Monte-Carlo Simulations of Self-Avoiding Random Walks in 2 and 3 Dimensions Using the Pivot Method

Zhixuan Li

December 5, 2024

1 Introduction

The self-avoiding walk (SAW) is a subset of the random walk where each site cannot be visited more than once. The properties of SAWs allow it to be used as a model for linear polymer molecules in good solvents [14]. Specifically, despite seeming to have many levels of abstraction from the polymer model, such as being on a lattice that does not have the same tetrahedral bond angles, binding energies, and different monomer-monomer interaction potentials, the SAW belongs to the same *universality class* as polymer molecules in good solvents. For models in the same universality class, near a critical point, the leading asymptotic behavior is the same for the universal quantities of those systems [14]. Therefore, it is shown that the model of SAWs on a lattice can be used to study universal characteristics of the polymer chain, such as the relationship of the mean squared end-to-end distance to the path length. It is sufficient to choose any model of the same universality class to compute universal quantities, so for the purpose of this study, as well as many past studies on the same subject, the SAW on a square (cubic) lattice is chosen as the model of interest due to its simplicity.

As the problem scales with dimensionality, it may not be feasible to solve the model analytically with an exact calculation, which calls for the use of further approximations (such as renormalization group analysis) and numerical simulations (such as exact enumeration studies and Monte Carlo studies)[14]. In this paper, we shall examine Monte Carlo methods for simulating self-avoiding random walks on a square lattice with dimension $d = 2$ and $d = 3$. The results of the simulation would then be used to estimate various universal critical exponents for SAWs.

2 Numerical Formulation for SAWs

2.1 Introductory Notation

This paper will follow the notation used by Madras and Sokal in the analysis of the self-avoiding random walk, and further, the pivot algorithm [8]. Start with a d-dimensional

cubic lattice $\mathscr{L}$ in $\mathbb{Z}^d$. An n-step self-avoiding walk on $\mathscr{L}$ is a sequence of distinct sites in $\omega_0, \omega_1 \ldots \omega_N$ such that each site is a nearest neighbor of the previous site. For the purposes of this analysis, the lattice is restricted to $\mathbb{Z}^d$, although a similar analysis can be done for other regular lattices. Furthermore, for this analysis, the first site ω_0 is taken to be the origin.

Denote the set of all SAWs of length N on the d-dimensional lattice $\mathbb{Z}^d$ to be $\mathcal{S}_N$. Let the subset of such a walk of length N that ends at lattice site x be $\mathcal{S}_N(x)$. The set $\mathcal{S}_N$ can be expressed as the sum over all x of $\mathcal{S}_N(x)$. Now denote the cardinality of each $\mathcal{S}_N(x)$ set to be $c_N(x)$.

2.2 Critical exponents

Now, relevant quantities can be expressed using the above notation. The first pair of values of interest are the mean-squared end-to-end distance of the walk and the mean-squared radius of gyration. Since the walk begins at $(0, 0)$ and ends at x, the end-to-end distance of walks in $\mathcal{S}_N(x)$ is $|x|^2$.

Now, the mean-squared distance for a SAW of length N can be expressed as

$$\langle \omega^2 \rangle \equiv \frac{1}{c_N} \sum |x|^2 c_N(x) \tag{1}$$

The mean-squared radius of gyration is related to the mean-squared distance. For a walk, the radius of gyration is mean-squared distance of every vertex to its center of mass. For a physical object, such as a polymer, this would be the distance between the object's point mass to its axis of rotation. This is shown in [8] to be

$$\langle S_N^2 \rangle \equiv \frac{1}{c_N} \sum_{\omega \in \mathcal{S}_N} \left(\frac{1}{N+1} \sum_{i=0}^{N} \omega_i^2 - \left(\sum_{i=0}^{N} \omega_i\right)^2\right) \tag{2}$$

These two quantities are also expressed as $\langle R_e^2 \rangle$ for mean-squared distance and $\langle R_g^2 \rangle$ for mean-squared radius of gyration in other notations. These quantities are shown through Renormalization Group analysis to have the asymptotic behavior as $N \to \infty$, up to leading order: [6]

$$\langle \omega^2 \rangle, \langle S_N^2 \rangle \propto N^{2\nu} \tag{3}$$

where ν is a universal critical exponent for d dimensions. In $d = 2$, this exponent has been calculated exactly to be $\nu = 3/4$ [10]. For higher dimensions, simulations are used to estimate the value of ν, with varying methods and implementations [1] [9] [6].

The second critical exponent is the connective constant of the lattice, μ, also referred to as the effective coordination number. It is shown that the following limits are equal [5] :

$$\mu = \lim_{N \to \infty} c_N^{1/N} = \lim_{\substack{N \to \infty \\ N = x \mod 2}} c_N(x)^{1/N} \tag{4}$$

It is also hypothesized that [8]

$$c_N \propto \mu^N N^{\gamma-1} \tag{5}$$

where γ is a critical exponent universal to d-dimensional lattices.

3 Monte Carlo Algorithms for SAWs

3.1 Static Monte Carlo Approaches

The most straightforward method of generating samples of self-avoiding random walks is to generate each walk independently step-by-step, until such a walk reaches the desired length. This is referred to as Elementary Simple Sampling (ESS) [9]. The algorithm is as follows:

1. Set the first step ω_0 to be the origin, and let that be the current step. Set $i = 0$.
2. Randomly choose a nearest neighbor of the current step and let that site be step ω_{i+1}, add (ω_i, ω_{i+1}) to the walk.
3. For all previous steps in the walk, check if there is an overlap: $(\omega_i, \omega_{i+1}) = (\omega_j, \omega_{j+1})$ for $j = 0, 1, \ldots i-1$. If the new step overlaps with some previous step, then discard the walk and return to step 1. Else, increase i by one, set the new step to be the current step, and repeat step 2.

To show that the above algorithm generates walks in $\mathcal{S}_N(x)$ with uniform probability, consider the process in each loop to be the generation of an ordinary random walk of N steps in $\mathscr{L}$. The process of checking at the end of each step is done to save time so that the final steps of a walk that is already intersecting won't be computed. If there is no check for each iteration, then a uniform distribution of random walks is generated. Thus, by keeping only the samples that are self-avoiding, a uniform distribution of SAWs on $\mathcal{S}_N(x)$ can be generated.

However, the time complexity of this algorithm grows very quickly with large N. For any random walk, the probability that it is a SAW is $\frac{c_N}{2d^N}$, so the expected number of trials to find such a walk is $\frac{2d^N}{c_N}$, which results in a time complexity of at least $O(2^N)$. There are some possible optimizations of the ESS algorithm, including removing the site ω_{i-2} from the possible nearest neighbors when choosing ω_i, however there are none that would improve upon the $O(2^N)$ time complexity [9].

3.2 Dynamic Algorithms and the Pivot Method

The methods described above generate samples that are statistically independent from one another, and are thus classified as *static*. Another way of producing a set of samples from $\mathcal{S}_N(x)$ are *dynamic* methods, which generate a sequence of correlated samples from

a Markov Chain process. These processes modify previously generated walks to produce a canonical ensemble of SAWs. The pivot method is one such algorithm.

The pivot algorithm is performed as follows:

1. Starting from a previously-generated SAW sample, select a random point k along the walk as a pivot point, with $0 \leq k \leq N-1$.

2. Select a random symmetry group transformation of the lattice as the pivot operation.

3. Perform a pivot operation to the latter portion of the SAW, starting from the pivot point ω_{k+1} and ending at ω_N.

4. Check if the resulting walk is self avoiding by checking for any repeated ω_i. If it is not self-avoiding, discard the current pivoted walk and start again at step 2. If it is self-avoiding, add the new pivoted walk to the generated set of SAWs.

For the pivot operation, the set of valid transformations, G, need to ensure ergodicity, such that any SAW can be transformed with a sequence of transformations from G into any other SAW. Furthermore, transformations in G should also be orthogonal, and have the same probability to be chosen as its own inverse operation. Thus, for $d = 2$, a valid group G is the dihedral group D_4, which involves the Identity, 90° rotations, and reflections about the axes and the diagonals. For $d = 3$, a valid group G is the octahedral group O_h, which includes rotations, reflections, and rotoflections of various angles, with a total of 48 transformations [8]. The proof of the validity of these transformations in generating a uniformly distributed set of samples is discussed in [8], and shown below.

4 Validity of the Pivot Algorithm

Since dynamic algorithms generate samples that are not statistically independent, it is crucial to show that the Markov chain, when run for a sufficient amount of time, produces a stationary distribution that converges to the desired distribution, which in this case is the uniform distribution of all self-avoiding walks.

Suppose the Markov chain has state space S, the collection of all possible states that it can occupy. Let the transition probability matrix P have elements $p_{i\to j} = p_{ij}$. We want to show that $S = \mathcal{S}_N(x)$ and P approaches the distribution $\pi = 1/c_N$, as well as ergodicity, which implies that each state can eventually be reached from any other state. More formally, the Markov chain should satisfy the following to be considered ergodic: for each p_{ij}, there exists a number of steps n for which the n-step transition probability $p_{ij}^n > 0$ [8].

Section 3.5 in Madras and Sokal prove the ergodicity for different variants of the pivot algorithm, essentially by showing that any N-step SAW can be transformed into a

straight rod in at most $2N-1$ pivot transformations, where the pivot transformations are selected from a distribution with non-zero probability for the following [9]:

($\pm 90°$ rotations OR both diagonal reflections) AND ($180°$ rotation OR both axis reflections)

For $\mathbb{Z}^d$ in d dimensions, the ergodic symmetry group G can be constructed as follows: an element $g \in G$ has columns which are permutations of the unit vectors in $\mathbb{Z}^d$, and are thus orthogonal matrices. Thus, the cardinality of G is $(d!)2^d$. This set can be condensed to have smaller cardinality and still be ergodic, as shown in the 2-dimensional case, but ergodicity can be ensured as long as non-zero probabilities are assigned to all elements in G.

A Markov Chain process that satisfies the above conditions would then be able to produce a valid set of samples for the probability distribution π, after generating for a period of time so that samples are effectively no longer correlated to each other. To characterize the time that it takes for such a system to reach equilibrium, introduce the *exponential autocorrelation time*, τ_{exp}, and the *integrated autocorrelation time* for an observable A, $\tau_{int,A}$ [8]. τ_{exp} determines the number of samples to discard at the beginning of the run, when the simulation has not reached equilibrium. $\tau_{int,A}$ is related to the variance of the measured variable $\langle A \rangle$ due to the statistical dependence of the Markov Chain samples, specifically that the variance is a factor of $2\tau_{int,A}$ larger than that for independent samples [8]. Madras and Sokal [8] analyze these variables in-depth for d=2, as they are related to the number of iterations needed for a simulation to have small systematic error as well as statistical error, which further relates to the efficiency of the pivot algorithm.

5 Efficiency of the Pivot Algorithm

Using the above definitions, the pivot algorithm can be used to generate enough samples to constitute an "effectively independent" observation. This section then expands on the analysis of the algorithm's time complexity, which turns out to be $O(N \log N)$, which is vastly better than the $O(2^N)$ exponential time constraint of naive approaches, and comes very close to the lower bound for generating random walks, which is $O(N)$.

The time complexity of the pivot algorithm can be divided into two steps: First, the time it takes to perform a single pivot transformation on an existing walk and check if the walk is self-avoiding. Suppose each iteration has a time complexity of N^q. Second, the amount of pivots it takes to find an acceptable SAW. The second time is especially of interest, since it scales the time of each loop proportionally, and this is referred to as the acceptance fraction f [8]. Madras and Sokal present a heuristic argument that the bound of the acceptance fraction is $f \sim N^{-p}$, which p being a critical exponent that can be evaluated for different implementations of the algorithm. This implies that every one in N^p pivots will result in an acceptance. Furthermore, the nature of the pivot algorithm tends to move large pieces of the walk at every iteration (as compared to other algorithms that only modify a few bonds), so after a few ($O(1)$)

accepted pivots, the walk would have reached an "essentially new" configuration [8]. Thus, the autocorrelation time also increases with $\sim N^p$. In their in-depth analysis of the integrated auto-correlation time for global variables, Madras and Slade show that in face there is another $\log N$ term, resulting in $\tau_{int,A} \sim N^p \log N$. Thus, the total time complexity amounts to $N^q N^p \log N = N^{p+q} \log N$. Madras and Slade [9] provide an in-depth analysis that show that $N^{p+q} = N$, by re-formulating the amount of work needed in each iteration to achieve the tighter bound on time complexity of $O(N^{1-p})$. This time complexity is achieved by improving the checking step of the pivot algorithm to apply the transformation to each site after the pivot site sequentially, and check for repeated sites after each transformation. By discarding walks that have sites close to the pivot already overlapping without computing the entire pivot transformation, the $O(N^{1-p})$ time complexity can be acheived.

Numerical estimations of p yield $p \simeq 0.19$ for $d = 2$ and $p \simeq 0.11$ for $d = 3$ [8]. This exponent is also measured and examined in the experiment, so as to evaluate the correctness in the acceptance algorithm, as well as the efficiency of the code.

6 Implementation and Results for 2D SAW

The complete code is made available on GitHub, in the form of a Jupyter notebook, with the first section being 2-dimensional SAWs [7]. The implementation follow the notation above, with code optimization and organization aided by GPT4 [11]. First, as a preliminary study, a naive ESS algorithm is written to generate SAWs step by step, and tested on small N samples. An example generation of $N = 20$ is shown below. This naive approached proved inefficient for larger N values, and is virtually unusable for producing large amounts of samples beyond $N = 100$. However, this method does provide a way to initialize a SAW that is ensured to be selected uniformly at random, which is taken into consideration when implementing the Markov Chain pivot algorithm.

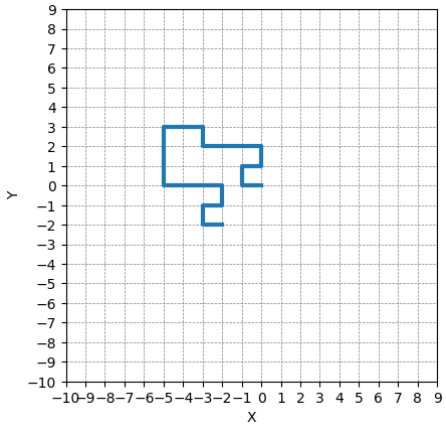

Figure 1: Example of Naive generation ESS algorithm with $N = 20$

Then, the pivot algorithm was implemented, as well as calculations for ω^2 and S^2. Examples with small N are produced, with one example generation below. An excerpt of the code is attached in Appendix A.

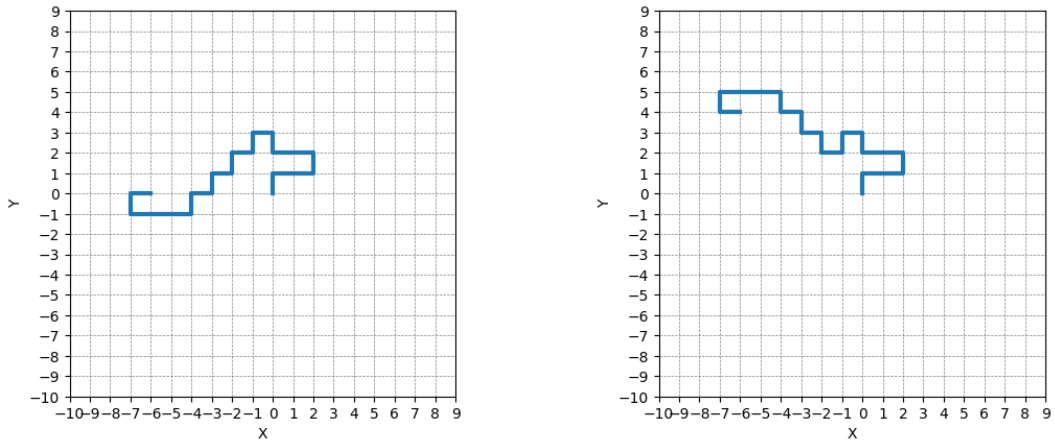

Figure 2: Before and after a pivot transformation for $N = 20$, with the pivot point being $(-2, 2)$, and the transformation being x-axis reflection.

The pivot algorithm for d=2 is implemented in python following the pseudo-code in part 3.2, with the following implementation-specific decisions:

- The algorithm is implemented in python, with optimizations to take advantage of multi-core processing and numpy vector operations. However, further optimizations could be achieved by going to cython, or other languages that have finer-grained control over memory access and function operations.
- For each SAW, the path ω_i is stored as a 2-d numpy array, and pivots are applied to the path by multiplying the corresponding matrices. Further optimization to the data structure for storing walks can be implemented to decrease the access time for each site as well as the time to compute the pivot, as shown in [8] and [1].
- The optimization to the checking step as shown in section 5 is not implemented, as the current SAW-checking code in python requires more computing time than $O(1)$ due to the inefficiency of numpy built-in functions, so adding it to the computations for each site significantly increases the computing time.
- A Deque buffer data structure is chosen to hold the generated SAWs, with a buffer capacity of around $N/10$. Although the actual implementation only iterates upon the most recently generated walk, so there is little need to store all walks, the walks inside buffer was used to examine the autocorrelation time and the actual effect of the pivots.

Furthermore, when generating a collection of samples with the pivot algorithm, the method of initializing the first SAW is taken into consideration. Two different methods of initialization are implemented, following the method in [8]:

1. Equilibrium start, where a SAW is generated with the naive ESS method, and used as the starting walk. Due to the inefficiency of the ESS algorithm scaling with N, this was only used for small-N sample generation.

2. Thermalization, where the initial SAW is chosen non-randomly, and enough pivot iterations are performed, so that the distribution of the generated samples approach the stationary distribution. In this implementation, the starting SAW is a straight-line walk of length N. Madras and Sokal showed that, for $N < 10000$, a thermalization time T of 20000 is sufficient to eliminate initialization bias [8].

For small N ($N = 15$ and $N = 20$), the results for ω^2 and S^2 are calculated and compared to known exact values from direct enumeration [4][3]. The measured values are produced with 10^7 samples, with the beginning 10^6 samples discarded to account for thermalization. The ω^2 and S^2 value for each walk is calculated, and the mean with 95% confidence limits are shown in 1.

Table 1: Measured and Actual $\langle\omega^2\rangle$, $\langle S^2\rangle$ values for $N = 15$ and $N = 20$.

N	Parameter	Measured Value	Actual Value
$N = 15$	$\langle\omega^2_{15}\rangle$	47.203 ± 0.0603	47.2177
	$\langle S^2_{15}\rangle$	6.7815 ± 0.0047	6.7843
$N = 20$	$\langle\omega^2_{20}\rangle$	72.3169 ± 0.0935	72.0765
	$\langle S^2_{20}\rangle$	10.2628 ± 0.0072	-

6.1 Estimating critical exponents

With the implementation of the pivot algorithm verified through comparison with calculated values, the model can now be used to generate data for larger values of N, in order to compute estimations of the critical exponents discussed above. The N value was varied from 20 to 2400, and 10^6 samples were produced for each set, with the first 3×10^5 values discarded to account for thermalization. The $\langle\omega^2\rangle$ and $\langle S^2\rangle$ values are calculated, as well as the ratio between the two, which should be constant. The acceptance fraction, f, is also recorded by counting the number of fails until each new walk is accepted and added to the set. Finally, the efficiency of the sample generation at each N was recorded in units of iterations per second.

Using the data, we are able to find the critical exponents ν and p. As discusses above, $\langle\omega^2\rangle, \langle S^2\rangle \sim AN^{2\nu}$, and $f \sim N^{-}p$. Both appear in an exponential relationship, so by performing least-squared regression on the log-log plots of the relevant observables,

Table 2: $\langle\omega^2\rangle$, $\langle S^2\rangle$, and acceptance fraction f for different Ns in $d = 2$

N	$\langle\omega^2\rangle$	$\langle S^2\rangle$	$\langle S^2\rangle/\langle\omega^2\rangle$	f	Time (iter/s)
20	75.3901 ± 0.1081	10.5753 ± 0.0084	0.140275	0.5487	10083.16
50	292.400 ± 0.4359	40.4726 ± 0.0340	0.138416	0.4583	6702.39
200	2310.79 ± 3.5072	319.462 ± 0.2784	0.138248	0.3480	1986.52
400	6532.65 ± 9.9466	903.556 ± 0.7929	0.138314	0.3040	956.10
600	11949.0 ± 18.1545	1655.98 ± 1.4477	0.138588	0.2807	560.34
800	18424.8 ± 28.0875	2542.90 ± 2.2453	0.138015	0.2644	510.87
1200	33818.9 ± 51.5435	4676.77 ± 4.1211	0.138289	0.2448	286.76
1600	52228.9 ± 79.4397	7209.94 ± 6.3461	0.138045	0.2318	203.05
2400	96211.0 ± 146.5231	13310.9 ± 11.7145	0.138352	0.2145	151.29

an estimate for the exponents can be achieved. The log-log plot and best fit lines for the following exponents are shown below: $\langle\omega^2\rangle$ in 3, $\langle S^2\rangle$ in 4, and f. in 5. All are fit to a function of the form AN^k, where k is the exponent of interest, which becomes $\log A + k \log N$ in the linear regression.

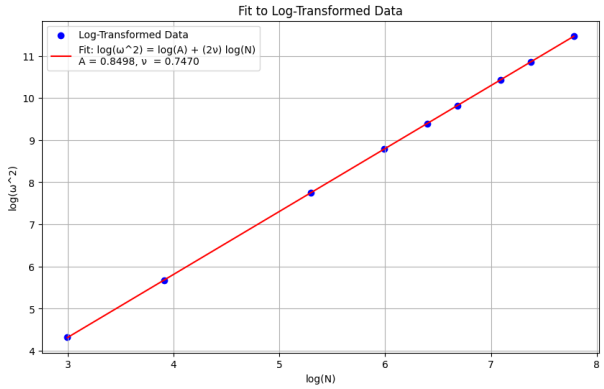

Figure 3: Least-squares regressions of logarithms of $\langle\omega^2\rangle$ against $\log N$

The same results are recorded in 3. Here, s^2 is the mean squared error from the regression line, and should take on a χ^2 distribution with (number of samples - 2) degrees of freedom. They are sufficiently small for all three exponents to indicate a good fit. For reference, Madras and Sokal obtained results of $\nu = 0.7503$ and $A = 0.7673$ for $\langle\omega^2\rangle$ and $\langle S^2\rangle$, and $p = 0.1918$ and $A = 0.9511$ for f, using more date simulated for larger N values (up to 10000) [8].

Overall, when comparing the data obtained from this implementation of the pivot algorithm and the results in [8], it can be seen that in general, for the estimations of the observables, both $\langle\omega^2\rangle$ and $\langle S^2\rangle$ for all N values are slightly larger than Madras and Sokal's results, by around 4%. When looking at $N = 20$, the measurements are also larger than the actual calculated values. This error seems to be systematic, and can

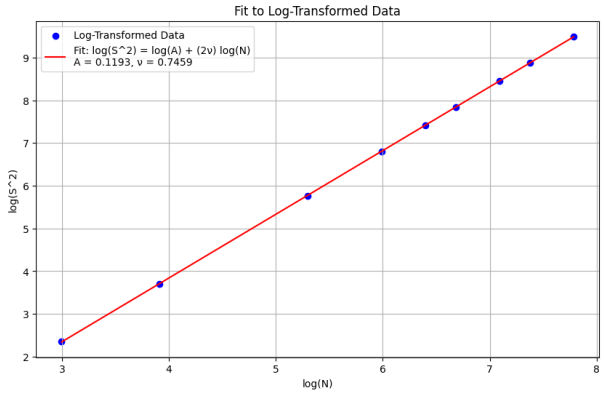

Figure 4: Least-squares regressions of logarithms of $\langle S^2 \rangle$ against $\log N$

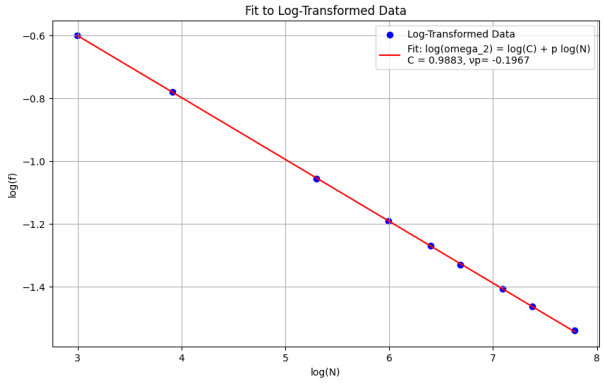

Figure 5: Least-squares regressions of logarithms of f against $\log N$

Table 3: Estimates for critical exponents based on observables.

Exponent	Value
$\langle \omega_N^2 \rangle$	$\nu = 0.7470$
	$A = 0.8498$
	$s^2 = 1.63$
$\langle S_N^2 \rangle$	$\nu = 0.7459$
	$A = 0.1193$
	$s^2 = 7.4 \times 10^{-5}$
f	$p = 0.1967$
	$A = 0.9883$
	$s^2 = 3.12 \times 10^{-6}$

be attributed to several factors: First, the initialization scheme used by Madras and Sokal is an optimized equilibrium start algorithm, whereas the data here was produced with a straight-line start. If the initializing walk is not sufficiently thermalized, ω^2 and

S^2 could possibly be biased to be higher than actual. Furthermore, another possible optimization is hinted at in [9], where multiple pivots can be done in succession before a walk is accepted into the sample, effectively making the samples less correlated. One item to note is that another optimization method was attempted for this issue, but did not yield significant results, namely to choose a random walk from the set of generated samples every time instead of using the previously generated walk. However, This seems to further skew the distribution of samples and produced much higher error for the resulting critical exponents. This systematic error reflects especially on the estimation for A_ω, as it is $\sim 8\%$ larger than Madras and Sokal's predicted values. The difference in the estimates for the actual exponents ν and p, however, are smaller, at 0.4% and 2.5%. In conclusion, possible improvements would rely on more generated samples as well as a better initialization schema.

7 Implementation and Results for 3D SAW

The pivot algorithm in $d = 3$ was implemented in a similar way as the 2D algorithm above. An example of a generated 3-d SAW is shown in 6.For the symmetry group transformations, the 48 matrices of the octahedral group O_h was generated with the aid of GPT4 [11].

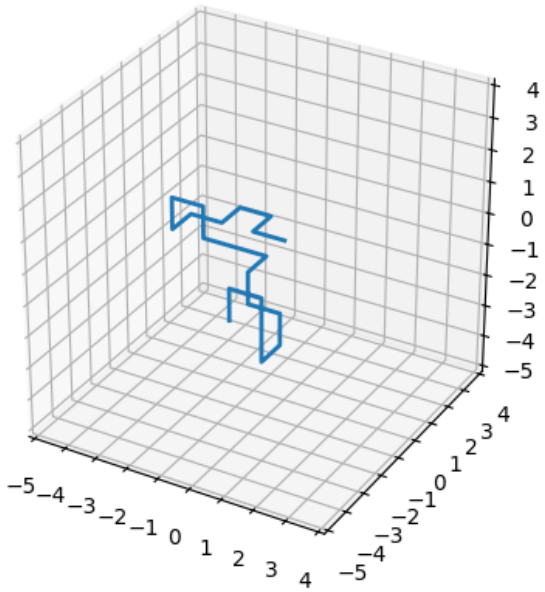

Figure 6: Visualization of a generated 3D SAW with $N = 20$

For small N ($N = 15$ and $N = 20$), the results for $\langle \omega^2 \rangle$ are calculated and compared to known exact values from direct enumeration [13][12], with results shown in 4. The estimations were performed with 10^6 samples and 10^5 discards, with a equilibrium

initialization. The measured results were, however, still slightly biased to be larger than calculated by a systematic error.

Table 4: Measured and Actual $\langle\omega^2\rangle$, $\langle S^2\rangle$ values for $N = 15, 20$ in $d = 3$.

N	Parameter	Measured Value	Actual Value
$N = 15$	$\langle\omega^2_{15}\rangle$	27.980 ± 0.0362	27.393
	$\langle S^2_{20}\rangle$	4.4654 ± 0.00318	-
$N = 20$	$\langle\omega^2_{20}\rangle$	39.682 ± 0.0524	38.741
	$\langle S^2_{20}\rangle$	6.2929 ± 0.00455	-

7.1 Estimating critical exponents

The same simulations were performed with N varying from 20 to 2400, for a sample size of 10^6 and a discard size of 3×10^5. The $\langle\omega^2\rangle$ and $\langle S^2\rangle$, and f values were recorded in 5. As expected for higher dimensions, the acceptance ratio for a give N was higher, and the overall runtime of each iteration was also around 10% faster than $d = 2$, despite the added computational complexity.

Table 5: $\langle\omega^2\rangle$, $\langle S^2\rangle$, and acceptance fraction f for different Ns in $d = 3$

N	$\langle\omega^2\rangle$	$\langle S^2\rangle$	$\langle S^2\rangle/\langle\omega^2\rangle$	f	Time (iter/s)
20	39.5803 ± 0.0594	6.27507 ± 0.00527	0.140275	0.7249	12000.02
50	119.387 ± 0.1878	18.7768 ± 0.0171	0.138416	0.6557	7670.61
200	619.795 ± 1.0058	98.0341 ± 0.0940	0.138248	0.5630	2938.23
400	1420.92 ± 2.3255	224.318 ± 0.2183	0.138314	0.5227	1552.33
600	2286.49 ± 3.7529	361.709 ± 0.3549	0.138588	0.4997	1155.81
800	3202.89 ± 5.2610	508.298 ± 0.5012	0.138015	0.4843	755.33
1200	5148.69 ± 8.4868	817.469 ± 0.8064	0.138289	0.4627	546.48
1600	7250.81 ± 11.966	1153.09 ± 1.1400	0.138045	0.4491	442.90
2400	11659.8 ± 19.336	1853.98 ± 1.8367	0.138352	0.4297	295.93

A least-squares regression line was fit to the log-log plots to find the critical exponents. The plots, like the 2D case, indicate a very straight fit, and thus are omitted from this paper, and can be found in the original Python notebook. The results from the fit are shown in 6. Comparing with previous studies, the estimations for ν range from 0.5877 ± 0.0006 [6] to 0.592 ± 0.002 [12]. The predicted acceptance fraction from Madras and Sokal is 0.1069 ± 0.0009, which puts this estimation at being around 2% larger, similar to the 2D case [8].

Table 6: Estimates for critical exponents based on observables for $d = 3$.

Exponent	Value
$\langle \omega_N^2 \rangle$	$\nu = 0.5936$
	$A = 1.1435$
	$s^2 = 1.635$
$\langle S_N^2 \rangle$	$\nu = 0.5943$
	$A = 0.0.1796$
	$s^2 = 3.07 \times 10^{-5}$
f	$p = 0.1093$
	$A = 1.0054$
	$s^2 = 3.32 \times 10^{-7}$

8 Possible Algorithmic Improvements

As noted above, for the measurements of the observables, there seems to be a systematic bias that results in a slightly skewed distribution towards larger $\langle \omega^2 \rangle$ and $\langle S^2 \rangle$ values. As discussed in Section 6, this can be attributed to an insufficient thermalization at the beginning of each simulation. Thus, possible improvements include implementing better initialization methods, such as the "dimerization" approach in [8], and generating a larger sample size to account for the autocorrelation time.

Furthermore, the speed of the algorithm can be improved, as the current Python code has not fully reached the $O(N \log N)$ limit. For one, Python functions have varying amounts of optimization, and although vector operations are fast through numpy, other operations, such as loops, have much higher overhead for time and memory usage, as compared to a language with tighter memory access controls like C. Structurally, it is possible to retain the speed of the vector operations, which in numpy are implemented through pre-compiled C code, and speed up the rest of the code, by switching to languages such as C [2]. Aside from structural improvements, more algorithmic improvements can be made to optimize the time for each pivot operation, such as implementing an efficient site-checking algorithm that can be executed after pivoting each vertex individually, following the analysis discussed above [9]. Different data structures can be used to allow for quick access for sites in a walk, such as using a binary tree data structure to store sequences of sites and walks [1]. Runtime improvements such as these can lead to simulations of much larger samples of data at much higher steps, improving the systematic errors seen in the current study.

9 Application of SAWs

Due to being in the same universality class, the universal critical exponents estimated by simulations such as this study are exactly equal to the critical exponents for problems

such as linear polymer chains in good solvents, with the only error being the error in the approximation of the experiment [14]. Similar studies of SAWs in 2 and 3 dimensions have been conducted, on much larger scales ($N = 8 \times 10^4$) [6]. The results have then been applied in many studies of linear polymers. For example, previous theories on long-chain polymer molecules in dilute solutions predict that the interpenetration ratio Ψ of polymers to be related to N with a universal critical exponent. However, Monte Carlo simulations using SAWs showed that the behavior of Ψ as a function of N is in fact not the same as predicted, which implies that the relationship is non-universal [14].

In addition, the pivot algorithm can also be adapted to models other than the linear polymers. For example, in modeling polymers with nearest-neighbor attractions, or generating branched polymers[14]. The increased complexity of the topics of interest also call for the need of ever more advanced and efficient algorithms to perform improved simulations.

References

[1] N. Clisby. Efficient implementation of the pivot algorithm for self-avoiding walks. *Journal of Statistical Physics*, 140(2):349–392, May 2010.

[2] T. N. Community. Numpy user guide, 2024. Accessed: 2024-12-03.

[3] P. Grassberger. The critical behaviour of two-dimensional self-avoiding random walks. *Z. Physik B - Condensed Matte*, 48:255–260, may 1982.

[4] A. J. Guttmann. On the critical behaviour of self-avoiding walks. *Journal of Physics A: Mathematical and General*, 20(7):1839, may 1987.

[5] J. M. Hammersley. Percolation processes: Ii. the connective constant. *Mathematical Proceedings of the Cambridge Philosophical Society*, 53(3):642–645, 1957.

[6] B. Li, N. Madras, and A. D. Sokal. Critical exponents, hyperscaling, and universal amplitude ratios for two- and three-dimensional self-avoiding walks. *Journal of Statistical Physics*, 80(3–4):661–754, Aug. 1995.

[7] Z. Li. SAW-MC, Dec. 2024.

[8] N. Madras and A. D. Sokal. The pivot algorithm: A highly efficient monte carlo method for the self-avoiding walk. *Journal of Statistical Physics*, 50:109–186, 1988.

[9] G. S. Neal Madras. *The Self-Avoiding Walk*. Birkhäuser, Boston, MA, 1996.

[10] B. Nienhuis. Exact critical point and critical exponents of O(n) models in two dimensions. *Phys. Rev. Lett.*, 49:1062–1065, Oct 1982.

[11] OpenAI. Gpt4. *Online conversation with GPT4 large language model*, 2024. *Retrieved from OpenAI's ChatGPT on December 2024*.

[12] D. C. Rapaport. On three-dimensional self-avoiding walks. *Journal of Physics A: Mathematical and General*, 18(1):113, jan 1985.

[13] R. D. Schram, G. T. Barkema, and R. H. Bisseling. Exact enumeration of self-avoiding walks. *Journal of Statistical Mechanics: Theory and Experiment*, 2011(06):P06019, June 2011.

[14] A. D. Sokal. Monte carlo methods for the self-avoiding walk, 1994.

Appendices

A Python Implementation for the pivot algorithm

The algorithm implementation here is condensed to leave out less relevant lines to best show the implementation. The full code can be found on GitHub [7].

```
class saw2d():
    def pivot(self):
        pivot_index = random.randint(1, len(self.walked_path) - 1)
        pivot_point = self.walked_path[pivot_index]
        transformation = random.choice(self.possible_trasforms)

        new_path = np.empty_like(self.walked_path)
        new_path[:pivot_index + 1] = self.walked_path[:pivot_index +
   1]

        vectors = self.walked_path[pivot_index+1:] - pivot_point
        transformed_vecs = vectors @ transformation.T
        new_points = transformed_vecs + pivot_point
        new_path[pivot_index+1:] = new_points

        if (len(new_path) == len(set(tuple(p) for p in new_path))):
          # print("pivot success")
          self.walked_path = new_path
          self.current_pos = self.walked_path[-1]
          return True
        else:
          # print("pivot failed")
          return False

def generate_samples(n_samples, N, buffer_size, init="rod"):
  saw_list = deque(maxlen=buffer_size)
  omega2_list, s2_list, ps_list = []

  if init == "rod":
    saw_initial = saw2d()
    saw_initial.initial_walk(length=N)
  elif init == "walk":
```

```
    saw_initial = saw2d()
    res = saw_initial.take_steps(N)
    while not res:
      saw_initial = saw2d()
      res = saw_initial.take_steps(N)
  saw_list.append(saw_initial.walked_path)

  for i in trange(n_samples):
    saw_pwalk_prev = saw_list[-1]
    saw_curr = saw2d()
    saw_curr.set_curr_walk(saw_pwalk_prev)
    failed_count =0

    success = saw_curr.pivot()
    while not success:
      failed_count += 1
      success = saw_curr.pivot()

    saw_list.append(saw_curr.walked_path)
    omega2_list.append(saw_curr.get_omega2())
    s2_list.append(saw_curr.get_S2())
    ps_list.append(failed_count)

    i += 1
```