Sharif University of Technology

Simulating two-dimensional lattice polymer with Monte Carlo method

Statistical mechanics II

Amirreza Vedadiyan
B.S. student
`ar.vadadian@physics.sharif.edu`

Department of Physics

May 14, 2023

Abstract

A polymer in an athermal solvent can be modeled as a self-avoiding walk (SAW) on a lattice. This model is very popular and has been used to describe the dynamic behavior of idealized polymer chains in solution both analytically and numerically. Because of its great importance in polymer science, the SAW problem has been studied by scientist and mathmathicians for more than half a century. So far, it has defied exact analytical solution even for relative short walks. In this project, we are going to simulate polymers as 2d Self Avoiding Walks on a lattice using Monte Carlo method with Pivot algorithm and try to find their properties in a good solvent.

Contents

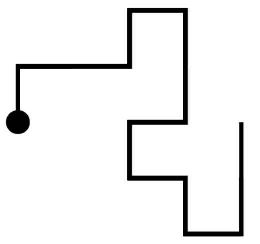
1 Introduction

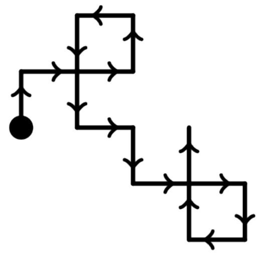
1.1 What are Polymers?

Polymers are long chain molecules consisting of a large number of units called "monomers", which are held together by chemical bonds. Due to their broad spectrum of properties, both synthetic and natural polymers play essential roles in everyday life. Polymers range from familiar synthetic plastics such as polystyrene to natural bio-polymers such as DNA and proteins that are fundamental to biological structure and function.
In the statistical mechanics we are interested mainly in universal properties, i.e. those properties that depend only on the fact that the polymer is a long linear molecule, and are determined by 'large scale quantities' such as the temperature, the quality of solvent in which the polymer is immersed, the presence of surfaces(on which a polymer an absorb) and so on.
There are two models of polymer, first the ideal one and second the real model. Ideal model assume that there are no interactions between chain monomers, But in the real model Interactions between chain monomers are modelled as excluded volume. This causes a reduction in the conformational possibilities of the chain, and leads to a self-avoiding random walk.

1.2 SAW

In mathematics, a self avoiding walk or SAW is a sequence of moves on a lattice that does not visit the same point more than once. In computational physics, a SWA is a chain-like path with a certain number of nodes, typically a fixed step length and has the property that is doesn't cross itself. A system of SAWs satisfies the so-called excluded volume condition.
Calculating the number of self-avoiding walks in any given lattice is a common computational problem. There is currently no known formula, although there are rigorous methods of approximation like statistical mechanics.


(a) A self-avoiding walk


(b) A simple random walk

1.3 First Part of this project (done)

In the first part of this project we are going to analyze three questions:

1- How many N-step self-avoiding walk exist(C_n)?

2- What is their square displacement (R_e^2)?

3- What is their gyration radius (R_g^2)?

Consider our polymer sites resting on a set of points $p = \{p_1, p_2, \dots, p_n\}$, now R_e^2 and their mean value can be calculated from:

$$R_e^2 = |p_N - p_1|^2$$

$$\langle R_e^2 \rangle = \frac{1}{c_N} \Sigma_i R_e^2(i)$$

When sum in taken over all different SAWs. For gyration radius we have:

$$R_g^2 = \frac{1}{2N^2} \Sigma_{i,j=1}^N |p_i - p_j|^2$$

$$\langle R_g^2 \rangle = \frac{1}{c_N} \Sigma_i R_g^2(i)$$

The radius of gyration of the macromolecule, R_g is more meaningful intuitively as it gives a sense of the size of the polymer coil.

There is no analytical method to calculate $C_N, \langle R_e^2 \rangle, \langle R_g^2 \rangle$ but like thermodynamics limit for very long walks $N \longrightarrow \infty$, we can use statistical methods. It is believed that all these three quantities have the asymptotic behavior:

$$c_n \approx A\mu^N N^{\gamma-1}; \quad \langle R_e^2 \rangle \approx BN^{2\nu}; \quad \langle R_g^2 \rangle \approx CN^{2\nu}$$

A,B,C and μ are some constants that depend on the type and dimension of the lattice. A,B and c are referred to as the amplitudes and μ the connective constant.

ν and γ (also called entropic exponent) but depend only on the dimension d of lattice and not a particular lattice chosen. In fact, the same scaling relations hold for walks in the continuum which explains why lattice models of polymers correctly predict the asymptotic scaling behavior of polymers.

The values of ν and γ for 2-dimensional lattice have been calculated by several authors and the results are :

$$\nu = 3/4, \quad \gamma = 43/32$$

For μ we have :

$$\mu = \log_{N \longrightarrow \infty}(c_N/c_{N-1})$$

Some of the best numerical methods estimates μ is:

$$\mu_{2d} \approx 2.63815853\dots$$

So after finding $C_n, \langle R_e^2 \rangle, \langle R_g^2 \rangle$, we are going to calculate these constants and after that we can have entropy of our system from:

$$S = k_B \ln c_n$$

2 Monte Carlo Simulation and Pivot algorithm

2.1 Monte Carlo simulation

Monte Carlo Simulations, are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. The underlying concept is to use randomness to solve problems that might be deterministic in principle. They are often used in physical and mathematical problems and are most useful when it is difficult or impossible to use other approaches. Monte Carlo methods are mainly used in three problem classes: optimization, numerical integration, and generating draws from a probability distribution. Although they might vary from case to case, the general steps to a Monte Carlo simulation are as follows:

1- Build the model. Determine the mathematical model or transfer algorithm.

2- Choose the variables to simulate. Pick the variables, and determine an appropriate probability distribution for each random variable.

3- Run repeated simulations. Run the random variables through the mathematical model to perform many iterations of the simulation.

4- Aggregate the results, and determine the mean, standard deviation and variant to determine if the result is as expected. Visualize the results on a histogram.

As we said we can't find the exact number of SAW's so we are going to use Monte Carlo simulation with Pivot algorithm.

2.2 Pivot algorithm

The Pivot Algorithm was first invented by Lal in 1969 and later studied by Neal Madras and Alan D. Sokal, which takes a Self Avoiding Walk and twists it into another Self Avoiding Walk. The process is as follows.
Let $\omega = \{\omega_0, \omega_1, \ldots, \omega_n\}$ be a Self Avoiding Walk of length n. Pick a vertex ω_i where $0 < i < n$. This divides our walk of length n into two walks of length i and $n - i$. Then , fixing veretex ω_i, apply a rotation by $+90°$ or $-90°$ or reflection (or some combination of them) to the Self Avoiding Walk $\{\omega_{i+1}, \ldots, \omega_n\}$ to get a new walk ω'.
If this walk is self avoiding, we may repeat this process on the new walk. If not we ignored it and continue with the original walk ω .
There are some useful theorems and definitions available at this article [3].

3 Results

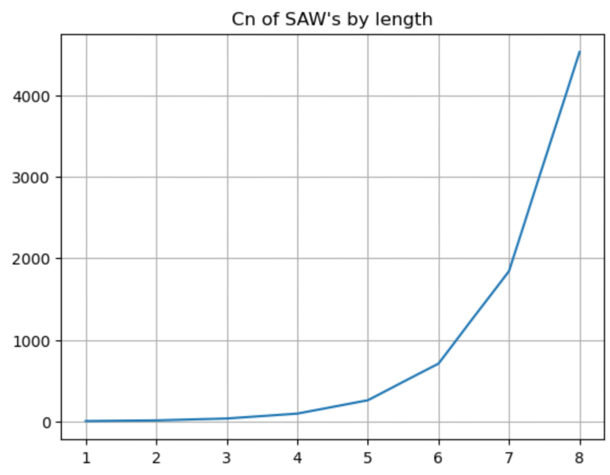
These are real values of C_n. As we see the numbers of C_n grows very fast and due to the computational complexity we did our simulations only for polymers with length of $n \leq 8$.

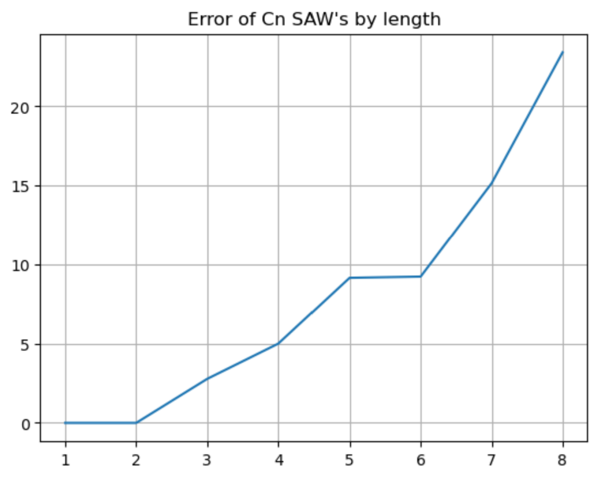
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	12	36	100	284	780	2172	5916	16268	44100	12092	324932	881500	2374444	6416596	17245332

Our Result of C_n :

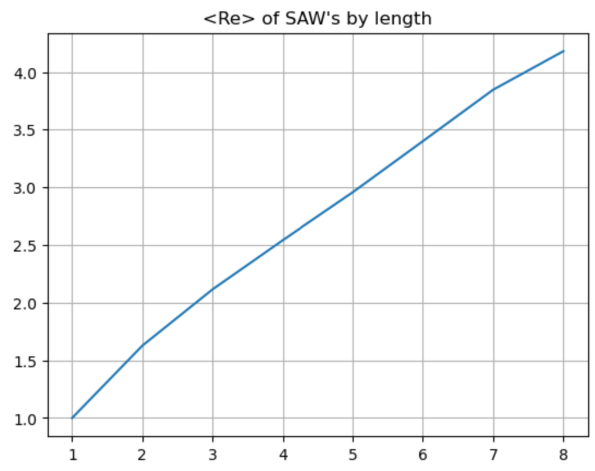
1	2	3	4	5	6	7	8
4	12	35	95	258	708	1843	4531

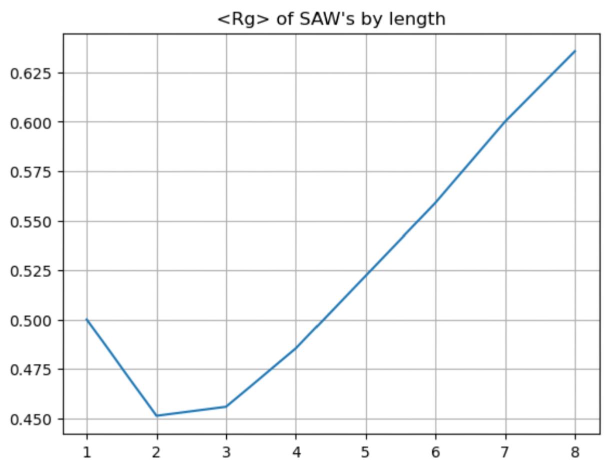
Graphs of my C_n and their error :



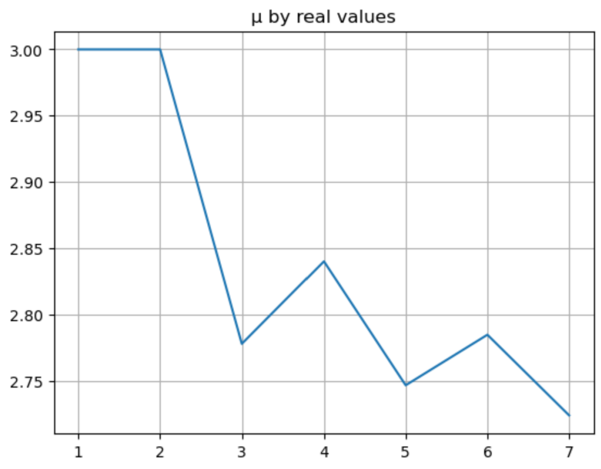


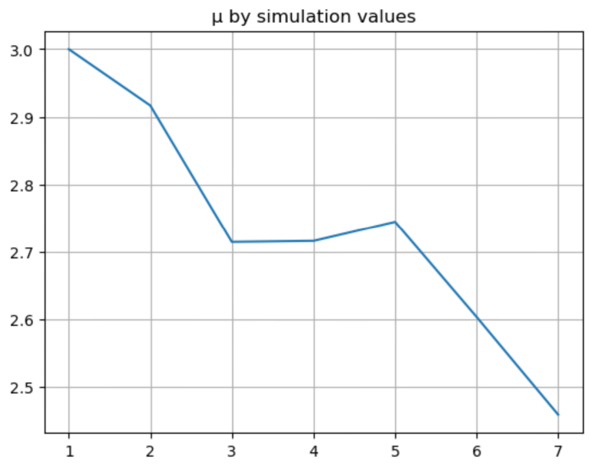
Graphs of $\langle R_e \rangle$ and $\langle R_g \rangle$ by length :



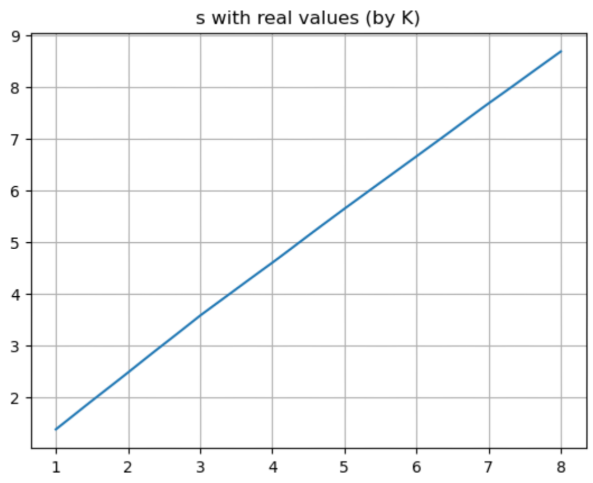


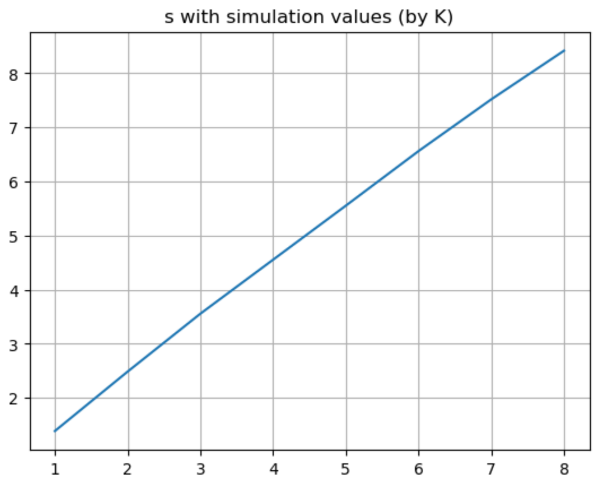
μ by real values and simulation values :





S(Entropy) by real values and simulation values :





4 Second Part (Interacting flexible polymers)

We know, In the model of polymers For a given configuration C, the total configurational energy is expressed as :

$$E(C) = -N_c\epsilon + N_b\epsilon_b - fx$$

Where N_c is the total number of nearest neighboring contacts, ϵ_b is bending energy for semiflexible polymers , N_b is the total number of turns in the configuration and $-fx$ is the energy contributed by the external force.
Now In this part we want to improve our model by taking interacting energy between monomers $\epsilon(\epsilon_b = 0, f = 0)$.
If two monomers i and j of the polymer (with $|j - i| \leq 2$) occupy nearest-neighboring sites on the square lattice, there is an attractive energy of magnitude ϵ. To do this we give a weight of $\alpha = \frac{-\epsilon}{K_B T} = -\beta\epsilon$ to our configurations.
The Partition Function is then :

$$Z_N = \Sigma_\omega e^{n(\omega)\alpha}$$

where the sum is over all SAWs ω of N steps, and n(ω) is the number of pairs of nearest-neighbor contacts which occur in ω.
The average energy is then obtained from :

$$\langle E\rangle = \frac{\partial}{\partial\beta}\ln Z_N = K_B T^2 \frac{\partial}{\partial T}\ln Z_N$$

and entropy is obtained from :

$$S = K_B \frac{\partial}{\partial T}(T \ln Z_N) = K_B \ln C_N$$

We know also the relation between R_e) and N is :

$$\langle R_N^2\rangle \propto N^{2\nu}$$

The scaling exponent ν in general may have a weak dependence on the polymer length . However, at the critical θ state, ν will be independent of N. If we plot $\langle R_N^2\rangle/N^{2\nu}$, then curves for different lengths N should intersect at the critical Temperate, So we can study phase transition of polymers and then compare our Monte Carlo simulation with real data.
** my main problem in this section is Time Complexity and Memory its taken **

5 Python Code

We use NumPy library in this project.

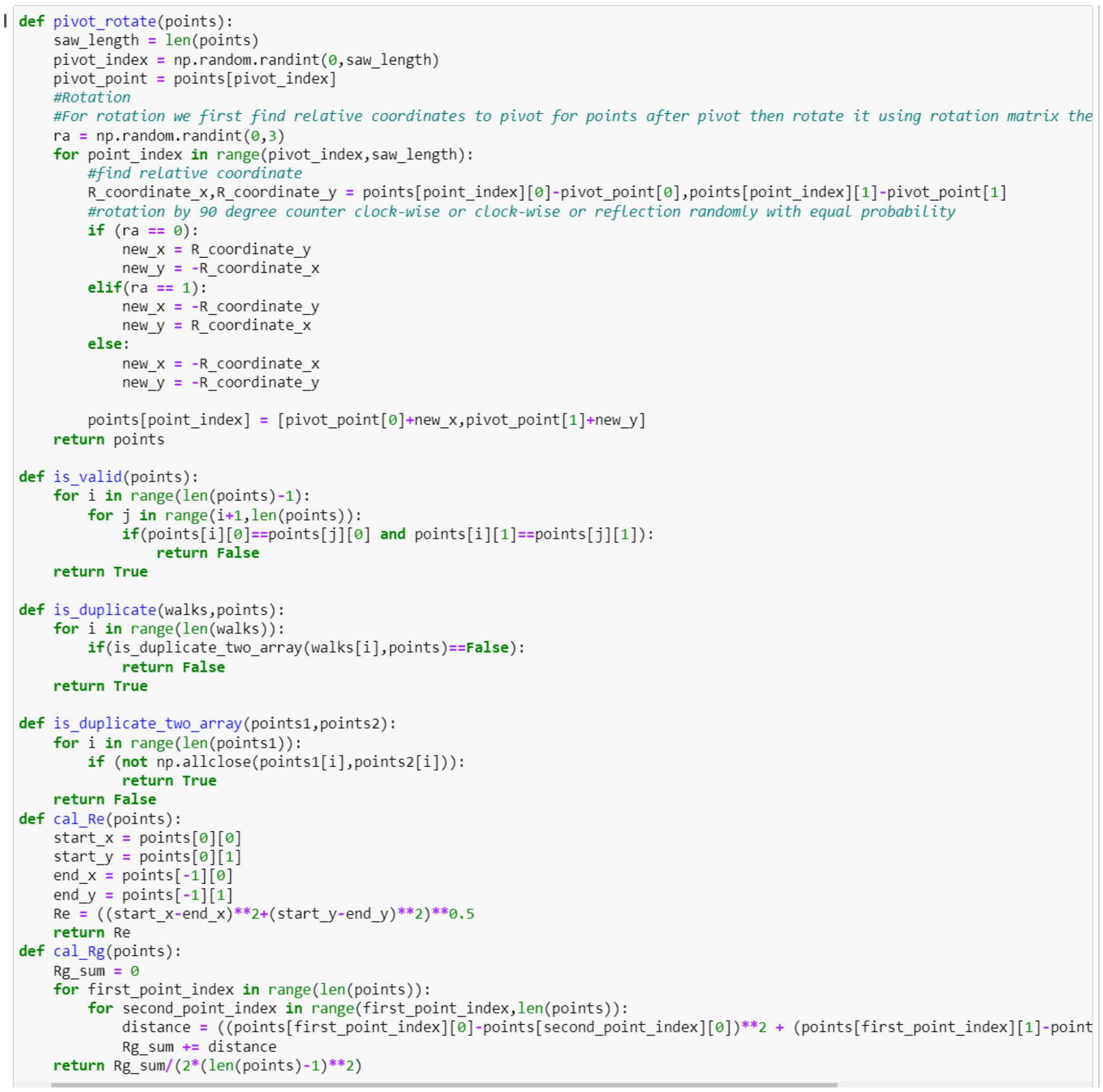
```python
import numpy as np
import pylab

def make_square_lattice(n):
    square_lattice = np.zeros((n,n),dtype='int8')
    return square_lattice
```

Function to make a random SAW

```python
def make_random_SAW(lattice,length):
    #setting up
    points = np.zeros((length+1,2),dtype='int8')
    x_dim,y_dim = lattice.shape
    p_x,p_y = x_dim//2,y_dim//2
    points[0] = [p_x,p_y]
    lattice[p_x][p_y] = 1
    #current_saw_length = 0
    for current_saw_length in range(1,length+1):
        #check if it's impossible
        if(lattice[p_x+1][p_y]==1 and lattice[p_x-1][p_y]==1 and lattice[p_x][p_y+1]==1 and lattice[p_x][p_y-1]==1):
            return (False,lattice,points)
        #it's possible so force to find it
        is_find = False
        while(not is_find):
            direction = np.random.randint(0,4)# 0-> up | 1-> down | 2-> right | 3-> left
            if(direction == 0):
                if(lattice[p_x-1][p_y] != 1):
                    lattice[p_x-1][p_y] = 1
                    p_x = p_x-1
                    is_find = True
            elif(direction == 1):
                if(lattice[p_x+1][p_y] != 1):
                    lattice[p_x+1][p_y] = 1
                    p_x = p_x+1
                    is_find = True
            elif(direction == 2):
                if(lattice[p_x][p_y+1] != 1):
                    lattice[p_x][p_y+1] = 1
                    p_y = p_y+1
                    is_find = True
            elif(direction == 3):
                if(lattice[p_x][p_y-1] != 1):
                    lattice[p_x][p_y-1] = 1
                    p_y = p_y-1
                    is_find = True
            points[current_saw_length] = [p_x,p_y]
            #print(lattice)
    return(True,lattice,points)
```

Pivot Algorithm



```python
def pivot_rotate(points):
    saw_length = len(points)
    pivot_index = np.random.randint(0,saw_length)
    pivot_point = points[pivot_index]
    #Rotation
    #For rotation we first find relative coordinates to pivot for points after pivot then rotate it using rotation matrix the
    ra = np.random.randint(0,3)
    for point_index in range(pivot_index,saw_length):
        #find relative coordinate
        R_coordinate_x,R_coordinate_y = points[point_index][0]-pivot_point[0],points[point_index][1]-pivot_point[1]
        #rotation by 90 degree counter clock-wise or clock-wise or reflection randomly with equal probability
        if (ra == 0):
            new_x = R_coordinate_y
            new_y = -R_coordinate_x
        elif(ra == 1):
            new_x = -R_coordinate_y
            new_y = R_coordinate_x
        else:
            new_x = -R_coordinate_x
            new_y = -R_coordinate_y

        points[point_index] = [pivot_point[0]+new_x,pivot_point[1]+new_y]
    return points

def is_valid(points):
    for i in range(len(points)-1):
        for j in range(i+1,len(points)):
            if(points[i][0]==points[j][0] and points[i][1]==points[j][1]):
                return False
    return True

def is_duplicate(walks,points):
    for i in range(len(walks)):
        if(is_duplicate_two_array(walks[i],points)==False):
            return False
    return True

def is_duplicate_two_array(points1,points2):
    for i in range(len(points1)):
        if (not np.allclose(points1[i],points2[i])):
            return True
    return False
def cal_Re(points):
    start_x = points[0][0]
    start_y = points[0][1]
    end_x = points[-1][0]
    end_y = points[-1][1]
    Re = ((start_x-end_x)**2+(start_y-end_y)**2)**0.5
    return Re
def cal_Rg(points):
    Rg_sum = 0
    for first_point_index in range(len(points)):
        for second_point_index in range(first_point_index,len(points)):
            distance = ((points[first_point_index][0]-points[second_point_index][0])**2 + (points[first_point_index][1]-point
            Rg_sum += distance
    return Rg_sum/(2*(len(points)-1)**2)
```

Calculations

```
iteration = 5000
maximum_length = 2
result = []
Re_result = []
Rg_result = []
for length in range(1,maximum_length):
    #Save path's for diffrent lengths of our polymer
    mFile = open(mFile_name, "a")
    mFile_name = 'array_result_length' + str(length)
    square_lattice = make_square_lattice(2*length+1)
    saw = make_random_SAW(square_lattice,length)
    mFile.write(str(saw[2])+',')
    saw_numbers = 1
    Re_sum = cal_Re(saw[2])
    Rg_sum = cal_Rg(saw[2])
    walks = []
    walks.append(saw[2])
    temp_point = np.array(saw[2])
    for i in range(iteration):
        copy_points = temp_point.copy()
        new_saw = pivot_rotate(copy_points)
        if(is_valid(new_saw) and is_duplicate(walks,new_saw)):
            walks.append(new_saw)
            temp_point = new_saw
            Re_sum += cal_Re(temp_point)
            Rg_sum += cal_Rg(temp_point)
            saw_numbers = saw_numbers+1
            mFile.write(str(temp_point)+',')
    Re_result.append([length,Re_sum/saw_numbers])
    Rg_result.append([length,Rg_sum/saw_numbers])
    result.append([length,saw_numbers])
    mFile.close()

print(result)
print(Re_result)
print(Rg_result)
```

My code is available at : https://github.com/amirvedadiyan/Polymer-simulation

References

[1] Polymers modeled as Self Avoiding Walks on Lattices, https://polymerdatabase.com/polymer%20physics/SAW.html#:~:text=A%20polymer%20in%20an%20athermal,solution%20both%20analytically%20and%20numerically..

[2] Self-avoiding walk https://en.wikipedia.org/wiki/Self-avoiding_walk.

[3] ISAAC OTTONI WILHELM - Counting Self Avoiding Walks Of Length N http://www.math.uchicago.edu/~may/VIGRE/VIGRE2009/REUPapers/Wilhelm.pdf