Self Avoiding Walks

PHY-3610-1: Statistical Mechanics
Instructor: Prof. Bikram Phookun
Graduate Assistant: Philip Cherian

Kasturi[*] and Sanjana Gupta[†]
Ashoka University, India
(Dated: 13th December 2023)

Self-avoiding walks (SAWs) are a concept primarily used in mathematics and physics, particularly in the study of polymer chains, lattice models, and statistical mechanics. They are paths or trajectories on a lattice grid where a walker moves from one lattice point to another without revisiting any previously visited point. This project delves into the behavior of SAWs on 1D and 2D square lattices employing computational simulations and analytical methods. The study's objective is to investigate various algorithms used for simulating SAWs and to conduct comparative analyses among them.

Keywords: Random Walks, Self-Avoiding Walks, Polymers, Rosenbluth, Pivot

I. INTRODUCTION:

In the realm of mathematics, the concept of self-avoiding walks unfolds as a captivating exploration within the domains of combinatorics and probability. These walks represent sequences of movements across a lattice or grid, where each step is meticulously orchestrated to ensure that no point is revisited during the journey.

Imagine you're on a grid, like a chessboard, and you take steps in any direction—up, down, left, or right—but you can't step onto a square you've already visited. The sequence of these steps without retracing your path is a self-avoiding walk.

While this mathematical construct may seem abstract, its implications reverberate across diverse disciplines, including polymer science, computer science, and statistical mechanics. Self-avoiding walks find utility in modeling polymer behavior, elucidating random processes, and unraveling connectivity dilemmas within networks, among other applications. It is noteworthy, however, that the computational complexity associated with identifying self-avoiding walks escalates dramatically as the grid size or dimensions increase.To delve deeper into the intricacies of self-avoiding walks, we commence our exploration by scrutinizing the more familiar concept of a random walk.

A. One-Dimensional Random Walk

A one-dimensional random walk is a mathematical model that describes the path of a walker along a straight line (usually represented by integers) in a series of discrete steps. We assume, at each step, that the walker

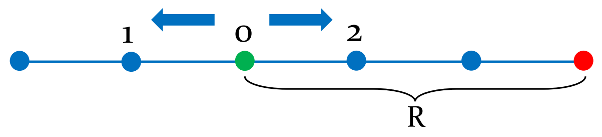


FIG. 1. A walker moving randomly in one dimension with equal probability to move to the left or right. From Position O (starting point marked in green), the walker can move to either position 1 or position 2 with an equal probability of 0.5. The mean end-to-end distance, R (distance between the starting point and the ending point (red) is also marked.

moves to the right with probability p and to the left with

[*] kasturi_ug24@ashoka.edu.in
[†] sanjana.gupta˙ug24@ashoka.edu.in

probability $q = 1 - p$. If there is no bias in the system, then $p = q = \frac{1}{2}$. The position R_N after N steps may be written

$$R = \sum_{i=1}^{N} \sigma_i$$

where $\sigma_i = +1$ if the walker moves to the right at step i, and $\sigma_i = -1$ if the walker moves to the left at step i. At each step, the probability for these two outcomes is given by

$$P_\sigma = p\delta_{\sigma,+1} + q\delta_{\sigma,-1} = \begin{cases} p & \text{if } \sigma = +1 \\ q & \text{if } \sigma = -1 \end{cases}$$

This is a normalized discrete probability distribution. The multivariate distribution for all the steps is then

$$P(\sigma_1, \ldots, \sigma_N) = \prod_{i=1}^{N} P(\sigma_i)$$

After N steps, the location of the walker is R. The average of R can be computed as:

$$\langle R \rangle = \left\langle \sum_{i=1}^{N} \sigma_i \right\rangle = N\langle\sigma\rangle = N \sum_{\sigma=\pm 1} \sigma P(\sigma) = N(p - q)$$
$$\langle R \rangle = 0 \text{ (as } p = q) \qquad (1)$$

Next, we compute the average of R^2 to understand what effective area is covered by the walk.

$$\langle \sigma_i \sigma_{i'} \rangle = \delta_{ii'} + (1 - \delta_{ii'})(p - q)^2 = \begin{cases} 1 & \text{if } i = i' \\ (p - q)^2 & \text{if } i \neq i' \end{cases}$$

$$\begin{aligned} \langle R^2 \rangle = \sum_{i=1}^{N} \sum_{i'=1}^{N} \langle \sigma_i \sigma_{i'} \rangle &= N^2(p - q)^2 + 4Npq \\ &= 0 + 4N\frac{1}{2}\frac{1}{2} \\ &= N \end{aligned}$$

$$\boxed{\langle R^2 \rangle \propto N} \qquad (2)$$

The complete probability distribution for the walk can be given as:

$$P_{N,n} = \binom{N}{N_\mathrm{r}} p^{N_\mathrm{r}} q^{N_\mathrm{l}}$$

where $N_{\mathrm{r/l}}$ are the numbers of steps taken to the right/left, with $N = N_\mathrm{r} + N_\mathrm{l}$, and $n = N_\mathrm{r} - N_\mathrm{l}$. There are many independent ways to take N_r steps to the right. For example, our first N_r steps could all be to the right, and the remaining $N_\mathrm{l} = N - N_\mathrm{r}$ steps would then all be to the left. Or our final N_r steps could all be to the right. For each of these independent possibilities, the probability is $p^{N_\mathrm{r}} q^{N_\mathrm{l}}$. From elementary combinatorics, we get

$$\binom{N}{N_\mathrm{r}} = \frac{N!}{N_\mathrm{r}! N_\mathrm{l}!}$$

Note that $N \pm n = 2N_{\mathrm{r/l}}$, so we can replace $N_{\mathrm{r/l}} = \frac{1}{2}(N \pm n)$. Thus,

$$P_{N,n} = \frac{N!}{\left(\frac{N+n}{2}\right)! \left(\frac{N-n}{2}\right)!} p^{(N+n)/2} q^{(N-n)/2}.$$

B. Two-Dimensional Random Walk

For the 2D random walk, we can follow an approach similar to the one we used for the 1D random walk. However, we would like to approach this a little differently, using a more general method using thermodynamic variables but obtaining the same result. We begin with the entropy $S(\mathbf{r})$, which is linked to the count of various walks ($\mathfrak{N}_N(\mathbf{r})$) originating from point (0) and reaching a lattice point $\mathbf{r}$. This relationship is straightforwardly expressed as the natural logarithm of the number of walks:

$$S(\mathbf{r}) = \ln[\mathfrak{N}_N(\mathbf{r})]$$

Now, considering each step having z possible directions; if each lattice site has z neighbors, the overall number of possibilities after N steps is given by:

$$\sum_{(\mathbf{r})} \Re_N(\mathbf{r}) = z^N \qquad (3)$$

z is also called the coordination number. For 2D square lattice, $z = 4$. Moving on to the end-to-end vector $\mathbf{r}$, it's essentially the accumulation of N "jump vectors," each represented by $\mathbf{a}_n$. Here, each $\mathbf{a}$ is a vector of length a

with z potential orientations, and these vectors have entirely independent orientations and as a result there is no correlation between the directions of successive steps. Consequently, the squared distance ($\mathbf{r}^2$) accumulates linearly as the number of steps (N) increases, resulting in a proportional relationship.

$$\langle \mathbf{r}^2 \rangle = Na^2$$

Additionally, the distribution function for $\mathbf{r}$, denoted as $p(\mathbf{r})$, takes on a Gaussian shape when there are numerous independent jump vectors ($N \gg 1$). The emergence of a Gaussian distribution in the probability function $p(\mathbf{r})$ stems from the central limit theorem which states that the sum of a large number of independent, identically distributed random variables tends to have a Gaussian distribution. In two dimensions, it assumes a form similar to:

$$\begin{aligned} p(x,y) = \text{ const. } N^{-1/2} \exp\left(\frac{-x^2}{2\langle x^2 \rangle}\right) N^{-1/2} \exp\left(\frac{-y^2}{2\langle y^2 \rangle}\right) \\ \cong N^{-1/2} \exp\left(\frac{-r^2}{Na^2}\right) \end{aligned}$$

This distribution is characterized by decreasing entropy as elongation increases. The formula for entropy at fixed elongation ($S(\mathbf{r})$) can be expressed as:

$$S(\mathbf{r}) = S(0) - \frac{r^2}{R_0^2} \text{ (in two dimensions)}$$

The entropy decreasing with elongation is often rewritten in terms of free energy , where the energy (E) is a constant, leading to the fundamental formula:

$$F(\mathbf{r}) = F(0) + \frac{Tr^2}{R_0^2} \tag{4}$$

This formula provides insights into the "spring constant" of an ideal chain (random-walk).

C. One-Dimensional Ballistic Walk

Returning to the 1D case, let's consider a scenario where the walker can only move in one direction, akin to a constant velocity problem. In the 1D random walk, the walker had permission to revisit a previously occupied position. However, in this case, the walker is constrained from doing so; it avoids retracing its steps and must continue moving in a single direction. The end-to-end distance is simply equal to the length of the walk multiplied by the step length. Consequently, the mean square end-to-end distance becomes proportional to N^2. Notably, there is a discernible difference in the exponent of N for a 1D ballistic system or self-avoiding walk compared to a random walk. It becomes intriguing to explore how the exponent varies with N for higher dimensions and how it compares with random walks.

$$\langle R^2 \rangle \propto N^2$$

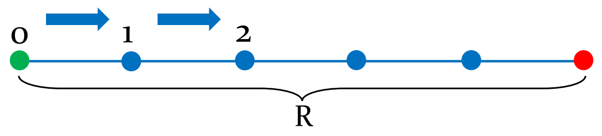


FIG. 2. A walker moving in one direction. From Position O (starting point marked in green), the walker can move to either position 1 and then to position 2 but it cannot go back to position 0. The mean end-to-end distance, R is also marked. Usually the mean end-to-end distance for a self-avoiding walk is larger than that for a random walk as seen in the diagram as well.

D. Self-Avoiding Walks (SAWs) in d dimensions

The total number of SAWs of N steps has the asymptotic form (at large N)

$$\mathfrak{N}_N(\text{ tot }) = \text{ constant } \tilde{z}^N N^{\gamma-1}$$

The first factor $\tilde{z}^N$ is reminiscent of the z^N which we had for random walks (equation 3), but $\tilde{z}$ is somewhat smaller than coordination number z. For the 3D simple cubic lattice, $z=6$ and $\tilde{z} \approx 4.68$. For a 2D square lattice, $z=4$ and $\tilde{z} \approx 2.64$. The effective coordination number $\tilde{z}$ depends on the previous steps of the walk, and it is always less than or equal to the coordination number z. The effective coordination number $\tilde{z}$ is the average number of possible directions for a walk to continue from a point, considering all possible walks and all possible points. The coordination number z is the maximum number of possible directions for a walk to continue from a point, regardless of the previous steps and the self-avoidance constraint. The second factor, $N^{\gamma-1}$, is called the enhancement factor. The exponent γ depends only on the dimensionality, d :

For all three-dimensional lattices $\gamma = \gamma_3 \cong 7/6$

For all two-dimensional lattices $\gamma = \gamma_2 \cong 4/3$

γ is a universal exponent; this is in contrast to $\tilde{z}$, which depends not only on d but also on the particular lattice chosen Note that for $d = 1, \mathfrak{N}_N(tot) = 2$, independently of N. Thus $\tilde{z}_1 = 1$ and $\gamma_1 = 1$.

The end-to-end distance r has a mean square average which we shall call R^2, and which scales as

$$R \cong \mathbf{a}N^{\nu}$$

ν is another universal exponent which depends on the dimension ($\nu_3 \cong 3/5, \nu_2 \cong 3/4, \nu_1 = 1$) and is called the Flory Parameter. The exponent γ is related to chain entropy, and the exponent ν is related to chain size. In our project, we will be calculating the Flory Parameter for 2D SAWs on a square lattice.

The distribution law for $\mathbf{r}$ depends on $\mathbf{r}$ only through the ratio $r/\boldsymbol{R}$

$$p_N(r) = \frac{1}{\boldsymbol{R}^d} f_p\left(\frac{r}{\boldsymbol{R}}\right) (\mathbf{a} \ll r \ll N\mathbf{a})$$

The prefactor $\frac{1}{R^d}$ is required to ensure the normalization

$$\int p_N(\mathbf{r}) d\mathbf{r} = 1$$

The Flory Calculation of the Exponent ν

The pioneering theoretical groundwork on polymers owes much of its depth to Paul Flory. His approach, a blend of dimensional reasoning, keen physical insight, and field theory computations, unveiled the relationship between the mean squared end-to-end distance of a polymer (or SAW) and the length of its walk (N). Flory's profound contributions earned him the Nobel Prize in 1974, with his Nobel lecture focusing on the "Spatial Configuration of Macromolecular Chains."

We briefly describe his method and the approximations involved. The starting point is a chain, with a certain unknown radius R and an internal monomer concentration

$$c_{\text{int}} \approx \frac{N}{R_0^d} \tag{5}$$

There is a certain repulsive energy in the chain due to monomer-monomer interactions. If c is the local concentration of monomers, the repulsive energy per cm^3 is proportional to the number of pairs present, i.e., to c^2. We write it (per unit volume) as:

$$F_{\text{rep}} = \frac{1}{2} T \nu(T) c^2 \tag{6}$$

where ν has the dimension of $\mathbf{a}$ (d-dimensional) volume and is positive. We call ν the excluded volume parameter or the Flory parameter. (In the Flory notation $\nu = (1-2\chi)\mathbf{a}^d$ where $\mathbf{a}^d$ is the monomer volume and χ is an interaction parameter. For good solvents $\chi < \frac{1}{2}$ and $\nu > 0$.)

One essential approximation is to replace the average of c^2 by the square of the average

$$\langle c^2 \rangle \rightarrow \langle c \rangle^2 \sim c_{\text{int}}^2 \tag{7}$$

This approximation is typical of a mean-field approach where all correlations between monomers are ignored. The overall repulsive energy after integration over a volume $\boldsymbol{R}_0^d$ scales as:

$$F_{\text{rep—total}} \approx T\nu(T) c_{\text{int}}^2 R_0^d = T\nu \frac{N^2}{R_0^d} \tag{8}$$

This tends to favor large values of R (i.e., to swell the chain). However, if the distortion is too large, the chain entropy becomes too small, and this is unfavorable. Flory includes this through an elastic energy term derived from a similar calculation for random walks as seen in equation 4 :

$$F_{\text{el}} \approx T \frac{R_0^2}{N\mathbf{a}^2} \tag{9}$$

Equation 9 is also a very strong approximation. Although the spring constant of a real chain (self-avoiding walk) is much smaller than that suggested by equation 9, we accept equations 8 and 9 and add them to get:

$$\frac{F}{T} \approx \nu \frac{N^2}{R_0^d} + \frac{R_0^2}{N\mathbf{a}^2} \tag{10}$$

Equation 10 has a minimum for a well-defined radius $R_0 = R$. Omitting all numerical coefficients, we find

$$R^{d+2} \approx \nu \mathbf{a}^2 N^3 \tag{11}$$

or $R \sim N^\nu$ with $\nu = \frac{3}{d+2}$.

$$\boxed{\nu = \frac{3}{d+2}} \tag{12}$$

Equation 11 is a very good approximation; it gives the correct value for $d = 1$ ($\nu_1 = 1$). The values for $d = 2$ and $d = 3$ are within a percent of the most accurate numerical results. For most practical applications, the Flory formula can be considered exact.

In our specific context, considering a 2-dimensional lattice, $\nu = \frac{3}{2+2} = \frac{3}{4}$, from which we derive the relationship $\langle R^2 \rangle = N^{2\times\frac{3}{4}} = N^{\frac{3}{2}}$. This formula highlights a fundamental property wherein self-avoiding walks exhibit greater extension compared to simple random walks on average. This distinction, termed the excluded volume effect, underscores a critical aspect: while simple random walks have the potential to fold upon and intertwine themselves, leading to tighter configurations, self-avoiding walks strictly prevent self-intersections. Moreover, in certain polymer systems, a repulsive force operates between non-adjacent monomer units, further contributing to the phenomenon of extension. Particularly in a favorable solvent environment, monomer units display a preference for associating with solvent molecules over other monomer units, consequently fostering increased extension in the polymer chain.

II. METHODS

1. *Monte Carlo*

Monte Carlo methods are useful for obtaining statistical estimates of the values of the connective constant, critical exponents, and other quantities related to self-avoiding walks. Essentially, a Monte Carlo simulation is a computer experiment that observes random versions of a particular system. After collecting sufficient data, statistical techniques can be employed to obtain estimates and confidence intervals for the desired quantities.

To estimate the exponent ν through a Monte Carlo experiment, the following steps are undertaken:

(1) **Select Values of N:** Choose several values for N, denoted as $N_1, \ldots, N_m$. The selection of these values is crucial and will impact the accuracy and reliability of the estimation.

(2) **Generate Self-Avoiding Walks:** For each chosen N_i, generate a substantial number of N_i-step self-avoiding walks at random. This process is integral to obtaining a reliable estimate, denoted as R_i, for the quantity of interest associated with N_i.

(3) **Log-Log Plot:** Create a log-log plot of the end-to-end distance $\langle R \rangle$ and N to explore the relationship between these variables.

(4) **Curve Fitting:** Employ a curve fitting technique to fit a curve of the form $\log\langle R \rangle = 2\nu \log N$ through the data points $(\log N, \log\langle R \rangle)$. The determination of the "best" value for ν is then derived from this curve fit.

While these steps provide a structured approach, considerations arise at each stage. In determining the number and specific values of N is critical. This decision influences the sensitivity and accuracy of the final estimation. Defining "many" regarding the number of generated walks is a nuanced decision. The adequacy of the estimate depends on the quantity and quality of the generated data. The curve fitting process relies on the assumption that the chosen curve form is asymptotically correct. Understanding the implications of this assumption is vital for interpreting the estimated values. This process is employed in different ways in different algorithms and the considerations are slightly different. We use different algorithms to calculate the Flory Parameter and study the dynamics of 2D SAWs.

A. Myopic Algorithm

The Myopic Self-Avoiding Walk (MSAW) is the most obvious algorithm which executes a random walk by choosing from directions that have already not been visited.

1. Let $w(0)$ be the origin, and set $i = 0$.
2. Increase i by one. Of the neighbours of $w(i-1)$ that are not in the set $\{w(0), \ldots, w(i-2)\}$, choose one at random, and let $w(i)$ be that point. (If all of the neighbours of $w(i-1)$ are in this set, then the walk is trapped, so return to Step 1.)
3. Repeat Step 2 if $i < N$, and stop if $i = N$.

This algorithm produces a walk in $\mathcal{S}_N$ ($\mathcal{S}_N$ is the set of all N-step self-avoiding walks), but with the wrong distribution. To see where the problem is, consider four-step walks on $\mathbf{Z}^2$: the probability of obtaining the walk (North-East-East-East) on a given attempt is $\frac{1}{4} \times \frac{1}{3} \times \frac{1}{3} \times \frac{1}{3}$, but the probability of obtaining the walk (North-East-South-East) is $\frac{1}{4} \times \frac{1}{3} \times \frac{1}{3} \times \frac{1}{2}$. Hence, the probabilities are not uniform on $\mathcal{S}_N$. Specifically, as N becomes large, these probabilities deviate significantly from uniformity. The MSAW algorithm introduces a distinct model, closely resembling the "true self-avoiding walk." However, to accurately estimate ν, it is imperative to explore alternative methods for generating walks that adhere to the correct probability distribution. Failing to do so could compromise the proper estimation of ν.

Listing 1. Pseudocode for Myopic algorithm

```
FUNCTION Myopic_SAW(N):
    Initialize x, y as arrays containing 0
    Set to store visited positions
    Initialize stop flag, step counter to 0

    FOR loop in range N:
    Generate possible directions
    Create empty array for feasible directions
        FOR loop in directions:
        Calculate and update new position based
            on
        if its not been visited

        IF available directions is not empty:
            choose a random direction from
                available directions
            Update positon, mark new position as
                 visited, add x and y values
        ELSE:
            Set stop flag to 1, store step count
            BREAK the loop

    RETURN x, y, stop flag,  stepscount
```

Results

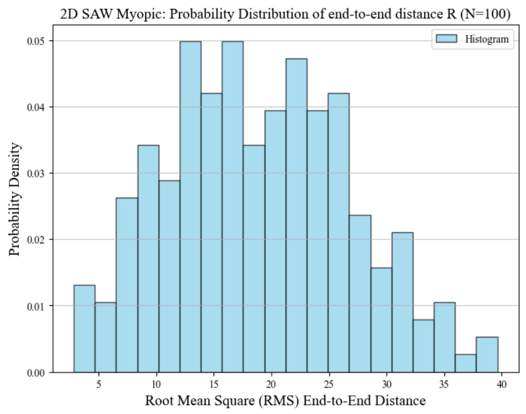


FIG. 3. Probability Distribution of R for a Walk of length N =100 generated by MSAW

Upon termination, the resulting walk $W \equiv (w(0), \ldots, w(N))$ is self-avoiding. Furthermore, we assert that for any $\omega \in \mathcal{S}_N$, the probability $\Pr\{W = \omega\} = 1/c_N$ where c_N is the number of N-step self-avoiding walks beginning at the origin. To comprehend this, let $\mathcal{S}_N^o$ represent the set of all N-step random walks. If we uniformly and randomly select members from $\mathcal{S}_N^o$ until finding one in $\mathcal{S}_N$, the final outcome is uniformly distributed across $\mathcal{S}_N$. Interestingly, this aligns with the ESS algorithm's behavior; Step 3 acts as a shortcut, avoiding the generation of the last $N - i$ steps of a walk already known to intersect itself by the i-th step. Therefore, ESS genuinely generates a self-avoiding walk randomly. However, its efficiency diminishes significantly as N grows moderately large. The probability that an N-step simple random walk is self-avoiding is $c_N/(2d)^N$, implying the anticipated number of attempts (returns to Step 1) is $(2d)^N/c_N$.

B. Elementary Simple Sampling / Basic SAW

Elementary Simple Sampling (ESS) is an algorithm designed to generate ordinary simple random walks until it successfully produces a self-avoiding walk of length N. The process unfolds as follows:
1. Begin at the origin, denoted as $w(0)$, and set i to 0.
2. Increment i by one. Randomly select one of the $2d$ neighbors of $w(i-1)$, assigning it as $w(i)$.
3. If $w(i)$ coincides with any of the previous points $w(j)$ for $j = 0, 1, \ldots, i-1$, return to Step 1. Otherwise, proceed to Step 2 if $i < N$; stop if $i = N$.

Listing 2. Pseudocode for ESS algorithm

```
FUNCTION self_avoiding_walk(cols, rows,
    desired_step_length):
    Initialize a grid of size (cols x rows) to
        track visited cells
    Mark the starting cell as visited
    Initialize path with the starting position
    Set current_step to 0

    WHILE current_step < desired_step_length:
        Generate available move options
        IF options list is not empty:
            Choose a random direction from
                options
            Update position and mark the new
                cell as visited
            Add the new position to the path
```

```
                Increment step count
            ELSE:
                Return the path and False if walker
                    is stuck

        Return the path and True if desired
            steps are completed
```

Results

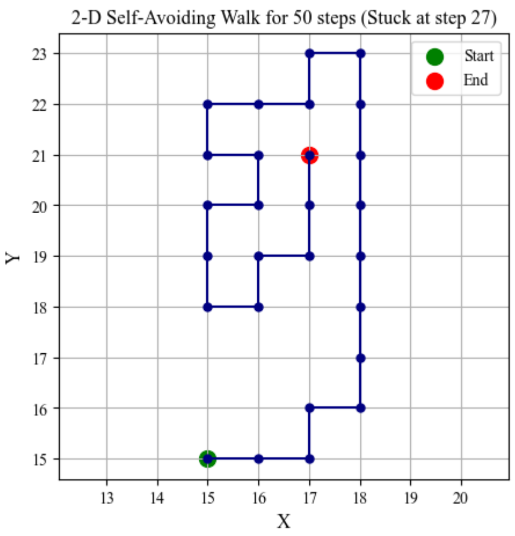


FIG. 4. Basic Self Avoiding Walk of 27 steps generated using Elementary Simple Sampling.

As we can see in Fig. 4, the algorithm is not efficient in generating long walks so for all practical purposes, we need to use other algorithms.

It is clearly seen in Fig. 5 that the SAW tends to spread out more than a RW of the same length and effectively covers a greater area. This difference in also seen in the values of the Flory Parameter for both.

Fig. 6 shows that the fraction of successful walks decreases exponentially with N as expected. Longer walks become much harder to generate when sampled from a random walk distribution.

It is observed in Fig. 7 that it follows a power law relationship. To understand the dependence better, we plot a log-log graph.

In Fig. 8, the slope is equal to 2ν. From this we estimate the Flory Parameter as 0.63 which is slightly off from the expected value of 0.75.

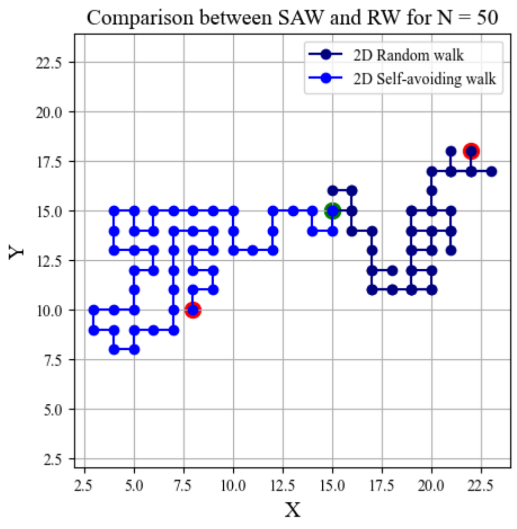


FIG. 5. Comparison between Random Walk and Self-Avoiding Walks.

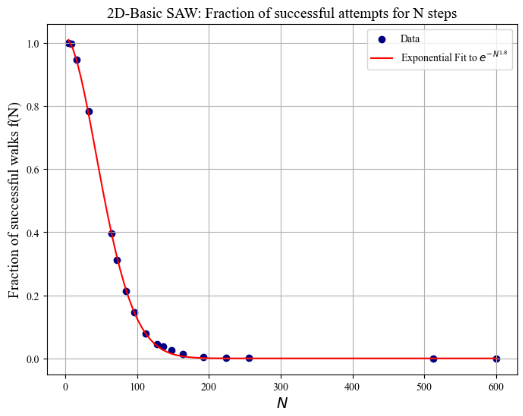


FIG. 6. Fraction of Successful walks vs. N

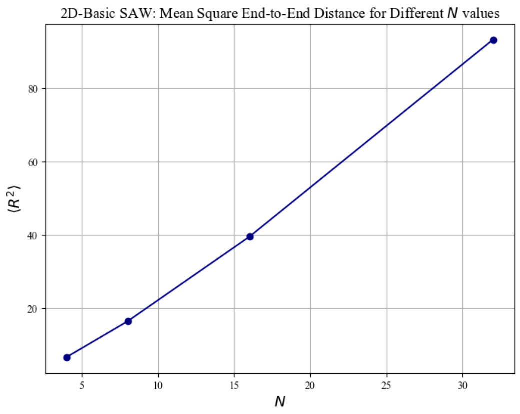


FIG. 7. $\langle R^2 \rangle$ vs. N graph for walk generated by ESS Algorithm.

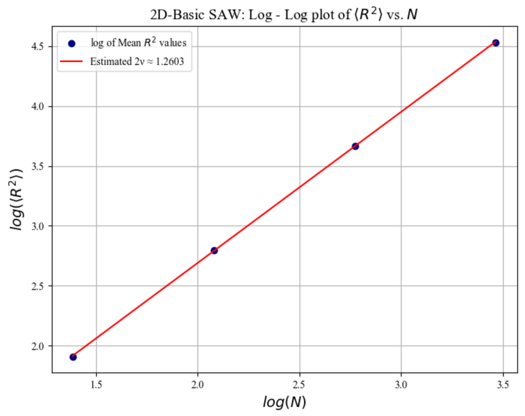


FIG. 8. $\log(\langle R^2\rangle)$ vs. $\log(N)$.

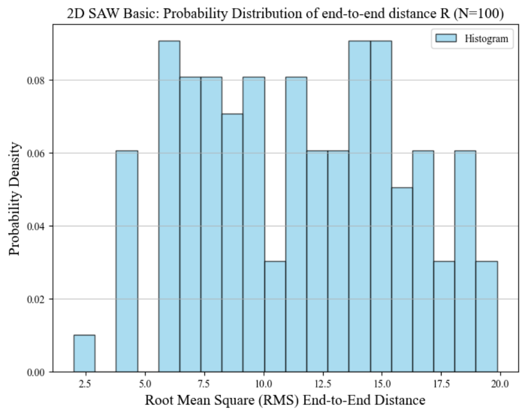


FIG. 9. Probability Distribution of R for N = 100 generated by ESS

C. Rosenbluth and Rosenbluth Enrichment Method

As the Elementary Simple Sampling Method becomes very inefficient for long walks (see Fig. 6), some enrichment methods are developed. The Rosenbluth and Rosenbluth algorithm is a method designed for the systematic generation of self-avoiding walks of length N. The uniqueness of this algorithm lies in its ability to assign weights to each walk, ensuring that all permissible configurations for a given length N are equally considered.

Here's a step-by-step breakdown of the algorithm:

Walk Generation and Weighting

1. Initialization: Begin with a walk of length $N = 1$ starting at the origin, denoted as $w(0)$.
2. Weighting Function: Associate a weighting function $w(N)$ with each walk of length N.
- For the first step to the north, $w(1) = 1$, as it is always possible.
- For subsequent steps ($N > 1$), weights are determined based on certain conditions.

3. Weight Determination:
- Case 1: If all three possible steps violate the self-intersection constraint, terminate the walk with $w(N) = 0$ and initiate a new walk from the origin.
- Case 2: If all three steps are possible, set $w(N) = w(N-1)$.
- Case 3: If only m steps ($1 \le m < 3$) are possible, set $w(N) = \frac{m}{3} w(N-1)$. Randomly choose one of the m possible steps.

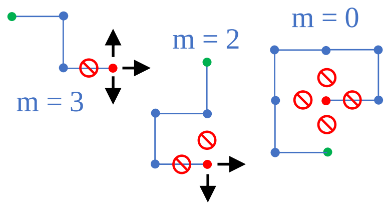


FIG. 10. Possible Moves

Unbiased Value Calculation

To obtain an unbiased estimate of the mean square end-to-end distance $\langle R^2\rangle$, a weighted average is computed over multiple trials (i):

$$\langle R^2\rangle = \frac{\sum_i w_i(N) R_i^2}{\sum_i w_i(N)}$$

where R_i^2 is the value of R^2 obtained in the ith trial, and $w_i(N)$ is the weight found for that specific t rial. Simply, the weight associated with the last step of a particular walk is multiplied with the mean square end-to-end distance of the walk and the average of this quantity is taken for multiple walks of the same length to obtain $\langle R^2\rangle$. The probability of moving in a particular direction is the weight associated with the step divided by the total number of feasible directions and each direction is given equal probability.

Listing 3. Pseudocode for Rosenbluth algorithm

```
FUNCTION rosenbluth_sampling(max_steps,
    num_trials):
    Initialize total weighted R-squared,
    total weights and successful walks as 0

    FOR loop in range(num_trials):
        Initialize lattice grid and start
            position
        Initialize weights and R-squared value

        FOR each step IN range(1, max_steps):
            Get valid moves
            Update weights based on available
                moves
            Case 1: no available moves, weight =
                 0
            Case 2: available moves < 3, weight
                = (available moves/3)*weight of
                the last walk
            Case 3: All 3 moves are available,
                weight from previous walk
                carries on

            IF no valid moves available:
                BREAK out of the loop

            Choose a random move and update
                position
            Update lattice and R-squared

        Update total weighted R-squared and
            total weights
        IF walk reaches max steps:
            Increment successful_walks

    IF total_weights > 0:
        fraction of successul walks = successful
             walks / number of trials
        RETURN total weighted R-squared /
            total_weights, fraction of
            successful walks
    ELSE:
          RETURN0       , 0
```

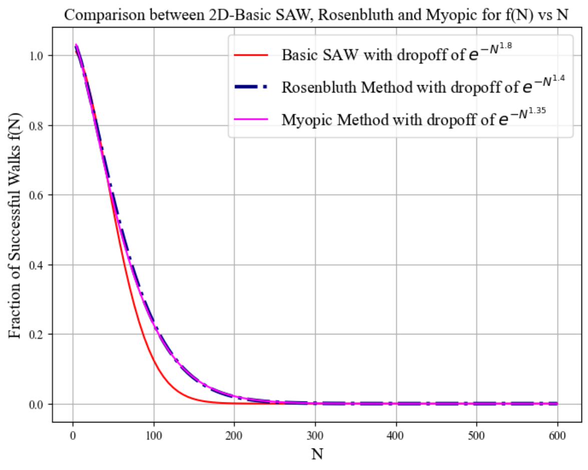


FIG. 11. Fraction of Successful Walks for Rosenbluth Algorithm

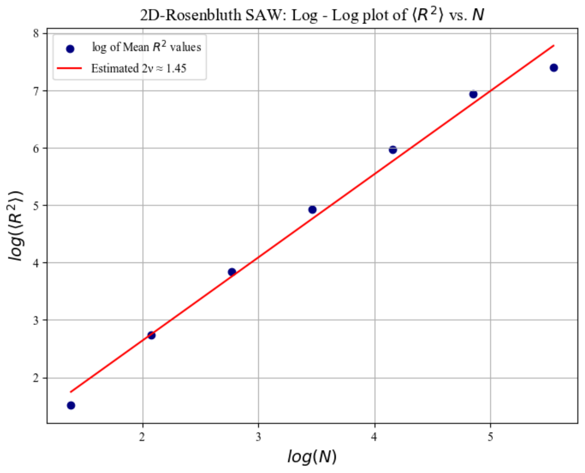


FIG. 12. $\log(\langle R^2 \rangle)$ vs. $\log(N)$

Results

For Rosenbluth Algorithm as well, the fraction of Successful walks decreases exponentially with N as seen in Fig. 11 but it falls of slowly compared to the ESS Algorithm as it can generate longer walks. The Myopic Algorithm and the Rosenbluth algorithm have a similar fall because both the algorithms consider the available directions and assign weights to them unlike the ESS method where the walks are sampled from random walks.

The Flory Parameter is estimated from the slope of Fig. 12. As we can generate longer walks with the Rosenbluth Algorithm, we get a better estimate of the Flory Parameter at 0.725.

The Rosenbluth method is an improvement over the basic self-avoiding walk due to its incorporation of a weighting scheme. Unlike the basic method that uniformly samples all possible walks, the Rosenbluth algorithm assigns weights to different walks, favoring those less likely to intersect themselves. By dynamically adjusting the weights based on self-avoidance criteria, the Rosenbluth method significantly enhances the efficiency of generating valid self-avoiding walks, especially in high-dimensional spaces, leading to a more targeted exploration of configuration space and improved sampling.

However, its efficiency also diminishes for larger values of N due to the increasing probability of self-intersection. Also, there are large fluctutations and errorbars for large N as seen in Fig. 14. To generate walks of a greater length that mimic real polymers, an improved version of the Rosenbluth algorithm is used called the Pruned-Enriched Rosenbluth Method (PERM) where if the weight at a particular step is much larger than the av-

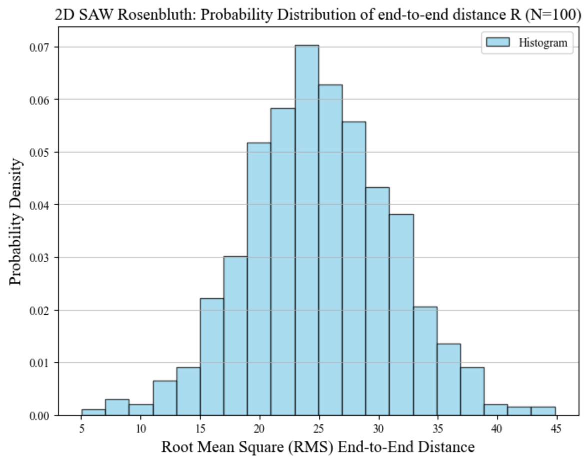


FIG. 13. Probability Distribution of R for a Walk of N = 100 generated by the Rosenbluth Algorithm

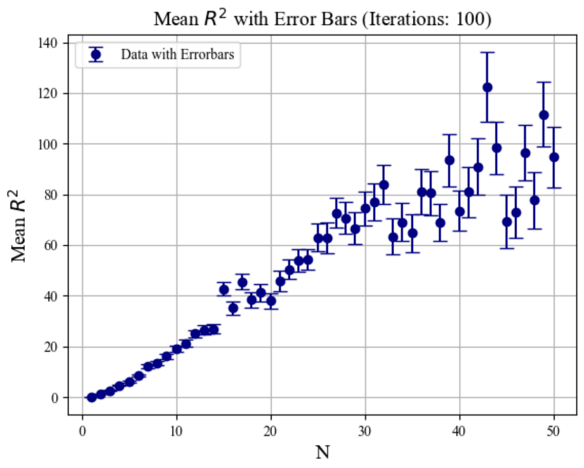


FIG. 14. $\langle R^2 \rangle$ vs. N for Rosenbluth with standard deviation calculated over 100 simulations using the formula $s\left(\langle R^2(N)\rangle\right) = \sqrt{\frac{L}{(L-1)}\frac{\sum_{i=1}^{L}\left(w_i^{(N)}\right)^2\left(R_i^2(N)-\langle R^2(N)\rangle\right)^2}{\left(\sum_{i=1}^{L} w_i^{(N)}\right)^2}}$ where L is the number of trials for a given walk length of N.

erage the polymer is doubled (copied), and if the weight is smaller than the average, it has a 50 % chance of being deleted and 50 % chance of its weight being doubled. We haven't simulated SAWs using PERM in this project but we have explored other algorithms to generate longer self avoiding walks.

D. Dimerization Method

Dimerization stands out as an exceptionally efficient algorithm for precisely and uniformly generating single walks in any dimension, exhibiting an expected running time of approximately $N^{O(\log N)}$. While its computational demands are substantial, it excels in generating lengthy walks. Operating as a recursive procedure, it uniformly generates self-avoiding walks on $\mathcal{S}_N$. The core concept involves generating two independent $(N/2)$-step self-avoiding walks ("dimers") and attempting to concatenate them. If the result remains self-avoiding, the task is complete; otherwise, both dimers are discarded, and the process restarts. This recursive approach extends to $(N/4)$-step walks and so on. The recursion halts at the k-th level if there's a swift method for generating self-avoiding walks of length $N/2^k$. This recursive procedure, denoted as $DIM(N)$, uniformly generates an N-step self-avoiding walk ω from $\mathcal{S}_N$, with a fixed small integer N_0 (e.g., $N_0 = 10$).

1. If $N \leq N_0$, generate an N-step walk ω using ESS Algorithm or some other algorithm that generates SAWs uniformly and stop.
2. For $N > N_0$, set $N_1 = \lfloor N/2 \rfloor$ and $N_2 = N - N_1$.
3. Recursively execute $DIM(N_1)$ and $DIM(N_2)$, obtaining self-avoiding walks ω^1 and ω^2.
4. Concatenate ω^2 to ω^1 to form ω. If ω is self-avoiding, stop; otherwise, return to Step 2 and restart.

Listing 4. Pseudocode for Dimerization algorithm

```
FUNCTION Check_SAW(x, y, N):
    RETURN N + 1 as length of unique coordinates
         (zip(x, y))

FUNCTION unit_SAW(N):
    SET directions
    WHILE True:
        INITIALIZE x, y as empty values
        SET coordinates as a set with (0, 0)
        SET stop as False

        WHILE len(x) < N and not stop:
            SELECT random direction dx, dy from
                direction
            IF (x[-1] + dx, y[-1] + dy) in
                coordinates:
                SET stop as True
            ELSE:
                APPEND x[-1] + dx to x and y[-1]
                     + dy to y
                ADD (x[-1] + dx, y[-1] + dy) to
                    coordinates

        IF not stop:
            RETURN x, y

FUNCTION Dimer_SAW(N):
    IF N <= 3:
        RETURN unit_SAW(N)
    WHILE True:
        GENERATE x_1, y_1 by calling Dimer_SAW
```

```
            with N // 2
        GENERATE x_2, y_2 by calling Dimer_SAW
            with N - N // 2
        SHIFT x_2 and y_2 to connect to the end
            of x_1 and y_1
        COMBINE x_1 and x_2 (excluding the first
            element) to create x_new
        COMBINE y_1 and y_2 (excluding the first
            element) to create y_new

        IF Check_SAW(x_new, y_new, N) is True:
            RETURN x_new, y_new
```

Results

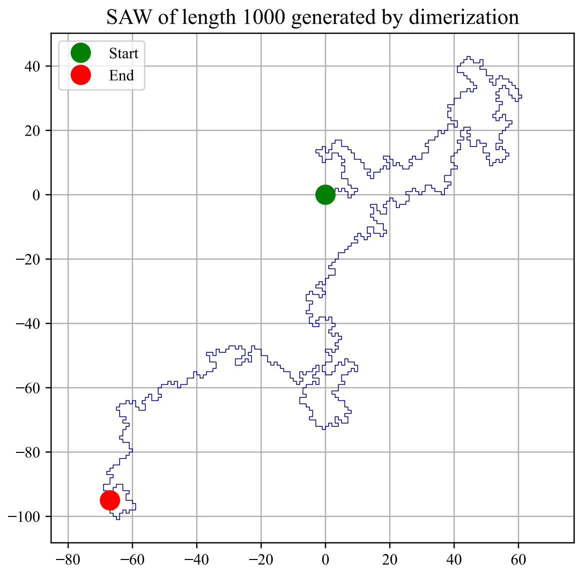


FIG. 15. Self Avoiding Walk of 1000 steps generated using Dimerization.

We could generate upto 1500 steps for dimerization method, beyond that it was taking hours to generate walks. The value of the Flory parameter estimated from the slope of Fig. 16 is ≈ 0.74 which is an improvement from the previous algorithms used.

E. Pivot Algorithm

The algorithms discussed so far are based on local deformations but the Pivot algorithm can make global changes and help in effectively generating long walks much faster than other algorithms. The pivot algorithm takes a different approach, aiming to relocate substantial segments of the walk in each iteration. Although

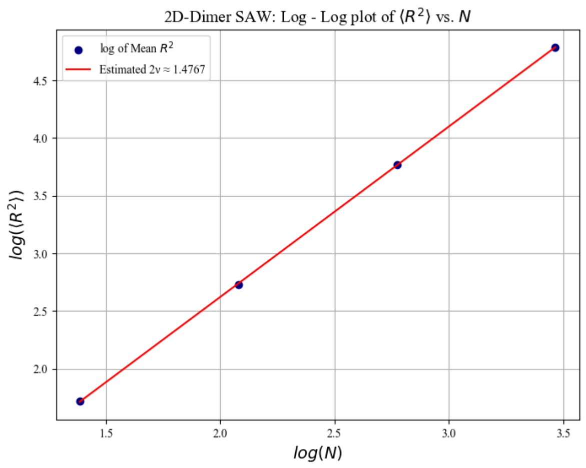


FIG. 16. $\log(\langle R^2\rangle)$ vs. $\log(N)$ graph. The Flory Parameter is estimated from the slope.

these significant moves have a higher likelihood of being rejected compared to local moves, a successful pivot is often rewarded with a substantial alteration in global observables, such as the end-to-end distance.

In the pivot algorithm, a ”pivot site” is randomly selected on the existing walk. The walk is then divided into two segments at this pivot site, and a symmetry operation from $\mathbf{Z}^d$ is randomly applied to one of the segments, utilizing the pivot site as the origin. The resulting configuration is accepted only if it remains self-avoiding. Despite the higher rejection rate for large moves, the pivot algorithm demonstrates remarkable efficiency when investigating global observables. It demands approximately $O(N \log N)$ computer time to generate an observation that can be considered ”effectively independent.” This efficiency is noteworthy, given that merely recording an N-step walk already requires a time complexity of $O(N)$ unlike the previous algorithms where the time complexity was in the order of power of N.
The procedure for a single attempt of the pivot algorithm (where $\omega^{[t]}$ is the current walk) can be given as follows:
(a) Choose the pivot site I and the symmetry G at random. Set $x = \omega^{[t]}(I), j = 1$, and $\tilde{\omega}(I) = \omega^{[t]}(I)$.
(b) Set $\tilde{\omega}(I+j) = G_x\left(\omega^{[t]}(I+j)\right)$ (if $I+j \leq N$) and set $\tilde{\omega}(I-j) = \omega^{[t]}(I-j)$ (if $I-j \geq 0$).
(c) If $I+j \leq N$, then check to see if $\tilde{\omega}(I+j)$ is in the set of sites $\tilde{\omega}[I-j+1, I+j-1]$. If it is, then the current attempt fails, so stop; otherwise, continue.
(d) If $I-j \geq 0$, then check to see if $\tilde{\omega}(I-j)$ is in the set of sites $\tilde{\omega}[I-j+1, I+j]$. If it is, then the current attempt fails, so stop; otherwise, continue.

(e) If $j < \max\{N-I, I\}$, then increase j by one and go to Step (b). Otherwise, the current attempt has succeeded, so set $\omega^{[t+1]} = \tilde{\omega}$ and stop.

Listing 5.Pseudocode for Pivot algorithm

```
CLASS Pivot_2D_SAW:
    FUNCTION _init_(N, l0):
        self.N = N
        self.l0 = l0
        self.initial_state = Create initial
            state
        self.state = Copy initial state
        self.rotate_matrix = Define rotation
            matrices

    FUNCTION pair_distances(chain1, chain2):
        Calculate distances between two chains
        RETURN distances

    FUNCTION pivot_step():
        Choose random pivot point and side
        Get old and temp chains based on side
        Choose a random symmetry operator
        Calculate new chain based on symmetry
            operation
        Calculate pair distances
        IF any distance is less than l0:
            RETURN False
        ELSE:
            Update state based on side
            RETURN True

    FUNCTION pivot_walk(t):
        accepted = 0
        WHILE accepted < t:
            IF pivot_step():
                Increment accepted count
        Update state after accepted steps
    END CLASS
```

Results

We could use the Pivot Algorithm to generate walks upto 5000 steps. Beyond that, it was becoming computationally expensive. It can be clearly seen that the Pivot Algorithm is way more efficient than any of the other algorithms studied here when it comes to generating long walks. The Flory Parameter estimated from the slope of Fig. 18 is 0.7533 which is the closest to the theoretically expected value of 0.75.

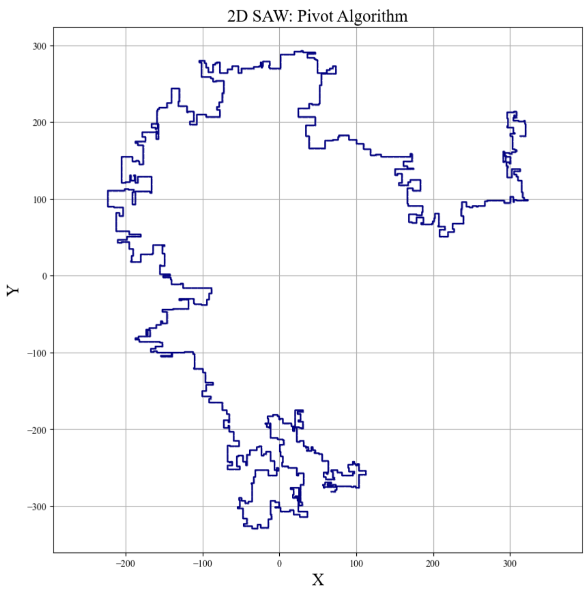


FIG. 17. Self Avoiding Walk of 5000 steps generated using the Pivot Algorithm

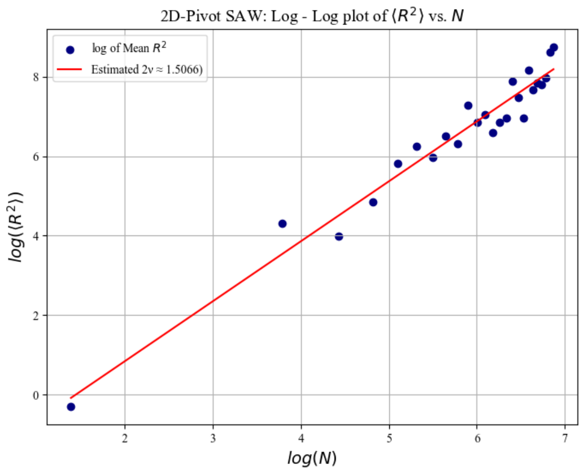


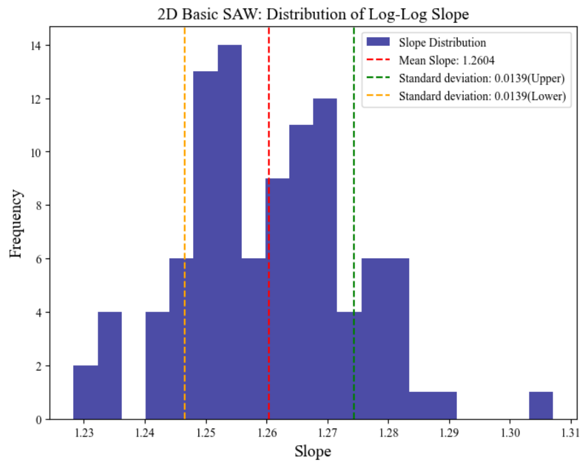
FIG. 18. $\log(\langle R^2 \rangle)$ vs. $\log(N)$ graph. The Flory Parameter is estimated from the slope.

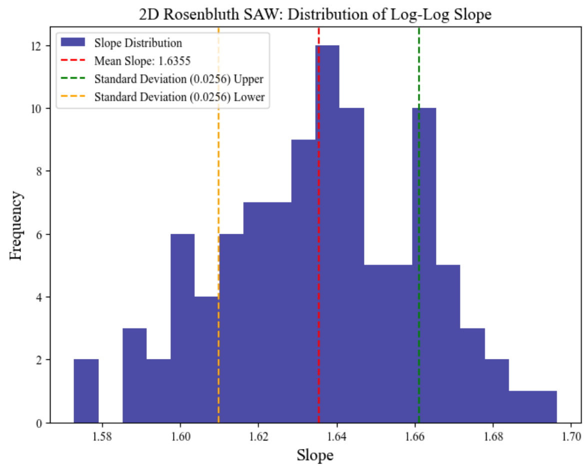
III. DISCUSSION

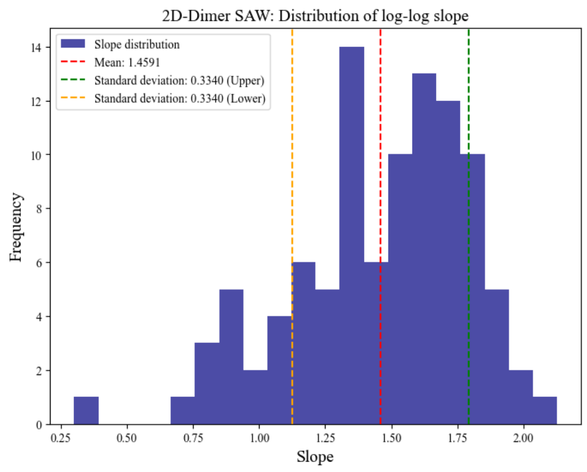
1. *Error in Flory Parameter ν*

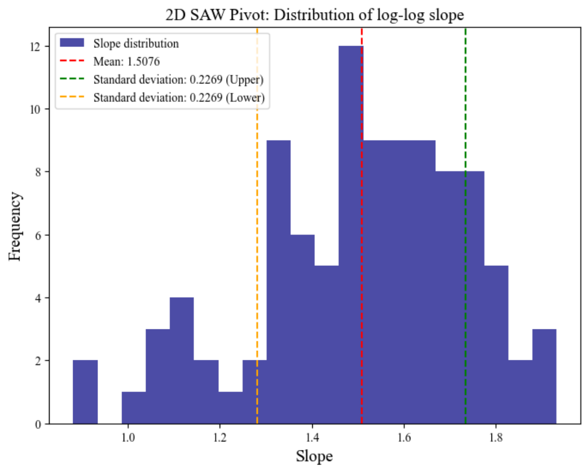
It is observed that the value of the Flory Parameter calculated becomes closer to the theoretical value of 3/4 as we use algorithms that can generate longer walks. This is probably due to 2 reasons. First, more number of data points ensures a better fit and the original derivation by Flory worked under the assumption that N is large, That is why ν calculated from walks with small N have a larger relative percentage error when compared to the theoretical value.

To better understand how the Flory Parameter changes for various algorithms, we run 100 simulations for N values 4,8,16 and 32 and plot the histograms to note the mean and standard deviation.









From the above histograms, we obtain the following table for the Flory Parameters estimated from our numerical simulations and we also calculate the relative percentage error.

TABLE I. Summary of ν value calculation for different algorithms

Algorithm	ν	**Standard Deviation**	**Relative % Error**
ESS/Basic	0.6302	0.0069	15.97
Rosenbluth	0.8178	0.0128	9.03
Dimer	0.7295	0.1670	2.73
Pivot	0.7538	0.1134	0.51

2. *CPU Time Comparison*

TABLE II. CPU Times for different algorithms

N	**CPU time for Algorithms**			
	Basic SAW	***Rosenbluth***	***Dimer***	***Pivot***
4	0.0010	0.0020	0.0000	0.1946
30	0.0009	0.0007	0.0022	0.2828
60	0.0011	0.0000	0.0281	0.3681
100	0.0018	0.0010	0.1910	0.6082
200	0.0063	0.0025	3.7021	1.5641
300	0.0126	0.0021	9.4121	3.0917
400	0.0242	0.0010	48.3271	4.6838
500	0.0203	0.0020	37.5702	7.0778
1000	0.1748	0.0010	410.625	35.7326

Some values have been marked in red because the algorithms were rarely able to generate walks of that length. It is clearly noticed how the CPU times increases rapidly with the number of walks for most algorithms. While noting the CPU times, `numba` wasn't used to speed up the code to avoid biases in the calculation.

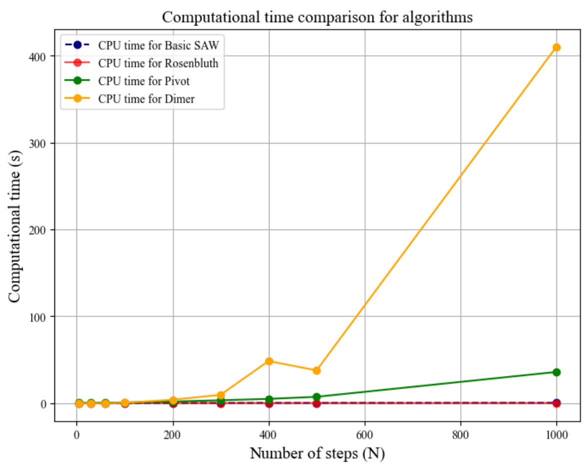


FIG. 19. The graph clearly illustrates how the CPU time increases with N which gives us an idea about the efficiency of the algorithms.

3. *Autocorrelation Function and Autocorrelation Time*

In the context of our study, the strength of correlation between two configurations is characterized by the minimal difference in the position of a single bead. To optimize computational resources, measuring the end-to-end distance and center of mass position after every individual move would be inefficient. Our objective is to evaluate these quantities for configurations that exhibit approximate statistical independence. To achieve this, it is prudent to estimate the mean number of Monte Carlo steps per bead required to obtain statistically independent configurations, a value not known beforehand. This estimation is particularly crucial during preliminary calculations.

The correlation time (τ) serves as a key parameter, representing the duration necessary to achieve statistically independent configurations. This value can be determined by computing the equilibrium-averaged time-autocorrelation function for a fixed-length chain (N) using the following expression:

$$C(t) = \frac{\langle R^2(t'+t)R^2(t')\rangle - \langle R^2\rangle^2}{\langle R^4\rangle - \langle R^2\rangle^2}$$

Here, $C(t)$ is designed such that $C(t=0)=1$, indicating maximum correlation at the initial time, and $C(t)=0$ when configurations are not correlated. As the time interval (t) between configurations lengthens, $C(t)$ tends toward zero ($C(t)\to 0$), signifying that configurations become uncorrelated. The expected decay of $C(t)$ follows an exponential pattern, $C(t)\sim e^{-t/\tau}$, where τ denotes the decay or correlation time. For small values of t, we can do a Taylor expansion of the exponential and get the slope as $-\frac{1}{\tau} \implies \tau = -\frac{1}{slope}$. This provides valuable insights into the correlation structure of configurations, guiding the selection of optimal time intervals for meaningful analyses. We compute the Autocorrelation function for the Dimerization and Pivot Algorithms and estimate the correlation time using the method described. To understand the kind of Correlation, we also look at the Durbin-Watson Statistic.

The autocorrelation time in self-avoiding walks (SAWs) indicates how rapidly configurations become statistically independent in a simulation. A shorter autocorrelation time is favorable, signifying that successive Monte Carlo steps yield nearly independent configurations. This efficiency accelerates convergence to equilibrium. Conversely, a long autocorrelation time implies prolonged memory retention of previous states, hindering statistical property convergence and necessitating more Monte Carlo steps for uncorrelated samples.

4. *The Durbin-Watson test*

The Durbin-Watson test serves as a valuable tool in analyzing the extent of autocorrelation present in the residuals derived from a regression analysis, particularly focusing on first-order autocorrelation. This statistical test is essential in assessing the independence of error terms and ensuring the reliability of regression results.

Assumptions for the Durbin-Watson Test:
1. The errors exhibit a normal distribution with a mean of 0.
2. The errors display stationarity.

Calculation of Test Statistics:
The Durbin-Watson test statistics, denoted as DW, are computed using the formula:

$$DW = \frac{\sum_{t=2}^{T}(e_t - e_{t-1})^2}{\sum_{t=1}^{T} e_t^2}$$

Here, e_t represents the residuals obtained from the Ordinary Least Squares (OLS) method.

Hypotheses for the Durbin-Watson Test:
- Null Hypothesis (H_0): Absence of first-order autocorrelation.
- Alternate Hypothesis (H_1): Presence of first-order correlation.

Interpretation of Durbin-Watson Test Values:
The Durbin-Watson test yields values ranging between 0 and 4, each offering specific interpretations:
- $DW = 2$: No autocorrelation. The range 1.5 to 2.5 is generally considered indicative of no correlation.
- $0 < DW < 2$: Positive autocorrelation. The closer to 0, the stronger the signs of positive autocorrelation.
- $2 < DW < 4$: Negative autocorrelation. The closer to 4, the stronger the signs of negative autocorrelation.

By examining these test results, analysts can make informed decisions about the presence and nature of autocorrelation, ensuring the robustness of regression analyses.

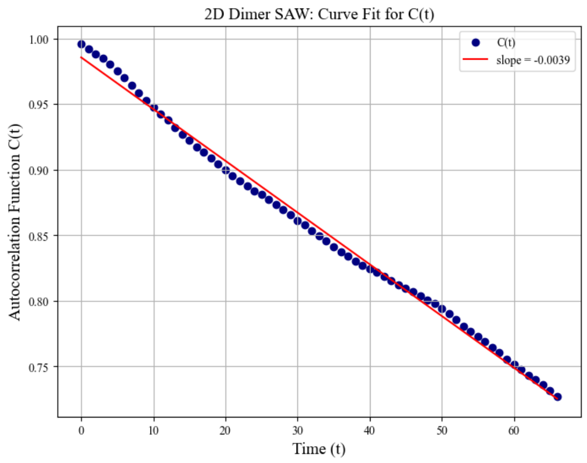


FIG. 20. Autocorrelation Function for the Dimerization Algorithm with Durbin-Watson Statistic ≈ 0.0011 which indicates positive correlation. According to Box-Jenkins, the maximum number of lags is n/4 for a time series n < 240 observations and √n + 45 for > 240 observations.

From Fig. 20, we calculate the correlation time, τ as 109.89 seconds. The dimerization algorithm, employed for the generation of self-avoiding walks, operates in a stepwise fashion by concatenating two independent walks at each iteration. The positive autocorrelation observed in the algorithm's residuals can be attributed to the inherent dependence between the characteristics of consecutive walks, arising from the algorithm's recursive nature. As the algorithm makes local decisions during each step, progressively assembling larger walks through the recursive combination of smaller ones, it introduces patterns and dependencies that endure across subsequent steps. This local decision-making process contributes to the positive autocorrelation observed in the generated walks.

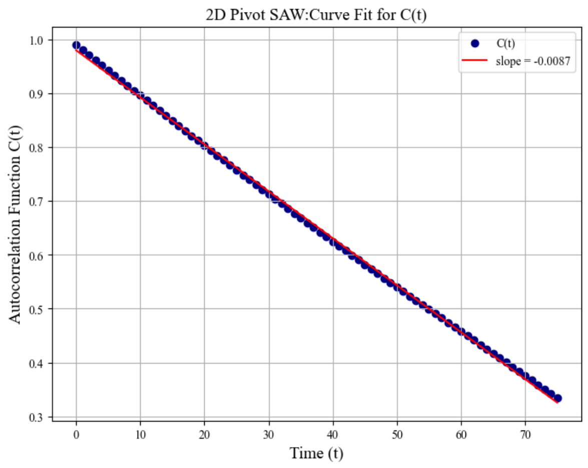


FIG. 21. Autocorrelation Function for the Pivot Algorithm with Durbin-Watson Statistic ≈ 0.0086 which indicates positive correlation

From Fig. 21, we calculate the correlation time, τ as 114.94 seconds. For the Pivot Algorithm used in generating self-avoiding walks, the expected correlation is generally positive. This is due to the nature of self-avoiding walks and the manner in which the Pivot Algorithm creates these walks. The Pivot Algorithm tends to create self-avoiding walks by iteratively rotating sections of the chain without allowing self-intersections. As the walk progresses, the algorithm constructs the walk in such a way that it avoids retracing steps and revisiting previously visited sites or positions, effectively creating a trajectory with a tendency to explore new areas while avoiding overlap with its existing path.

ACKNOWLEDGMENTS

We express our gratitude to Philip Cherian for his invaluable suggestions, encouragement, and general assistance throughout the project. His constructive criticism has greatly improved the quality of this work. We also extend our thanks to Prof. Somendra Mohan Bhattacharjee for recommending essential resources that significantly contributed to our theoretical understanding.

[1] H. Gould, J. Tobochnik, and W. Christian, *An Introduction to Computer Simulation Methods: Applications to Physical Systems* (Addison-Wesley, 2007).

[2] G. Slade, The mathematics of percolation theory, The Mathematical Intelligencer **41**, 1 (2019).

[3] Wikipedia, Self-avoiding walk - Wikipedia (2023).

[4] N. Madras and G. Slade, *The Self-Avoiding Walk*, Probability and Its Applications (Birkhäuser Boston, 1996).

[5] P.-G. de Gennes, *Scaling Concepts in Polymer Physics* (Cornell University Press, 1979).

[6] D. Arovas, Lecture notes on thermodynamics and statistical mechanics (2019), a Work in Progress.

[7] B. Phookun and P. Cherian, Lecture notes on statistical mechanics (Monsoon 2023).

[8] GeeksforGeeks, Autocorrelation - geeksforgeeks (2021).

IV. APPENDICES

Appendix A: Additional Figures

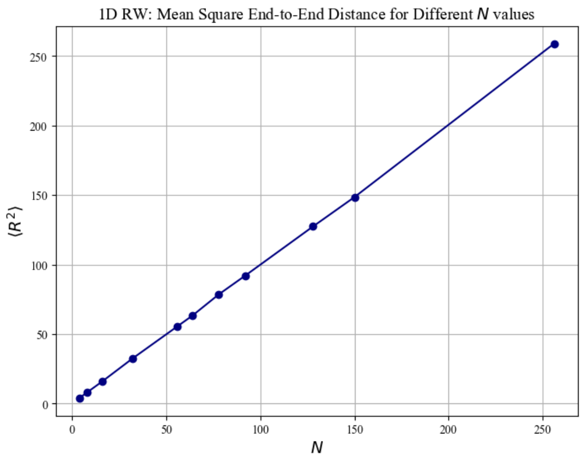


FIG. 22. It is clearly visible that $\langle R^2 \rangle$ varies linearly with N as expected from our theoretical calculations.

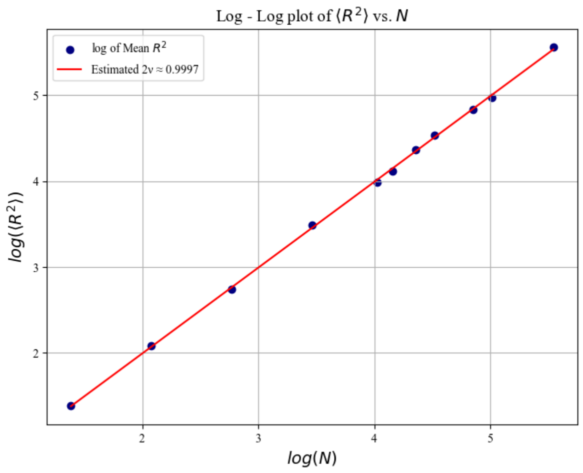


FIG. 23. For 2D Random Walks, we tried to calculated the Flory Parameter from the log-log method followed for the SAW plots and got the ν value as 0.499 which is almost equal to the expected value of 0.5

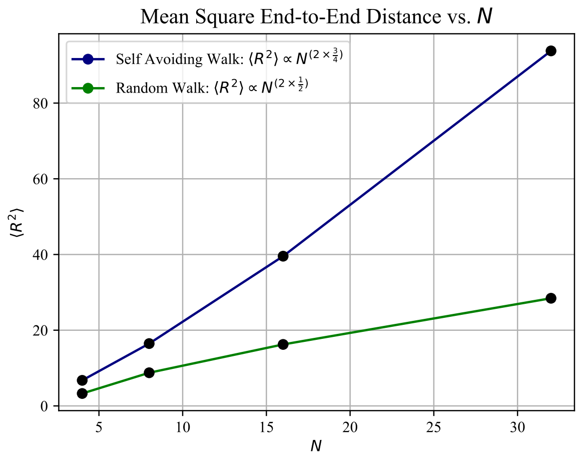


FIG. 24. $\langle R^2 \rangle$ vs. N for RW and SAW

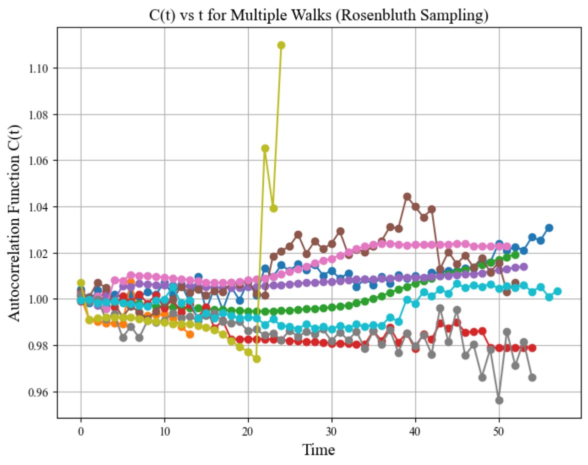


FIG. 25. The Correlation Function calculated for the Rosenbluth algorithm did not show any specific patterns.

Points to Critique

While the coding exercise wasn't inherently challenging, our lack of experience made it more difficult to grasp certain concepts. Specifically, understanding the weighting of walks in the Rosenbluth algorithm posed some confusion, leaving uncertainties about the probability distributions the algorithms sample from. The ESS algorithm introduced confusion in terms of direction selection and the significance of including one direction as the self-intersecting criterion. Despite expectations of the Pivot algorithm delivering faster results, our observations indicated quicker computations only when compared to longer walks in the dimerization algorithm. Nevertheless, considering its high success rate and efficient computation, the Pivot algorithm emerged as the optimal choice for generating and plotting longer walks.