Self Avoiding Random Walk

Cihan Cicek
cicek154@gmail.com

February 20, 2021

Contents

***Important note:**

This project has prepared for "Advanced Monte Carlo Methods" lecture in winter term 2020/2021, Heidelberg University.

Instead of adding same citations of the book in each part, I hereby declare that almost all of the explanations and formulas are taken from the book of "The Self-Avoiding Walk, Neal Madras and Gordon Slade, 1996". [1]

Only python codes are implemented by myself.

1 Preparation

A self-avoiding walk (SAW) on a graph is a walk that never visits the same vertex twice. it is typical example of non-Markov random walks on graphs. This most basic and fully random process gives us usefull model to apply numerous type of statistical physics problems

One of the basic questions concerning random walks are:
- What is the asymptotic behavior of the walk as the number of steps tends to infinity? To be more specific, if X(N) denotes the location of the walker starting at the origin after N steps, does the mean square displacement show a power behavior? In other words, does the following hold in some sense?

$$E[|X(n)|^2] \sim N^{2\nu}$$

where $|X(N)|$ denotes the Euclidean distance from the starting point and ν is a positive constant. If it is the case, what is the value of the displacement exponent ν?

The question originated from the problem of the end-to-end distance of long polymers. Since no two monomers can occupy the same place, a self-avoiding walk is expected to model polymers. [2]

These properties of SAW have been studied in various papers and There has been numerous different numerical aproaches for mean square displacement estimation and the exponent ν calculations. In this paper I will try to estimate ν parameter by using linear regrassion method and for Slitherin Snake Length conserved SAW algorithm.

To be able to carry out that I need some standart Python Libraries:

1

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import sparse
from matplotlib.ticker import PercentFormatter
import sys
import time
import random
import copy
```

[1]Very usefull reading: https://www.physicsforums.com/insights/fun-self-avoiding-walks/

```
print (sys.getrecursionlimit()) ## to see current recursionlimit

def recursionlimit(recursionlimit): # set recursionlimit to another value
    return sys.setrecursionlimit(recursionlimit)

recursionlimit(20000)
print('original recursionlimit was 3000')
sys.getrecursionlimit()

#printed
##############################

20000
original recursionlimit was 3000

20000
```

2 Elementary Simple Sampling (ESS)

This algorithm generates ordinary simple random walks until it obtains an N-step walk that is self-avoiding.

The code:

1. Let $\omega(O)$ be the origin and set i = 0.

2. Increase i by one. Choose one of the 2d neighbours of $\omega(i-1)$ at random, and let $\omega(i)$ be that point.

3. If $\omega(i) = \omega(j)$ for some j = 1,2, ... N-1. then go back to Step 1. Otherwise go to Step 2 if i < N , and stop if i = N.

the walk W = $(\omega(0), ..., \omega(N)\ )$ is selfavoiding. We claim that for any $\omega\ \epsilon\ S_N$, we have Pr(W=ω)=$\frac{1}{c_N}$

To see this, let S_N be the set of all N step (ordinary) simple walks. If we keep choosing members of S_N uniformly at random until one of them is in S_N , then the final result is evidently uniformly distributed on S_N.

– the probability that an N-step simple random walk is self-avoiding; $\frac{C_N}{(2d)^N}$, the $(2d)^N$ term is the number of all possible simple random walks. so the expected number of attempts (i.e.returns to Step 1) is $\frac{(2d)^N}{C_N}$ Therefore, using the notation T_X to represent the expected amount of computer time required for algorithm X to generate a single N -step self-avoiding walk, we have,

$$T_{ESS} = (\frac{2d}{\mu})^{N+o(N)}$$

We can improve on the efficiency of ESS by only generating simple random walks with no immediate reversals.

```
def ESS_SAW(N):

    # Possible directions
    deltas = [[1,0], [0,1], [-1,0], [0,-1]]

    # container
    a = [0,0]
    wi = []
    for j in range(N+1):
        wi.append([0,0])
    w = wi

    # Main
    for i in range(N):

        # randomly chosen step
        dw= deltas[np.random.randint(0,4)]
        a[0] = w[i][0]+ dw[0]
        a[1] = w[i][1]+ dw[1]

        # whether SAW or not
        if a in w:
            #if not call again the same function recursively
            w = ESS_SAW(N)
            break
        else:
            w[i+1][0] =  a[0]
            w[i+1][1] =  a[1]
    return w
```

```

def plot_ess_SAW(N, fnc):

    """
    Plots the output of the ESS_SAW algorithm

    Args:
        N (int): the length of the walk
    Returns:
        Plot of the output of the ESS_RAW algorithms
    """

    s_time=time.time()
    w = fnc(N)
```

```
    e_time=time.time()
    print('Computation time:',e_time-s_time, 's')
    x =[]
    y =[]
    for i in range(N+1):
        x.append(w[i][0])
        y.append(w[i][1])
    plt.figure(figsize = (10, 10))
    plt.title(str(fnc)+ 'Lenght N =' + str(N), fontsize=14, fontweight='bold',
    y = 1.05)
    plt.plot(x, y, 'bo-', linewidth = 1)
    plt.plot(0, 0, 'go', ms = 12, label = 'Start')
    plt.plot(x[-1], y[-1], 'ro', ms = 12, label = 'End')
    plt.axis('equal')
    plt.legend(fontsize=15)
    plt.show()
    plt.savefig('plot_ess_SAW.png')
```

```

plot_ess_SAW(15,ESS_SAW)
```

Computation time: 0.0007064342498779297 s

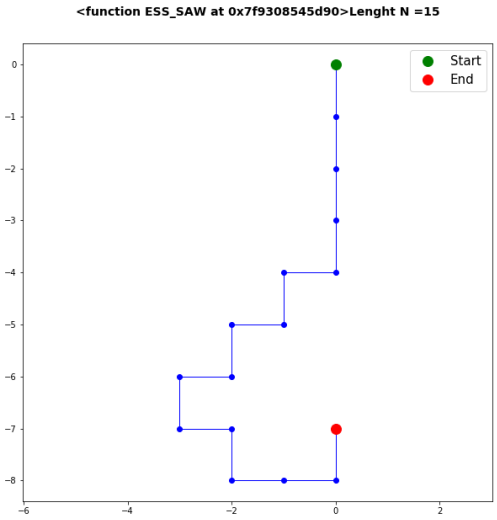


Figure 1: ESS SAW.

3 The Slithering Snake Algorithm

The Slithering Snake length-conserving dynamic algorithm was devised by Kron (1965) and by Wall and Mandel (1975) [see also Kron et at. (1967) and Mandel (1979)]. The basic move of the algorithm is to remove a bond from one end of the current walk while simultaneously trying to add a bond to the other end (rejecting the result if it is not self-avoiding). For an explicit description, use the following procedure as Step 2 in the Generic Fixed-Length Dynamic Algorithm.

1) Generate a random variable X which equals 0 with probability 1/2 and equals N with probability 1/2.

2) If X = 0, then let Y be one of the 2d nearest neighbours of $\omega^{[t]}(0)$ (chosen uniformly at random)

3) and set $\tilde{\omega} = (Y, \omega^{[t]}(0), ..., \omega^{[t]}(N-1))$

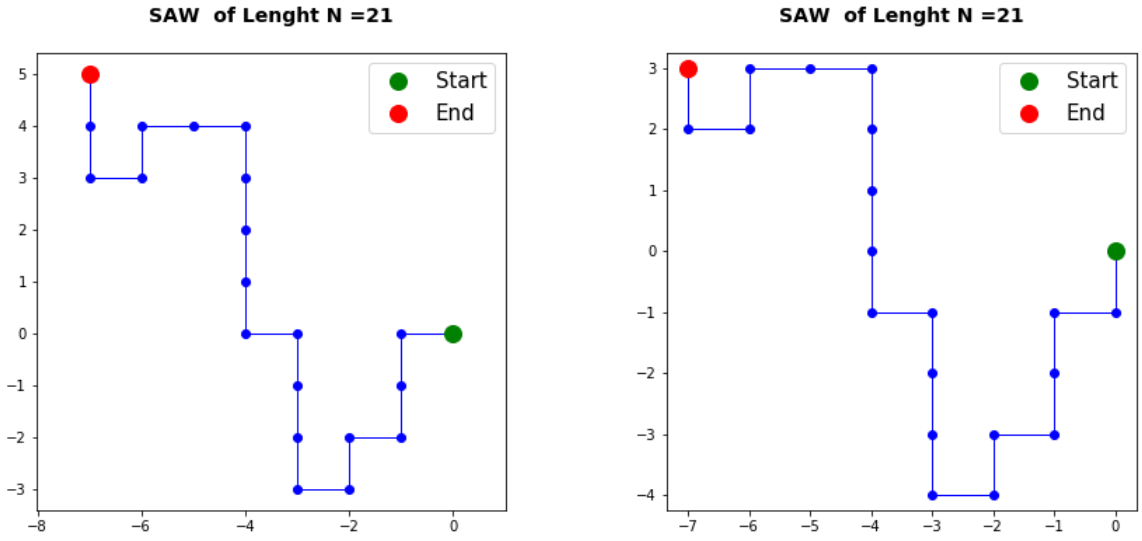
4) If X = N then let Y be one of the 2d nearest neighbours of $\omega^{[t]}(N)$

5) and set $\tilde{\omega} = (\omega^{[t]}(1), ..., \omega^{[t]}(N), Y)$

```
1
def slithering_snake(w):
3    '''
     by choosing randomly one of the end (first or last) point of a given w-
     SAW
5
     is going to change with new nearest neighbours of the point
7
     so that produce a new SAR w_new.
9
     Args:
11        w: (x, y) (list , list) SAW
13   Returns:
15        w_new: (x, y) (list , list) SAW
17   '''
     N = len(w)
19
     # take randomly 0 or N
21
     x = random.choice([0,N])
23
     # SAW possible steps
25
     deltas = [[1,0], [0,1], [-1,0], [0,-1]]
27
     # get one of them randomly
29
     dw= deltas[np.random.randint(0,4)]
31
     w_new = copy.deepcopy(w)
```

```
33
    wnew =[]
35
    # change first point of SAW, w
37
    if x == 0:
39
        # delete the last element of w
41
        del w_new[-1]
43
        # add first step to do 3)
45
        w_new[0:0] = [dw]
47
    else:
49
        # change last point of SAW, w
51
        del w_new[0]
53
        w_new.append([w_new[-1][0] + dw[0],w_new[-1][1] + dw[1]])
55

57  # check SAW or not

59  # to do that create a new list which has only unique elements
    b=[]
61
    for [a,c] in w_new:
63  # Add to the new list
    # only if not present
65      if [a,c] not in b:
            b.append([a,c])
67
    if len(b) != len(w_new):
69      # print('not saw')
        wnew = slithering_snake(w)
71
    else:
73      # print('saw')
    #translate all points to be first step (0,0)
75
        # get the first element of the list
77
        f = copy.deepcopy(w_new[0])
79
        # translate so that it begins at the origin.
81
        for [a,b] in w_new:
83          wnew.append([a-f[0],b-f[1]])

85  return wnew
```

```
def plot_cicek(w):
    x =[]
    y =[]
    for i in range(len(w)):
        x.append(w[i][0])
        y.append(w[i][1])
    plt.figure(figsize = (6, 6))
    plt.title('SAW  of Lenght N =' + str(len(w)), fontsize=14, fontweight='
    bold', y = 1.05)
    plt.plot(x, y, 'bo-', linewidth = 1)
    plt.plot(0, 0, 'go', ms = 12, label = 'Start')
    plt.plot(x[-1], y[-1], 'ro', ms = 12, label = 'End')
    plt.axis('equal')
    plt.legend(fontsize=15)
    plt.savefig('plot_cicek2.png')
    plt.show()

w = ESS_SAW(20)
ws = slithering_snake(w)

print(ws)

plot_cicek(ws)

#printed
##################################

[[0, 0], [-1, 0], [-1, -1], [-1, -2], [-2, -2], [-2, -3], [-3, -3], [-3, -2],
   [-3, -1], [-3, 0], [-4, 0], [-4, 1], [-4, 2], [-4, 3], [-4, 4], [-5, 4],
   [-6, 4], [-6, 3], [-7, 3], [-7, 4], [-7, 5]]
```



(a) slithering snake. (b) slithering snake2.

Figure 2: For a given SAW two different results of Slithering Snake

```
plot_cicek(w)
```

4 Irreducibility

The nature of these moves has earned this algorithm and its variants the names ”slithering snake” and ”reptation” (the latter term is also used in polymer dynamics to describe similar motions of real polymers.) This algorithm is reversible, but it is not irreducible:for example a given ESS SAW below:

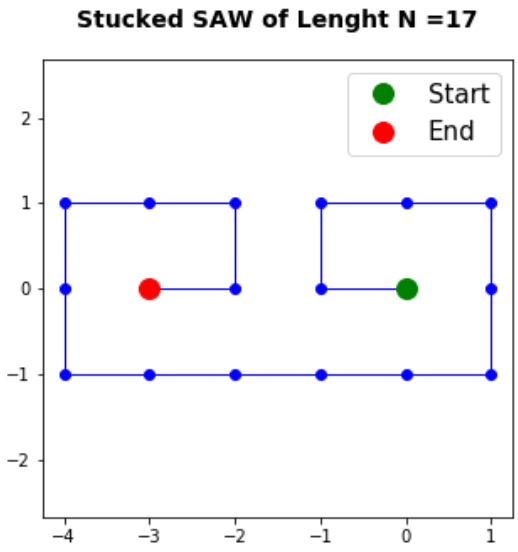


Figure 3: Stuck.

the walk is frozen with respect to the slithering-snake algorithm in Z^2. In fact, for sufficiently large N, it turns out that a positive fraction of all N-step walks are frozen, because there is a positive probability that both ends of the walk are "trapped" and cannot be extended by a single step in any direction.

To be more precise, let Φ_N denote the set of all walks in S_N which are frozen with respect to the slithering-snake algorithm (that is, ω is in Φ_N if and only if the ergodicity class containing ω has cardinality one). let P be a proper front pattern with the property that the 2d nearest neighbours of the first site of P are all sites of P.Let R be the walk whose sites are the sites of P in reverse order Then any self-avoiding walk that begins with the pattern P and ends with the pattern R must be frozen $S_N(P, R) \subset \Phi_N$. Therefore;

$$lim_{N->\infty} \frac{|\Phi_N|}{c_N} > 0$$

More detailed discussion, (Madras and Slade, 1996, p. 320)

5 Mean Square Displacement

Denoting expectation with respect to the uniform measure by angular brackets, the average distance (squared) from the origin after N steps is then given by the mean-square displacement

$$< |\omega(N)|^2 >= \frac{1}{c_N} \Sigma_{\omega:|\omega|=N} |\omega(N)|^2$$

Where C_N denote the number of N-step self-avoiding walks beginning at the origin.

Like C_N, the mean-square displacement can also be calculated by hand for very small values of N, but the combinatorics quickly become intractable as N increases.

It is instructive to compare the behaviour of the self-avoiding walk with that of the simple random walk. An N-step simple random walk on Z^d starting at the origin, is a sequence $\omega = (\omega(0), \omega(1), ..., \omega(N))$ of sites with $\omega(0) = 0$ and $|\omega(j+1) - \omega(j)| = 1$, with the uniform measure on the set of all such walks. Without the self-avoidance constraint the situation is rather easy. Indeed, since each site has 2d nearest neighbours, the number of N -step simple random walks is exactly $(2d)^N$. To analyse the mean square displacement, we represent the simple random walk in the following way. Let $X^{(i)}$ be independent and identically distributed random variables with $X^{(i)}$ uniformly distributed over the 2d (positive and negative) unit vectors. Then the position after N steps can be represented as the sum $S_N = X^{(1)} + X^{(2)} + ... + X^{(N)}$. Expanding $|S_N|^2$ the mean-square displacement is given by

$$< |S_N|^2 >= \sum_{i,j=1}^{N} < X^{(i)}.X^{(j)} >$$

For $i \neq j$, $< X^{(i)}.X^{(j)} >= 0$, using independence and the fact that $< X^{(i)} >= 0$. Since $< X^{(i)}.X^{(j)} >= 1$, it follows that the mean-square displacement is equal to N. Similarly, if we consider a random walk in Z^d in which steps lie in a symmetric finite set $\Omega \subset Z^d$ of cardinality $|\Omega|$, with each possible step equally likely, then the number of N-step walks is $|\Omega|^N$ and the mean-square displacement is $N\sigma^2$ where σ^2 is the mean-square displacement of a single step. (in our question it is eqal to "1")

For the self-avoiding walk it is believed that there is exponential growth of C_N with power law corrections, unlike the pure exponential growth of the simple random walk. It is also believed that the mean-square displace- ment will not always be linear in the number of steps, in contrast to the diffusive behaviour of the simple random walk. These beliefs are in harmony with known properties of other models of statistical mechanics, and are supported by numerical and nonrigorous calculations. The conjectured behaviour of C_N and $< |\omega(N)|^2 >$ is thus:

$$C_n \sim A\mu^N N^{\gamma-1}$$

and,

$$< |\omega(N)|^2 > \sim DN^{2\nu}$$

(Madras and Slade, 1996, p. 3,4,5)

[2]

[2]to see more detailed discussion, (Madras and Slade, 1996, p. 292)

```
def S(w):
    '''calculate displacement square dx**2 + dy**2 for w'''

    S = w[-1][0]**2 + w[-1][1]**2

    return S
```

```
def mean_square_displacement(times,w):
    '''
    Args:

        times: integer , number of desired samples

        w: (x, y) (list , list) SAW

    Returns:

        s_n/times: float type Mean Square

    '''
    # print('given SAW w:')
    # print('------------')
    # print(w)

    # s_time=time.time()

    # by using slithering_snake method produce a new SAW
    w123 = slithering_snake(w)


    s_n = 0.0 # for mean

    for i in range(times):
        w123 = slithering_snake(w123)
        s_n +=S(w123)

    # e_time=time.time()

    # print('Computation time:',e_time-s_time, 's')
    return s_n/times
```

```
mean_square_displacement(10000,w)

#printed
##################################
```

```
72.2278
```

```
def main_cicek(N,times):

    #times: number of samples

    S_N =[]

    print('Computation time  N step size ESS_SAW')

    #s_time3=time.time()

    for i in range(2,N+1): #to create from 2 step to N step SAW by using
    ESS_SAW function

        s_time2=time.time()

        # ESS_SAW algorithm might give RecursionError
        # then try it again
        while N > 0:
            try:
                w = ESS_SAW(i)
                break
            except RecursionError:
                pass

        e_time2=time.time()

        print("{:.5f}".format((e_time2-s_time2)*100), 's                    ' +
    str(i), end="\r", flush=True)

        # for N<11 step SAWs, it is enough to work with 10000 samples
        if i < 11 and times>20000:
            Sl = mean_square_displacement(20000,w)
            S_N.append(Sl)
        else:
            # calculate mean_square_displacements
            Sl = mean_square_displacement(times,w)
            S_N.append(Sl)

            #e_time3=time.time()

    #print(e_time3-s_time3, ' s        ')

    return S_N
```

```

def plot_cico(N,times):
    '''Possible improvement is that one can create times list,
```

```
    because it does not need too much samples for few steps.
    Actually the mean can easily stabilize N<10 step SAW'''

    s_time=time.time()

    Ni = np.arange(2,N+1,1) # contains different N steps, from 1 to N
    y = main_cicek(N,times)

    plt.figure(figsize = (7, 7))
    plt.plot(Ni, y)
    plt.title( 'SAWs of Length up to = ' + str(N) + '   ( Mean calculated by '
    + str(times) + ' samples SAW )' ,fontsize=14, fontweight='bold', y =
    1.05)
    plt.ylabel('S_N mean')
    plt.xlabel('N')
    plt.show()
    plt.savefig('plot_cico.png')

    e_time=time.time()

    print(' Totat Computation time:',(e_time-s_time)*100, 's')

plot_cico(25,100000)

#printed
###################################

Computation time  N step size ESS_SAW
16.35380 s                 25
```

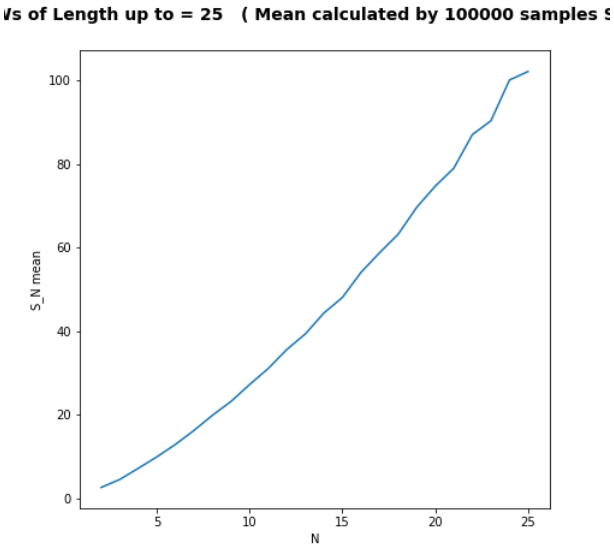


Figure 4: SAWs of Mean Square Length up to 25 steps.

Totat Computation time: 20481.652569770813 s

6 Estimating v with Linear Regression

On a double-logarithmic scale it can be seen more precisely

$$< |\omega(N)|^2 > \sim DN^{2\nu}$$

$$Y = DN^{2\nu}$$

$$log(Y) = log(D) + 2\nu log(N)$$

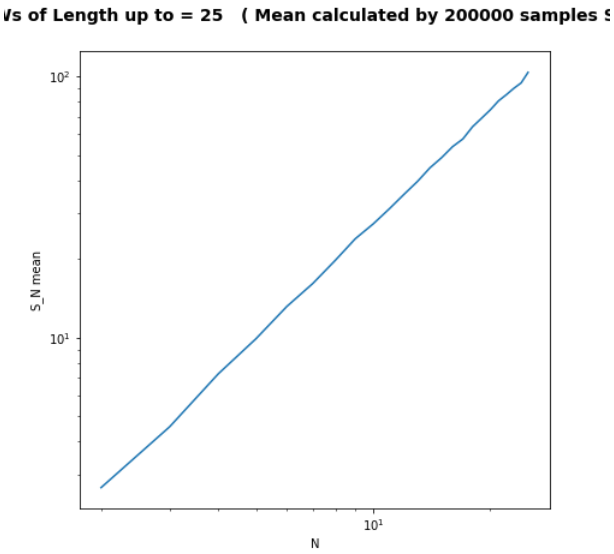


Figure 5: SAWs of Mean Square Length up to 25 steps (log. scaled).

Computation time N step size ESS SAW 45.16487 s 25

```
1
    # linear fit
3 from sklearn.linear_model import LinearRegression

5 N=25
  times = 200000
7
  # get mean square displacements up to N steps
9 S_N = main_cicek(N, times)

11 # in logarithmic scale Number of steps
```

```
x = np.log(np.arange(2,N+1,1)).reshape((-1, 1))

# convert S_N into log scale
y = np.log(S_N)

model = LinearRegression()

model.fit(x, y)

'''
Once you have your model fitted, you can get the results to check

whether the model works satisfactorily and interpret it.
'''
r_sq = model.score(x, y)
print('coefficient of determination, (R^2 score):', r_sq)

print('-----------------------------------------------------')

print('intercept (log(D) value) :', - model.intercept_)

print('-----------------------------------------------------')

print('slope (2v value) :', model.coef_)

print('-----------------------------------------------------')

print('v, exponent :', 0.5*model.coef_)
```

7 Conclusion

```
coefficient of determination, (R^2 score): 0.9997542016816572
-----------------------------------------------------
intercept (log(D) value) : 0.05309610360375849
-----------------------------------------------------
slope (2v value) : [1.45570481]
-----------------------------------------------------
v, exponent : [0.7278524]
```

We have very efficient model with $R^2 = 0.99$ and the ν value is "0.7278524", which is very close to current value of the exponent. (The current value of the exponent 0.7766(5) according to "N Fricke and W Janke, J. Phys. A: Math. Theor. 50 (2017) 264002")

Possible improvement would be by changing our initial SAW (ESS_SAW) we could avoid stucking cases (irreducibility), but in the case we need to get same end to end distance SAW's (or they can only differ in our error bar in our exponent value), but this gets more and more

computational time, because after getting each successfully N step SAW, we need to check whether they have same lenght.

8 Appendix

A One more example "Verdier and Stockmayer (1962)"

This algorithm turns one self-avoiding walk into another by moving one or two bonds of the walk. Briefly, it picks a site at random and tries to "flip" the two incident bonds if they form a right angle (or tries to wiggle the end bond if the chosen site is an endpoint of the walk).

1. Let $\omega^{[0]}$ be any self-avoiding walk in S_N. Set t = 0.

2. Choose an integer I uniformly at random from 0, 1, ... , N.

3. Define a new walk $\tilde{\omega} = (\tilde{\omega}(0), ..., \tilde{\omega}(N))$, which is not necessarily self-avoiding, as follows. First set $\tilde{\omega}(l) = \omega^{[t]}(l)$ for all $l \neq I$. Then:

   (a) if 0 ¡ I ¡ N , then set $\tilde{\omega}(I) = \omega^{[t]}(I-1) + (\omega^{[t]}(I+1) - \omega^{[t]}(I))$;

   (b) if I = N, then set $\tilde{\omega}(N)$ equal to any neighbour of $\omega^{[t]}(N-1)$ except for $\omega^{[t]}(N-2)$ and $\omega^{[t]}(N)$, chosen at random;

   (c) if I = 0, then set $\tilde{\omega}(0)$ equal to any neighbour of $\omega^{[t]}(1)$ except for $\omega^{[t]}(0)$ and $\omega^{[t]}(2)$, chosen at random. Then translate $\tilde{\omega}$ so that it begins at the origin.

4. If $\tilde{\omega}$ is self-avoiding, then set $\omega^{[t+1]} = \tilde{\omega}$; otherwise, set $\omega^{[t+1]} = \omega^{[t]}$

5. Increase t by one and go to Step 2.

```
1
3     '''
      The following code does not contains 4th and 5th parts,
5 it is testing algorithm to see what it makes.

7 The complete code will be given after plot func.

9 '''


11
def V_S_SAW_test(N,t): # N step RAW, t Sample
13     deltas = [[1,0], [0,1], [-1,0], [0,-1]] # directions
       wr =  ESS_SAW(N) # get self avoiding walk
15     wr = np.array(wr)
       S = np.zeros((t,N+1,2),dtype= np.int64) # container
17     S[0] = wr.copy() # get first RAW into 0th sample

19     # t samples
       for j in range(t):
21         I = np.random.randint(0,N+1) #  I uniformly at random from {0, 1, ...
      , N}.
           w_new = S[j].copy()
23         if N > I > 0:
               w_new[I,:] = S[0,I-1,:] + S[0,I+1,:] - S[0,I,:] # (x,y)
25         elif I == N:
               # get f_deltas (feasible deltas)
27             x1 = S[j,N,0] - S[j,N-1,0] # S[j,N,:] - S[j,N-1,:]
               y1 = S[j,N,1] - S[j,N-1,1] #
```

```
            r1 = [x1,y1]
            f_deltas = [s for s in deltas if s != r1]
            x2 = S[j,N-2,0] - S[j,N-1,0]
            y2 = S[j,N-2,1] - S[j,N-1,1]
            r2 = [x2,y2]
            f_deltas = [s for s in f_deltas if s != r2]
            dw = f_deltas[np.random.randint(0,2)]
            w_new[N,:] = S[j,N-1,:] + dw[:]
        elif I == 0:
            x3 = S[j,0,0] - S[j,1,0]
            y3 = S[j,0,1] - S[j,1,1]
            r3 = [x3,y3]
            f_deltas = [s for s in deltas if s != r3]
            x4 = S[j,2,0] - S[j,1,0]
            y4 = S[j,2,1] - S[j,1,1]
            r4 = [x4,y4]
            f_deltas = [s for s in f_deltas if s != r4]
            dw1 = f_deltas[np.random.randint(0,2)]
            w_new[0,:] = S[j,1,:] + dw1[:]
            dx = w_new[0]  # following steps for translate so that it begins
    at the origin.
            for k in range(N+1):
                w_new[k,:] = w_new[k,:] - dx[:]
    # wr is the original RAW and w_new after applying the algorithm.
    return wr, w_new
```

```
def plot_SSAW(N,t):
    """
    Plots the output of the ESS_SAW algorithm

    Args:
        N (int): the length of the walk
    Returns:
        Plot of the output of the ESS_SAW algorithms
    """
    s_time=time.time()
    w,w2 = V_S_SAW_test(N,t)
    e_time=time.time()
    print('Computation time:',e_time-s_time, 's')
    x =[]
    y =[]
    for i in range(N+1):
        x.append(w[i][0])
        y.append(w[i][1])
    x1 =[]
    y1 =[]
    for i in range(N+1):
        x1.append(w2[i][0])
        y1.append(w2[i][1])
    plt.figure(figsize = (10, 10))
    plt.title('Lenght N =' + str(N) + '  Blue arrows indicate original SAW',
```

```
    fontsize=14, fontweight='bold', y = 1.05)
27   plt.plot(x, y, 'bo-', linewidth = 1)
     plt.plot(x1, y1, 'yo-', linewidth = 1)
29   plt.plot(0, 0, 'go', ms = 12, label = 'Start')
     plt.plot(x[-1], y[-1], 'ro', ms = 12, label = 'End')
31   plt.plot(x1[-1], y1[-1], 'ro', ms = 12, label = 'End')
     for i in range(N): # in particular we dont need arrows, it will increase
    our calculation time, ( one extra loop)
33       plt.arrow((x[i]+x[i+1])/2,(y[i]+y[i+1])/2, (x[i+1] - x[i])*0.01, (y[i
    +1]-y[i])*0.01 , shape='full', lw=0, length_includes_head=True, head_width
    =.1)
     plt.axis('equal')
35   plt.legend(fontsize=15)
     plt.savefig('plot_SSAW22.png')
37   plt.show()

39 plot_SSAW(20,1) # only one sample. this is for testing and to see what makes
    the algorithm
```

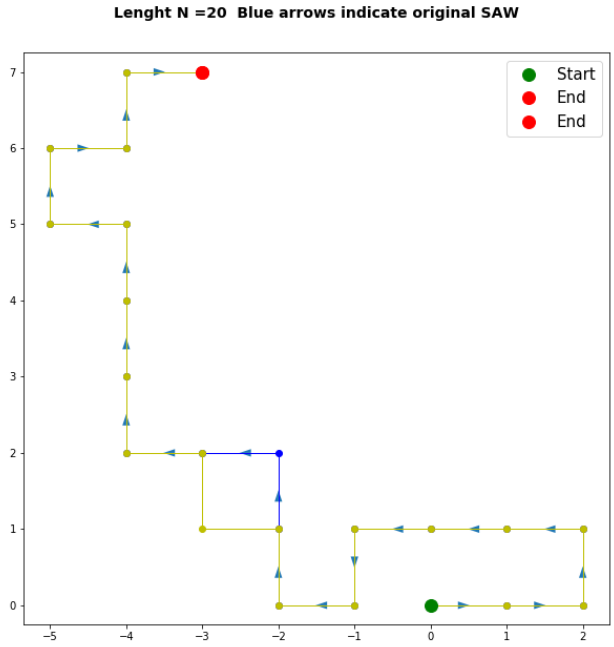


Figure 6: Verdier and Stockmayer (1962).

B Chronological History[1]

Rosenbluth (1955), biased sampling, Section 9.3.1, up to 64 steps, 0.61 for v

Stellman and Gans (1972), a continuum version of the pivot algorithm, Section 9.4.3, up to 298 steps, 0.610 $\pm$0.008 for ν

Grishman (1973), a combination of the dimerization and enrichment algorithms, Sections 9.3.2 and 9.3.3, 500 steps, 0.602 $\pm$ 0.009

However, these early results, which used relatively short walks, are biased by significant systematic errors due to unincluded correction to scaling terms. (Section 9.2.1)

Rapaport (1985), a combination of dimerization and enrichment, Sections 9.3.2 and 9.3.3, 2400 steps, 0.592$\pm$0.004

Madras and Sokal (1988), the pivot algorithm, Section 9.4.3, 3000 steps, 0.592$\pm$0.003,

Li and Sokal, recently, pivot algorithm, Section 9.4.3, 80,000 steps, 0.5883 $\pm$ 0.0013

**** which is in remarkable agreement with the field theoretic renormalization group prediction of 0.5880 $\pm$ 0.0015 obtained by Le Guillou and Zinn-Justin (1989).

**** p.243, Table 11.1 Critical exponents for the Ising model and the Heisenberg model, Andreas Wipf Statistical Approach to Quantum Field Theory An Introduction (2012)

References

[1] Neal Madras and Gordon Slade, 1996. *The Self-Avoiding Walk.*

[2] Kumiko Hattori, Noriaki Ogo and Takafumi Otsuka, 2018. *A family of self avoiding random walks interpolating the loop erased random walk and a self avoiding walk on the Sierpiski gasket.*