MSc thesis

External magnetic field in 2-dimensional O(3) model

Gábor Oszkár Dénes

ELTE TTK, Physics MSc

2021. May

Supervisor:
Dániel Nógrádi, Department of Theoretical Physics

STATEMENT

Name: Gábor Oszkár Dénes
Neptun ID: T0QAI2

ELTE Faculty of Science: Physics MSc

specialization: Particle Physics

Title of diploma work:

External magnetic field in 2-dimensional O(3) model

As the author of the diploma work I declare, with disciplinary responsibility that my thesis is my own intellectual product and the result of my own work. Furthermore I declare that I have consistently applied the standard rules of references and citations.

I acknowledge that the following cases are considered plagiarism:
– using a literal quotation without quotation mark and adding citation;
– referencing content without citing the source;
– representing another person's published thoughts as my own thoughts.

Furthermore, I declare that the printed and electronical versions of the submitted diploma work are textually and contextually identical.

Budapest, 2021. May 04.

Dénes Gábor

Signature of Student

Contents

1 Introduction

1.1 Nonlinear sigma model

The nonlinear sigma model – which was developed, and first used extensively by Murray Gell-Mann and his colleagues around 1960 [1], [2] – is used as an example for many different things in quantum field theory. Among the examples are the spontaneous symmetry breaking and a theory with asymptotically free properties. Because of these, this model is a good candidate to examine, particularly because some conclusions might be applicable in theories like the quantum chromodynamics. In this thesis I examine the β function of a modified version of the nonlinear sigma model.

The 2 dimensional $\mathrm{O}(N)$ nonlinear sigma model contains bosonic fields, with N component unit vectors. By definition it has an $\mathrm{O}(N)$ symmetry.

$$\mathcal{L}(\phi, \partial_\mu \phi) = \frac{1}{2g^2}\ \partial_\mu \phi_i\ \partial^\mu \phi_i\ , \tag{1.1.1}$$

$$\text{where } \phi_i \phi_i = 1\ . \tag{1.1.2}$$

To calculate the β function, first the Green functions (n-point functions) are determined from the Feynman rules. These can be calculated in the perturbative expansion of the Lagrangian in order of the field components, which are perpendicular to the spontaneous symmetry breaking N direction [3]. The n-point functions can then be calculated to one-loop order, and inserted into Callan–Symanzik equation to obtain how much the renormalized coupling constant changes with regards to the energe scale μ.

$$\beta^\mu(g_{\overline{\mathrm{MS}}}) = \mu \frac{\mathrm{d} g_{\overline{\mathrm{MS}}}}{\mathrm{d}\mu} = -(N-2)\frac{1}{4\pi} g_{\overline{\mathrm{MS}}}^3 + \mathcal{O}(g_{\overline{\mathrm{MS}}}^5), \tag{1.1.3}$$

where $g_{\overline{\mathrm{MS}}}$ is the renormalized coupling constant. Note, that this equation is true, if we define the coupling constant in the modified minimal subtraction renormalization scheme.

Let's introduce a new term into the Lagrangian which explicitly breaks the symmetry.

$$\mathcal{L}(\phi, \partial_\mu \phi) = \frac{1}{2g^2}\ \partial_\mu \phi_i\ \partial^\mu \phi_i + H_i\ \phi_i\ . \tag{1.1.4}$$

H is called the external field[1]. In this document I'm only interested in the $\mathrm{O}(3)$ model $N = 3$, in 2 spacetime dimensions, except in some proofs, which are more general.

[1]Later we will see, that if we discretize this field, H can be thought of as an external magnetic field acting on a spin field. This is the only reason we can call H an external *magnetic* field, this model does not relate to electrodynamics directly.

Let's assume first, that H is parallel to x_3 without loss of generality. In this case, the last term becomes just $H_3\phi_3$. By defining $\phi'_i := g^{-1}\phi_i$, the Lagrangian becomes

$$\mathcal{L}(\phi', \partial_\mu \phi') = \frac{1}{2}\, \partial_\mu \phi'_i\, \partial^\mu \phi'_i + H_3\sqrt{1 - g^2(\phi'^2_1 + \phi'^2_2)}\ . \tag{1.1.5}$$

If $g^2 \ll 1$, then

$$\partial_\mu \phi'_3 = -\frac{g^2}{2}(1 - g^2(\phi'^2_1 + \phi'^2_2))^{-1/2}\partial_\mu(\phi'^2_1 + \phi'^2_2), \tag{1.1.6}$$

which means, that $\partial_\mu \phi'_3\ \partial_\mu \phi'_3 = \mathcal{O}(g^4)$, I will ignore this, and

$$\phi'_3 = \sqrt{1 - g^2(\phi'^2_1 + \phi'^2_2)} \approx 1 - \frac{g^2}{2}(\phi'^2_1 + \phi'^2_2) + \mathcal{O}(g^4). \tag{1.1.7}$$

The perturbative Lagrangian up until $\mathcal{O}(g^2)$ then becomes

$$\mathcal{L}(\phi', \partial_\mu \phi') \approx \frac{1}{2}\, (\partial_\mu \phi'_1\, \partial^\mu \phi'_1 + \partial_\mu \phi'_2\, \partial^\mu \phi'_2) - \frac{H_3 g^2}{2}(\phi'^2_1 + \phi'^2_2)\ , \tag{1.1.8}$$

which can be thought of as a theory with fields ϕ'_1, ϕ'_2 with $\sqrt{H_3}g$ masses. This is why we are interested in this theory. I introduced a term with explicit symmetry breaking, and masses appeared.

In the calculations, speed of light $c = 1$, therefore, time has length dimensions. The reduced Planck constant $\hbar = 1$, energy has inverse length dimension. The full Lagrangian ($L = \int \mathrm{d}x\ \mathcal{L}$) has energy dimension, therefore, in this one space dimension model, the Lagrangian $\mathcal{L}$ has dimension length^{-2}. The fields ϕ are dimensionless, and ∂_μ has dimension inverse length, therefore, g is also dimensionless, and H has a dimension of length^{-2}.

In this document I'm interested in the finite volume model. It's linear size is L, and in two dimension, it's volume is L^2. It's convenient here to observe how $g_{\overline{\text{MS}}}$ changes with regards to changing the volume. By doing this carefully, so that the physics do not change (dimensionful quantities stay the same relative to each other), L can be used instead of μ to measure β. Because the dimension of L is the inverse of the dimension of μ, β with regards to L will be -1 times β^μ,

$$\beta(g_{\overline{\text{MS}}}) = L\frac{\mathrm{d}g_{\overline{\text{MS}}}}{\mathrm{d}L} = \frac{\mathrm{d}g_{\overline{\text{MS}}}}{\mathrm{d}\ln(L)} = -\frac{\mathrm{d}g_{\overline{\text{MS}}}}{\mathrm{d}\ln(\mu)} = -\beta^\mu(g_{\overline{\text{MS}}}). \tag{1.1.9}$$

For the calculations I'm doing, it's not possible to change L infinitesimally, the coupling constant can only be measured between $s \times L$ and L. s can be anything greater than 1, the only restriction is that the coupling constant has to change smooth enough between sL and L. Practical values are $s = 2, 3/2$. The change of the coupling constant in this case is called the discrete beta function

$$\frac{\Delta g_{\overline{\text{MS}}}}{\Delta \ln(L)} = \frac{g_{\overline{\text{MS}}}(sL) - g_{\overline{\text{MS}}}(L)}{\ln(sL) - \ln(L)} = \frac{g_{\overline{\text{MS}}}(sL) - g_{\overline{\text{MS}}}(L)}{\ln(s)}. \tag{1.1.10}$$

1.2 Work done in thesis

In this thesis I'm calculating the discrete beta function of the coupling constant g_2 (defined in detail in section 2), in a finite $L \times L$ volume, with external field, using Monte-Carlo methods to see the effects of the external field. It is important to note, that the later defined g_2 is not arbitrary: in the limit $g_2 \ll 1$, $g_2 \approx g_{\overline{\mathrm{MS}}}$, if the external field is small.

The whole theory has to be discretized for the Monte-Carlo simulations, which generate fields. The software I wrote measure g_2 explicitly as a function of the characteristic length, namely L.

The original work in this thesis are the followings.

- Finding out what is the algorithm based on rough instructions from my supervisor. For the Wolff algorithm, he only specified, that the clusters have to be flipped with unequal probabilities in case $H \neq 0$, but didn't specify how. For the Shadow algorithm, he only pointed out, that external field conceptionally is the same as having one extra point which all the other points connect to without an external field, but didn't specify what is the update method. Note, that these algorithms are not original ideas, but I couldn't find in the literature what exactly is the algorithm.

- Proving that the detailed algorithms work, in a generic case: in D dimensions and with a generic F function.

- Implementing the algorithms.

- Comparing the different methods.

- Calculating discrete beta functions in this model for $H \neq 0$.

2 Definitions

2.1 g_2

2.1.1 Continuous, infinite spacetime

First, let's define g_2 in an infinite spacetime, and using a familiar renormalization scheme. Let's use modified minimal subtraction[2] here. The bare field expressed with the renormalized (R) field is

$$\phi = Z_\phi^{1/2}\,\phi^{\mathrm{R}}. \tag{2.1.1}$$

I'm going to use the renormalized field now, so that we can easily verify, that the following quantities are finite, well defined. The correlation function and moments are

$$\overline{C}_i^{\mathrm{R}}(x_0) := \int \mathrm{d}x_1\ \langle \phi_i^{\mathrm{R}}(x_0,x_1)\phi_i^{\mathrm{R}}(0,0)\rangle, \tag{2.1.2}$$

$$\overline{M}_0^{\mathrm{R}} := \sum_i \int \mathrm{d}x_0\ \overline{C}_i^{\mathrm{R}}(x_0), \tag{2.1.3}$$

$$\overline{M}_2^{\mathrm{R}} := \frac{1}{(2\pi)^2}\sum_i \int \mathrm{d}x_0\ x_0^2\ \overline{C}_i^{\mathrm{R}}(x_0). \tag{2.1.4}$$

Then, g_2 is defined as

$$\overline{g}_2(\mu) := \frac{1}{\sqrt{2}}\left(2\,\frac{\overline{M}_0^{\mathrm{R}}}{\overline{M}_2^{\mathrm{R}}} - 4\pi^2\right)^{1/2}. \tag{2.1.5}$$

This, of course, depends on μ through the moments and correlation function, because the renormalization scheme defines ϕ^R differently for different values of μ. I'm not analysing this further, because, in this document I'm only interested in the finite volume case, but it is important to see that g_2 can be defined similarly in infinite volume.

2.1.2 Continuous, finite spacetime

The finite spacetime's boundary condition is that it's periodic.

$$\phi(x_0 + L, x_1) = \phi(x_0, x_1), \tag{2.1.6}$$

$$\phi(x_0, x_1 + L) = \phi(x_0, x_1). \tag{2.1.7}$$

[2]This renormalization scheme is too restrictive in this subsection, there could be other schemes for which these calculations hold, but for pedagogical reasons, I'm choosing this scheme here.

The correlation function and moments in this case are very similar, but for convenience[3] let's divide the integrals with L.

$$C_i^{\mathrm{R}}(x_0) := \frac{1}{L}\int \mathrm{d}x_1 \, \langle \phi_i^{\mathrm{R}}(x_0, x_1)\phi_i^{\mathrm{R}}(0,0)\rangle, \tag{2.1.8}$$

$$M_0^{\mathrm{R}} := \frac{1}{L}\sum_i \int \mathrm{d}x_0 \; C_i^{\mathrm{R}}(x_0), \tag{2.1.9}$$

$$M_2^{\mathrm{R}} := \frac{1}{L}\frac{1}{(2\pi)^2}\sum_i \int \mathrm{d}x_0 \; (2\sin(\pi x_0/L))^2 \; C_i^{\mathrm{R}}(x_0). \tag{2.1.10}$$

Then, g_2 is defined the same way

$$g_2(L) := \frac{1}{\sqrt{2}}\left(2\,\frac{M_0^{\mathrm{R}}}{M_2^{\mathrm{R}}} - 4\pi^2\right)^{1/2}. \tag{2.1.11}$$

It is convenient to choose $\mu = 1/L$, and then g_2 only depends on L.

2.1.3 Discrete, finite spacetime

In the discretized finite case, I am using the lattice regularization, with $N \times N$ samples of the field, since we are in one time and one space dimension. This way $L = a \times N$, and a is the lattice spacing. The boundary condition is the same as before: periodic in both directions, which is also sometimes called a torus topology.

The definitions in this case [4] are very similar, with the modification that $\int \mathrm{d}x \mapsto \sum a$. Here, I'm going to define the quantities using the bare fields, but g_2 can be easily given with the renormalized values, using this regulatization scheme too.

$$C_i(x_0) := \frac{1}{L/a}\sum_{x_1}\langle \phi_i(x_0, x_1)\phi_i(0,0)\rangle, \tag{2.1.12}$$

$$M_0 := \frac{1}{L/a}\sum_{x_0,i} C_i(x_0), \tag{2.1.13}$$

$$M_2 := \frac{1}{L/a}\frac{1}{(2\pi)^2}\sum_{x_0,i}(2\sin(\pi x_0/L))^2 \; C_i(x_0). \tag{2.1.14}$$

Note, that because $2\sin^2(x) = 1 - \cos(2x)$, we can rewrite M_2

$$M_2 = \frac{1}{L/a}\frac{1}{2\pi^2}\sum_{x_0,i}(1 - \cos(2\pi x_0/L)) \; C_i(x_0) = \tag{2.1.15}$$

$$= \frac{1}{2\pi^2}(M_0 - F), \tag{2.1.16}$$

where I defined F so that it is

$$F = \frac{1}{L/a}\sum_{x_0,i}\cos(2\pi x_0/L) \; C_i(x_0) = \frac{1}{L/a}\sum_{x_0,i}\exp(i \; 2\pi x_0/L) \; C_i(x_0), \tag{2.1.17}$$

[3]When calculating these values in practice for large fields, it's more practical use them in a normalized form: dividing by volume.

and I used the fact, that the correlation function is symmetric.

The second-moment correlation length is defined in [5] (4.13) as

$$\xi(L)/a = \left(\frac{M_0/F - 1}{4\sin^2(\pi a/L)}\right)^{1/2}. \tag{2.1.18}$$

Using the previous equations, we can express this using the moments.

$$a/\xi(L) = \frac{\sin(\pi a/L)}{\pi}\left(2\frac{M_0}{M_2} - 4\pi^2\right)^{1/2}. \tag{2.1.19}$$

According to [6], the leading order expansion of the step-scaling function in the current model (2 dimensions, $N = 3$) is

$$\frac{\xi(sL)}{\xi(L)} \approx s\left(1 - \ln(s)\frac{1}{8\pi}\left(\frac{\xi(L)}{L}\right)^{-2}\right). \tag{2.1.20}$$

Assuming that the leading order is enough (the correction is small in $\mathcal{O}((\frac{\xi(L)}{L})^{-4})$), we can write, that

$$\frac{\frac{sL}{\xi(sL)}}{\frac{L}{\xi(L)}} \approx 1 + \ln(s)\frac{1}{8\pi}\left(\frac{L}{\xi(L)}\right)^2. \tag{2.1.21}$$

Multiplying both sides with $\frac{L}{\xi(L)\ln(s)}$, and rearranging the terms gets us

$$\frac{\frac{sL}{\xi(sL)} - \frac{L}{\xi(L)}}{\ln(s)} \approx \frac{1}{8\pi}\left(\frac{L}{\xi(L)}\right)^3. \tag{2.1.22}$$

Let's use the following identity.

$$\ln(s) = \ln\left(\frac{sL/a}{L/a}\right) = \ln(sL/a) - \ln(L/a). \tag{2.1.23}$$

With these results, we can clearly see how much $\frac{L/a}{\xi(L)\sqrt{2}}$ changes with L.

$$\frac{\frac{sL}{\xi(sL)\sqrt{2}} - \frac{L}{\xi(L)\sqrt{2}}}{\ln(sL/a) - \ln(L/a)} \approx \frac{1}{4\pi}\left(\frac{L}{\xi(L)\sqrt{2}}\right)^3, \tag{2.1.24}$$

which is exactly the discrete beta function, and on the right hand side we can see, that the quantity we use behaves the same way as $g_{\overline{\text{MS}}}$ in leading order. Which is exactly what we require from g_2; let's define g_2 as

$$g_2(L/a) := \frac{L}{\xi(L)\sqrt{2}} = \frac{\sin(\pi a/L)}{\sqrt{2}\pi a/L}\left(2\frac{M_0}{M_2} - 4\pi^2\right)^{1/2}. \tag{2.1.25}$$

As we can see, this definition is very similar to the previous continuous cases.

For practical purposes, s should be big enough so that the difference between $g_2(sL)$ and $g_2(L)$ are bigger than their measured error, but small enough, so that the discrete beta function can be compared to continuous β. In our case, $s = 2$. The discrete beta function is then

$$\frac{g_2(2L/a) - g_2(L/a)}{\ln(2)}. \tag{2.1.26}$$

2.2 Choice of H

H is a bare parameter of the theory, it has dimensions of length^{-2} and it has to be renormalized. There are lots of ways to do this, and how this bare parameter is calibrated at different volumes. In this document I'm choosing a renormalization scheme which is convenient for me, based on the calculation method, and the already available quantities. It's important to see, that the choice I made can work on continuous fields too, however, working out the details in that case is not part of this thesis.

2.2.1 Renormalization of H

First, let's work in the infinite, continuous case. The Lagrangian with the bare quantities is

$$\mathcal{L}(\phi, \partial_\mu \phi) = \frac{1}{2g^2}\ \partial_\mu \phi_i\ \partial^\mu \phi_i + H_i\ \phi_i\ . \tag{2.2.1}$$

To express the Lagrangian with the renormalized quantities, first, the Z_ϕ field renormalization constant have to be introduced such that

$$\mathcal{L}(\phi^{\mathrm{R}}, \partial_\mu \phi^{\mathrm{R}}) = \frac{1}{2g^2}\ Z_\phi\ \partial_\mu \phi_i^{\mathrm{R}}\ \partial^\mu \phi_i^{\mathrm{R}} + H_i\ Z_\phi^{1/2}\ \phi_i^{\mathrm{R}}\ , \tag{2.2.2}$$

$$\phi = Z_\phi^{1/2}\ \phi^{\mathrm{R}}. \tag{2.2.3}$$

To have a finite renormalized H^{R}, H should be multiplied by the field renormalization, and some finite value

$$H^{\mathrm{R}} \propto Z_\phi^{1/2} H. \tag{2.2.4}$$

Let's say, that in this infinite, continuous theory we already calculated M_0 bare moment for a given H.

$$\overline{M}_0 = Z_\phi \underbrace{\sum_i \int \mathrm{d}x_0\ \mathrm{d}x_1\ \langle \phi_i^{\mathrm{R}}(x_0, x_1) \phi_i^{\mathrm{R}}(0,0) \rangle}_{\text{finite}}. \tag{2.2.5}$$

As we can see, because the summation and integration parts only contain the renormalized fields, which is finite, it's very convenient to define the renormalized external field as

$$H^{\mathrm{R}} := \overline{M}_0^{1/2} H. \tag{2.2.6}$$

This choice can be made analogously in the finite theory too: $H^{\mathrm{R}} := M_0^{1/2} H$.

2.2.2 Constant line of physics

In a finite field theory, to measure the discrete beta function, the change of the coupling constant needs to be calculated with regards to the linear size. It has to be decided how the renormalized external field H^{R} changes when the linear size changes. It's not necessarily constant with regards to L, it depends on how we define the physical properties of the model. H has a dimension of L^{-2}, therefore for practical purposes, to make calculations easier and dimensionless, the following value is constant in the model

$$L^2 H^{\mathrm{R}} = R \text{ (constant)} . \tag{2.2.7}$$

This can be expressed with the bare quantities

$$L^2 H M_0^{1/2} = R \text{ (constant)} . \tag{2.2.8}$$

On the lattice regularization, the same exact restriction is kept completely analogously.

In order to find out the right H for a given g, I could iteratively modify H, so that the R is a constant across every value of g and H, or interpolate the results to find out which H would give me the correct condition.

3 Calculation method

To calculate g_2 (and ultimately the discrete beta function from there), I need to calculate moments M_0, M_2. I'm going to use the Feynman path integral method, which says, that the expectation value for a given $\mathcal{O}$ operator is

$$\langle \mathcal{O}(\phi) \rangle = \frac{1}{Z} \int [\mathrm{d}\phi]\ \mathcal{O}(\phi) e^{iS(\phi)}, \tag{3.0.1}$$

$$Z = \int [\mathrm{d}\phi]\ e^{iS(\phi)}, \tag{3.0.2}$$

$$S(\phi) = \int \mathrm{d}^2x\ \mathcal{L}(\phi(x), \partial_\mu \phi(x)). \tag{3.0.3}$$

Z can just be thought of as a normalization for the whole integral.
Here $[\mathrm{d}\phi] = \mathrm{d}\phi(x_0)\ \mathrm{d}\phi(x_1) \cdots$, which means, that the integration is over all possible values of the field, all possible paths of ϕ. This is a complex integral, the S action is multiplied by i in the exponent. It's much more convenient to use real integrals. Wick rotation's theorem can be used to switch from real time to imaginary time, which simplifies the integral to a real valued one, provided, that we consistently work in Euclidean spacetime from now on instead of Lorentzian.

$$\langle \mathcal{O}(\phi) \rangle = \frac{1}{Z} \int [\mathrm{d}\phi]\ \mathcal{O}(\phi) e^{-S(\phi)}, \tag{3.0.4}$$

$$S(\phi) = \frac{1}{2g^2} \int \mathrm{d}^2x\ \partial_\mu \phi_i \partial_\mu \phi_i + \int \mathrm{d}^2x\ H_i \phi_i. \tag{3.0.5}$$

In the $O(3)$ model, the components of ϕ are real, so every value is real in the above equations. To calculate the expectation value, we can think of $e^{-S(\phi)}/Z$ as the probability on the the space $[\phi]$ (different functions of $\phi(x)$) for $\mathcal{O}(\phi)$ to occur.

$$p(\phi) \propto e^{-S(\phi)}. \tag{3.0.6}$$

3.1 Correlation function

To calculate M_0 and M_2 (and from there g_2 using equation (2.1.25)), I need to calculate the correlation function, and then use equations (2.1.13) and (2.1.14).

I'm using a slightly different method than equation (2.1.12) to calculate correlation function. This makes calculations a little bit faster.

$$C_i(x_0) = \frac{1}{(L/a)^2} \sum_{x_1} \sum_{y_1} \langle \phi_i(x_0, x_1) \phi_i(0, y_1) \rangle. \tag{3.1.1}$$

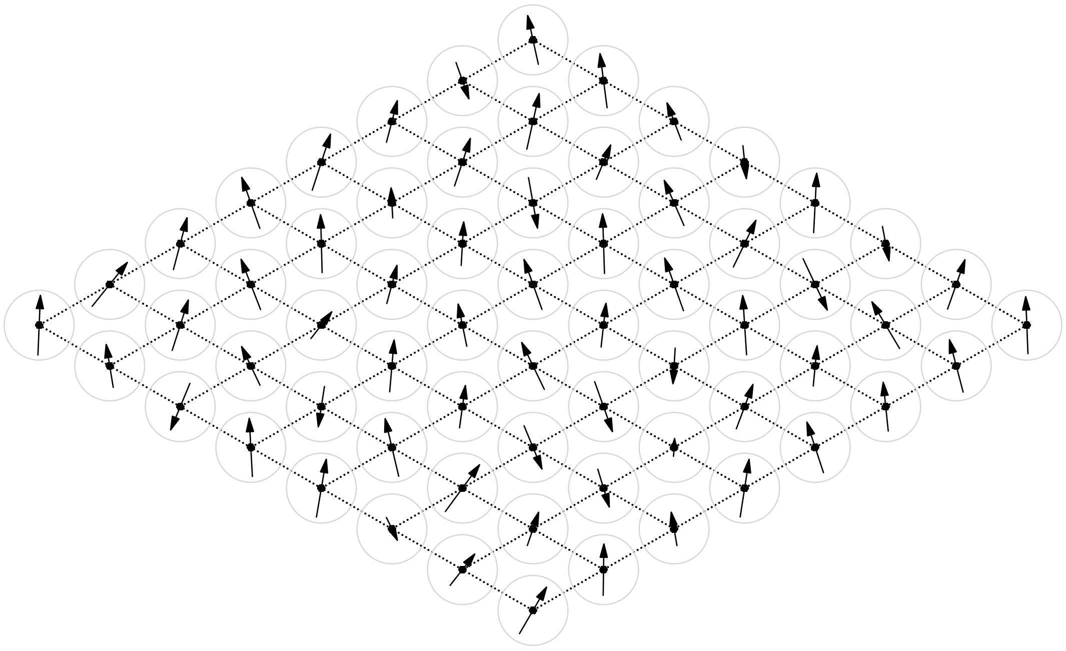
The result is the same as using the original equation (2.1.12), because

$$\langle \phi_i(x_0, x_1) \phi_i(0, y_1) \rangle = \langle \phi_i(x_0, x_1 - y_1) \phi_i(0, 0) \rangle, \tag{3.1.2}$$

where I used the property, that the 2-point function has a translational invariance.

3.2 Discretization

The regularization method is to discretize the field.



The field is defined, only on an $N \times N$ lattice $\phi(x)$. Also, the boundary condition is that ϕ is periodic

$$\phi(x_0 + L, x_1) = \phi(x_0, x_1), \tag{3.2.1}$$

$$\phi(x_0, x_1 + L) = \phi(x_0, x_1). \tag{3.2.2}$$

The actions S has to be calculated on this lattice. Let's calculate the first part of the action.

$$\frac{1}{2g^2} \int \mathrm{d}^2x \ \partial_\mu \phi_i \partial_\mu \phi_i \mapsto \tag{3.2.3}$$

$$\mapsto \frac{1}{2g^2} \sum_{x_0, x_1} a^2 \left[\frac{1}{a^2} (\phi(x_0 + a, x_1) - \phi(x_0, x_1))^2 + \frac{1}{a^2} (\phi(x_0, x_1 + a) - \phi(x_0, x_1))^2 \right] = \tag{3.2.4}$$

$$= \frac{1}{2g^2} \sum_{x_0, x_1} [-2(\phi_i(x_0 + a, x_1) + \phi_i(x_0, x_1 + a))\phi_i(x_0, x_1)$$

$$+ \underbrace{\phi^2(x_0 + a, x_1) + \phi^2(x_0, x_1 + a) + 2\phi^2(x_0, x_1)}_{=4}] = \tag{3.2.5}$$

$$= -\frac{1}{g^2} \sum_{(x,y)} \phi_i(x)\phi_i(y) + C, \tag{3.2.6}$$

where I used the fact, that in the $O(3)$ model, $\phi^2(x) = 1$, for every x, and the summation (x, y) means, the x and y are "neighbors" in a sense, that only those x and y pairs are counted, where the distance between them is just a in any of the two directions. C accounts for everything which is field independent – we know, that if we add a constant to the lagrangian, it does not change any of the measurable quantities.

The second half of the action can be easily accounted for on the lattice.

$$\int \mathrm{d}^2x\ H_i\phi_i \mapsto \sum_x\ a^2H_i\phi_i(x) = \sum_x\ h_i\phi_i(x). \tag{3.2.7}$$

I defined $h_i := a^2H_i$, because it is a more convenient quantity to be used for the calculations. For convenience I also define $\beta := g^{-2}$, because our model is mathematically very similar to the Ising model, with $\beta^{-1} = k_BT$. Note, that it has nothing to do to the β function, which relates the coupling constant to the energy scale. From now on, β will refer to this newly defined quantity.

The full action is[4]

$$S(\phi) = -\beta \sum_{(x,y)} \phi_i(x)\phi_i(y) + \sum_x h_i\phi_i(x), \tag{3.2.8}$$

and using equation (2.2.8) the renormalization condition is

$$M_0^{1/2}\left(\frac{L}{a}\right)^2 h = \text{constant}. \tag{3.2.9}$$

3.3 Field generation

To calculate the correlation function, the fields have to be generated with the probability

$$p(\phi) \propto e^{-S(\phi)}. \tag{3.3.1}$$

I am using a method which is well-known in Lattice field theory calculations. I am generating a chain of fields – a Markov chain –, where each subsequent field is calculated from the previous one.

$$\phi_1(x) \mapsto \phi_2(x) \mapsto \cdots \mapsto \phi_n(x) \tag{3.3.2}$$

3.3.1 Conditions for field generation

There are two conditions which need to be satisfied in order to generate these values – the fields – with the right probability. I only outline the conditions here shortly, a more detailed description and a proof is in [7].

Let's say, that f is the functions which generates the new field configuration from the previous one,

$$f : \phi \mapsto \phi', \tag{3.3.3}$$

[4]Here, I neglected the C constant I wrote in earlier equations, because this does not affect the results.

and that $W_f(\phi \mapsto \phi')$ is the probability, that f generates ϕ' from ϕ.

Then, the first condition is that this f should preserve the probability after the "update" (applying f to the field), namely

$$\int \mathrm{d}\phi \; p(\phi) \; W_f(\phi \mapsto \phi') = p(\phi'). \tag{3.3.4}$$

Intuitively, we can see, that this might be a good condition, because when we apply f to a large ensamble of fields which already has a distribution $p(\phi)$, we are going to get the same distribution, which is what we would like. However, the generation process starts out with only one field, and this first condition can only guarantee, that the distribution will get closer and closer to the distribution we want, but not infinitesimally close this distribution we require.

The second condition says, that: starting from any ϕ, the repeated application of f brings us arbitrarily close to any other ϕ'. This is called ergodicity.

These two conditions together will ensure, that applying f finite times gets us arbitrarily close to the required $p(\phi)$ distribution.

All of the functions f I used in this work have the property, that

$$W_f(\phi \mapsto \phi') = \frac{p(\phi')}{p(\phi)} \; W_f(\phi' \mapsto \phi). \tag{3.3.5}$$

This is called *detailed balance.* It is evident, that this satisfies the first condition, because the probability that ϕ' updates to *any* ϕ is 1

$$\int \mathrm{d}\phi \; W_f(\phi' \mapsto \phi) = 1. \tag{3.3.6}$$

In our case, equation (3.3.5) means

$$W_f(\phi \mapsto \phi') = \exp(-\Delta S) \; W_f(\phi' \mapsto \phi), \tag{3.3.7}$$

where $\Delta S = S(\phi') - S(\phi)$. I'm going to only prove this equation for the update algorithms I used instead of the more general condition.

There are several algorithms which can be used. I implemented some candidates, and I tried them out to see which was the the most efficient. For the final calculation, I only used one of the fastest one. In the next subsections, I present these algorithms, and the proof that they satisfy the conditions in the generalized case, the D dimensional $O(N)$ model, and that the external field can depend on x, however, I only used a constant H in my calculations.

3.3.2 Metropolis algorithm

The Metropolis is the simplest, and the slowest among all I've tried. It is important to implement this also, because the implementation of this algorithm can contain the least potential mistakes, therefore all other algorithms can be tested against this. The description and proof if there is no external field is in [7]. I slightly modified this here for $h \neq 0$.

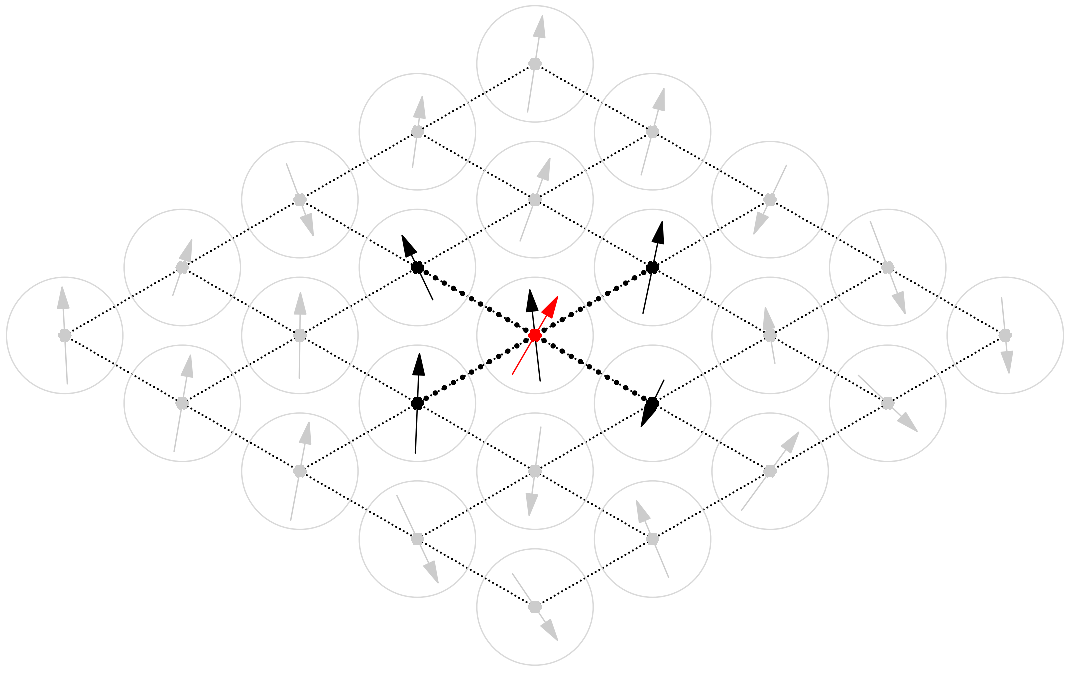


Figure 1: In the figure, the algorithm chooses the center point to update. The field's value at that point and it's neighbors are shown with black, and the possible new value is with red. Only the vectors shown with black and red (and not the lightgrey ones) are used to calculate ΔS. The algorithm then updates to the new red vector with probability $\min(1, \exp(-\Delta S))$

The $f_{\text{Metropolis}}(\phi)$ function only updates the field at only one position potentially. The outline of the algorithm is the following.

1. Choose a random position x on the field,
2. generate a random unit vector φ' uniformly on the sphere,
3. update the field value at x to φ' with probability $\min(1, \exp(-\Delta S))$.

$\Delta S = S(\phi') - S(\phi)$ and ϕ' is the same as ϕ, except at position x, where the field value is φ'. It is obvious, that ΔS only depends on the field values at x, which are $\phi(x)$, φ', and it's neighbors.

$$\Delta S = \left[-\beta \sum_y \phi(y) + h(x) \right] (\varphi' - \phi(x)), \tag{3.3.8}$$

where y are neighbors of x.

The advantage of this algorithm is that it's very easy to implement, however, because in each step only one point is updated, it is very slow.

Proof of the algorithm

There are 3 cases, where the detailed balance has to be proven.

First, detailed balance is evident, if $\phi = \phi'$, because in this case we only have to prove, that $W_f(\phi \mapsto \phi) = \exp(-\Delta S)\ W_f(\phi \mapsto \phi)$. This is clearly true, because in this case $\Delta S = 0$, therefore $\exp(-\Delta S) = 1$.

If $\phi \neq \phi'$, then there are two cases. Since $f_{\text{Metropolis}}$ updates only one point, then if ϕ and ϕ' differ at two or more points, W_f is 0. In this case, the detailed balance obviously holds, with both sides being 0.

Lastly, if ϕ and ϕ' differ only at one point, then the probability is nonzero.

$$W_f(\phi \mapsto \phi') = p(x(\phi, \phi'))\ q(\phi'(x(\phi, \phi')))\ \min(1, \exp(-\Delta S)). \tag{3.3.9}$$

There are multiple new functions in this expression, but it is very simple. $x(\phi, \phi')$ is only defined on fields where only one point is different, and it will return with that point's coordinate. $p(x)$ will return the probability, that the update function chooses x to update, and $q(\varphi')$ will return the probability[5], that the update function chooses to update $\phi(x)$ to φ'.

Note, that the update function does not have any preference which point it chooses to update, nor which new field value it chooses there, therefore p and q are constants,

$$W_f(\phi \mapsto \phi') = pq\ \min(1, \exp(-\Delta S)), \tag{3.3.10}$$
$$W_f(\phi' \mapsto \phi) = pq\ \min(1, \exp(+\Delta S)). \tag{3.3.11}$$

From these, we can conclude, that

$$\frac{W_f(\phi \mapsto \phi')}{W_f(\phi' \mapsto \phi)} = \frac{\min(1, \exp(-\Delta S))}{\min(1, \exp(+\Delta S))} = \exp(-\Delta S), \tag{3.3.12}$$

which is identical to equation (3.3.7). In every possible case, detailed balance is satisfied.

To prove ergodicity, I have to prove, that from every possible nonzero probability state, repeated application of f can give any other nonzero probability state. Each

[5] φ' (and any other values of $\phi(x)$) can take values from an uncountably infinite set, and using probability densities would be a better choise, with $W_f(\phi \mapsto \phi')\ \mathrm{d}\phi'$. However, we can only represent a finite set of possible field values in the computer, therefore, I treat W_f as a probability with discrete random variable.

update changes one point with a nonzero probability, because $\min(1, \exp(-\Delta S))$ can never be zero. Any point has the nonzero probability to be updated to any other possible field value, since choosing any x and φ' has the same probability. It is possible to construct a long chain of f update functions, which update any ϕ to any other ϕ' point by point, and we just saw, that this has a nonzero probability.

3.3.3 Wolff algorithm with updating all clusters

The Wolff algorithm updates whole clusters instead of one point in each step. (A cluster is just an interconnected set of points. Each point can be potentially connected with their neighbors.) Because of this, it is much faster than the previously introduced Metropolis, but also more complicated. The algorithm – as with Metropolis – is in [7] with $h = 0$. I modified and generalized this for $h \neq 0$, and for a spacetime dependent $h(x)$.

This algorithm can be specified in two slightly different ways. f_{Wolff} can update every cluster, or just one cluster in each step. First I describe the former. In practice, only one cluster will be updated at a time, but it makes understanding much easier if both of them are described and proved.

For the sake of clarity, let's define the field value's component parallel to r.

$$\phi_{r,x} = r \cdot \phi(x). \tag{3.3.13}$$

$f_{\text{Wolff}}(\phi)$ updates the field the following way in each step.

1. Generate a random unit vector r with uniform probability on a 1 sphere,
2. connect each neighboring points x, y with probability

   $$1 - F(\phi_{r,x}\ \phi_{r,y}) \exp(-\beta\ \phi_{r,x}\ \phi_{r,y}), \tag{3.3.14}$$

   where F can be any function which has the properties that $0 < F(x) \leq \exp(\beta x)$ and $F(x) = F(-x)$. Two points are in one cluster if there is a path connecting them.

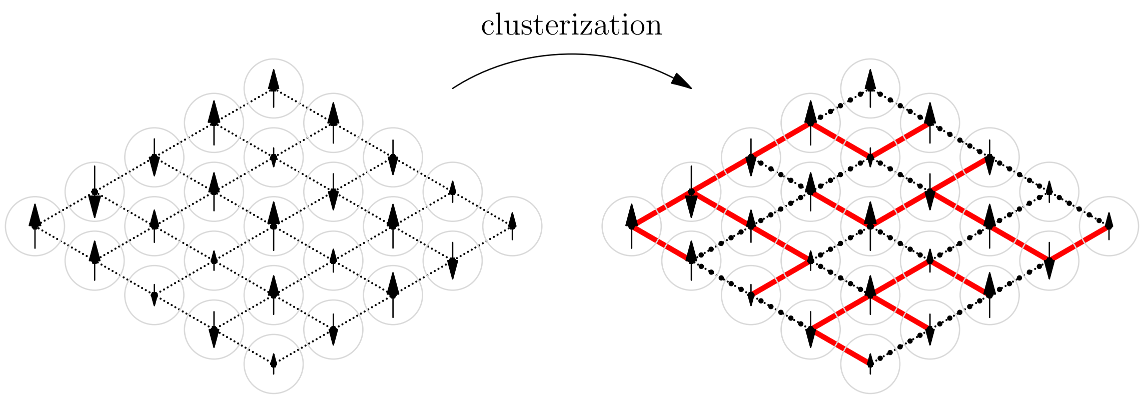


Figure 2: After the clusterization, the neighbors are connected, or disconnected. In the illustration, red means two neighbors are connected, black is that they are disconnected. Two points belong to one cluster if there is a path between them with red lines. Note, that on this figure the connections between $x = 0$ and $x = L$ (from the periodic boundary condition) are not shown for the sake of clarity. Also it's worth observing, that because only one component of the field vector is taken into account here, these values have no direction.

3. Each γ cluster is updated with probability

$$\frac{\exp\left(\sum_{x\in\gamma} h_{r,x}\ \phi_{r,x}\right)}{\exp\left(-\sum_{x\in\gamma} h_{r,x}\ \phi_{r,x}\right)+\exp\left(\sum_{x\in\gamma} h_{r,x}\ \phi_{r,x}\right)}. \tag{3.3.15}$$

If the cluster is updated, each field value in it is modified as $\phi_{r,x} \mapsto -\phi_{r,x}$, otherwise the cluster is left as it is.

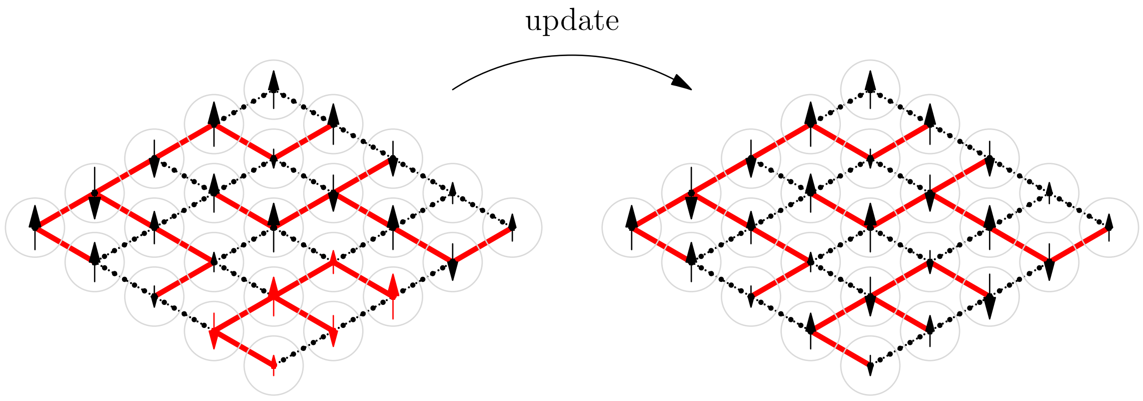


Figure 3: The update process flips clusters with the previously defined probability. Here, as we can see only the bottom cluster is flipped, which is highlighted with red.

4. Do previous substeps again from the beginning with as much random unit vectors perpendicular to each other as possible.

3.3.4 Proof of the Wolff algorithm with updating all clusters

The probability, that the update function maps configuration ϕ to configuration ϕ' is $W(\phi \mapsto \phi')$. In this section I will prove the detailed balance. All of the subsequent proofs for the algritms are generalized to a field with D dimensional vectors.

At the beginning of the step, the algorithm chooses one random unit vector r, and it generates the clusters based **only** on the field values projected to r. Not only that, but if it decides to update a cluster, it will **only** change the field values' component which is parallel to r. In each step the chosen r_1, r_2, ... r_D are perpendicular to each other – see the last substep –, which means, that an update for r_i is completely independent of the update r_j, where r_i and r_j are perpendicular. We can express this as

$$W_f(\phi \mapsto \phi') = \sum_{\substack{r_1,\cdots r_D \\ r_k \cdot r_l = \delta_{k,l}}} u(r_1, ..., r_D)\; W_f(\phi_{r_1} \mapsto \phi'_{r_1}) \;\cdots\; W_f(\phi_{r_D} \mapsto \phi'_{r_D}). \quad (3.3.16)$$

ϕ_{r_n} and ϕ'_{r_n} are just the projection of the fields to the vector r_n, and $u(r_1, ..., r_D)$ is the probability that these particular unit vectors are chosen.

The function u is constant, because all the possible set of unit vectors which are perpendicular to each other have the same probability to be chosen. For this reason, I will ignore this from now on.

Let's define $S(\phi_r)$ as

$$S(\phi_r) = -\beta \sum_{(x,y)} \phi_r(x)\phi_r(y) + \sum_x h_r(x)\phi_r(x), \quad (3.3.17)$$

and $\Delta S_r = S(\phi'_r) - S(\phi_r)$. This is just part of the action which comes from the r component of the fields.
Then, if I can somehow prove, that

$$W_f(\phi_r \mapsto \phi'_r) = \exp(-\Delta S_r)\; W_f(\phi'_r \mapsto \phi_r), \quad (3.3.18)$$

then, detailed balance (equation (3.3.7)) is satisfied

$$W_f(\phi \mapsto \phi') = \exp(-\Delta S)\; W_f(\phi' \mapsto \phi), \quad (3.3.19)$$

if we insert equation (3.3.18) into equation (3.3.16), and use the fact, that

$$\exp(\Delta S_{r_1}) \;\cdots\; \exp(\Delta S_{r_D}) = \exp\left(\sum_i \Delta S_{r_i}\right) = \exp(\Delta S). \quad (3.3.20)$$

Now, the task is reduced to prove equation (3.3.18). The f_{Wolff} update function first connects the points, and then updates the field. Let's denote $\mathcal{C}$ clusterization

as one of the possible ways the algorithm connects the neighbors to create clusters on the field. $\mathcal{P}\big(\phi_r \mapsto (\phi_r, \mathcal{C})\big)$ is the probability, that the algorithm creates $\mathcal{C}$ clusterization on the ϕ_r field, and $\mathcal{Q}\big((\phi_r, \mathcal{C}) \mapsto \phi'_r\big)$ is the probability, that the algorithm updates the ϕ_r field with $\mathcal{C}$ clusterization to ϕ'_r field.
The algorithm can choose from several possible $\mathcal{C}$ clusterization when the update happens, thus, the probability that it will update from ϕ_r to ϕ'_r is

$$W_f(\phi_r \mapsto \phi'_r) = \sum_{\mathcal{C}} \mathcal{P}\big(\phi_r \mapsto (\phi_r, \mathcal{C})\big)\ \mathcal{Q}\big((\phi_r, \mathcal{C}) \mapsto \phi'_r\big). \tag{3.3.21}$$

Let's break up $S_r(\phi)$ into two components: E as "external" and I as "internal" parts.

$$S(\phi_r) = \underbrace{-\beta \sum_{(x,y)} \phi_r(x)\phi_r(y)}_{S^{\mathrm{I}}(\phi_r)} + \underbrace{\sum_x h_r(x)\phi_r(x)}_{S^{\mathrm{E}}(\phi_r)} = S^{\mathrm{I}}(\phi_r) + S^{\mathrm{E}}(\phi_r). \tag{3.3.22}$$

I will prove, that

$$\mathcal{P}\big(\phi_r \mapsto (\phi_r, \mathcal{C})\big) = \exp(-\Delta S_r^{\mathrm{I}})\ \mathcal{P}\big(\phi'_r \mapsto (\phi'_r, \mathcal{C})\big), \tag{3.3.23}$$

$$\mathcal{Q}\big((\phi_r, \mathcal{C}) \mapsto \phi'_r\big) = \exp(-\Delta S_r^{\mathrm{E}})\ \mathcal{Q}\big((\phi'_r, \mathcal{C}) \mapsto \phi_r\big), \tag{3.3.24}$$

$$\Delta S_r^{\mathrm{I}} := S^{\mathrm{I}}(\phi'_r) - S^{\mathrm{I}}(\phi_r), \tag{3.3.25}$$

$$\Delta S_r^{\mathrm{E}} := S^{\mathrm{E}}(\phi'_r) - S^{\mathrm{E}}(\phi_r), \tag{3.3.26}$$

if the update is *valid*[6] for a given $\mathcal{C}$ clusterization from ϕ_r to ϕ'_r. For *invalid* updates, $\mathcal{Q}\big((\phi_r, \mathcal{C}) \mapsto \phi'_r\big) = 0$, and

$$(\phi_r, \mathcal{C}) \mapsto \phi'_r \text{ is } \textit{valid} \Leftrightarrow (\phi'_r, \mathcal{C}) \mapsto \phi_r \text{ is } \textit{valid}. \tag{3.3.27}$$

(Note, that this also means, that $(\phi_r, \mathcal{C}) \mapsto \phi'_r$ is *invalid* $\Leftrightarrow$ $(\phi'_r, \mathcal{C}) \mapsto \phi_r$ is *invalid*, because *valid* and *invalid* property is mutually exclusive.)
Let's insert these into equation (3.3.21).

$$W_f(\phi_r \mapsto \phi'_r) = \tag{3.3.28}$$

$$= \sum_{\textit{valid}\ \mathcal{C}} \mathcal{P}\big(\phi_r \mapsto (\phi_r, \mathcal{C})\big)\mathcal{Q}\big((\phi_r, \mathcal{C}) \mapsto \phi'_r\big) +$$

$$+ \underbrace{\sum_{\textit{invalid}\ \mathcal{C}} \mathcal{P}\big(\phi_r \mapsto (\phi_r, \mathcal{C})\big)\mathcal{Q}\big((\phi_r, \mathcal{C}) \mapsto \phi'_r\big)}_{=0,\text{ because } \mathcal{Q}=0} = \tag{3.3.29}$$

$$= \underbrace{\exp(-\Delta S_r^{\mathrm{I}} - \Delta S_r^{\mathrm{E}})}_{\exp(-\Delta S_r)} \sum_{\textit{valid}\ \mathcal{C}} \mathcal{P}\big(\phi'_r \mapsto (\phi'_r, \mathcal{C})\big)\mathcal{Q}\big((\phi'_r, \mathcal{C}) \mapsto \phi_r\big). \tag{3.3.30}$$

[6] I will define what is *valid* in detail in the next paragraphs. For now, let's just say, that this is the condition for the equations presented just now.

Similarly, for $W_f(\phi'_r \mapsto \phi_r)$ we can get[7]

$$W_f(\phi'_r \mapsto \phi_r) = \sum_{valid\ \mathcal{C}} \mathcal{P}\big(\phi'_r \mapsto (\phi'_r, \mathcal{C})\big)\mathcal{Q}\big((\phi'_r, \mathcal{C}) \mapsto \phi_r\big). \tag{3.3.31}$$

With these, it is easy to prove equation (3.3.18)

$$W_f(\phi_r \mapsto \phi'_r) = \tag{3.3.32}$$

$$= \exp(-\Delta S_r) \sum_{valid\ \mathcal{C}} \mathcal{P}\big(\phi'_r \mapsto (\phi'_r, \mathcal{C})\big)\mathcal{Q}\big((\phi'_r, \mathcal{C}) \mapsto \phi_r\big) = \tag{3.3.33}$$

$$= \exp(-\Delta S_r) W_f(\phi'_r \mapsto \phi_r), \tag{3.3.34}$$

In the next paragraphs, I'm going to prove equations (3.3.23), (3.3.24) and (3.3.27) along with defining what *valid* means.

Clusterization

In this paragraph I'm only going to prove equation (3.3.23), which define the probability a clusterization happens. These equations also have a condition, and only then they are true: the update $(\phi_r, \mathcal{C}) \mapsto \phi'_r$ should be *valid*.

The update $\phi_r \mapsto \phi'_r$ with clusterization $\mathcal{C}$ is considered *valid*, if in each cluster every point "flips" (multiplies with -1), or does not "flip" it's values alongside the vector r. This just means, that there cannot be two points in one cluster where $\phi_r(x) \mapsto \phi_r(x)$, and $\phi_r(y) \mapsto -\phi_r(y)$. This would be an *invalid* update. But, of course, an update can be *valid*, if one cluster "flips", and an other cluster does not "flip" all of it's values alongside the r vector. Let's define $\mathcal{C}^{\mathrm{c}}$ as the set of pair[8] of points which are connected in the clusterization $\mathcal{C}$. Then, we can write down a very important property. If two points x, y are connected in a cluster, then their update along the r vector does not change it's product along r

$$(x, y) \in \mathcal{C}^{\mathrm{c}} \Rightarrow \phi_{r,x}\phi_{r,y} = \phi'_{r,x}\phi'_{r,y}. \tag{3.3.35}$$

It is also important to note, that if two points are disconnected, then the absolute value of their product does not change, because their value can only be multiplied by -1 or +1

$$(x, y) \in \mathcal{C}^{\mathrm{d}} \Rightarrow |\phi_{r,x}\phi_{r,y}| = |\phi'_{r,x}\phi'_{r,y}|. \tag{3.3.36}$$

For a chosen r vector, the clusterization process consists of connecting each two neighboring points on the lattice with probability $1 - F(\phi_{r,x}\phi_{r,y}) \exp(-\beta\phi_{r,x}\phi_{r,y})$,

[7]For the sake of clarity, I'm not going to write down the full condition for the summation. Of course, by definition in this equation ϕ and ϕ' are swapped compared to the previous equation, but because of equation (3.3.27), it doesn't matter whether they are swapped or not in the summation condition.

[8]Also, let's define $\mathcal{C}^{\mathrm{c}}$ so that if (x, y) is in this set, that (y, x) is not in this set $x \neq y$.

and keep them disconnected with probability $F(\phi_{r,x}\phi_{r,y})\exp(-\beta\phi_{r,x}\phi_{r,y})$. Note, that $F(x) = F(-x)$, and if an update is *valid*, then for any two neighboring points F will stay the same because of equation (3.3.36). Now, for a *valid* update $\mathcal{P}$ can be written as

$$\mathcal{P}\big(\phi_r \mapsto (\phi_r, \mathcal{C})\big) = \tag{3.3.37}$$

$$= \prod_{(x,y)\in\mathcal{C}^{\mathrm{c}}} [1 - F(\phi_{r,x}\phi_{r,y})\exp(-\beta\phi_{r,x}\phi_{r,y})] \prod_{(z,w)\in\mathcal{C}^{\mathrm{d}}} F(\phi_{r,z}\phi_{r,w})\exp(-\beta\phi_{r,z}\phi_{r,w}). \tag{3.3.38}$$

Let's examine the second product, and use the fact, that $\prod_{(i,j)\in\mathcal{C}} = \prod_{(i,j)\in\mathcal{C}^{\mathrm{c}}} \times \prod_{(k,l)\in\mathcal{C}^{\mathrm{d}}}$

$$\prod_{(z,w)\in\mathcal{C}^{\mathrm{d}}} F(\phi_{r,z}\phi_{r,w})\exp(-\beta\phi_{r,z}\phi_{r,w}) = \prod_{(z,w)\in\mathcal{C}^{\mathrm{d}}} F(\phi_{r,z}\phi_{r,w}) \prod_{(i,j)\in\mathcal{C}^{\mathrm{d}}} \exp(-\beta\phi_{r,i}\phi_{r,j}) = \tag{3.3.39}$$

$$= \prod_{(z,w)\in\mathcal{C}^{\mathrm{d}}} F(\phi_{r,z}\phi_{r,w}) \prod_{(i,j)\in\mathcal{C}^{\mathrm{c}}} \exp(+\beta\phi_{r,i}\phi_{r,j}) \underbrace{\prod_{(k,l)\in\mathcal{C}} \exp(-\beta\phi_{r,k}\phi_{r,l})}_{\exp(S^{\mathrm{I}}(\phi_r))}. \tag{3.3.40}$$

After we insert this result into the previous equation we get

$$\mathcal{P}\big(\phi_r \mapsto (\phi_r, \mathcal{C})\big) = \tag{3.3.41}$$

$$= \exp(S^{\mathrm{I}}(\phi_r)) \underbrace{\prod_{(x,y)\in\mathcal{C}^{\mathrm{c}}} [\exp(\beta\phi_{r,x}\phi_{r,y}) - F(\phi_{r,x}\phi_{r,y})] \prod_{(z,w)\in\mathcal{C}^{\mathrm{d}}} F(\phi_{r,z}\phi_{r,w})}_{f_{\mathcal{P}}(\phi_r,\mathcal{C}):=} = \tag{3.3.42}$$

$$= \exp(S^{\mathrm{I}}(\phi_r)) f_{\mathcal{P}}(\phi_r, \mathcal{C}). \tag{3.3.43}$$

where I introduced $f_{\mathcal{P}}(\phi_r, \mathcal{C})$. Notice, that for *valid* updates, $f_{\mathcal{P}}(\phi_r, \mathcal{C}) = f_{\mathcal{P}}(\phi'_r, \mathcal{C})$, because F is symmetric, and $|\phi_{r,z}\phi_{r,w}| = |\phi'_{r,z}\phi'_{r,w}|$, and if x, y are connected, that means they are in the same cluster, so they change sign together, $(x, y) \in \mathcal{C}^{\mathrm{c}} \Rightarrow \exp(\beta\phi_{r,x}\phi_{r,y}) = \exp(\beta\phi'_{r,x}\phi'_{r,y})$.
This last result also means, that

$$\mathcal{P}\big(\phi'_r \mapsto (\phi'_r, \mathcal{C})\big) = \exp(S^{\mathrm{I}}(\phi'_r)) f_{\mathcal{P}}(\phi'_r, \mathcal{C}) = \tag{3.3.44}$$

$$= \exp(S^{\mathrm{I}}(\phi'_r)) f_{\mathcal{P}}(\phi_r, \mathcal{C}). \tag{3.3.45}$$

Combining these we get

$$\mathcal{P}\big(\phi_r \mapsto (\phi_r, \mathcal{C})\big) = \tag{3.3.46}$$

$$= \exp(S^{\mathrm{I}}(\phi_r)) f_{\mathcal{P}}(\phi_r, \mathcal{C}) = \exp(-\Delta S^{\mathrm{I}}) \exp(S^{\mathrm{I}}(\phi'_r)) f_{\mathcal{P}}(\phi'_r, \mathcal{C}) = \tag{3.3.47}$$

$$= \exp(-\Delta S^{\mathrm{I}}) \mathcal{P}\big(\phi'_r \mapsto (\phi'_r, \mathcal{C})\big). \tag{3.3.48}$$

Updating

In this paragraph, I'm going to prove equations (3.3.24) and (3.3.27).

There is also an other claim which needs to be proven, which is trivial: if the update $\phi_r \mapsto \phi'_r$ using $\mathcal{C}$ clusterization is *invalid*, then $\mathcal{Q}\big((\phi_r, \mathcal{C}) \mapsto \phi'_r\big) = 0$. The previous paragraph defined what *valid* means. The update step's 3rd point says, that every point in a cluster will change sign or stay as it is together. This makes it impossible that in one cluster one point changes sign, and an other does not, it's probability is zero ($\mathcal{Q} = 0$) in this case.

Similarly, equation (3.3.27) is also trivial. If an update from ϕ_r to ϕ'_r is *valid* given $\mathcal{C}$, then the opposite update ϕ'_r to ϕ_r given the same clusterization is also *valid*, because it's not possible for two points in a cluster to change so that $\phi'_r(x) = \phi_{r,x}$ and $\phi'_{r,y} = -\phi_{r,y}$, if the cluster changed it's sign $\phi_{r,x} = -\phi'_{r,x}$ and $\phi_{r,y} = -\phi'_{r,y}$, or if the whole cluster left as it is.

To prove equation (3.3.24), first, let's write down what is the probability of a sign change $\phi_{r,x} \mapsto -\phi_{r,x} = \phi'_{r,x}$ for one γ cluster.

$$\frac{\exp\left(-\sum\limits_{x\in\gamma} h_{r,x}\ (-\phi_{r,x})\right)}{\exp\left(-\sum\limits_{x\in\gamma} h_{r,x}\ \phi_{r,x}\right) + \exp\left(\sum\limits_{x\in\gamma} h_{r,x}\ \phi_{r,x}\right)}. \tag{3.3.49}$$

To get the probability, that the cluster is left as it is $\phi_{r,x} \mapsto \phi_{r,x} = \phi'_{r,x}$, we have to subtract this from 1,

$$\frac{\exp\left(-\sum\limits_{x\in\gamma} h_{r,x}\ \phi_{r,x}\right)}{\exp\left(-\sum\limits_{x\in\gamma} h_{r,x}\ \phi_{r,x}\right) + \exp\left(\sum\limits_{x\in\gamma} h_{r,x}\ \phi_{r,x}\right)}. \tag{3.3.50}$$

Notice, that in both cases, this is equal to

$$\frac{\exp\left(-\sum\limits_{x\in\gamma} h_{r,x}\ \phi'_{r,x}\right)}{\exp\left(-\sum\limits_{x\in\gamma} h_{r,x}\ \phi'_{r,x}\right) + \exp\left(\sum\limits_{x\in\gamma} h_{r,x}\ \phi'_{r,x}\right)}. \tag{3.3.51}$$

To get the probability that the whole lattice changes to ϕ'_r, these probabilities have to be multiplied together for all the clusters, since these are independent events.

$$\mathcal{Q}\big((\phi_r, \mathcal{C}) \mapsto \phi'_r\big) = \prod_{\substack{\gamma \\ \text{clusters in } \mathcal{C}}} \frac{\exp\left(-\sum\limits_{x\in\gamma} h_{r,x}\ \phi'_{r,x}\right)}{\exp\left(-\sum\limits_{x\in\gamma} h_{r,x}\ \phi'_{r,x}\right) + \exp\left(\sum\limits_{x\in\gamma} h_{r,x}\ \phi'_{r,x}\right)}. \tag{3.3.52}$$

The numerator is exactly $\exp(-S^{\mathrm{E}})$,

$$\prod_{\substack{\gamma \\ \text{clusters in } \mathcal{C}}} \exp\left(-\sum_{x\in\gamma} h_{r,x}\ \phi'_{r,x}\right) = \tag{3.3.53}$$

$$= \exp\left(-\sum_{\substack{\gamma \\ \text{clusters in } \mathcal{C}}} \sum_{x\in\gamma} h_{r,x}\ \phi'_{r,x}\right) = \exp(-S^{\mathrm{E}}(\phi'_r)), \tag{3.3.54}$$

and the denominator is invariant under the change $\phi'_r \mapsto -\phi_r$ or $\phi'_r \mapsto \phi_r$ if it happens simultaneously in a cluster for all points, that is: for *valid* updates. To finish proving equations (3.3.24), let's define

$$f_{\mathcal{Q}}(\phi_r, \mathcal{C}) := \prod_{\substack{\gamma \\ \text{clusters in } \mathcal{C}}} \frac{1}{\exp\left(-\sum_{x\in\gamma} h_{r,x}\ \phi_{r,x}\right) + \exp\left(\sum_{x\in\gamma} h_{r,x}\ \phi_{r,x}\right)}. \tag{3.3.55}$$

Then,

$$\mathcal{Q}\big((\phi_r, \mathcal{C}) \mapsto \phi'_r\big) = \exp(-S^{\mathrm{E}}(\phi'_r)) f_{\mathcal{Q}}(\phi_r, \mathcal{C}), \tag{3.3.56}$$

$$\mathcal{Q}\big((\phi'_r, \mathcal{C}) \mapsto \phi_r\big) = \exp(-S^{\mathrm{E}}(\phi_r)) f_{\mathcal{Q}}(\phi_r, \mathcal{C}), \tag{3.3.57}$$

which can be combined to get

$$\mathcal{Q}\big((\phi_r, \mathcal{C}) \mapsto \phi'_r\big) = \exp(-\Delta S^{\mathrm{E}}_r)\ \mathcal{Q}\big((\phi'_r, \mathcal{C}) \mapsto \phi_r\big). \tag{3.3.58}$$

Ergodicity

Ergodicity depends on the F function. Let's say, that $F = 0$. In this case, the probability, that two neighboring points are connected is $1 - F(\phi_{r,x}\phi_{r,y}) \exp(-\beta\phi_{r,x}\phi_{r,y}) = 1$. In every update step, the whole lattice will be one cluster, and every point changes signs together. Vectors don't change direction relative to their neighbors. This is clearly not ergodic, this will only generate a limited number of fields.

On the other hand, if $F > 0$, then $f_{\text{Wolff}}(\phi \mapsto \phi')$ is ergodic. To prove this, we are going to imagine a chain of steps from one arbitrary state to an other, where in each step only one point is flipped. Clearly, any nonzero probability state can be updated to any other by flipping each point one by one using a chosen set of r normal vectors alongside the flipping happens.

There is a nonzero probability, that each neighboring point is disconnected at each step, because the disconnection probability is $F(\phi_{r,x}\phi_{r,y}) \exp(-\beta\phi_{r,x}\phi_{r,y}) > 0$. Each point can now be viewed separately. Now, there is also a nonzero probability, that only the point we want is flipped, because equation (3.3.15) tells us that this value can never be zero, nor one. It is possible to flip through each point one by one to update from one arbitrary field to an other by carefully selecting the r vectors each step. It is highly unlikely, that the vectors will be generated randomly by the algorithm so that it is the same as we would like them to be, but since every r vector has the same probability to be chosen, it is still possible[9].

Now, I have just proven, that from each nonzero probability state the algorithm can update to any other nonzero probability state, so the algorithm is ergodic, if $F > 0$. Maybe this requirement is too strict, and there can be other F where ergodicity is true, but I'm not concerned about it in this document, I only used update functions where $F(x) > 0$.

3.3.5 Wolff algorithm with updating one cluster

In the previous section every cluster was updated. In practice it is much easier to just update one cluster with each application of f_{Wolff}.

The algorithm is slightly modified. The only difference is in bold compared to the previous method.

1. Generate a random unit vector r with uniform probability on a 1 sphere,
2. connect each neighboring points x, y with probability
$$1 - F(\phi_{r,x}\ \phi_{r,y}) \exp(-\beta\ \phi_{r,x}\ \phi_{r,y}), \tag{3.3.59}$$
where F can be any function which has the properties that $0 < F(x) \leq \exp(\beta x)$ and $F(x) = F(-x)$. Two points are in one cluster if there is a path connecting them.
3. **Pick a random point on the field, and update the cluster it belongs to** with probability
$$\frac{\exp\left(\sum\limits_{x\in\gamma} h_{r,x}\ \phi_{r,x}\right)}{\exp\left(-\sum\limits_{x\in\gamma} h_{r,x}\ \phi_{r,x}\right) + \exp\left(\sum\limits_{x\in\gamma} h_{r,x}\ \phi_{r,x}\right)}. \tag{3.3.60}$$
If the cluster is updated, each field value in it is modified as $\phi_{r,x} \mapsto -\phi_{r,x}$, otherwise the cluster is left as it is.
4. Do previous steps again from the beginning with as much random unit vectors perpendicular to each other as possible.

[9]Remember, that I'm still assuming here, that r vectors are discretized, and every probability density has been discretized to a probability.

After the clusterization process, only one cluster is updated, therefore, it is unnecessary to calculate all clusters. In fact, in practice, the same process can be simulated by first picking a random point, and then growing the connections from there, because the connection probability is the same if we first pick the starting point and then create the cluster. However, here I will assume that each cluster is generated, and then I pick a point, because it is easier to prove detailed balance in this case.

Proof of the algorithm

The proof is exactly the same as for the previous case except the last part, where I proved equation (3.3.24) for *valid* updates.

Note, that *valid* still means the same here. The definition introduced in the paragraph **Clusterization** in section 3.3.4 is so generic, that nothing has to be changed in that proof for equation (3.3.23).

There are 3 types of cases for we have to calculate $\mathcal{Q}$. The first type is when two or more clusters in clusterization $\mathcal{C}$ are flipped in ϕ'_r compared to ϕ_r. Then, $\mathcal{Q}\big((\phi_r, \mathcal{C}) \mapsto \phi'_r\big) = 0$, because this is impossible. Note, that this is symmetric is we swap ϕ_r to ϕ'_r, so in this case, the following equation is true with both sides being equal to 0.

$$\mathcal{Q}\big((\phi_r, \mathcal{C}) \mapsto \phi'_r\big) = \exp(-\Delta S_r^{\mathrm{E}})\ \mathcal{Q}\big((\phi'_r, \mathcal{C}) \mapsto \phi_r\big). \tag{3.3.61}$$

Only one or zero cluster can be changed in one update step.

The second type is when $\phi_r = \phi'_r$. Here, $\Delta S_r^{\mathrm{E}} = 0$, and $\mathcal{Q}$ are the same on both sides,

$$\mathcal{Q}\big((\phi_r, \mathcal{C}) \mapsto \phi_r\big) = \underbrace{\exp(-\Delta S_r^{\mathrm{E}})}_{=1}\ \mathcal{Q}\big((\phi_r, \mathcal{C}) \mapsto \phi_r\big), \tag{3.3.62}$$

which means, that that equation (3.3.24) is satisfied for this type of updates too.

The third type is when one and only one cluster is flipped. The probability that this happens is the probability that the chosen point is in cluster γ – which is the flipped cluster –, and that the flip happens.

$$\mathcal{Q}\big((\phi_r, \mathcal{C}) \mapsto \phi'_r\big) = \frac{|\gamma|}{V} \frac{\exp\left(\sum\limits_{x\in\gamma} h_{r,x}\ \phi_{r,x}\right)}{\exp\left(-\sum\limits_{x\in\gamma} h_{r,x}\ \phi_{r,x}\right) + \exp\left(\sum\limits_{x\in\gamma} h_{r,x}\ \phi_{r,x}\right)}, \tag{3.3.63}$$

where $|\gamma|$ is the number of points in the chosen cluster, and V is the number of points on the whole lattice. Notice, that the denominator, and the probability to choose γ cluster does not change for this type of update when we swap ϕ_r and ϕ'_r.

Which means, that

$$\mathcal{Q}\big((\phi_r,\mathcal{C}) \mapsto \phi'_r\big) = \exp\left(\sum_{x\in\gamma} h_{r,x}\ \phi_{r,x}\right) \frac{\mathcal{Q}\big((\phi'_r,\mathcal{C}) \mapsto \phi_r\big)}{\exp\left(\sum_{x\in\gamma} h_{r,x}\ \phi'_{r,x}\right)}. \tag{3.3.64}$$

Also, for this type of update $-\phi_{r,x} = \phi'_{r,x}$ for $x \in \gamma$,

$$\exp(-\Delta S^{\mathrm{E}}_r) = \exp\left(2\sum_{x\in\gamma} h_{r,x}\ \phi_{r,x}\right). \tag{3.3.65}$$

Combining these last two equations we get, that equation (3.3.24) is true for this last type too

$$\mathcal{Q}\big((\phi_r,\mathcal{C}) \mapsto \phi'_r\big) = \exp(-\Delta S^{\mathrm{E}}_r)\ \mathcal{Q}\big((\phi'_r,\mathcal{C}) \mapsto \phi_r\big). \tag{3.3.66}$$

Ergodicity can be proven exactly the same way as in section 3.3.4.

3.3.6 Shadow algorithm

In the Wolff algorithm, there is an external field, and each point is affected by it. It is analogous to a theory where there is no external field, but every point has one more neighbor, which is the same magnitude as the previous external field. This theory is not new, and not my original idea, but I didn't find the algorithm and it's details in the literature, I had to figure out the details in this work. For convenience I named it after how the connections look schematically, the "shadow" is just all the connections to the external point. The algorithm can simulate an inhomogeneous field as before, if each lattice point is connected differently to the external point, but, for simplicity, I'm simulating a homogeneous field here.

The points are connected just as before with one extra connection to a new point outside of the lattice, which has a value h/β. The steps are the same to the previous Wolff algorithm too, except for the 2nd and 3rd point. I highlighted in bold the difference.

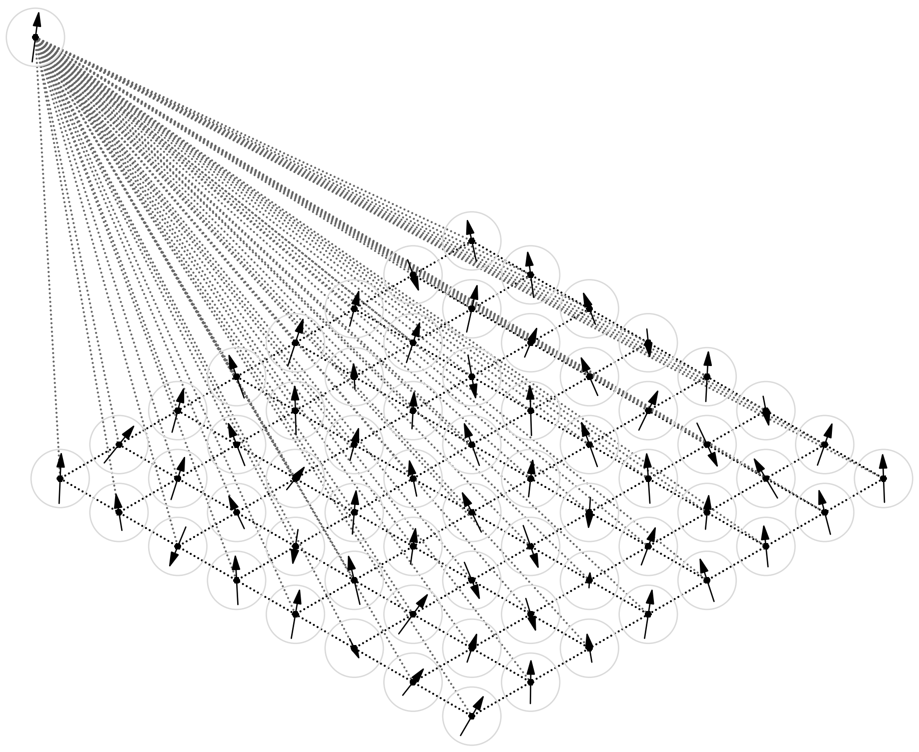


Figure 4: Because every lattice point has the external point as it's neighbor, they are all connected to it. It looks like the external point casts a shadow to the lattice, that's the origin of the name.

1. Generate a random unit vector r with uniform probability on a 1 sphere,

2. connect each neighboring points x, y on the lattice with probability

$$1 - F(\phi_{r,x}\ \phi_{r,y}) \exp(-\beta\ \phi_{r,x}\ \phi_{r,y}), \tag{3.3.67}$$

   and connect the point on the lattice with the external point with probability

$$1 - F(\phi_{r,x}\ \tilde{h}_r) \exp(\beta \phi_{r,x}\ \tilde{h}_r), \tag{3.3.68}$$

$$\tilde{h} := h/\beta, \tag{3.3.69}$$

   where F can be any function which has the properties that $0 < F(x) \leq \exp(\beta x)$ and $F(x) = F(-x)$. Two points are in one cluster if there is a path connecting them.

3. **Pick a random point on the field, and update the cluster it belongs to, if the external point belongs to the cluster.**
   If the cluster is updated, each field value in it is modified as $\phi_{r,x} \mapsto -\phi_{r,x}$, otherwise the cluster is left as it is.

4. Do previous steps again from the beginning with as much random unit vectors perpendicular to each other as possible.

Notice, that the external point never changes.

In a later chapter I will compare Wolff ans Shadow algorithms in practice.

Proof of the algorithm

The beginning of the proof is exactly the same as the proof for the Wolff algorithm up until the equation (3.3.21), which says, that

$$W_f(\phi_r \mapsto \phi'_r) = \sum_{\mathcal{C}} \mathcal{P}\big(\phi_r \mapsto (\phi_r, \mathcal{C})\big)\ \mathcal{Q}\big((\phi_r, \mathcal{C}) \mapsto \phi'_r\big). \tag{3.3.70}$$

I will prove, that

$$\mathcal{P}\big(\phi_r \mapsto (\phi_r, \mathcal{C})\big) = \exp(-\Delta S_r)\ \mathcal{P}\big(\phi'_r \mapsto (\phi'_r, \mathcal{C})\big), \tag{3.3.71}$$

$$\mathcal{Q}\big((\phi_r, \mathcal{C}) \mapsto \phi'_r\big) = \mathcal{Q}\big((\phi'_r, \mathcal{C}) \mapsto \phi_r\big), \tag{3.3.72}$$

$$\Delta S_r := S(\phi'_r) - S(\phi_r), \tag{3.3.73}$$

if the update $\phi_r \mapsto \phi'_r$ is *valid* with clusterization $\mathcal{C}$. Here, *valid* still means the same, including the external point, which is the neighbor of all the points in the lattice. Also, for an *invalid* update, $\mathcal{Q} = 0$, as before, and

$$(\phi_r, \mathcal{C}) \mapsto \phi'_r \text{ is } \textit{valid} \Leftrightarrow (\phi'_r, \mathcal{C}) \mapsto \phi_r \text{ is } \textit{valid}. \tag{3.3.74}$$

Also note, that the external point does not change, because only those clusters flip, which do not have this external point in them.
Very similarly to the Wolff proof, if these previous equations are true, then, for the Shadow algorithm

$$W_f(\phi_r \mapsto \phi'_r) = \tag{3.3.75}$$

$$= \sum_{\textit{valid } \mathcal{C}} \mathcal{P}\big(\phi_r \mapsto (\phi_r, \mathcal{C})\big)\mathcal{Q}\big((\phi_r, \mathcal{C}) \mapsto \phi'_r\big) +$$

$$\underbrace{+ \sum_{\textit{invalid } \mathcal{C}} \mathcal{P}\big(\phi_r \mapsto (\phi_r, \mathcal{C})\big)\mathcal{Q}\big((\phi_r, \mathcal{C}) \mapsto \phi'_r\big)}_{=0, \text{ because } \mathcal{Q}=0} = \tag{3.3.76}$$

$$= \exp(-\Delta S_r) \sum_{\textit{valid } \mathcal{C}} \mathcal{P}\big(\phi'_r \mapsto (\phi'_r, \mathcal{C})\big)\mathcal{Q}\big((\phi'_r, \mathcal{C}) \mapsto \phi_r\big) = \tag{3.3.77}$$

$$= \exp(-\Delta S_r)\ W_f(\phi'_r \mapsto \phi_r). \tag{3.3.78}$$

To prove equation (3.3.71), we can to calculate the probability of any clusterization the same way as before, including the external point. Keep in mind, they are connected with probability $1 - F(\phi_{r,x}\phi_{r,y})\exp(-\beta\phi_{r,x}\phi_{r,y})$, and they are disconnected with probability $F(\phi_{r,x}\phi_{r,y})\exp(-\beta\phi_{r,x}\phi_{r,y})$.

As before, $\mathcal{C}^{\mathrm{c}}$ means the connected pair of points, and $\mathcal{C}^{\mathrm{d}}$ the disconnected pair of points, and let's say, that $\mathcal{C}_{\mathrm{I}}$ means the lattice connections/disconnections, and $\mathcal{C}_{\mathrm{E}}$ means the points connected/disconnected between the external point and a lattice point. Then,

$$\mathcal{P}\big(\phi_r \mapsto (\phi_r, \mathcal{C})\big) = \tag{3.3.79}$$

$$= \prod_{(x,y)\in\mathcal{C}_{\mathrm{I}}^{\mathrm{c}}} [1 - F(\phi_{r,x}\phi_{r,y}) \exp(-\beta\phi_{r,x}\phi_{r,y})] \prod_{(z,w)\in\mathcal{C}_{\mathrm{I}}^{\mathrm{d}}} F(\phi_{r,z}\phi_{r,w}) \exp(-\beta\phi_{r,z}\phi_{r,w}) \times$$

$$\times \prod_{x\in\mathcal{C}_{\mathrm{E}}^{\mathrm{c}}} \Big[1 - F(\phi_{r,x}\tilde{h}_r) \exp(\beta\phi_{r,x}\tilde{h}_r)\Big] \prod_{z\in\mathcal{C}_{\mathrm{E}}^{\mathrm{d}}} F(\phi_{r,z}\tilde{h}_r) \exp(\beta\phi_{r,z}\tilde{h}_r). \tag{3.3.80}$$

As previously, we can use the fact, that $\prod_{(i,j)\in\mathcal{C}_{\mathrm{I}}} = \prod_{(i,j)\in\mathcal{C}_{\mathrm{I}}^{\mathrm{c}}} \times \prod_{(k,l)\in\mathcal{C}_{\mathrm{I}}^{\mathrm{d}}}$ to get

$$\prod_{(z,w)\in\mathcal{C}_{\mathrm{I}}^{\mathrm{d}}} F(\phi_{r,z}\phi_{r,w}) \exp(-\beta\phi_{r,z}\phi_{r,w}) = \prod_{(z,w)\in\mathcal{C}_{\mathrm{I}}^{\mathrm{d}}} F(\phi_{r,z}\phi_{r,w}) \prod_{(i,j)\in\mathcal{C}_{\mathrm{I}}^{\mathrm{d}}} \exp(-\beta\phi_{r,i}\phi_{r,j}) = \tag{3.3.81}$$

$$= \prod_{(z,w)\in\mathcal{C}_{\mathrm{I}}^{\mathrm{d}}} F(\phi_{r,z}\phi_{r,w}) \prod_{(i,j)\in\mathcal{C}_{\mathrm{I}}^{\mathrm{c}}} \exp(+\beta\phi_{r,i}\phi_{r,j}) \underbrace{\prod_{(k,l)\in\mathcal{C}_{\mathrm{I}}} \exp(-\beta\phi_{r,k}\phi_{r,l})}_{\exp(S^{\mathrm{I}}(\phi_r))} = \tag{3.3.82}$$

$$= \exp(S^{\mathrm{I}}(\phi_r)) \prod_{(z,w)\in\mathcal{C}_{\mathrm{I}}^{\mathrm{d}}} F(\phi_{r,z}\phi_{r,w}) \prod_{(i,j)\in\mathcal{C}_{\mathrm{I}}^{\mathrm{c}}} \exp(+\beta\phi_{r,i}\phi_{r,j}). \tag{3.3.83}$$

Similarly, for the $\mathcal{C}_{\mathrm{E}}^{\mathrm{d}}$ product

$$\prod_{z\in\mathcal{C}_{\mathrm{E}}^{\mathrm{d}}} F(\phi_{r,z}\tilde{h}_r) \exp(\beta\phi_{r,z}\tilde{h}_r) = \prod_{z\in\mathcal{C}_{\mathrm{E}}^{\mathrm{d}}} F(\phi_{r,z}\tilde{h}_r) \prod_{i\in\mathcal{C}_{\mathrm{E}}^{\mathrm{d}}} \exp(\beta\phi_{r,i}\tilde{h}_r) = \tag{3.3.84}$$

$$= \prod_{z\in\mathcal{C}_{\mathrm{E}}^{\mathrm{d}}} F(\phi_{r,z}\tilde{h}_r) \prod_{i\in\mathcal{C}_{\mathrm{E}}^{\mathrm{c}}} \exp(-\beta\phi_{r,i}\tilde{h}_r) \underbrace{\prod_{k\in\mathcal{C}_{\mathrm{E}}} \exp(\beta\phi_{r,k}\tilde{h}_r)}_{\exp(S^{\mathrm{E}}(\phi_r))} = \tag{3.3.85}$$

$$= \exp(S^{\mathrm{E}}(\phi_r)) \prod_{z\in\mathcal{C}_{\mathrm{E}}^{\mathrm{d}}} F(\phi_{r,z}\tilde{h}_r) \prod_{i\in\mathcal{C}_{\mathrm{E}}^{\mathrm{c}}} \exp(-\beta\phi_{r,i}\tilde{h}_r). \tag{3.3.86}$$

Let's use these results to get

$$\mathcal{P}\big(\phi_r \mapsto (\phi_r, \mathcal{C})\big) = \tag{3.3.87}$$

$$= \exp(S^{\mathrm{I}}(\phi_r)) \exp(S^{\mathrm{E}}(\phi_r)) \times \tag{3.3.88}$$

$$\times \prod_{(x,y)\in\mathcal{C}_{\mathrm{I}}^{\mathrm{c}}} [\exp(\beta\phi_{r,x}\phi_{r,y}) - F(\phi_{r,x}\phi_{r,y})] \prod_{(z,w)\in\mathcal{C}_{\mathrm{I}}^{\mathrm{d}}} F(\phi_{r,z}\phi_{r,w}) \times \tag{3.3.89}$$

$$\times \prod_{x\in\mathcal{C}_{\mathrm{E}}^{\mathrm{c}}} \Big[\exp(-\beta\phi_{r,x}\tilde{h}_r) - F(\phi_{r,x}\tilde{h}_r)\Big] \prod_{z\in\mathcal{C}_{\mathrm{E}}^{\mathrm{d}}} F(\phi_{r,z}\tilde{h}_r). \tag{3.3.90}$$

The last four products are symmetric under the change $\phi \to \phi'$, because for a *valid* update connected points change sign together, or they don't even change at all, and for disconnected points, only a sign change happens potentially, and F is symmetric. Therefore

$$\mathcal{P}\big(\phi_r \mapsto (\phi_r, \mathcal{C})\big) = \tag{3.3.91}$$

$$= \underbrace{\exp(S^{\mathrm{I}}(\phi_r))\exp(S^{\mathrm{E}}(\phi_r))}_{\exp(S(\phi_r))} f_{\mathcal{P}}(\phi_r, \mathcal{C}), \tag{3.3.92}$$

$$f_{\mathcal{P}}(\phi_r, \mathcal{C}) = f_{\mathcal{P}}(\phi'_r, \mathcal{C}). \tag{3.3.93}$$

It is very easy to get equation (3.3.71) from this.

$$\mathcal{P}\big(\phi_r \mapsto (\phi_r, \mathcal{C})\big) = \underbrace{\exp(S(\phi_r))\exp(-S(\phi'_r))}_{\exp(-\Delta S_r)}\underbrace{\exp(S(\phi'_r)) f_{\mathcal{P}}(\phi_r, \mathcal{C})}_{\mathcal{P}(\phi_r \mapsto (\phi_r, \mathcal{C}))}. \tag{3.3.94}$$

Now, the only thing left is to prove is equation (3.3.72) for the detailed balance. There are three cases for a *valid* update. The first case is when the cluster which the external point belongs to is flipped, or there are multiple clusters flipped between ϕ_r and ϕ'_r. $\mathcal{Q}$ is 0 in this case, on both sides. The second case is when $\phi_r = \phi'_r$. This can only happen if the chosen cluster is the one where the external point is. This case is obviously symmetric to the swap ϕ_r, ϕ'_r. The last case is when only one cluster is flipped, which does not have the external point in it. Because in this case the cluster is flipped with 1 probability regardless what are the field values, this is also symmetric to the swap ϕ_r, ϕ'_r.

Ergodicity can be proven the same way as before. Imagining a carefully selected chain of steps, where the vectors r are just as we would like them to be, and only flipping one point at the time, without any connections. This is still possible, if $F > 0$.

3.4 Continuum limit

In practice, it is only possible to calculate values on a discrete, finite lattice, but we would like to examine the expectation values of quantities which are as close to the continuum limit lattice as possible. To do that, I'm going to extrapolate the quantities with $(a/L)^2 \to 0$. This is not a straightforward task, because to extrapolate the discrete beta function for a specific g_2, one has to consider

$$\lim_{(a/L)^2 \to 0} \frac{g_2(2L/a, \beta) - g_2(L/a, \beta)}{\ln(2)}. \tag{3.4.1}$$

I can only measure any quantity using L/a, β and h as input parameters, and g_2 changes with L/a too.

3.4.1 Case $h = 0$

In case $h = 0$, first, I will decide for which g_2 values I would like to extrapolate the discrete beta function at $(a/L)^2 \to 0$. Let's call these points g_2^i.

Next, I'm going to calculate the quantities $g_2(L/a, \beta)$, $(g_2(2L/a) - g_2(L/a))/\ln(2)$ for a fixed L/a, and a range of β. This gives me the discrete beta function for a range of g_2, and I'm able to fit a polynomial on this.

Then, I can interpolate $(g_2(2L/a) - g_2(L/a))/\ln(2)$ for g_2^i values. This process is applied to different L/a. From this, it is possible to extrapolate the discrete beta function at each g_2^i.

In practice, I used 50 g_2^i values, and extrapolated $(g_2(2L/a) - g_2(L/a))/\ln(2)$ pointwise, and plotted the result. g_2^i are sufficiently close to each other, no other interpolation is needed at the end. See section **6. Results** for examples and plots.

3.4.2 Case $h \neq 0$

When $h \neq 0$, I used the same method, but only after h is renormalized using equation (3.2.9) for a chosen $\bar{R}$.

$$M_0(L/a, \beta, h)^{1/2} \left(\frac{L}{a}\right)^2 h = R(L/a, \beta, h). \tag{3.4.2}$$

The aim is to calculate $g_2(L/a, \beta, \bar{h})$, where $R(L/a, \beta, \bar{h}) = \bar{R}$.

$R(L/a, \beta, h)$ and $g_2(L/a, \beta, h)$ is measured for multiple h, and both are fitted with polynomials. $\bar{h}$ is calculated inverting the fitted $R(L/a, \beta, h)$ where $R(L/a, \beta, \bar{h}) = \bar{R}$, and $g_2(L/a, \beta, \bar{h})$ is calculated by interpolating the fitted polynomial at $\bar{h}$. I will assume here, that $\bar{h}$ doesn't have any errors, but g_2 has. See section **6.4. Discrete beta function for** $R = 10$ for examples and plots.

3.4.3 Fitting, extrapolation, interpolation

For the continuum limit calculation I need to extrapolate and interpolate values. The implementations for these are based on a code my supervisor gave me. For the purposes of these calculations, fitting, extrapolation, interpolation are the same algorithms, and work based on the same method [9].

Let's say, that there are N number of $(x_i, y_i, \Delta y_i)$ points, and we would like to fit a $y = f(x)$ function, with a D degree polynomial

$$f(x) = \sum_{k=0}^{D} c_k x^k. \tag{3.4.3}$$

The coefficients, and their error are calculated the following way.

$$A = \sum_{i=1}^{N} \frac{1}{(\Delta y_i)^2} \begin{pmatrix} x_i^0 x_i^0 & x_i^0 x_i^1 & \dots & x_i^0 x_i^D \\ x_i^1 x_i^0 & x_i^1 x_i^1 & \dots & x_i^1 x_i^D \\ \vdots & \vdots & \ddots & \vdots \\ x_i^D x_i^0 & x_i^D x_i^1 & \dots & x_i^D x_i^D \end{pmatrix}, \tag{3.4.4}$$

$$b = \sum_{i=1}^{N} \frac{1}{(\Delta y_i)^2} \begin{pmatrix} x_i^0 \\ x_i^1 \\ \vdots \\ x_i^D \end{pmatrix}, \tag{3.4.5}$$

$$c_k = (A^{-1}b)_k, \tag{3.4.6}$$

$$\Delta c_k = \sqrt{A^{-1}_{k,k}}. \tag{3.4.7}$$

If we are only interested in the value of $f(x)$ for a given x, then the following can be used.

$$f(x) = \sum_k c_k x^k, \tag{3.4.8}$$

$$\Delta f(x) = \sqrt{\sum_{k=0}^{D} \sum_{l=0}^{D} x^k \, A^{-1}_{k,l} \, x^l}. \tag{3.4.9}$$

It is also important to measure χ^2/dof (dof is the degree of freedom). χ^2 divided by the degree of freedom tells us how the chosen model ($f(x)$ with D degrees) compares to the measured points.

$$\chi^2 = \sum_{i=1}^{N} \frac{(y_i - f(x_i))^2}{(\Delta y_i)^2}, \tag{3.4.10}$$

$$dof = N - D. \tag{3.4.11}$$

If the degrees of freedom is roughly equal to chi-squared ($\chi^2/dof \approx 1$), then we can say the model is satisfactory. On the other hand, if it is much less than one, the degrees of freedom supersede the errors of the fit, which means that either not enough points are measured, or the degree of f (D) is too high: if the degree of a polynomial is sufficiently high, the fitted f can be made arbitrarily close to the measured points, but it is not necessarily correct. If $\chi^2/dof \gg 1$, then the measurement errors are much less than the error of the fit. The model has to be adjusted.

SymPy is only used to invert matrix A. **NumPy** is used to take the root of the polynomial. This is used for the renormalization process, to determine the correct h.

4 Technical details, implementation

I created a program which generates fields, and output the calculated M_0 and M_2 for each field. The inputs are L/a, β, $|h|$. This program is written in C++11, because the performance is critical here, and I'm familiar with this language. The output can then be analysed to calculate the average of these values, and it's errors. This was done with Python 3 scripts, because there are available libraries which can be used, and performance is not critical for these calculations. I also wrote bash scripts which can be used to determine where are the input and output files for subtasks, and run the program or scripts with correct arguments.

In the following sections only parts of the implementation are shown, which have some significance.

4.1 Sigmamodel

I made the C++ sigmamodel source code very flexible and abstract, so that it's easy to modify, and to separate each algorithm and type from each other. This way someone who reads or modifies the code only has to understand small parts, instead of a huge interdependent mess. For example, the code which grows the clusters doesn't include how and with which probability points have to be connected, nor does it contain what are the neighbors, or what is the lattice configuration. These are all just input arguments for the function. The understanding of the function is much easier this way, in this abstract level. It also gave me the opportunity to quickly modify and experiment with the code, for example to have a 3 dimensional lattice instead of 2.

I used the `std::ranlux48` random engine in the code, however, it's very easy to change the random engine used by all the processes, because the random generator is passed as an argument to all the methods that need it. There are only a couple of predefined C++11 random number engines [8]. There is no measurable difference between `std::ranlux48` and `std::mt19937` engines in performance, because the other calculations take much more time than random generation in the program.

In the actual program, external field corresponds to $-h$, there is a -1 multiplier compared to how it was introduced in the previous sections. This is only a convention. The input of the program only depends on the magnitude of h, not it's direction, and we are not using any direction specific information from the output, so it makes no difference, but important to note this here to avoid confusion.

4.1.1 Configuration

The first abstraction is the `Point`. It's only property by itself is that it has a value, and it has neighbors as pointers[10] to other `Points`. This is an abstract class with the point type (`Value_type`) as template parameter. The `set_neighbor` function just simply inserts the new point to the `set` of `m_neighbors`.

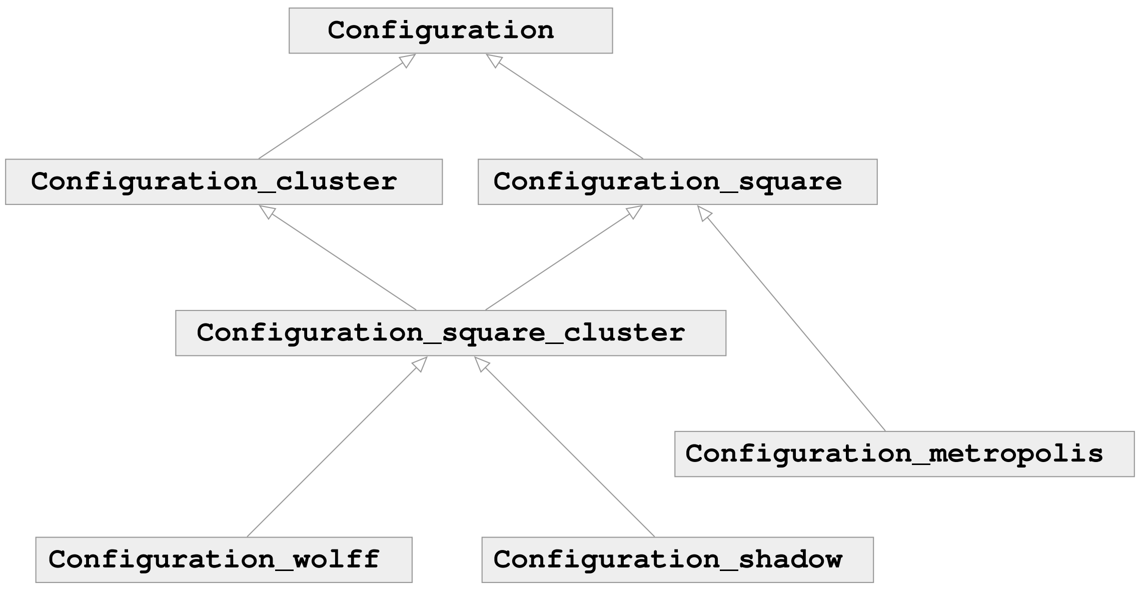
Point
- m_value : Value_type
- m_neighbors : set<shared_ptr<Point>>
+ value() : Value_type&
+ neighbors() : set<shared_ptr<Point>>
- set_neighbor(neighbor : shared_ptr<Point>)

In the source code the lattice with all it's points is called `Configuration`. The abstract interface class `Configuration` only implements just one simple utility function, `set_neighbor`, which calls `Point::set_neighbor` for both points, so that they both know that they are neighbors. The `update` function is a pure virtual function, that has to be implemented by anyone who implements this class.

<<interface>>
Configuration
+ update()
+ set_neighbor(point_0 : shared_ptr<Point>,
point_1 : shared_ptr<Point>)

A `Configuration` can be a square lattice, or a lattice which can grow clusters. For example, the Metropolis algorithm uses a `Configuration` which is going to be a square lattice, but it will not use clusters. Let's separate these two properties into two different classes.

[10] I'm using `shared_ptr<>` whenever I need a pointer. In my program all the pointers are shared across different parts of the code, and this is a much safer and reliable way to handle these than raw pointers.

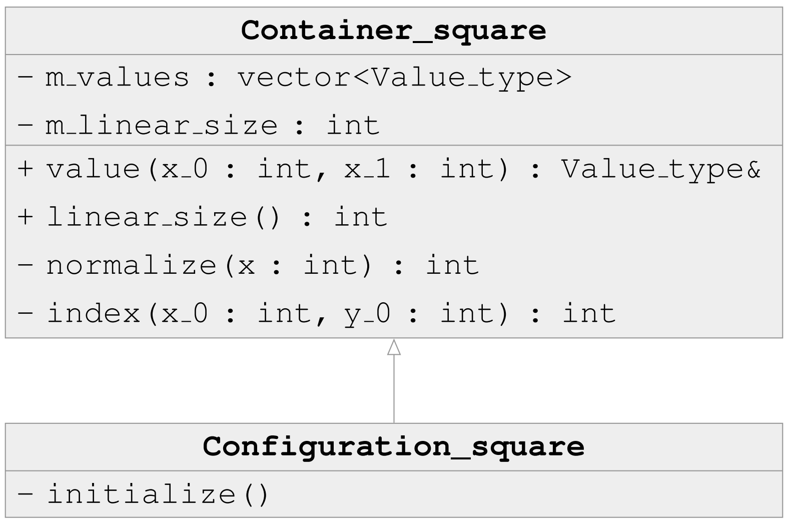


`Configuration_square` allocates memory for the square lattice, and sets up all the neighbors with the correct boundary conditions. `Configuration_cluster` implements a utility function, which can be used to grow a cluster.

There are other classes in the actual program which relate to `Configuration`, and sometimes the inheritence is not as straighforward as in the figure. Also, sometimes for the sake of clarity I didn't write out that arguments and return values are references, or that there are other template parameters. These are not important details, but I mention this for the sake of completeness.

4.1.2 Square lattice generation

The square lattice memory allocation is done in the class `Container_square`, and the class `Configuration_square` handles that the neighbors are connected correctly.


Container_square
- m_values : vector<Value_type>
- m_linear_size : int
+ value(x_0 : int, x_1 : int) : Value_type&
+ linear_size() : int
- normalize(x : int) : int
- index(x_0 : int, y_0 : int) : int
Configuration_square
- initialize()

Here, I only want to highlight three functions from the `Container_square` class. The `m_values` is just simply a `list`, which has $(L/a)^2$ (`linear_size*linear_size`) elements. As we can see, when I would like to obtain the element at x_0, x_1 position, first, the index function translates this to the range $[0, (L/a)^2 - 1]$. This makes sure, that the lattice's boundary condition is periodic.

```
Value_type& Container_square::value(int x_0, int x_1)
{
    return m_values.at(index(x_0, x_1));
}

int Container_square::normalize(int x) const
{
    if (x < 0)
        x += linear_size() * (1 - (x / linear_size()));
    return x % linear_size();
}

int Container_square::index(int x_0, int x_1) const
{
    return normalize(x_0) * linear_size() + normalize(x_1);
}
```

The class Configuration_square initializes the neighbors so that only the ones which are one lattice spacing apart are neighbors. It calls into Container_square's functions to obtain a Point at a specific position.

```
void Configuration_square::initialize()
{
    ...

    for (int x = 0; x != linear_size(); ++x)
    {
        for (int y = 0; y != linear_size(); ++y)
        {
            set_neighbor(value(x, y), value(x + 1, y));
            set_neighbor(value(x, y), value(x, y + 1));
        }
    }
}
```

4.1.3 Cluster building

The Configuration_cluster implements the utility function build_cluster, which grows a cluster given a starting point and a functor, which determines whether two neighbors should be connected or not.

```
                 Configuration_cluster
# build_cluster(
      starting_point : shared_ptr<Point>,
      connect_functor : Connect_functor&)
       : set<shared_ptr<Point>>
- build_cluster(
      starting_point : shared_ptr<Point>,
      connect_functor : Connect_functor&
      cluster : set<shared_ptr<Point>>&)
       : set<shared_ptr<Point>>
```

The first (protected #) is just a wrapper function. It just calls the second recursive `build_cluster` function.

```cpp
set<shared_ptr<Point<Value_type>>>
    Configuration_cluster::build_cluster(
        const shared_ptr<Point<Value_type>>& starting_point,
        Connect_functor& connect_functor)
{
    set<shared_ptr<Point<Value_type>>> cluster;
    build_cluster(starting_point, connect_functor, cluster);
    return cluster;
}
```

The recursive function goes through all the `starting_point`'s neighbors, and if the `connect_functor` returns with `true`, then they are connected: the neighbor is also going to be part of the cluster. From this new point, the cluster continues to grow recursively. If a neighbor is already part of the cluster, or the `connect_functor` decides the neighbors should not be connected, the function stops.

```cpp
void Configuration_cluster::build_cluster(
    const shared_ptr<Point<Value_type>>& point,
    Connect_functor& connect_functor,
    set<shared_ptr<Point<Value_type>>>& cluster)
{
    cluster.insert(point);
    for (auto neighbor : point->neighbors())
        if (cluster.count(neighbor) == 0)
            if (connect_functor(point->value(), neighbor->value()))
                build_cluster(neighbor, connect_functor, cluster);
}
```

Tail recursivity

For recursive functions, it is important to keep in mind, that the stacktrace can overflow. If there is something left that needs to be calculated after the recursive call, the program needs to keep the state of that execution step, with all of it's variables', because the program will potentially return to that location after the recursive call finishes. If there are too many recursive calls, all of these states cannot be stored, because there might be not enough memory. The solution is to rewrite the function as a tail-recursive one. The drawback is that the tail-recursive implementation is slower in this case. Fortunately, I didn't run into memory allocation problems in practice, but I still provided the tail-recursive implementation.

Every recursive algorithm can be rewritten as a tail-recursive one by cleverly adding and modifying it's arguments.

The idea is that the new `edges` variable stores all of neighbors, with the following property: the edge's first point is inside the current cluster, and the second is outside the current cluster. We can loosely say, the `edges` variable stores the neighbors at the boundary of the current cluster. `build_cluster` first goes through all of these neighbors, and potentially connects them. If a connection is made, it stores all of this new points' neighbors in a new variable, called `new_edges`. When this whole procedure finishes, the cluster has grown, and all of it's new neighbors are stored in the `new_edges` variable. The cluster grow can happen all over again.

For the sake of clarity, I removed some of the type definitions in this next code section.

```
void Configuration_cluster::build_cluster(
    shared_ptr<Point<Value_type>> point,
    Connect_functor& connect_functor,
    set<shared_ptr<Point<Value_type>>>& cluster)
{
    list<pair<...>> new_edges;
    new_edges.push_back(pair<...>(nullptr, point));
    build_cluster(new_edges, connect_functor, cluster);
}
void Configuration_cluster::build_cluster(
    const list<pair<...>>& edges,
    Connect_functor& connect_functor,
    set<shared_ptr<Point<Value_type>>>& cluster)
{
    list<pair<...>> new_edges;
    for (auto edge : edges)
        if (edge.first == nullptr ||
            connect_functor(edge.first->value(), edge.second->value()))
        {
            cluster.insert(edge.second);
            for (auto neighbor : edge.second->neighbors())
                if (cluster.count(neighbor) == 0)
                    new_edges.push_back(
                        pair<...>(edge.second, neighbor));
        }

    if (new_edges.empty())
        return;
    return build_cluster(new_edges, connect_functor, cluster);
}
```

The compiler optimizes the binary, so that it is tail-recursive. gcc needs `return` at the end of the function even if it "returns" `void`, to optimize it to a tail-recursive function.

I also put the tail-recursive and the other implementation in an `#ifdef` directive, so that it is easy to compile with or without it using a compilation option.

Connecting neighbors for a cluster

As we saw earlier, both Wolff and the Shadow algorithm connects the lattice points with the same algorithm. The connection probability for these points are

$$1 - F(\phi_{r,x}\phi_{r,y}) \exp(-\beta\phi_{r,x}\phi_{r,y}). \tag{4.1.1}$$

Also, in case of the Shadow algorithm, external point is connected with probability

$$1 - F(\phi_{r,x}\tilde{h}_r) \exp(\beta\phi_{r,x}\tilde{h}_r), \tag{4.1.2}$$

$$\tilde{h} = h/\beta. \tag{4.1.3}$$

Apart from the minus sign, $\tilde{h}$ connects with the same probability as a lattice point. The external point can be implemented exactly the same way as any lattice point. The only difference regarding cluster building is that this new external point has to be added with magnitude h/β, and it has to be set as neighbor to all lattice points.

The actual implementation of the `build_cluster` functor depends on F. I will present the source code in a different section later.

4.1.4 Random vector generation

In all three algorithms (Metropolis, Wolff and Shadow) we need to generate random unit vectors. To achieve this, first, I'm going to generate a random point in a cube $x \in [-1, 1]$, $y \in [-1, 1]$ and $z \in [-1, 1]$. If the generated point is outside of a 1 unit radius solid sphere, then I throw away the point, and generate a new one. I generate a new point until I get one which is inside of this solid sphere. The probability that this point is inside of the sphere for a given V volume is the same in any direction. This vector is then normalized.

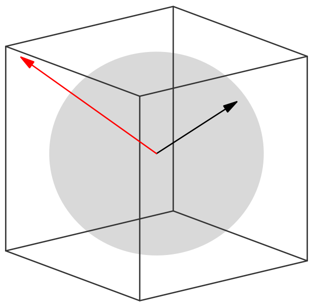


Figure 5: Only those vectors are returned (after normalization) which are inside the unit sphere (black vector), the ones which are outside are discarded (red vector).

```
Vector3 random_vector_in_one_sphere(Random_generator& random_generator)
{
    uniform_real_distribution<> distribution(-1.0, 1.0);
    Vector3 v({
            distribution(random_generator),
            distribution(random_generator),
            distribution(random_generator)});
    if (vector_length(v) > 1.0)
        return random_vector_in_one_sphere(random_generator);
    return v;
}

Vector3 random_normalized_vector(Random_generator& random_generator)
{
    Vector3 v = random_vector_in_one_sphere(random_generator);
    if (vector_length(v) < epsilon)
        return random_normalized_vector(random_generator);
    return vector_normalize(v);
}
```

`epsilon` is a very small value (0.00000001 in the code). The algorithm generates a new vector if the vector's magnitude is smaller than this, so that we avoid dividing two very small numbers. It is necessary, because dividing by a small number which are represented in the computer can cause big errors.

Random perpendicular vectors

A set of random and perpendicular vector generation is also needed. This is done by generating the first vector with the previous method. The subsequent one is calculated by normalizing the cross product of the first, and a new random unit vector. The third vector is calculated by the cross product of the first and second one, and multiplying this with -1 with a 0.5 probability, so that the three vectors don't always have the same handedness.

```
array<Vector3, 3>
    random_normalized_perpendicular_vectors(
        Random_generator& random_generator)
{
    Vector3 r_0 = random_normalized_vector(random_generator);

    Vector3 r_n = random_normalized_vector(random_generator);
    Vector3 r_1 = vector_normalize(vector_cross(r_0, r_n));

    Vector3 r_2 = vector_normalize(vector_cross(r_0, r_1));
    uniform_int_distribution<> distribution(0, 1);
    if (distribution(random_generator))
        r_2 = -1.0 * r_2;

    return {r_0, r_1, r_2};
}
```

4.1.5 Metropolis configuration

Let's quickly remind ourselves what is the Metropolis algorithm. It picks a random point x on the lattice, and we generate a random unit vector φ'. The $\phi(x)$ is switched to φ' with probability

$$\min(1, \exp(-\Delta S)), \tag{4.1.4}$$

$$\Delta S = \left[-\beta \sum_y \phi(y) + h \right] (\varphi' - \phi(x)). \tag{4.1.5}$$

In the code, `r_0` corresponds to $\phi(x)$, `r_1` to φ' and `ds` to ΔS.

```
void Configuration_metropolis::update()
{
    // Picking a random point on the L/a * L/a lattice
    uniform_int_distribution<> distribution_linear_size(0,
            linear_size() - 1);
    auto p = value(distribution_linear_size(m_random_generator),
            distribution_linear_size(m_random_generator));

    // Storing the value of the point and it's local action
    Vector3 r_0 = p->value();
    double local_action_0 = get_local_action(*p);

    // Generating a value for the potential update,
    // and calculating it's local action
    Vector3 r_1 = random_normalized_vector(...);
    p->value() = r_1;
    double local_action_1 = get_local_action(*p);

    // Calculating the difference of the action,
    // and updating with min(1, exp(-ds)) probability
    uniform_real_distribution<> distribution_1(0.0, 1.0);
    double ds = local_action_1 - local_action_0;
    if (min(1.0, exp(-ds)) < distribution_1(...))
        p->value() = r_0;
}

double Configuration_metropolis::get_local_action(Point<Vector3>& p)
{
    double local_action = 0.0;
    for (auto& neighbor : p.neighbors())
        local_action -= m_beta * p.value() * neighbor->value();
    local_action -= p.value() * m_external_field;
    return local_action;
}
```

4.1.6 Wolff configuration

The Wolff algorithm builds a cluster, and then updates it for 3 different perpendicular, but random vectors in each step. After the γ cluster is built, the flip probability is

$$\frac{n}{n+1}, \tag{4.1.6}$$

$$n := \exp\left(2h_r \sum_{x\in\gamma} \phi_r\right). \tag{4.1.7}$$

In the following code, $\sum_{x\in\gamma} \phi_r$ is `cluster_magnetization_r`,
and n is `flip_probability_nominator`. Of course, the flip function is $\phi(x) \mapsto \phi(x) - 2\ (\phi(x) \cdot r)\ r$. These vectors are represented in the computer with finite precision, and this function can potentially change it's magnitude, therefore, a normalization is needed after the flip to make sure that the field vectors stay unit vectors.

```cpp
void Configuration_wolff::update()
{
    array<Vector3, 3> r =
        random_normalized_perpendicular_vectors(m_random_generator);
    update_single_component(r[0]);
    update_single_component(r[1]);
    update_single_component(r[2]);
}

void Configuration_wolff::update_single_component(Vector3 r)
{
    Connect<Random_generator> connect(m_beta, r, m_random_generator);

    // Picking a random point on the L/a * L/a lattice
    uniform_int_distribution<>
        distribution_linear_size(0, linear_size() - 1);
    shared_ptr<Point<Vector3>> starting_point =
        value(distribution_linear_size(m_random_generator),
            distribution_linear_size(m_random_generator));

    // Building the cluster
    set<...> cluster = build_cluster(starting_point, connect);

    // Calculating the cluster's magnetization's component
    // with regards to r vector
    double cluster_magnetization_r = 0.0;
    for (const auto& point : cluster)
        cluster_magnetization_r += point->value() * r;

    // Calculating the probability for the flip
    double flip_probability_nominator =
        exp(-2.0 * (m_external_field * r) * cluster_magnetization_r);
    double flip_probability =
        flip_probability_nominator / (flip_probability_nominator+1.0);

    // Flipping the whole cluster potentially
    uniform_real_distribution<> distribution(0.0, 1.0);
    if (distribution(m_random_generator) < flip_probability)
        for (auto& point : cluster)
            point->value() =
                normalize(point->value()-(2.0*(r*point->value()))*r);
}
```

4.1.7 Shadow configuration

In the case of the Shadow algorithm, all the lattice points have one extra neighbor: the external point. The external point has a magnitude of h/β (because in the implementation h is multiplied with -1, and the external point behaves as a field multiplied by -1 regarding the connection probability, here, we have to write $+h$). The following code snippets make sure that this happens.

```
Configuration_shadow::Configuration_shadow
    (..., double beta, double external_field_length, ...) : ...
{
    m_external_point->value() =
        Vector3({0.0, 0.0, external_field_length / beta});
    set_neighbors_external_point();
}

void Configuration_shadow::set_neighbors_external_point()
{
    for (int x = 0; x != linear_size(); ++x)
        for (int y = 0; y != linear_size(); ++y)
            set_neighbor(value(x, y), m_external_point);
}
```

The Shadow algorithm's update is simpler than the Wolff algorithm, because the flip always happens, if the cluster does not contain the external point. Other than these, it's the exact same algorithm.

```
void Configuration_shadow::update()
{
    array<Vector3, 3> r =
        random_normalized_perpendicular_vectors(m_random_generator);
    update_single_component(r[0]);
    update_single_component(r[1]);
    update_single_component(r[2]);
}

void Configuration_shadow::update_single_component(Vector3 r)
{
    Connect<Random_generator> connect(m_beta, r, m_random_generator);

    // Picking a random point on the L/a * L/a lattice
    uniform_int_distribution<>
        distribution_linear_size(0, linear_size() - 1);
    shared_ptr<Point<Vector3>> starting_point =
        value(distribution_linear_size(m_random_generator),
            distribution_linear_size(m_random_generator));

    // Building the cluster
    set<...> cluster = build_cluster(starting_point, connect);

    // Flipping the whole cluster
    if (cluster.count(m_external_point) == 0)
        for (auto& point : cluster)
            point->value() =
                normalize(point->value()-(2.0*(r*point->value()))*r);
}
```

4.1.8 Serialization

The program needs to be able to save and load fields to and from files. This is especially important for *thermalization*, which will be discussed in a later section. The most convenient and fastest way is to store these in a binary structure. I made a very generic solution, so that to implement the store or load mechanism, only the necessary code has to be written. For example, for the `Vector` class, it has to store `dimension` number of it's components (`m_values`), the serialization is only a few lines

```
void Vector::write(std::ostream& os) const
    for (int i = 0; i != dimension; ++i)
        ::write(m_values[i], os);
void Vector::read(std::istream& is)
    for (int i = 0; i != dimension; ++i)
        ::read(m_values[i], is);
```

The global `write` and `read` functions figure out whether the variable that needs to be written/read (here `m_values[i]`) has a `read` or `write` member function. If yes, then that's used, but if not, then it will try to use the following method to read

```
is.read(reinterpret_cast<char*>(&value), sizeof(Value_type));
```

and the following to write

```
os.write(reinterpret_cast<const char*>(&value), sizeof(Value_type));
```

`reinterpret_cast` makes sure, that the value's starting memory address is selected as a pointer to a `char`, and the `*stream::read/write` reads/writes `sizeof(Value_type)` number of bytes.
For other complex types, like the square lattice, the `Point::read/write` function has to be called for all points, which in turn ultimately calls the `Vector::read/write`, which calls the `*stream::read/write` functions for doubles. In the program, the lattice size, β, the external field is also saved and loaded.

The global `read/write(value, ...)` functions has to figure out whether `value` has a `read/write` member function, so that they can determine whether to call those, or call the `*stream` directly. This is done using *SFINAE* (Substitution Failure Is Not An Error), which is a common trick to use in C++. This programming technique has less relevance in a physics thesis, but I find it worthwile to at least mention shortly how I used this in the program.

I created the `Has_write` class, where `Has_write<Value_type>::Has` is used to decide with which method to write: `ostream::write`, or call `Value_type::write`. If this is `std::true_type`, then the latter is used, otherwise, the former. From here, it's not very challenging to implement the global `write` function, I'm not detailing it here.

The interesting part is the `Has_write` class. `Has_write<Value_type>::Has` is declared as the return type of the `test<Value_type>(0)` in line 12. As we can see, `test` function is declared two times here. The compiler first tries to use the declaration in line 8. The return type here can be read from the second expression from the comma separated expressions in the argument of the `decltype`, which is `std::true_type`. Here, the first expression is only needed to force the compiler to try to make sense of this expression. If it fails, it will try to use the second declaration of the `test`, which is in line 11. This has a return type of `std::false_type`. The compiler will fail to make sense of the expression in line 9 (the first it tries), if `Value_type` does not have a member function with a specific name and argument type: `write(std::ostream&) const`.

```
1  template <typename Value_type>
2  class Has_write
3  {
4      Has_write();
5      template <typename T>
6          static void sfinae(void (T::*)(std::ostream&) const);
7      template <typename T>
8          static auto test(void*) ->
9              decltype(sfinae(&T::write), std::true_type());
10     template <typename T>
11         static std::false_type test(...);
12     typedef decltype(test<Value_type>(0)) Has;
13 };
```

4.2 Expectation value and error calculation

The program only outputs M_0 and M_2 values for each kth field. It is necessary, because if a new field is generated only a few steps after the original one, they are highly correlated, and it's not worth storing all data for each field, it takes up a lot of disk space, without new information. How many configurations are used and why will be discussed later.

The definition of moments from equations (2.1.12), (2.1.13) and (2.1.14) are

$$M_0 = \frac{1}{(L/a)^2} \sum_{x_0,x_1,i} \langle \phi_i(x_0,x_1)\phi_i(0,0)\rangle, \tag{4.2.1}$$

$$M_2 = \frac{1}{(L/a)^2}\frac{1}{(2\pi)^2} \sum_{x_0,x_1,i} (2\sin(\pi x_0/L))^2 \ \langle \phi_i(x_0,x_1)\phi_i(0,0)\rangle. \tag{4.2.2}$$

Very similarly, let's denote the moment for one field as

$$M_0(\phi) := \frac{1}{(L/a)^2} \sum_{x_0,x_1,i} \phi_i(x_0,x_1)\phi_i(0,0), \tag{4.2.3}$$

$$M_2(\phi) := \frac{1}{(L/a)^2}\frac{1}{(2\pi)^2} \sum_{x_0,x_1,i} (2\sin(\pi x_0/L))^2 \ \phi_i(x_0,x_1)\phi_i(0,0). \tag{4.2.4}$$

Because of linearity,

$$M_0 = \langle M_0(\phi)\rangle, \tag{4.2.5}$$

$$M_2 = \langle M_2(\phi)\rangle. \tag{4.2.6}$$

From here, it's easy to calculate g_2 using equation (2.1.25) from lots of measured fields, using the average. However, I cannot use the standard error propagation method to calculate it's error, because the values of the 2-point function at different space, time and component were calculated using the same fields, these values are correlated. It also follows, that M_0 and M_2 correlate too. I'm using the Jackknife method to calculate the error, which is a systematic way to obtain error of stachastic measurements [7].

The M_0, M_2 pairs for all fields are all divided into N blocks, so that the block length is much more than the correlation length between the generated fields. The averages of the moments for these blocks are M_0^n, M_2^n. Let's define $M_0^{(n)}$, $M_2^{(n)}$ as the average of the moments using all the blocks except the nth block.

$$M_0^{(n)} := \frac{1}{N-1} \sum_{n' \neq n} M_0^n, \tag{4.2.7}$$

$$M_2^{(n)} := \frac{1}{N-1} \sum_{n' \neq n} M_2^n. \tag{4.2.8}$$

Then, the error of any ρ function of the moments

$$\Delta\rho = \sqrt{\frac{N-1}{N} \sum_n \left(\rho(M_0^{(n)}, M_2^{(n)}) - \rho(M_0, M_2)\right)^2}. \tag{4.2.9}$$

The error of g_2 then has to be calculated with $\rho(M_0, M_2) = g_2(M_0, M_2)$, and the error for the moments are calculated using the identity functions ($\rho = M_0$ or $\rho = M_2$).

5 Practical analysis

In this section, I'm analysing the results of the actual calculations. There are multiple questions regarding the parameters, like what should F look like, which algorithms perform the better, or how many generated fields should be discarded, because most of them are correlated.

In this section $L/a = 8$, overall 1.5×10^7 fields were generated for the Metropolis (and from here every 10th measurement is discarded, because they are so highly correlated), and 10^6 fields were generated for the Wolff and Shadow algorithms (for the $F(x) = \exp(-\beta|x|)$ case, 2×10^6 fields). Jackknife calculations used blocks of 1000 samples.

5.1 Choosing F

The probability to connect two neighboring points when building the cluster is $p = 1 - F(\phi_{r,x}\phi_{r,y})\exp(-\beta\phi_{r,x}\phi_{r,y})$, $0 < F(x) \leq \exp(\beta x)$ and $F(x) = F(-x)$. I present 3 different functions and it's results. Figure 6 show plots of them.

All of these are implemented in the same manner. The connection is a functor which is passed to the cluster building algorithm as an argument.

```cpp
bool Connection::operator()(Vector3 point_0, Vector3 point_1)
{
    double x = (point_0 * m_r) * (point_1 * m_r);
    uniform_real_distribution<> distribution(0.0, 1.0);
    // Returns true if connection is made, false otherwise.
    return ...
}
```

$F(x) = \exp(-\beta|x|)$ results the connection probability to be zero when $\phi_{r,x}\phi_{r,y} \leq 0$. It is implemented the following way.

```cpp
    if (x < 0.0)
        return false;
    return distribution(m_random_generator) < 1.0 - exp(-2.0*m_beta*x);
```

$F(x) = 1/(\exp(-\beta x) + \exp(+\beta x))$ causes the connection probability to be the Fermi-Dirac distribution. It is implemented the following way.

```cpp
    return distribution(m_random_generator) <
        1.0 / (exp(-2.0*m_beta*x) + 1.0);
```

$F(x) = \exp(-1.1\beta|x|)$ has unusally high probability for connection when $\phi_{r,x}\phi_{r,y} \approx -1$. It is implemented the following way.

```cpp
    return distribution(m_random_generator) <
        1.0 - exp(-m_beta*(1.1*abs(x) + x));
```

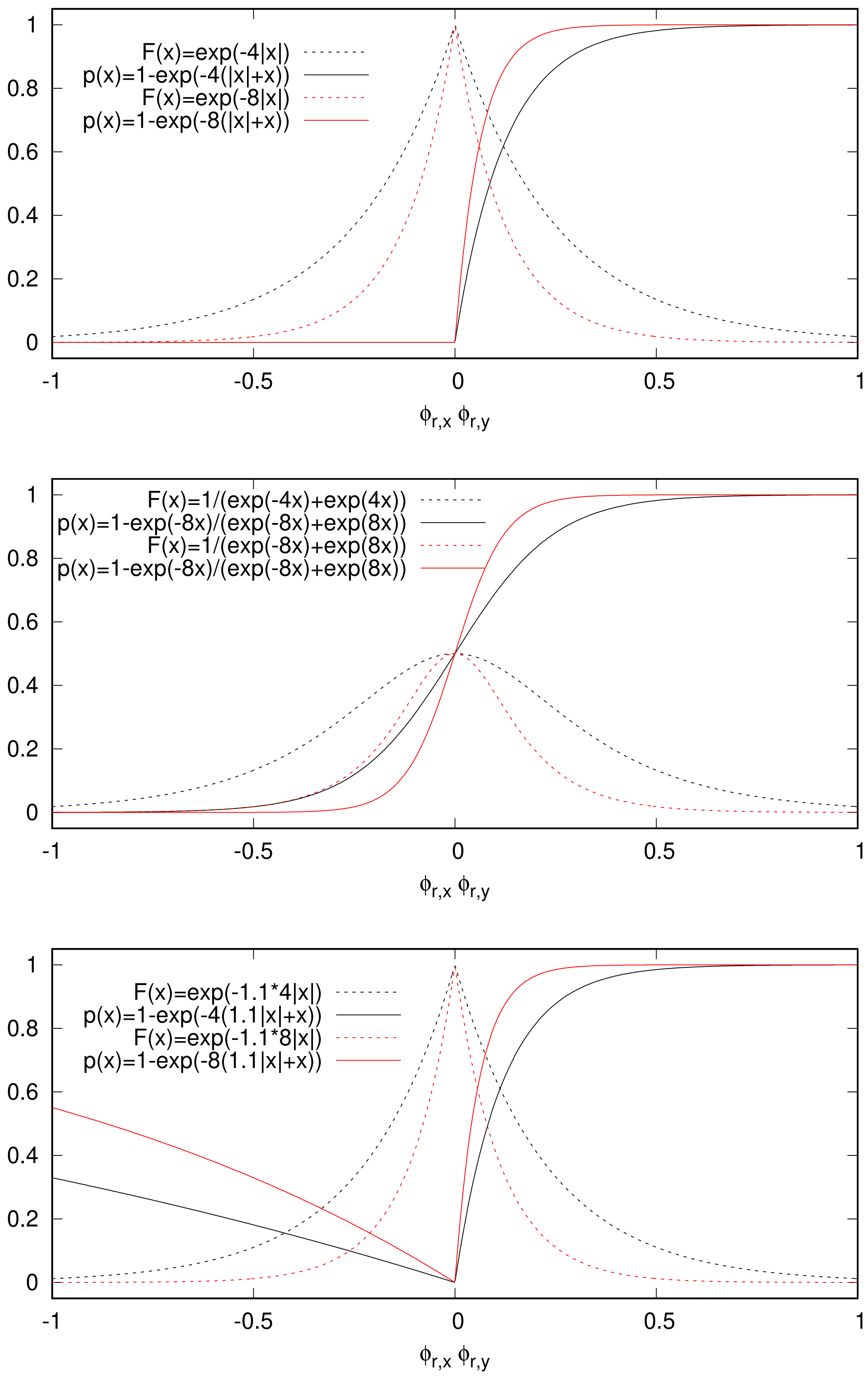


Figure 6: These $F(x)$ were tried. The plots also show the corresponding probability of connection p.

These different functions should behave the same way with regards to the averages, however, there are performance differences between them, which we will see. There are numerous other F which could work, this is not an exhaustive list, but I couldn't find any other (or variation of these) which performed better. For example the constant $F(x) = \exp(-\beta|x|)$ is extremely slow, it couldn't even thermalize (thermalization is discussed in a later section) while the other algorithms could give good results in the same timeframe for $h = 0$. It would be also possible to change F depending on β and h to maximize performance, but in this document I only picked one for all parameters.

5.2 Thermalization

The initial field I chose is that all field vectors are constant $(x, y, z) = (0, 0, 1)$. This is a highly unprobable configuration in most cases, which means, that at the beginning, the system is not in equilibrium. The first few measurements should be discarded until the equilibrium happens. By looking at the measurements, it's easy to determine when the system is in equilibrium i.e. *thermalized.*

Figure 7 shows the thermalization process for the Metropolis algorithm. As we can see, if we discard the first few samples, we can measure the averages for the equilibrium system. It would be impractical to show all thermalization graphs for every algorithm, F, β and h which I use in this section. What is important to note here is that Wolff and Shadow thermalize themselves much faster than the Metropolis, which means, that discarding the first[11] 40000 - for Wolff and Shadow - measurements works. Figure 8 show some examples of the thermalization for Wolff and Shadow algorithms.

[11]Note, that in case of Metropolis, every 10th element was discarded, and for the Wolff and Shadow in this section no measurement was discarded. Discarding the first 40000 elements for Wolff and Shadow is comparable to discarding the first 4000 measurements for Metropolis.

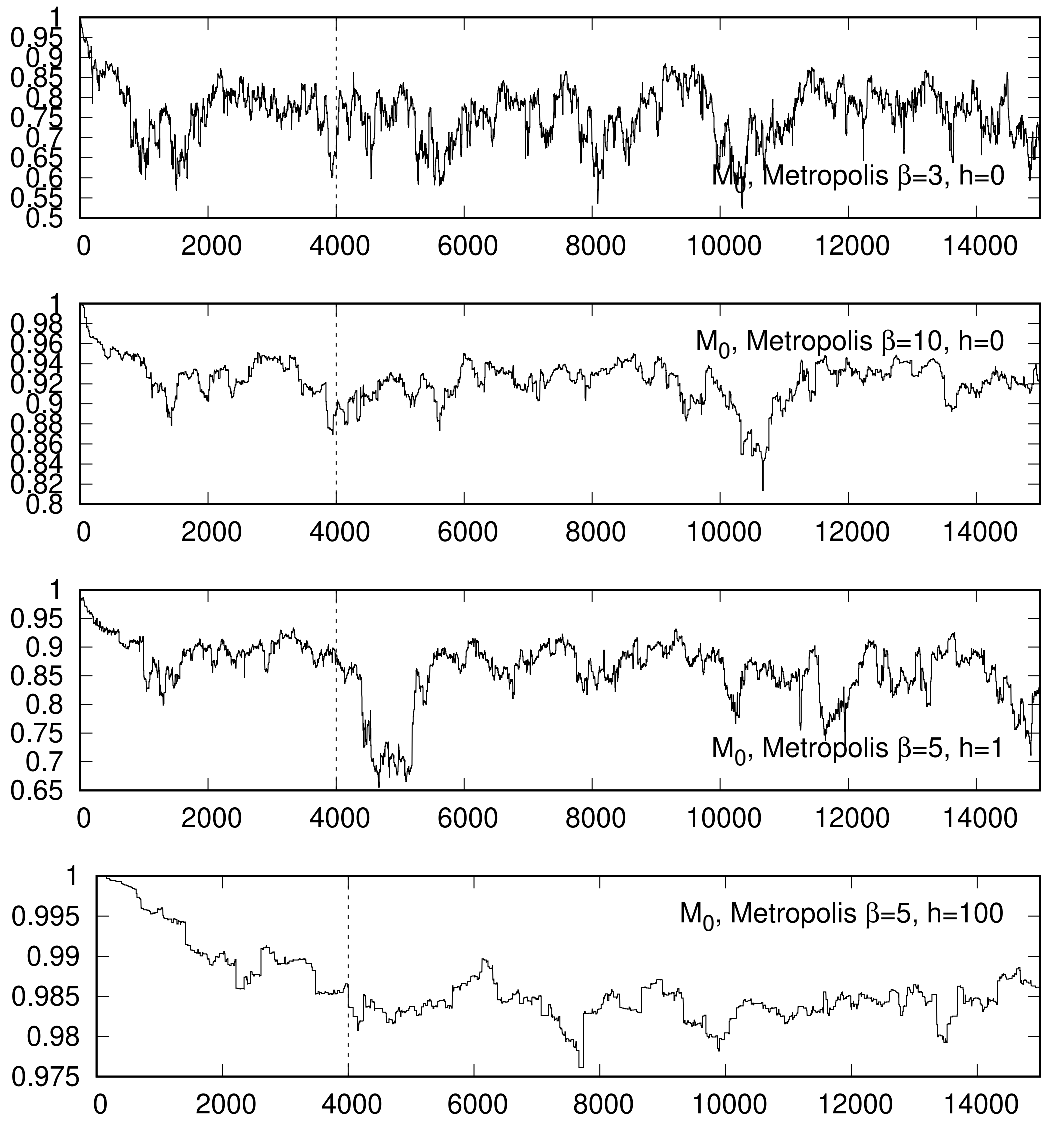


Figure 7: Thermalization for various parameters for the Metropolis algorithm. The thermalization length was determined to be 4000 (dashed line) by looking at the results. The first few fields have $M_0 \approx 1$, and gradually the system thermalizes itself.

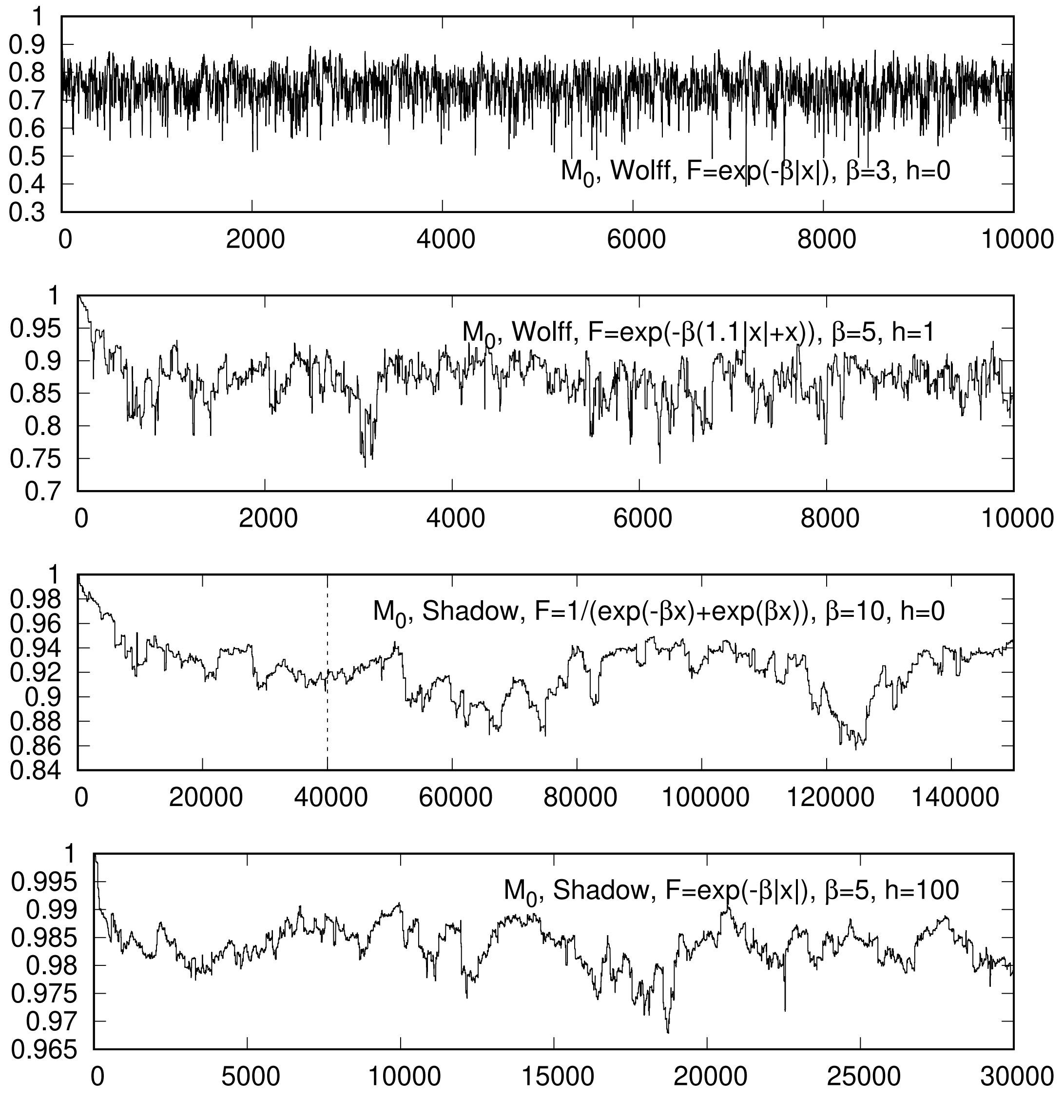


Figure 8: Thermalization for various algorithms and F. These are only examples where thermalization length 40000 is a good choice.

5.3 Consistency among algorithms

One of the most important things to verify is whether the different algorithms provide the same results. If they do, that's a strong indicator, that there is no mistake in the implementation.

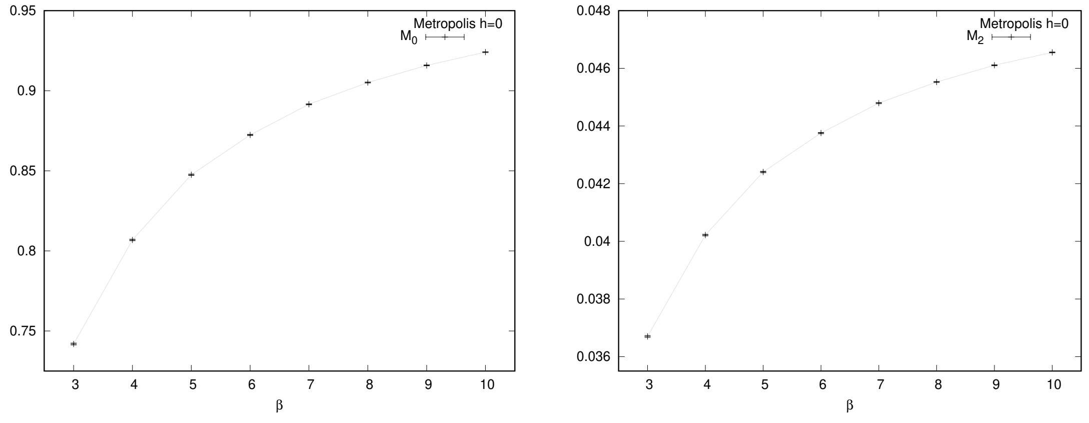


Figure 9: Moments plotted against β using Metropolis algorithm.

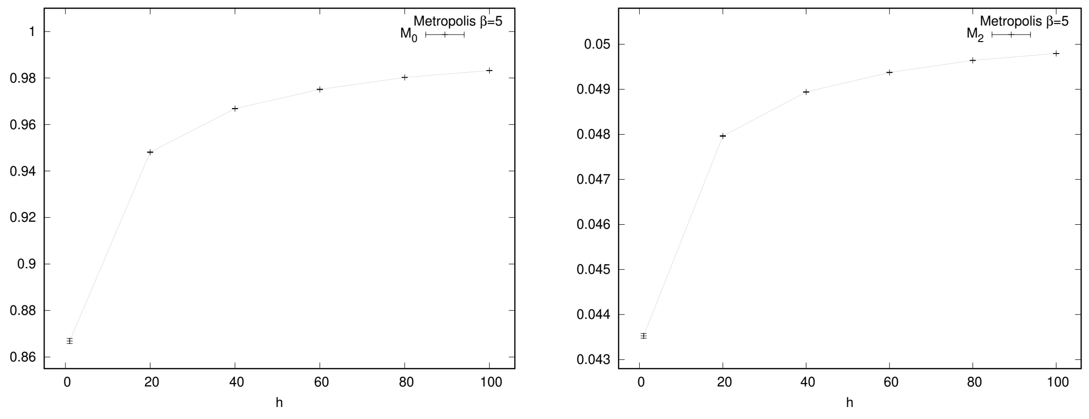


Figure 10: Moments plotted against h using Metropolis algorithm.

Figure 9 shows what are the moments as the function of β, and figure 10 shows the moments as the function of h. There is at least 10 % variation among the data, and the error is much less than the difference between the values.

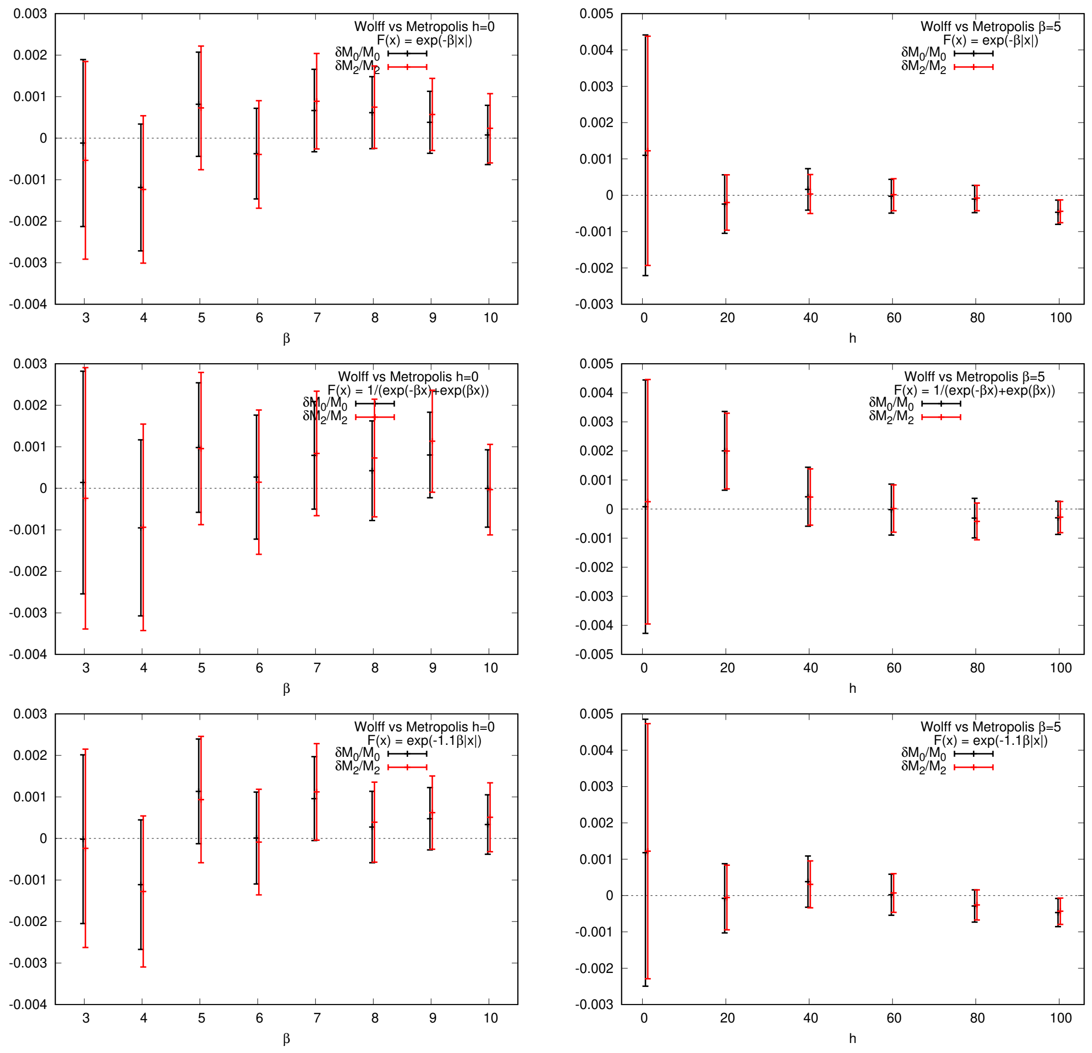


Figure 11: Measured $\delta M/M_{\text{Metropolis}}$, $\delta M = M_{\text{Wolff}} - M_{\text{Metropolis}}$, for M_0 and M_2, and it's error. Note, that the error shown in the plots are $2(\Delta M_{\text{Metropolis}} + \Delta M_{\text{Wolff}})$.

I measured the same values with the Wolff algorithm, and compared them to Metropolis. Figure 11 shows, that the relative discrepancy between them is in the order 10^{-3}, sometimes even less, and it is comparable to the measurement error. This is also much less, than the difference between the measurements with different parameters of $(\beta,\, h)$.

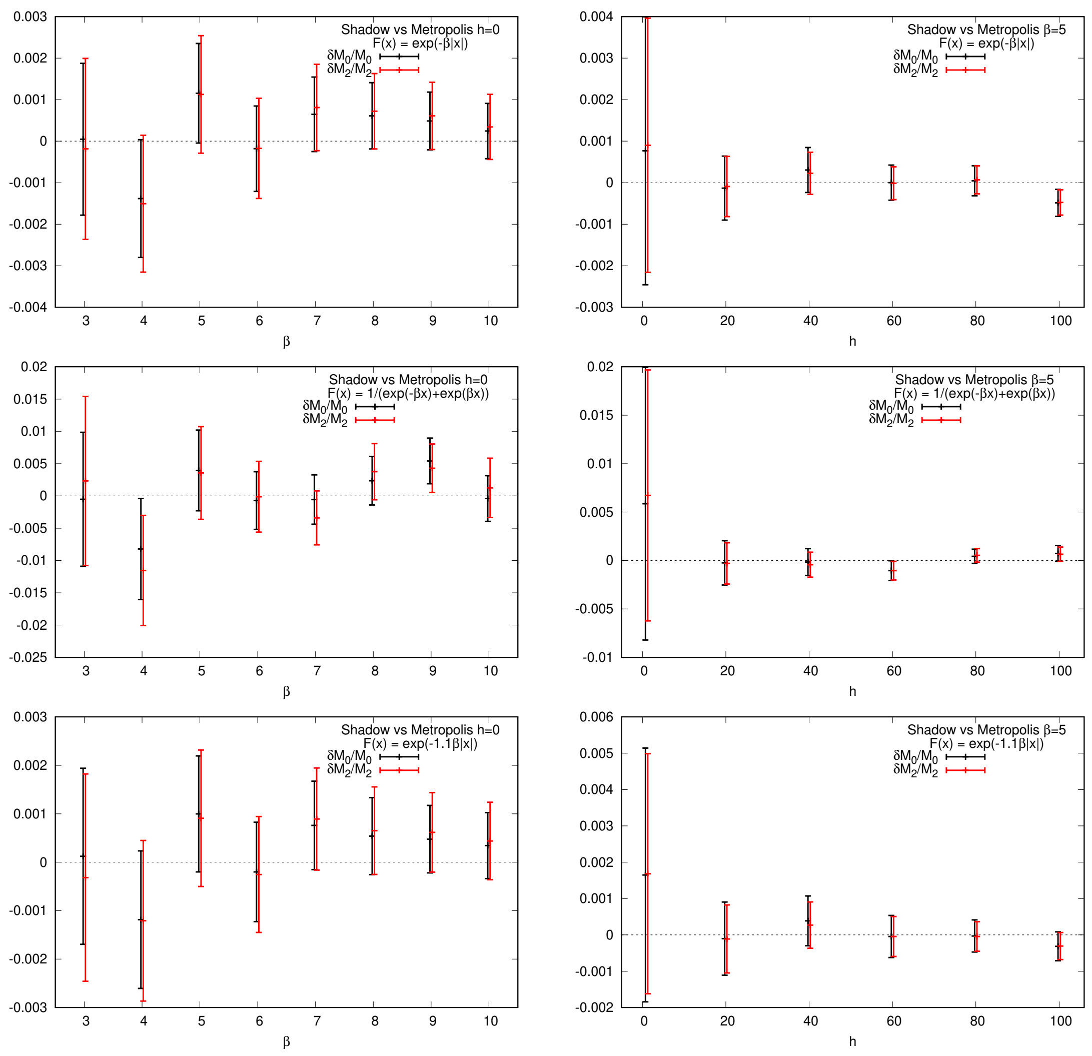


Figure 12: Measured $\delta M/M_{\text{Metropolis}}$, $\delta M = M_{\text{Shadow}} - M_{\text{Metropolis}}$, for M_0 and M_2, and it's error. Note, that the error shown in the plots are $2(\Delta M_{\text{Metropolis}}+\Delta M_{\text{Shadow}})$.

I also measured the same values with the Shadow algorithm, and compared them to Metropolis. Figure 12 shows the results. The same can be said here too. The relative discrepancy between them is again in the order 10^{-3}, and it is also comparable to the measurement error.

The results agree in the margin of error using different algorithms and different F. Also, the discrepancy is much less for the same parameters than the difference between moments using different parameters β, h. The conclusion is that all of these algorithms provide the same results, it is safe to use all of them.

5.4 Performance

The most practical algorithm and F is the one, which can generate the highest amount of uncorrelated fields in unit time. The time to generate one uncorrelated field is

$$\tau = \lambda \frac{T}{N}, \tag{5.4.1}$$

where N is the number of all the generated fields, and T is the time it took to generate them. λ is the number of fields needed to be generated to get an uncorrelated measurement, which is the autocorrelation length.

In reality, the generated fields do not become fully uncorrelated after λ steps, but rather, this is a continuous transition. They become gradually uncorrelated. Let's define the autocorrelation function from the measurements of M_0.

$$c_N(k) := \frac{\frac{1}{N-k}\sum_{i=1}^{N} M_0^k M_0^{i+k} - \langle M_0 \rangle^2}{\langle M_0^2 \rangle - \langle M_0 \rangle^2}. \tag{5.4.2}$$

If the sample size (N) goes to infinity, this function is an exponential according to [7], and from there, we can define λ as

$$\lim_{N\to\infty} c_N(k) = e^{-k/\lambda}. \tag{5.4.3}$$

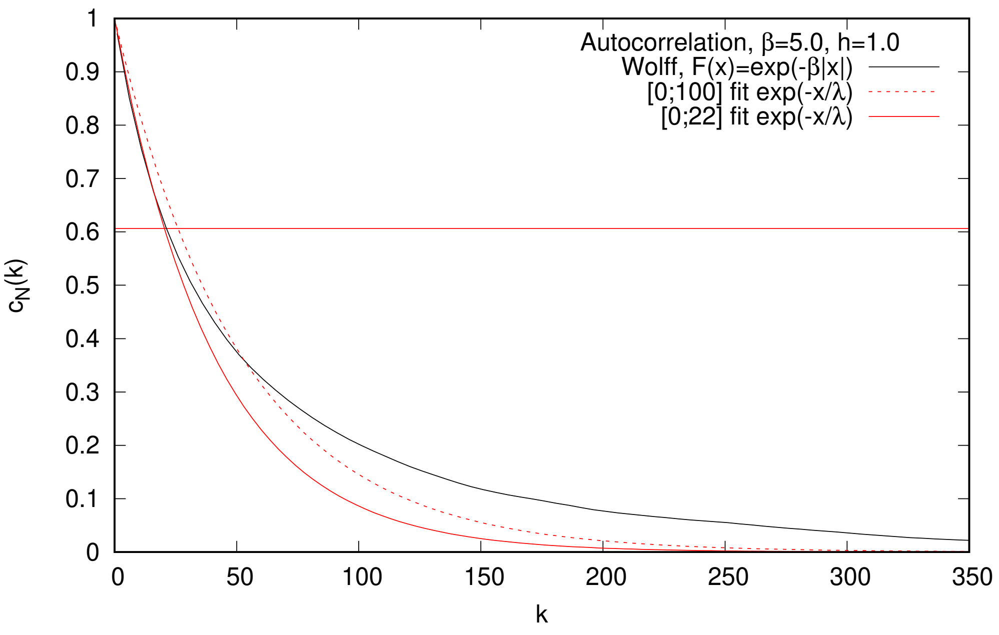


Figure 13: The horizontal line shows $\exp(-0.5)$ threshold. The measured (black) line intersects the threshold at $k = 22$, and the continuous red line is a fitted exponential function between $k = 0$ and $k = 22$. The dashed red line is a fit between the arbitrary $k = 0$ and $k = 100$. The continuous red fit is a better approximation for small k.

The measured $c_N(k)$ autocorrelation function is not an exponent, but a fairly good approximation only if k is small enough as we can see from figure 13. I define λ to be the result of the fit $\exp(-k/\lambda)$ between the range $k = 0$, and $k_{\max}$, where $c_N(k_{\max}) = \exp(-0.5)$. This can also be seen on figure 13.

Some other example measurements can be seen on figure 14. It is important not to conclude which algorithm is better based only on k, because this does not take into account how much time it takes to generate one field.

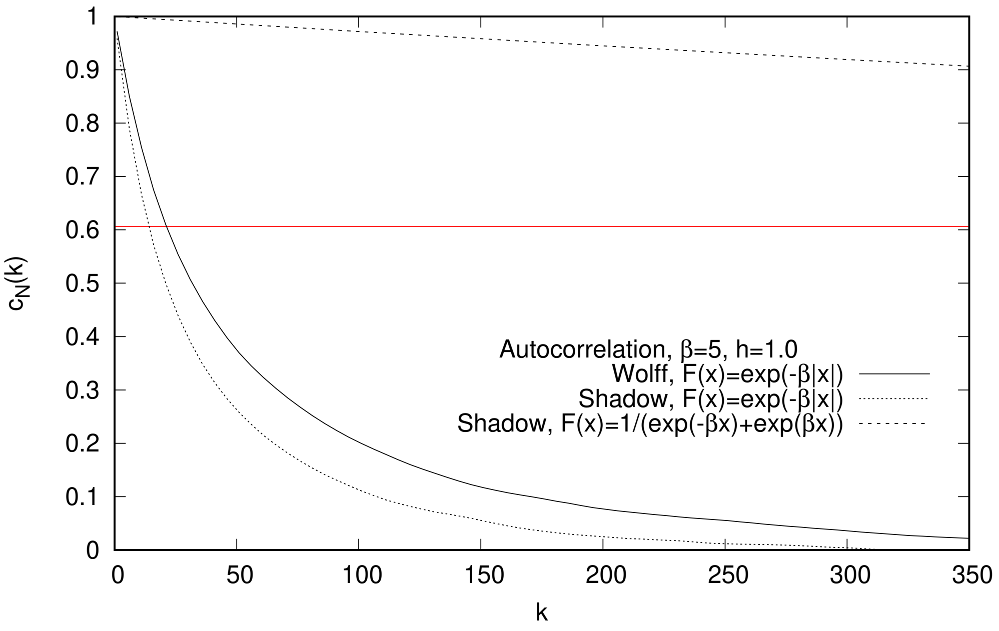


Figure 14: Some example autocorrelation measurements.

Choosing algorithm

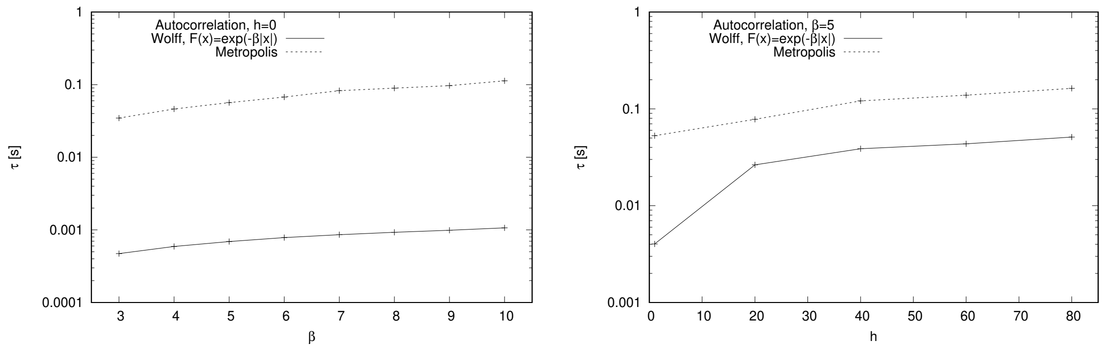


Figure 15: Comparing performance of Metropolis with Wolff.

Metropolis is the slowest algorithm as we can see from figure 15. This figure only shows $F(x) = \exp(-\beta|x|)$. Metropolis can be eliminated as the algorithm to use.

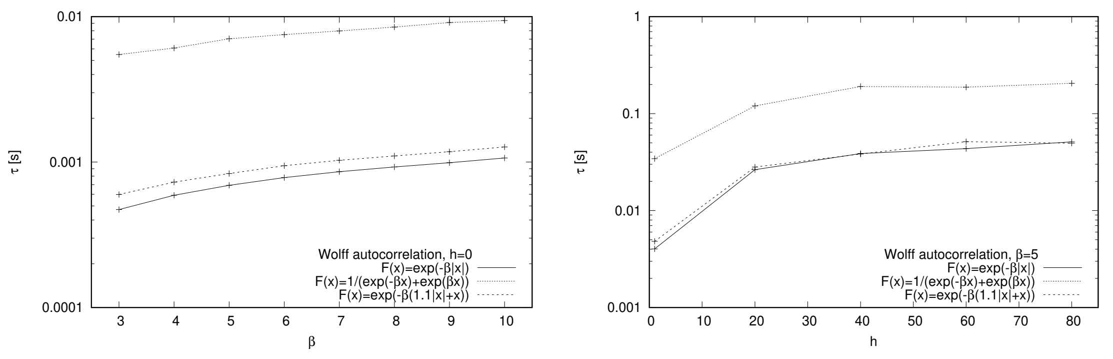


Figure 16: Comparing performance of different F for Wolff.

Figure 16 shows what's the performance of different F for Wolff. I chose $F(x) = \exp(-\beta|x|)$ out of these.

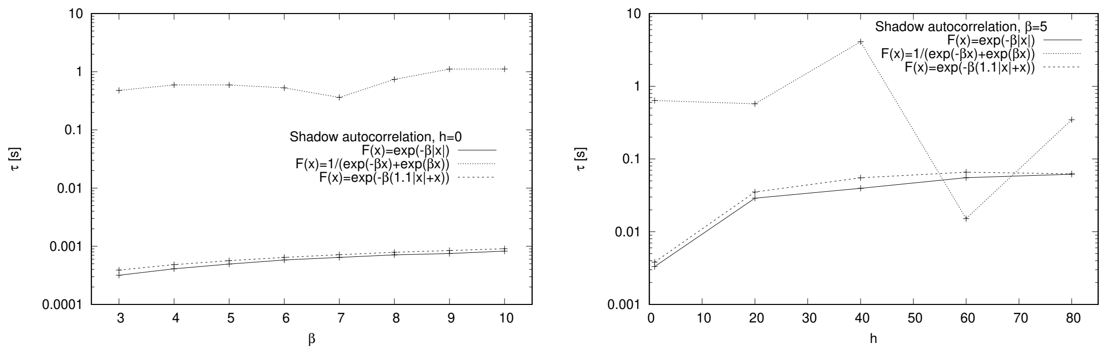


Figure 17: Comparing performance of different F for Shadow.

Figure 17 shows what's the performance of different F for Shadow. I also chose $F(x) = \exp(-\beta|x|)$.

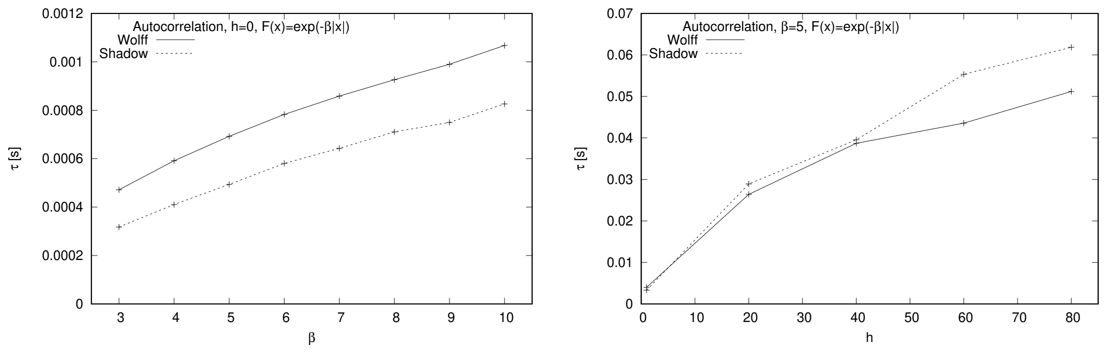


Figure 18: Comparing performance between Wolff and Shadow.

Figure 18 compares between Wolff and Shadow for the chosen $F(x) = \exp(-\beta|x|)$. It is not obvious what to choose, because for $h = 0$ Shadow performs better, and for $h > 10$, $\beta = 5$ Wolff performs better. Before I made detailed performance calculations, I already had lots of measurements for the Wolff, $F(x) = \exp(-\beta|x|)$ case, and for small h there is no big difference between Wolff and Shadow in the presented range, as it can be seen from the figures. I chose Wolff algorithm for further calculations.

Note, that these results are only to choose the calculation method, they cannot be regarded as rigorous tests between different algorithms regarding performance. I did not calculate errors, and did not take into account other parameter ranges. However, they can be used as a basis for further performance measurements.

All calculations were done on an Intel i5-7500 CPU at 3.40 GHz, and DDR4 2133 MHz RAM on Linux, compiled on gcc 9.3.0 with `-O2` optimization. The time measurements were done separately to the correlation measurements, without writing anything to disk, so that disk write time does not interfere with performance results.

6 Results

The discrete beta function can now be calculated for $h = 0$ and $h \neq 0$, and they can be compared to each other.

To not use up too much disk space, every nth field was saved (and used for calculations) only: a preliminary measurement measured k, where the autocorrelation at k is $c_N(k) = \exp(-0.5)$, and $n = 6 \times k$. This ensures, that no measurements were saved which are too correlated. Also, this is the definition of one sweep: every nth generated field is one sweep. This definition is only important to interpret **Data tables** section.

6.1 Discrete beta function for $h = 0$, $g_2 < 1$

Measuring the discrete beta function for small g_2 at $h = 0$ lets us compare results with the perturbation theory.

Figure 19 shows examples of the measurements. The results are very close to each other compared to it's errors. I fitted a third order polynomial on $(g_2(2L/a) - g_2(L/a))/\ln(2)$ for $L/a \in \{12, 16, 24, 32, 48\}$. (Note, that this also means, that I needed to calculate g_2 for $L/a \in \{64, 96\}$.) Also, for the fit, I added manually the point for the discrete beta function at $g_2 = 0$, so that it is 0 ± 10^{-10}.

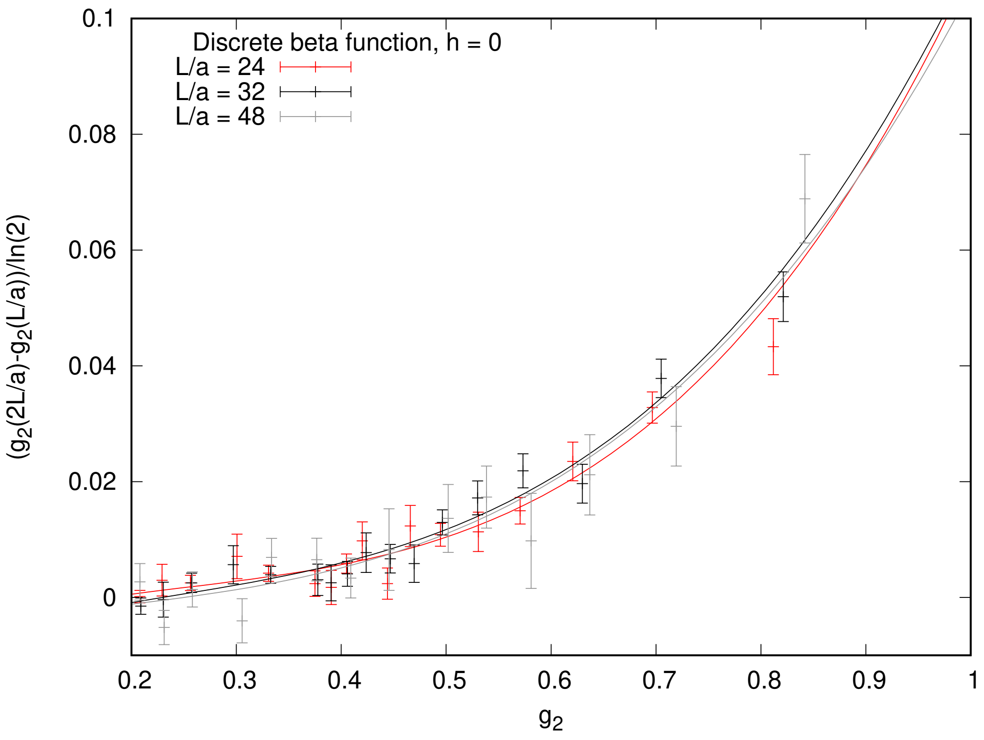


Figure 19: Measured $(g_2(2L/a) - g_2(L/a))/\ln(2)$ in the range $[0.2; 1]$ for $L/a \in \{24, 32, 48\}$. The continuous line is the fitted function (fitting error is not included).

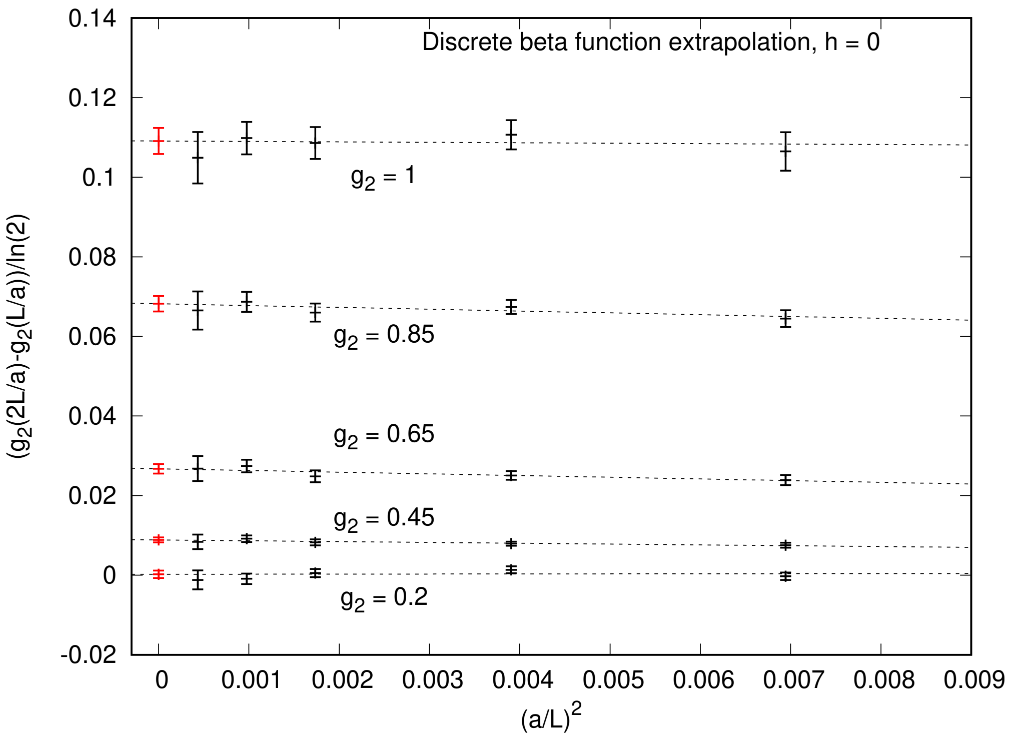


Figure 20: $(g_2(2L/a) - g_2(L/a))/\ln(2)$ extrapolation for some of the g_2, and $L/a \in \{12, 16, 24, 32, 48\}$. The extrapolated value for $(a/L)^2 \to 0$ is shown in red.

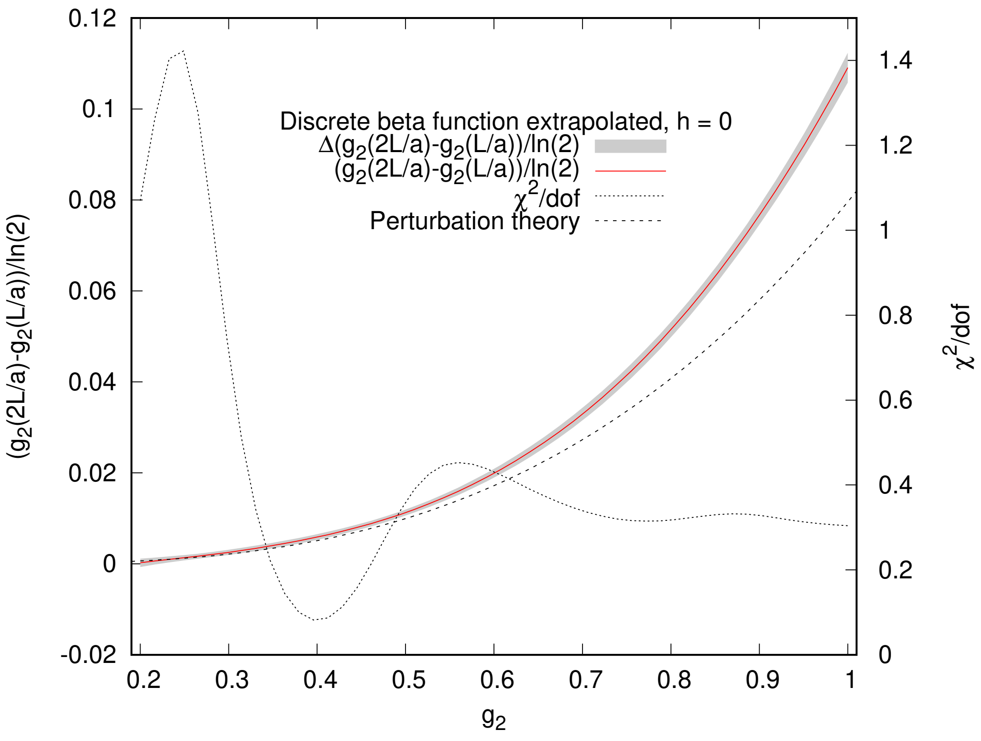


Figure 21: Extrapolated $(g_2(2L/a) - g_2(L/a))/\ln(2)$ for $h = 0$. It's error can also be seen, and χ^2/dof as well as the one loop perturbation theory prediction.

The fitted discrete beta functions for different L/a were then pointwise extrapolated for 50 different points. Figure 20 shows extrapolation for $(g_2(2L/a) - g_2(L/a))/\ln(2)$ as $(a/L)^2 \to 0$. As we can see, the a simple linear equation is a good fit, but most of the fitted lines are flat, the steepness is almost zero. For these g_2 a simple $f(x) = a$ could have been also fitted, which has less degrees of freedom than the actually fitted $f(x) = a + bx$. That means, that χ^2/dof should be very small for these g_2 as we will shortly see.

Figure 21 shows the final, continuum extrapolated result for the $(g_2(2L/a) - g_2(L/a))/\ln(2)$ discrete beta function at $h = 0$. We can see the expected low values of χ^2/dof, which is not an issue, this can be considered a good fit.

We can also see, that the calculated discrete beta function is a good fit for the perturbative result in the region $g_2 < 0.45$.

Table 2 lists all the measurements this section used.

6.2 Discrete beta function for $h = 0$, $g_2 > 1$

At $g_2 < 1$ there are no significant differences between the discrete beta functions at $h = 0$ and $h \neq 0$ as we will see in the next section. It is worthwhile to calculate it in the region $0.7 < g_2 < 2.5$. First, let's discuss the discrete beta function for $h = 0$.

The method is exactly the same as in the previous section, except that I fitted an 8th order polynomial for the discrete beta function[12] for each $L/a \in \{12, 16, 24, 32\}$, and without assuming that it's value is 0 at $g_2 = 0$. The results can be seen in figures 22, 23 and 24.

As we can see, the measurements for different L/a are very close to each other, however, χ^2/dof is greater compared to the case $g_2 < 1$, but not significantly, the extrapolation to continuum can be considered valid. The discrete beta function is not an x^3 polynomial. What is notable is that it is a monotonic function, and by looking at it, approximately linear in the region $g_2 \in [1.4; 2.4]$.

Table 3 lists all the measurements this section used.

[12]The polynomial order was determined based on the final extrapolation value of χ^2/dof. This gave the best result (closest to 1), however, even the 4th order gave the same extrapolated discrete beta function within the margin of error.

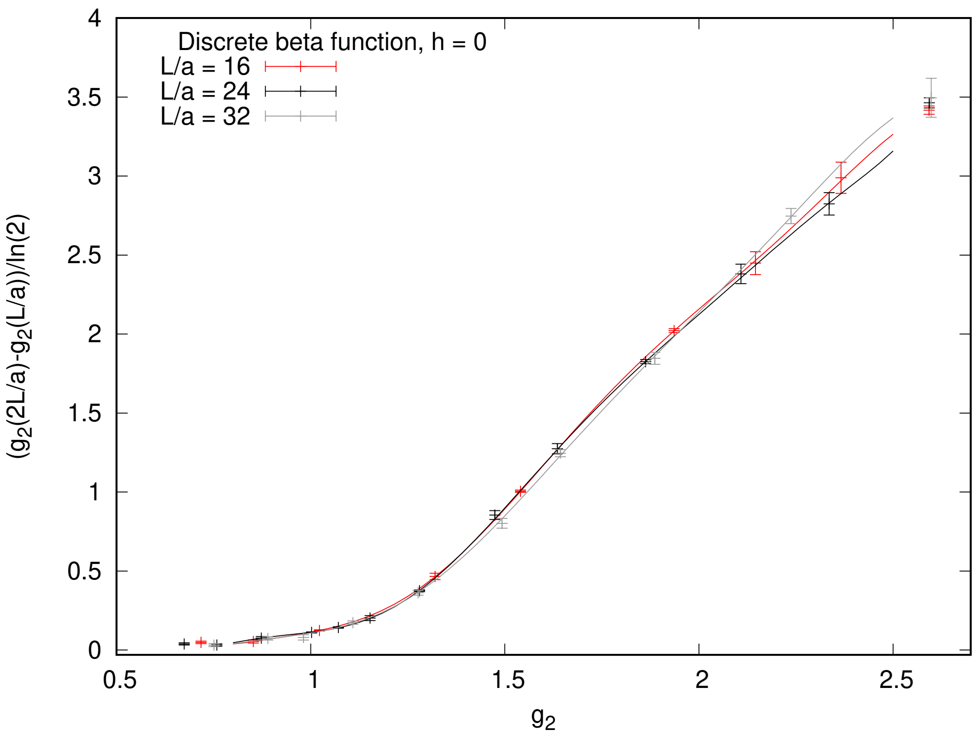


Figure 22: Measured $(g_2(2L/a) - g_2(L/a))/\ln(2)$ in the range $[0.7; 2.5]$ for $L/a \in \{16, 24, 32\}$. The continuous line is the fitted function (fitting error is not included).

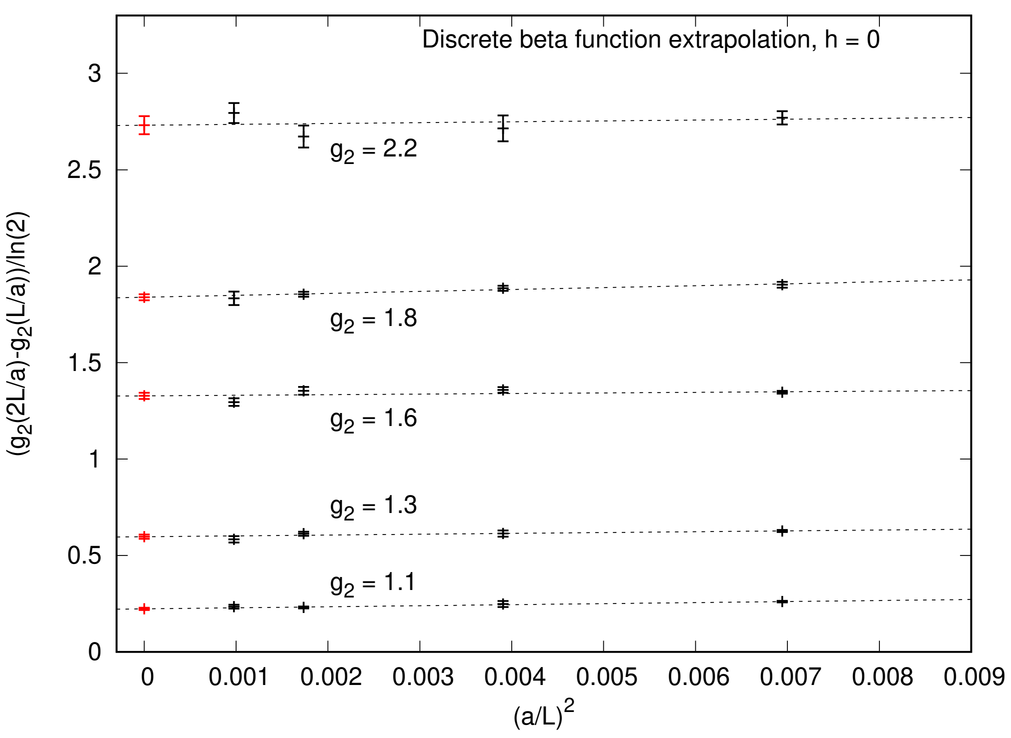


Figure 23: $(g_2(2L/a) - g_2(L/a))/\ln(2)$ extrapolation for some of the g_2, and $L/a \in \{12, 16, 24, 32\}$. The extrapolated value for $(a/L)^2 \to 0$ is shown in red.

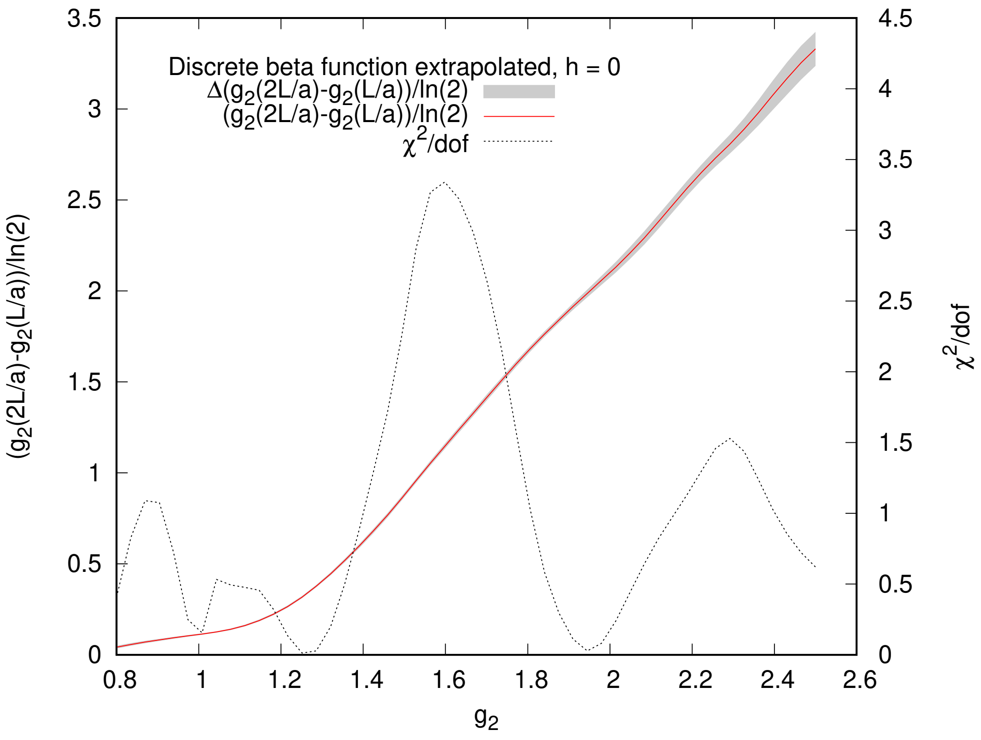


Figure 24: Extrapolated $(g_2(2L/a) - g_2(L/a))/\ln(2)$ for $h = 0$ and $L/a \in \{12, 16, 24, 32\}$. It's error can also be seen, and χ^2/dof.

6.3 Discrete beta function for small R

It is important to see whether the discrete beta function for small R is close to the $h = 0$ case or not. If we do not get back the $h = 0$ case as $R \to 0$, then there is an error in the calculation method. This section presents rough discrete beta calculations for $R = 1, 2, 5$. Here, I only would like to show is that small R results are consistent with $h = 0$.

In the next section, we will see, that discrete beta does not change significantly with L/a. In fact, the discrete beta function is the same within margin of error for $L/a = 16$ and $L/a = 32$ for our calculations. In this section I'm only going to use $L/a = 8$ because of practical reasons (less calculation time), and no interpolation was used to find the renormalized h, they were manually found. Figure 25 shows renormalization constant (R) for multiple L/a and β.

Figure 26 shows discrete beta functions for the renormalization constant $R = 1, 2, 5$. As we can see, greater R deviates from $h = 0$ discrete beta function gradually, as expected.

Table 3 and 4 lists all the measurements this section used.

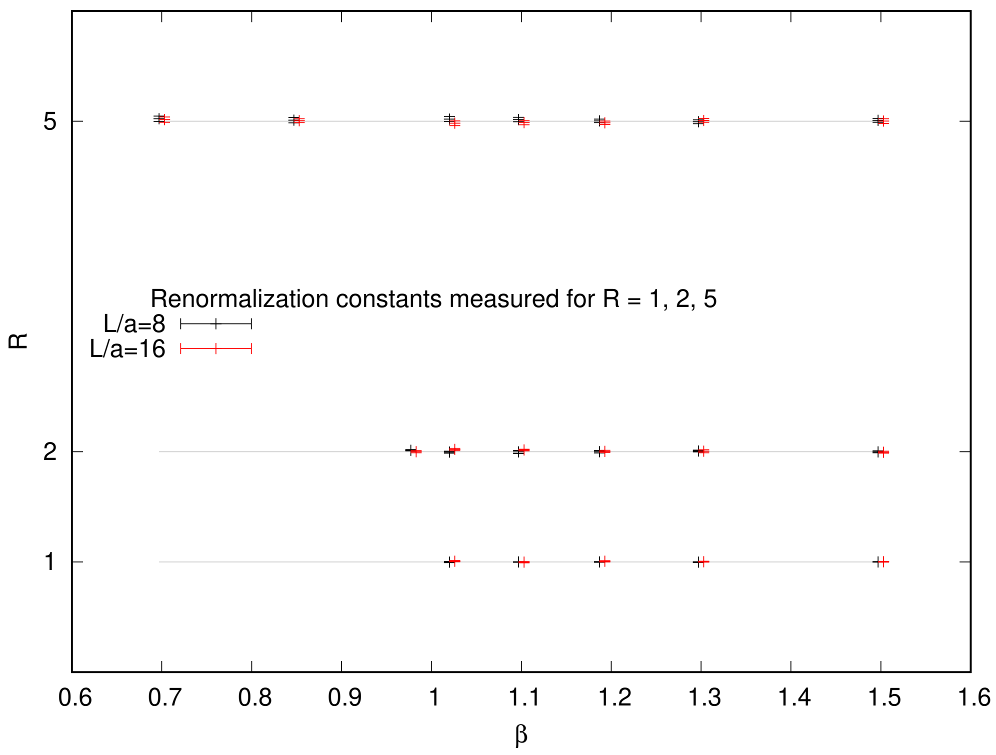


Figure 25: Measured $R(L/a, \beta, h)$ values. As we can see, manually found $h(L/a, \beta)$ satisfy the condition that $R = 1, 2, 5$.

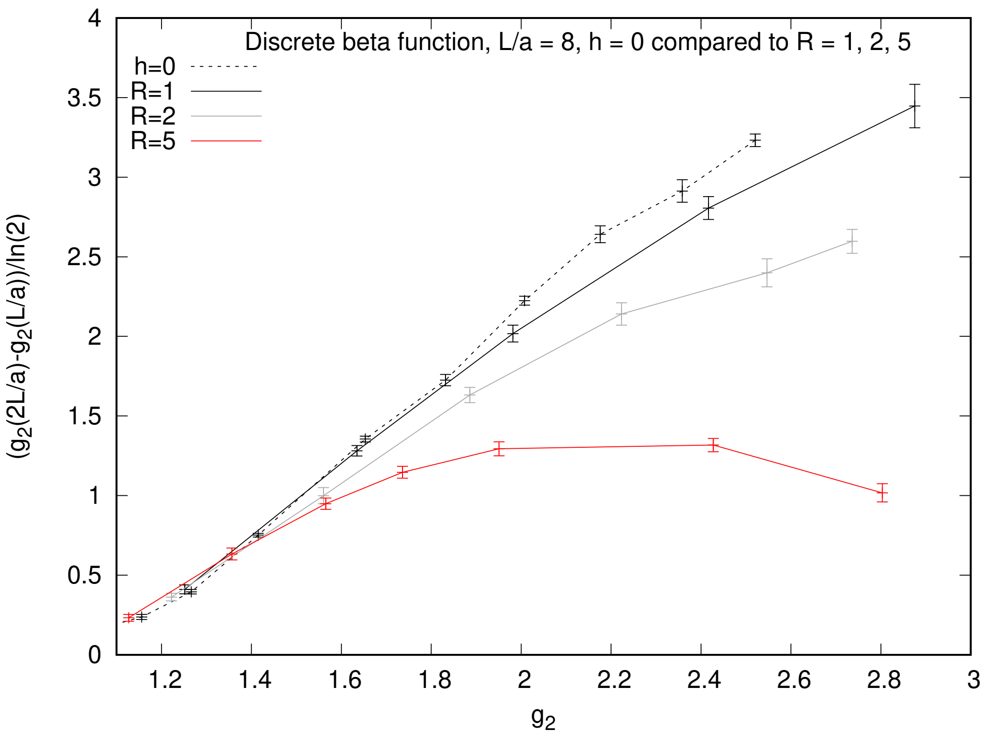


Figure 26: Measured $(g_2(2L/a) - g_2(L/a))/\ln(2)$ for multiple $R = 1, 2, 5$ and the dashed line is $h = R = 0$. $L/a = 8$. No interpolation was used.

6.4 Discrete beta function for $R = 10$

To calculate the discrete beta function in case $h \neq 0$, first, g_2 needs to be calculated for an R renormalization constant. I used $L/a \in \{8, 12, 16, 24, 32\}$. I chose $R = 10$, and $g_2 \in [0.8; 2.5]$, because β and h can be found and fitted relatively easily in this region, and gives us a nontrivial result. From the data collected, any other $R \in [5; 15]$ can be calculated, but this is not part of this document.

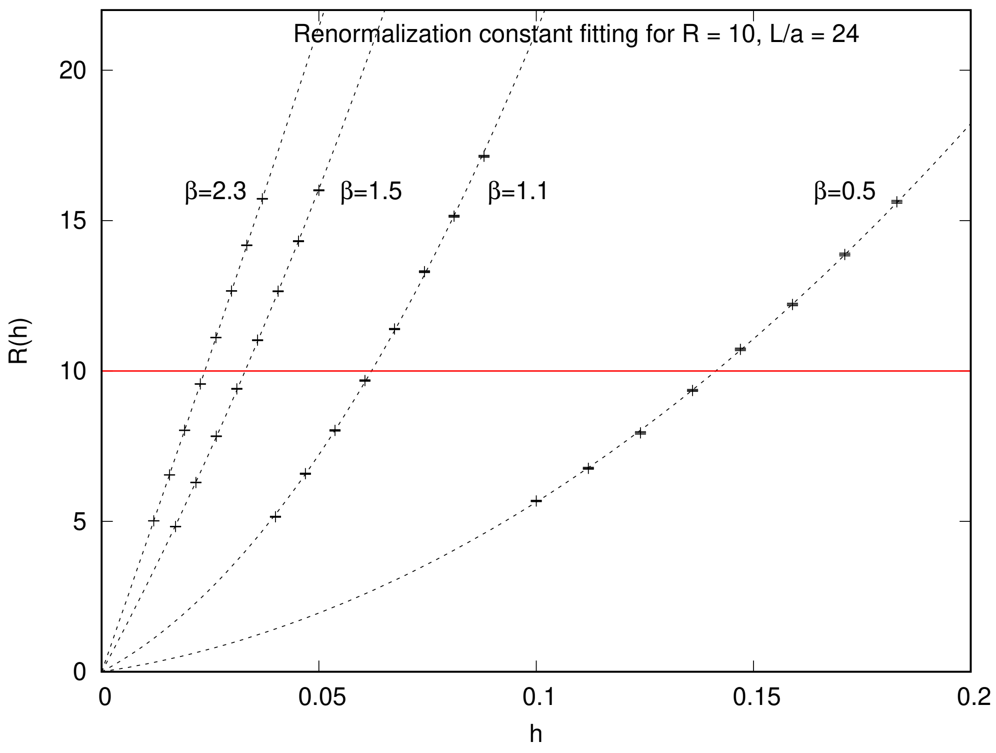


Figure 27: Some examples for the renormalization constant calculations for $L/a = 24$. The figure shows R as a function of h. The fitted function is a quadratic equation, which is a good approximation. The chosen renormalization value is shown in red.

Figure 27 shows $R(h)$. I fitted a second order polynomial on $R(h)$. Inverting the fitted function for $R = 10$ we get $\bar{h}$ for each L/a and β. From this, we can interpolate $g_2(L/a, \beta) = g_2(L/a, \beta, \bar{h})$ using an other fitting shown in figure 28. I fitted a third order polynomial on $g_2(h)$. These polynomials turned out to be a good fit in the region.

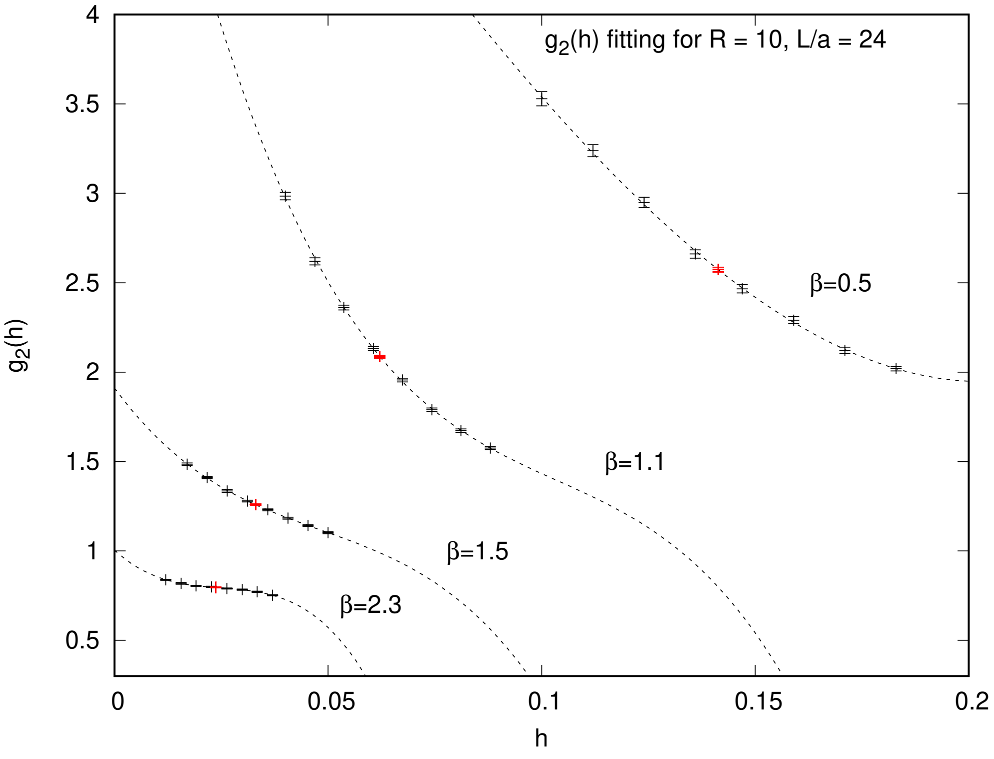


Figure 28: Some examples for g_2 as the function of h. The interpolated g_2 is shown in red, where $R = 10$. $L/a = 24$ as on figure 27.

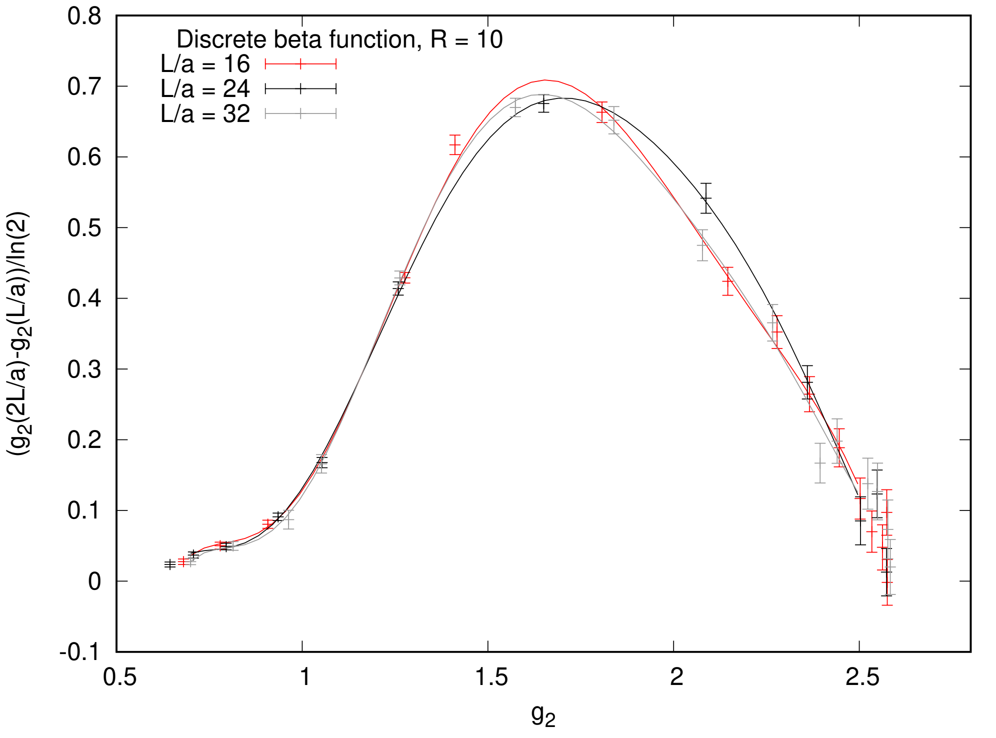


Figure 29: Measured $(g_2(2L/a) - g_2(L/a))/\ln(2)$. The continuous line is the fitted function (the fitting error is not included).

Figure 29 shows some of the discrete beta functions where I used the the previously calculated $g_2(L/a, \beta, \bar{h})$. I fitted a 7th order polynomial on $(g_2(2L/a) - g_2(L/a))/\ln(2)$.

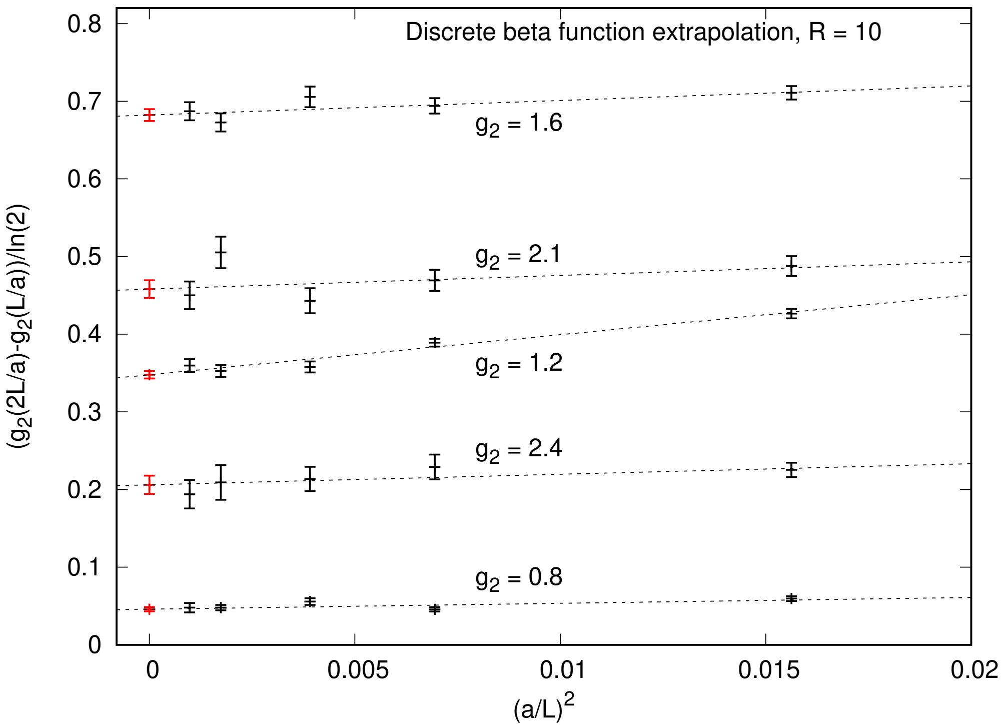


Figure 30: Extrapolated $(g_2(2L/a) - g_2(L/a))/\ln(2)$ for $(a/L) \to 0$, $L/a \in \{8, 12, 16, 24, 32\}$. The extrapolated value with error is shown in red.

The fitted $(g_2(2L/a) - g_2(L/a))/\ln(2)$ functions were then pointwise extrapolated for 50 different points for $(a/L)^2 \to 0$. Some examples of this extrapolation are shown in figure 30. If the linear equation is not a good fit, the χ^2/dof value will be high, and if a simple constant function would be a good fit, χ^2/dof is very close to 0. The extrapolated discrete beta function is shown in figure 31 as the function of g_2. As we can see, the extrapolation is valid, because χ^2/dof is close to 1.

Figure 32 shows the extrapolated discrete beta function, so that it only shows $R = 10$ result fully, not the $h = 0$ result. We can clearly see the measurement error, which is fairly small.

Table 3, 5-11 lists all the measurements this section used.

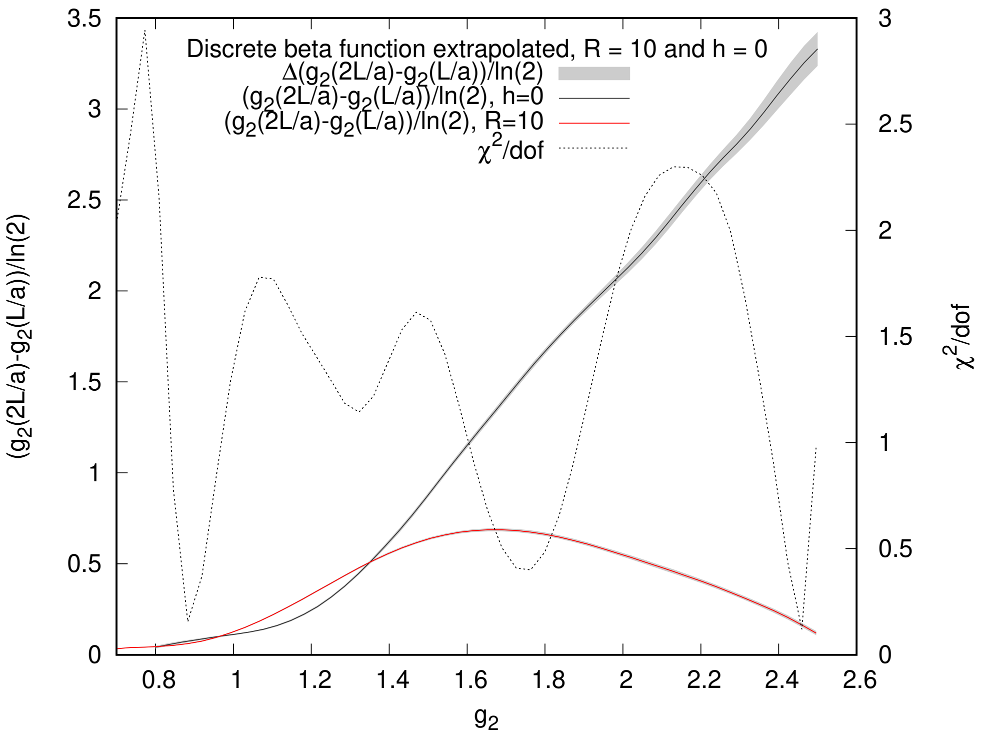


Figure 31: Red line shows the extrapolated discrete beta function with error. The error is is fairly small. The χ^2/dof is a dotted line. The calculated discrete beta function for $h = 0$ also plotted for reference.

h = 0		R = 10	
g_2	$\frac{g_2(2L/a)-g_2(L/a)}{\ln(2)}$	g_2	$\frac{g_2(2L/a)-g_2(L/a)}{\ln(2)}$
0.80	0.0419(79)	0.70	0.0332(24)
0.97	0.1041(45)	0.88	0.0612(30)
1.15	0.1885(56)	1.07	0.1843(41)
1.32	0.4448(80)	1.25	0.3919(51)
1.49	0.861(12)	1.43	0.5878(63)
1.67	1.327(16)	1.62	0.6822(76)
1.84	1.763(15)	1.80	0.6629(91)
2.01	2.131(29)	1.98	0.561(10)
2.19	2.565(41)	2.17	0.431(11)
2.36	2.980(70)	2.35	0.278(11)

Table 1: Discrete beta functions for $h = 0$ and $R = 10$ compared to each other at multiple g_2 values.

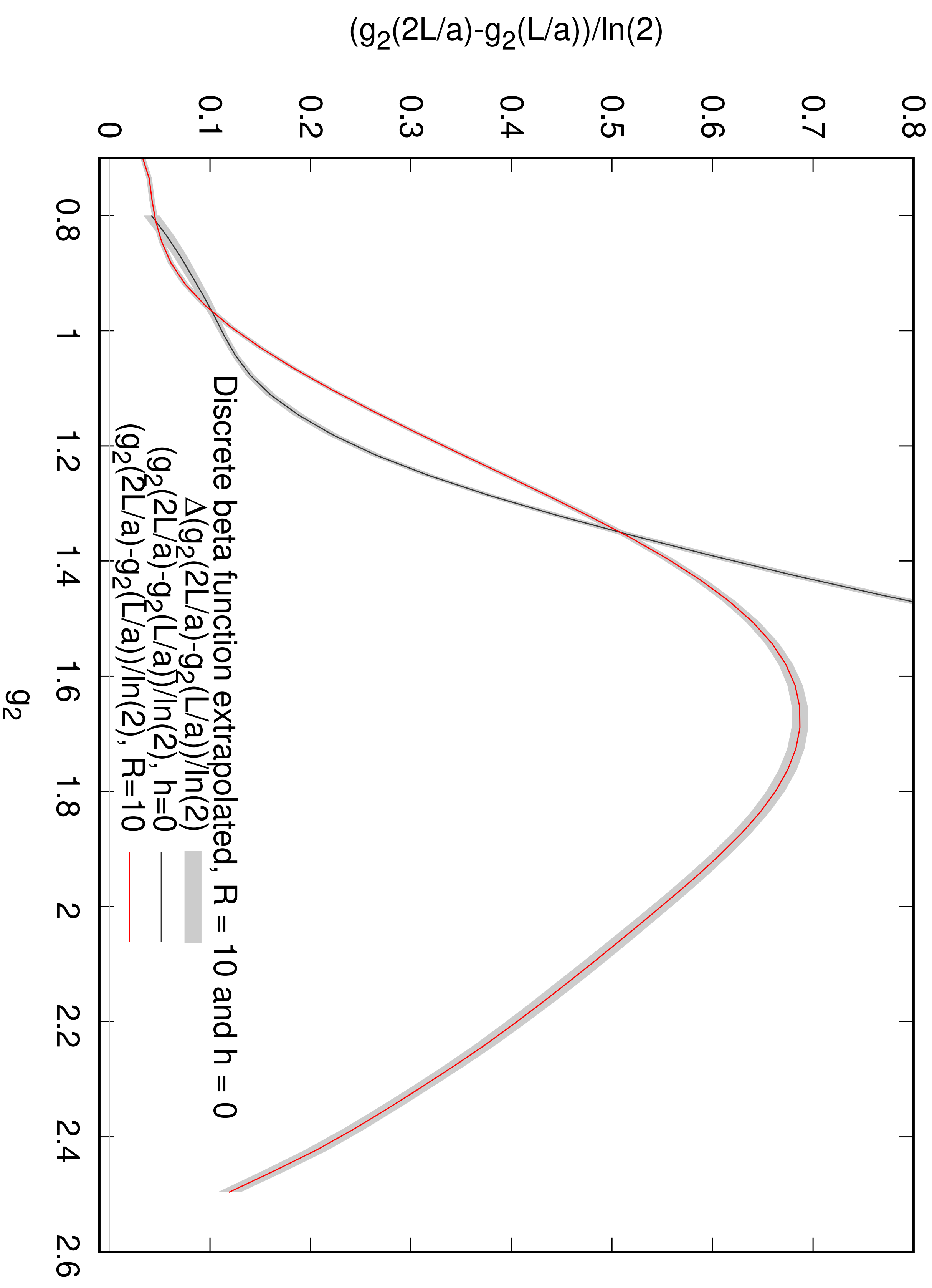


Figure 32: The extrapolated discrete beta function for $R = 10$ and $h = 0$. The y axis is scaled only to show $R = 10$ result in it's full length.

6.5 Interpretation of the results, discussion

Nonlinear sigmamodel is an extensively studied theory, because it includes asymptotic freedom, and spontaneous symmetry breaking, which are all important features of quantum chromodynamics. In quantum chromodynamics chiral symmetry is broken both explicitly and spontaneously. If the quark masses are zero, only spontaneous breaking takes place. This is analogous to the original $O(3)$ sigmamodel with $H = 0$. If the quark masses are nonzero, they break chiral symmetry explicitly, for which the analogous scenario in the $O(3)$ sigmamodel is a $H\phi$ term in the Lagrangian with a nonzero H. These parallel features were my original motivation for the study in this thesis.

I renormalized H_{R}, so that it changes with L as $H_{\mathrm{R}} \sim 1/L^2$, and an analogue in quantum chromodynamics would be $m_{\text{quark}} \sim 1/L$ because of dimensional reasons, but this is only a suggestion to see the analogy.

With this new explicit symmetry breaking term, this thesis answers how the β^{μ} function changes in a specific region. Keep in mind, that discrete beta function and beta function are related

$$\beta^{\mu}(g_{\mathrm{R}}) \sim -\frac{g_2(2L/a) - g_2(L/a)}{\ln(2)}. \tag{6.5.1}$$

The running of coupling constant became closer to 0 in a wide region significantly. This means, that the ultraviolet fixed point high-energy limit $g_{\mathrm{R}} = 0$ is reached slower with energy going to infinity ($p \to \infty$) than in $H = 0$ case. Also, more calculations are needed to confirm that but, it looks like, that there might be a nontrivial infrared fixed point. This might indicate, that the renormalized coupling constant doesn't change substantially from g_{critical} ($\beta^{\mu}(g_{\text{critical}}) = 0$) in a broad region of low energy processes, however, it depends on the exact nature of the β^{μ} function near g_{critical}, if it exists.

It's worth noting, that cut-off effects are small, the measurements do not change significantly with L/a, which is why I didn't have to use large lattice sizes.

I also mention, that calculation methods and proofs in this work can be applicable in statistical and condensed matter physics because of the analogy to the Ising model.

7 Data tables

L/a	β	sweeps	g_2
12	10.0	137500	0.33034(66)
12	12.0	15500	0.2975(18)
12	16.0	105500	0.25595(63)
12	2.0	278000	0.9427(13)
12	20.0	15500	0.2275(12)
12	2.25	10000	0.8421(65)
12	24.0	105500	0.20792(49)
12	2.5	190000	0.7785(14)
12	2.75	8000	0.7330(57)
12	3.0	288500	0.67967(92)
12	3.25	8000	0.6390(51)
12	3.5	190000	0.61192(96)
12	3.75	16000	0.5859(36)
12	4.0	288500	0.55901(74)
12	4.25	8000	0.5356(37)
12	4.5	190000	0.52057(85)
12	4.75	8000	0.5063(41)
12	5.0	288500	0.48766(66)
12	5.25	12000	0.4677(39)
12	5.5	178000	0.46069(75)
12	5.75	8000	0.4499(35)
12	6.0	200500	0.43855(73)
12	6.5	175500	0.41843(73)
12	7.0	278000	0.40079(54)
12	7.5	173500	0.38513(66)
12	8.0	182500	0.37321(65)
16	10.0	139500	0.33111(56)
16	12.0	15500	0.2968(19)
16	1.25	1500	3.094(55)
16	1.5	1000	1.5857(59)
16	15.0	1500	0.2616(90)
16	16.0	105500	0.25556(59)
16	1.75	1500	1.167(14)
16	2.0	281000	0.9678(13)
16	20.0	17500	0.2268(11)
16	2.25	6000	0.8644(96)
16	24.0	105500	0.20833(44)
16	2.5	184000	0.7874(14)
16	2.75	6000	0.7412(65)
16	3.0	291000	0.68361(95)
16	3.25	6000	0.6461(66)
16	3.5	176500	0.6169(11)
16	3.75	6000	0.5969(53)
16	4.0	291000	0.56369(78)
16	4.25	6000	0.5466(63)
16	4.5	184000	0.52198(88)
16	4.75	6000	0.5008(67)
16	5.0	291000	0.49154(69)
16	5.25	6000	0.4762(52)
16	5.5	170000	0.46351(76)
16	5.75	6000	0.4520(54)
16	6.0	199000	0.43969(79)
16	6.5	169000	0.41924(79)
16	7.0	270000	0.40237(59)
16	7.5	167000	0.38665(73)
16	8.0	197000	0.37363(59)
24	10.0	145500	0.33075(59)
24	12.0	15500	0.3006(15)
24	16.0	105500	0.25630(61)
24	2.0	135500	1.0026(19)
24	20.0	15500	0.2292(14)
24	24.0	105500	0.20803(48)
24	2.5	65500	0.8119(24)
24	3.0	155500	0.6965(12)
24	3.5	65500	0.6207(14)
24	4.0	155500	0.5706(11)
24	4.5	65500	0.5306(16)
24	5.0	167500	0.49450(97)
24	5.5	45500	0.4659(17)
24	6.0	65500	0.4440(13)
24	6.5	45500	0.4201(14)
24	7.0	165500	0.40502(79)
24	7.5	45500	0.3905(14)
24	8.0	75500	0.3750(11)
32	10.0	165500	0.33279(62)
32	12.0	15500	0.2971(16)
32	16.0	125500	0.25745(50)
32	2.0	222500	1.0356(17)
32	20.0	15500	0.2304(16)
32	24.0	125500	0.20901(45)
32	2.5	131500	0.8213(18)
32	3.0	241500	0.7050(10)
32	3.5	131500	0.6298(12)
32	4.0	239500	0.57325(87)
32	4.5	129500	0.5300(10)
32	5.0	256500	0.49644(76)
32	5.5	107500	0.4696(10)
32	6.0	127500	0.44674(98)
32	6.5	107500	0.4237(10)
32	7.0	233500	0.40588(65)
32	7.5	99500	0.39040(92)
32	8.0	119500	0.37760(83)
48	10.0	131500	0.33364(77)
48	12.0	15500	0.3055(22)
48	16.0	91500	0.25804(67)
48	2.0	127500	1.0793(22)
48	20.0	15500	0.2313(13)
48	24.0	91500	0.20813(56)
48	2.5	69500	0.8420(23)
48	3.0	143500	0.7193(14)
48	3.5	67500	0.6369(18)
48	4.0	143500	0.5809(12)
48	4.5	67500	0.5384(17)
48	5.0	143500	0.50200(97)
48	5.5	47500	0.4744(18)
48	6.0	67500	0.4456(13)
48	6.5	47500	0.4269(18)
48	7.0	143500	0.40906(83)
48	7.5	47500	0.3917(15)
48	8.0	67500	0.3766(11)
64	10.0	63500	0.33550(80)
64	12.0	15500	0.3010(17)
64	16.0	35500	0.2592(10)
64	2.0	63500	1.1221(32)
64	20.0	15500	0.2302(14)
64	24.0	31500	0.20797(87)
64	2.5	59500	0.8573(24)
64	3.0	73500	0.7312(21)
64	3.5	51500	0.6434(20)
64	4.0	71500	0.5884(18)
64	4.5	47500	0.5419(18)
64	5.0	73500	0.5054(13)
64	5.5	31500	0.4736(20)
64	6.0	39500	0.4514(14)
64	6.5	19500	0.4291(21)
64	7.0	55500	0.4087(13)
64	7.5	19500	0.3921(19)
64	8.0	31500	0.3797(17)
96	10.0	11500	0.3384(22)
96	12.0	11500	0.3027(14)
96	16.0	7500	0.2590(20)
96	2.0	11500	1.1708(65)
96	20.0	7500	0.2277(16)
96	24.0	7500	0.2100(21)
96	2.5	11500	0.8897(48)
96	3.0	11500	0.7397(45)
96	3.5	11500	0.6516(45)
96	4.0	11500	0.5877(56)
96	4.5	11500	0.5504(33)
96	5.0	11500	0.5115(39)
96	6.0	11500	0.4513(47)
96	7.0	11500	0.4114(22)
96	8.0	11500	0.3811(23)

Table 2: Measured g_2 for the calculations $h = 0$, $g_2 < 1$.

L/a	β	sweeps	g_2
8	1.1	34500	2.521(13)
8	1.13	15500	2.358(11)
8	1.16	15500	2.176(16)
8	1.2	11500	2.007(17)
8	1.25	15500	1.832(10)
8	1.3	11500	1.653(12)
8	1.4	19500	1.4152(76)
8	1.5	11500	1.2662(93)
8	1.6	11500	1.1559(56)
8	1.7	11500	1.0708(82)
8	1.9	19500	0.9549(55)
8	2.3	19500	0.8083(42)
8	2.8	19500	0.7033(34)
12	1.2	191500	2.7365(63)
12	1.26	15500	2.299(14)
12	1.23	15500	2.537(19)
12	1.3	211500	2.0833(36)
12	1.35	15500	1.8577(94)
12	1.4	221500	1.6652(27)
12	1.5	151500	1.4042(25)
12	1.6	11500	1.2526(71)
12	1.7	143500	1.1341(22)
12	1.8	11500	1.0546(64)
12	1.9	151500	0.9954(17)
12	2.3	19500	0.8326(37)
12	2.7	19500	0.7327(37)
12	3.1	19500	0.6590(35)
16	1.1	115500	4.761(24)
16	1.13	15500	4.378(48)
16	1.16	15500	4.007(33)
16	1.2	251500	3.5493(77)
16	1.25	17500	3.028(23)
16	1.3	223500	2.5926(51)
16	1.33	15500	2.366(17)
16	1.36	15500	2.146(12)
16	1.4	219500	1.9361(32)
16	1.45	15500	1.705(11)
16	1.5	185000	1.5406(26)
16	1.6	11500	1.3205(96)
16	1.7	183500	1.1893(21)
16	1.8	11500	1.1001(71)
16	1.9	191500	1.0225(17)
16	2.0	345000	0.9679(12)
16	2.3	19500	0.8524(45)
16	2.8	19500	0.7179(34)
24	1.2	215500	5.293(23)
24	1.23	15500	4.766(69)
24	1.26	15500	4.327(44)
24	1.3	231500	3.7442(97)
24	1.35	15500	3.173(24)
24	1.4	261500	2.5934(47)
24	1.43	15500	2.335(17)
24	1.46	15500	2.108(16)
24	1.5	191500	1.8628(34)
24	1.55	15500	1.6356(94)
24	1.6	11500	1.4745(87)
24	1.7	191500	1.2803(21)
24	1.8	11500	1.1529(72)
24	1.9	191500	1.0715(19)
24	2.0	203500	1.0023(16)
24	2.3	19500	0.8729(47)
24	2.7	19500	0.7583(35)
24	3.1	19500	0.6742(31)
32	1.3	259500	4.961(17)
32	1.4	219500	3.3374(78)
32	1.43	15500	2.962(26)
32	1.46	15500	2.598(13)
32	1.5	215500	2.2373(43)
32	1.55	19500	1.8863(90)
32	1.6	15500	1.643(10)
32	1.65	11500	1.493(16)
32	1.75	19500	1.2765(60)
32	1.9	219500	1.1087(17)
32	2.1	11500	0.9816(69)
32	2.3	19500	0.8896(51)
32	2.8	19500	0.7512(42)
32	1.25	11500	5.93(10)
32	1.33	11500	4.438(66)
32	1.36	11500	3.843(48)
48	1.4	155500	4.994(22)
48	1.43	15500	4.293(46)
48	1.46	15500	3.758(40)
48	1.5	159500	3.1280(83)
48	1.55	15500	2.519(20)
48	1.6	11500	2.067(17)
48	1.7	119500	1.5406(34)
48	1.8	11500	1.2930(79)
48	1.9	119500	1.1704(24)
48	2.0	215500	1.0797(17)
48	2.1	11500	1.0209(66)
48	2.2	11500	0.9550(68)
48	2.3	19500	0.9251(50)
48	2.7	19500	0.7805(42)
48	3.1	19500	0.7012(38)
64	1.5	27500	4.141(33)
64	1.55	19500	3.166(24)
64	1.6	27500	2.507(12)
64	1.65	11500	2.049(13)
64	1.75	19500	1.5281(89)
64	1.9	19500	1.2289(70)
64	2.1	11500	1.0368(95)
64	2.3	19500	0.9405(51)
64	2.8	19500	0.7727(47)
64	1.43	11500	5.80(11)
64	1.46	11500	5.021(85)

Table 3: Measured g_2 for the calculations $h = 0$, $g_2 > 1$.

L/a	β	h	sweeps	M_0	g_2
8	1.5	0.02335	14500	0.4483(21)	1.2516(67)
8	1.3	0.027	14500	0.3341(18)	1.6348(96)
8	1.19	0.0308	9500	0.2576(14)	1.982(15)
8	1.1	0.035	9500	0.1991(12)	2.417(19)
8	1.023	0.03983	9500	0.1536(16)	2.875(19)
8	1.5	0.0464	5500	0.4534(38)	1.222(13)
8	1.3	0.0532	3500	0.3475(26)	1.560(13)
8	1.19	0.0595	3500	0.2761(33)	1.886(24)
8	1.1	0.0668	5500	0.2185(30)	2.223(29)
8	1.023	0.0748	5500	0.1740(16)	2.547(28)
8	0.98	0.08	15500	0.15476(97)	2.736(19)
8	1.5	0.113	5500	0.4795(34)	1.1272(93)
8	1.3	0.125	5500	0.3898(28)	1.357(18)
8	1.19	0.137	3500	0.3256(20)	1.565(15)
8	1.1	0.149	5500	0.2765(22)	1.736(16)
8	1.023	0.162	5500	0.2344(20)	1.951(22)
8	0.85	0.197	11500	0.1579(15)	2.427(14)
8	0.7	0.234	11500	0.1125(11)	2.803(22)
16	1.5	0.00725	7500	0.2917(17)	1.536(18)
16	1.3	0.0102	9500	0.1474(13)	2.523(20)
16	1.19	0.0133	9500	0.08736(97)	3.380(33)
16	1.1	0.0162	13500	0.05790(65)	4.362(46)
16	1.023	0.0192	17500	0.04200(47)	5.265(93)
16	1.5	0.0142	5500	0.3015(26)	1.474(11)
16	1.3	0.0195	3500	0.1613(22)	2.253(32)
16	1.19	0.02478	7500	0.0997(10)	3.017(23)
16	1.1	0.03033	23500	0.06741(53)	3.707(39)
16	1.023	0.0352	17500	0.05030(51)	4.210(54)
16	0.98	0.0377	19500	0.04296(40)	4.537(49)
16	1.5	0.03425	3500	0.3253(28)	1.288(12)
16	1.3	0.0432	7500	0.2049(14)	1.796(19)
16	1.19	0.0523	7500	0.13864(87)	2.223(20)
16	1.1	0.061	13500	0.10195(78)	2.531(20)
16	1.023	0.07	11500	0.07728(67)	2.848(21)
16	0.85	0.0921	19500	0.04504(32)	3.341(26)
16	0.7	0.112	15500	0.03059(29)	3.508(33)

Table 4: Measured M_0, g_2 for the calculations for small $h \neq 0$, $L/a = 8, 16$.

L/a	β	h	sweeps	M_0	g_2
8	0.07	0.35	62500	0.03290(19)	4.397(30)
8	0.07	0.411	47500	0.03912(22)	3.731(26)
8	0.07	0.473	47500	0.04552(24)	3.286(19)
8	0.07	0.534	47500	0.05278(25)	2.879(15)
8	0.07	0.596	47500	0.06161(26)	2.568(12)
8	0.07	0.657	47500	0.07017(30)	2.343(11)
8	0.07	0.719	47500	0.07966(30)	2.161(10)
8	0.07	0.78	52500	0.08991(32)	1.9763(87)
8	0.08	0.35	47500	0.03343(20)	4.326(34)
8	0.08	0.411	47500	0.03971(24)	3.646(23)
8	0.08	0.473	47500	0.04664(24)	3.223(17)
8	0.08	0.534	47500	0.05445(28)	2.845(13)
8	0.08	0.596	47500	0.06252(25)	2.594(14)
8	0.08	0.657	47500	0.07141(27)	2.332(10)
8	0.08	0.719	47500	0.08120(32)	2.133(11)
8	0.08	0.78	47500	0.09060(32)	1.9766(88)
8	0.09	0.35	47500	0.03418(23)	4.336(35)
8	0.09	0.411	47500	0.04073(26)	3.605(24)
8	0.09	0.473	47500	0.04712(23)	3.211(17)
8	0.09	0.534	47500	0.05542(26)	2.811(15)
8	0.09	0.596	47500	0.06406(29)	2.544(12)
8	0.09	0.657	47500	0.07334(30)	2.3077(96)
8	0.09	0.719	47500	0.08365(33)	2.1123(99)
8	0.09	0.78	52500	0.09405(33)	1.9617(77)
8	0.15	0.35	47500	0.03894(24)	4.080(32)
8	0.15	0.411	47500	0.04583(22)	3.466(19)
8	0.15	0.473	47500	0.05436(28)	3.035(17)
8	0.15	0.534	47500	0.06362(27)	2.673(13)
8	0.15	0.596	47500	0.07364(33)	2.417(11)
8	0.15	0.657	47500	0.08420(32)	2.1785(84)
8	0.15	0.719	47500	0.09609(36)	2.0012(68)
8	0.15	0.78	52500	0.10713(35)	1.8623(67)
8	0.2	0.33	47500	0.04108(22)	4.056(27)
8	0.2	0.386	47500	0.04796(26)	3.550(22)
8	0.2	0.441	47500	0.05604(27)	3.110(16)
8	0.2	0.497	47500	0.06476(28)	2.774(13)
8	0.2	0.553	47500	0.07462(30)	2.4706(98)
8	0.2	0.609	47500	0.08515(30)	2.249(10)
8	0.2	0.664	47500	0.09573(37)	2.0740(97)
8	0.2	0.72	47500	0.10688(41)	1.9251(88)
8	0.25	0.31	47500	0.04322(23)	4.150(25)
8	0.25	0.363	47500	0.05064(28)	3.531(20)
8	0.25	0.416	47500	0.05908(28)	3.067(17)
8	0.25	0.469	47500	0.06814(30)	2.787(15)
8	0.25	0.521	47500	0.07825(32)	2.502(12)
8	0.25	0.574	47500	0.08743(32)	2.293(11)
8	0.25	0.627	47500	0.09915(37)	2.104(11)
8	0.25	0.68	47500	0.11148(35)	1.9435(78)
8	0.3	0.29	47500	0.04534(27)	4.132(33)
8	0.3	0.34	47500	0.05314(27)	3.547(19)
8	0.3	0.39	47500	0.06207(29)	3.141(17)
8	0.3	0.44	47500	0.07116(33)	2.805(14)
8	0.3	0.49	47500	0.08071(31)	2.554(11)
8	0.3	0.54	47500	0.09224(36)	2.3062(96)
8	0.3	0.59	47500	0.10298(35)	2.1262(93)
8	0.3	0.64	47500	0.11504(40)	1.9597(76)
8	0.5	0.27	52500	0.07080(33)	3.423(18)
8	0.5	0.317	47500	0.08260(35)	2.989(14)
8	0.5	0.364	47500	0.09646(42)	2.642(12)
8	0.5	0.411	47500	0.11080(42)	2.369(11)
8	0.5	0.459	47500	0.12594(46)	2.143(10)
8	0.5	0.506	47500	0.14070(48)	1.9708(89)
8	0.5	0.553	47500	0.15571(47)	1.8274(63)
8	0.5	0.6	52500	0.17199(42)	1.7067(68)
8	0.7	0.23	47500	0.11092(46)	2.838(13)
8	0.7	0.274	47500	0.12998(54)	2.502(10)
8	0.7	0.319	47500	0.14830(59)	2.2275(86)
8	0.7	0.363	47500	0.16843(50)	2.0233(88)
8	0.7	0.407	47500	0.18811(56)	1.8406(65)
8	0.7	0.451	47500	0.20667(62)	1.6999(62)
8	0.7	0.496	47500	0.22477(59)	1.5887(60)
8	0.7	0.54	47500	0.24381(58)	1.4774(49)
8	0.9	0.185	47500	0.17547(59)	2.2812(88)
8	0.9	0.224	47500	0.19852(68)	2.0548(62)
8	0.9	0.264	47500	0.22445(61)	1.8462(63)
8	0.9	0.303	47500	0.24456(64)	1.7106(60)
8	0.9	0.342	47500	0.26548(66)	1.5715(57)
8	0.9	0.381	47500	0.28397(60)	1.4792(49)
8	0.9	0.421	47500	0.30146(62)	1.3947(44)
8	0.9	0.46	47500	0.31988(61)	1.3228(46)
8	1.1	0.14	47500	0.27235(81)	1.7757(69)
8	1.1	0.177	47500	0.29614(73)	1.6389(57)
8	1.1	0.214	47500	0.31873(74)	1.5020(49)
8	1.1	0.251	47500	0.33744(75)	1.4168(55)
8	1.1	0.289	47500	0.35629(63)	1.3418(47)
8	1.1	0.326	47500	0.37253(74)	1.2712(44)
8	1.1	0.363	47500	0.38720(70)	1.2171(35)
8	1.1	0.4	52500	0.40046(66)	1.1689(41)
8	1.4	0.125	47500	0.44023(99)	1.2106(39)
8	1.4	0.16	47500	0.45432(91)	1.1707(41)
8	1.4	0.195	47500	0.46486(93)	1.1234(37)
8	1.4	0.23	47500	0.47754(77)	1.0905(34)
8	1.4	0.265	47500	0.48767(76)	1.0557(36)
8	1.4	0.3	47500	0.49714(71)	1.0229(37)
8	1.4	0.335	47500	0.50709(65)	0.9946(33)
8	1.4	0.37	47500	0.51496(65)	0.9679(35)
8	1.9	0.11	47500	0.5973(12)	0.9000(29)
8	1.9	0.143	47500	0.6040(10)	0.8847(32)
8	1.9	0.176	47500	0.60782(95)	0.8660(29)
8	1.9	0.209	47500	0.61440(88)	0.8441(27)
8	1.9	0.241	47500	0.61890(75)	0.8348(27)
8	1.9	0.274	47500	0.62327(81)	0.8214(32)
8	1.9	0.307	47500	0.62822(74)	0.8033(28)
8	1.9	0.34	47500	0.63303(69)	0.7923(31)
8	2.3	0.095	47500	0.6683(13)	0.7807(26)
8	2.3	0.124	47500	0.6716(11)	0.7676(28)
8	2.3	0.154	47500	0.67489(99)	0.7601(26)
8	2.3	0.183	47500	0.67849(98)	0.7479(26)
8	2.3	0.212	47500	0.68129(82)	0.7438(23)
8	2.3	0.241	47500	0.68370(81)	0.7286(24)
8	2.3	0.271	47500	0.68721(79)	0.7211(26)
8	2.3	0.3	47500	0.69033(82)	0.7111(24)
8	2.8	0.08	52500	0.7275(13)	0.6847(20)
8	2.8	0.109	47500	0.7296(12)	0.6790(22)
8	2.8	0.137	47500	0.7320(10)	0.6662(22)
8	2.8	0.166	47500	0.73447(96)	0.6606(22)
8	2.8	0.194	47500	0.73538(86)	0.6583(23)
8	2.8	0.223	47500	0.73849(82)	0.6562(21)
8	2.8	0.251	47500	0.74022(71)	0.6429(17)
8	2.8	0.28	52500	0.74216(72)	0.6350(22)

Table 5: Measured M_0, g_2 for the calculations $h \neq 0$, $L/a = 8$.

L/a	β	h	sweeps	M_0	g_2
12	0.1	0.24	15500	0.01602(20)	4.242(57)
12	0.1	0.27	15500	0.01882(19)	3.675(43)
12	0.1	0.312	55500	0.02185(11)	3.277(19)
12	0.1	0.357	15500	0.02572(29)	2.848(35)
12	0.1	0.438	55500	0.03481(17)	2.326(12)
12	0.1	0.469	55500	0.03867(14)	2.174(11)
12	0.1	0.51	15500	0.04370(27)	2.023(24)
12	0.3	0.22	15500	0.02328(29)	3.919(41)
12	0.3	0.243	55500	0.02634(17)	3.489(23)
12	0.3	0.292	55500	0.03293(15)	2.920(15)
12	0.3	0.34	55500	0.04029(19)	2.510(13)
12	0.3	0.365	55500	0.04464(17)	2.363(11)
12	0.3	0.413	55500	0.05334(22)	2.0834(95)
12	0.3	0.438	55500	0.05797(20)	1.9703(84)
12	0.3	0.44	15500	0.05822(28)	1.950(13)
12	0.5	0.19	15500	0.03552(35)	3.528(33)
12	0.5	0.208	55500	0.03828(18)	3.295(19)
12	0.5	0.243	95500	0.04615(16)	2.8147(94)
12	0.5	0.26	55500	0.05041(24)	2.661(12)
12	0.5	0.292	55500	0.05831(21)	2.3836(97)
12	0.5	0.312	55500	0.06380(24)	2.227(10)
12	0.5	0.33	55500	0.06922(26)	2.1146(89)
12	0.5	0.34	55500	0.07222(24)	2.0415(86)
12	0.5	0.347	55500	0.07427(25)	2.0190(74)
12	0.5	0.365	55500	0.07919(25)	1.9385(78)
12	0.5	0.37	15500	0.08050(56)	1.889(17)
12	0.7	0.13	23500	0.04628(34)	3.680(32)
12	0.7	0.15	23500	0.05148(35)	3.351(29)
12	0.7	0.17	23500	0.06016(44)	2.941(22)
12	0.7	0.19	23500	0.06682(41)	2.715(16)
12	0.7	0.21	23500	0.07550(49)	2.453(13)
12	0.7	0.23	23500	0.08391(44)	2.283(14)
12	0.7	0.25	23500	0.09300(52)	2.100(12)
12	0.7	0.27	23500	0.10201(53)	1.971(10)
12	0.7	0.3	23500	0.11548(54)	1.803(11)
12	0.9	0.11	15500	0.08475(64)	2.983(27)
12	0.9	0.125	75500	0.09498(29)	2.6704(97)
12	0.9	0.132	55500	0.09909(39)	2.5518(92)
12	0.9	0.139	55500	0.10344(39)	2.462(10)
12	0.9	0.146	35500	0.10857(51)	2.380(11)
12	0.9	0.156	35500	0.11514(46)	2.2774(91)
12	0.9	0.174	35500	0.12801(49)	2.0736(94)
12	0.9	0.188	35500	0.13737(53)	1.9537(87)
12	0.9	0.208	55500	0.15222(41)	1.8130(65)
12	0.9	0.243	35500	0.17561(58)	1.6151(72)
12	0.9	0.25	15500	0.18149(78)	1.5687(88)
12	1.1	0.08	15500	0.15369(82)	2.266(15)
12	1.1	0.0972	43500	0.17193(57)	2.0414(88)
12	1.1	0.104	43500	0.18045(65)	1.9574(78)
12	1.1	0.118	43500	0.19502(65)	1.8358(70)
12	1.1	0.125	43500	0.20056(65)	1.7752(77)
12	1.1	0.132	43500	0.20769(57)	1.7220(60)
12	1.1	0.139	43500	0.21461(59)	1.6725(61)
12	1.1	0.15	15500	0.2236(11)	1.607(11)
12	1.1	0.17	15500	0.2402(10)	1.5011(94)
12	1.1	0.19	15500	0.25729(99)	1.4141(93)
12	1.1	0.21	15500	0.2699(10)	1.3406(89)
12	1.3	0.06	15500	0.2699(14)	1.627(14)
12	1.3	0.0833	55500	0.29509(65)	1.4788(47)
12	1.3	0.0972	55500	0.30830(70)	1.4028(39)
12	1.3	0.118	55500	0.32605(59)	1.3141(43)
12	1.3	0.132	55500	0.33571(64)	1.2631(39)
12	1.3	0.139	55500	0.34005(59)	1.2468(39)
12	1.3	0.146	55500	0.34607(59)	1.2214(36)
12	1.3	0.153	55500	0.34996(68)	1.2002(43)
12	1.3	0.16	55500	0.35575(59)	1.1785(36)
12	1.3	0.174	55500	0.36314(55)	1.1523(40)
12	1.3	0.18	15500	0.3666(10)	1.1298(58)
12	1.5	0.05	15500	0.3860(17)	1.2409(84)
12	1.5	0.0694	55500	0.39897(84)	1.1904(40)
12	1.5	0.0833	55500	0.40797(79)	1.1466(38)
12	1.5	0.0972	55500	0.41495(82)	1.1186(35)
12	1.5	0.118	55500	0.42645(70)	1.0702(35)
12	1.5	0.132	55500	0.43387(64)	1.0409(35)
12	1.5	0.139	55500	0.43679(69)	1.0351(34)
12	1.5	0.146	55500	0.43942(58)	1.0203(31)
12	1.5	0.153	55500	0.44378(68)	1.0118(35)
12	1.5	0.16	55500	0.44765(68)	0.9959(34)
12	1.9	0.045	15500	0.5285(19)	0.9445(67)
12	1.9	0.05	15500	0.5313(20)	0.9382(56)
12	1.9	0.0694	55500	0.53809(91)	0.9068(32)
12	1.9	0.0833	55500	0.54211(82)	0.8985(33)
12	1.9	0.0972	55500	0.54629(69)	0.8805(30)
12	1.9	0.118	55500	0.55290(70)	0.8573(28)
12	1.9	0.132	55500	0.55615(70)	0.8425(29)
12	1.9	0.139	55500	0.55852(63)	0.8356(28)
12	1.9	0.14	15500	0.5578(11)	0.8328(43)
12	2.3	0.04	15500	0.6110(22)	0.8044(59)
12	2.3	0.05	15500	0.6162(22)	0.8003(57)
12	2.3	0.06	15500	0.6162(18)	0.7894(45)
12	2.3	0.08	15500	0.6211(14)	0.7716(42)
12	2.3	0.09	15500	0.6247(15)	0.7644(44)
12	2.3	0.1	15500	0.6265(13)	0.7498(44)
12	2.3	0.12	15500	0.6303(13)	0.7526(36)
12	2.3	0.14	15500	0.6342(13)	0.7268(42)
12	2.3	0.15	15500	0.6362(12)	0.7173(45)
12	2.7	0.04	15500	0.6709(24)	0.7113(40)
12	2.7	0.05	15500	0.6739(18)	0.7076(40)
12	2.7	0.06	15500	0.6740(17)	0.7012(53)
12	2.7	0.08	15500	0.6780(18)	0.6938(47)
12	2.7	0.09	15500	0.6799(13)	0.6866(41)
12	2.7	0.1	15500	0.6818(14)	0.6849(42)
12	2.7	0.12	15500	0.6856(13)	0.6653(40)
12	2.7	0.14	15500	0.6863(12)	0.6605(42)
12	2.7	0.15	15500	0.68740(97)	0.6560(47)
12	3.1	0.04	15500	0.7141(23)	0.6447(37)
12	3.1	0.05	15500	0.7153(17)	0.6450(45)
12	3.1	0.06	15500	0.7165(19)	0.6435(40)
12	3.1	0.07	15500	0.7184(18)	0.6325(45)
12	3.1	0.08	15500	0.7200(16)	0.6230(33)
12	3.1	0.09	15500	0.7203(16)	0.6256(35)
12	3.1	0.1	15500	0.7212(14)	0.6253(36)
12	3.1	0.12	15500	0.7241(16)	0.6122(50)
12	3.1	0.13	15500	0.7249(11)	0.6059(29)

Table 6: Measured M_0, g_2 for the calculations $h \neq 0$, $L/a = 12$.

L/a	β	h	sweeps	M_0	g_2
16	0.07	0.19	32500	0.009097(80)	4.090(42)
16	0.07	0.219	27500	0.01063(10)	3.604(39)
16	0.07	0.247	27500	0.012271(97)	3.205(30)
16	0.07	0.276	27500	0.01460(11)	2.864(24)
16	0.07	0.304	27500	0.01635(13)	2.586(20)
16	0.07	0.333	27500	0.01867(13)	2.396(16)
16	0.07	0.361	27500	0.02130(16)	2.182(18)
16	0.07	0.39	32500	0.02392(14)	2.015(15)
16	0.08	0.18	27500	0.008849(99)	4.296(69)
16	0.08	0.21	27500	0.01039(11)	3.734(47)
16	0.08	0.24	27500	0.012165(94)	3.292(34)
16	0.08	0.27	27500	0.01407(12)	2.936(27)
16	0.08	0.3	27500	0.01650(12)	2.614(19)
16	0.08	0.33	27500	0.01898(14)	2.398(22)
16	0.08	0.36	27500	0.02170(14)	2.178(18)
16	0.08	0.39	27500	0.02485(15)	1.995(16)
16	0.09	0.18	27500	0.008964(96)	4.343(64)
16	0.09	0.209	27500	0.01067(11)	3.658(40)
16	0.09	0.237	27500	0.01227(12)	3.309(34)
16	0.09	0.266	27500	0.01431(13)	2.918(22)
16	0.09	0.294	27500	0.01654(14)	2.601(26)
16	0.09	0.323	27500	0.01896(13)	2.431(16)
16	0.09	0.351	27500	0.02153(14)	2.218(19)
16	0.09	0.38	27500	0.02419(16)	2.061(16)
16	0.15	0.18	32500	0.01026(10)	4.037(53)
16	0.15	0.207	27500	0.01188(11)	3.549(31)
16	0.15	0.234	27500	0.01393(13)	3.175(31)
16	0.15	0.261	27500	0.01604(12)	2.857(29)
16	0.15	0.289	27500	0.01867(15)	2.563(19)
16	0.15	0.316	27500	0.02118(15)	2.352(17)
16	0.15	0.343	27500	0.02442(16)	2.149(19)
16	0.15	0.37	27500	0.02682(17)	2.019(11)
16	0.2	0.17	27500	0.010904(99)	4.168(59)
16	0.2	0.197	27500	0.01289(12)	3.540(34)
16	0.2	0.224	27500	0.01461(11)	3.201(26)
16	0.2	0.251	27500	0.01737(14)	2.845(26)
16	0.2	0.279	27500	0.02000(13)	2.540(25)
16	0.2	0.306	27500	0.02303(17)	2.331(17)
16	0.2	0.333	27500	0.02617(17)	2.129(14)
16	0.2	0.36	27500	0.02962(18)	1.977(16)
16	0.25	0.17	27500	0.01225(14)	3.963(57)
16	0.25	0.196	27500	0.01444(12)	3.491(36)
16	0.25	0.221	27500	0.01665(13)	3.080(27)
16	0.25	0.247	27500	0.01928(15)	2.754(26)
16	0.25	0.273	27500	0.02207(16)	2.533(20)
16	0.25	0.299	27500	0.02567(16)	2.281(18)
16	0.25	0.324	27500	0.02874(17)	2.106(13)
16	0.25	0.35	27500	0.03261(16)	1.937(13)
16	0.3	0.16	32500	0.01286(12)	4.029(44)
16	0.3	0.184	27500	0.01497(12)	3.599(34)
16	0.3	0.209	27500	0.01774(14)	3.079(25)
16	0.3	0.233	27500	0.02065(16)	2.795(23)
16	0.3	0.257	27500	0.02345(15)	2.579(19)
16	0.3	0.281	27500	0.02663(19)	2.348(16)
16	0.3	0.306	27500	0.03048(18)	2.105(18)
16	0.3	0.33	32500	0.03404(15)	1.997(12)
16	0.4	0.147	19500	0.01542(16)	3.902(55)
16	0.4	0.169	19500	0.01810(17)	3.435(28)
16	0.4	0.191	19500	0.02103(17)	3.065(30)
16	0.4	0.213	19500	0.02396(17)	2.784(24)
16	0.4	0.234	19500	0.02767(24)	2.516(21)
16	0.4	0.256	19500	0.03063(22)	2.370(19)
16	0.4	0.278	19500	0.03520(24)	2.157(21)
16	0.4	0.3	19500	0.03946(27)	2.012(19)
16	0.5	0.135	27500	0.01898(18)	3.861(40)
16	0.5	0.156	27500	0.02187(18)	3.432(30)
16	0.5	0.176	27500	0.02515(17)	3.048(27)
16	0.5	0.197	27500	0.02982(21)	2.710(19)
16	0.5	0.218	27500	0.03426(20)	2.442(15)
16	0.5	0.239	27500	0.03859(22)	2.258(13)
16	0.5	0.259	27500	0.04296(23)	2.084(12)
16	0.5	0.28	32500	0.04953(24)	1.902(10)
16	0.6	0.122	19500	0.02341(26)	3.781(45)
16	0.6	0.14	19500	0.02720(28)	3.323(37)
16	0.6	0.159	19500	0.03143(29)	2.942(25)
16	0.6	0.177	19500	0.03626(26)	2.663(20)
16	0.6	0.195	19500	0.04117(27)	2.413(20)
16	0.6	0.213	19500	0.04689(30)	2.246(13)
16	0.6	0.232	19500	0.05163(34)	2.069(17)
16	0.6	0.25	24500	0.05797(26)	1.906(11)
16	0.7	0.11	27500	0.02952(23)	3.649(31)
16	0.7	0.127	27500	0.03467(28)	3.159(23)
16	0.7	0.143	27500	0.03973(25)	2.830(22)
16	0.7	0.16	27500	0.04610(27)	2.563(21)
16	0.7	0.177	27500	0.05240(30)	2.326(14)
16	0.7	0.194	27500	0.05864(38)	2.135(15)
16	0.7	0.21	27500	0.06589(33)	1.979(13)
16	0.7	0.227	27500	0.07326(38)	1.846(10)
16	0.8	0.097	19500	0.03872(31)	3.401(36)
16	0.8	0.112	19500	0.04490(32)	3.041(25)
16	0.8	0.126	19500	0.05097(38)	2.737(24)
16	0.8	0.141	19500	0.05829(37)	2.464(17)
16	0.8	0.156	19500	0.06621(38)	2.235(15)
16	0.8	0.171	19500	0.07494(44)	2.062(13)
16	0.8	0.185	19500	0.08221(40)	1.907(12)
16	0.8	0.2	29500	0.09157(40)	1.762(10)
16	0.9	0.085	52500	0.05193(24)	3.171(17)
16	0.9	0.0979	27500	0.05990(30)	2.832(20)
16	0.9	0.111	27500	0.06849(41)	2.520(12)
16	0.9	0.124	27500	0.07769(37)	2.301(12)
16	0.9	0.136	27500	0.08595(41)	2.132(13)
16	0.9	0.149	27500	0.09508(41)	1.972(12)
16	0.9	0.162	27500	0.10535(47)	1.8414(89)
16	0.9	0.175	32500	0.11416(43)	1.7132(88)
16	1.1	0.065	27500	0.10665(56)	2.425(15)
16	1.1	0.0757	27500	0.12011(58)	2.2015(94)
16	1.1	0.0864	27500	0.13188(61)	2.0145(100)
16	1.1	0.0971	27500	0.14561(57)	1.8611(93)
16	1.1	0.108	27500	0.16002(54)	1.7196(91)
16	1.1	0.119	27500	0.17117(50)	1.6315(80)
16	1.1	0.129	27500	0.18181(63)	1.5319(83)
16	1.1	0.14	32500	0.19471(57)	1.4492(66)
16	1.4	0.046	27500	0.27976(89)	1.4417(67)
16	1.4	0.0559	27500	0.2937(10)	1.3616(60)
16	1.4	0.0657	27500	0.30574(81)	1.3016(56)
16	1.4	0.0756	27500	0.31694(91)	1.2501(55)
16	1.4	0.0854	27500	0.32570(73)	1.1993(57)
16	1.4	0.0953	27500	0.33566(87)	1.1369(62)
16	1.4	0.105	27500	0.34392(81)	1.0995(56)
16	1.4	0.115	27500	0.35142(82)	1.0763(47)
16	1.9	0.027	32500	0.4837(11)	0.9624(34)
16	1.9	0.0374	27500	0.4908(12)	0.9505(52)
16	1.9	0.0479	27500	0.4967(10)	0.9159(38)
16	1.9	0.0583	27500	0.5014(10)	0.9015(43)
16	1.9	0.0687	27500	0.50642(87)	0.8913(49)
16	1.9	0.0791	27500	0.51081(98)	0.8620(43)
16	1.9	0.0896	27500	0.51567(89)	0.8409(36)
16	1.9	0.1	27500	0.52006(93)	0.8312(43)
16	2.3	0.0255	27500	0.5771(15)	0.8127(36)
16	2.3	0.0344	27500	0.5813(13)	0.7993(37)
16	2.3	0.0434	27500	0.5833(11)	0.7922(37)
16	2.3	0.0523	27500	0.5867(12)	0.7756(34)
16	2.3	0.0612	27500	0.5908(11)	0.7666(40)
16	2.3	0.0701	27500	0.5916(10)	0.7541(36)
16	2.3	0.0791	27500	0.59579(99)	0.7477(36)
16	2.3	0.088	27500	0.5991(11)	0.7277(37)
16	2.8	0.024	37500	0.6527(16)	0.7031(23)
16	2.8	0.0313	27500	0.6547(15)	0.7010(31)
16	2.8	0.0386	27500	0.6573(14)	0.6917(33)
16	2.8	0.0459	27500	0.6590(11)	0.6792(25)
16	2.8	0.0531	27500	0.6600(10)	0.6768(35)
16	2.8	0.0604	27500	0.6623(11)	0.6669(36)
16	2.8	0.0677	27500	0.66301(87)	0.6619(28)
16	2.8	0.075	32500	0.66560(90)	0.6518(32)
16	1.3	0.005	16500	0.14106(93)	2.586(17)
16	1.3	0.02	11500	0.1602(13)	2.276(14)
16	1.3	0.035	11500	0.1885(13)	1.927(12)
16	1.3	0.05	11500	0.2148(12)	1.7069(99)
16	1.3	0.065	11500	0.2373(14)	1.529(13)
16	1.3	0.08	11500	0.2567(12)	1.418(12)
16	1.3	0.095	11500	0.2757(13)	1.295(11)
16	1.3	0.11	16500	0.29120(87)	1.2291(69)

Table 7: Measured M_0, g_2 for the calculations $h \neq 0$, $L/a = 16$.

L/a	β	h	sweeps	M_0	g_2
24	0.1	0.24	27500	0.010181(93)	2.151(20)
24	0.1	0.279	27500	0.013315(88)	1.848(18)
24	0.1	0.317	27500	0.01633(10)	1.631(17)
24	0.1	0.356	27500	0.02028(12)	1.458(11)
24	0.1	0.394	27500	0.02412(15)	1.298(11)
24	0.1	0.433	27500	0.02847(14)	1.181(10)
24	0.1	0.471	27500	0.03319(16)	1.1101(74)
24	0.1	0.51	27500	0.03809(14)	1.0208(98)
24	0.3	0.12	27500	0.006619(68)	3.683(49)
24	0.3	0.134	27500	0.007675(70)	3.259(35)
24	0.3	0.149	27500	0.008734(77)	2.902(31)
24	0.3	0.163	27500	0.009993(83)	2.695(27)
24	0.3	0.177	27500	0.011118(95)	2.491(25)
24	0.3	0.191	27500	0.01245(11)	2.265(19)
24	0.3	0.206	27500	0.014237(97)	2.121(16)
24	0.3	0.22	27500	0.01535(12)	1.989(13)
24	0.5	0.1	27500	0.009727(80)	3.529(39)
24	0.5	0.112	27500	0.01100(11)	3.238(34)
24	0.5	0.124	27500	0.01236(12)	2.949(29)
24	0.5	0.136	27500	0.014259(96)	2.661(23)
24	0.5	0.147	27500	0.01602(12)	2.466(23)
24	0.5	0.159	27500	0.01779(12)	2.291(19)
24	0.5	0.171	27500	0.01984(14)	2.122(18)
24	0.5	0.183	32500	0.02198(14)	2.019(12)
24	0.6	0.085	27500	0.01108(12)	3.731(41)
24	0.6	0.0964	27500	0.01264(11)	3.314(31)
24	0.6	0.108	27500	0.01464(11)	3.032(29)
24	0.6	0.119	27500	0.01684(13)	2.719(26)
24	0.6	0.131	27500	0.01918(14)	2.492(20)
24	0.6	0.142	27500	0.02172(17)	2.271(19)
24	0.6	0.154	27500	0.02467(14)	2.091(15)
24	0.6	0.165	27500	0.02692(18)	1.973(14)
24	0.7	0.07	37500	0.01281(10)	3.994(44)
24	0.7	0.0807	27500	0.01479(13)	3.476(36)
24	0.7	0.0914	27500	0.01746(15)	3.077(29)
24	0.7	0.102	27500	0.02043(17)	2.774(23)
24	0.7	0.113	27500	0.02316(18)	2.511(17)
24	0.7	0.124	27500	0.02604(18)	2.315(17)
24	0.7	0.134	27500	0.02966(18)	2.132(15)
24	0.7	0.145	27500	0.03331(18)	1.959(12)
24	0.9	0.05	27500	0.02170(17)	3.908(33)
24	0.9	0.0594	27500	0.02574(17)	3.330(25)
24	0.9	0.0689	27500	0.03028(24)	2.958(27)
24	0.9	0.0783	27500	0.03551(26)	2.599(18)
24	0.9	0.0877	27500	0.04170(25)	2.324(14)
24	0.9	0.0971	27500	0.04661(27)	2.124(12)
24	0.9	0.107	27500	0.05385(23)	1.934(11)
24	0.9	0.116	27500	0.05915(27)	1.807(11)
24	1.0	0.045	27500	0.03188(23)	3.532(26)
24	1.0	0.0457	27500	0.03248(25)	3.499(27)
24	1.0	0.0464	27500	0.03306(25)	3.443(27)
24	1.0	0.0471	27500	0.03338(28)	3.355(26)
24	1.0	0.0477	27500	0.03443(25)	3.297(23)
24	1.0	0.0484	27500	0.03480(24)	3.293(24)
24	1.0	0.0491	27500	0.03536(27)	3.256(24)
24	1.0	0.0498	27500	0.03540(28)	3.203(26)
24	1.1	0.04	27500	0.05007(34)	2.984(21)
24	1.1	0.0469	27500	0.05939(38)	2.620(20)
24	1.1	0.0537	27500	0.06735(39)	2.361(13)
24	1.1	0.0606	27500	0.07694(35)	2.132(11)
24	1.1	0.0674	27500	0.08616(33)	1.956(10)
24	1.1	0.0743	27500	0.09662(43)	1.7896(94)
24	1.1	0.0811	27500	0.10512(48)	1.6726(96)
24	1.1	0.088	27500	0.11434(44)	1.5755(80)
24	1.3	0.024	27500	0.11444(53)	2.260(12)
24	1.3	0.0291	27500	0.12720(52)	2.0659(98)
24	1.3	0.0343	27500	0.14226(63)	1.8778(98)
24	1.3	0.0394	27500	0.15528(60)	1.7330(85)
24	1.3	0.0446	27500	0.16650(62)	1.6330(78)
24	1.3	0.0497	27500	0.17834(60)	1.5253(65)
24	1.3	0.0549	27500	0.18953(55)	1.4282(74)
24	1.3	0.06	27500	0.19867(63)	1.3656(71)
24	1.5	0.017	27500	0.24295(84)	1.4849(68)
24	1.5	0.0217	27500	0.25356(84)	1.4107(64)
24	1.5	0.0264	27500	0.26507(77)	1.3353(75)
24	1.5	0.0311	27500	0.27578(75)	1.2786(51)
24	1.5	0.0359	27500	0.28420(77)	1.2300(54)
24	1.5	0.0406	27500	0.29269(76)	1.1834(53)
24	1.5	0.0453	27500	0.30102(83)	1.1431(60)
24	1.5	0.05	27500	0.30895(66)	1.1020(52)
24	1.7	0.015	27500	0.3519(10)	1.1587(48)
24	1.7	0.0192	27500	0.3578(10)	1.1202(47)
24	1.7	0.0234	27500	0.36533(91)	1.0909(53)
24	1.7	0.0276	27500	0.3704(10)	1.0603(51)
24	1.7	0.0319	27500	0.37651(77)	1.0295(40)
24	1.7	0.0361	27500	0.38126(76)	1.0116(47)
24	1.7	0.0403	27500	0.38625(71)	0.9808(41)
24	1.7	0.0445	27500	0.39144(81)	0.9648(37)
24	1.9	0.013	27500	0.4248(13)	1.0093(47)
24	1.9	0.0167	27500	0.4305(11)	0.9813(51)
24	1.9	0.0204	27500	0.4334(11)	0.9650(45)
24	1.9	0.0241	27500	0.4375(11)	0.9441(37)
24	1.9	0.0279	27500	0.4405(11)	0.9281(34)
24	1.9	0.0316	27500	0.44609(76)	0.9127(40)
24	1.9	0.0353	27500	0.44841(76)	0.8981(44)
24	1.9	0.039	27500	0.45266(90)	0.8881(38)
24	2.3	0.012	27500	0.5269(12)	0.8387(29)
24	2.3	0.0156	27500	0.5306(11)	0.8187(48)
24	2.3	0.0191	27500	0.5327(12)	0.8045(29)
24	2.3	0.0227	27500	0.5354(10)	0.7991(35)
24	2.3	0.0263	27500	0.53791(98)	0.7900(33)
24	2.3	0.0299	27500	0.54050(84)	0.7835(36)
24	2.3	0.0334	27500	0.54336(85)	0.7716(36)
24	2.3	0.037	27500	0.54467(84)	0.7523(31)
24	2.7	0.011	32500	0.5965(13)	0.7328(27)
24	2.7	0.0144	27500	0.5992(10)	0.7219(30)
24	2.7	0.0179	27500	0.6011(13)	0.7127(31)
24	2.7	0.0213	27500	0.60199(94)	0.7147(37)
24	2.7	0.0247	27500	0.60405(99)	0.6996(35)
24	2.7	0.0281	27500	0.60520(91)	0.7006(39)
24	2.7	0.0316	27500	0.60766(100)	0.6887(37)
24	2.7	0.035	27500	0.60982(92)	0.6808(35)
24	3.1	0.01	27500	0.6485(17)	0.6651(31)
24	3.1	0.0133	27500	0.6497(14)	0.6549(27)
24	3.1	0.0166	27500	0.6509(11)	0.6525(35)
24	3.1	0.0199	27500	0.6523(11)	0.6447(24)
24	3.1	0.0231	27500	0.6531(11)	0.6515(30)
24	3.1	0.0264	27500	0.65539(94)	0.6346(28)
24	3.1	0.0297	27500	0.6570(10)	0.6228(31)
24	3.1	0.033	27500	0.6571(11)	0.6283(32)

Table 8: Measured M_0, g_2 for the calculations $h \neq 0$, $L/a = 24$.

L/a	β	h	sweeps	M_0	g_2
32	0.2	0.089	24500	0.002785(45)	4.054(70)
32	0.2	0.101	19500	0.003319(45)	3.584(52)
32	0.2	0.114	19500	0.003799(45)	3.154(33)
32	0.2	0.126	19500	0.004402(57)	2.859(36)
32	0.2	0.138	19500	0.004919(63)	2.696(36)
32	0.2	0.15	19500	0.005883(74)	2.371(34)
32	0.2	0.163	19500	0.006437(61)	2.234(23)
32	0.2	0.175	24500	0.007286(55)	2.016(21)
32	0.25	0.085	19500	0.003073(44)	4.069(83)
32	0.25	0.0957	19500	0.003567(55)	3.593(60)
32	0.25	0.106	19500	0.004107(50)	3.138(51)
32	0.25	0.117	19500	0.004602(52)	2.917(37)
32	0.25	0.128	19500	0.005242(58)	2.657(30)
32	0.25	0.139	19500	0.005937(62)	2.431(31)
32	0.25	0.149	19500	0.006360(74)	2.342(33)
32	0.25	0.16	19500	0.007281(76)	2.186(23)
32	0.3	0.08	19500	0.003154(48)	4.258(77)
32	0.3	0.0911	19500	0.003660(54)	3.715(72)
32	0.3	0.102	19500	0.004361(51)	3.255(41)
32	0.3	0.113	19500	0.004990(61)	2.930(51)
32	0.3	0.125	19500	0.005705(80)	2.653(33)
32	0.3	0.136	19500	0.006551(70)	2.399(32)
32	0.3	0.147	19500	0.007402(79)	2.282(26)
32	0.3	0.158	19500	0.008141(84)	2.099(21)
32	0.4	0.076	24500	0.003973(66)	4.033(84)
32	0.4	0.0866	19500	0.004675(55)	3.473(50)
32	0.4	0.0971	19500	0.005339(73)	3.165(37)
32	0.4	0.108	19500	0.006399(69)	2.760(33)
32	0.4	0.118	19500	0.007244(80)	2.556(27)
32	0.4	0.129	19500	0.008108(84)	2.335(26)
32	0.4	0.139	19500	0.00930(10)	2.144(23)
32	0.4	0.15	24500	0.010381(70)	2.038(19)
32	0.5	0.069	19500	0.004843(69)	4.027(64)
32	0.5	0.0784	19500	0.005611(64)	3.484(49)
32	0.5	0.0879	19500	0.006552(91)	3.059(38)
32	0.5	0.0973	19500	0.007589(88)	2.814(35)
32	0.5	0.107	19500	0.008541(83)	2.612(33)
32	0.5	0.116	19500	0.009675(99)	2.375(31)
32	0.5	0.126	19500	0.01121(12)	2.128(22)
32	0.5	0.135	19500	0.012426(97)	2.026(19)
32	0.6	0.062	19500	0.006012(94)	3.905(76)
32	0.6	0.0703	19500	0.007032(83)	3.409(49)
32	0.6	0.0786	19500	0.007920(86)	3.150(42)
32	0.6	0.0869	19500	0.009173(75)	2.832(28)
32	0.6	0.0951	19500	0.01052(11)	2.550(30)
32	0.6	0.103	19500	0.011851(90)	2.383(25)
32	0.6	0.112	19500	0.01325(12)	2.203(24)
32	0.6	0.12	19500	0.01490(13)	2.032(21)
32	0.7	0.055	24500	0.007697(79)	3.931(46)
32	0.7	0.0624	19500	0.008642(89)	3.492(46)
32	0.7	0.0699	19500	0.01010(12)	3.142(42)
32	0.7	0.0773	19500	0.01158(13)	2.839(33)
32	0.7	0.0847	19500	0.01351(12)	2.526(24)
32	0.7	0.0921	19500	0.01501(11)	2.336(23)
32	0.7	0.0996	19500	0.01691(13)	2.151(21)
32	0.7	0.107	24500	0.01881(15)	2.032(18)
32	0.8	0.0482	19500	0.01016(13)	3.827(54)
32	0.8	0.0547	19500	0.01151(13)	3.418(32)
32	0.8	0.0611	19500	0.01343(12)	3.064(42)
32	0.8	0.0676	19500	0.01518(15)	2.741(29)
32	0.8	0.0741	19500	0.01719(13)	2.549(22)
32	0.8	0.0806	19500	0.01953(20)	2.316(23)
32	0.8	0.087	19500	0.02184(19)	2.149(20)
32	0.8	0.0935	19500	0.02424(17)	2.017(14)
32	0.9	0.0414	19500	0.01343(14)	3.756(41)
32	0.9	0.0469	19500	0.01589(17)	3.252(43)
32	0.9	0.0524	19500	0.01808(18)	2.961(28)
32	0.9	0.0579	19500	0.02027(19)	2.707(27)
32	0.9	0.0635	19500	0.02348(20)	2.439(22)
32	0.9	0.069	19500	0.02585(17)	2.266(19)
32	0.9	0.0745	19500	0.02866(21)	2.144(15)
32	0.9	0.08	19500	0.03202(22)	1.990(16)
32	1.0	0.0347	19500	0.01946(20)	3.576(38)
32	1.0	0.0396	19500	0.02181(17)	3.215(30)
32	1.0	0.0445	19500	0.02509(19)	2.896(22)
32	1.0	0.0494	19500	0.02919(23)	2.597(26)
32	1.0	0.0543	19500	0.03269(23)	2.375(18)
32	1.0	0.0592	19500	0.03670(25)	2.2077(97)
32	1.0	0.0641	19500	0.04102(23)	2.054(16)
32	1.0	0.069	19500	0.04498(30)	1.933(14)
32	1.1	0.028	19500	0.02830(28)	3.449(27)
32	1.1	0.0323	19500	0.03255(23)	3.002(26)
32	1.1	0.0366	19500	0.03719(30)	2.730(20)
32	1.1	0.0409	19500	0.04214(30)	2.455(16)
32	1.1	0.0451	19500	0.04782(36)	2.283(15)
32	1.1	0.0494	19500	0.05322(31)	2.081(18)
32	1.1	0.0537	19500	0.05949(31)	1.924(12)
32	1.1	0.058	19500	0.06453(41)	1.813(11)
32	1.2	0.0234	59500	0.04596(20)	2.976(15)
32	1.2	0.0272	19500	0.05325(33)	2.616(20)
32	1.2	0.031	19500	0.06135(41)	2.389(15)
32	1.2	0.0348	19500	0.06810(39)	2.182(14)
32	1.2	0.0386	19500	0.07570(50)	1.982(14)
32	1.2	0.0424	19500	0.08380(43)	1.852(10)
32	1.2	0.0462	19500	0.09218(48)	1.711(12)
32	1.2	0.05	19500	0.09890(47)	1.627(11)
32	1.3	0.0188	23500	0.07716(48)	2.499(15)
32	1.3	0.022	23500	0.08677(42)	2.259(14)
32	1.3	0.0251	23500	0.09716(46)	2.057(13)
32	1.3	0.0283	23500	0.10642(50)	1.908(11)
32	1.3	0.0315	23500	0.11647(52)	1.7592(96)
32	1.3	0.0347	23500	0.12530(55)	1.6452(81)
32	1.3	0.0378	23500	0.13457(55)	1.5505(81)
32	1.3	0.041	23500	0.14254(49)	1.4793(80)
32	1.4	0.0142	24500	0.12256(61)	2.087(10)
32	1.4	0.017	19500	0.13593(70)	1.889(12)
32	1.4	0.0197	19500	0.14720(62)	1.737(10)
32	1.4	0.0225	59500	0.15814(38)	1.6446(54)
32	1.4	0.0253	19500	0.16917(57)	1.5354(83)
32	1.4	0.0281	19500	0.17787(73)	1.4632(73)
32	1.4	0.0308	19500	0.18564(57)	1.3886(78)
32	1.4	0.0336	24500	0.19367(67)	1.3406(61)
32	1.55	0.0131	19500	0.2325(10)	1.4138(72)
32	1.55	0.0158	19500	0.24177(91)	1.3264(78)
32	1.55	0.0185	19500	0.24995(71)	1.2880(87)
32	1.55	0.0212	19500	0.25748(95)	1.2343(71)
32	1.55	0.024	19500	0.26612(87)	1.1874(51)
32	1.55	0.0267	19500	0.27277(74)	1.1459(77)
32	1.55	0.0294	19500	0.27988(77)	1.1076(67)
32	1.55	0.0321	19500	0.28508(90)	1.0827(79)
32	1.75	0.0121	19500	0.3373(10)	1.1039(63)
32	1.75	0.0148	19500	0.3434(10)	1.0718(63)
32	1.75	0.0175	19500	0.34893(89)	1.0449(59)
32	1.75	0.0202	19500	0.3530(11)	1.0168(51)
32	1.75	0.0229	19500	0.35756(95)	0.9851(60)
32	1.75	0.0256	19500	0.36155(94)	0.9781(59)
32	1.75	0.0283	19500	0.3679(10)	0.9452(54)
32	1.75	0.031	19500	0.37164(76)	0.9260(43)
32	1.9	0.011	19500	0.3903(14)	1.0077(51)
32	1.9	0.0136	19500	0.3953(14)	0.9841(63)
32	1.9	0.0161	19500	0.3998(11)	0.9536(46)
32	1.9	0.0187	19500	0.40364(91)	0.9366(42)
32	1.9	0.0213	19500	0.40859(94)	0.9273(48)
32	1.9	0.0239	19500	0.41360(85)	0.8961(42)
32	1.9	0.0264	19500	0.41694(89)	0.8837(47)
32	1.9	0.029	19500	0.41940(95)	0.8661(38)
32	2.3	0.009	19500	0.4952(15)	0.8352(48)
32	2.3	0.0111	19500	0.4983(13)	0.8278(49)
32	2.3	0.0133	19500	0.5012(12)	0.8170(40)
32	2.3	0.0154	19500	0.5029(11)	0.8082(41)
32	2.3	0.0176	19500	0.5062(10)	0.7859(37)
32	2.3	0.0197	19500	0.5095(11)	0.7786(36)
32	2.3	0.0219	19500	0.51063(79)	0.7682(40)
32	2.3	0.024	19500	0.51339(88)	0.7536(43)
32	2.8	0.0065	31500	0.5825(15)	0.7257(29)
32	2.8	0.00829	19500	0.5841(14)	0.7166(32)
32	2.8	0.0101	19500	0.5853(14)	0.7129(45)
32	2.8	0.0119	19500	0.5876(12)	0.7047(42)
32	2.8	0.0136	19500	0.5883(13)	0.6938(38)
32	2.8	0.0154	19500	0.5908(12)	0.6865(40)
32	2.8	0.0172	19500	0.59125(98)	0.6852(36)
32	2.8	0.019	24500	0.59415(85)	0.6763(36)

Table 9: Measured M_0, g_2 for the calculations $h \neq 0$, $L/a = 32$.

L/a	β	h	sweeps	M_0	g_2
48	0.5	0.0455	24500	0.002180(32)	4.092(72)
48	0.5	0.0517	19500	0.002486(37)	3.586(56)
48	0.5	0.0579	19500	0.002908(48)	3.168(50)
48	0.5	0.0641	19500	0.003308(50)	2.882(49)
48	0.5	0.0704	19500	0.003796(39)	2.561(36)
48	0.5	0.0766	19500	0.004314(50)	2.378(25)
48	0.5	0.0828	19500	0.004842(52)	2.180(27)
48	0.5	0.089	19500	0.005430(59)	2.066(23)
48	0.6	0.04	19500	0.002541(42)	4.246(84)
48	0.6	0.0453	19500	0.002989(44)	3.578(64)
48	0.6	0.0506	19500	0.003470(46)	3.255(47)
48	0.6	0.0559	19500	0.003974(42)	2.926(39)
48	0.6	0.0611	19500	0.004479(47)	2.721(36)
48	0.6	0.0664	19500	0.005088(72)	2.463(34)
48	0.6	0.0717	19500	0.005666(58)	2.373(27)
48	0.6	0.077	19500	0.006407(77)	2.094(24)
48	0.7	0.033	19500	0.003045(44)	4.314(91)
48	0.7	0.0376	19500	0.003497(46)	3.891(65)
48	0.7	0.0421	19500	0.004000(63)	3.480(58)
48	0.7	0.0467	19500	0.004494(62)	3.152(50)
48	0.7	0.0513	19500	0.005258(69)	2.807(38)
48	0.7	0.0559	19500	0.005986(56)	2.547(31)
48	0.7	0.0604	19500	0.006524(63)	2.431(29)
48	0.7	0.065	19500	0.007345(79)	2.224(27)
48	0.9	0.027	24500	0.006015(64)	3.889(55)
48	0.9	0.0309	19500	0.006951(87)	3.498(43)
48	0.9	0.0347	19500	0.00807(10)	3.061(38)
48	0.9	0.0386	19500	0.009232(75)	2.784(24)
48	0.9	0.0424	19500	0.01064(10)	2.544(25)
48	0.9	0.0463	19500	0.01205(11)	2.326(24)
48	0.9	0.0501	19500	0.01351(14)	2.181(20)
48	0.9	0.054	24500	0.01531(14)	1.991(19)
48	1.0	0.021	19500	0.008054(97)	4.083(59)
48	1.0	0.0243	19500	0.00939(11)	3.629(54)
48	1.0	0.0276	19500	0.01066(10)	3.270(37)
48	1.0	0.0309	19500	0.01237(13)	2.857(34)
48	1.0	0.0341	19500	0.01386(12)	2.646(25)
48	1.0	0.0374	19500	0.01590(17)	2.371(25)
48	1.0	0.0407	19500	0.01802(13)	2.170(19)
48	1.0	0.044	19500	0.02032(16)	2.047(18)
48	1.1	0.016	19500	0.01112(12)	4.297(43)
48	1.1	0.0186	19500	0.01310(15)	3.616(44)
48	1.1	0.0211	19500	0.01504(13)	3.281(37)
48	1.1	0.0237	19500	0.01702(15)	2.982(33)
48	1.1	0.0263	19500	0.01914(17)	2.690(27)
48	1.1	0.0289	19500	0.02196(18)	2.459(21)
48	1.1	0.0314	19500	0.02421(17)	2.306(15)
48	1.1	0.034	19500	0.02762(19)	2.102(16)
48	1.3	0.011	24500	0.03445(26)	3.220(23)
48	1.3	0.0129	19500	0.04025(31)	2.812(18)
48	1.3	0.0147	19500	0.04552(32)	2.547(18)
48	1.3	0.0166	19500	0.05159(37)	2.309(13)
48	1.3	0.0184	19500	0.05851(31)	2.086(12)
48	1.3	0.0203	19500	0.06455(34)	1.908(15)
48	1.3	0.0221	19500	0.07030(38)	1.824(12)
48	1.3	0.024	24500	0.07812(41)	1.667(11)
48	1.5	0.0086	19500	0.13540(50)	1.743(10)
48	1.5	0.0102	19500	0.14673(60)	1.628(11)
48	1.5	0.0119	19500	0.15796(67)	1.4894(81)
48	1.5	0.0135	19500	0.16674(67)	1.4050(81)
48	1.5	0.0151	19500	0.17640(72)	1.3181(84)
48	1.5	0.0167	19500	0.18330(56)	1.2621(75)
48	1.5	0.0184	19500	0.19182(73)	1.2078(74)
48	1.5	0.02	19500	0.19881(64)	1.1627(78)
48	1.7	0.0062	19500	0.25321(85)	1.2583(66)
48	1.7	0.00754	19500	0.26045(91)	1.1909(63)
48	1.7	0.00889	19500	0.26765(87)	1.1541(72)
48	1.7	0.0102	19500	0.27235(75)	1.1196(67)
48	1.7	0.0116	19500	0.27927(69)	1.0783(62)
48	1.7	0.0129	19500	0.28354(81)	1.0434(59)
48	1.7	0.0143	19500	0.28804(72)	1.0094(49)
48	1.7	0.0156	19500	0.29311(78)	0.9998(54)
48	1.9	0.0038	19500	0.3304(10)	1.0848(63)
48	1.9	0.0049	19500	0.3347(12)	1.0460(45)
48	1.9	0.006	29500	0.33983(90)	1.0282(53)
48	1.9	0.0071	19500	0.3453(11)	1.0067(53)
48	1.9	0.0082	19500	0.34805(92)	0.9862(60)
48	1.9	0.0093	19500	0.35159(88)	0.9665(55)
48	1.9	0.0104	79500	0.35518(44)	0.9434(24)
48	1.9	0.0115	19500	0.35876(74)	0.9339(55)
48	2.3	0.0035	19500	0.4471(15)	0.8676(43)
48	2.3	0.00443	19500	0.4509(13)	0.8639(51)
48	2.3	0.00536	19500	0.4538(14)	0.8367(60)
48	2.3	0.00629	19500	0.4557(11)	0.8300(43)
48	2.3	0.00721	19500	0.45724(92)	0.8219(42)
48	2.3	0.00814	19500	0.4611(11)	0.8050(39)
48	2.3	0.00907	19500	0.46155(82)	0.8060(43)
48	2.3	0.01	19500	0.46472(92)	0.7847(48)
48	2.7	0.0032	19500	0.5264(17)	0.7609(39)
48	2.7	0.00409	19500	0.5289(14)	0.7469(40)
48	2.7	0.00497	19500	0.5303(13)	0.7422(40)
48	2.7	0.00586	19500	0.5328(13)	0.7331(52)
48	2.7	0.00674	19500	0.5353(12)	0.7309(42)
48	2.7	0.00763	19500	0.53540(93)	0.7135(46)
48	2.7	0.00851	19500	0.5366(12)	0.7118(47)
48	2.7	0.0094	19500	0.53922(90)	0.7027(39)
48	3.1	0.0029	19500	0.5860(17)	0.6794(28)
48	3.1	0.00369	19500	0.5870(15)	0.6799(40)
48	3.1	0.00447	19500	0.5886(11)	0.6675(35)
48	3.1	0.00526	19500	0.5898(12)	0.6577(35)
48	3.1	0.00604	19500	0.5911(12)	0.6561(37)
48	3.1	0.00683	19500	0.5925(10)	0.6595(37)
48	3.1	0.00761	19500	0.5935(11)	0.6528(33)
48	3.1	0.0084	24500	0.59403(89)	0.6447(27)

Table 10: Measured M_0, g_2 for the calculations $h \neq 0$, $L/a = 48$.

L/a	β	h	sweeps	M_0	g_2
64	0.5	0.034	28500	0.001234(19)	4.042(69)
64	0.5	0.0387	23500	0.001399(21)	3.543(58)
64	0.5	0.0434	23500	0.001594(29)	3.225(61)
64	0.5	0.0481	23500	0.001852(21)	2.811(47)
64	0.5	0.0529	23500	0.002128(27)	2.610(36)
64	0.5	0.0576	23500	0.002409(28)	2.389(27)
64	0.5	0.0623	23500	0.002677(32)	2.253(34)
64	0.5	0.067	28500	0.003089(33)	2.120(21)
64	0.6	0.03	15500	0.001448(32)	4.05(11)
64	0.6	0.0343	15500	0.001695(30)	3.601(84)
64	0.6	0.0386	15500	0.001927(31)	3.269(61)
64	0.6	0.0429	15500	0.002299(47)	2.855(51)
64	0.6	0.0471	15500	0.002608(46)	2.674(49)
64	0.6	0.0514	15500	0.002972(36)	2.433(35)
64	0.6	0.0557	15500	0.003373(41)	2.205(31)
64	0.6	0.06	15500	0.003769(47)	2.046(31)
64	0.7	0.026	15500	0.001832(40)	4.147(98)
64	0.7	0.03	15500	0.002146(45)	3.743(91)
64	0.7	0.034	15500	0.002486(45)	3.252(82)
64	0.7	0.038	15500	0.002859(49)	2.914(53)
64	0.7	0.042	15500	0.003439(59)	2.609(46)
64	0.7	0.046	15500	0.003811(49)	2.406(28)
64	0.7	0.05	15500	0.004402(53)	2.212(29)
64	0.7	0.054	15500	0.004963(59)	2.002(22)
64	0.8	0.023	20500	0.002430(41)	4.160(80)
64	0.8	0.0265	15500	0.002751(40)	3.653(56)
64	0.8	0.0299	15500	0.003186(50)	3.210(67)
64	0.8	0.0334	15500	0.003810(66)	2.807(45)
64	0.8	0.0369	15500	0.004292(60)	2.678(35)
64	0.8	0.0404	15500	0.005124(62)	2.379(32)
64	0.8	0.0438	15500	0.005763(81)	2.193(30)
64	0.8	0.0473	15500	0.006431(84)	2.031(25)
64	0.9	0.02	15500	0.003371(48)	4.052(86)
64	0.9	0.023	15500	0.003903(61)	3.559(48)
64	0.9	0.026	15500	0.004572(61)	3.133(45)
64	0.9	0.029	15500	0.005279(78)	2.815(43)
64	0.9	0.032	15500	0.005932(87)	2.548(33)
64	0.9	0.035	15500	0.007086(81)	2.302(20)
64	0.9	0.038	15500	0.008098(67)	2.117(24)
64	0.9	0.041	15500	0.00893(10)	2.015(27)
64	1.0	0.017	15500	0.004887(69)	3.955(68)
64	1.0	0.0194	15500	0.005621(79)	3.499(64)
64	1.0	0.0219	15500	0.006547(86)	3.038(39)
64	1.0	0.0243	15500	0.007600(78)	2.688(33)
64	1.0	0.0267	15500	0.008784(91)	2.477(30)
64	1.0	0.0291	15500	0.00972(11)	2.327(29)
64	1.0	0.0316	15500	0.01093(16)	2.137(30)
64	1.0	0.034	15500	0.01249(13)	1.989(23)
64	1.1	0.0138	20500	0.00732(10)	3.828(60)
64	1.1	0.0158	15500	0.00827(11)	3.425(47)
64	1.1	0.0179	15500	0.00986(11)	3.007(36)
64	1.1	0.0199	15500	0.01126(13)	2.692(35)
64	1.1	0.0219	15500	0.01300(13)	2.512(26)
64	1.1	0.0239	15500	0.01468(15)	2.294(26)
64	1.1	0.026	15500	0.01640(16)	2.106(21)
64	1.1	0.028	20500	0.01861(14)	1.939(18)
64	1.2	0.0112	15500	0.01202(16)	3.700(50)
64	1.2	0.0129	15500	0.01417(14)	3.152(24)
64	1.2	0.0146	15500	0.01650(19)	2.787(32)
64	1.2	0.0163	15500	0.01901(21)	2.583(27)
64	1.2	0.0179	15500	0.02151(19)	2.314(21)
64	1.2	0.0196	15500	0.02403(19)	2.124(16)
64	1.2	0.0213	15500	0.02725(18)	1.969(15)
64	1.2	0.023	15500	0.03040(22)	1.840(18)
64	1.3	0.0086	15500	0.02095(21)	3.395(48)
64	1.3	0.00994	15500	0.02482(23)	2.946(31)
64	1.3	0.0113	15500	0.02880(27)	2.629(31)
64	1.3	0.0126	15500	0.03273(31)	2.370(25)
64	1.3	0.014	15500	0.03733(31)	2.190(14)
64	1.3	0.0153	15500	0.04098(34)	2.003(16)
64	1.3	0.0167	15500	0.04600(29)	1.838(14)
64	1.3	0.018	15500	0.05103(27)	1.727(16)
64	1.4	0.006	20500	0.03910(29)	2.969(17)
64	1.4	0.007	20500	0.04398(32)	2.683(18)
64	1.4	0.008	15500	0.05037(43)	2.396(21)
64	1.4	0.009	20500	0.05600(34)	2.174(17)
64	1.4	0.01	15500	0.06288(35)	2.035(15)
64	1.4	0.011	20500	0.06829(40)	1.858(13)
64	1.4	0.012	15500	0.07520(49)	1.768(13)
64	1.4	0.013	20500	0.08115(35)	1.6268(85)
64	1.55	0.0055	19500	0.12517(57)	1.7088(90)
64	1.55	0.00643	19500	0.13636(62)	1.5701(80)
64	1.55	0.00736	19500	0.14589(47)	1.4764(87)
64	1.55	0.00829	19500	0.15321(57)	1.3875(64)
64	1.55	0.00921	19500	0.16049(52)	1.3166(83)
64	1.55	0.0101	19500	0.16769(43)	1.2638(65)
64	1.55	0.0111	19500	0.17444(60)	1.2001(70)
64	1.55	0.012	19500	0.17912(59)	1.1672(54)
64	1.75	0.0051	19500	0.25050(82)	1.1497(70)
64	1.75	0.00594	19500	0.25525(73)	1.1135(59)
64	1.75	0.00679	19500	0.25998(70)	1.0874(67)
64	1.75	0.00763	19500	0.26618(73)	1.0354(48)
64	1.75	0.00847	19500	0.27086(77)	1.0057(57)
64	1.75	0.00931	19500	0.27413(77)	0.9953(60)
64	1.75	0.0102	19500	0.28047(52)	0.9640(51)
64	1.75	0.011	19500	0.28368(46)	0.9289(45)
64	1.9	0.00454	15500	0.3101(11)	1.0188(78)
64	1.9	0.00532	15500	0.3147(11)	0.9899(57)
64	1.9	0.0061	15500	0.31844(97)	0.9631(52)
64	1.9	0.00688	15500	0.32290(70)	0.9495(63)
64	1.9	0.00766	15500	0.32677(80)	0.9276(49)
64	1.9	0.00844	15500	0.33011(79)	0.9041(52)
64	1.9	0.00922	15500	0.33383(85)	0.8895(55)
64	1.9	0.01	15500	0.33589(69)	0.8712(60)
64	2.3	0.0031	15500	0.4212(14)	0.8652(54)
64	2.3	0.0038	15500	0.4242(11)	0.8551(56)
64	2.3	0.0045	15500	0.4274(15)	0.8280(52)
64	2.3	0.0052	15500	0.43092(74)	0.8127(56)
64	2.3	0.0059	15500	0.4330(11)	0.8132(48)
64	2.3	0.0066	15500	0.4340(10)	0.7983(55)
64	2.3	0.0073	15500	0.43745(95)	0.7793(55)
64	2.3	0.008	15500	0.43909(78)	0.7664(54)
64	2.8	0.00164	15500	0.5156(15)	0.7521(52)
64	2.8	0.00219	15500	0.5177(17)	0.7404(48)
64	2.8	0.00274	15500	0.5200(15)	0.7318(42)
64	2.8	0.00329	15500	0.5215(11)	0.7148(38)
64	2.8	0.00385	15500	0.5218(15)	0.7159(45)
64	2.8	0.0044	15500	0.5251(13)	0.7038(32)
64	2.8	0.00495	15500	0.5266(12)	0.6934(45)
64	2.8	0.0055	15500	0.5284(13)	0.6871(49)

Table 11: Measured M_0, g_2 for the calculations $h \neq 0$, $L/a = 64$.

References

[1] Gell-Mann M, Lévy M (1960), The axial vector current in beta decay, Nuovo Cimento 16, 705

[2] Murray Gell-Mann on the sigma model, WEB of STORIES, https://www.webofstories.com/play/murray.gell-mann/89

[3] Michael E. Peskin, Daniel V. Schroeder (1995), An Introduction To Quantum Field Theory, Chapter 13.3, Perseus Books Publishing

[4] Daniel Nogradi (2012), An ideal toy model for confining, walking and conformal gauge theories: the O(3) sigma model with theta-term, High Energy Physics - Lattice (hep-lat); High Energy Physics - Theory (hep-th), JHEP 05 (2012) 089

[5] Sergio Caracciolo, Robert G. Edwards, Andrea Pelissetto, Alan D. Sokal (1992), Wolff-Type Embedding Algorithms for General Nonlinear σ-Models, Nucl.Phys. B403 (1993) 475-541

[6] Sergio Caracciolo, Robert G. Edwards, Andrea Pelissetto, Alan D. Sokal (1995), Asymptotic Scaling in the Two-Dimensional O(3) σ-Model at Correlation Length 10^5 Phys.Rev.Lett. 75 (1995) 1891-1894

[7] Kari Rummukainen lecture notes, University of Helsinki, Monte Carlo simulation methods, https://www.mv.helsinki.fi/home/rummukai/lectures/montecarlo_oulu/

[8] C++ Pseudo-random number generation, https://en.cppreference.com/w/cpp/numeric/random

[9] Numerical recipes in C, Cambridge, W H Press, Saul A. Teukolsky