Lab 3: Random processes

SI1136 Simulation and Modeling

Axel Strömberg

January 15, 2019

Contents

3.1

3.1 Project

Use the template and add the decision of moving a particle. Because each particle has the same chance to go through the hole, the probability that a particle goes from left to right equals the number of particles on the left divided by the total number of particles, that is $p = n/N$. Thus:

a) Generate a random number r from a uniform distribution between 0 and 1.

b) If $r \leq p = n/N$, move a particle from left to right $n \to n - 1$, otherwise to the left $n \to n + 1$.

c) Run simulations for $N = 8, 64, 800$. Does the system reach equilibrium? What is your qualitative criterion for equilibrium? Does n, the number of particles on the left-hand side, change when the system is in equilibrium?

3.1 Background

An equilibrium can, vaguely, be described as:

Equilibrium (1)

The condition of a system in which all competing influences are balanced.

In our example we give the following qualitative criterion for an equilibrium:

Equilibrium (2)

A point in a system where a small perturbation from the equilibrium causes the a change to the system so that it will have a higher probability to move back to the equilibrium with than to move away from it.

3.1 Method i

a) A random $r \in [0, 1]$ is generated after importing.

```
1 r = rnd.random()
```

3.1 Method ii

b) Moving particles left $\leftrightarrow$ right is done by.

```
p = nLeft/nTot
r = rnd.random()
if r < p:
    nLeft -= 1
    nRight += 1
elif r >= p:
    nLeft += 1
    nRight -= 1
```

3.1 Method iii

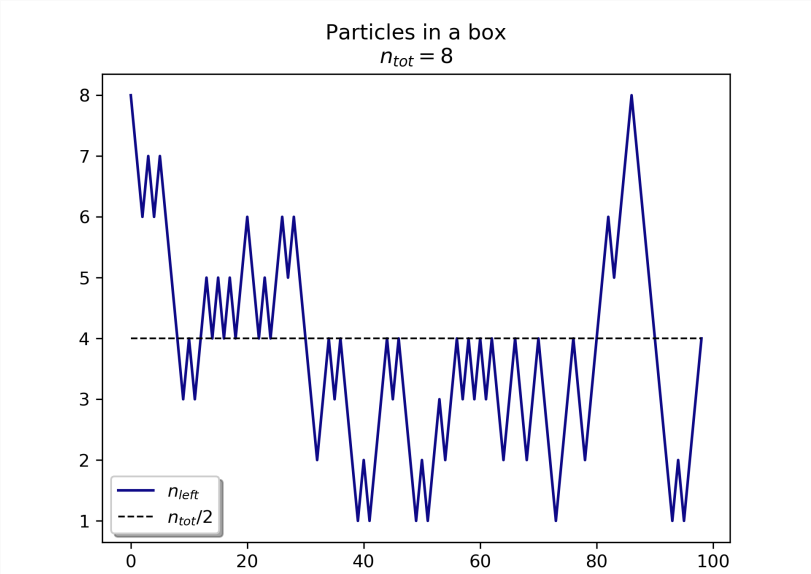
c) 10 000 simulations are run. The first one for each N is plotted. All simulations are then plotted in a histogram. With an even number of steps $n_{\text{end, left}}$ will be even due to the fact that it is initially even and n_{left} shifts once per step. Respectively for odd steps or odd $n_{\text{initial, left}}$. This causes the histogram for n_{tot} to only use 4 of 8 available integer endings and this the number of steps. Thus the ending step is randomized and the loop run according to:

```
1 oddEvenEnding = round(rnd.random())
2 for step in range(0,nSteps-oddEvenEnding):
3     # itterate nLeft according to b)
```

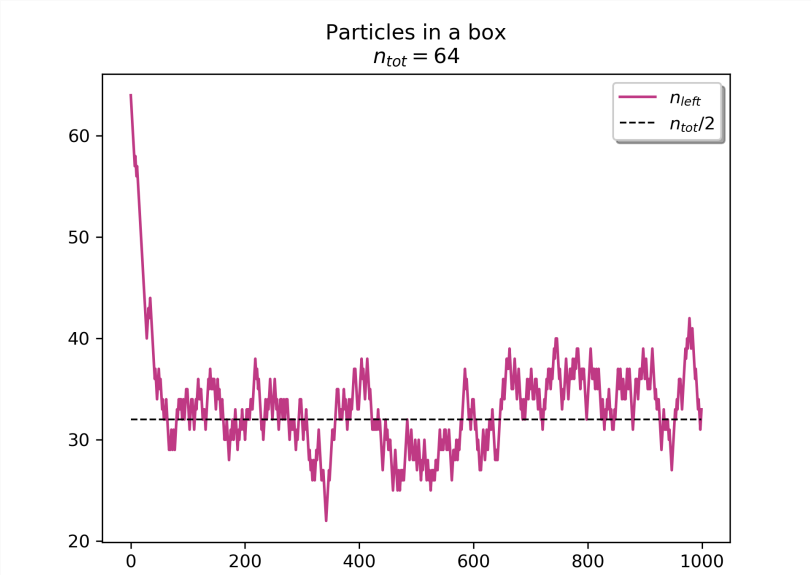
The size of the bins in the histogram are forced to be 1 by running:

```
plt.hist(endLeftArray,
        # color choice ,
        bins = [x-0.5 for x in range(min(endLeftArray), max(
            ↪ endLeftArray)+2)])
```

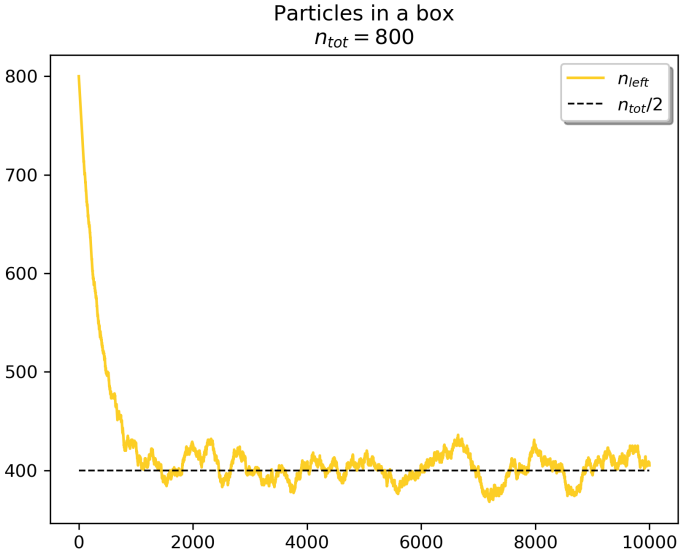
3.1 Results i


Particles in a box
$n_{tot} = 8$
8
7
6
5
4
3
2
1
0
20
40
60
80
100
n_{left}
$n_{tot}/2$

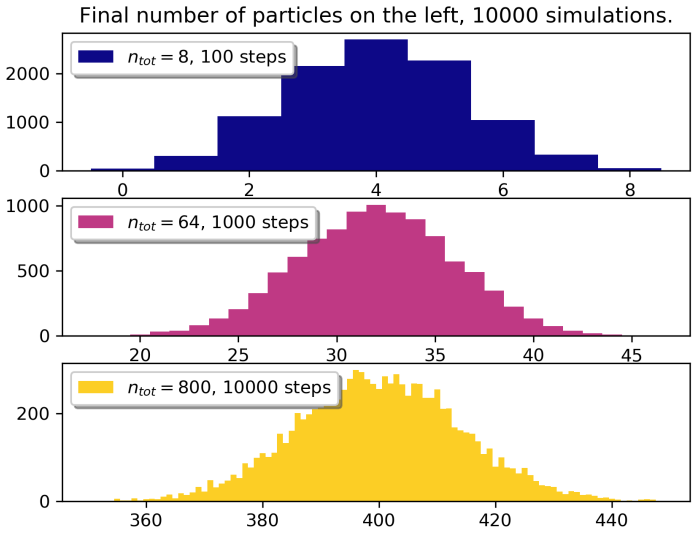
3.2 Results ii



3.1 Results iii



3.1 Results iv



3.1 Results v

- The system reaches an equilibrium according to the second definition. However, a perturbation from the equilibrium does not strictly imply that the system will move back to the equilibrium at once. The system only has a larger probability to move towards the equilibrium the further away it is from it.
- n_{left} *will change* when we are at an equilibrium since it is forced by the method to change in each step.

3.2

3.2 Project

Does the time dependence of n appear to be deterministic for sufficiently large N?
What is the qualitative behavior of $n(t)$?

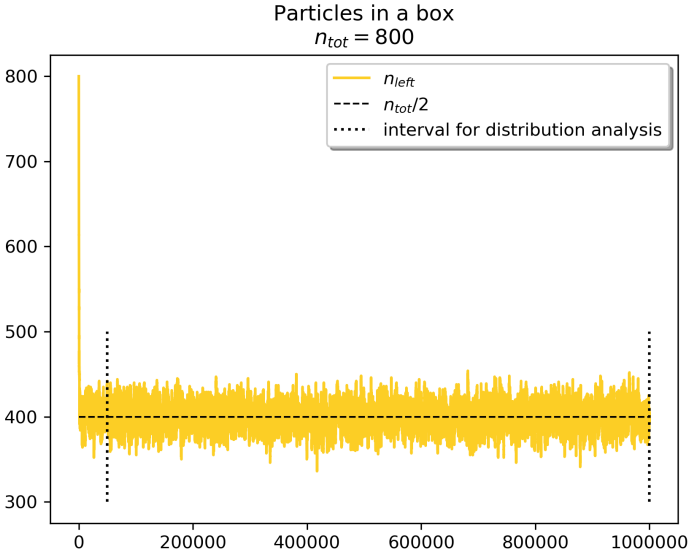
3.2 Method

The following parameters are chosen:

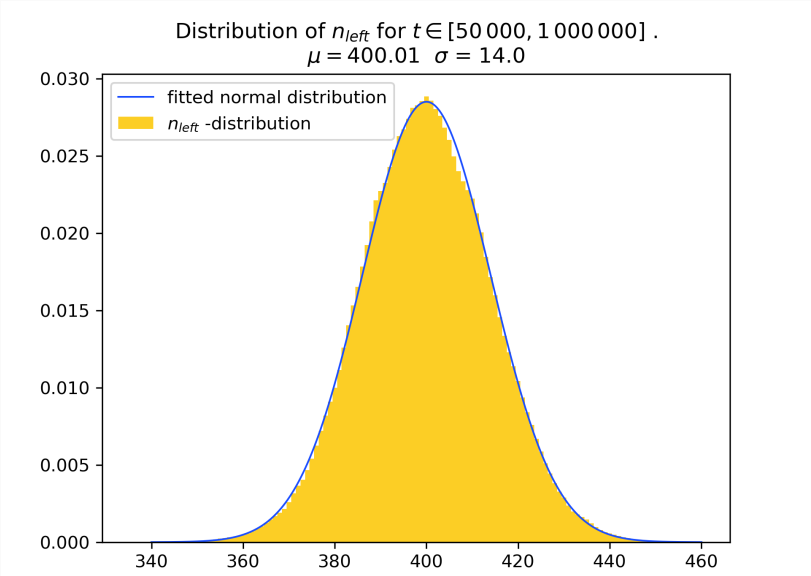
$$n_{\text{steps}} = 1\,000\,000$$
$$n_{\text{particles}} = 800\ .$$

The simulation is run and then the data for `nLeft[50000:]` is gathered. In other words, the data for n_{left} when $t \in [50\,000, 1\,000\,000]$. The data is then plotted in a histogram and fitted to a normal distribution.

3.2 Results i


Particles in a box
$n_{tot} = 800$
n_{left}
$n_{tot}/2$
interval for distribution analysis
800
700
600
500
400
300
0
200000
400000
600000
800000
1000000

3.2 Results ii



3.2 Results iii

- For sufficiently large $n_{\text{particles}}$ and n_{steps}, $n(t)$ is a normal-distributed stochastic variable after the initial transition. If $n_{\text{tot}} = 800$:

$$\mu = 400$$
$$\sigma = 14$$

3.3

3.3 Project

Now look at the effect of seeding the random generator. What does `random.seed()` exactly do in python? What is the effect of the time behavior of the first simulation? What is the effect on the histogram of the equilibrium end values of many simulations?

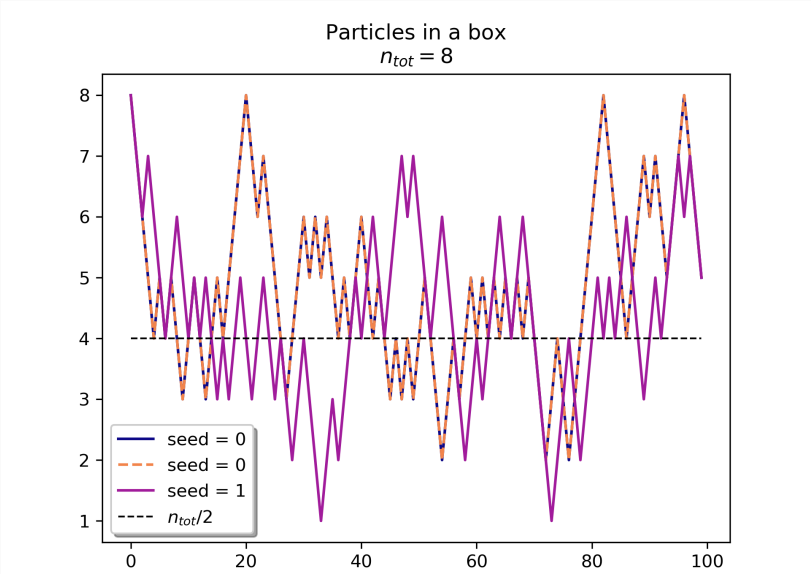
A *pseudorandom number generator* generates number sequences that are not truly random, but approximates the properties of sequence of random numbers. The numbers generated are completely determined by the initial value, called *seed*. The same seed will always produce the same sequence.[1]

In python the current system time is used as a seed if none is given.[2]This assures that the same seed will not be used twice.

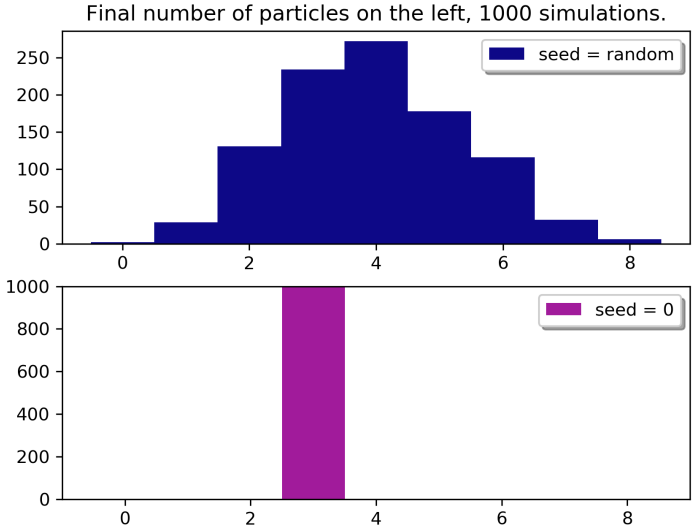
[1] `en.wikipedia.org/wiki/Pseudorandom_number_generator`
[2] `docs.python.org/3.1/library/random.html`

The dependence on seed is examined for $n_{\text{tot}} = 8$. The *particles in a box* simulation runs for the three seeds in [0,0,1] (0 runs twice) and then the particles on the left side is plotted vs time. Then the simulation runs 1 000 times first for random seeds and then for seed being initially 0 for all simulations.

3.3 Results i


Particles in a box
$n_{tot} = 8$
8
7
6
5
4
3
2
1
0
20
40
60
80
100
seed = 0
seed = 0
seed = 1
$n_{tot}/2$

3.3 Results ii



3.3 Results iii

- The time behavior of the simulation is the same if the seed is the same.
- All end values will be the same if the same seed is used each time the program runs. Thus the histogram will only have values in one bin.
- If a unique seed is used when the program initialized (for example with current date and time in ms) there's no way to differentiate the outcome from that of a truly random one.

3.4

3.4 Project

A measure of the equilibrium fluctuations is the mean square fluctuations (1).

$$\Delta n^2 = \left\langle (n - \langle n \rangle)^2 \right\rangle = \langle n^2 \rangle - \langle n \rangle^2 \tag{1}$$

Determine this when averaging over the end values of many simulations as well as for averaging over time for a long, equilibrium part of a single simulation. How do these two different fluctuations compare? How do they depend on N?

3.4 Method i

Firstly a function is defined that returns the mean square fluctuations.

```
1  def getDeltaN(nArray):
2      """This function takes nArray (list with floats) and then
           ↪ returns the mean square fluctuations as a float."""
3      def mean(array):
4          """This function returns the mean of an array of floats.
               ↪ """
5          sum = 0
6          for x in array: sum += x
7          return sum/len(array)
8      nSquaredArray = [x**2 for x in nArray]
9      return mean(nSquaredArray) - mean(nArray)**2
```

Then the mean square is determined when

N	8	64	800
n_{steps}	1 000	10 000	100 000

both for values around equilibrium when running one simulation for a long time and for the end values of 10 000 simulations. This result can be seen in Table 1. All values are determined after initial transition by starting the simulation with `nLeft = nTot/2`.

N	End values	Around equilibrium
8	2.045	1.917
64	15.932	16.813
800	197.35	161.28

Table 1: Table over Δn^2 for end values and around the equilibrium when running simulation for a long time. All values are determined after initial transition.

3.4 Results ii

- Approximately, $\Delta n^2 \propto N$.
- The two fluctuations are very similar. However, the mean square fluctuations around the equilibrium seem increase less then linearly as $N \to 800$. This could be random for the last measurement or it could depend on the fact that only one particle can be moved during each time step.

3.5

3.5 Project

Compute the average $\langle n \rangle$ from the end values of many simulations as well as for averaging over time for a long, equilibrium part of a single simulation. Which average might be more accurate?

3.5 Method i

The mean is found with the function:

```
1 def mean(array):
2     """This function returns the mean of an array of floats."""
3     sum = 0
4     for x in array: sum += x
5     return sum/len(array)
```

The mean determined when

N	8	64	800
n_{steps}	1 000	10 000	100 000

both for values around equilibrium when running one simulation for a long time and for the end values of 10 000 simulations. This result can be seen in Table 2. All values are determined after initial transition by starting the simulation with `nLeft = nTot/2`.

N	End values	Around equilibrium
8	4.012	4.059
64	32.055	32.711
800	399.94	401.57

Table 2: Table over $\langle n \rangle$ (`mean(nArray)`) for end values and around the equilibrium when running simulation for a long time. All values are determined after initial transition.

3.5 Results ii

- $\langle n \rangle \approx N/2$.
- The average from end values might be more accurate than averaging over a long time. This is since each end value is more independent from the previous end value than two following values that are always ± 1 apart. Furthermore, the pseudonumber generator uses the last random number as a seed for the next and thus two following n is more dependent than two following n_{end}. However, both should show $\langle n \rangle = N/2$. The most accurate should be the the average from the longest list of values.

3.6

3.6 Project

Write a program that generates a two-dimensional random walk with single steps along x or y. The simplest way to do this is by generating a number randomly taken out of the sequence $(0, 1, 2, 3)$ by multiplying `random.rnd()` by 4 and rounding the result down to an integer using `int()`. Then you can increase x by one for 0, decrease x by one for 1, and the same for y with 2 and 3. Plot random walks for 10, 100 and 1000 steps.

3.6 Method i

A function for randomized steps is defined.

```
1  def randomStep(x,y):
2      """This function takes x and y arrays and returns new x and y
           ↪ arrays with a new, random step taken at the end of the
           ↪ arrays."""
3      directions = {
4          0:  [1,  0],
5          1:  [0,  1],
6          2:  [-1,  0],
7          3:  [0,  -1]
8      }
9      direction = directions[int(4*rnd.random())]
```

3.6 Method ii

```
10      x.append(x[-1]+direction[0])
11      y.append(y[-1]+direction[1])
12      return x,y
```

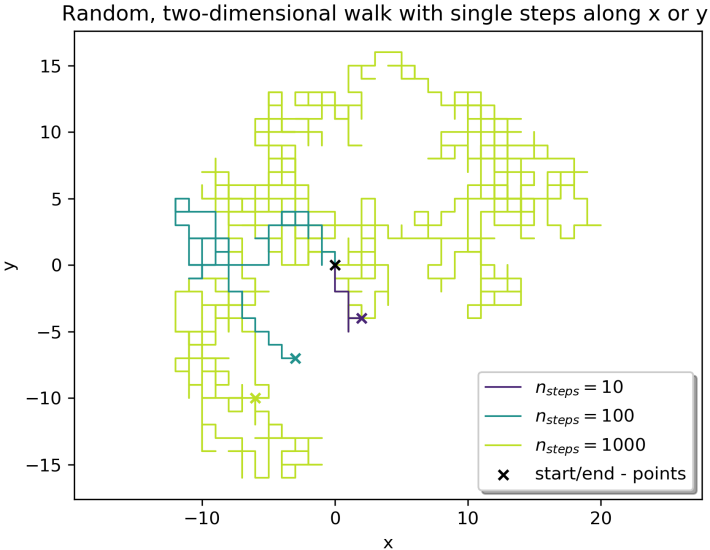
3.6 Method iii

A loop is iterated for each `step` in `stepsArray = [10, 100, 1000]`.

```
for i in range(len(stepsArray)):
    x = [0]
    y = [0]
    steps = stepsArray[i]
    for step in range(steps):
        x,y = randomStep(x,y)
    xDict[i] = x
    yDict[i] = y
```

The resulting walks are plotted in a common figure.

3.6 Results


Random, two-dimensional walk with single steps along x or y.
15
10
5
0
−5
−10
−15
y
−10
0
10
20
x
$n_{steps} = 10$
$n_{steps} = 100$
$n_{steps} = 1000$
start/end - points

3.7

3.7 Project

Instead of `rnd.random()` use the simple random generator

$$r_n = (a r_{n-1} + c) \ \% \ \mathrm{m} \tag{2}$$

which generates random number in the range $0 \to m - 1$. How does the walk look like for

$$r_0 = 1, \quad a = 3, \quad c = 4, \quad m = 128 \ ? \tag{3}$$

Also try

$$m = 129, \quad m = 130 \tag{4}$$

and some other values for all four parameters.

From section 3.6 walking 100 seems most informative.

The dependence of the walk on r_0 is determined by calculating walks of 100 steps when

$$r_0 = 1, \quad a = 3, \quad c = 4, \quad m = 128$$
$$r_0 = 3, \quad a = 3, \quad c = 4, \quad m = 128$$
$$r_0 = 7, \quad a = 3, \quad c = 4, \quad m = 128 \ .$$

3.7 Method ii

The dependence of the walk on a is determined by calculating walks of 100 steps when

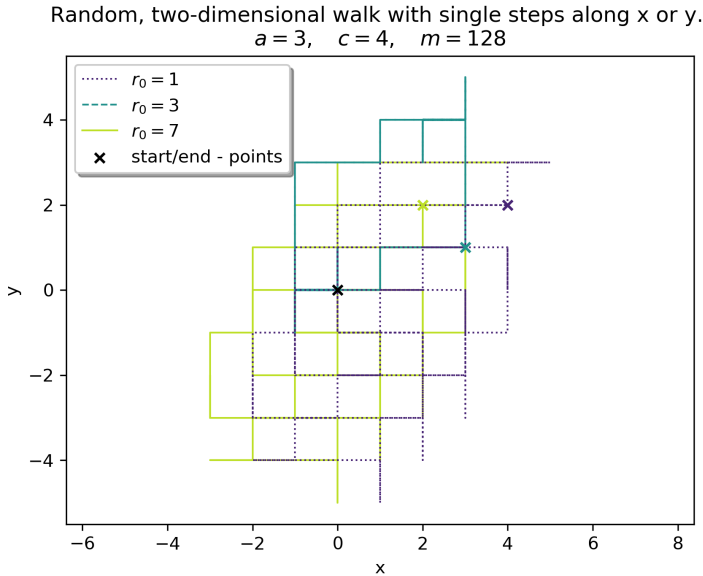
$$r_0 = 1, \quad a = 2, \quad c = 4, \quad m = 128$$
$$r_0 = 1, \quad a = 3, \quad c = 4, \quad m = 128$$
$$r_0 = 1, \quad a = 5, \quad c = 4, \quad m = 128\ .$$

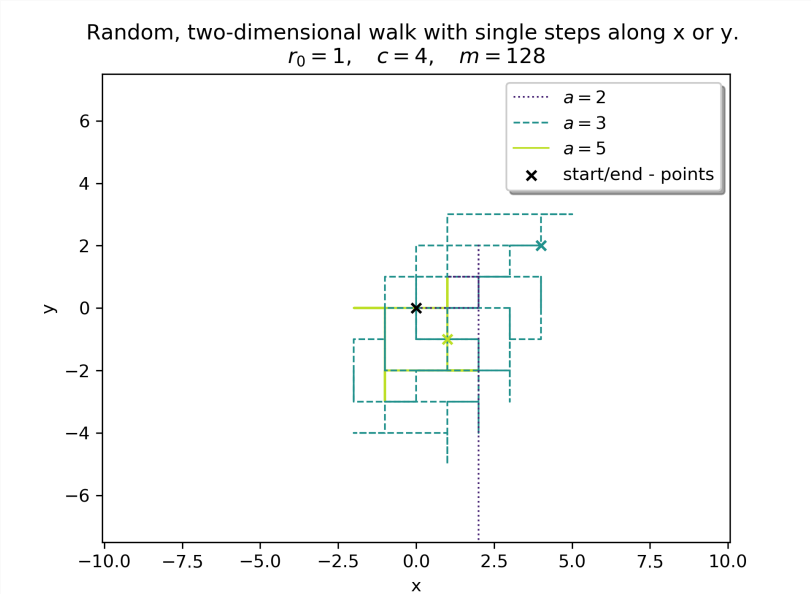
The dependence of the walk on c is determined by calculating walks of 100 steps when

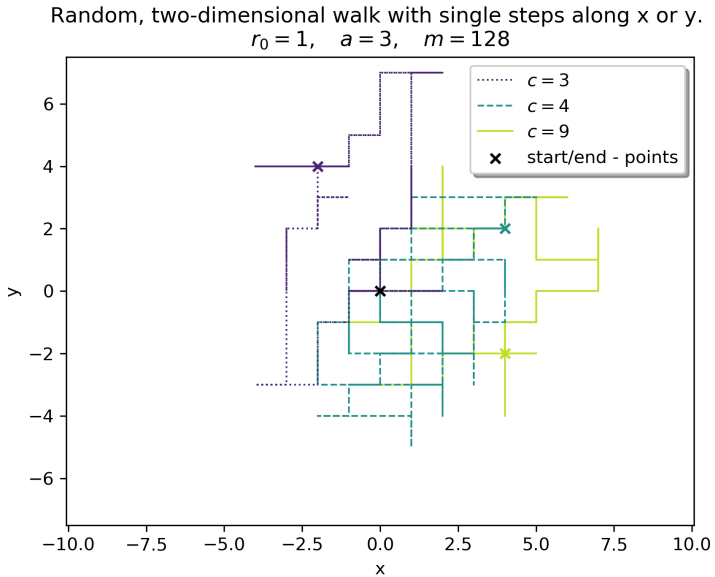
$$r_0 = 1, \quad a = 3, \quad c = 3, \quad m = 128$$
$$r_0 = 1, \quad a = 3, \quad c = 4, \quad m = 128$$
$$r_0 = 1, \quad a = 3, \quad c = 9, \quad m = 128\ .$$

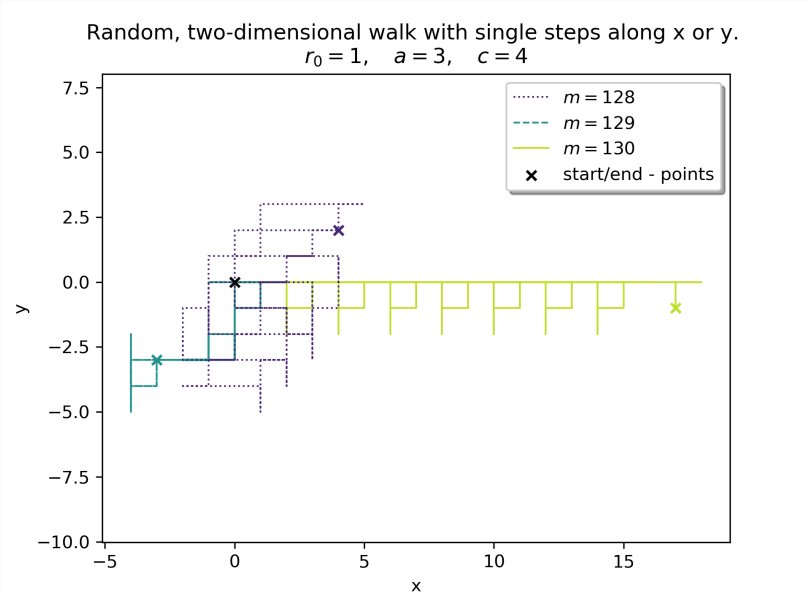
The dependence of the walk on m is determined by calculating walks of 100 steps when

$$r_0 = 1, \quad a = 3, \quad c = 3, \quad m = 128$$
$$r_0 = 1, \quad a = 3, \quad c = 3, \quad m = 129$$
$$r_0 = 1, \quad a = 3, \quad c = 3, \quad m = 130\ .$$


Random, two-dimensional walk with single steps along x or y.
$a = 3, \quad c = 4, \quad m = 128$
$r_0 = 1$
$r_0 = 3$
$r_0 = 7$
start/end - points
y
x
4
2
0
−2
−4
−6
−4
−2
0
2
4
6
8


Random, two-dimensional walk with single steps along x or y.
$r_0 = 1, \quad c = 4, \quad m = 128$
$a = 2$
$a = 3$
$a = 5$
start/end - points
y
x
6
4
2
0
−2
−4
−6
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


Random, two-dimensional walk with single steps along x or y.
$r_0 = 1, \quad a = 3, \quad m = 128$
$c = 3$
$c = 4$
$c = 9$
start/end - points
y
x
6
4
2
0
−2
−4
−6
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


Random, two-dimensional walk with single steps along x or y.
$r_0 = 1, \quad a = 3, \quad c = 4$
$m = 128$
$m = 129$
$m = 130$
start/end - points
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
y
−5
0
5
10
15
x

3.7 Results i

- A bad pseudonumber generator generates sequences containing patterns.
- When $a = 2$ the seed, r, got stuck in a loop and the walk continued indefinitely in negative y-direction.

3.8

3.8 Project

Generate many random walks with lengths in the range from 1 to 1000 and determine the mean squared end-to-end distance $\langle R^2 \rangle$ for each length. How does it depend on N?

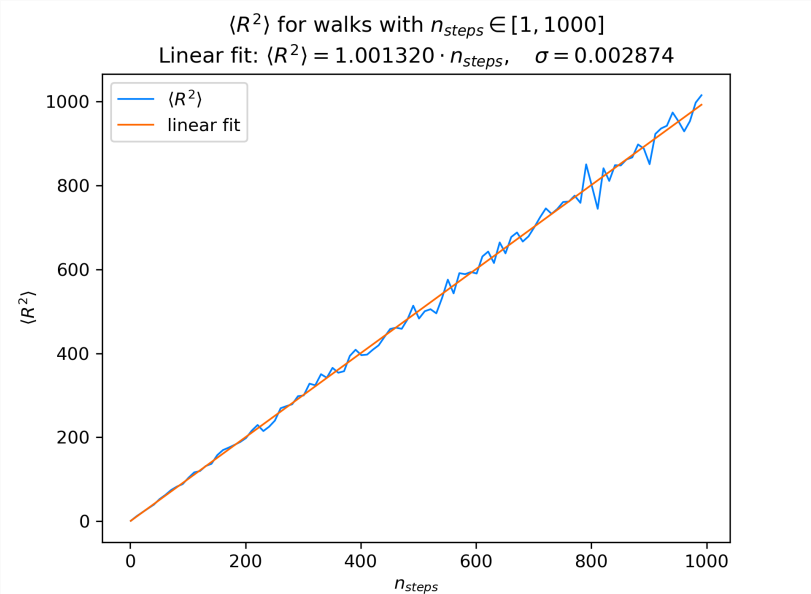
3.8 Method

- $N = n_{\text{steps}} \in \{1, 11, \ldots, 991, 1000\}$.
- $\langle R^2 \rangle$ is determined for 1 000 simulations for each n_{steps}.
- A linear function of the type $\langle R^2 \rangle = f_{\text{linear}}(n_{\text{steps}}) = k \cdot n_{\text{steps}}$ is derived using `curve_fit()` from `scipy.optimize`.[3]

```
1 def fLinear(x,k):
2     return k*x
3
4 popt, pcov = curve_fit(fLinear, stepsArray, meanRSquaredArray)
5 k = popt[0]
6 std = np.sqrt(pcov[0][0])
```

[3] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html

3.8 Results i


$\langle R^2 \rangle$ for walks with $n_{steps} \in [1, 1000]$
Linear fit: $\langle R^2 \rangle = 1.001320 \cdot n_{steps}, \quad \sigma = 0.002874$
$\langle R^2 \rangle$
linear fit
$\langle R^2 \rangle$
n_{steps}
0
200
400
600
800
1000

3.8 Results ii

Dependence on n_{steps}: $\langle R^2 \rangle = 1 \cdot n_{\text{steps}}$

Fluctuations: Increasing the number of simulations for each n_{steps} reduces the fluctuations in the graph.

3.9

3.9 Project

A real polymer can not cross itself, i.e. different atoms can not occupy the same space. Generate self-avoiding random walks by storing all previously visited sites of the same walk and terminate and discard the walk when a previously visited is revisited. How does the fraction of successful walks depend on $N = n_{\text{steps}}$? What is the maximum value of N that you can reasonably consider? You can improve the algorithm somewhat by only generating moves in three directions, not back in the direction where you just came from. How does that improve the success?

3.9 Method i

I. Three self avoiding random walks are generated and plotted for $n_{\text{steps}} \in \{10, 20, 30\}$.

II. The random-step algorithm is improved.

```
1 def improvedRandomStep(x,y):
2     """This function takes integer x and y arrays of same
          ↪ length and returns new x and y arrays with a new,
          ↪ random step taken at the end of the arrays. It only
          ↪ takes steps left, forwards or right in relation to
          ↪ the current direction, thus it never walks back into
          ↪  itself. If the current walk length is zero a random
          ↪  step is taken in any of the four directions."""
```

3.9 Method ii

```
3     if len(x) == 1: return randomStep(x,y)
4     a = x[-1]-x[-2]
5     b = y[-1]-y[-2]
6     currentDirection     = [a,b]
7     leftDirection        = [-b,a]
8     rightDirection       = [b,-a]
9     directions = {
10        0:  leftDirection,
11        1:  currentDirection,
12        2:  rightDirection,
13    }
14    direction = directions[int(3*rnd.random())]
15    x.append(x[-1]+direction[0])
```

```
16      y.append(y[-1]+direction[1])
17      return x,y
```

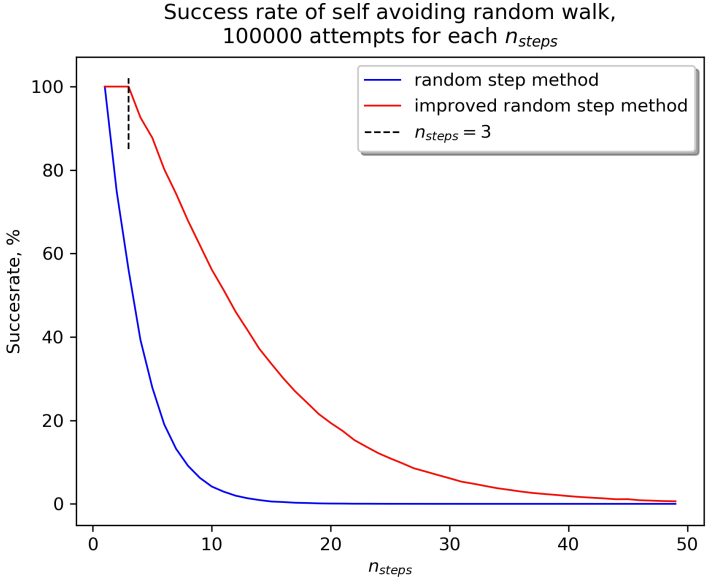
With $n_{\text{attemps}} = 100\,000$ as the number of attempts for each $n_{\text{steps}} \in \{1, 2, \ldots, 50\}$ the successful fraction is determined and plotted in a graph. This is done both for the random step method and the improved random step method.

III. To find the dependence on n_{steps} the success rate is logarithmized on the y-axis. The slope is found and a relationship is derived.

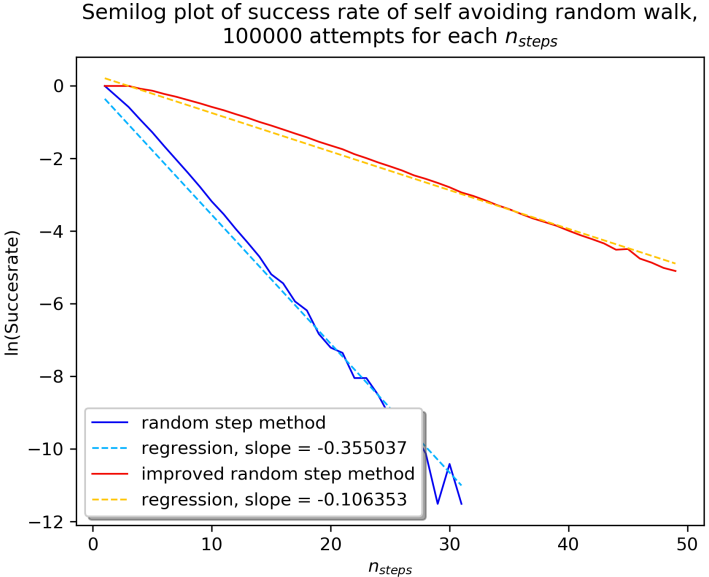
3.9 Results i


Random, two-dimensional, self-avoiding walk
with single steps along x or y.
$n_{steps} = 10$
$n_{steps} = 20$
$n_{steps} = 30$
start/end - points
y
x
10
8
6
4
2
0
−2
−4
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0

3.9 Results ii



3.9 Results iii



3.9 Results iv

Max n_{steps}: For the first random step method, as $n_{\text{steps}} \to 30$ there are next to no successes in 100 000 attempts. The maximum value of n_{steps} one can consider depends on patience and computer power, however for $n_{\text{steps}} > 30$ one should consider a better method. With the improvised random step method n_{step} up toward 50 or more can be used.

Improvement: The improved random step algorithm substantially increased the success rate. The improved algorithm had 100 % success rate for $n_{\text{steps}} \leq 3$ since it has no possibility to walk on an already occupied point before completing a square graph.t

Dependence on n_{steps}: The success rate drastically decreased for increasing n_{steps}. An analysis of the slopes gives relationship between success rate (r) and n_{steps}.

$$\ln r = k \cdot n_{\text{step}} + \mathit{constant}$$

$$\Longleftrightarrow$$

$$r = r_0 \cdot e^{k \cdot n_{\text{step}}}$$

From this, the calculated relationship can be found in Table 3.

method	dependence on n_{steps}
random step	$r = \exp(-0.35 \cdot n_{\text{steps}})$
improved random step	$r = \begin{cases} 1, & n_{\text{steps}} \leq 3 \\ \exp\big(-0.11 \cdot (n_{\text{steps}} - 3)\big), & \text{else} \end{cases}$

Table 3: Relationship between r (success rate) and n_{steps}.

3.10

3.10 Project

Compute the mean of the square of the end-to-end distances for the self-avoiding random walk for the range of N that you can cover. What is the difference with the normal random walk? (suggestion: use log-log scale)

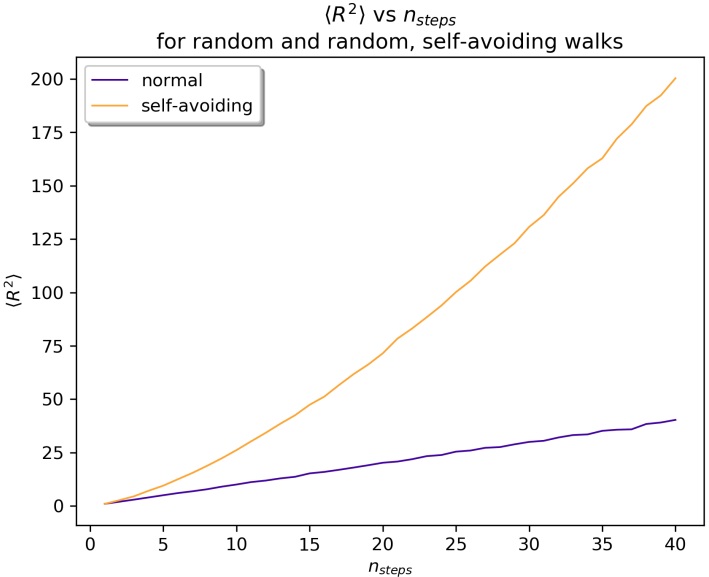
3.10 Method

- $N = n_{\text{steps}} \in \{1, 2, \ldots, 40\}$.
- $\langle R^2 \rangle$ is determined for 10 000 simulations for each n_{steps}.
- A linear function of the type $\langle R^2 \rangle = f_{\text{linear}}(n_{\text{steps}}) = k \cdot n_{\text{steps}} + m$ is derived using `curve_fit()` from `scipy.optimize`.[4]

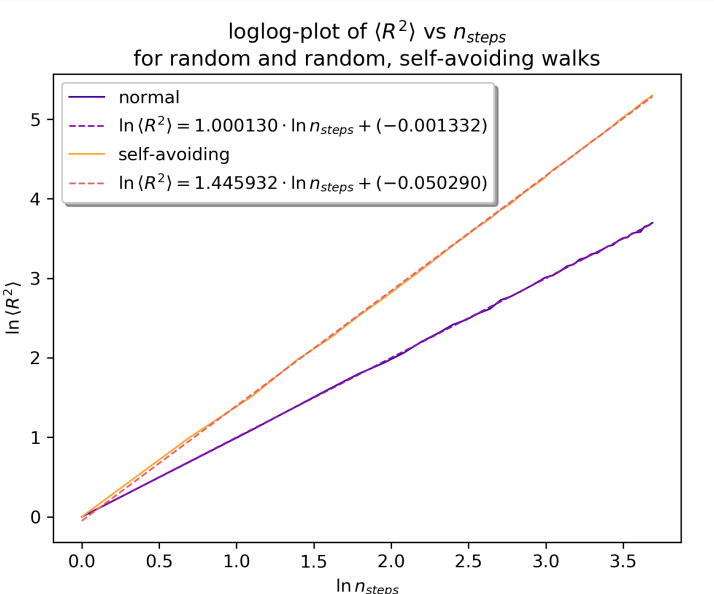
```
1 def fLinear(x,k):
2     return k*x
3
4 popt, pcov = curve_fit(fLinear, stepsArray_logged,
    ↪ meanRSquaredArray_logged)
5 k = popt[0]
6 m = popt[1]
```

[4] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html

3.10 Results i



3.10 Results ii



3.10 Results iii

An analysis of the slopes gives relationship between $\langle R^2 \rangle$ and n_{steps}.

$$\ln\langle R^2 \rangle = k \cdot \ln n_{\text{step}} + m$$

$$\Longleftrightarrow$$

$$\langle R^2 \rangle = e^m \cdot (n_{\text{step}})^k$$

From this, the calculated relationship can be found in Table 4.

walk type	dependence on n_{steps}
random step	$\langle R^2 \rangle = 1 \cdot (n_{\text{steps}})^1 = n_{\text{steps}}$
self-avoiding random step	$\langle R^2 \rangle = 0.95 \cdot (n_{\text{steps}})^{1.45} \approx \sqrt{(n_{\text{steps}})^3}$

Table 4: Relationship between $\langle R^2 \rangle$ and n_{steps}.
$n_{\text{steps}} \in \{1, 2, \ldots, 40\}$, 10 000 simulated walks for each walk type.

Appendix

Source code

Available at:

https://github.com/axelstr/SI1336-Simulations-and-Modeling/tree/master/3%20Random%20processes