chemistry.* Cornell University Press, 1953.

[3] H. E. Stanly. Scaling, universality, and renormalization: three pillars of modern critical phenomena. *Rev. of Mod. Phys.*, **71**:358, 1999.

[4] N. Clisby. Accurate estimate of the critical exponent ν for self-avoiding walks via a fast implementation of the pivot algorithm. *Phys. Rev. Lett.*, **104**:055702, 2010.

[5] B. Nienhuis. Critical behavior of two-dimensional spin models and charge asymmetry in the coulomb gas. *Journal of Statistical Physics*, **34**:731, 1984.

[6] P. G. de Gennes. *Scaling concepts in polymer physics.* Cornell University Press, 1979.

[7] T. Kennedy. Ballistic behavior in a 1D weakly self-avoiding walk with decaying energy penalty. *J. of Stat. Phys.*, **77**:565, 1994.

[8] F. Wegner. Corrections to scaling laws. *Phys. Rev. B*, **5**:4529, 1972.

[9] S. Caracciolo, A. J. Guttmann, I. Jensen, A. Pelissetto, A. N. Rogers, and A. D. Sokal. Correction-to-scaling exponents for two-dimensional self-avoiding walks. *J. of Stat. Phys.*, **120**:1037, 2005.

[10] I. Jensen. Enumeration of self-avoiding walks on the square lattice. *J. of Phys. A*, **37**:5503, 2004.

[11] J. Dayantis and J.F. Palierne. Scaling exponents of the self-avoiding-walk-problem in three dimensions. *Phys. Rev. B*, **49**:3217, 1994.

[12] R. D. Schram, G. T. Barkema, and R. H. Bisseling. Exact enumeration of self-avoiding walks. *J. Stat. Mech.*, **6**:19, 2011.

[13] A.W. Rosenbluth, M. L. N. Rosenbluth. Monte Carlo calculation of the average extension of molecular chains. *J. of Chem. Phys.*, **49**:356, 1955.

[14] G. T. Barkema. *Monte Carlo simulation of lattice polymer models.* lecture notes, 1999.

6 C Programs

6.1 C-Program for exact enumeration

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#define N 10
double Z[N+1],sumr2[N+1];
int bezet[2*N+1][2*N+1];
double ZD[N+1], sumr2D[N+1];
double ZR[N+1], sumr2R[N+1];
int bezetD[2*N+1][2*N+1],bezetR[2*N+1][2*N+1];
void godown(int x,int y, int n)
{
        if (!bezetD[x+N][y+N])
        {
                bezetD[x+N][y+N]=1;
                ZD[n]++;
                sumr2D[n]+=x*x+y*y;
                if (n<N)
                {
                        godown(x+1,y   ,n+1);
                        godown(x-1,y   ,n+1);
                        godown(x   ,y+1,n+1);
                        godown(x   ,y-1,n+1);
                }
                bezetD[x+N][y+N]=0;
        }
}
void goright(int x,int y, int n)
{
        if (!bezetR[x+N][y+N])
        {
                bezetR[x+N][y+N]=1;
                ZR[n]++;
                sumr2R[n]+=x*x+y*y;
                if (n<N)
                {
                        goright(x+1,y   ,n+1);
                        goright(x-1,y   ,n+1);
                        goright(x   ,y+1,n+1);
                        goright(x   ,y-1,n+1);
                }
                bezetR[x+N][y+N]=0;
        }
}
void threadD(void *vargp)
{
        bezetD[0+N][0+N]=1;
        bezetD[1+N][0+N]=1;
        godown(2, 0, 2);
}
void threadR(void *vargp)
{
        bezetR[0+N][0+N]=1;
        bezetR[1+N][0+N]=1;
        goright(1, 1, 2);
}
int main_random1()
{
        unsigned old_clock = clock ();
        int old_time  = time(0);
        pthread_t down,right;
        int i,j;
        for (i=0;i<=N;i++)
        {
                ZD[i]=0;
                ZR[i]=0;
                sumr2D[i]=0.0;
                sumr2R[i]=0.0;
        }
        for(i=0;i<=2*N;i++)
                for(j=0;j<=2*N;j++)
                {
                        bezetD[i][j]=0;
                        bezetR[i][j]=0;
                }
        pthread_create(&down, NULL, (void*) &threadD, NULL);
        pthread_create(&right, NULL, (void*) &threadR, NULL);
        pthread_join(down, NULL);
        pthread_join(right, NULL);
        for (i=2;i<=N;i++)
                printf("%d\t%lf\t%lf\t%lf\n",
                           i,4*(sumr2D[i]+2*sumr2R[i]),4*(ZD[i]+2*ZR[i]),
                           (sumr2D[i]+2*sumr2R[i])/(ZD[i]+2*ZR[i]));
                printf("%d seconds elapsed (%.3lf seconds CPU time)\n",
                           (int)time(0)-old_time,((double)(clock()-old_clock))/CLOCKS_PER_SEC);
        return 0;
}
```

6.2 C-Program for Statistical Enumeration including Pruning and Enrichment

```
/*
 *  enriched.c
 *  randomwalks
 *
 *  Created by Quirine Krol on 6/6/11.
 *  Copyright 2011 Utrecht University. All rights reserved.
 *
 */

#include "enriched.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#define RUNS 1000// aantal runs over
#define ITER 100000
#ifndef N
#define N 200
#endif
#define NUM_THREADS 1
#define T ITER/10 //correlatietijd
extern double drand48();
int seed;
double sumsd[RUNS];
double sumrw[RUNS];
double avgRB[N+1];
int bezet2d[2*N+1][2*N+1];
int bezet3d[2*N+1][2*N+1][2*N+1];
void floatinglotus(int i, int j, int k, int x, int y, double wN, double wPE){
        if (k==N) {
                sumrw[i]+=wPE;
                sumsd[i]+=wPE*((x-N)*(x-N)+(y-N)*(y-N));
        }
        if (k<N){
                int xl[4];
                int yl[4];
                int m=0;
                bezet2d[x][y]=1;
                int p;
                if (!(bezet2d[x+1][y]) ){xl[m]=x+1;yl[m]=y;m++;}
                if (!(bezet2d[x-1][y]) ){xl[m]=x-1;yl[m]=y;m++;}
                if (!(bezet2d[x][y+1]) ){xl[m]=x;yl[m]=y+1;m++;}
                if (!(bezet2d[x][y-1]) ){xl[m]=x;yl[m]=y-1;m++;}
                if (m>0){
                        p=floor(drand48()*m);
                        avgRB[k]+=m*wPE;
                        if (j>T){
                                if (j*wPE<.5*avgRB[k]){//pruning
                                        if (drand48() < 0.5 ){}
                                        else floatinglotus(i,j,k+1,xl[p],yl[p],wN*m,wPE*m*2);
                                }
                                else {
                                        if (j*wPE>5*avgRB[k]){//enrichment
                                                if (drand48() < 0.5 ) floatinglotus(i,j,k+1,xl[p],yl[p],wN*m,wPE*m);
                                                else {
                                                        floatinglotus(i,j,k+1,xl[p], yl[p],wN*m,wPE*m*.5);
                                                        floatinglotus(i,j,k+1,xl[p], yl[p],wN*m,wPE*m*.5);
                                                }
                                        }
                                else floatinglotus(i,j,k+1,xl[p],yl[p],wN*m,wPE*m);
                                }
                        }
                        else floatinglotus(i,j,k+1,xl[p],yl[p],wN*m,wPE*m);
                }
                bezet2d[x][y]=0;
        }
}
void flyinglotus(int i, int j, int k, int x, int y,int z, double wN, double wPE){
        if (k==N) {
                sumrw[i]+=wPE;
                sumsd[i]+=wPE*((x-N)*(x-N)+(y-N)*(y-N)+(z-N)*(z-N));
        }
        if (k<N) {
                int xl[6];
                int yl[6];
                int zl[6];
                int m=0;
                bezet3d[x][y][z]=1;
                int p;
                if (!(bezet3d[x+1][y][z]) ){xl[m]=x+1;yl[m]=y;zl[m]=z;m++;}
                if (!(bezet3d[x-1][y][z]) ){xl[m]=x-1;yl[m]=y;zl[m]=z;m++;}
                if (!(bezet3d[x][y+1][z]) ){xl[m]=x;yl[m]=y+1;zl[m]=z;m++;}
                if (!(bezet3d[x][y-1][z]) ){xl[m]=x;yl[m]=y-1;zl[m]=z;m++;}
                if (!(bezet3d[x][y][z+1]) ){xl[m]=x;yl[m]=y;zl[m]=z+1;m++;}
                if (!(bezet3d[x][y][z-1]) ){xl[m]=x;yl[m]=y;zl[m]=z-1;m++;}
                if (m>0){
                        p=floor(drand48()*m);
                        avgRB[k] += m*wPE;
                        if (j>T){
                                if (j*wPE<.5*avgRB[k]){//pruning
                                        if (drand48() < 0.5 ){}
                                        else flyinglotus(i,j,k+1,xl[p],yl[p],zl[p],wN*m,wPE*m*2);
                                }
                                else {
                                        if(j*wPE>5*avgRB[k]){//enrichment
                                                if (drand48() < 0.5 )flyinglotus(i,j,k+1,xl[p],yl[p],zl[p],wN*m,wPE*
                                                else{
                                                        flyinglotus(i,j,k+1,xl[p], yl[p],zl[p],wN*m,wPE*m*.5);
                                                        flyinglotus(i,j,k+1,xl[p], yl[p],zl[p],wN*m,wPE*m*.5);
                                                }
                                        }
                                        else flyinglotus(i,j,k+1,xl[p],yl[p],zl[p],wN*m,wPE*m);
                                }
                        }
                        else flyinglotus(i,j,k+1,xl[p],yl[p],zl[p],wN*m,wPE*m);
                }
                bezet3d[x][y][z]=0;
        }
}
void rosenbluth2D(void){
        int i,j, ii, jj,k ;
        for(ii=0;ii<2*N+1;ii++){
                for (jj=0; jj<2*N+1; jj++){
                        bezet2d[ii][jj]=0;
                }
        }
        for(i=0;i<RUNS;i++){
                sumsd[i]=0;
                sumrw[i]=0;
                for(k=0;k<N;k++)avgRB[k]=0;
                for(j=1;j<=ITER;j++)floatinglotus(i,j,0,N,N,1,1);
        }
}
void rosenbluth3D(void){
        int i,j,k, ii, jj,kk ;
        for(ii=0;ii<2*N+1;ii++){
                for (jj=0;jj<2*N+1;jj++){
                        for (kk=0; kk<2*N+1; kk++)bezet3d[ii][jj][kk]=0;
                }
        }
        for(i=0;i<RUNS;i++){
                sumsd[i]=0;
                sumrw[i]=0;
                for (k=0;k<N;k++)avgRB[k]=0;
                for(j=1;j<=ITER;j++)flyinglotus(i,j,0,N,N,N,1,1);
        }
}
int main(int argc, char* argv[])
{
        seed=time(NULL);
        srand48(seed);
        rosenbluth2D();// either 2D or 3D
    //rosenbluth3D();
        double msd_avg, msd_sig;
        msd_avg = 0;
        msd_sig = 0;
        int i;
        for (i=0; i<RUNS; i++)msd_avg+=sumsd[i]/sumrw[i];
        msd_avg /= RUNS;
        for (i=0;i<RUNS;i++)msd_sig += (sumsd[i]/sumrw[i]-msd_avg)*(sumsd[i]/sumrw[i]-msd_avg);
        msd_sig = sqrt(msd_sig)/RUNS;
        printf("{%d, %f, %f, %f, %f}, ", N, msd_avg, msd_sig, sqrt(msd_avg), msd_sig/(2*sqrt(msd_avg)));
        return 0;
}
```

6.3 C-Program for the tick and pivot algorithm

```
#include <stdio.h>
#include <math.h>
#include "rngmit.h"
#define Nmax 64
#define Nstart 36
#define Nhalf 60
#define Nstarthalf 40
#define Nstep 8
double pi;
double beta=1.0;
double x[Nmax+1],y[Nmax+1],z[Nmax+1],Nmsd[Nmax+1];
double ener;
int tacc,trej,pacc,prej;
double a=100.0,b=5.0,K=1.0;
int term=100;
int iterations=5000000;
double pstepsize=3.14,tstepsize=1.0;
FILE *tickpivotdata;
double energy(int N){
        int i,j;
        double sumstretch=0.0,sumgauss=0.0;
        double dx,dy,dz,dr,dr2;
        for (i=0;i<N;i++) {
                dx=x[i+1]-x[i];
                dy=y[i+1]-y[i];
                dz=z[i+1]-z[i];
                dr=sqrt(dx*dx+dy*dy+dz*dz);
                sumstretch+=(dr-1)*(dr-1);
        }
        for (i=0;i<=N;i++) {
                for (j=i+2;j<=N;j++) {
                        dx=x[j]-x[i];
                        dy=y[j]-y[i];
                        dz=z[j]-z[i];
                        dr2=dx*dx+dy*dy+dz*dz;
                        sumgauss+=exp(-b*dr2);
                }
        }
        return K*sumstretch+a*sumgauss;
```

```
}
pivotmovez(int iter,int N){
        int it,i,p;
        double ebef,eaft;
        double xp,yp,dx,dy;
        double alpha;
        double xb[N+1],yb[N+1];
        for (it=0;it<iter;it++){
                p=1+(N-1)*rngmit;
                alpha=1.0-2*rngmit;
                alpha*=pstepsize;
                double cosa=cos(alpha);
                double sina=sin(alpha);
                xp=x[p];
                yp=y[p];
                ebef=ener;
                for (i=p+1;i<=N;i++){
                        xb[i]=x[i];
                        yb[i]=y[i];
                        dx=x[i]-xp;
                        dy=y[i]-yp;
                        x[i]=xp+cosa*dx-sina*dy;
                        y[i]=yp+sina*dx+cosa*dy;
                }
                eaft=energy(N);
                if ((eaft<ebef)||(rngmit<exp(-beta*(eaft-ebef)))){
                        ener=eaft;
                        pacc++;
                }
                else {
                        for (i=p+1;i<=N;i++){
                                x[i]=xb[i];
                                y[i]=yb[i];
                        }
                        prej++;
                }
        }
}
pivotmovex(int iter,int N){
        int it,i,p;
        double ebef,eaft;
        double yp,zp,dy,dz;
        double alpha;
        double yb[N+1],zb[N+1];
        for (it=0;it<iter;it++){
                p=1+(N-1)*rngmit;
                alpha=1.0-2*rngmit;
                alpha*=pstepsize;
                double cosa=cos(alpha);
                double sina=sin(alpha);
                yp=y[p];
                zp=z[p];
                ebef=ener;
                for (i=p+1;i<=N;i++){
                        yb[i]=y[i];
                        zb[i]=z[i];
                        dy=y[i]-yp;
                        dz=z[i]-zp;
                        y[i]=yp+cosa*dy-sina*dz;
                        z[i]=zp+sina*dy+cosa*dz;
                }
                eaft=energy(N);
                if ((eaft<ebef)||(rngmit<exp(-beta*(eaft-ebef)))){
                        ener=eaft;
                        pacc++;
                }
                else {
                        for (i=p+1;i<=N;i++){
                                y[i]=yb[i];
                                z[i]=zb[i];
                        }
                        prej++;
                }
        }
}
pivotmovey(int iter,int N){
        int it,i,p;
        double ebef,eaft;
        double zp,xp,dz,dx;
        double alpha;
        double zb[N+1],xb[N+1];
        for (it=0;it<iter;it++){
                p=1+(N-1)*rngmit;
                alpha=1.0-2*rngmit;
                alpha*=pstepsize;
                double cosa=cos(alpha);
                double sina=sin(alpha);
                zp=z[p];
                xp=x[p];
                ebef=ener;
                for (i=p+1;i<=N;i++){
                        zb[i]=z[i];
                        xb[i]=x[i];
                        dz=z[i]-zp;
                        dx=x[i]-xp;
                        z[i]=zp+cosa*dz-sina*dx;
                        x[i]=xp+sina*dz+cosa*dx;
                }
                eaft=energy(N);
                if ((eaft<ebef)||(rngmit<exp(-beta*(eaft-ebef)))){
                        ener=eaft;
                        pacc++;
                }
                else {
                        for (i=p+1;i<=N;i++){
                                z[i]=zb[i];
                                x[i]=xb[i];
                        }
                        prej++;
                }
        }
}
tickmove(int iter,int N)
{
        int it,i;
        double dx,dy,dz;
        double ebef,eaft;
        for (it=0;it<iter;it++){
                ebef=ener;
                i=(N+1)*rngmit;
                do{
                        dx=1.0-2*rngmit;
                        dy=1.0-2*rngmit;
                        dz=1.0-2*rngmit;
                } while (dx*dx+dy*dy+dz*dz>1.0);
                dx*=tstepsize;
                dy*=tstepsize;
                dz*=tstepsize;
                x[i]+=dx;
                y[i]+=dy;
                z[i]+=dz;
                eaft=energy(N);
                if((eaft<ebef)||(rngmit<exp(-beta*(eaft-ebef)))){
                        ener=eaft;
                        tacc++;
                }
                else{
                        x[i]-=dx;
                        y[i]-=dy;
                        z[i]-=dz;
                        trej++;
                }
        }
}
init(int N){
        int i;
        for (i=0;i<=N;i++){
                x[i]=0.0;
                y[i]=i*2.0;
                z[i]=0.0;
        }
        ener=energy(N);
}
main(){
        int i,k,p,q;
        int seed;
        double sumdr2;
        int N;
        for (i=0; i<=200; i++) {
                Nmsd[i]=0;
        }
        seed=time(NULL);
        rngseed(seed);
        pi=3*acos(0.5);
        printf("%d\t%lf\t%lf\t%lf\n",Nmax,a,b,K);
        for (N=Nstart; N<=Nhalf; N+=Nstep){
                init(N);
                for (k=0;k<term;k++){
                        tickmove(2,N);
                        for (q=1; q<=10; q++){
                                pivotmovex(1,N);
                                pivotmovey(1,N);
                                pivotmovez(1,N);
                        }
                }
                sumdr2=0;
                tacc=trej=pacc=prej=0;
                for (k=0;k<iterations;k++){
                        tickmove(2,N);
                        for (q=1; q<=10; q++){
                                pivotmovex(1,N);
                                pivotmovey(1,N);
                                pivotmovez(1,N);
                        }
                        sumdr2+=(x[N]-x[0])*(x[N]-x[0])+(y[N]-y[0])*(y[N]-y[0])+(z[N]-z[0])*(z[N]-z[0]);
                }
                Nmsd[N]=sumdr2/iterations;
                printf("%d\t%lf\n",N,Nmsd[N]);
        }
        for (N=Nstarthalf; N<=Nmax; N+=Nstep){
                init(N);
                for (k=0;k<term;k++){
                        tickmove(2,N);
                        for (q=1; q<=10; q++){
                                pivotmovex(1,N);
                                pivotmovey(1,N);
                                pivotmovez(1,N);
                        }
                }
                sumdr2=0;
                tacc=trej=pacc=prej=0;
                for (k=0;k<iterations;k++){
                        tickmove(2,N);
                        for (q=1; q<=10; q++){
                                pivotmovex(1,N);
                                pivotmovey(1,N);
                                pivotmovez(1,N);
                        }
                        sumdr2+=(x[N]-x[0])*(x[N]-x[0])+(y[N]-y[0])*(y[N]-y[0])+(z[N]-z[0])*(z[N]-z[0]);
                }
                Nmsd[N]=sumdr2/iterations;
                printf("%d\t%lf\n",N,Nmsd[N]);
        }
        char filename[Nmax+10];
        sprintf(filename,"%g m%iGauss2a%gb%gK%gB%g.dat",1.0*iterations/1000000,Nmax,a,b,K,beta);
        tickpivotdata= fopen(filename,"w");
        for (p=1; p<=Nmax+1; p++){
                fprintf(tickpivotdata,"%d\t%lf\n",p,Nmsd[p]);
        }
```

```
}
```