Implementation of a Cluster Compatible Renormalization Group Transformation for the Ising Model in a Periodic Volume

Meryl Schio

Supervised by Prof. Dr. Uwe-Jens Wiese
University of Bern
May 2025

Contents

Abstract

In this thesis, a cluster compatible Renormalization Group (RG in the following) transformation for the 2-dimensional Ising model is implemented. For this, the Swendsen-Wang algorithm is modified to include said RG transformation. The results of the implemented algorithm are compared to exact answers, checking the validity of the implementation.

Acknowledgements

First, I would like to thank Prof. Dr. Uwe-Jens Wiese for supervising my thesis and taking the time each week to discuss my progress and supporting me throughout this process. I would also like to thank him for the numerous occasions of debugging my simulation together. Furthermore, I would like to thank Fabio Arz and Nico Scheidegger for the countless meetings in which we discussed our ideas with each other, greatly helping with progressing in our theses. I would also like to thank my friends Norina Michel and Thierry Imboden for always listening to my endless ramblings about this project. Special thanks go out to Wout and Mila Mergaerts for providing me with continuous support and motivating me to finish this thesis. Lastly, I want to thank my parents for always being there for me, especially my dad who always believed in me; I would not be where I am today if not for him.

1 Introduction

Spontaneous symmetry breaking is a phenomenon that occurs in ferromagnets. It is related to a second order phase transition in the system, causing spontaneous magnetization without the presence of an external magnetic field [1]. Said phase transition occurs at a critical temperature. It can be measured experimentally how observables such as the magnetization or the magnetic susceptibility change near the critical temperature. This behaviour is dictated by the so-called critical exponents of those observables. There are different systems that have the same critical exponents, even though they do not seem to be related to each other. To perform numerical computations for such systems, classical spin models are used, such as the Ising model. Symmetries play an important role in the universal behaviour of different systems.

Kadanoff [2] first discussed so-called block spin transformations to describe the scaling of critical phenomena at different length scales. This is achieved by taking a lattice and grouping its spins into blocks which result in a block spin. This results in a new lattice with a larger lattice spacing, causing the length scale to increase. Near criticality, a system behaves in a similar way independent of the length scale, it exhibits scale invariance. An example for this is the correlation length that diverges at the critical point. After coarse-graining the system with such a block spin transformation, the correlation length still diverges.

Wilson [3], [4] formalised this idea of the renormalization group (RG) that discusses how parameters of a system change at different length scales, such as how couplings change under RG transformations, leading to new Hamilton functions with new couplings. In the space of couplings, there exists a high-dimensional manifold called the critical surface. It has the property that the correlation length is infinite at every point on the critical surface. Repeatedly performing RG transformations on the critical surface causes the system's Hamilton function to converge to a fixed point on the critical surface. This gives rise to universality in different statistical models, explaining why they can have the same critical exponents; their Hamilton functions flow to the same fixed point on the critical surface.

This thesis is motivated by the idea of Prof. Dr. Uwe-Jens Wiese to develop a new RG transformation for the Ising model that can be simulated with an efficient cluster algorithm. Such a cluster algorithm includes other interactions besides the nearest-neighbour interaction with their own couplings, hopefully allowing to study the fixed point Hamilton function. The efficiency of a cluster algorithm at criticality should make it possible in the future to implement sufficiently many RG steps to reach the vicinity of the fixed point. Additionally, the ability to implement RG transformations on larger lattices implies a reduction of finite-size effects.

First, the Ising model with its basic properties on a triangular lattice and the cluster representation used for the cluster algorithm are discussed. Then, the general approach to RG transformations and its specialisation to a cluster compatible RG transformation are outlined. This is achieved by changing the parametrisation of the Hamilton function. With this, nontrivial fixed points in a finite, periodic volume are found. For example, said finite volume consists of a fine lattice with 12 sites and a blocked coarse lattice with 4 sites. Next, the implementation of a cluster algorithm is shown, divided in two parts. The first part concerns itself with including other interactions besides the nearest-neighbour interaction and the second part introduces the RG transformation. Finally, different observables are computed using the cluster algorithm and compared to results obtained from an exact calculation on the same lattice.

2 Theoretical Background

2.1 Hamilton Function, Observables, and Critical Coupling

Let us consider the Ising model on a triangular lattice Λ with lattice spacing a with the Hamilton function

$$\mathcal{H}[s] = -J \sum_{\langle xy \rangle} s_x s_y, \tag{1}$$

where J is the coupling constant, $\langle xy \rangle$ denotes the nearest neighbours and $s_x = \pm 1$ is a spin at lattice point x. Such a lattice is depicted in Fig. 1.

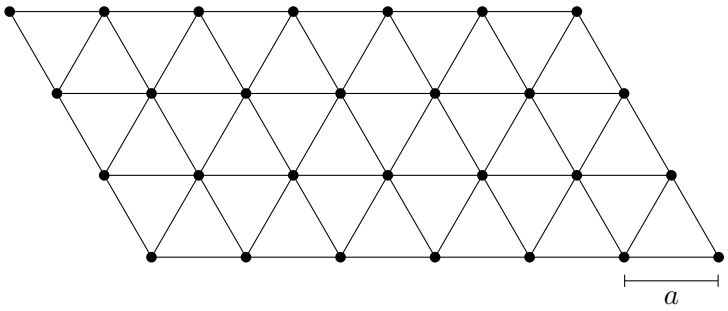

Figure 1: Triangular lattice with lattice spacing a.

The partition function [5] is given by

$$Z = \sum_{[s]} e^{-\beta \mathcal{H}[s]} = \prod_{x \in \Lambda} \sum_{s_x = \pm 1} e^{-\beta \mathcal{H}[s]} = \prod_{x \in \Lambda} \sum_{s_x = \pm 1} \prod_{\langle xy \rangle} e^{\beta J s_x s_y}, \tag{2}$$

where $\beta = 1/k_B T$ is the inverse temperature and $\sum_{[s]}$ indicates the sum over all configurations $[s]$ that can also be expressed as a product which refers to the sum over both possible spin values for each spin s_x for each site x. For a given observable $O[s]$, the expectation value $\langle O \rangle$ is defined as a sum over all configurations of the observable's value $O[s]$ weighted with its Boltzmann factor $e^{-\beta \mathcal{H}[s]}$, and divided by the partition function

$$\langle O \rangle = \frac{1}{Z} \sum_{[s]} O[s] e^{-\beta \mathcal{H}[s]}. \tag{3}$$

The magnetization $M[s]$ for a given configuration is given by

$$M[s] = \sum_x s_x. \tag{4}$$

Using the expectation values of M and M^2, the magnetic susceptibility χ is defined as follows

$$\chi = \frac{1}{N} \left(\langle M^2 \rangle - \langle M \rangle^2 \right) = \frac{1}{N} \langle M^2 \rangle, \tag{5}$$

where N is the number of lattice sites and $\langle M \rangle = 0$ since the Ising model has a $\mathbb{Z}_2$ symmetry if there is no magnetic field present. The spin correlation function $\langle s_x s_y \rangle$ decays exponentially and is given by

$$\langle s_x s_y \rangle = \frac{1}{Z} \sum_{[s]} s_x s_y e^{-\beta \mathcal{H}[s]} \sim e^{-|x-y|/\xi}, \tag{6}$$

where ξ is the correlation length. In $d \geq 2$ dimensions, the Ising model has a second-order phase transition at which the correlation length diverges at a critical coupling βJ_c. At the critical point, the correlation length diverges. Moreover, there is spontaneous symmetry breaking with the magnetization as the corresponding order parameter. In the broken phase with $\beta J > \beta J_c$, there is a spontaneous magnetization that approaches $M = \pm N$ at zero temperature meaning that all spins are aligned, whereas in the unbroken phase with $\beta J < \beta J_c$ the spins are unordered with no magnetization. The critical coupling for the triangular lattice in two dimensions can be derived as follows [6]: Using its duality to a honeycomb lattice, which leads to a triangular lattice again upon decimation together with its self-duality, giving the critical coupling

$$e^{2\beta J_c} = \sqrt{3} \implies \beta J_c = \frac{1}{2} \log \sqrt{3} \approx 0.274653. \tag{7}$$

Lastly, the distribution of the magnetization on the lattice can be considered in the broken and unbroken phase. It is computed as

$$p(M) = \frac{1}{Z} \sum_{[s]} e^{-\beta \mathcal{H}[s]} \delta_{M[s],M}. \tag{8}$$

2.2 Swendsen-Wang Cluster Algorithm

We can introduce bond variables $b_{\langle xy \rangle} = 0, 1$ to rewrite the partition function as follows

$$Z = \sum_{[s]} \sum_{[b]} e^{-\beta \mathcal{H}[s,b]} = \sum_{[s]} \prod_{\langle xy \rangle} \sum_{b_{\langle xy \rangle}=0,1} e^{-\beta \mathcal{H}[s,b]}, \tag{9}$$

where the Boltzmann factor can be further rewritten as

$$e^{-\beta \mathcal{H}[s,b]} = \prod_{\langle xy \rangle} e^{-\beta h(s_x, s_y, b_{\langle xy \rangle})}, \tag{10}$$

where $e^{-\beta h(s_x, s_y, b_{\langle xy \rangle})}$ stands for the contribution to the Boltzmann weight of two neighbouring spins depending on whether their bond is activated or not. The possible contributions of two spins can be written as

$$e^{-\beta h(s_x, s_y)} = \sum_{b_{\langle xy \rangle}=0,1} e^{-\beta h(s_x, s_y, b_{\langle xy \rangle})}, \tag{11}$$

with their explicit values

$$\begin{aligned} e^{-\beta h(s,s)} &= e^{\beta J} \\ e^{-\beta h(s,-s)} &= e^{-\beta J}. \end{aligned} \tag{12}$$

Bonds can connect two neighbouring spins with an activated bond. An activated bond $b = 1$ connects two parallel spins while a deactivated bond $b = 0$ leaves two spins uncorrelated. The bond between two antiparallel spins is never activated, whereas the bond between two parallel spins may be activated. To obtain the probability to activate a bond between two spins, the following cluster rules are used:

- If the two spins are parallel, the bond may be activated with a certain probability.
- It does not matter whether the spins are parallel or not if the bond is deactivated; both cases have the same probability.
- A bond between antiparallel spins is never activated.

The above rules are illustrated in Fig. 2. First, we consider the case of two antiparallel spins. The contribution to the Boltzmann weight is $e^{-\beta J}$ and the bond is never activated, therefore the antiparallel spins with inactive bond receive a weight of $e^{-\beta h(s,-s,0)} = e^{-\beta J}$. Thus, the parallel spins with inactive bond have a weight of $e^{-\beta h(s,s,0)} = e^{-\beta J}$ as well. In order to obtain an overall factor of $e^{\beta J}$ for the parallel spins, the parallel spins with active bond receive the weight $e^{-\beta h(s,s,1)} = e^{\beta J} - e^{-\beta J}$.

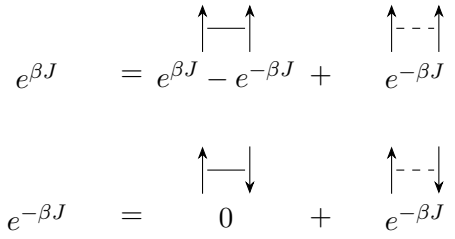

Figure 2: Cluster rules for the Ising model.

To arrive at the probability p for the bond to be activated between parallel spins, we take the weight of the parallel spin pair with activated bond divided by the overall Boltzmann factor of the parallel spins

$$p = \frac{e^{\beta J} - e^{-\beta J}}{e^{\beta J}} = 1 - e^{-2\beta J}. \tag{13}$$

This means that a configuration can be decomposed into clusters of parallel spins given by the active bonds that connect said spins. Spins within the same cluster are correlated, while spins from different clusters are uncorrelated, even if the spins in both clusters are parallel. Moreover, a cluster can consist of a single spin. The decomposition into clusters C can be used to rewrite the susceptibility in terms of the clusters. First, the magnetization can be written as the sum of the magnetizations of the individual clusters M_C

$$M[s] = \sum_C M_C. \tag{14}$$

Plugging this in Eq. (5) yields

$$\chi = \frac{1}{N} \left\langle \left(\sum_C M_C \right)^2 \right\rangle = \frac{1}{N} \left\langle \sum_C M_C^2 \right\rangle, \tag{15}$$

which can be simplified further by using the fact that different clusters are uncorrelated, namely $\langle M_{C_1} M_{C_2} \rangle = 0$, meaning that only the square terms $\langle M_C^2 \rangle$ are left and that the magnetization of a cluster is given by its size $|C|$ up to the sign. This gives an improved estimator for the susceptibility

$$\chi = \frac{1}{N} \left\langle \sum_C |C|^2 \right\rangle. \tag{16}$$

The Swendsen-Wang algorithm [7] uses clusters to efficiently simulate the Ising model. One sweep over the lattice consists of three steps: In the first step, it is decided for each nearest-neighbour pair whether their bond is activated according to the above rules. Then, the individual clusters are identified and each cluster is flipped with 50% probability.
Explicitly, set up a vector[1] of booleans that stores the spin values of all N spins. In the following, the index going from 0 to $N-1$ is used to refer to the spins in the vector. With this, generate a vector with $3N$ entries and store the indices of each spin's three nearest-neighbours. Technically, each spin has 6 nearest-neighbours on the triangular lattice, but for this purpose only consider three of them to avoid double counting when performing a loop over the lattice. Otherwise, each nearest-neighbour pair would be checked twice, and a bond that is not activated in the first check gets another chance to be activated, which indirectly changes the probability to activate a bond. Now set up an $N \times N$ matrix of zeros and loop over the lattice to check which bonds are activated. If a bond between two spins with index x_1 and x_2 is activated, set the entries in the bond matrix at the indices (x_1, x_2) and (x_2, x_1) to 1, since the bonds are symmetric. Next, start identifying the individual clusters. For this, set up another vector of booleans which records the spins that have been visited, or have been added to the cluster already. Start with the first spin on the lattice and flag this spin as visited, assign it the cluster number 0 and add it to a queue[2]. Then, assign that spin to a variable z, remove it from the front of the queue and add all the spins with an activated bond to z and have not been visited yet to the queue. Repeat the following steps until the queue is empty: Assign the new front of the queue the cluster number 0, assign that spin to z and add all spins with activated bond to z and have not been visited yet to the queue. It is allowed that the cluster only consists of the starting spin and that no additional spins are added with the above procedure. After the first cluster is identified, increase the current cluster number to 1 and move on to the next spin on the lattice that is not part of a cluster yet. Repeat the building of the cluster in the same way as for the first cluster and then move on to the next cluster again. This is repeated until all spins are assigned to a cluster. When all the clusters are built, they are flipped with a probability of 50% each. In preparation for the next sweep, all bonds are reset and all spins are flagged as not yet visited again.

[1] In the following, vectors refer to containers that store elements in a defined order.
[2] A queue is a container that stores its elements in FIFO (First In, First Out) order.

2.3 General Approach to RG Transformation

We consider the Ising model on a triangular lattice as in Section 2.1. We now decompose this fine lattice Λ into blocks $c_{x'}$ that are centered at the coarse lattice points x' that make up the coarse lattice Λ'. Such a lattice is depicted in Fig. 3 below, where the grey dots and dashed lines indicate the coarse lattice. This lattice has the lattice spacing $a' = \sqrt{3}a$, therefore this blocking transformation has a blocking factor of $\sqrt{3}$.

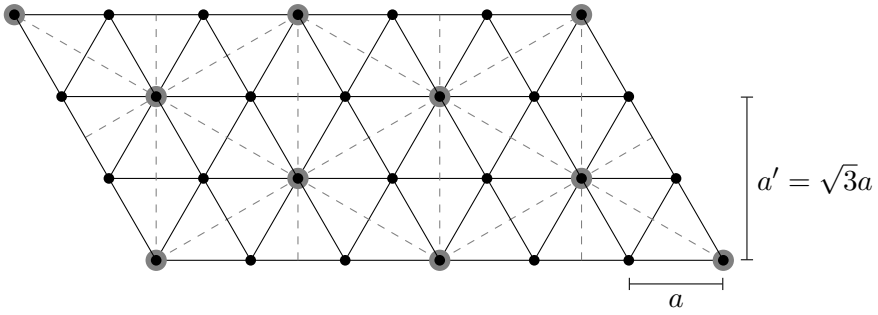

Figure 3: Fine and coarse triangular lattice.

Now we define a blocking kernel $T[s, s']$ that blocks the fine lattice onto the coarse lattice and satisfies

$$\sum_{[s']} T[s, s'] = \prod_{x'} t(x') = 1, \tag{17}$$

where $t(x')$ represents the contribution of a single block to the blocking kernel. With this, the effective Hamilton function on the coarse lattice is then given by

$$e^{-\beta\mathcal{H}'[s']} = \sum_{[s]} T[s, s'] e^{-\beta\mathcal{H}[s]}. \tag{18}$$

The above equation together with the normalisation condition for the blocking kernel show that the blocking transformation maintains the partition function

$$Z' = \sum_{[s']} e^{-\beta\mathcal{H}'[s']} = \sum_{[s]} \overbrace{\sum_{[s']} T[s, s']}^{=1} e^{-\beta\mathcal{H}[s]} = Z. \tag{19}$$

The critical point of the Ising model lives in a high-dimensional space of couplings and is part of the so-called critical surface. The critical surface contains all points with $\xi = \infty$. If one starts on the critical surface and performs the RG transformation, the blocked Hamilton function is still on the critical surface, but closer to the fixed point; the RG transformation moves the Hamilton function along an irrelevant direction. If one starts outside of the critical surface and then performs the RG transformation, the Hamilton function is moved toward a relevant direction, causing it to approach either the zero-temperature or the infinite-temperature fixed point.

2.4 Cluster-compatible RG

Let us consider the Ising model on a triangular lattice as in the previous sections. Now we define the blocks on the fine lattice such that they overlap as depicted in Fig. 4. The shaded hexagon represents one of the blocks. The blocks consist of 7 fine spins; 1 central spin that is unique to its block and 6 peripheral spins that belong to three different blocks.

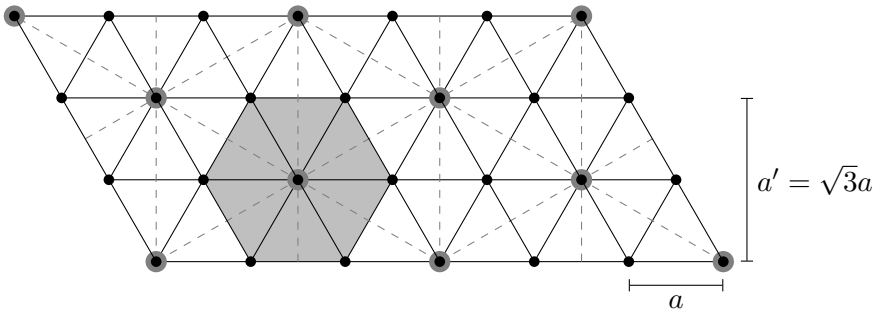

Figure 4: Triangular lattice with overlapping blocks.

The blocking kernel for a given fine and coarse configuration can be decomposed into the factors contributed by the configurations on the blocks themselves

$$T[s, s'] = \prod_{x'} t(s_x, x \in c_{x'};\, s'_{x'}), \tag{20}$$

where t denotes the factor contributed by one block, $s_x, x \in c_{x'}$ are the fine spins in the block, and $s'_{x'}$ denotes the coarse spin of the block. In order to define cluster-compatible blocking transformations, we consider the following cluster rules:

- No bond is set between the coarse spin and one of the fine spins in the block (parameter A)
- The bond is set between the coarse spin and the parallel central fine spin in the block (parameter B)
- The bond is set between the coarse spin and one of the peripheral fine spins in the block that is parallel to the coarse spin (parameter C)

The above cluster rules are illustrated in Fig. 5 below

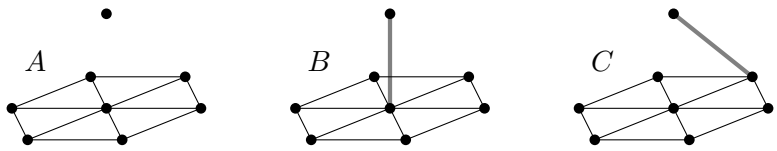

Figure 5: Cluster rules for overlapping blocks.

A bond can only be set between 2 spins if both of these spins are parallel to each other. The above parameters contribute to the factors of the blocking kernel for every bond that could be set for a given block configuration. Listed below are all possible values for t. Note that the first spin listed among the fine spins is the central spin in the block:

$$\begin{aligned}
t(+++++++;+) &= A+B+6C, & t(-------;+) &= A,\\
t(+-++++++;+) &= A+B+5C, & t(+------;+) &= A+C,\\
t(+--+++++;+) &= A+B+4C, & t(-++----;+) &= A+2C,\\
t(+---++++;+) &= A+B+3C, & t(-+++---;+) &= A+3C,\\
t(+----+++;+) &= A+B+2C, & t(-++++--;+) &= A+4C,\\
t(+-----++;+) &= A+B+C, & t(-+++++-;+) &= A+5C,\\
t(+------+;+) &= A+B, & t(-++++++;+) &= A+6C.
\end{aligned} \tag{21}$$

Note that the kernel values are the same for the cases where all spins are flipped due to the $\mathbb{Z}_2$ symmetry of the Ising model. Asserting that there is always exactly one active bond between each coarse spin and one of their respective fine spins, A can be set to 0. Additionally, leaving $A \neq 0$ would imply that at some point in the iteration of the blocking transformations there would be a gap in the connection between the current fine and coarse lattice, causing the inability to reach the fixed point Hamiltonian. Using that, the symmetry as well as Eq. (17), it can be seen that the sum of both kernels on each line satisfy the normalisation condition for this blocking transformation

$$B + 6C = 1. \tag{22}$$

The special case of $B = 1$ and $C = 0$ corresponds to decimation, meaning that the coarse spin in each block is always determined by its central spin on the fine lattice. In the following, the cases with $(B = 0, C = 1/6)$, $(B = 1/7, C = 1/7)$ and $(B = 1/4, C = 1/8)$ are considered.

2.5 Modified RG in a Periodic Volume

Implementing RG transformations requires using finite, periodic lattices of different volumes. Performing the following exact calculations on a multigrid with a sufficient number of RG transformations is not practical due to the rapidly growing size of the starting lattice. Explicitly, after each RG step, the number of lattice sites is reduced to a third of the lattice sites of the previous lattice. As an example, we assert that the coarse lattice on the multigrid consists of 4 lattice sites. This corresponds to $2^4 = 16$ configurations. The previous lattice then consists of 12 lattice sites with $2^{12} = 4096$ configurations, which is still manageable. Going back another step means the lattice now has 36 lattice sites with $2^{36} = 68'719'476'736$ configurations, which becomes unmanageable.

2.5.1 Insertion Operation

This issue can be solved by introducing an insertion operation that follows the blocking transformation. After the blocking transformation an insertion is performed to map the couplings on the coarse lattice back to the fine lattice, such that the blocking can be performed again without needing a third lattice. This is illustrated in Fig. 6 below. The unprimed Hamilton functions denote the Hamilton functions on the fine lattice, the primed ones denote the ones on the coarse lattice and the index stands for the

current iteration. Note that after a certain number of iterations, the fixed point $\mathcal{H}^*$ is reached assuming that $\mathcal{H}_0$ corresponds to the Ising model at the pseudo-critical coupling associated with the finite system.

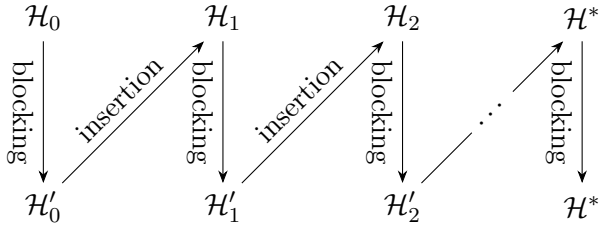

Figure 6: Scheme of blocking and insertion steps.

2.5.2 Parametrisation of the Hamilton Function

For this, we parametrise the Hamilton function in a different way that also allows it to be simulated with a cluster algorithm. We introduce a graph Γ, a set of sites on the lattice. Not all of the sites in the graph have to be connected to each other. Graphs that are the same under symmetry transformations (translations, rotations and reflections) belong to the same equivalence class $\tilde{\Gamma}$. All graphs in the same equivalence class $\tilde{\Gamma}$ contribute to the Boltzmann factor with the same weight $W_{\tilde{\Gamma}}$. With this, we define a new Kronecker δ-function that is 1 if all connected spins in the graph are parallel and 0 otherwise

$$\delta_\Gamma[s] = \begin{cases} 1 & \text{if } s_x = s_\Gamma \ \forall\ x \in \Gamma,\ s_\Gamma = \pm 1 \\ 0 & \text{otherwise} \end{cases}. \tag{23}$$

With this, the Hamilton function can be parametrised as

$$e^{-\beta\mathcal{H}[s]} = \prod_{\tilde{\Gamma}} \prod_{\Gamma\in\tilde{\Gamma}} W^{\delta_\Gamma[s]} = \prod_i W_i^{n_i([s])}, \tag{24}$$

where i runs over all graph equivalence classes and n_i denotes the number of times the corresponding graph is found in the configuration $[s]$. This means that a graph Γ contributes a factor of $W_{\tilde{\Gamma}}$ if all spins in the graph are parallel, otherwise it contributes a factor of 1. While the detailed explanation of the insertion operation can be found in [8] and [9], the process shall be outlined here for the case of a $12 \to 4$ blocking.

2.5.3 12 → 4 Blocking with Overlapping Blocks

The lattices chosen for the computations consist of a fine lattice with 12 sites and a coarse lattice with 4 sites. As pictured in Fig. 7, the 12 fine lattice sites have been numbered from 0 to 11 in the following way, where the greyed out sites denote periodic copies of the lattice sites.

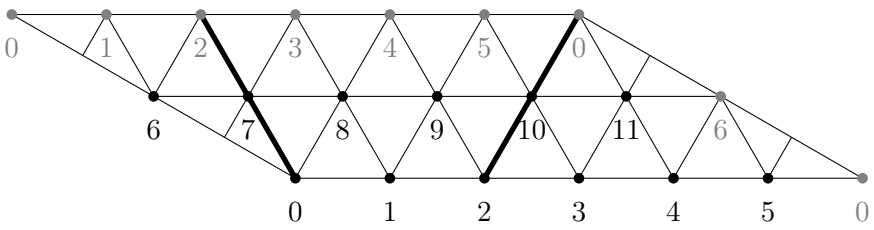

Figure 7: Numbering of the fine lattice sites.

For a more intuitive picture of the coarse lattice and what the blocks on the fine lattice are, the above parallelogram can be "cut" along the thicker lines and be rearranged as shown in Fig. 8. The 4 blocks on the lattice for this case are shaded in different colors, and the numbers of the coarse lattice sites are denoted with the grey numbers. The blocks in this case consist of 7 spins each, where the central spin only contributes to 1 block, whereas the peripheral spins in each block contribute to 3 blocks.

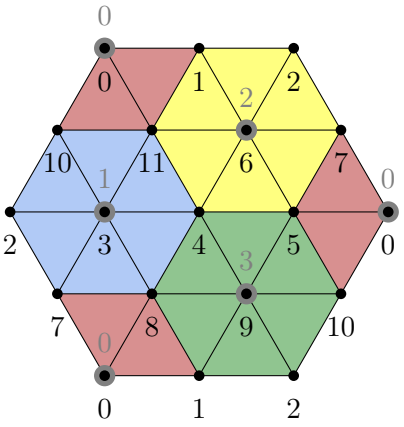

Figure 8: The blocks on the fine lattice and their respective coarse spin.

2.5.4 The Used Graphs

For the iteration process in this case, two graphs are needed. Additionally to the nearest-neighbour graph that corresponds to the standard Ising model interaction, a 4-spin graph of two disconnected pairs is chosen. In the following, the nearest neighbour graph and its weight are denoted with W_1 and the 4-spin graph and its weight are denoted with W_2. It is essential to account for every instance of a graph on the lattice. In order to go over the lattice to find the number of nearest-neighbour graphs n_1 exactly once, the procedure is as follows: In the triangular lattice, each lattice site has 6 nearest neighbours. It is sufficient to consider 3 of those directions, which also avoids double counting. An example of 3 such directions is illustrated in Fig. 9. For the 4-spin graph, there are in total 6 variations to be considered starting from each lattice site, also illustrated in Fig. 9. The 4-spin graph can appear in 3 different orientations, with 2 possible types of pairs in each of them. Therefore, the number of 4-spin graphs n_2 can be accounted for by looping over the entire lattice and checking the aforementioned variations of the graphs for every lattice site.

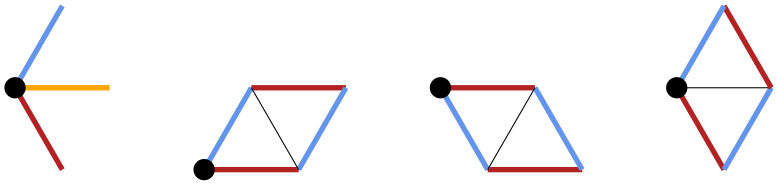

Figure 9: All variations of the chosen graphs to be considered.

2.5.5 Explicit Blocking Transformation and Insertion Operation

For the blocking transformation, the coarse configurations are grouped according to their unique Boltzmann weights. Due to symmetries and periodic boundary conditions, there are three such types of coarse configurations. They are pictured in Fig. 10 below with the corresponding powers (n_1, n_2) and are numbered as shown.

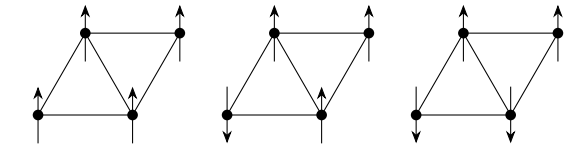

Figure 10: Independent coarse configurations with the number of instances of the graphs W_1 and W_2.

Then, the Boltzmann weights of these three coarse configurations are computed using Eqs. (18) and (24)

$$e^{-\beta\mathcal{H}[s'_i]} = \sum_{[s]} T[s, s'_i] W_1^{n_1([s])} W_2^{n_2([s])}, \tag{25}$$

where $n_{1,2}$ now denote the number of graphs appearing in the fine configuration $[s]$. The goal of the insertion operation is to determine new weights W'_1 and W'_2 such that the coarse Boltzmann weights can be expressed as

$$e^{-\beta\mathcal{H}[s'_i]} = W_1'^{n_1([s'_i])} W_2'^{n_2([s'_i])}, \tag{26}$$

where $n_{1,2}$ are the number of graphs in the coarse configurations again. This can be achieved by writing the above equations as a system of linear equations, for which the logarithm needs to be taken on both sides.

$$\begin{pmatrix} \log(e^{-\beta\mathcal{H}[s'_1]}) \\ \log(e^{-\beta\mathcal{H}[s'_2]}) \\ \log(e^{-\beta\mathcal{H}[s'_3]}) \end{pmatrix} = \begin{pmatrix} n_1([s'_1]) & n_2([s'_1]) & 1 \\ n_1([s'_2]) & n_2([s'_2]) & 1 \\ n_1([s'_3]) & n_2([s'_3]) & 1 \end{pmatrix} \cdot \begin{pmatrix} \log W'_1 \\ \log W'_2 \\ 1 \end{pmatrix}. \tag{27}$$

The added column of 1s in the matrix and the additional 1 in the vector of weights W'_i is needed such that the matrix can be inverted and the new weights W'_i can be obtained. These new weights are now treated as the weights W_i in the beginning of the blocking transformation in Eq. (25) and the next iteration can begin. This process can be repeated until one of the fixed points is reached. In order to find the nontrivial fixed point, the initial weights are chosen to be the pseudo-critical coupling $W_{1,c}$ of the Ising model for W_1 and W_2 is set to 1. $W_{1,c}$ is not known a priori, therefore it needs to be fine-tuned to lead to the nontrivial fixed point. If the initial value is lower than $W_{1,c}$, the iteration will lead to the $T = \infty$ fixed point. Similarly, if the initial value is higher than $W_{1,c}$, the iteration will lead to the $T = 0$ fixed point. The nontrivial fixed point can also be reached by starting with the critical coupling of the 4-spin interaction $W_{2,c}$ and setting W_1 to 1. Note that in the following, we only consider the nontrivial fixed points where both weights are at least 1. This is needed such that the Monte Carlo simulation explained in Section 3.1 can be performed.
Additional computations can be performed with this exact calculation to ensure that the cluster algorithm works as desired. The susceptibility and the magnetization distribution

are calculated from Eqs. (5) and (8). The Boltzmann weights of the coarse configuration classes can also be used to verify the results. For this comparison, we need to take into account in the exact calculation how many times each of the independent configurations appears.

2.6 Fixed Points in 12 → 4 Blocking

Using the approach outlined in Section 2.5, different pairs of the parameters B and C that lead to a nontrivial fixed point are determined. Three such pairs are listed in Table 1 below. Note that $W_{i,c}$ denote the critical couplings used to start the iteration process and W_i^* are the couplings of the fixed point Hamiltonian.

Table 1: Parameters B and C and their respective critical and fixed point couplings.

B	C	$W_{1,c}$	$W_{2,c}$	W_1^*	W_2^*
0	1/6	2.0850432	1.23602718	1.86816	1.03087
1/7	1/7	2.179655	1.2470534	2.05816	1.01566
1/4	1/8	2.268248	1.25709285	2.24875	1.00231

It can be seen that increasing B causes the fixed point to shift to larger values of W_1^* and smaller values of W_2^*. Increasing the value of B even further will lead to at least one of the weights becoming smaller than 1. This also happens in the case of decimation ($B = 1$ and $C = 0$). Comparison of the pseudo-critical coupling of the Ising model with the analytical value of $\sqrt{3} \approx 1.732051$ shows a difference that is most likely due to finite size effects. Decreasing values of B show decreasing values of the pseudo-critical coupling that might decrease even more if B could be further decreased. Having a negative value of B does not make sense in this context, so a fine-tuning of the RG parameters to force the pseudo-critical coupling to be equal to the critical coupling of the infinite system is not possible in this case.

3 Cluster Algorithm on a 2-Layer Multigrid

3.1 Cluster Rules for RG Step and Graphs

The cluster algorithm consists of 2 parts: A modified version of the Swendsen-Wang algorithm for the Ising model on the fine lattice and the part that represents the RG transformation between the fine and coarse lattice.

3.1.1 Setting the Bonds on the Fine Lattice

Each sweep starts with setting the bonds on the fine lattice only. For every site it is checked whether the graphs occur as in Section 2.5. For any graph that is found to be satisfied during said process, the bond between the parallel spins is activated with a certain probability. The cluster rules are equivalent to those of the Ising model in Section 2.2. For clarity, they are illustrated for the nearest-neighbour graph in Fig. 11 below.

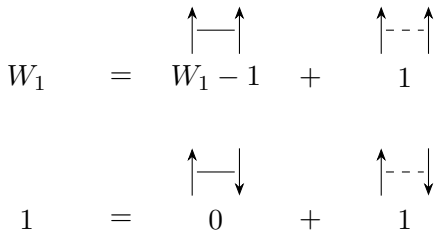

Figure 11: Rules for bond activation and the respective probabilities.

This relation holds for the 4-spin graph as well, therefore the probability to activate the bond of a graph with weight W_i is:

$$p_i = \frac{W_i - 1}{W_i} = 1 - \frac{1}{W_i}. \tag{28}$$

In order for the cluster logic to be applicable in the Monte Carlo simulation, we can see in the above equation that for all weights $W_i \geq 1$ must hold such that $p_i \geq 0$ is fulfilled.

3.1.2 Setting the RG Bonds

In each time step, after setting the bonds on the fine lattice, the bonds between the fine and coarse lattice are set. Each coarse spin couples to exactly one fine spin in each sweep. This is achieved by determining each fine spin in the block that is parallel to the coarse spin and then randomly choosing one of said spins by taking the values of the parameters B and C into account. In order to be able to sample, a random number in the interval $[0, 1)$ is used to choose the fine spin that couples to the coarse spin, the factors of all parallel spins are summed as a normalisation factor q; namely a factor of B if the central spin is parallel, and a factor of C for each peripheral spin that is parallel. Then, the probability to choose one of the parallel spins is the value of the parameter of the respective spin (B or C) divided by q.
In order to only sample one random number r per coarse spin, the fine spin that will couple to the coarse spin is determined as pictured in Fig. 12. Note that the equations in the flowchart refer to the assignments of q and r in the sense of programming. First, q is determined by checking all spins in the block that are parallel and adding them to

a queue. Then, a random number r is sampled. A spin couples to the coarse spin if its probability (B/q for the central spin and C/q for the peripheral spins) is larger than r. If that is not the case for the first spin in the queue, r is decreased by the probability of that spin and the spin is removed from the queue. This is iteratively repeated until the probability of the current spin becomes larger than r. It is guaranteed that at least one spin in the block is parallel to the coarse spin because configurations where no spin is parallel are forbidden. In the case of only one parallel spin, the condition to activate the bond becomes $r < 1$, which is always fulfilled.

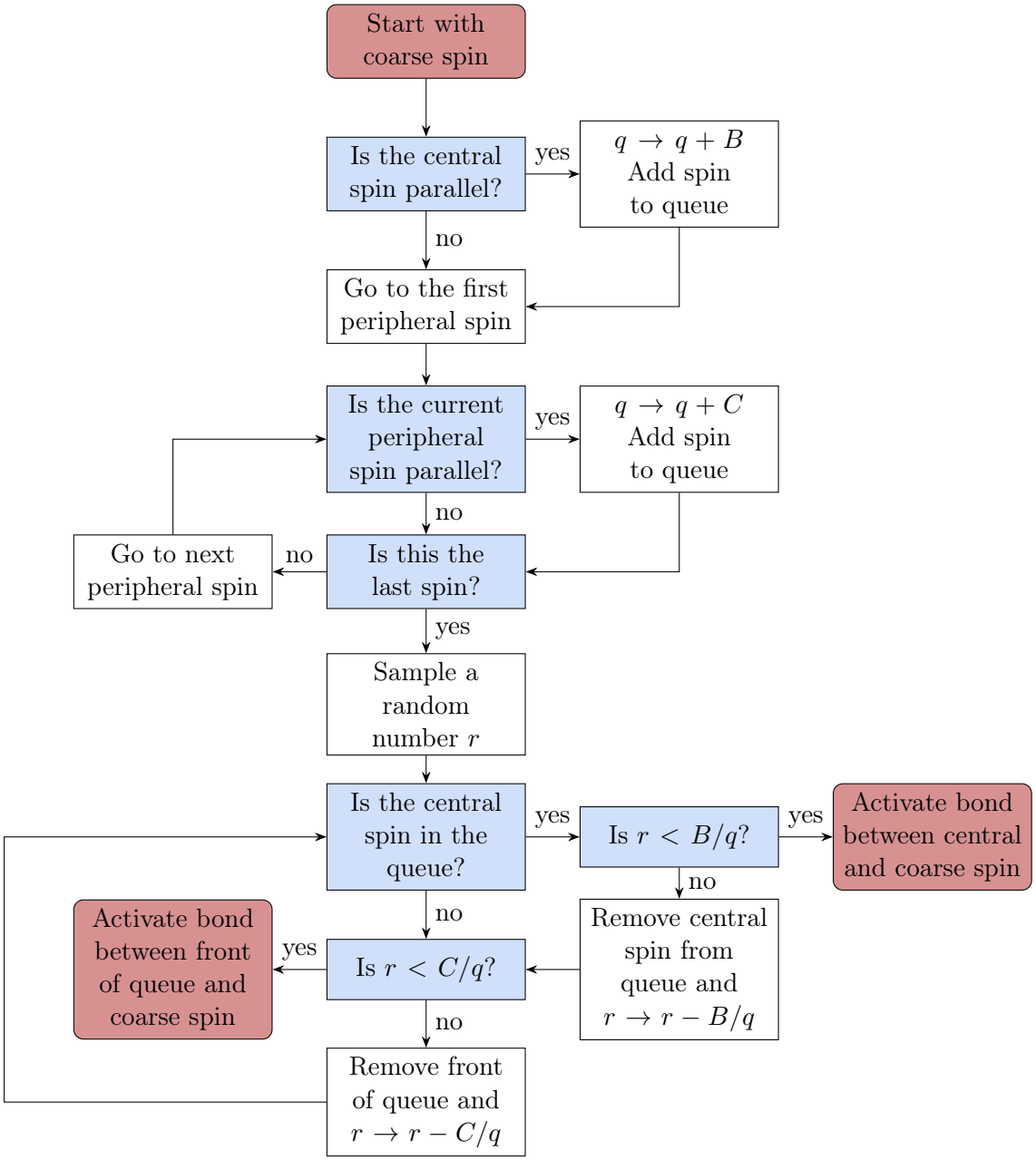

Figure 12: Flowchart depicting setting of one coarse bond.

3.1.3 Building and Flipping the Clusters

After setting the bonds on the fine lattice and between the fine and coarse lattice, the individual clusters are identified. Start with the first spin on the fine lattice and flag this spin as visited, assign it the cluster number 0 and add it to a queue. Then, assign that spin to a variable z, remove the front of the queue and add all the spins with an activated bond to z that have not been visited yet to the queue. All these spins are also assigned the cluster number 0 and are flagged as visited. This is repeated until the queue is empty and no more spins are added to the cluster. It is also allowed that only the starting spin makes up a cluster and that no spins are added to its cluster. After the first cluster has been built, increase the current cluster number to be 1 and move on to the next spin on the lattice that is not part of cluster 0. Repeat the building of the cluster in the same way as for the first cluster and then move on to the next cluster again. This is repeated until all spins on both lattices have been assigned to a cluster. In this procedure, there is no special distinction between the coarse spins and the fine spins. Because every coarse spin has exactly one activated bond with a fine spin, they will be added to the cluster when the respective fine spin is reached in the cluster building procedure. When all the clusters are built, they are flipped with a probability of 50% each. In preparation for the next sweep, all bonds are reset and all spins are flagged as not yet visited again.

3.1.4 Observables

The magnetic susceptibility χ on the fine and coarse lattice are computed according to Eq. (16). Since there are two lattices involved, simply taking the overall size of the cluster does not distinguish between the fine and coarse lattice. Therefore, during the building of the clusters it is determined whether the spin to be added to the current cluster is a fine or coarse spin and two separate counters are kept to determine the correct cluster sizes. The magnetization distribution is computed by counting how often a fine or coarse configuration with a certain magnetization appears. This is done by using a map where the key is the magnetization and the value is the number of times the corresponding magnetization occurs. The approach for the coarse Boltzmann weights is similar, but it uses another map; the key is now a tuple (n_1, n_2) of the number of graphs that occur in the coarse configuration and the value is the number of times that the corresponding type of coarse configuration occurs.
The error $\Delta\chi$ of the susceptibility is estimated using the following relation

$$\Delta\chi = \frac{1}{\sqrt{N_{\text{meas}} - 1}}(\langle\chi^2\rangle - \langle\chi\rangle^2), \tag{29}$$

where N_{meas} is the number of measurements taken during the simulation. The error of the coarse Boltzmann weights and the magnetization distribution is given by the square root of their respective number of occurences.

3.2 Comparison of Exact and Monte Carlo Results

In this section, the results from the implemented cluster algorithm are presented and compared to exact values.

3.2.1 Critical Surface and Fixed Points

In Fig. 13, the exact values of the coarse susceptibility χ_c are displayed in the range of $W_1, W_2 \in [1.0, 2.3]$ for three pairs of the parameters B and C. The values of the

pseudo-critical couplings of both graphs and the couplings at the fixed point are listed in Table 1. Performing one iteration step causes the couplings to be close to the fixed point already. In order to gather more points, the iteration process can be started on additional points on the critical surface, not just where one of the couplings is set to 1. It can be set to 1.1, 1.2 and so on.

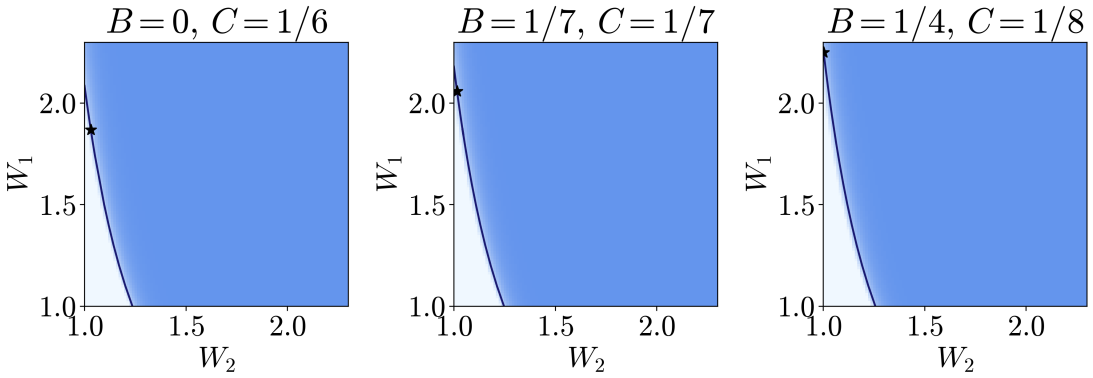

Figure 13: Coarse susceptibility for three pairs of B and C. The shading indicates χ_c, where the darker color corresponds to a higher value. The lines represent the critical surface and the star shows the fixed point.

The fact that the fixed point seems to move to smaller values of W_2 and bigger values of W_1 for larger values of B is also visible in the plot above. Adding more coordinate points to the critical surface as explained above results in the critical surface aligning nicely with χ_c.

3.2.2 Susceptibility

The values for both the fine and coarse susceptibility are listed in Table 2 below. Both the values obtained from the Monte Carlo simulation and the exact values are listed. The results from the simulation are taken from 10^6 measurements after 10^3 thermalisation sweeps.

Table 2: Fine and coarse susceptibility at the fixed point for three different pairs of B and C.

B	C	χ_f	$\chi_{f,\text{sim}}$	χ_c	$\chi_{c,\text{sim}}$
0	1/6	11.280	11.281(2)	3.8243	3.8241(3)
1/7	1/7	11.474	11.473(1)	3.8679	3.8678(2)
1/4	1/8	11.6017	11.6024(9)	3.8980	3.8982(2)

It can be seen that the simulated values are within the respective error ranges, but the error ranges are not big, indicating that the simplified error formula which ignores autocorrelations from Eq. (29) is sufficient for the error estimation with this cluster algorithm. Another detail is that the value of the susceptibility increases with the increase of the parameter B. This might be related to the fact that for bigger values of B, there is no more fixed point where both weights are > 1. It could be that the value of the susceptibility reaches its maximum value (12 for the fine and 4 for the coarse lattice) at the fixed point. This would break the clear separation between the broken and unbroken phase, since the fixed point is already in what is expected to be the broken phase.

3.2.3 Coarse Boltzmann Weights

The coarse Boltzmann weights given by Eq. (25) are compared to their respective values obtained from the implemented cluster algorithm. This is done at different points $w_i = (W_{1,i}, W_{2,i})$, which are labeled in Fig. 14 below. $w_{\mathrm{FP}} = (1.86816, 1.03087)$ indicates the fixed point and $w_1 = (1.05, 1.05)$ represents a point near the infinite temperature point. Then, $w_2 = (1.1, 1.1)$ and $w_3 = (1.5, 1.5)$ are near the critical surface in the unbroken and broken phase, respectively. The parameter pair $B = 0$ and $C = 1/6$ is chosen for this purpose, because in this case the fixed point has the biggest value of W_2. This ensures that not only the nearest-neighbour graph W_1 contributes to the fixed point Hamilton function, such that it can be shown that the computations work when introducing other interactions besides the nearest-neighbour interaction.

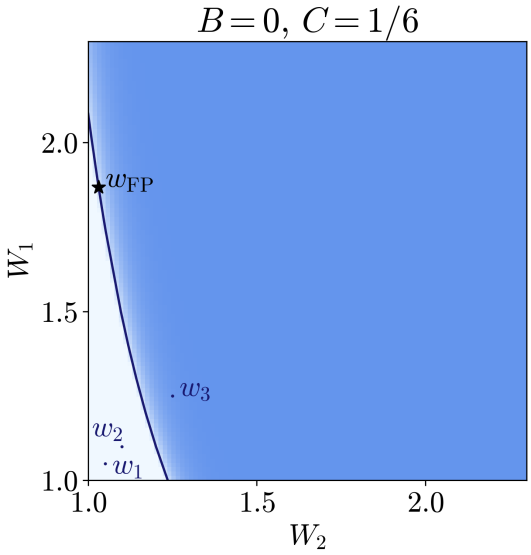

Figure 14: Labelled points to show for which weights the observables are computed.

Table 3 shows the coarse Boltzmann weights of the three independent types of coarse configurations, both from the simulation and the exact values. In most cases, the exact value lies within the error range of the respective simulated value. For the cases where the Boltzmann weight is small, the difference is likely due to the fact that it is difficult to get a precise and correct result with such a low count of that Boltzmann weight.

Table 3: Coarse Boltzmann weights evaluated at the chosen points.

$B = 0, C = 1/6$	w_{FP}	w_1	w_2	w_3
$(12, 24)$	0.945	0.2649	0.4904	0.9943
$(12, 24)_{\mathrm{sim}}$	0.945(1)	0.2646(5)	0.4911(7)	0.9945(10)
$(6, 0)$	0.0429	0.4596	0.3347	0.00455
$(6, 0)_{\mathrm{sim}}$	0.0429(2)	0.4598(6)	0.3341(5)	0.00439(7)
$(4, 8)$	0.0118	0.2755	0.1749	0.00110
$(4, 8)_{\mathrm{sim}}$	0.0115(1)	0.2756(5)	0.1747(4)	0.00109(3)

Fig. 15 shows the coarse Boltzmann weights of the three indpendent types of coarse configurations, obtained from the simulation at the different points w_i. The exact values are left out on purpose, since they would not be visible due to the small errorbars. The transition from the unbroken to the broken phase is visible in the points $w_{1,2,3}$, where the Boltzmann weights at the fixed point show a similar distribution to w_3. The two configuration types that are not completely aligned still appear sometimes, as it should be.

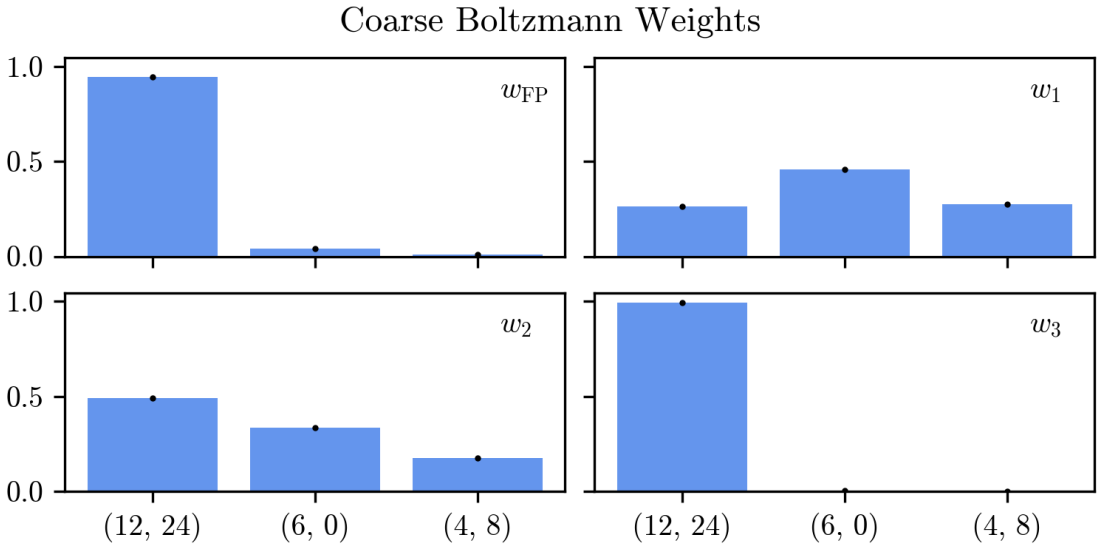

Figure 15: Coarse Boltzmann weights evaluated from the simulation. Note that the black dots indicate the error bars that are too small to be visible.

3.2.4 Magnetization Distribution

The coarse Boltzmann weights can be visualised as the distribution of the magnetization as well. This is done for the fine and coarse lattice in Fig. 16 below with exact values. The fine and coarse distribution show similar behaviour at each evaluated point. Again, the transition from the unbroken phase to the broken phase is visible. At w_1, the distribution resembles a Gaussian distribution, indicating the unordered alignment of the spins at high temperatures. Approaching the critical surface, it can be seen that at w_2 the peaks at ± 12 and ± 4 start to form and that the distribution loses its resemblance to a Gaussian distribution. As expected, in the broken phase the alignment of the spins dominates, with only very few instances where not all spins are aligned. Again, the distribution at the fixed point resembles the distribution in the broken phase.

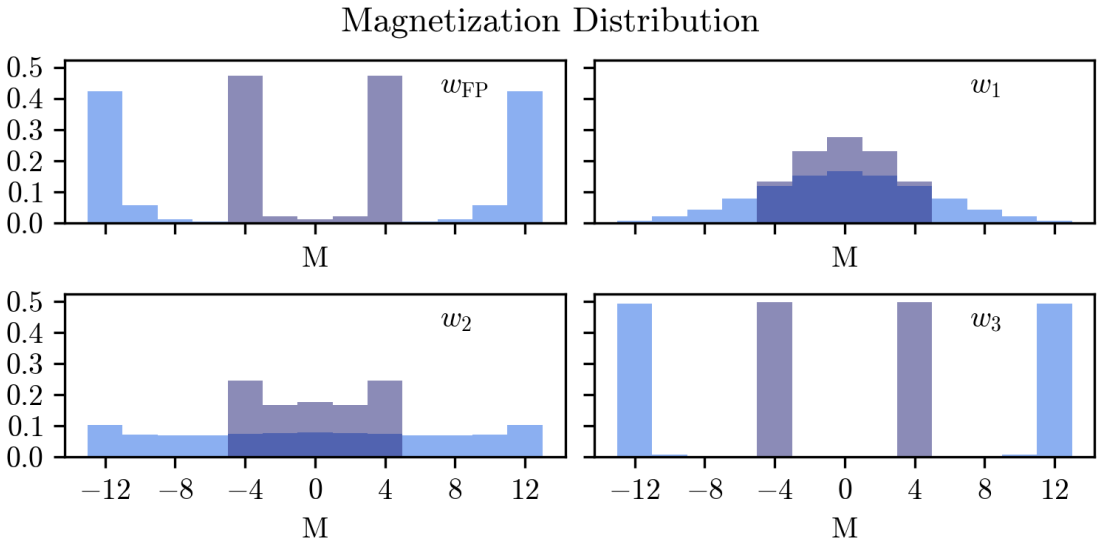

Figure 16: Magnetization distribution for the fine lattice (bright blue) and the coarse lattice (dark blue) at the chosen points.

4 Conclusions

The results from Section 3.2 show that it is indeed possible to simulate a cluster compatible RG transformation on a finite volume. This is ensured by checking the implementation in two parts. The first part is to check whether the susceptibility on the fine lattice agrees with its exact value. This verifies the correct implementation of the setting of the fine bonds of both graphs. This part is independent of the RG transformation, such that after verifying the susceptibility, the second part can be checked. The second part consists of the RG transformation, which is now checked with both the susceptibility and the Boltzmann weights of the coarse configurations. This guarantees the correct setting of the RG bonds, as well as the building and flipping of the clusters. Especially, the results show that the simulation also works correctly at the fixed point.
This enables the implementation of a multigrid with more than two layers, where the finest lattice can now be larger. In order to have an efficient implementation in such a case, slight modifications to the program would be necessary, which is possible to do. In the case of the $12 \to 4$ blocking, such optimisations are not necessary and would potentially result in slower calculations. An example for an optimisation that would be needed is the bond matrix. In this thesis, it is implemented as a 16×16 matrix, since there are 16 lattice sites on both lattices together. With a larger lattice however, it would be inefficient to have the bond matrix be that size. Most entries would remain zero throughout the entire simulation, because the graphs are chosen to be as local as possible. This could be solved by having a smaller bond matrix for every lattice site on the finest lattice that only includes its vicinity. For the coarse lattice, it would suffice to note the fine spin it connects to, since the cluster rules dictate that each coarse spin is connected to exactly one of its fine spins. It is not clear how many graphs would be needed, since for an exact computation the number of graphs would become unmanageable. One would have to restrict the graphs to be local. Having a larger finest lattice is expected to also reduce finite-size effects, which are prominent in such a small system as the one covered in this case. This should make it possible to simulate a truncated version of the fixed point Hamiltonian in larger systems.

References

[1] Nigel Goldenfeld. *Lectures on Phase Transitions and the Renormalization Group*. Addison-Wesley, 1992.

[2] Leo P. Kadanoff. "Scaling laws for ising models near T_c". In: *Physics Physique Fizika* 2 (6 June 1966), pp. 263–272. DOI: 10.1103/PhysicsPhysiqueFizika.2.263. URL: https://link.aps.org/doi/10.1103/PhysicsPhysiqueFizika.2.263.

[3] Kenneth G. Wilson. "Renormalization Group and Critical Phenomena. I. Renormalization Group and the Kadanoff Scaling Picture". In: *Phys. Rev. B* 4 (9 Nov. 1971), pp. 3174–3183. DOI: 10.1103/PhysRevB.4.3174. URL: https://link.aps.org/doi/10.1103/PhysRevB.4.3174.

[4] Kenneth G. Wilson. "The renormalization group: Critical phenomena and the Kondo problem". In: *Rev. Mod. Phys.* 47 (4 Oct. 1975), pp. 773–840. DOI: 10.1103/RevModPhys.47.773. URL: https://link.aps.org/doi/10.1103/RevModPhys.47.773.

[5] Uwe-Jens Wiese. "Statistical Thermodynamics". Lecture Notes, 2022, University of Bern.

[6] Uwe-Jens Wiese. "Renormalization Group of the Ising Model".

[7] Robert Swendsen and Jian-Sheng Wang. "Nonuniversal Critical Dynamics in Monte Carlo Simulation". In: *Physical Review Letters* 58 (Feb. 1987), pp. 86–88. DOI: 10.1103/PhysRevLett.58.86.

[8] Nico Scheidegger. "Modified RG-Theory: Bridging Renormalization and Computation on Finite-Lattices". MA thesis. University of Bern, 2025.

[9] Fabio Arz. "Modified Renormalization Group Transformations". MA thesis. University of Bern, 2025.

A Code for 12 → 4 Blocking

The goal of the implementation of a cluster compatible RG transformation (besides making sure that it works correctly) is to make the program easy to use, adapt, or expand. This is achieved by having the code for the simulation in a header file that can be included in a main file, where the simulation can be accessed from. Said header file is shown in Listing 1. The parameters of the simulation consist of W_1, W_2, p_1, p_2, B and C which are stored in a struct `Params`. These are the only things that are necessary to change outside of the header file to run the simulation. The structure of the header file consists of a class called `Lattice` and two inheriting classes `Simulation` and `Exact`. The `Lattice` class contains the things needed in both of the inheriting classes. In this case, this includes a struct that stores the index lists of the graphs and blocks and a struct that is used to store the calculated Boltzmann weights. The constructor of the `Lattice` class generates said index lists.
The `Simulation` class has a new struct that stores the two vectors containing the fine and coarse lattice. For testing purposes, there is a struct that is used in the generating of the initial configurations, giving the options to choose from different types of initial configurations. Finally, there is a struct to store observables such that the computation of the error can be done conveniently at the end of the simulation. The private functions in this class consist of the different parts of the cluster algorithm. They are called in a public function `sim()`, which is used to run the entire simulation with its given parameters. The observables are stores in public attributes, such that they can be accessed easily.
The `Exact` class contains the exact computations for the chosen system. It consists of functions that compute the observables and can be called all at once by calling the function `calc()`. Additionally, this class also contains the iteration process to find fixed points.
The code listed below can also be found on GitHub under `https://github.com/UmmagummaIV/Cluster-Compatible-RG-Transformation`.

Listing 1: C++ header file containing the code the simulation and exact computation.

```
#ifndef SIMULATION_HPP
#define SIMULATION_HPP

#include <iostream>
#include <fstream>
#include <cmath>
#include <vector>
#include <random>
#include <queue>
#include <ctime>
#include <chrono>
#include <bitset>
#include <map>
#include <string>
#include <numeric>
#include <windows.h>

#define Lf1 2   // width of fine lattice
#define Lf2 6   // length of fine lattice
#define Nf 12   // number of fine spins
#define Lc1 2   // width of coarse lattice
#define Lc2 2   // length of coarse lattice
#define Nc 4    // number of spins on coarse lattice
#define Nt 16   // total number of spins on both lattices
#define INDEP 3 // number of independent coarse configs
```

```
// parameters of the RG transformation
struct Params
{
    double W01; // coupling of nearest-neighbour graph
    double W02; // coupling of 4-spin graph
    double p01; // probability to activate nearest-neighbour bond
    double p02; // probability to activate 4-spin bonds
    double B;   // probability to activate bond with central spin in block
    double C;   // probability to activate bond with 1 peripheral spin in block
};

////////////////////////////////////////////////////////////////////////////
///////////////////////////// Initialise RNGesus ///////////////////////////
////////////////////////////////////////////////////////////////////////////

std::random_device rd;
int seed = rd();        // stores the seed for debugging purposes
std::mt19937 gen(seed); // mersenne twister RNG, could be replaced with ranlux
std::uniform_int_distribution<int> randbool(0, 1);       // returns a boolean
std::uniform_real_distribution<double> random(0.0, 1.0); // returns double [0,1)
std::uniform_int_distribution<int> randedge(1, 6);       // returns an int [1,6]
std::uniform_int_distribution<int> randblock(0, 6);      // returns an int [0,6]

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

// contains attributes needed in both Simulation and Exact class
class Lattice
{
    public:
        // index lists of both graphs and the blocks
        struct All_ind
        {
            std::vector<int> W01;    // indes list of nearest-neighbour graph
            std::vector<int> W02;    // index list of 4-spin graph
            std::vector<int> blocks; // index list of blocks
        };
        // contains computed Boltzmann weights
        struct BWeight
        {
            std::tuple<int,int> graphs; // number of graphs present in config
            double weight;              // Boltzmann weight
            double err;                 // error of computed Boltzmann weight
        };
        // generate index lists for fine and coarse lattice
        Lattice ();

    protected:
        All_ind ind_f; // indices of all graphs on the fine lattice
        All_ind ind_c; // indices of all graphs on the coarse lattice

    private:
        // generate index list of nearest-neighbour graph for a given lattice
        void get_W01_ind(std::vector<int> &ind, int L1, int L2);
        // generate index list of 4-spin graph for a given lattice
        void get_W02_ind(std::vector<int> &ind, int L1, int L2);
        // generate index list of blocks on the fine lattice
        void get_spins_per_block(std::vector<int> &blocks);
};

// contains the cluster algorithm
class Simulation : public Lattice
{
    public:
        // contains spins of both lattices
        struct Spins
        {
```

```
            std::vector<bool> f; // fine lattice
            std::vector<bool> c; // coarse lattice
        };
        // used to generate different initial configs (debugging purposes)
        enum Condition
        {
            COLDSTART,
            CENTER,
            EDGE,
            RANDOM
        };
        // contains one observable
        struct Obs
        {
            double obs; // value of the observable
            double sq;  // squared value of the observable
            double err; // error of the observable
        };
        // initialise fine and coarse lattice
        Simulation();
        Spins spins;                    // fine and coarse spins
        Obs sus_f, sus_c;               // susceptibility
        Params p;                       // parameters used in simulation
        std::vector<BWeight> bweights; // coarse Boltzmann weights
        // perform the simulation
        void sim();

    private:
        // initialise fine and coarse lattice with given initial config
        void initialise(Spins &spins, std::vector<int> &blocks,Condition start);
        // count nearest-neighbour instances on a given lattice
        int get_W01_count(int N,std::vector<bool> &spins,std::vector<int> &ind);
        // count 4-spin instances on a given lattice
        int get_W02_count(int N,std::vector<bool> &spins,std::vector<int> &ind);
        // count instances of both graphs on a given lattice
        std::tuple<int,int>
            count_graphs(int N, std::vector<bool>&spins, All_ind all_ind);
        // set bonds on the fine lattice including RG bonds
        void set_bonds(Params p, Spins &spins, All_ind ind,
            std::vector<std::vector<bool>> &bonds);
        // build the clusters and compute susceptibility
        void build_clusters(std::vector<std::vector<bool>> &bonds,
            std::vector<int> &clusters);
        // flip the clusters with 50% probability
        void flip_clusters(Spins &spins, std::vector<int> &clusters);
};

// contains exact computations
class Exact : public Lattice
{
    public:
        // generate number of fine and coarse configs
        Exact();
        Params p;                                // parameters of computation
        double Z_f, Z_c;                         // partition function
        double chi_f, chi_c;                     // susceptibility
        std::vector<double> dist_f, dist_c;      // magnetic distribution
        std::vector<BWeight> bweights;           // coarse Boltzmann weights
        std::vector<std::vector<double>> W_iter; // weights during iteration
        // generate histograms for each independent coarse config
        void make_hist();
        // iteration of blocking and insertion steps
        void iteration(int iterations);
        // compute coarse Boltzmann weights
        void calc_bw();
        // compute coarse susceptibility
        void sus_f();
        // compute fine susceptibility
        void sus_c();
```

```
        // compute fine magnetic distribution
        void mag_dist_f();
        // compute coarse magnetic distribution
        void mag_dist_c();
        // perform computation of all observables
        void calc();

    private:
        // count nearest-neighbour instances on a given lattice
        int get_W01_count(int N, std::string config, std::vector<int> &ind);
        // count 4-spin instances on a given lattice
        int get_W02_count(int N, std::string config, std::vector<int> &ind);
        // count instances of both graphs on a given lattice
        std::tuple<int,int> count_graphs
            (int N, std::string config, All_ind all_ind);
        // compute magnetisation for a given lattice and config
        int mag(int N, std::string config);
        // compute blocking kernel for a given fine and coarse config
        double kernel_cluster(std::string config_f,
            std::string config_c, std::vector<int> &blocks);
        // compute Boltzmann weights of independent coarse configs
        double boltzmann_weight(std::vector<std::tuple<int,int,double>>
            hist_indep);
        int numf; // number of fine configurations
        int numc; // number of coarse configurations
        // number of unique combinations of (mag,n1,n2) on coarse lattice
        std::map<std::tuple<int,int,int>, int> map_obs_c;
        // order of independent coarse configs
        std::vector<std::tuple<int,int>> powers_help;
        // number of unique combinations of (mag,n1,n2) on fine lattice
        std::map<std::tuple<int,int,int>, int> map_obs_f;
        // histograms of all independent coarse configs
        std::vector<std::vector<std::tuple<int,int,double>>> hist;
        // inverse matrix used in insertion step
        std::vector<std::vector<double>> mat_inv;
};

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

Lattice::Lattice()
{
    ind_f.W01 = std::vector<int> (3*Nf, 0);
    ind_f.W02 = std::vector<int> (9*Nf, 0);
    ind_f.blocks = std::vector<int> (7*Nc,0);
    get_W01_ind(ind_f.W01, Lf1, Lf2);
    get_W02_ind(ind_f.W02, Lf1, Lf2);
    get_spins_per_block(ind_f.blocks);

    ind_c.W01 = std::vector<int> (3*Nc, 0);
    ind_c.W02 = std::vector<int> (9*Nc, 0);
    ind_c.blocks = std::vector<int> (7*Nc,0);
    get_W01_ind(ind_c.W01, Lc1, Lc2);
    get_W02_ind(ind_c.W02, Lc1, Lc2);
    get_spins_per_block(ind_c.blocks);
}

void Lattice::get_W01_ind(std::vector<int> &ind, int L1, int L2)
{
    for (int i=0; i<L1; ++i)
    {
        for (int j=0; j<L2; ++j)
        {
            ind[3*(i*L2+j)+0] = ((i+1)%L1)*L2 + (j+2)%L2;
            ind[3*(i*L2+j)+1] = i*L2 + (j+1)%L2;
            ind[3*(i*L2+j)+2] = ((i+1)%L1)*L2 + ((j-1)%L2+L2)%L2;
        }
    }
```

```
}

void Lattice::get_W02_ind(std::vector<int> &ind, int L1, int L2)
{
    for (int i=0; i<L1; ++i)
    {
        for (int j=0; j<L2; ++j)
        {
            ind[9*(i*L2+j)+0] = ((i+1)%L1)*L2 + (j+2)%L2;
            ind[9*(i*L2+j)+1] = ((i+1)%L1)*L2 + (j+3)%L2;
            ind[9*(i*L2+j)+2] = i*L2 + (j+1)%L2;
            ind[9*(i*L2+j)+3] = i*L2 + (j+1)%L2;
            ind[9*(i*L2+j)+4] = ((i+1)%L1)*L2 + j;
            ind[9*(i*L2+j)+5] = ((i+1)%L1)*L2 + ((j-1)%L2+L2)%L2;
            ind[9*(i*L2+j)+6] = ((i+1)%L1)*L2 + (j+2)%L2;
            ind[9*(i*L2+j)+7] = i*L2 + (j+1)%L2;
            ind[9*(i*L2+j)+8] = ((i+1)%L1)*L2 + ((j-1)%L2+L2)%L2;
        }
    }
}

void Lattice::get_spins_per_block(std::vector<int> &blocks)
{
    for (int i=0; i<Lc1; ++i)
    {
        for (int j=0; j<Lc2; ++j)
        {
            blocks[7*(i*Lc2+j)+0] = i*Lf2 + j*3;
            blocks[7*(i*Lc2+j)+1] = ((i+1)%Lf1)*Lf2 + (j*3+2)%Lf2;
            blocks[7*(i*Lc2+j)+2] = i*Lf2 + (j*3+1)%Lf2;
            blocks[7*(i*Lc2+j)+3] = ((i+1)%Lf1)*Lf2 + ((j*3-1)%Lf2+Lf2)%Lf2;
            blocks[7*(i*Lc2+j)+4] = ((i+1)%Lf1)*Lf2 + ((j*3-2)%Lf2+Lf2)%Lf2;
            blocks[7*(i*Lc2+j)+5] = i*Lf2 + ((j*3-1)%Lf2+Lf2)%Lf2;
            blocks[7*(i*Lc2+j)+6] = ((i+1)%Lf1)*Lf2 + (j*3+1)%Lf2;
        }
    }
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

Simulation::Simulation()
{
    spins.f = std::vector<bool>(Nf, 0);
    spins.c = std::vector<bool>(Nc, 0);
}

void Simulation::sim()
{
    initialise(spins, ind_f.blocks, COLDSTART);
    // generate the map for the types of coarse configs
    std::map<std::tuple<int,int>, int> map_configs;
    map_configs.insert(std::pair<std::tuple<int,int>,int>
            (std::tuple<int,int>(12,24),0));
    map_configs.insert(std::pair<std::tuple<int,int>,int>
            (std::tuple<int,int>(6,0),0));
    map_configs.insert(std::pair<std::tuple<int,int>,int>
            (std::tuple<int,int>(4,8),0));

    // initialise susceptibility to be measured
    sus_f.obs = 0;
    sus_f.sq = 0;
    sus_f.err = 0;
    sus_c.obs = 0;
    sus_c.sq = 0;
    sus_c.err = 0;

    // initialise bond matrix and cluster vector
```

```
    std::vector<std::vector<bool>> bonds(Nt, std::vector<bool>(Nt,0));
    std::vector<int> clusters(Nt,-1);
    // thermalisation
    for (int i=0; i<1000; ++i)
    {
        bonds = std::vector<std::vector<bool>>(Nt, std::vector<bool>(Nt,0));
        set_bonds(p, spins, ind_f, bonds);
        build_clusters(bonds, clusters);
        flip_clusters(spins, clusters);
    }

    // start actual measurements (and count the coarse configs here)
    std::tuple<int,int> powers_c;
    int iterations = 1000000;
    int counter = 0;
    sus_f.obs = 0;
    sus_f.sq = 0;
    sus_c.obs = 0;
    sus_c.sq = 0;
    for (int i=0; i<iterations; ++i)
    {
        bonds = std::vector<std::vector<bool>>(Nt, std::vector<bool>(Nt,0));
        set_bonds(p, spins, ind_f, bonds);
        build_clusters(bonds, clusters);
        flip_clusters(spins, clusters);
        powers_c = count_graphs(Nc, spins.c, ind_c);
        ++map_configs[powers_c];
    }

    // computation of susceptibility
    sus_f.obs = double(sus_f.obs)/double(iterations);
    sus_f.sq = double(sus_f.sq)/double(iterations);
    sus_f.err = (sus_f.sq - sus_f.obs*sus_f.obs)/sqrt(iterations-1);
    sus_c.obs = double(sus_c.obs)/double(iterations);
    sus_c.sq = double(sus_c.sq)/double(iterations);
    sus_c.err = (sus_c.sq - sus_c.obs*sus_c.obs)/sqrt(iterations-1);

    // computation of coarse boltzmann weights
    bweights.clear();
    std::map<std::tuple<int,int>, int>::reverse_iterator rit;
    for (rit = map_configs.rbegin(); rit != map_configs.rend(); ++rit)
    {
        BWeight hihi;
        hihi.graphs = rit->first;
        hihi.weight = rit->second/float(iterations);
        hihi.err = sqrt(rit->second)/float(iterations);
        bweights.push_back(hihi);
    }
}

void Simulation::initialise(Spins&spins,std::vector<int>&blocks,Condition start)
{
    switch (start)
    {
        case 0: // COLDSTART
            for (int i=0; i<Nf; ++i) {spins.f[i] = 1;}
            for (int i=0; i<Nc; ++i) {spins.c[i] = 1;}
            break;
        case 1: // CENTER
            for (int i=0; i<Nf; ++i) {spins.f[i] = randbool(gen);}
            for (int i=0; i<Nc; ++i)
                {spins.c[i] = spins.f[blocks[7*i]];}
            break;
        case 2: // EDGE
            for (int i=0; i<Nf; ++i) {spins.f[i] = randbool(gen);}
            for (int i=0; i<Nc; ++i)
                {spins.c[i] = spins.f[blocks[7*i+randedge(gen)]];}
            break;
        case 3: // RANDOM
```

```
            for (int i=0; i<Nf; ++i) {spins.f[i] = randbool(gen);}
            for (int i=0; i<Nc; ++i)
                {spins.c[i] = spins.f[blocks[7*i+randblock(gen)]];}
            break;
    }
}

int Simulation::get_W01_count
    (int N, std::vector<bool> &spins, std::vector<int> &ind)
{
    int count = 0;
    for (int i=0; i<N; ++i)
    {
        if (spins[i] == spins[ind[3*i+0]]) {++count;}
        if (spins[i] == spins[ind[3*i+1]]) {++count;}
        if (spins[i] == spins[ind[3*i+2]]) {++count;}
    }
    return count;
}

int Simulation::get_W02_count
    (int N, std::vector<bool> &spins, std::vector<int> &ind)
{
    int count = 0;
    for (int i=0; i<N; ++i)
    {
        for (int j=0; j<3; ++j)
        {
            if (spins[i] == spins[ind[9*i+0+3*j]] &&
                spins[ind[9*i+1+3*j]] == spins[ind[9*i+2+3*j]]) {++count;}
            if (spins[i] == spins[ind[9*i+2+3*j]] &&
                spins[ind[9*i+0+3*j]] == spins[ind[9*i+1+3*j]]) {++count;}
        }
    }
    return count;
}

std::tuple<int,int>
    Simulation::count_graphs(int N,std::vector<bool> &spins,All_ind all_ind)
{
    int W01_count = get_W01_count(N, spins, all_ind.W01);
    int W02_count = get_W02_count(N, spins, all_ind.W02);
    return std::tuple<int, int>(W01_count, W02_count);
}

void Simulation::set_bonds(Params p, Spins &spins, All_ind ind,
    std::vector<std::vector<bool>> &bonds)
{
    int n;
    for (int i=0; i<Nf; ++i)
    {
        // set W01 bonds
        n = ind.W01.size()/Nf;
        for (int j=0; j<n; ++j)
        {
            if ((spins.f[i] == spins.f[ind.W01[n*i+j]]) && random(gen) < p.p01)
            {
                bonds[i][ind.W01[n*i+j]] = 1;
                bonds[ind.W01[n*i+j]][i] = 1;
            }
        }

        // set W02 bonds
        n = ind.W02.size()/Nf;
        for (int j=0; j<3; ++j)
        {
            if (spins.f[i] == spins.f[ind.W02[n*i+3*j+0]] &&
                spins.f[ind.W02[n*i+3*j+1]] == spins.f[ind.W02[n*i+3*j+2]] &&
                random(gen) < p.p02)
```

```
            {
                bonds[i][ind.W02[n*i+3*j+0]] = 1;
                bonds[ind.W02[n*i+3*j+0]][i] = 1;

                bonds[ind.W02[n*i+3*j+1]][ind.W02[n*i+3*j+2]] = 1;
                bonds[ind.W02[n*i+3*j+2]][ind.W02[n*i+3*j+1]] = 1;
            }
            if (spins.f[i] == spins.f[ind.W02[n*i+3*j+2]] &&
                spins.f[ind.W02[n*i+3*j+0]] == spins.f[ind.W02[n*i+3*j+1]] &&
                random(gen) < p.p02)
            {
                bonds[i][ind.W02[n*i+3*j+2]] = 1;
                bonds[ind.W02[n*i+3*j+2]][i] = 1;

                bonds[ind.W02[n*i+3*j+1]][ind.W02[n*i+3*j+0]] = 1;
                bonds[ind.W02[n*i+3*j+0]][ind.W02[n*i+3*j+1]] = 1;
            }
        }
    }

    // set RG bonds
    for (int i=0; i<Nc; ++i)
    {
        double prob = random(gen);
        double total = 0;
        bool theresabee = 0;
        std::queue<int> q;
        if (spins.c[i] == spins.f[ind.blocks[7*i]])
        {
            theresabee = 1;
            total += p.B;
            q.push(ind.blocks[7*i]);
        }
        for (int j=1; j<7; ++j)
        {
            if (spins.c[i] == spins.f[ind.blocks[7*i+j]])
            {
                total += p.C;
                q.push(ind.blocks[7*i+j]);
            }
        }

        if (theresabee)
        {
            if (prob < p.B/total)
            {
                bonds[Nf+i][q.front()] = 1;
                bonds[q.front()][Nf+i] = 1;
                continue;
            }
            q.pop();
            prob -= p.B/total;
        }

        while (prob > p.C/total)
        {
            prob -= p.C/total;
            q.pop();
        }
        bonds[Nf+i][q.front()] = 1;
        bonds[q.front()][Nf+i] = 1;
    }
}

void Simulation::build_clusters(std::vector<std::vector<bool>> &bonds,
    std::vector<int> &clusters)
{
    std::vector<bool> visited(Nt, 0);
    int cluster = 0;
```

```
    int counter_f = 1;
    int counter_c = 0;
    double sus_temp_f = 0.0;
    double sus_temp_c = 0.0;
    for (int i=0; i<Nt; ++i)
    {
        if (!visited[i])
        {
            std::queue<int> q;
            q.push(i);
            visited[i] = true;
            clusters[i] = cluster;
            while (!q.empty())
            {
                int x = q.front();
                q.pop();
                for (int j=0; j<Nt; ++j)
                {
                    if (!visited[j] && bonds[x][j])
                    {
                        q.push(j);
                        visited[j] = true;
                        clusters[j] = cluster;
                        if (j<Nf) {++counter_f;}
                        else {++counter_c;}
                    }
                }
            }
            sus_temp_f += counter_f*counter_f;
            sus_temp_c += counter_c*counter_c;
            counter_f = 1;
            counter_c = 0;
            ++cluster;
        }
    }
    sus_f.obs += sus_temp_f/Nf;
    sus_f.sq += (sus_temp_f/Nf)*(sus_temp_f/Nf);
    sus_c.obs += sus_temp_c/Nc;
    sus_c.sq += (sus_temp_c/Nc)*(sus_temp_c/Nc);
}

void Simulation::flip_clusters(Spins &spins, std::vector<int> &clusters)
{
    std::vector<int> cluster_spin(Nt, -1);
    for (int i=0; i<Nt; ++i)
    {
        if (cluster_spin[clusters[i]] == -1)
        {
            cluster_spin[clusters[i]] = randbool(gen);
        }
        if (i<Nf) {spins.f[i] = cluster_spin[clusters[i]];}
        else {spins.c[i-Nf] = cluster_spin[clusters[i]];}
    }
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

Exact::Exact()
{
    numf = std::pow(2,Nf);
    numc = std::pow(2,Nc);
}

void Exact::make_hist()
{
    // count graphs and filter coarse configs
    std::map<std::tuple<int,int>, std::string> map_configs;
```

```
    std::vector<std::string> indep_c;
    map_obs_c.clear();
    powers_help.clear();

    for (int i=0; i<numc; ++i)
    {
        std::string config_c = std::bitset<Nc>(i).to_string();
        std::tuple<int,int> powers_c = count_graphs(Nc, config_c, ind_c);
        int mag_c = mag(Nc, config_c);
        std::tuple<int,int,int>
            comb_c(std::tuple_cat(std::tuple<int>(mag_c), powers_c));

        // stores the combinations of graphs of magnetisations
        if (map_obs_c.count(comb_c) == 0)
        {
            map_obs_c.insert(std::pair<std::tuple<int,int,int>,int>(comb_c,1));
        }
        else {++map_obs_c[comb_c];}

        // map for histogram with kernel values
        if (map_configs.count(powers_c) == 0)
        {
            map_configs.insert(std::pair<std::tuple<int,int>, std::string>
                               (powers_c, config_c));
            indep_c.push_back(config_c);
            powers_help.push_back(powers_c);
        }
    }

    // write the inverse matrix for the iteration process
    mat_inv = {{0.08333333,  0.1666667, -0.25},
               {0.02083333, -0.0833333,  0.0625},
               {-0.5,        0,          1.5}};

    map_obs_f.clear();
    std::vector<std::vector<std::tuple<int,int,double>>> aua(3);
    hist = aua;
    // loop over all fine configurations
    for(int i=0; i<numf; ++i)
    {
        // variables for the observables
        std::string config_f = std::bitset<Nf>(i).to_string();
        std::tuple<int,int> powers_f = count_graphs(Nf, config_f, ind_f);
        int mag_f = mag(Nf, config_f);
        std::tuple<int,int,int>
            comb_f(std::tuple_cat(std::tuple<int>(mag_f), powers_f));

        // stores the combinations of graphs and magnetisations
        if (map_obs_f.count(comb_f) == 0)
        {
            map_obs_f.insert(std::pair<std::tuple<int,int,int>,int>(comb_f,1));
        }
        else {++map_obs_f[comb_f];}

        // stores the graphs and kernels
        for (int j=0; j<INDEP; ++j)
        {
            double kernel = kernel_cluster(config_f,indep_c[j],ind_c.blocks);
            if (kernel>0)
            {
                hist[j].push_back(std::tuple_cat(powers_f,
                                                 std::make_tuple(kernel)));
            }
        }
    }
}

void Exact::iteration(int iterations)
{
```

```
    std::vector<double> weights (INDEP, 0);
    std::vector<double> solution (INDEP, 0.0);
    W_iter.clear();
    W_iter.push_back(std::vector<double>({p.W01,p.W02}));
    for (int k=0; k<iterations; ++k)
    {
        // compute all boltzmann factors
        for (int i=0; i<INDEP; ++i) {weights[i] = boltzmann_weight(hist[i]);}

        // solve the system of linear equations
        solution = std::vector<double> (INDEP, 0.0);
        solution[INDEP-1] = 1.0;
        for (int i=0; i<INDEP-1; ++i)
        {
            for (int j=0; j<INDEP; ++j)
            {
                solution[i] += mat_inv[i][j]*log(weights[j]);
            }
            solution[i] = exp(solution[i]);
        }
        p.W01 = solution[0];
        p.W02 = solution[1];
        W_iter.push_back(std::vector<double>({solution[0],solution[1]}));

        if (1==1)
        {
            // print graph weights after each iteration
            std::cout<<"------ new graph weights -----"<<std::endl;
            for (int i=0; i<INDEP; ++i) {std::cout << solution[i] << std::endl;}
            std::cout << "------------------------------" << std::endl;
        }
    }
}

void Exact::calc_bw()
{
    // compute all boltzmann factors and use mat_help
    bweights.clear();
    std::vector<double> bw (INDEP, 0.0);
    std::vector<int> mult({2, 8, 6});
    for (int i=0; i<INDEP; ++i) {bw[i] = mult[i]*boltzmann_weight(hist[i]);}
    for (int i=0; i<INDEP; ++i)
    {
        BWeight hihi;
        hihi.graphs = powers_help[i];
        hihi.weight = bw[i]/std::reduce(bw.begin(), bw.end());
        hihi.err = 0;
        bweights.push_back(hihi);
    }
}

void Exact::sus_f()
{
    double mag_sq = 0.0;
    Z_f = 0.0;
    chi_f = 0.0;
    std::map<std::tuple<int,int,int>, int>::iterator it;
    for (it = map_obs_f.begin(); it != map_obs_f.end(); ++it)
    {
        Z_f += (it->second)*std::pow(p.W01,std::get<1>(it->first))
                           *std::pow(p.W02,std::get<2>(it->first));
        mag_sq += (it->second)*std::get<0>(it->first)*std::get<0>(it->first)
                               *std::pow(p.W01,std::get<1>(it->first))
                               *std::pow(p.W02,std::get<2>(it->first));
    }
    chi_f = mag_sq/(Z_f*Nf);
}

void Exact::sus_c()
```

```
{
    Z_c = 0.0;
    double Z_temp = 0.0;
    std::vector<int> mult({2, 8, 6});
    for (int i=0; i<hist.size(); ++i)
    {
        for (int j=0; j<hist[i].size(); ++j)
        {
            int pow0 = std::get<0>(hist[i][j]);
            int pow1 = std::get<1>(hist[i][j]);
            double kernel = std::get<2>(hist[i][j]);
            Z_temp += kernel*std::pow(p.W01,pow0)*std::pow(p.W02,pow1);
        }
        Z_c += mult[i]*Z_temp;
        Z_temp = 0.0;
    }

    chi_c = 0.0;
    double sus_help = 0.0;
    std::map<std::tuple<int,int,int>, int>::iterator it;
    for (it = map_obs_c.begin(); it != map_obs_c.end(); ++it)
    {
        std::tuple<int,int> hehe
        = std::make_tuple(std::get<1>(it->first),std::get<2>(it->first));
        for (int i=0; i<powers_help.size(); ++i)
        {
            if (hehe == powers_help[i])
            {
                for (int j=0; j<hist[i].size(); ++j)
                {
                    int pow0 = std::get<0>(hist[i][j]);
                    int pow1 = std::get<1>(hist[i][j]);
                    double kernel = std::get<2>(hist[i][j]);
                    int count = it->second;
                    int mag = std::get<0>(it->first);
                    sus_help += count*kernel*mag*mag*std::pow(p.W01,pow0)
                    *std::pow(p.W02,pow1);
                }
            }
        }
    }

    chi_c = sus_help/(Z_c*Nc);
}

void Exact::mag_dist_f()
{
    dist_f = std::vector<double>(Nf+1, 0.0);
    Z_f = 0.0;
    std::map<std::tuple<int,int,int>, int>::iterator it;
    for (it = map_obs_f.begin(); it != map_obs_f.end(); ++it)
    {
        int mag = std::get<0>(it->first);
        double temp = (it->second)*std::pow(p.W01,std::get<1>(it->first))
                                   *std::pow(p.W02,std::get<2>(it->first));
        Z_f += temp;
        dist_f[(mag+Nf)/2] += temp;
    }
    for (int i=0; i<dist_f.size(); ++i)
    {
        dist_f[i] = dist_f[i]/Z_f;
    }
}

void Exact::mag_dist_c()
{
    Z_c = 0.0;
    double Z_temp = 0.0;
    std::vector<int> mult({2, 8, 6});
```

```
    for (int i=0; i<hist.size(); ++i)
    {
        for (int j=0; j<hist[i].size(); ++j)
        {
            int pow0 = std::get<0>(hist[i][j]);
            int pow1 = std::get<1>(hist[i][j]);
            double kernel = std::get<2>(hist[i][j]);
            Z_temp += kernel*std::pow(p.W01,pow0)*std::pow(p.W02,pow1);
        }
        Z_c += mult[i]*Z_temp;
        Z_temp = 0.0;
    }

    dist_c = std::vector<double>(Nc+1, 0.0);
    std::map<std::tuple<int,int,int>, int>::iterator it;
    for (it = map_obs_c.begin(); it != map_obs_c.end(); ++it)
    {
        std::tuple<int,int> hehe
        = std::make_tuple(std::get<1>(it->first),std::get<2>(it->first));
        for (int i=0; i<powers_help.size(); ++i)
        {
            if (hehe == powers_help[i])
            {
                for (int j=0; j<hist[i].size(); ++j)
                {
                    int pow0 = std::get<0>(hist[i][j]);
                    int pow1 = std::get<1>(hist[i][j]);
                    double kernel = std::get<2>(hist[i][j]);
                    int count = it->second;
                    int mag = std::get<0>(it->first);
                    dist_c[(mag+Nc)/2] += count*kernel*std::pow(p.W01,pow0)
                                                      *std::pow(p.W02,pow1);
                }
            }
        }
    }
    for (int i=0; i<dist_c.size(); ++i)
    {
        dist_c[i] = dist_c[i]/Z_c;
    }
}

void Exact::calc()
{
    make_hist();
    calc_bw();
    sus_c();
    sus_f();
}

int Exact::get_W01_count(int N, std::string config, std::vector<int> &ind)
{
    int count = 0;
    for (int i=0; i<N; ++i)
    {
        if (config[i] == config[ind[3*i+0]]) {++count;}
        if (config[i] == config[ind[3*i+1]]) {++count;}
        if (config[i] == config[ind[3*i+2]]) {++count;}
    }
    return count;
}

int Exact::get_W02_count(int N, std::string config, std::vector<int> &ind)
{
    int count = 0;
    for (int i=0; i<N; ++i)
    {
        if (config[i] == config[ind[9*i+0]] &&
            config[ind[9*i+1]] == config[ind[9*i+2]]) {++count;}
```

```
        if (config[i] == config[ind[9*i+2]] &&
            config[ind[9*i+0]] == config[ind[9*i+1]]) {++count;}

        if (config[i] == config[ind[9*i+3]] &&
            config[ind[9*i+4]] == config[ind[9*i+5]]) {++count;}
        if (config[i] == config[ind[9*i+5]] &&
            config[ind[9*i+3]] == config[ind[9*i+4]]) {++count;}

        if (config[i] == config[ind[9*i+6]] &&
            config[ind[9*i+7]] == config[ind[9*i+8]]) {++count;}
        if (config[i] == config[ind[9*i+8]] &&
            config[ind[9*i+6]] == config[ind[9*i+7]]) {++count;}
    }
    return count;
}

std::tuple<int, int> Exact::count_graphs
    (int N, std::string config, All_ind all_ind)
{
    int WO1_count = get_WO1_count(N, config, all_ind.WO1);
    int WO2_count = get_WO2_count(N, config, all_ind.WO2);
    return std::tuple<int, int>(WO1_count, WO2_count);
}

int Exact::mag(int N, std::string config)
{
    int sum = 0;
    for (int i=0; i<N; ++i) {if(config[i]-'0'){++sum;}}
    return 2*sum-N;
}

double Exact::kernel_cluster(std::string config_f,
    std::string config_c, std::vector<int> &blocks)
{
    double kernel = 1.0;
    double temp = 0;
    for (int i=0; i<Nc; ++i)
    {
        if (config_c[i] == config_f[blocks[7*i]]) {temp += p.B;}
        for (int j=1; j<7; ++j)
        {
            if (config_c[i] == config_f[blocks[7*i+j]]) {temp += p.C;}
        }
        kernel *= temp;
        temp = 0;
    }
    return kernel;
}

double Exact::boltzmann_weight(std::vector<std::tuple<int,int,double>>
    hist_indep)
{
    double bw = 0.0;
    for (int i=0; i<hist_indep.size(); ++i)
    {
        int pow0 = std::get<0>(hist_indep[i]);
        int pow1 = std::get<1>(hist_indep[i]);
        double kernel = std::get<2>(hist_indep[i]);
        bw += kernel*std::pow(p.WO1,pow0)*std::pow(p.WO2,pow1);
    }
    return bw;
}

#endif
```

Erklärung

gemäss Art. 30 RSL Phil.-nat. 18

Name/Vorname: Schio, Meryl

Matrikelnummer: 20-118-006

Studiengang: Physik

Bachelor ☐ Master ☑ Dissertation ☐

Titel der Arbeit: Implementation of a Cluster Compatible Renormalization Group Transformation for the Ising Model in a Periodic Volume

LeiterIn der Arbeit: Prof. Dr. Uwe-Jens Wiese

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.
Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Köniz, 19.05.2025

Ort/Datum

Unterschrift