Home Assignment 2

Theodora M. Gaiceanu
Felix E. Slothower

February 22, 2022

Task 1

Let c_n be the number of n-step self-avoiding walks on a lattice. In [3], from the elementary bounds on c_n ($\sqrt{2}^n \leq c_n \leq 3 \cdot 2^{n-1}$), it is known that the growth of c_n is exponentially. What is more, let's say we have a $(n+m)$ step self-avoiding walk. This walk can be cut in a unique manner into a n step self-avoiding walk and a parallel translation of a m step self-avoiding walk [3]. Let S_n be the set of all possible n steps self avoiding walks, S_m the set of all possible m steps self-avoiding walks and S_{n+m} the set of all $n+m$ steps self avoiding walks. As some of the combinations of S_n and S_m will give non self avoiding walks, then it means that we will not have all the possible combinations of S_n and S_m in S_{n+m}. Therefore, we have that $|S_{n+m}(d) \leq |S_n(d)| \times |S_m(d)|$, which implies that $c_{n+m}(d) \leq c_n(d)c_m(d)$.

Task 2

First, we can take the logarithm of the limit. This will give us the following:

$$\log(\mu_d) = \lim_{n\to\infty} \frac{\log(c_n(d))}{n}$$

But in order to this limit to exist, one should prove that $\{\log(c_n(d))_{n=1}\}^{\infty}$ is a sub additive sequence. From Task 1, we know that $c_{n+m}(d) \leq c_n(d)c_m(d)$. We can take the logarithm of this one:

$$\log(c_{n+m}(d)) \leq \log(c_n(d)c_m(d))$$

$$\log(c_{n+m}(d)) \leq \log(c_n(d)) + \log(c_m(d))$$

This last equation shows that $\{\log(c_n(d))_{n=1}\}^{\infty}$ is a sub additive sequence. Therefore, from the Fekete's lemma we have that the limit $\log(\mu_d) = \lim_{n\to\infty} \frac{\log(c_n(d))}{n}$ exists and is equal to:

$$\lim_{n\to\infty} \frac{\log(c_n(d))}{n} = inf_{n\geq 1} \frac{c_n(d)}{n}$$

Consequently, the limit $\mu_d = \lim_{n\to\infty} (c_n(d))^{\frac{1}{n}}$ exists.

Task 3

In SIS (Sequential importance sampling), we have the particles $X_i^{0:n+1}$ and the weights w_{n+1}^i, where n is the number of steps. We always keep the previous particle $X_i^{0:n}$ and we simulate the next particle X_i^{n+1} by drawing it from a instrumental distribution g_n. Then we set the particle $X_i^{0:n+1} = (X_i^{0:n}, X_i^{n+1})$ and we compute the weight w_{n+1}^i. So the weights are sequentially updated [4].

We consider only 2 dimensions for this problem. Therefore our moves can be codes from 1 to 4, with the following meaning: 3 - move right, 2 - move up, 1 - move left, 4 -move down. We store our walks in a cell array *roads* that has 2 columns: the first column represents a kind of tuple with the sequence of moves of a walk (it has an array for the X coordinate and an array for the Y coordinate), the second column is just a number telling if the walk is self avoiding or not (if it is 1 it means it is self avoiding, if it is 0, it means it is not self avoiding).

We can make the experiment for different numbers of steps *no_steps*. For each step, we consider N iterations. For each iteration we do the following:

- we define a uniform distribution which has numbers inside the interval $[1, 4]$. These will be our possible directions to move. Each direction has the same probability of being chosen. Therefore, our instrumental distribution g_n is the uniform distribution.
- as $X_0 = 0$, we start from the centre (coordinates (0,0))
- we compute the neighbors by updating the tuple with the walks considering the movement encoding described before. We draw the next move from the uniform distribution with the numbers inside the interval $[1, 4]$. So the particle $(X_k)_{k=0}^{n}$ is a random walk which is stored in the first column of the *roads* cell array and each (X_{k+1}) is drawn uniformly at this step among the four neighbours of X_k, without checking if they are free or not.
- we check if the walk is self avoiding or not and we mark this on the second column of the *roads* cell array. A self avoiding walk is a walk that does not have any steps that go back to the previous points. So, in order to be self avoiding, a walk should not have the same pair of coordinates more than one time. This can be easily verified in MATLAB using the *unique* function on the tuple with the movements.

Then we count the number of walks that are self-avoiding. We compute the estimate as the ratio $\frac{N_{SA}}{N}$ multiplied with $4^{currentStepSize}$, where N_{SA} is the number of self avoiding walks and N is the number of iterations, and $4^{currentStepSize}$ have the significance of the weights.

As we do not check for the available neighbours before drawing the next move, this is a simulation of a N number of walks in the 2d space, counting the N_{SA} number of self-avoiding ones and estimating c_n using the ratio $\frac{N_{SA}}{N}$.

n	c_n
1	4
2	12
3	38
4	102
5	272
6	705
7	2097
8	5046
9	17826
10	53477

Table 1: Estimate of c_n when using $n = 10$ steps and $N = 1000$ iterations - task 3

The correct estimate for the number of self-avoiding walks c_n is known from [6]. As it can be seen from tables 1 and 2, when using bigger N, the values are more accurate. Even so, when n is bigger than 3, the estimate starts to have modified values from the correct ones and the difference starts to get bigger when n is higher than 5. This is partly because, in general, for bigger n it is harder to get a self-avoiding random walk. So this naive approach is not so accurate.

Summarising, this naive approach can be seen as using Monte Carlo to simulate the Markov chain $(X_k)_{k\geq 0}$ N times, yielding the moving directions $(X_i^{0:n})_{i=1}^{N}$, count the number N_{SA} of self-avoiding

n	c_n
1	4
2	12
3	35
4	113
5	348
6	983
7	1802
8	7209
9	20972
10	73400

Table 2: Estimate of c_n when using $n = 10$ steps and $N = 100$ iterations - task 3

walks and estimate c_n using Monte Carlo estimator. This is actually the approach presented in [5].

Task 4

We consider again only 2 dimensions. We store our walks in the cell array *roads*, but this time it has 3 columns: the first column has the same meaning, it represents a kind of tuple with the sequence of moves of a walk (it has an array for the X coordinate and an array for the Y coordinate), the second column is again just a number telling if the walk is self avoiding or not (if it is 1 it means it is self avoiding, if it is 0, it means it is not self avoiding), and the third column represents the weights of the walk.

We make the experiment again for different numbers of steps *no_steps*. For each step, we consider N iterations. For each iteration we do the following:

- we define the lattice and we initialize the weight with 1
- as $X_0 = 0$, we start from the centre (coordinates (0,0)); we also save the coordinates separately, for the lattice. So we initilze also the coordinates and we update the lattice, as it should have a 1 in the starting point.
- we define all the possible directions (up, down, left, right) for the current position using the lattice.
- we compute the available free neighbours that can be chosen for the current position
- we check if we are in a dead end (if we do not have any available free neighbours). If we are in a dead end, we mark the walk as not being self-avoiding (we put a 0 in the second column of *roads* for the current walk and we store the weight.
- we update the weights by multiplying them with the number of unoccupied neighbors of the current position.
- this time we define our direction to move by drawing from the unoccupied neighbors of the current position and we update the tuple with the walks (the first column of the cell array *roads*).
- we update the coordinates and the lattice
- we check if the walk is self avoiding or not and we mark this on the second column of the *roads* cell array. This time we check the length of the walk, as the non self-avoiding walks are shorter (because they are cut whenever we encounter a dead-end). We also store the weights for the current walk in the third column of the cell array *roads*.

Then we count the number of walks that are self-avoiding. We compute the estimate as the mean of the weights. This is in correspondence with [4], where it is stated that for each n, an unbiased estimate c_n can be $\frac{1}{N}\sum_{i=1}^{N} w_n^i$.

n	c_n
1	4
2	12
3	36
4	100
5	284
6	775
7	2178
8	5893
9	16314
10	44114

Table 3: Estimate of c_n when using $n = 10$ steps and $N = 1000$ iterations - task 4

As it can be observed from Table 3, the accuracy of the estimate improved significantly when drawing directions to move from the set of free neighbors. The results are from an experiments with 1000 iterations. There are still some differences from the correct estimate in [6] for n bigger than 6, but they are substantially reduced compared to the results from the previous task.

Task 5

SISR (Sequential importance sampling with resampling) is based on the idea that the particles with small weights should be elminiated and the particles with large weights should be duplicated [4]. According to [4], this is like a selection and a way to do this is to draw with replacement new particles $\tilde{X}_1^{0:n}, \tilde{X}_2^{0:n}, ..., \tilde{X}_N^{0:n}$ among the particles produced by SIS $X_1^{0:n}, X_2^{0:n}, ..., X_N^{0:n}$ with probabilities given by the normalized importance weights.

We consider again only 2 dimensions. We store our walks in the cell array *roads*; it has 3 columns again, but with the following meaning: the first column has the same meaning, it represents a kind of tuple with the sequence of moves of a walk (it has an array for the X coordinate and an array for the Y coordinate), the second column represents the weights of the walk, and the third column stores the lattice for the walk.

First, we make an initialization or we simulate the first component. We consider N iterations. For each iteration, we consider our starting point for the walk the centre (coordinates (0,0)) and we initialize the tuple (the first column of the cell array *roads)* with this point. Then we initialize the weights (the second column of the cell array) and we also define the lattice (the third column of the cell array). We initialize the coordinates (which are stored also separately) and we set the initial weights to 1.

Then we make the experiment for different numbers of steps *no_steps*. For each step, we consider N iterations. For each iteration we do the following:

- as $X_0 = 0$, we start from the centre (coordinates (0,0)) and we save the coordinates separately, for the lattice.
- we define all the possible directions (up, down, left, right) for the current position using the lattice (which is now the third column of the cell array *roads*) .
- we compute the available free neighbours that can be chosen for the current position
- we update the weights as the sum of unoccupied neighbors of the current position.

- we check if we are in a dead end (if we do not have any available free neighbours). If we are in a dead end, we update the second column of the cell array *roads* with the current weight and we simply continue with the next step.
- we define our direction to move by drawing from the unoccupied neighbors of the current position and we update the tuple with the walks (the first column of the cell array *roads*).
- we update the coordinates and the lattice (now the third column of the cell array *roads*)

Then we estimate the number of walks that are self-avoiding. We compute the estimate as the product of the mean of the weights (stored in the second column of the cell array *roads*). This is in correspondence with [4], where it is stated that for estimation constants using SISR, c_n can be computed as $\prod_{k=0}^{n}(\frac{1}{N}\sum_{i=1}^{N} w_n^i)$. After this, we do the resampling with replacement. We use the MATLAB function *randsample* for this using the normalized weights taken with replacement. Then we update the movements of the walk (the first column of *roads*) and the lattice (the third column of the cell array).

n	c_n
1	4
2	12
3	36
4	100
5	282
6	776
7	2127
8	5793
9	16047
10	43021

Table 4: Estimate of c_n when using $n = 10$ steps and $N = 1000$ iterations - task 5

As it can be seen from Table 4, the accuracy of the estimate when using SISR with 1000 iterations is comparable to the accuracy of the estimate from task 4 for n smaller than 8. For n bigger and equal to 8, the estimate has some differences from the correct answer in [6], and the estimate is slightly less accurate than the estimate from task 4. But anyway, SISR seems to be faster than SIS from task 4.

Task 6

In order to solve the linear regression problem one has to do the following steps:

$$c_n(2) = A_2\mu_2^n n^{\gamma_2-1} \tag{1}$$

One should take the logarithm of Equation (1).

$$\log(c_n(2)) = \log(A_2\mu_2^n n^{\gamma_2-1})$$

$$\log(c_n(2)) = \log(A_2) + n\log(\mu_2) + (\gamma_2 - 1)\log(n)$$

$$\log(c_n(2)) = n\log(\mu_2) + (\gamma_2 - 1)\log(n) + \log(A_2) \tag{2}$$

From Equation (2), it can be seen that we have a linear regression problem with the form $y = ax_1 + bx_2 + c$, where $x_1 = n$, $x_2 = \log n$, and $a = \log(\mu_2)$, $b = \gamma_2 - 1$, $c = \log(A_2)$.

This linear regression problem can be easily solved in MATLAB. The orginal parameters can be estimated as:

- μ_2 is the exponential of the first element of the linear regression result
- γ_2 is the second element of the linear regression result added with 1
- A_2 is the exponential of the third element of the linear regression result

μ_2	γ_2	A_2
2.6454	1.2334	1.4944

Table 5: Estimate of μ_2, γ_2 and A_2 parameters for two dimensions when using $n = 10$ and $N = 1000$ iterations

μ_2	γ_2	A_2
2.6585	1.2171	1.4895

Table 6: Estimate of μ_2, γ_2 and A_2 parameters for two dimensions when using $n = 20$ and $N = 1000$ iterations

μ_2	γ_2	A_2
2.6628	1.2099	1.4925

Table 7: Estimate of μ_2, γ_2 and A_2 parameters for two dimensions when using $n = 30$ and $N = 1000$ iterations

We know that the correct value of $\gamma_2 = \frac{43}{32} = 1.34$. From Tables 5, 6, 7, it can be seen that the closest γ_2 to the correct value is obtained when using $n = 20$ and $N = 1000$ iterations. When increasing the number of steps n, it seems that γ_2 is slightly decreasing, μ_2 is slightly increasing and A_2 does not seem to change.

From Tables 5, 6, 7 and the paragraph above, it can be observed that μ_2 and A_2 have the lowest variance. Therefore, μ_2 and A_2 are the most easy parameters to estimate.

Task 7

We know that:

$$c_{n+m} \leq c_n c_m \tag{3}$$

and we know that the limit

$$\mu = \lim_{n\to\infty} c_n^{\frac{1}{n}} \tag{4}$$

exists.

From [1] it is known that $\mu^n \leq c_n$ for all n. For a dimension d, we have d^n n step walks taking steps only in the positive coordinate directions (for example, for $d = 2$, consider walks taking steps only in the up and right directions). As they are taking the steps only in the positive coordinate directions, these walks are self-avoiding. Therefore we have that $d \leq \mu$. Moreover, the set of n steps walks without immediate reversals has cardinality $(2d)(2d-1)^{n-1}$ and has all the n steps walks [1]. Consequently, it holds that $\mu \leq 2d - 1$. By putting all things together, we have that $d \leq \mu_d \leq 2d - 1$.

Task 8

For $5 \leq d$ we have that:

$$c_n(d) = A_d \mu_d^n n^{\gamma_d - 1} \tag{5}$$

And we also have that:

$$c_{n+m}(d) \leq c_n(d) c_m(d) \tag{6}$$

So we plugin Equation (6) in (5) and we have that:

$$A_d \mu_d^{n+m} n + m^{\gamma_d - 1} \leq A_d \mu_d^n n^{\gamma_d - 1} A_d \mu_d^m m^{\gamma_d - 1}$$

$$A_d \mu_d^{n+m} n + m^{\gamma_d - 1} \leq A_d^2 \mu_d^{n+m} n + m^{\gamma_d - 1}$$

$$A_d \leq A_d^2$$

$$0 \leq A_d^2 - A_d$$

$$0 \leq A_d(A_d - 1)$$

We know that $0 \leq A_d$ is true. Therefore we have that $0 \leq A_d - 1$. Which mean that $1 \leq A_d$ for $5 \leq d$.

Task 9

We generalize the problem for any number of dimensions d. We store our walks in the cell array *roads*; it has 2 columns, with the following meaning: the first column has the same meaning, it represents a kind of tuple with the sequence of moves of a walk (it has an array for the X coordinate and an array for the Y coordinate), the second column represents the weights of the walk.

First, we make an initialization or we simulate the first component. We consider N iterations. For each iteration, we consider our starting point for the walk the centre (coordinates (0,0), or (0,0,0) and so on, depending on the dimension) and we initialize the tuple (the first column of the cell array *roads)* with this point. Then we initialize the weights (the second column of the cell array).

Then we make the experiment for different numbers of steps *no_steps*. For each step, we consider N iterations. For each iteration we do the following:

- we define all the possible directions for the current position depending on the dimension. *roads*) .
- we compute the available free neighbours that can be chosen for the current position
- we update the weights as the sum of unoccupied neighbors of the current position.
- we check if we are in a dead end (if we do not have any available free neighbours). If we are in a dead end, we update the second column of the cell array *roads* with the current weight and we simply continue with the next step.
- we define our direction to move by drawing from the unoccupied neighbors of the current position and we update the tuple with the walks (the first column of the cell array *roads*).

Then we estimate the number of walks that are self-avoiding. We compute the estimate as the product of the mean of the weights (stored in the second column of the cell array *roads*). After this, we do the resampling with replacement. We use the MATLAB function *randsample* for this using the normalized weights taken with replacement. Then we update the movements of the walk (the first column of *roads*).

We estimate the parameters μ_d, γ_d and A_d in the same way we did in Task 6, for $d \neq 4$. For $d = 4$, we have to do some changes as it follows:

$$c_n(4) = A_4 \mu_4^n \log(n)^{\frac{1}{4}}$$

$$\log(c_n(4)) = \log(A_4 \mu_4^n \log(n)^{\frac{1}{4}})$$

$$\log(c_n(4)) = \log(A_4) + n\log(\mu_4) + \frac{1}{4}\log(\log(n))$$

$$\log(c_n(4)) - \frac{1}{4}\log(\log(n)) = n\log(\mu_4) + \log(A_4)$$

This means that for $d = 4$, we have that $y = \log(c_n(4)) - \frac{1}{4}\log(\log(n))$, $x_1 = n$, $a = \log(\mu_4)$ and $c = \log(A_4)$. Therefore, the case $d = 4$ was treated separately, in order to compute the parameters in this way.

real μ_3	μ_3	γ_3	A_3
4.5972	4.17153	1.1122	1.2635

Table 8: Estimate of μ_3, γ_3 and A_3 parameters for three dimensions when using $n = 10$ and $N = 1000$ iterations

real μ_4	μ_4	A_4
6.7720	6.8752	1.1926

Table 9: Estimate of μ_4 and A_4 parameters for four dimensions when using $n = 10$ and $N = 1000$ iterations

real μ_6	μ_6	γ_6	A_6
10.8817	10.8658	1.0224	1.1034

Table 10: Estimate of μ_6, γ_6 and A_6 parameters for six dimensions when using $n = 10$ and $N = 1000$ iterations

We know from [2] how to compute the true values for μ_d for different dimension. We also computed this expression in MATLAB and we compared it with our estimated values μ_d. The results can be observed in Tables 8, 9 and 10. As it can be noticed, the estimates are good (very close to the real values) for the three dimensions tested (three, four and six). Therefore, the SISR approach to estimate the parameters for any dimension is a robust method.

Task 10

In general, as it is stated in [5], a hidden Markov model has to stochastic processes: a Markov chain $(X_k)_{k\geq 0}$ with transition density q (in our case, the Markov chain is the relative population size X_k) and an observation process $(Y_k)_{k\geq 0}$. The observation process is conditioned on the chain X_k so that the Y_k's are independent with the conditional distribution of each Y_k depending on the corresponding X_k only [5]. The conditional distribution density $Y_k|X_k$ is noted as $p(y_k|x_k)$. In our case, the conditional distribution is $Y_k|X_k = x \in U(\frac{x}{2}, x)$ and it can be also called the observation density.

Here, the procedure is generally the same as in [4], but there are a few modifications that needs to be done. For simplicity, I kept the same structure and names of the variable as in [4]. First, we use 50 observations, which means that $n = 50$. The observation density for weights is $Y_k|X_k = x \in U(\frac{x}{2}, x)$, which means that p will have the following form: $p = @(x,y)unifpdf(y, x/2, x)$. The particle X_0 is initialized as $X_0 \in U(\frac{1}{5}, \frac{3}{5})$, which means that, for initialization, one should write $part = unifrnd(1/5, 3/5, [N1])$. This is the initial distribution. One should define the stochastic reproduction rate B_{k+1} and initialize it as $B_{k+1} \in U(\frac{1}{2}, 3)$. One last thing needs to be change, which is the mutation rule or the transition density $q(X_{k+1}|X_k)$. The mutation rule is $X_{k+1} = B_{k+1}X_k(1-X_k)$. Therefore one should have $part = B(:, k+1). * part. * (1 - part)$ inside the for loop of the code.

In general, this is the SISR implementation. The observation density is defined, the the particles and the estimation are initialized. Then the approximative 95% confidence interval is estimated at time zero using the weighted sample $(X_i^{0:k}, w_k^i)_{i=1}^N$. The resampling with replacement is done for time zero and the stochastic reproduction rate density B_{k+1} is defined. Then for 50 iterations (or generations) the following steps are done:

- the transition density (the mutation) is defined and the particles are changed accordingly
- the weights are updated according to the observation density and the mutation
- we estimate the filter expectation
- we do the selection by resampling with replacement
- we compute the 95% confidence interval estimated at the current iteration using the weighted sample $(X_i^{0:k}, w_k^i)_{i=1}^N$

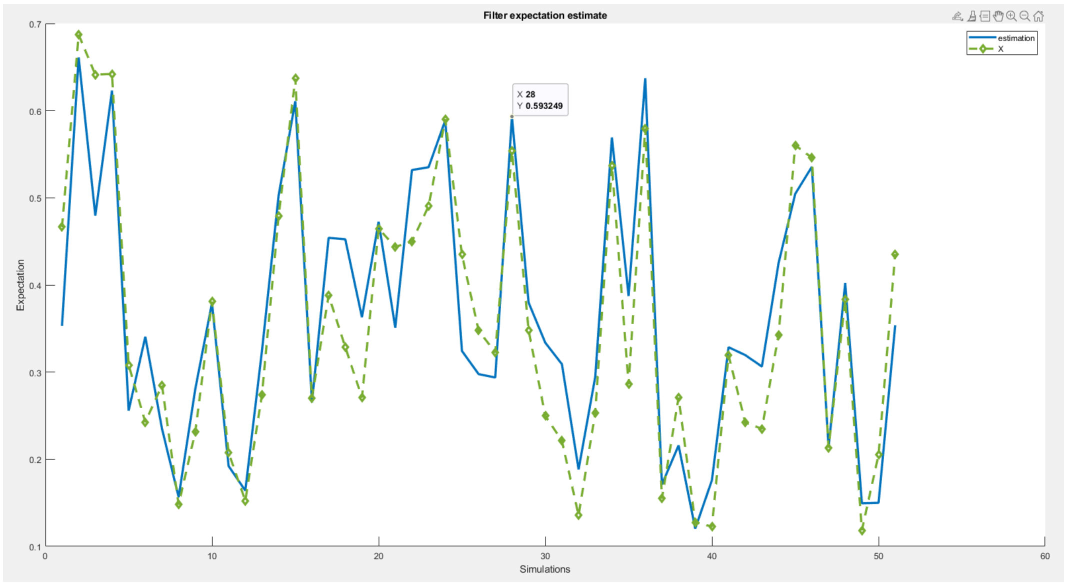

Figure 1: The estimation of the filter expectation τ_k for $k = 0, 1, 2, ..., 50$. Blue line - the estimation τ_k, green line - population size X

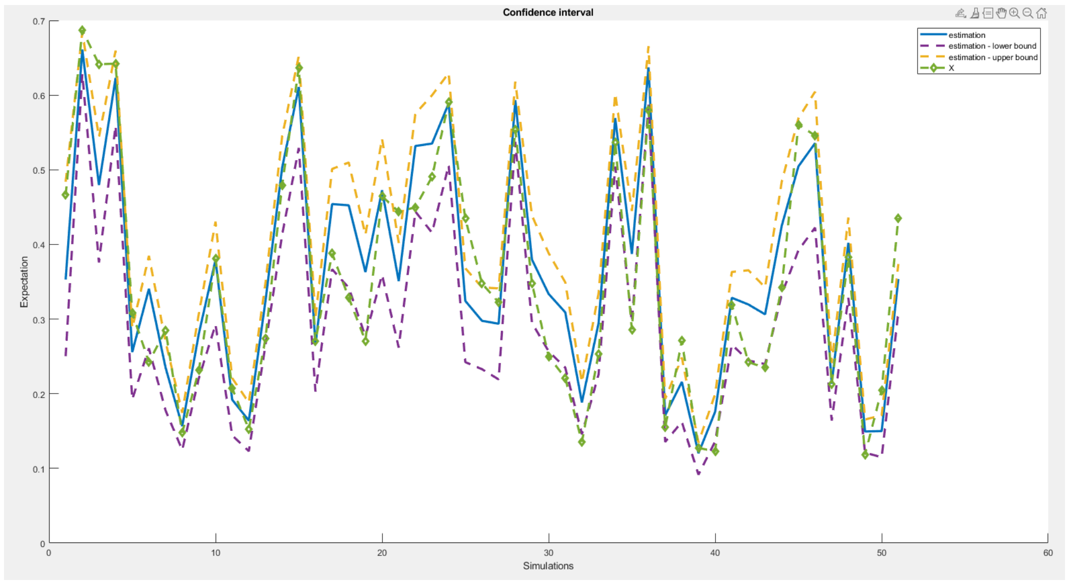

Figure 2: The estimation of the filter expectation τ_k for $k = 0, 1, 2, ..., 50$ with the confidence interval. Blue line - the estimation τ_k, green line - population size, purple line - the lower bound of the confidence interval, yellow line - the upper bound of the confidence interval

The estimated filter expectation τ_k for 50 samples can be observed in Figure 1. In general, the estimation follows the true value X (the blue line has a similar for to the green line), even though that there are some missed peaks. Overall though, the estimation using SISR seems to be robust. The values for the estimated filter expectation τ_k can be also seen in Table 11.

sample	1	2	3	4	5	6	7	8	9	10
τ	0.3539	0.6634	0.4834	0.6237	0.2549	0.3442	0.2418	0.1509	0.2792	0.3768
sample	11	12	13	14	15	16	17	18	19	20
τ	0.1877	0.1645	0.3191	0.5027	0.6135	0.2678	0.4464	0.4575	0.3506	0.4689
sample	21	22	23	24	25	26	27	28	29	30
τ	0.3556	0.5368	0.5284	0.5924	0.3279	0.2932	0.2920	0.5870	0.3800	0.3397
sample	31	32	33	34	35	36	37	38	39	40
τ	0.3176	0.1920	0.2921	0.5622	0.3945	0.6390	0.1722	0.2138	0.1209	0.1751
sample	41	42	43	44	45	46	47	48	49	50
τ	0.3304	0.3120	0.3088	0.4261	0.5172	0.5334	0.2096	0.4002	0.1547	0.1511

Table 11: The estimation of the filter expectation τ_k for $k = 0, 1, 2, ..., 50$

Figure 2 shows the 95% confidence interval for X_k for 50 samples along with the true value X_k and its estimation. In the majority of cases, the true value X_k (green line) is inside the confidence interval (yellow and purple line), but there are some situations where X_k is not inside the interval (for $k = 3, k = 19, k = 30$ etc.). These are the values where the estimation has big differences from the true value.

The numerical value for the 95% point wise confidence interval for X_k and the true values for X_k can be seen in Table 12. Here it can be seen more easily where the true value is inside the confidence interval and where it is not.

sample	1	2	3	4	5	6	7	8	9	10
LB	0.2501	0.6259	0.3720	0.5512	0.1985	0.2574	0.1845	0.1153	0.2223	0.2903
UB	0.4879	0.6790	0.5376	0.6645	0.2828	0.3905	0.2712	0.1667	0.3052	0.4191
X	0.4664	0.6872	0.6413	0.6421	0.3082	0.2426	0.2849	0.1479	0.2315	0.3809
sample	11	12	13	14	15	16	17	18	19	20
LB	0.1398	0.1228	0.2620	0.4077	0.5282	0.2029	0.3634	0.3406	0.2674	0.3612
UB	0.2125	0.1818	0.3410	0.5452	0.6519	0.3079	0.4907	0.5153	0.3979	0.5226
X	0.2075	0.1523	0.2741	0.4791	0.6386	0.2703	0.3880	0.3287	0.2708	0.4646
sample	21	22	23	24	25	26	27	28	29	30
LB	0.2737	0.4510	0.4027	0.5064	0.2490	0.2283	0.2206	0.5362	0.2903	0.2530
UB	0.4024	0.5674	0.5905	0.6376	0.3749	0.3340	0.3319	0.6035	0.4370	0.3886
X	0.4437	0.4498	0.4905	0.5902	0.4347	0.3476	0.3226	0.5536	0.3480	0.2499
sample	31	32	33	34	35	36	37	38	39	40
LB	0.2491	0.1425	0.2236	0.5057	0.2886	0.5835	0.1337	0.1644	0.0896	0.1323
UB	0.3609	0.2157	0.3301	0.5837	0.4637	0.6665	0.1948	0.2424	0.1336	0.1982
X	0.2212	0.1356	0.2533	0.5370	0.2859	0.5797	0.1548	0.2709	0.1275	0.1227
sample	41	42	43	44	45	46	47	48	49	50
LB	0.2652	0.2344	0.2364	0.3316	0.3914	0.4118	0.1586	0.3300	0.1288	0.1175
UB	0.3552	0.3500	0.3508	0.4808	0.5897	0.5868	0.2438	0.4362	0.1722	0.1700
X	0.3193	0.2422	0.2349	0.3428	0.5603	0.5459	0.2131	0.3832	0.1184	0.2050

Table 12: The lower bound (LB), the upper bound (UB) of confidence interval for X_k and the true value X_k for $k = 0, 1, 2, ..., 50$

References

[1] Graham BT. *Borel-type bounds for the self-avoiding walk connective constant.* 2018. URL: https://arxiv.org/pdf/0911.5163.pdf.

[2] Slade G. "The self-avoiding walk: A brief survey." In: (2011), pp. 181–199. URL: https://personal.math.ubc.ca/~slade/spa_proceedings.pdf.

[3] Duminil-Copin H. and Smirnov S. "The connective constant of the honeycomb lattice equals $\sqrt{2+\sqrt{2}}$". In: 175 (2012), pp. 1653–1665. URL: https://doi.org/10.4007/annals.2012.175.3.14.

[4] Wiktorsson M. *Monte Carlo and Empirical Methods for Stochastic Inference.* Lund University, 2022. Chap. Lecture 7.

[5] Wiktorsson M. *Monte Carlo and Empirical Methods for Stochastic Inference.* Lund University, 2022. Chap. Lecture 5.

[6] Kroese D. P. Rubinstein R. Y. *Simulation and the Monte Carlo Method.* John Wiley & Sons, 2017. Chap. 5 Controlling the variance.