Optimization of SU(2) Landau Gauge-Fixing Algorithms

Michael Müller
1530049

Department of Physics
Karl-Franzens University

Bachelor Thesis (BSc)
Supervisor: Axel Maas

April 3, 2018

Abstract

In the context of the numerical treatment of non-Abelian lattice gauge theories, Gribov copies pose an important problem. For the standard lattice Landau gauge condition the Los Alamos method (LAM) can be used for gauge fixing or a modified version that applies stochastic overrelaxation to the LAM and which shows a higher efficiency. To account for Gribov copies these methods have to be applied multiple times to the same lattice configuration and the average is evaluated, which is referred to as multi-start approach.
This work attempts to modify the gauge fixing method with stochastic overrelaxation, by selectively manipulating the probability involved during the calculations, to achieve better results in combination with the multi-start approach, with regard to the original method. In this process, a selection criterion will be identified to regulate the manipulation of the involved probability and a test algorithm shall be implemented to compare the new method to the existing one.

Contents

1 Introduction

The bosonic interactions of the standard model as well as Quantum Electrodynamics (QED) are described by gauge theories of the Yang-Mills type. The theories for the strong and weak interactions are of non-Abelian type, whereas QED is represented by an Abelian gauge theory [5]. These dynamical theories are based on a gauge principle of local phase invariance and the property of being Abelian merely states, whether the corresponding phase factors commute with each other or not [1]. However not all quantities of these theories are gauge independent, that is feature local phase invariance, which is why it is required to fix a gauge for calculating these quantities. While Abelian gauge theories permit gauge fixing by introducing local constraints (e.g. Landau gauge), this holds true for non-Abelian gauge theories no longer. Due to the geometric structure of the non-Abelian gauge group the local gauge conditions are not unique anymore, which results in the Gribov-Singer ambiguity. The independent solutions satisfying a local gauge condition are called Gribov copies [5].

In the context of numerical evaluations the lattice formulation of these theories occupies an important role, since replacing space-time by a Euclidean lattice has proven to be an efficient approach which allows for both theoretical understanding and computational analysis [3]. The lattice formulation of Quantum Chromodynamics (QCD) provides a regularization which makes the gauge group compact, so that the Gibbs average of any gauge-invariant quantity is well-defined and thus gauge fixing is, in principle, not necessary [3]. However, it is sometimes of use to consider gauge-dependent quantities on the lattice as well, which requires gauge fixing. It is therefore important to devise numerical algorithms to efficiently gauge fix a lattice configuration. At this point the Gribov-Singer ambiguity becomes important, since the existence of Gribov copies dampens the efficiency of these algorithms. Furthermore, it is usually not clear how an algorithm selects among different Gribov copies, implying that the numerical results using gauge fixing might depend on the gauge fixing algorithm. Therefore, multiple Gribov copies of the same thermalized lattice have to be produced and compared, to analyze the dependence on the Gribov-Singer ambiguity [2]. This will be referred to as multi-start approach in the rest of this paper. [2, 3]

The algorithms discussed in this paper are intended to be used for gauge fixing under the standard lattice Landau gauge condition in the SU(3) gauge theory of QCD [3], however to facilitate numerical calculations the problems are discussed within the SU(2) gauge group as in [2], and only on $dim = 2$ lattice with small lattice sizes. The Landau gauge condition on the lattice is formulated as a minimization problem for the energy of a nonlinear σ-model with disordered couplings [2]. To achieve this minimization several deterministic local algorithms have been introduced, the one relevant for this paper being the "Los Alamos" method [2]. The Los Alamos method uses local updating which entails that updates travel at each sweep only from one site to its nearest neighbors (a sweep is defined as a complete gauge transformation of the lattice, i.e. the gauge transformation of every site). This creates waiting times of the order of the square of the lattice size, until a significant change in the configuration can be observed. To decrease the required time, the local updates can be modified and one way to do this is to apply stochastic overrelaxation to the Los Alamos method [1]. [2]

The purpose of this thesis is to improve the multi-start approach applied to the Stochastic Overrelaxation Method, by varying the used probability p during the updating procedure to achieve convergence to deeper minima for the energy. The goals of this work are:

- studying the descent rate for the energy and its derivative to identify selection criteria for the variation of the probability p used with the Stochastic Overrelaxation Method,
- implementing the selection criteria for the probability together with an algorithm that uses these rules to vary the probability used with the Stochastic Overrelaxation Method after a defined number of sweeps,

[1]Stochastic overrelaxation applied to the Los Alamos method will be referred to as Stochastic Overrelaxation Method or SOM form now on;

- verifying that this kind of algorithm does indeed lead to lower average energies when used together with the multi-start method.

This paper will be structured as follows. In Section 2 the required mathematical formulas to implement the Los Alamos Method as well as the Stochastic Overrelaxation Method are introduced. Section 3 will cover the first two points in the list above, discussing the ideas on how to improve the Stochastic Overrelaxation Method combined with the multi-start approach by a selective manipulation of the probability p and the implementation. In Section 5 the results are represented, followed by the conclusion in Section 6.

2 Methods of Landau Gauge-Fixing in $SU(2)$ on the Lattice

This chapter follows closely [2,3].

This chapter covers the definition of the Landau gauge condition on the lattice in the first part and then introduces methods that are utilized to perform Landau gauge-fixing on the lattice in the $SU(2)$ case.

2.1 Landau Gauge-Fixing Condition on the Lattice

The d-dimensional lattice is defined as a fixed configuration of link variables $U_\mu(\mathbf{x}) \in SU(2)$ where the lattice sites are labeled by d-dimensional vectors $\mathbf{x}$. The constant a specifies the spacing of the lattice points and $\mathbf{e}_\mu$ is a unit vector in the positive μ-direction. The values of $d = 2$ and $a = 1$ are defined to be the same for the rest of this work [2]. In the μ-direction the physical lattice size is $L_\mu \equiv aN_\mu$ where N_μ represents the number of lattice points in the μ-direction and is be referred to as lattice size in the rest of this work. Furthermore, periodic boundary conditions are assumed for the lattice that are defined as follows

$$\mathbf{x} + L_\mu \mathbf{e}_\mu \equiv \mathbf{x}\,, \tag{1}$$

and the lattice volume is given by

$$V \equiv \prod_{\mu=1}^{d} L_\mu = a^d \prod_{\mu=1}^{d} N_\mu\,. \tag{2}$$

It can be shown that in the lattice formalism the gauge field $A_\mu(\mathbf{x})$, which is used to define the Landau gauge condition on the lattice (4), can be expressed in terms of the so-called link variables $U_\mu(\mathbf{x})$, see [3].

$$A_\mu(\mathbf{x}) \equiv \frac{1}{2ag_0}[U_\mu(\mathbf{x}) - U^\dagger_\mu(\mathbf{x})] \tag{3}$$

$$\partial_\mu A_\mu = \frac{1}{a}\sum_{\mu=1}^{d}(A_\mu(\mathbf{x}) - A_\mu(\mathbf{x} - a\mathbf{e}_\mu)) \equiv 0 \tag{4}$$

Here g_0 denotes the bare coupling constant of the lattice [2] and it is set to "1" for all following steps, to reduce the complexity (this is possible, because the results of this work are not used for any physical interpretations). In lattice field theory the Landau gauge condition is formulated as a minimization problem, *i.e.* to fix the Landau gauge the absolute minimum of the so-called minimizing function $\mathcal{E}$ needs to be found [3]. The minimizing function is defined as follows

$$\mathcal{E}(\{g\}) \equiv 1 - \frac{a^d}{2dV}\sum_{\mu=1}^{d}\sum_{\mathbf{x}}\frac{1}{2}Tr[U^{(g)}_\mu(\mathbf{x}) + U^{(g)\dagger}_\mu(\mathbf{x})]\,, \tag{5}$$

[2] a is a factor that defines the physical distance between two points of the lattice and for simplicity this value is set to 1. The dimension d of the lattice is also set to two, because it is the simplest case and it decreases the complexity of calculations, which saves computing time.

[3] The minimizing function $\mathcal{E}$ usually has multiple extrema which is why the Landau gauge condition is not unique and the Gribov-Singer ambiguity arises. This work looks into approximating the absolute minimum of $\mathcal{E}$, which defines the so-called *minimal Landau gauge* [2].

and has a lower bound of 0 and an upper bound of 2. The configuration of link variables $\{U_\mu(\mathbf{x})\}$ is kept fixed and the local gauge transformation is given by

$$U_\mu(\mathbf{x}) \to U_\mu^{(g)}(\mathbf{x}) \equiv g(\mathbf{x}) U_\mu(\mathbf{x}) g^\dagger(\mathbf{x} + a\mathbf{e}_\mu) \,, \tag{6}$$

where the $g(\mathbf{x}) \in SU(2)$ are defined at site variables, and the transformation of the whole lattice is then $\mathcal{G} \equiv \{g(\mathbf{x})\}$. If the Landau gauge-fixing condition is satisfied then the quantities

$$Q_\nu(x_\nu) \equiv \sum_{\mu \neq \nu} \sum_{x_\mu} A_\nu(\mathbf{x}) \qquad \nu = 1, \cdots, d \tag{7}$$

are constant, *i.e.* independent of x_ν, when substituting the $A_\nu(\mathbf{x})$ calculated from the gauge fixed $U_\nu(\mathbf{x})$ with (3), see [2]. Here x_μ denotes the components of a lattice vector $\mathbf{x}$.

2.2 Gauge-Fixing Algorithms on the Lattice

The algorithms discussed in this section do not use the absolute Landau gauge condition and look for an absolute minimum of the minimizing function, but rather search for local minima. There is a multitude of algorithms to find a gauge transformation $\{g(\mathbf{x})\}$ such that for a given initial lattice configuration $\{U_\mu(\mathbf{x})\}$ the gauge-transformed link variables $U_\mu^{(g)}(\mathbf{x})$ bring the function $\mathcal{E}(\{g\})$ to a local minimum. There are two algorithms relevant for this work which are presented in the following sections. The complex representation of $\mathcal{E}(\{g\})$ as in (5) is replaced by

$$\begin{aligned} \mathcal{E}(g) &= 1 - \frac{a^d}{2dV} \sum_{\mu=1}^{d} \sum_{\mathbf{x}} \frac{1}{2} Tr\, U_\mu^{(g)}(\mathbf{x}) \qquad (8)\\ &= 1 - \frac{a^d}{2dV} \sum_{\mu=1}^{d} \sum_{\mathbf{x}} \frac{1}{2} Tr[g(\mathbf{x}) U_\mu(\mathbf{x}) g^\dagger(\mathbf{x} + a\mathbf{e}_\mu)] \qquad (9) \end{aligned}$$

and the starting configuration for $\{g(\mathbf{x})\}$ is that $g(\mathbf{x}) = \mathbb{1}$ for all $\mathbf{x}$. The transition from the complex representation in (5) to the representation in (8) is possible, since the trace of a unitary $SU(2)$ matrix is always real. The following algorithms are trying to achieve the minimization of $\mathcal{E}(\{g\})$ by applying an iterative process which, from one iteration step to the next, decreases the value of the minimizing function monotonically [2].

2.2.1 Los Alamos Method

By utilizing the "single-site effective magnetic field" $h(\mathbf{y})$ one can introduce the quantity $w(\mathbf{y})$ as in (10), where $h(\mathbf{y})$ is given by (11).

$$w(\mathbf{y}) \equiv g(\mathbf{y})\, h(\mathbf{y}) \tag{10}$$

$$h(\mathbf{y}) \equiv \sum_{\mu=1}^{d} [U_\mu(\mathbf{y})\, g^\dagger(\mathbf{y} + a\mathbf{e}_\mu) + U_\mu^\dagger(\mathbf{y} - a\mathbf{e}_\mu)\, g^\dagger(\mathbf{y} + a\mathbf{e}_\mu)] \tag{11}$$

The matrices $h(\mathbf{y})$ and $w(\mathbf{y})$ can be expressed by $SU(2)$ matrices using the relations in (12) and (13):

$$h(\mathbf{y}) \equiv \sqrt{det\, h(\mathbf{y})}\, \tilde{h}(\mathbf{y}) \tag{12}$$

$$w(\mathbf{y}) \equiv \sqrt{det\, w(\mathbf{y})}\, \tilde{w}(\mathbf{y}) \tag{13}$$

Since the gauge transformations $\{g(\mathbf{x})\}$ are $SU(2)$, it follows that $h(\mathbf{y})$ and $w(\mathbf{y})$ share the same determinant and the prefactor in (12) and (13) can be denoted as

$$\mathcal{N}(\mathbf{y}) \equiv \sqrt{det\, h(\mathbf{y})} = \sqrt{det\, w(\mathbf{y})} \tag{14}$$

The trace of the $SU(2)$ matrix proportional to $w(\mathbf{y})$ will also become important in chapter 2.3, hence it will be denoted as follows, according to the notation in [2].

$$\mathcal{T}(\mathbf{y}) \equiv Tr\, \tilde{w}(\mathbf{y}) = Tr\, w(\mathbf{y}) \,/\, \mathcal{N}(\mathbf{y}) \tag{15}$$

The updates applied to the gauge transformations $g(\mathbf{y}) \Rightarrow g^{(new)}(\mathbf{y})$ to minimize the quantity $\mathcal{E}(\{g\})$ are considered to have multiplicative nature as in (16), based on [2], where $R^{(update)}(\mathbf{y})$ represents another $SU(2)$ matrix.

$$g(\mathbf{y}) \;\Rightarrow\; g^{(new)}(\mathbf{y}) \equiv R^{(update)}(\mathbf{y})\, g(\mathbf{y}) \tag{16}$$

The Los Alamos method is then implemented by setting $R^{(update)}$ to $\tilde{w}^{\dagger}(\mathbf{y})$ which results in the update relation (17). see [2].

$$g^{(new)}(\mathbf{y}) = g^{(LosAl.)}(\mathbf{y}) \equiv \tilde{h}^{\dagger}(\mathbf{y}) \tag{17}$$

2.2.2 Stochastic Overrelaxation Method

The stochastic overrelaxation method (SOM) is an extension of the Los Alamos method, where the update is not applied as in (17) for all sweeps anymore. Instead, $g^{(LosAl.)}(\mathbf{y})$ is used as an update only with the probability $1-p$ and in the rest of the cases the new update relation with $g^{(new)}(\mathbf{y}) \equiv [\tilde{w}^{\dagger}(\mathbf{y})]^2\, g(\mathbf{y})$ is applied. This leads to the following expression for the updated gauge transformations:

$$g(\mathbf{y}) \;\Rightarrow\; g^{(new)}(\mathbf{y}) = g^{(stoc)}(\mathbf{y}) \equiv \begin{cases} [\tilde{w}^{\dagger}(\mathbf{y})]^2\, g(\mathbf{y}) & \text{with probability } p \\ \\ g^{(LosAl.)}(\mathbf{y}) & \text{with probability } 1-p \end{cases} \tag{18}$$

with $0 < p < 1$. Furthermore, when $p = 0$ the stochastic overrelaxation method is equivalent to the Los Alamos method. The stochastic overrelaxation method is useful due to the fact that with the update $g^{(new)}(\mathbf{y}) \equiv [\tilde{w}^{\dagger}(\mathbf{y})]^2\, g(\mathbf{y})$ a big move is done in configuration space with probability p, compared to the Los Alamos update, and this has the potential to speed up the convergence of the value for the minimization function [2].

The updated gauge transformations $\{g(\mathbf{x})\}$ can also be expressed in terms of the "single-site effective magnetic field", when using the relation expressed in (19). This leads to the expression used to calculate $g^{(stoc)}(\mathbf{y})$ that is implemented in the algorithm for this work, see (20).

$$[\tilde{w}^{\dagger}(\mathbf{y})]^2 = \tilde{w}^{\dagger}(\mathbf{y})\, \mathcal{T}(\mathbf{y}) - \mathbb{1} \tag{19}$$

$$g^{(stoc)}(\mathbf{y}) = \begin{cases} \tilde{h}^{\dagger}(\mathbf{y})\, \mathcal{T}(\mathbf{y}) - g(\mathbf{y}) & \text{with probability } p \\ \\ \tilde{h}^{\dagger}(\mathbf{y}) & \text{with probability } 1-p \end{cases} \tag{20}$$

The length of the move $g(\mathbf{y}) \to g^{(new)}(\mathbf{y})$ in the configuration space is determined by the quantity

$$\begin{aligned} \mathcal{D}[g(\mathbf{y}), g^{(new)}(\mathbf{y})] \equiv \mathcal{D}(\mathbf{y}) &\equiv \sqrt{\frac{1}{2} Tr\left\{[g(\mathbf{y}) - g^{(new)}(\mathbf{y})][g(\mathbf{y}) - g^{(new)}(\mathbf{y})]^{\dagger}\right\}} && (21) \\ &= \sqrt{2 - Tr R^{(update)}(\mathbf{y})} && (22) \end{aligned}$$

as introduced by [2]. This quantity satisfies the defining properties of a distance function for any set of matrices and, if the $SU(2)$ matrices are interpreted as four-dimensional unit vectors, it coincides with the standard euclidean distance in $\mathbb{R}^4$ [2]. Using this expression, the difference in distance between the two possible terms in $g^{(stoc)}(\mathbf{y})$ can be determined. This gives the following (see [2]):

$$\mathcal{D}^{(LosAl.)}(\mathbf{y}) = \sqrt{2 - \mathcal{T}(\mathbf{y})} \tag{23}$$

$$\mathcal{D}^{(stoc)}(\mathbf{y}) = \sqrt{4 - \mathcal{T}^2(\mathbf{y})} = \sqrt{2 + \mathcal{T}(\mathbf{y})}\, \mathcal{D}^{(LosAl.)}(\mathbf{y}) \tag{24}$$

These expressions indicate that the updates that are selected with the probability p in (20) represent a larger move in the configuration space than the updates according to the Los Alamos Method, which are applied with the probability $1 - p$. This implies that by modifying the probability p the likelihood for "longer" moves to occur in the configuration space, with respect to the likelihood for "shorter" moves, can be manipulated. The considerations in Section 3 will make use of this property.

Tuning of the probability p:

The optimum probability p used in the calculations for the gauge transformations with SOM is dependent on the lattice size and can be tuned to maximize the speed of convergence for $\mathcal{E}(\{g\})$ [2]. For this work the tuning of the parameter p was not done as extensively as in [2]. In total six different lattice sizes were considered (2x2, 6x6, 10x10, 16x16, 20x20 and 30x30) to get an estimate on the shift of p towards larger values ($p \in [0, 1]$) with increasing lattice sizes. To find the optimum of p for a given lattice size, the value of p was varied in an interval with a step size of 1% and the boundaries of the interval were then gradually narrowed to pinpoint the optimum value. The indicator for which probability represents the optimum was how many sweeps were required to reach a predefined threshold. This threshold was set for each lattice individually, in a way that the number of sweeps, required to reach it, is in the range of 100. Beginning with the smallest lattice (2x2) the value of the optimized probability p is expected to be close to zero and with increasing lattice sizes the optimum value approximates 100% [2]. After the optimum value for the first lattice had been obtained, it was set to be the lower boundary for the starting interval of the lattice next in size. The results are shown in Table 1.

Table 1: Optimized probability values for the stochastic overrelaxation method depending on the lattice size

Lattice size	**Optimum value for p [%]**	**Error [%]**
$2x2$	8.5	1.2
$6x6$	38.2	1.6
$10x10$	45.4	0.6
$16x16$	70.8	1.3
$20x20$	85.6	0.8
$30x30$	89.5	0.7

The optimized probabilities for lattices with sizes not listed in Table 1 were extrapolated from the six values listed above, as can be seen in Figure 1.

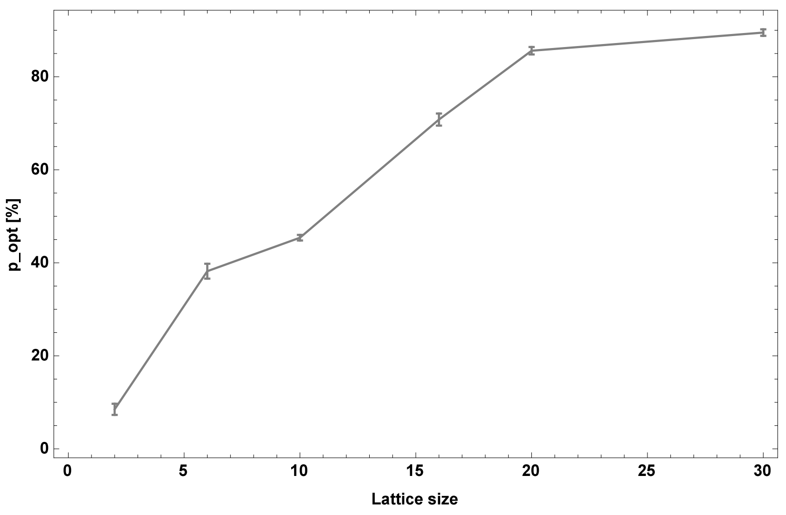


Figure 1: Optimized probability values for the stochastic overrelaxation method depending on the lattice size, including the interpolation between those points.

2.3 Convergence indicating quantities

To check the convergence of the minimizing function with consecutive sweeps, several quantities can be used that are listed below, based on the discussion in [2]:

$$e_1(t) \equiv \mathcal{E}(t-1) - \mathcal{E}(t) \tag{25}$$

$$e_2(t) \equiv \frac{a^{d+4} g_0^2}{V} \sum_{\mathbf{x}} \sum_{j=1}^{3} [(\nabla * A)(\mathbf{x})]_j^2 \tag{26}$$

$$e_3(t) \equiv \frac{a^d}{V} \sum_{\mathbf{x}} \frac{1}{2} Tr\{[\mathbb{1} - R^{(update)}(\mathbf{x})][\mathbb{1} - R^{(update)}(\mathbf{x})]^\dagger\} \tag{27}$$

$$e_4(t) \equiv \max_{\mathbf{x}} [1 - \frac{1}{2} Tr\, R^{(update)}(\mathbf{x})] \tag{28}$$

$$e_5(t) \equiv 1 - \frac{a^d}{2V} \sum_{\mathbf{x}} Tr\, R^{(update)}(\mathbf{x}) \tag{29}$$

$$e_6(t) \equiv \frac{1}{d} \sum_{\nu=1}^{d} \frac{1}{3N_\nu} \sum_{j=1}^{3} \sum_{x_\nu=1}^{N_\nu} \frac{[Q_\nu(x\nu) - \hat{Q_\nu}]_j^2}{[\hat{Q_\nu}]_j^2} \tag{30}$$

Where

$$\hat{Q_\nu} \equiv \frac{1}{N_\nu} \sum_{x_\nu=1}^{N_\nu} Q_\nu(x_\nu)\,. \tag{31}$$

The parameter t labels the number of update sweeps that have been applied to the lattice, with the right-hand side being evaluated after the t-th sweep is executed, as in [2]. According to [7] the six quantities $e_i(t)$ all converge to zero exponentially with the same rate. This fact will become important in Section 3 in combination with the quantity $e_1(t)$, which represents the negative discrete derivative of the minimization function according to the backward Euler method. The relaxation time τ_i can be introduced ([2,4]) that satisfies the following relation:

$$e_i(t) \approx c_i \exp(-t/\tau_i) \tag{32}$$

with

$$\tau_i \equiv \lim_{t\to\infty} \frac{-1}{\ln[e_i(t+1)/e_i(t)]} \tag{33}$$

3 Stochastic Overrelaxation with Selective Probability Manipulation

The purpose of this work is to investigate, if deeper minima can be reached with the stochastic overrelaxation method in combination with the multi-start algorithm by selectively manipulating the probability p used to calculate the gauge transformations as in (20). The idea is to define an indicator/prediction parameter Π that determines, whether the minimization function $\mathcal{E}(\{g\})$ approaches a local minimum, or not. Based on this indicating quantity the decision is taken to increase p or keep it constant for the calculations of the following sweep, to prevent the minimization function from converging to the local minimum. This is based on the considerations in Section 2.2.2 that by increasing p the likelihood to move a larger distance in configuration space, and thus to "escape" the local minimum, is raised. On the other hand, when descending to the actual deepest minimum, the value of p can be decreased. This reduces the likelihood for a wider move in configuration space and hence ensures that the function does not jump to another local minimum in the vicinity (in configuration space). The step size Δp with which p is manipulated is selected to be 1% and the SOM combined with the selective probability manipulation is from now on referred to as SPM.

3.1 Implementation of Test Algorithm

Before addressing the possible ways of introducing the prediction parameter Π, a few remarks regarding the algorithms used for the calculations.

3.1.1 Implementation of Random SU(2) Matrices

All sets of matrices used in the lattices, as well as the initial set of gauge matrices, are represented by random $SU(2)$ matrices. The general representation of an arbitrary $SU(2)$ matrix is given by (34).

$$SU(2) = \left\{ \begin{bmatrix} a & b \\ -b^* & a^* \end{bmatrix} \in \mathbb{C}^{2x2} \;\middle|\; |a|^2 + |b|^2 = 1 \right\} \tag{34}$$

One method of generating a random $SU(2)$ matrix from the criteria in (34) is choosing the parameters a and b according to a respective distribution, which turns out to be of the following nature (see [6]):

$$a := e^{i\psi} \cos\phi \quad and \quad b := e^{i\chi} \sin\phi, \tag{35}$$

where $0 \leq \phi \leq \frac{\pi}{2}$, $0 \leq \psi, \chi < 2\pi$, and ϕ is parametrized as in (36). It follows that by sampling $\psi, \chi \in [0, 2\pi]$ and $\xi \in [0, 1]$ from a uniform distribution over the respective intervals at random the required random $SU(2)$ matrix is generated.

$$\phi = \arcsin\sqrt{\xi} \tag{36}$$

The final equation generating random $SU(2)$ matrices $M(\psi, \chi, \xi)$ that was implemented into the test algorithm is represented by (37).

$$M(\psi, \chi, \xi) = \begin{bmatrix} e^{i\psi}\sqrt{1-\xi} & e^{i\chi}\xi \\ -e^{-i\chi}\xi & e^{-i\psi}\sqrt{1-\xi} \end{bmatrix} \tag{37}$$

3.1.2 Lattice Update Algorithm

The gauge fixing of the lattice is implemented into the algorithm in a way that always packages of 4 lattice points are gauge transformed, i.e. the matrices U_x and U_y are transformed. The 4 lattice points are assembled in a 2x2 matrix, where every element includes an U_x and an U_y matrix. At first the two elements on the counter-diagonal of the 2x2 matrix are updated, followed by the two diagonal elements, and after this the algorithm continues with the next package of 4 lattice points. Since the updating algorithm regroups the lattice points in a new superimposed lattice, where every lattice point represents a 2x2 matrix containing the elements of the original lattice, the very same must always have an even number of lattice points in the x- and y-direction, to be compatible with the algorithm. This lattice property is a necessary precondition for the implemented algorithms, as the SOM as well as the SPM rely on it. The respective code for the lattice updating can be found in the Appendix, Section B.8.

3.2 Prediction Parameter

There are several ideas of how the prediction parameter could be introduced, the relation that will be utilized in this work, is that there is a correlation between the alternating value of τ_1 as a function of sweeps t, as defined by equation (33), and the deepness of the minimum of $\mathcal{E}(\{g\})$ that is reached. The approach to the definition of Π that was selected for this work, will be discussed in Subsection 3.2.1, but prior to that a few additional definitions shall be introduced.

As mentioned in Section 2.3, the quantity $e_1(t)$ represents the negative first derivative of the minimization function with regard to the number of sweeps. For the following considerations the quantity $e_1^-(t)$, which equals the actual discrete derivative of $\mathcal{E}(\{g\})$, shall be used.

$$e_1^-(t) \equiv -e_1(t) = \mathcal{E}(t) - \mathcal{E}(t-1) \tag{38}$$

Assuming the relationship in (32) to be true, $e_1^+(t)$ also has an exponential form and by integrating it, it follows that $\mathcal{E}(\{g\})$ itself follows an exponential proportionality:

$$e_1^-(t) \approx -c_1 \exp(-t/\tau_1) = c_1' \exp(-t/\tau_1) \tag{39}$$

$$\mathcal{E}(t) = -\tau_1 c_1' \exp(-t/\tau_1) = c_{\mathcal{E}} \exp(-t/\tau_1) \tag{40}$$

Since the number of sweeps applied to the lattice is finite in every algorithm, τ_1 is not likely to be a constant, as suggested by (33). The algorithm used for this work evaluates τ_1 after every sweep (excluding the first one) and does not treat it as a constant. To evaluate τ_1 the relationship in (41) is utilized that is derived from (40).

$$\tau_1(t) \equiv \frac{-1}{\ln(\mathcal{E}(t)/\mathcal{E}(t-1))} \tag{41}$$

Figure 2 below shows the change in τ_1 as a function of the number of executed sweeps, with a logarithmic scale on the ordinate.

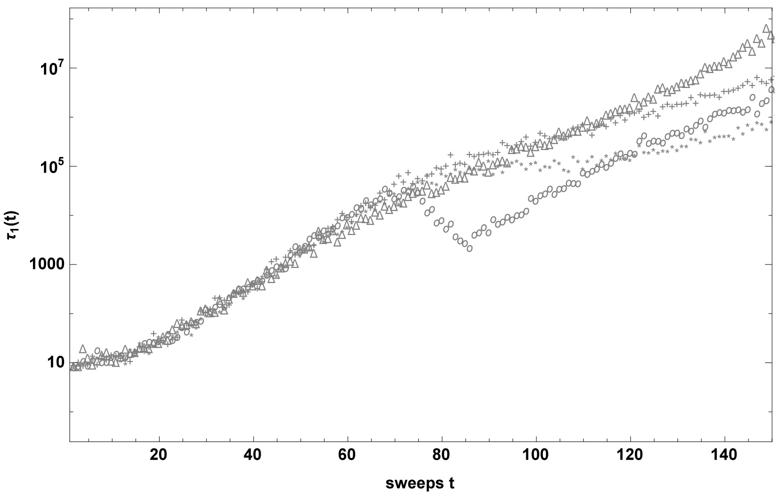


Figure 2: $\tau_1(t)$ calculated according to equation (41) for a lattice with the size $20x20$, with four random restarts.

Figures 3 and 4 depict the functions $e_1(t)$ and $\mathcal{E}(t)$ that go with it. For $e_1(t)$ again a logarithmic scale is used on the ordinate.

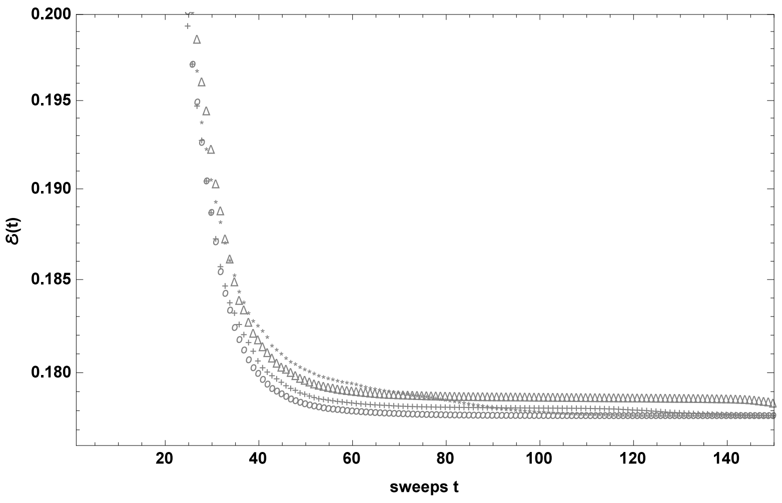


Figure 3: $\mathcal{E}(t)$ calculated according to equation (8) for a lattice with the size $20x20$, with four random restarts.

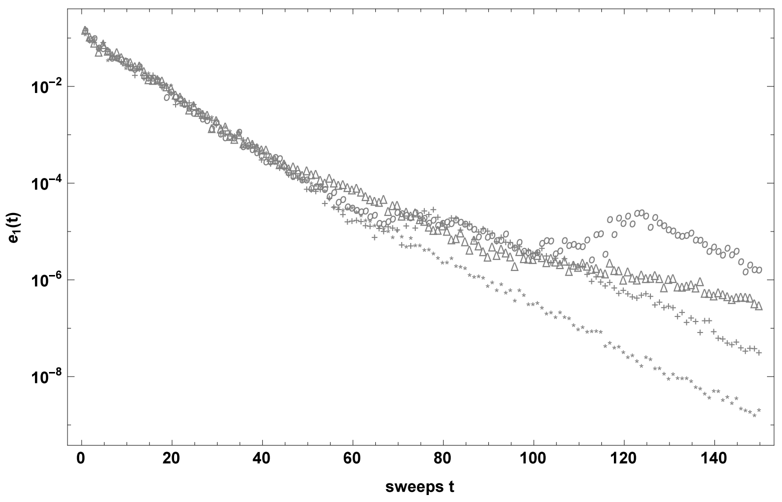


Figure 4: $e_1(t)$ calculated according to equation (25) for a lattice with the size $20x20$, with four random restarts.

As can be seen in Figure 2, the quantity $\tau_1(t)$ obeys an almost linear proportionality with regard to the number of sweeps. Considering that a logarithmic scale was used for the ordinate, this means that $\tau_1(t)$ features an exponential growth during most of the calculated interval. In the following discussion the quantity $\tilde{\tau}(t)$ will be used instead of $\tau(t)$, since the exponential growth of the quantity is easier to distinguish in a logarithmic description.

$$\tilde{\tau}(t) \equiv \ln(\tau(t)) \tag{42}$$

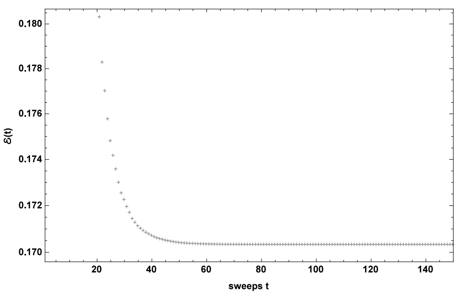
3.2.1 Approach to Defining the Prediction Parameter

To define the prediction parameter Π the average ascent rates $\{\partial_t \tilde{\tau}(t)\}_n$, after a certain number of sweeps n in the beginning, is evaluated. This parameter is then used to force $\tilde{\tau}(t)$ to maintain that ascent rate, by manipulating p in a way that the actual ascent rate $\partial_t\tilde{\tau}(t)$ is increased, when it is below $\{\partial_t\tilde{\tau}(t)\}_n$, and decreased, when it is above $\{\partial_t\tilde{\tau}(t)\}_n$. A flowchart illustrating this workflow can be found in Figure 11 in the Appendix, Section A.1.

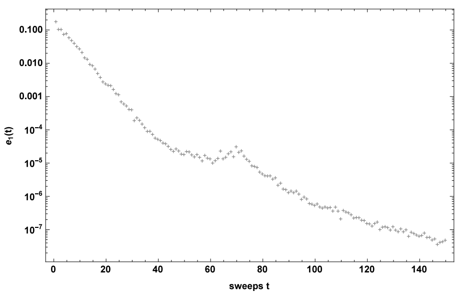
This approach is based on the relation between $\tau_1(t)$ or rather $\tilde{\tau}(t)$ and $\mathcal{E}(t)$ that can be observed, when comparing the three figures 5a, 5b and 5c below. In case $\mathcal{E}(t)$ approaches a local minimum, the "derivative" (represented by $e_1(t)$) converges to a constant value (in this case $\mathcal{E}(t)$ gets out of the local minimum and decreases further). At the same time, $\tilde{\tau}(t)$ does not increase linearly anymore, but the curve bends over and decreases compared to to its ideal linear progression. After overcoming the local minimum, the ascent rate $\partial_t\tilde{\tau}(t)$ increases again and resumes its original value before the local minimum was reached. It should be mentioned that in this context $\partial_t\tilde{\tau}(t)$ is assumed to be the derivative of a continuous function that represents the data points of the calculated discontinuous $\tilde{\tau}(t)$ function, since the actual numeric derivative of these data points fluctuates, see discussion at the end of this section. Thus, by preventing $\partial_t\tilde{\tau}(t)$ from decreasing, compared to the ideal linear progression, it should be possible to bridge the local minimum and descent to a deeper minimum in $\mathcal{E}(t)$ straight away. From now on, $\{\partial_t\tilde{\tau}(t)\}_n$ shall be used for the prediction parameter. However, it is still necessary to determine which number of sweeps "n" is required in the beginning to get a good indicator value $\{\partial_t\tilde{\tau}(t)\}_n$.

As can be seen in Figure 5d $\{\partial_t\tilde{\tau}(t)\}_n$ comes differently close to the actual ascent rate of $\tilde{\tau}(t)$, depending on how large the interval $[0, n]$, over which $\{\partial_t\tilde{\tau}(t)\}_n$ is determined, is chosen. It is evident form Figure 5d that $\{\partial_t\tilde{\tau}(t)\}_{30}$ is closer to the actual $\partial_t\tilde{\tau}(t)$ before and after the minimum than for instance $\{\partial_t\tilde{\tau}(t)\}_{10}$, however, if the interval is chosen too large, the likelihood of a local minimum occurring within it increases, which would in turn decrease the efficiency of the algorithm. Thus, for the test algorithm the default parameter $n = 10$ is used and the optimization of the same will not be part of the following discussion.

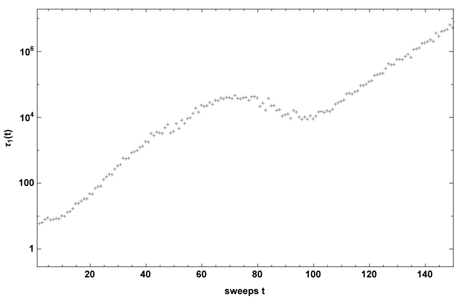
Furthermore, the numeric derivative $\partial_t\tilde{\tau}(t)$ is subject to strong fluctuations with sign changes, as can be seen in Figure 6, therefore it does not make sense to compare every value of $\partial_t\tilde{\tau}(t)$ with the prediction parameter $\{\partial_t\tilde{\tau}(t)\}_{10}$. To generate values that can be compared in a meaningful way, the mean value for $\partial_t\tilde{\tau}(t)$ over an interval of five sweeps $\overline{\partial_t\tilde{\tau}(t)}$ is calculated and the result of the comparison is utilized in the decision making process for increasing/decreasing the probability p. The updated flowchart is depicted in Figure 12 in the Appendix, Section A.1.



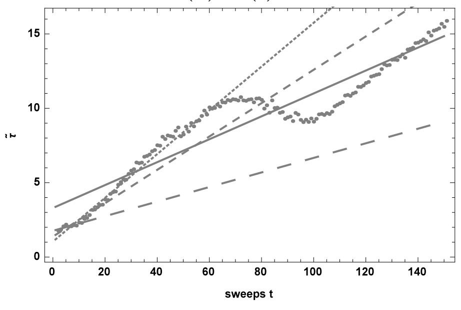
(a) $\mathcal{E}(t)$



(b) $e_1(t)$



(c) $\tau_1(t)$



(d) $\tilde{\tau}(t)$

Figure 5: Exemplary depictions of the quantities $\mathcal{E}(t)$ (Figure 5a), $e_1(t)$ (Figure 5b), $\tau_1(t)$ (Figure 5c) and $\tilde{\tau}(t)$ (Figure 5d) for a 22x22 lattice. Furthermore, Figure 5d includes the linear extrapolation for $\tilde{\tau}(t)$, evaluated with the average ascent rates for the first 10, 20 and 30 sweeps ($\{\partial_t \tilde{\tau}(t)\}_{10} \rightarrow$ large dashes, $\{\partial_t \tilde{\tau}(t)\}_{20} \rightarrow$ medium dashes and $\{\partial_t \tilde{\tau}(t)\}_{30} \rightarrow$ tiny dashes), as well as the linear fit over the whole range (continuous line).

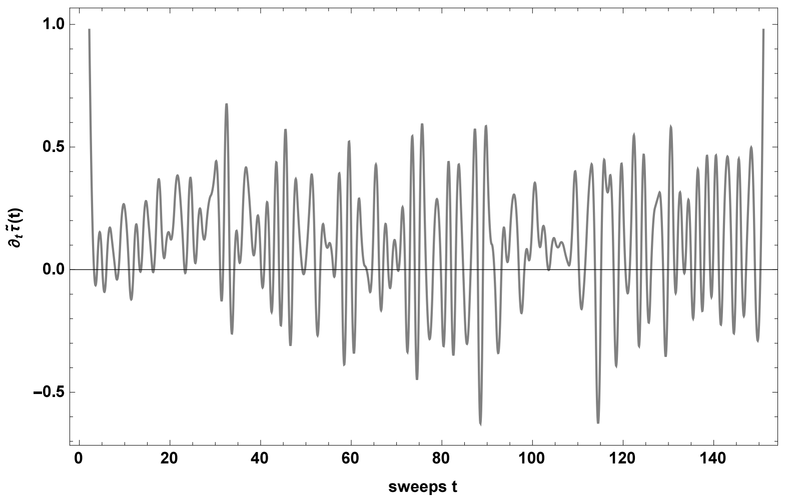


Figure 6: Exemplary depiction of the numerical derivative $\partial_t \tilde{\tau}(t)$ from a $24x24$ lattice.

Another parameter that is expected to have an influence on the performance of the SPM method combined with the multi-start approach, with regard to the actual SOM with multi-start approach, is the number of random restarts used. The default number of random restarts for this work is selected to be 5, since a higher value increases the computing time significantly, however, this parameter remains subject to optimization.

4 Results and Interpretation

The algorithm, using the prediction parameter to enhance the standard SOM in combination with the multi-start approach, was implemented according to the decision making workflow depicted in Figure 12, see Appendix Section A.1 and B. As mentioned in the previous chapter, the number of initial sweeps used to determine the prediction parameter is set to be 10 and the ascent rates that are compared to this value are averaged over five sweeps to dampen the influence of individual fluctuations in the numeric derivative. All calculations use five random restarts, instead of a higher number, to save computing time, however, it is assumed that this parameter can be optimized. For every lattice configuration the minimum of $\mathcal{E}(t)$ was calculated according to the standard SOM method as well as the SPM method with a variable probability p, applying five random restarts in both cases. The resulting values for $min\,\mathcal{E}(t)$ are then compared to each other to determine which algorithm reached the deeper minimum. The starting values for the probability in the SPM algorithm as well as the fixed probability in the SOM algorithm are the optimized values that were determined as described in Section 2.2.2, and which are listed together with the respective lattice sizes in Table 3. All together both algorithms were applied to 200 different lattice configurations for every lattice size respectively and evaluated after 150 sweeps. From this data the gain of the SPM algorithm with regard to the standard SOM was calculated as a function of the lattice size. In this context the gain is understood to be the number of successful runs divided by the number of overall runs, where a successful run is defined as a run for which the SPM algorithm reaches a deeper minimum in $\mathcal{E}(t)$ than the standard SOM. The results are depicted in Figure 7.

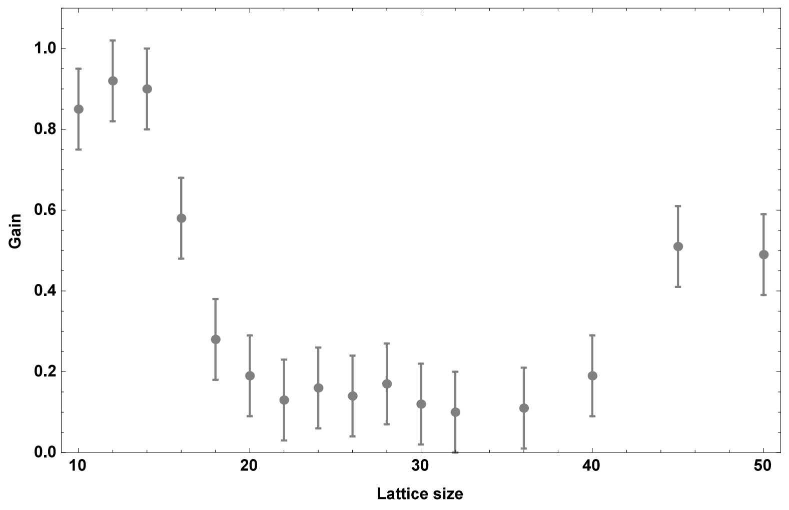


Figure 7: Gain of the SPM algorithm with regard to the standard SOM, for a multi-start approach with 5 random restarts.

Figure 7 shows a very high gain, in the range of 80% to 90%, for the first three lattice sizes, then it drops below 30% for lattice sizes between 18 and 40 and rises again above 40% for the last two considered lattice sizes. This suggests that the determined optimum values for the probability p in Table 3 are not the true optimum values for the SPM algorithm. Based on this information new optimum values for the starting probability were investigated, by calculating for which value of p the gain of the SPM algorithm is maximized. This calculation was done for six different lattice sizes, which are listed in Table 2 together with the resulting optimum values. 100 different lattice configurations were used for each calculation. The optimum probability for all lattice sizes that lie between the six in Table 2 is determined by interpolation and the resulting values are depicted in Figure 8.

Table 2: Optimum values for the probability p with regard to the SPM algorithm

Lattice size	New optimum value for p [%]	Error [%]
10	47	0.5
20	71	0.5
30	79	0.5
40	81	0.5
45	95	1
50	97	1

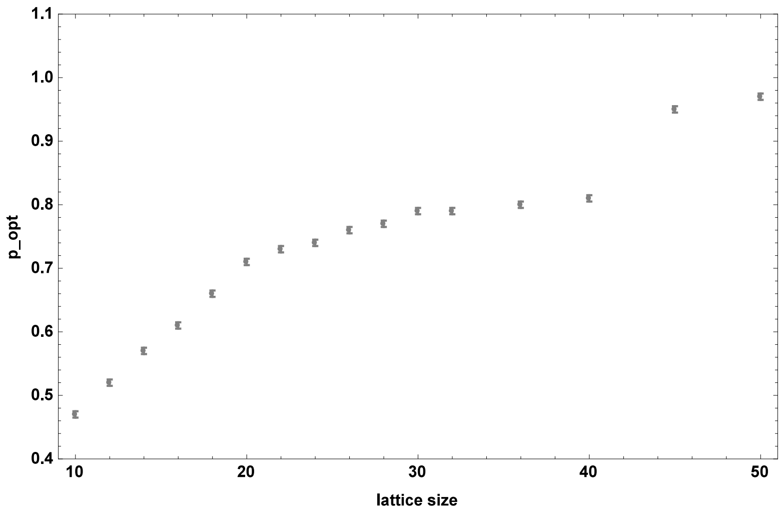


Figure 8: Optimized values for the probability p with regard to the SPM algorithm

Based on the new optimum values for the probability p the calculations for the gain were conducted once more with the same set of parameters and the results are displayed in Figure 9. The same 200 individual lattice configurations were used for every lattice size as for the first calculation. The gain for the new calculations starts out at values above 80% and stabilizes in the range of 40% to 50% with increasing lattice sizes above 26.

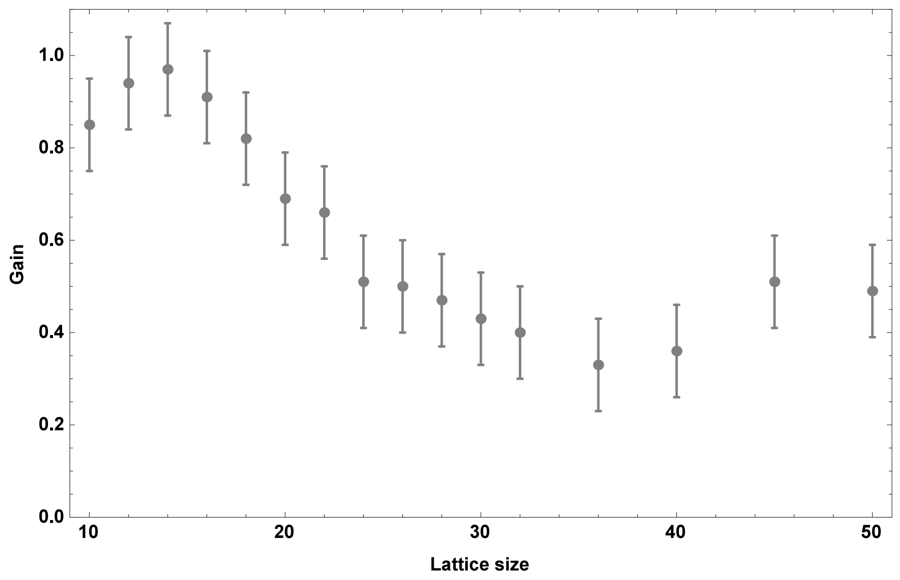


Figure 9: Gain of the SPM algorithm with regard to the standard SOM, with newly determined optimum values for the probability p (see Figure 8), for a multi-start approach with 5 random restarts.

Figure 10 indicates the growth of calculation time with increasing lattice sizes, where the measured time is the time used to calculate a full cycle with 150 sweeps for both methods, the standard SOM and the SPM algorithm.

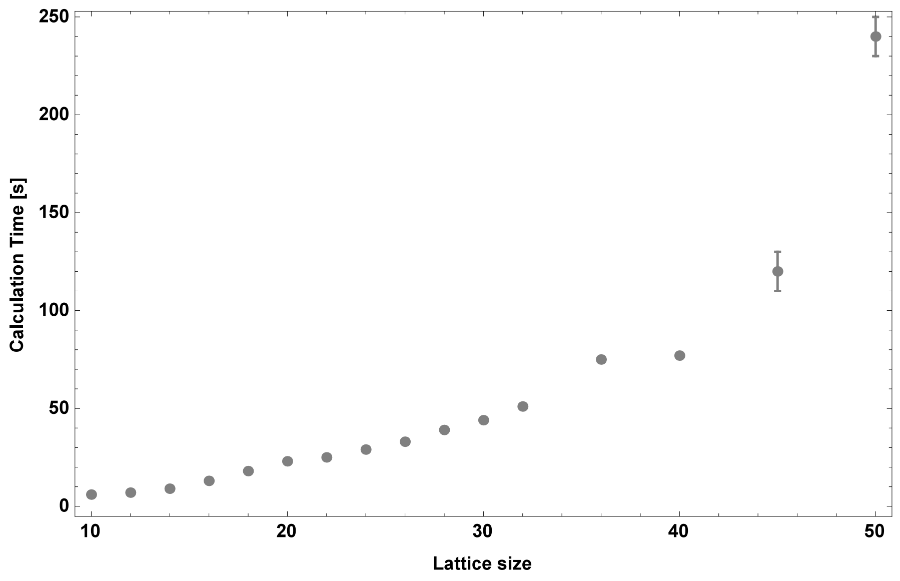


Figure 10: Calculation time as a function of lattice size, with one random restart with 150 sweeps, for the SOM and the SPM method. The errors for the first 14 data points are in the range of $1s$ and can thus not be seen in this representation.

5 Summary and Outlook

The results in Figure 9 show that for small lattice sizes below $L = 22$ the altered stochastic overrelaxation method that uses a selective probability manipulation process features indeed a gain higher than 50%, with regard to the classical stochastic overrelaxation method (SOM), when applied with five random restarts. For lattice sizes above $L = 22$ the gain is below 50% and seems to stabilize at a value around 40%. This means that for small lattice sizes the SPM method poses an advantage over the SOM, however, for increasing lattice sizes this advantage vanishes and transforms into a disadvantage compared to the SOM. The reason for this is that, similar to the probability p for the SOM, there are parameters in this method that need to be optimized for every lattice size, so as to achieve a higher gain. The remaining parameters for the SPM method, which need to be optimized, are

- the number of random restarts, used for the multi-start approach,
- the size of the interval$[0, n]$ over which the prediction parameter $\{\partial_t \tilde{\tau}(t)\}_n$ is calculated,
- the size of the probability increment, over which the probability p is increased/decreased during the process of the SPM method,
- and the size of the interval over which the average for the numeric derivative $\partial_t \tilde{\tau}(t)$ is calculated, to flatten the values out for comparison with the prediction parameter.

By optimizing these parameters it should be possible to increase the gain for larger lattice sizes. Furthermore, the accuracy used for the determination of the optimized probability (p_{opt}), see Sections 2.2.2 and 4, is 1% and thus very low, i.e. by increasing the accuracy of p_{opt} the gain might also be raised. To conclude, it was shown that the changes made in the SPM method, with regard to the SOM, can effectively improve the results by reaching deeper minima, compared to the SOM. However, the optimization of the involved parameters (as listed above) is still required.

References

[1] Ian J. R. Aitchison and Anthony J. G. Hey. *Gauge theories in particle physics, Volume II: QCD and the Electroweak Theory.* CRC Press, 2003.

[2] Attilio Cucchieri and Tereza Mendes. Critical slowing-down in su (2) landau gauge-fixing algorithms, 1996 nucl. *Phys. B*, 471:263.

[3] Christof Gattringer and Christian Lang. *Quantum chromodynamics on the lattice: an introductory presentation*, volume 788. Springer Science & Business Media, 2009.

[4] Arjan Hulsebos. Gribov copies and other gauge fixing beasties on the lattice. *arXiv preprint hep-lat/9211018*, 1992.

[5] Axel Maas. Gauge bosons at zero and finite temperature. *Physics Reports*, 524(4):203–300, 2013.

[6] Maris Ozols. How to generate a random unitary matrix, 2009.

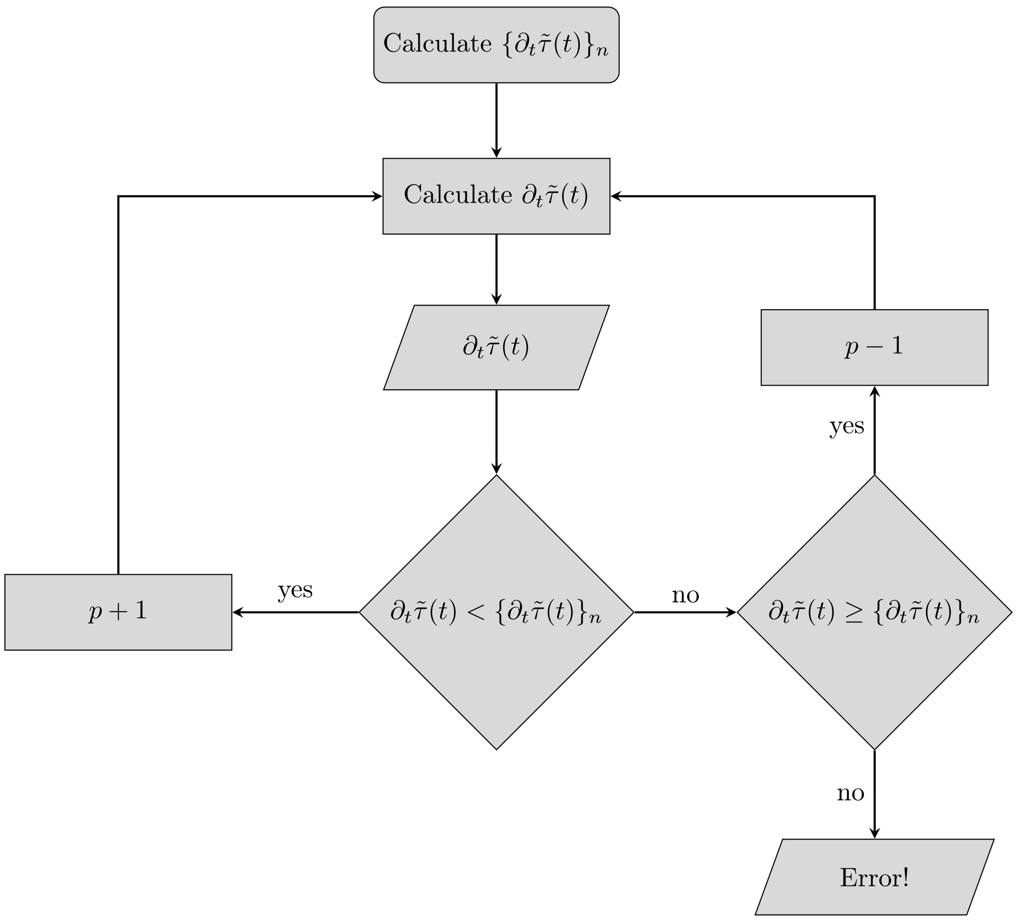
[7] H. Suman and Klaus Schilling. A comparative study of gauge fixing procedures on the connection machines cm2 and cm5. *Parallel Computing*, 20(7):975–990, 1994.

A Appendix 1

Table 3: Selected lattice sizes and optimized probability values for the stochastic overrelaxation method, used in most of the calculations, unless stated otherwise.

Lattice size	**Optimum value for p [%]**	**Number of calculated sweeps**
10x10	45	200
12x12	48	200
14x14	58	200
16x16	70.8	200
18x18	78	150
20x20	85.6	150
22x22	86	150
24x24	88	150
26x26	89	150
28x28	89	150
30x30	89.5	150
32x32	90	150
36x36	92	150
40x40	93	150
45x45	95	150
50x50	97	150

A.1 Indicator Decision Workflow



Calculate $\{\partial_t \tilde{\tau}(t)\}_n$
Calculate $\partial_t \tilde{\tau}(t)$
$\partial_t \tilde{\tau}(t)$
$p - 1$
yes
$p + 1$
yes
$\partial_t \tilde{\tau}(t) < \{\partial_t \tilde{\tau}(t)\}_n$
no
$\partial_t \tilde{\tau}(t) \geq \{\partial_t \tilde{\tau}(t)\}_n$
no
Error!

Figure 11: Flowchart for the implementation of the indicator quantity into the decision making process of increasing or decreasing the probability p.

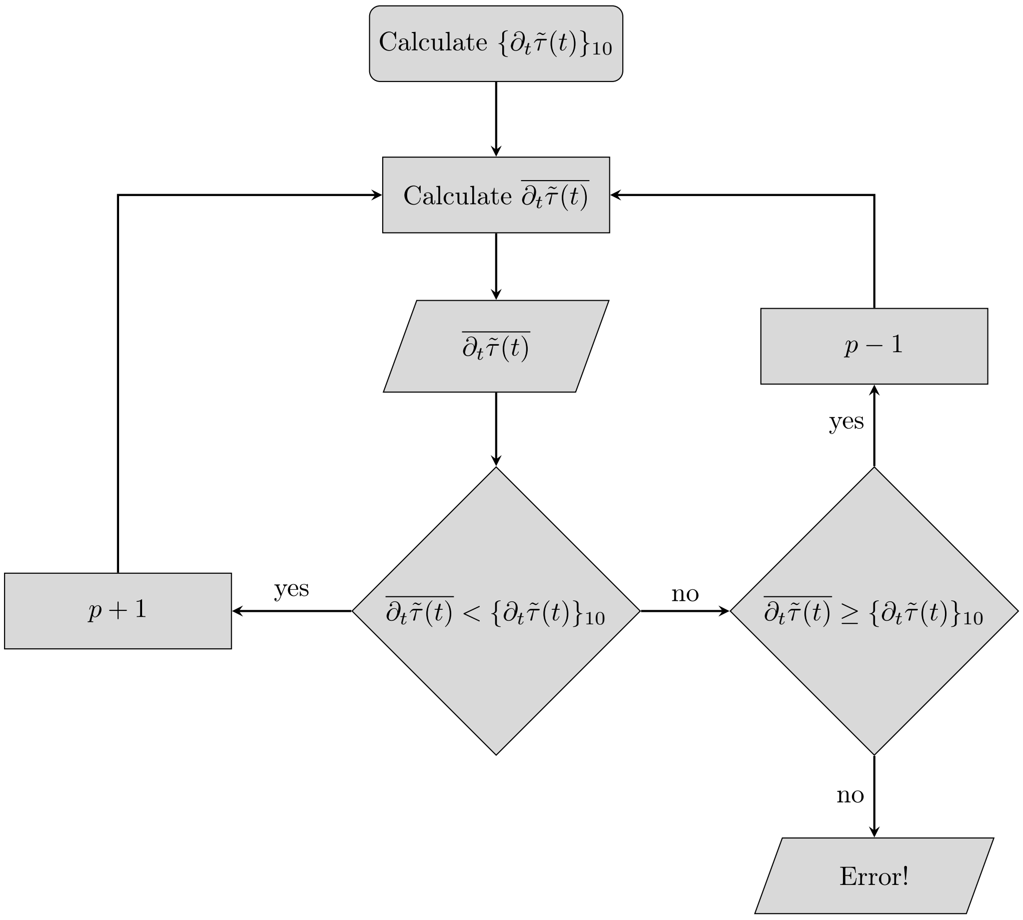

Calculate $\{\partial_t\tilde{\tau}(t)\}_{10}$
Calculate $\overline{\partial_t\tilde{\tau}(t)}$
$\overline{\partial_t\tilde{\tau}(t)}$
$p-1$
yes
$p+1$
yes
$\overline{\partial_t\tilde{\tau}(t)} < \{\partial_t\tilde{\tau}(t)\}_{10}$
no
$\overline{\partial_t\tilde{\tau}(t)} \geq \{\partial_t\tilde{\tau}(t)\}_{10}$
no
Error!

Figure 12: Updated flowchart for the implementation of the indicator quantity into the decision making process of increasing or decreasing the probability p.

A.2 Optimized p-values for the SPM Algorithm

The following Tables 4 to 7 contain the data sets used to estimate the optimum values of the probability p for the SPM algorithm, as discussed in Section 4.

Table 4: Data set from the probability optimization process for the 10x10 lattices.

Value for p [%]	**Gain**
40	83
41	77
42	80
43	80
44	76
45	85
46	79
47	85
48	70

Table 5: Data set from the probability optimization process for the 20x20 lattices.

Value for p [%]	Gain
70	66
71	69
72	62
73	51
74	44
75	44
86	14

Table 6: Data set from the probability optimization process for the 30x30 lattices.

Value for p [%]	Gain
78	40
79	43
80	36
81	16
82	22
83	18
90	12

Table 7: Data set from the probability optimization process for the 40x40 lattices.

Value for p [%]	Gain
81	36
82	33
83	29
85	21
86	19
87	15
93	16

B Appendix 2

In this section all packages, functions, parameters and definitions are listed, that are required to use the SPM algorithm. A few remarks in advance:
In the following the two integers x and y will always represent the two indices of a matrix, this matrix may be a mere $2x2$ matrix, but it can also represent the two-dimensional spacetime lattice, for which every entry consists of two $2x2$ matrices. Moreover, the integer mu is used to distinguish between the two matrices U_x and U_y at every lattice point, where $mu = 1$ gives the U_x matrix and $mu = 2$ gives the U_y matrix.

B.1 Used Packages

Below, all existing C++ packages that were used for the algorithm are listed:

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cmath>
#include <random>
#include <chrono>
#include <complex>
```

```
#include <algorithm>
#include <cstring>
```

B.2 Parameters and Structs Package

This section defines all parameters used for the calculations, for further discussion of those parameters, see Section 2 or the following subsections. Furthermore, four different structs are implemented that are used to deal with matrices and lattice points of a two-dimensional spacetime lattice. The default value for the lattice size is 10 at the moment and for the initial probability it is 47%

```
using namespace std;

// Defines an arbitrary 2x2 matrix with complex elements.
struct M2x2{
    complex<long double> u00;
    complex<long double> u01;
    complex<long double> u10;
    complex<long double> u11;
};

// Defines a lattice point of a two-dimensional spacetime
// lattice, with the matrices U_x and U_y.
struct MatrixVector{
    M2x2 ux;
    M2x2 uy;
};

// Defines a lattice point of the gauge matrices lattice,
// which accompanies the spacetime lattice and contains
// the respective gauge matrices for each point of the
// spacetime lattice. This splitting is intoduced to
// make calculations easier.
struct GaugeMatrices{
    M2x2 g;
};

// Defines a special output variable, that is used to pass
// on integers and pointers to another function.
struct ReturnVal{
    long double script_e;
    long double* ld_pointer1;
    long double* ld_pointer2;
    int* i_pointer;
};

complex<long double> kA(-1,0);  // defines the complex negative unit;
complex<long double> kI(0, 1);  //defines the imaginary unit;
complex<long double> kOne(1, 0); // defines the complex unit element;
complex<long double> kZero(0, 0); // defines zero in complex notation;
complex<long double> kTwo(2, 0); // defines two in complex notation;

int kN_x = 10;
int kN_y = kN_x;
int p_start = 47;                           // same as p_opt for a given
                                            // lattice;
int ID = 1;
int a = 1;                                          // a is the distance between
                                                    // two lattice points;
int d = 2;                                          // d is the lattice dimension;
int kLx = a*kN_x;
int kLy = a*kN_y;
int V = kLx*kLy;                                    // V is the total number of
                                                    // lattice points
int g0 = 1;                                         // g0 is a coupling constant;
```

B.3 Matrix Package

The matrix package includes all functions that are required to do the following matrix operations for arbitrary $2x2$ matrices:

- Calculate the trace of a matrix;
- Calculate the determinant of a matrix;
- Calculate the product of two matrices;
- Calculate the multiplication of a scalar with a matrix;
- Calculate the adjunct matrix;
- Calculate the sum of two matrices;

All scalars in this context are treated as complex long double numbers, which are defined as follows

```
complex<long double> k(-1,0),
```

where the first number is the real value and the second number represents the complex value. Furthermore, this package includes two functions that can be applied to the points of the used two dimensional lattice. The first function gives out either the U_x or the U_y matrix of an arbitrary lattice point and the second function can be used to calculate the gauge transformed matrix of an arbitrary matrix in the lattice. Listed below are the functions of this package, as implemented into the algorithm:

```
// This function calculates the trace of an arbitrary 2x2 matrix.
complex<long double> Trace(M2x2 matrix){
    complex<long double> trace;
    trace = matrix.u00 + matrix.u11;
    return trace;
}

// This function calculates the determinant of an arbitrary 2x2 matrix.
complex<long double> Determinant(M2x2 matrix){
    complex<long double> determinant;
    determinant = (matrix.u00*matrix.u11) - (matrix.u01*matrix.u10);
    return determinant;
}

// This function calculates the product of of arbitrary 2x2 matrices.
M2x2 MatrixProduct(M2x2 matrix1, M2x2 matrix2){
    M2x2 matrix_product;

    matrix_product.u00 = matrix1.u00*matrix2.u00 + matrix1.u01*matrix2.u10;
    matrix_product.u01 = matrix1.u00*matrix2.u01 + matrix1.u01*matrix2.u11;
    matrix_product.u10 = matrix1.u10*matrix2.u00 + matrix1.u11*matrix2.u10;
    matrix_product.u11 = matrix1.u10*matrix2.u01 + matrix1.u11*matrix2.u11;
    return matrix_product;
}

// This function calculates the scalar product of an arbitrary
// 2x2 matrix with a complex scalar.
M2x2 ScalarMultiplication(complex<long double> a, M2x2 matrix){
    M2x2 matrix_result;

    matrix_result.u00 = a*matrix.u00;
    matrix_result.u01 = a*matrix.u01;
    matrix_result.u10 = a*matrix.u10;
    matrix_result.u11 = a*matrix.u11;

    return matrix_result;
}

// This function calculates the adjoint matrix of an arbitrary 2x2 matrix.
M2x2 MatrixAdjungation(M2x2 matrix){
    M2x2 adjunct_matrix;

    adjunct_matrix.u00 = conj(matrix.u00);
    adjunct_matrix.u01 = conj(matrix.u10);
    adjunct_matrix.u10 = conj(matrix.u01);
    adjunct_matrix.u11 = conj(matrix.u11);

    return adjunct_matrix;
}

// This function calculates the sum of two arbitrary 2x2 matrices.
M2x2 MatrixAddition(M2x2 matrix1, M2x2 matrix2){
    M2x2 matrix_sum;
    matrix_sum.u00 = matrix1.u00 + matrix2.u00;
    matrix_sum.u01 = matrix1.u01 + matrix2.u01;
    matrix_sum.u10 = matrix1.u10 + matrix2.u10;
    matrix_sum.u11 = matrix1.u11 + matrix2.u11;

    return matrix_sum;
}

// Takes an arbitrary lattice point and gives out either the U_x or the
// U_y matrix of that point.
M2x2 UGauge_mu(MatrixVector** u_gauge, int x, int y, int mu){
    M2x2 aux_matrix;
    switch (mu){
        case 1:
            aux_matrix = u_gauge[x][y].ux;
            break;
        case 2:
            aux_matrix = u_gauge[x][y].uy;
            break;
    }
    return aux_matrix;
}

// Calculates the gauge transformed matrix for an arbitrary lattice
// matrix U_mu.
M2x2 ConstructU_g(MatrixVector** spl, GaugeMatrices** gm, int x,
int y, int mu){
    M2x2 result;

    if (mu == 1){
        if (x == kLx-1){
            result = MatrixProduct(MatrixProduct(gm[x][y].g,
                                                   spl[x][y].ux),
                                     MatrixAdjungation(gm[0][y].g));
        } else
            result = MatrixProduct(MatrixProduct(gm[x][y].g,
                                                   spl[x][y].ux),
                                     MatrixAdjungation(gm[x+1][y].g));
    }

    if (mu == 2){
        if (y == kLy-1){
            result = MatrixProduct(MatrixProduct(gm[x][y].g,
                                                   spl[x][y].uy),
                                     MatrixAdjungation(gm[x][0].g));
        } else
            result = MatrixProduct(MatrixProduct(gm[x][y].g,
                                                   spl[x][y].uy),
                                     MatrixAdjungation(gm[x][y+1].g));
    }

    return result;
}
```

B.4 Lattice Package

For this section, it is important to mention at this point that all calculations use two different two-dimensional lattices, one that contains the lattice entries (often denoted as STL, for spacetime lattice), i.e. the matrices U_x and U_y, and a second one that contains the respective gauge matrices for every point (often denoted as GML, for gauge matrix lattice). In the following the functions are listed, which can be used to read in an arbitrary STL or GML lattice, as well as to give out a GML lattice with random entries as an initial condition.

```
// Creates a pointer to a two dimensional lattice, with the
// dimensions kLx x kLy, with matrix vectors as entries.
// The lattice is not yet filled.
MatrixVector** CreateLatticePointer(){
    MatrixVector** pointer = new MatrixVector*[kLx];
    for (int i = 0; i < kLy; i++) {
        pointer[i] = new MatrixVector[kLy];
    }

    return pointer;
}

// Creates a pointer to a two dimensional lattice, with the
// dimensions kLx x kLy, containing the gauge matrices (GM),
// one for every lattice point.
GaugeMatrices** CreateGMVectorPointer(){
    GaugeMatrices** pointer = new GaugeMatrices*[kLx];
    for (int i = 0; i < kLy; i++){
        pointer[i] = new GaugeMatrices[kLy];
    }

    return pointer;
}

// Reads in a 2x2 matrix that is saved as a text file with the
// following formatting: {{1,1},{1,1}}. The string "fname"
// denotes the file directory, e.g. "C:\\Users\\User\\gml.txt".
M2x2 ReadInMatrix(ifstream& fileIn){
    string substring;
    M2x2 matrix;
    //ifstream fileIn(fname.c_str());
    fileIn.ignore(2);

    getline(fileIn,substring,';');
    complex<long double> c1;
    istringstream is1(substring);
    is1 >> c1;

    getline(fileIn,substring,'}');
    complex<long double> c2;
    istringstream is2(substring);
    is2 >> c2;

    fileIn.ignore(2);

    getline(fileIn,substring,';');
    complex<long double> c3;
    istringstream is3(substring);
    is3 >> c3;

    getline(fileIn,substring,'}');
    complex<long double> c4;
    istringstream is4(substring);
    is4 >> c4;

    fileIn.ignore(1);

    matrix.u00 = c1;
    matrix.u01 = c2;
    matrix.u10 = c3;
    matrix.u11 = c4;

    return matrix;
}

// Reads in a MatrixVector, as defined above, that is saved in a text file with
// the following formatting: {{{1,1},{1,1}},{{1,1},{1,1}}}. The string "fname"
// denotes the file directory, e.g. "C:\\Users\\User\\gml.txt".
MatrixVector ReadInMatrixVector(ifstream& fileIn){
    string substring;
    MatrixVector mv;

    fileIn.ignore(1);
    mv.ux = ReadInMatrix(fileIn);
    fileIn.ignore(1);
    mv.uy = ReadInMatrix(fileIn);
    fileIn.ignore(1);

    return mv;
}

// Reads in a two-dimensional array with entries of type MatrixVector, as
// defined above, from a text file. The string "fname"
// denotes the file directory, e.g. "C:\\Users\\User\\stl.txt".
void ReadInSTL(MatrixVector** spl, string fname){
    ifstream fileIn(fname.c_str());
    // read in one row from the Space Time Lattice in the x-Direction //
    // ignore the curly bracket that marks the beginning of the lattice
    fileIn.ignore(1);
    int x = 0;
    while (x < kN_x){
        // ignore curly bracket that marks the beginning of a row
        fileIn.ignore(1);
        int y = 0;
        while (y < kN_y){
            spl[x][y] = ReadInMatrixVector(fileIn);
            // ignore the semicolon that marks the ending of one and the
beginning of the next Gauge Matrix or
            // ignore curly bracket that marks the ending of a row
            fileIn.ignore(1);

            y++;
        }
        // ignore the white space at the end of the line
        fileIn.ignore(1);
        x++;
    }
    // ignore the curly bracket that marks the end of the lattice
    fileIn.ignore(1);
    fileIn.close();
}

// Reads in a two-dimensional array with entries of type GaugeMatrices,
// as defined above, from a text file. The string "fname"
// denotes the file directory, e.g. "C:\\Users\\User\\gml.txt".
void ReadInGML(GaugeMatrices** gm, string fname){
    ifstream fileIn(fname.c_str());
    // read in one row from the Space Time Lattice in the x-Direction //
    // ignore the curly bracket that marks the beginning of the lattice
    fileIn.ignore(1);
    int x = 0;
    while (x < kN_x){
        // ignore curly bracket that marks the beginning of a row
        fileIn.ignore(1);
        int y = 0;
        while (y < kN_y){
            gm[x][y].g = ReadInMatrix(fileIn);
            // ignore the semicolon that marks the ending of one and the
// beginning of the next Gauge Matrix or
            // ignore curly bracket that marks the ending of a row
            fileIn.ignore(1);

            y++;
        }
        // ignore the white space at the end of the line
        fileIn.ignore(1);
        x++;
    }
    // ignore the curly bracket that marks the end of the lattice
    fileIn.ignore(1);
    fileIn.close();
}

//reads out an uniformly distributed number
long double UniformDist(long double range_from, long double range_to)
{
    long seed = std::chrono::system_clock::now().time_since_epoch().count();
    std::default_random_engine generator (seed);
    std::uniform_real_distribution<long double>
    distribution (range_from,range_to);

    return distribution(generator);
}

//multiplies an uniformly distributed number with 2pi
//This relates to the variables alpha, psi, chi or xi in "equation 1"
long double RandomRealExponents()
{
    long double alpha = 2*M_PI*UniformDist(0, 1);
    return alpha;
}

// Fills a 2x2 matrix with random entries;
M2x2 FillU(complex<long double> negative_one,
complex<long double> imaginary_unit)
{
    complex<long double> psi(RandomRealExponents(),0);
    complex<long double> chi(RandomRealExponents(),0);
    long double xi = UniformDist(0, 1);
    complex<long double> phi(sqrt(xi),0);
    phi = asin(phi);

    M2x2 random_matrix;
    M2x2 result;

    random_matrix.u00 = exp(imaginary_unit*psi)*cos(phi);
    random_matrix.u01 = exp(imaginary_unit*chi)*sin(phi);
    random_matrix.u10 =
    negative_one*exp(negative_one*imaginary_unit*chi)*sin(phi);
    random_matrix.u11 = exp(negative_one*imaginary_unit*psi)*cos(phi);

    result = ScalarMultiplication(kOne/sqrt(Determinant(random_matrix)),
    random_matrix);
    return result;
}

// Assigns a random matrix as gauge matrix in the initial lattice system;
GaugeMatrices InitializeGaugeMatrices(){
    M2x2 g;
    GaugeMatrices gauge_matrix_vector;

    g = FillU(kA,kI);
    gauge_matrix_vector.g = g;

    return gauge_matrix_vector;
}

// Fills a GML with random matrices as initial condition;
GaugeMatrices** FillInitialGML(){
    GaugeMatrices** p_gm_aux = CreateGMVectorPointer();
    int x3 = 0;
    while (x3<kN_x){
        int y3 = 0;
        while (y3<kN_y){
            p_gm_aux[x3][y3] = InitializeGaugeMatrices();
            y3++;
        }
        x3++;
    }
    return p_gm_aux;
}

// Writes a 2x2 matrix into a text file with the following
// formatting: {{1,1},{1,1}}.
void GiveOutMatrix(M2x2 matrix, ofstream& out){
    out << "{{";
    out << matrix.u00 << ";" << matrix.u01;
    out << "};{";
    out << matrix.u10 << ";" << matrix.u11;
    out << "}}";
}

// Writes a gauge matrix into a text file with the following
// formatting: {{1,1},{1,1}}.
void GiveOutGaugeMatrix(GaugeMatrices gauge, ofstream& out){
    GiveOutMatrix(gauge.g,out);
}

// Writes a Gauge Matrices Lattice as a two-dimensional array
// with entries of type GaugeMatrices, as defined above, into
// a text file. The string "fdirectory" denotes the file directory,
// e.g. "C:\\Users\\User\\".
void GiveOutGML(GaugeMatrices** gm, string fdirectory,
int lattice_size, int ID){
    string fname = fdirectory;
    string aux = to_string(lattice_size);
    fname += aux;
    fname += "x";
    fname += aux;
    fname += ".";
    fname += "gml.";
    fname += to_string(ID);
    fname += ".txt";
    ofstream out(fname.c_str());
    out << "{";

    int x = 0;
    while (x < kN_x){
        out << "{";
        int y = 0;
        while (y < kN_y){
            GiveOutGaugeMatrix(gm[x][y],out);
            if (y == kN_y-1)
                out << "}";
            else
                out << ";";
            if(x == kN_x-1 && y == kN_y-1)
                out << "}";

            y++;
        }

        out << endl;
        x++;
    }

    out.close();
}
```

B.5 Minimizing Function Package

This package provides the code that is used to calculate the minimizing function $\mathcal{E}(t)$ according to (8).

```
//Calculates the minimizing function.
long double MinimizingFunctionE(MatrixVector** u_gauge){
    long double x_sum = 0;
    long double y_sum = 0;
    long double sum;
    long double result;

    // Inner sum over x
    int x_mu_1 = 0;
    while (x_mu_1 < kLx){
        int y_mu_1 = 0;
        while (y_mu_1 < kLy){
            x_sum += real(Trace(u_gauge[x_mu_1][y_mu_1].ux));

            y_mu_1++;
        }
        x_mu_1++;
    }

    // Inner sum over y
    int x_mu_2 = 0;
    while (x_mu_2 < kLx){
        int y_mu_2 = 0;
        while (y_mu_2 < kLy){
            y_sum += real(Trace(u_gauge[x_mu_2][y_mu_2].uy));

            y_mu_2++;
        }
        x_mu_2++;
    }

    sum = x_sum + y_sum;
    long double prefactor = pow(a, d)/(2 * d * V);
    result = 1 - prefactor*sum;

    return result;
}
```

B.6 LAM Package

The following code provides the functions that are used to calculate $\tilde{h}$, $\tilde{\omega}$ and $\mathcal{T}$, according to equations (11), (12), (10), (13) and (15). These quantities are then used to calculate the gauge matrices for the Los Alamos Method (LAM), and for the SOM and SPM method as well, which are based on the LAM.

```
// Calculates the singlesite magnetic field h_tilde for a given
// lattice point.
M2x2 ConstructHTilde(MatrixVector** spl, GaugeMatrices** gm, int x,
int y, bool Tilde){
    M2x2 h_tilde;
    M2x2 h;
    M2x2 result;
    M2x2 sum_mu_1;
    M2x2 sum_mu_2;

    // Inner sum over x -> mu = 1
    int i = 0;
    if (x == kLx-1){
        i = 1;
    }
    if (x == 0){
        i = 2;
    }

    switch (i) {
        case 1:
            sum_mu_1 = MatrixAddition(MatrixProduct(spl[x][y].ux,
                                  MatrixAdjungation(gm[0][y].g)),
                       MatrixProduct(MatrixAdjungation(spl[x - 1][y].ux),
                                  MatrixAdjungation(gm[x - 1][y].g)));
            break;
        case 2:
            sum_mu_1 = MatrixAddition(MatrixProduct(spl[x][y].ux,
                                  MatrixAdjungation(gm[x + 1][y].g)),
                       MatrixProduct(MatrixAdjungation(spl[kLx - 1][y].ux),
                                  MatrixAdjungation(gm[kLx - 1][y].g)));
            break;
        default:
            sum_mu_1 = MatrixAddition(MatrixProduct(spl[x][y].ux,
                                      MatrixAdjungation(gm[x + 1][y].g)),
                       MatrixProduct(MatrixAdjungation(spl[x - 1][y].ux),
                                      MatrixAdjungation(gm[x - 1][y].g)));
            break;
    }

    // Inner sum over y -> mu = 2
    int j = 0;
    if (y == kLy-1){
        j = 1;
    }
    if (y == 0){
        j = 2;
    }

    switch (j) {
        case 1:
            sum_mu_2 = MatrixAddition(MatrixProduct(spl[x][y].uy,
                                      MatrixAdjungation(gm[x][0].g)),
                       MatrixProduct(MatrixAdjungation(spl[x][y - 1].uy),
                                      MatrixAdjungation(gm[x][y - 1].g)));
            break;
        case 2:
            sum_mu_2 = MatrixAddition(MatrixProduct(spl[x][y].uy,
                                      MatrixAdjungation(gm[x][y + 1].g)),
                       MatrixProduct(MatrixAdjungation(spl[x][kLy - 1].uy),
                                      MatrixAdjungation(gm[x][kLy - 1].g)));
            break;
        default:
            sum_mu_2 = MatrixAddition(MatrixProduct(spl[x][y].uy,
                                      MatrixAdjungation(gm[x][y + 1].g)),
                       MatrixProduct(MatrixAdjungation(spl[x][y - 1].uy),
                                      MatrixAdjungation(gm[x][y - 1].g)));
            break;
    }

    h = MatrixAddition(sum_mu_1,sum_mu_2);
    h_tilde = ScalarMultiplication(kOne/sqrt(Determinant(h)),h);

    if (Tilde){
        result = h_tilde;
    } else result = h;

    return result;
}

// Calculates omega_tilde for a given lattice point;
M2x2 ConstructOmegaTilde(MatrixVector** spl, GaugeMatrices** gm,
int x, int y, bool Tilde){
    M2x2 omega;
    M2x2 omega_tilde;
    M2x2 result;
    omega = MatrixProduct(gm[x][y].g,ConstructHTilde(spl,gm,x,y, false));
    omega_tilde = ScalarMultiplication(kOne/sqrt(Determinant(omega)),omega);

    if (Tilde){
        result = omega_tilde;
    } else result = omega;

    return result;
}

// Calculates script_t for a given lattice point;
complex<long double> ConstructScriptT(MatrixVector** spl,
GaugeMatrices** gm, int x, int y){
    complex<long double> script_t;

    script_t = Trace(ConstructOmegaTilde(spl,gm,x,y, true));

    return script_t;
}
```

B.7 SOM Package

This package contains the algorithms used to calculate the updated gauge matrices according to the SOM, see (20). However, by setting the probability p to zero these functions can also be used to calculate the updated gauge matrices for the LAM.

```
// Implements the probability p that ranges from 1 to 99,
// while p=0 can be entered as well to use the LAM;
bool probability_p(int p){
    bool TrueFalse;
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_int_distribution<int> dist(1,99);

    if (p == 0){
        TrueFalse = false;
    } else
        TrueFalse = dist(mt) <= p;

    return TrueFalse;
}

// this function generates gauge transformations according to the SOM
// that is defined by equation (20), the gauge transformations for the
// Los Alamos Method can also be generated by setting p to zero
M2x2 ConstructGStoc(MatrixVector **spl, GaugeMatrices** gm, int x,
int y, int p){
    M2x2 g_stoc;
    M2x2 h_tilde_adj = MatrixAdjungation(ConstructHTilde(spl,gm,x,y,true));
    bool TrueFalse = probability_p(p);
    if (TrueFalse){
        complex<long double> script_T = ConstructScriptT(spl,gm,x,y);
        g_stoc = MatrixAddition(ScalarMultiplication(script_T,h_tilde_adj),
                                ScalarMultiplication(kA,gm[x][y].g));
    } else
        g_stoc = h_tilde_adj;
    return g_stoc;
}
```

B.8 Lattice Update Package

This package includes the functions that are used to update the lattice.

```
// Calculates the two gauge transformed matrices for a given
// lattice point.
void GaugeTransform(MatrixVector** u_gauge, MatrixVector** spl,
GaugeMatrices** gm){
    int x = 0;
    while (x < kLx){
        int y = 0;
        while (y < kLy){
            u_gauge[x][y].ux = ConstructU_g(spl,gm,x,y,1);
            u_gauge[x][y].uy = ConstructU_g(spl,gm,x,y,2);
            y++;
        }
        x++;
    }
}

// Updates the lattice for every sweep, by gauge transforming the lattice
// points. For the Los Alamos Method use p = 0.
void LatticeUpdate(MatrixVector** spl,MatrixVector** u_gauge,
GaugeMatrices** gm, int p){
    int x = 0;
    while (x < kLx/2){
        int y = 0;
        while (y < kLy/2){
            M2x2 new_g[2][2];

            //update the counterdiagonal of the 2x2 matrix of gauge matrices
            new_g[0][1] = ConstructGStoc(spl,gm,2*x,2*y+1,p);
            new_g[1][0] = ConstructGStoc(spl,gm,2*x+1,2*y,p);
            gm[2*x][2*y+1].g = new_g[0][1];
            gm[2*x+1][2*y].g = new_g[1][0];

            //update the diagonal of the 2x2 matrix of gauge matrices
            new_g[0][0] = ConstructGStoc(spl,gm,2*x,2*y,p);
            new_g[1][1] = ConstructGStoc(spl,gm,2*x+1,2*y+1,p);
            gm[2*x][2*y].g = new_g[0][0];
            gm[2*x+1][2*y+1].g = new_g[1][1];

            y++;
        }
        x++;
    }
    GaugeTransform(u_gauge,spl,gm);
}

// this is initializing an auxiliary array with the gauge matrices (gm)
GaugeMatrices** FillGMVector(GaugeMatrices** gm_aux){
    GaugeMatrices** gm = CreateGMVectorPointer();
    int x2 = 0;
    while (x2<kLx){
        int y2 = 0;
        while (y2<kLy){
            gm[x2][y2] = gm_aux[x2][y2];
            y2++;
        }
        x2++;
    }

    return gm;
}

// this is initializing an auxiliary lattice with the gauge
// transformed matrices
MatrixVector** FillGaugeTransformedLattice(MatrixVector** spl,
GaugeMatrices** gm){
    MatrixVector** u_gauge = CreateLatticePointer();
    GaugeTransform(u_gauge,spl,gm);

    return u_gauge;
}
```

B.9 Prediction Parameter Package

In this subsection the functions, required to calculate the prediction parameter as well as to calculate $\tilde{\tau_1}(t)$ and $\tau_1(t)$ (according to (33) and the flowchart in Figure 12), are introduced.

```
// Calculates the parameter Tau_1;
long double Tau1_t(long double E1_t, long double E1_t_minus_1){
    long double result;
    result = -1/(log(E1_t)-log(E1_t_minus_1));
    return result;
}

// This function is used to calculate the average growth rate of
// Tau1 in a given interval, where the integer interval specifies
// the number of sweeps, over which the average is calculated.
ReturnVal Tau1Array(MatrixVector **spl, MatrixVector **u_gauge,
                    GaugeMatrices **gm, int p_start, int interval){
    long double E_t_minus_1 = MinimizingFunctionE(u_gauge);
    long double E_t;

    long double* tau1_pointer = new long double[interval+1];
    //int* t_pointer = new int[interval+1];
    long double* e_pointer = new long double[interval+2];
    e_pointer[0] = E_t_minus_1;
    //t_pointer[0] = 0;

    ReturnVal result;

    int k = 0;
    while (k < interval+1) {

        LatticeUpdate(spl, u_gauge, gm, p_start);
        E_t = MinimizingFunctionE(u_gauge);
        //t_pointer[k+1] = k+1;
        e_pointer[k+1] = E_t;
        tau1_pointer[k] = Tau1_t(E_t,E_t_minus_1);

        E_t_minus_1 = E_t;
        k++;
    }

    result.ld_pointer1 = tau1_pointer;
    result.ld_pointer2 = e_pointer;
    // result.i_pointer = t_pointer;
    return result;
}

// The default value for the integer interval, that is the size of the interval
// over which we will take the average for the growth rate of Tau1,
// will be ten. That means that we need ten values for Tau1, for which
// 11 values of ScriptE are required.
ReturnVal GetPredicParam(MatrixVector **spl, MatrixVector **u_gauge,
                         GaugeMatrices **gm, int p_start, int interval){
    long double tau1_gr;
    long double tau1_delta_mean;
    ReturnVal tau1_array = Tau1Array(spl, u_gauge, gm, p_start, interval);
    ReturnVal result;

    int k = 0;
    while (k < interval){
        // The negative factor before the Tau1 function is there,
        // because the Tau1_t function is defined for an exponential
        // decrease, but for Tau1 itself we have an exponential growth
        // and we just use the same function;
        tau1_gr = log(tau1_array.ld_pointer1[k+1])-
        log(tau1_array.ld_pointer1[k]);
        tau1_delta_mean += tau1_gr;

        k++;
    }

    result.script_e = tau1_delta_mean/(interval);
    result.ld_pointer1 = tau1_array.ld_pointer1;
    result.ld_pointer2 = tau1_array.ld_pointer2;
    return result;
}
```

B.10 SPM Package

This package provides the functions that are used to manipulate the probability p, based on the prediction parameter Π, as well as the final function that combines all functions used so far, to apply the SPM method to a given lattice.

```
// this function implements the decision making process as depicted
// in the flow chart in Figure (13);
int ManipP(long double tau1_t, long double tau1_threshold, int p_input,
           int p_step_size){
    int manipulated_p;
    bool change_indicator_1 = (tau1_t < tau1_threshold);
    bool change_indicator_2 = (tau1_t > tau1_threshold);

    if (!(change_indicator_1||change_indicator_2)){
        manipulated_p = p_input;
    } else{
        if (change_indicator_1)
            if (p_input < 100){
                manipulated_p = p_input+p_step_size;
            } else
                manipulated_p = 99;
        else
            manipulated_p = p_input-p_step_size;
    }

    return manipulated_p;
}

// The "VP" stands for variable p.
// This function applies the SPM method to a lattice spl,
// with the initial gauge matrices gm, and returns the result
// with the formatting type ReturnVal, where ReturnVal.script_e
// gives the resulting minimum value for Script_E,
// ReturnVal.i_pointer is the list of the resulting probabilities
// for every step, ReturnVal.ld_pointer1 is the list for Tau_1(t)
// and ReturnVal.ld_pointer2 is the list for Script_E(t).
ReturnVal ScriptEDescentVP(int ind, MatrixVector** spl,
                           MatrixVector** u_gauge,
                           GaugeMatrices** gm,
                           int p_start, int p_step_size,
                           int interval, int t){
    ReturnVal initial_calc = GetPredicParam(spl,u_gauge,gm,
    p_start,interval);
    long double predic_param = initial_calc.script_e;
    long double* e_pointer = new long double[t];
    long double* tau1_pointer = new long double[t];
    int* p_pointer = new int[t];
    int p = p_start;

    // This loop transfers the data from the first 11 calculation
    // steps into the array of the current calculation, for the
    // completion of following outputs.
    int i = 0;
    while (i < interval+1){
        e_pointer[i] = initial_calc.ld_pointer2[i];
        tau1_pointer[i] = initial_calc.ld_pointer1[i];
        p_pointer[i] = p;
        i++;
    }
    e_pointer[interval+1] = initial_calc.ld_pointer2[interval+1];

    long double E_t_minus_1 = e_pointer[interval];
    long double E_t;
    long double tau1_t_minus_1 = tau1_pointer[interval];
    long double tau1_t;
    long double tau1_delta;
    long double tau1_delta_mean = 0;
    ReturnVal result;

    int k = interval;
    int j = 0;
    // The 1 in t+1 is a safety factor to make sure to also get a measurement
    // from the last 10 sweeps.
    while (k < t+1){
        p_pointer[k] = p;

        LatticeUpdate(spl,u_gauge,gm,p);

        E_t = MinimizingFunctionE(u_gauge);
        tau1_t = Tau1_t(E_t,E_t_minus_1);
        //tau1_pointer[k] = tau1_t;
        tau1_delta = log(tau1_t)-log(tau1_t_minus_1);
        tau1_delta_mean += tau1_delta;

        //e_pointer[k+1] = E_t;
        E_t_minus_1 = E_t;
        tau1_t_minus_1 = tau1_t;

        j++;
        if (j == ind){
            p = ManipP(tau1_delta_mean/interval,predic_param,p,p_step_size);
            tau1_delta_mean = 0;
            j = 0;
        }

        k++;
    }

    result.script_e = E_t;
    result.i_pointer = p_pointer;
    result.ld_pointer1 = tau1_pointer;
    result.ld_pointer2 = e_pointer;
    return result;
}
```

B.11 Multi-Start Package

The following functions combine the SPM method with the multi-start approach and create a useful output into a text file. The used parameters have the following meaning:

- **p_step_size** is the size of the increments to increase/decrease the probability p. The default value is 1.
- **t_max** is the maximal number of calculated sweeps. The default value is 150.
- **ind** is number of sweeps, over which the average of the ascent rate is calculated to compare it to the prediction parameter. The default value is 5.
- **rr** represents the number of random restarts used and the default value is 5.
- **ID** can be used to give the output an ID, if multiple calculations are done. The default value is 1.

- **interval** defines the interval of sweeps, used to get the prediction parameter. The default value is 10.

```
// This function generates a proper file name, based on
// the given file directory and the file ID;
string FileNameGen(string ID, string fdirectory){
    string fname = fdirectory;
    fname += ID;
    fname += ".vp_result";
    fname += ".txt";

    return fname;
}

// reads in the fixed lattices
// lattice_size specifys the lattice size;
// lattice_nr gives the number of the lattice that we want to read in;
// lattice_type selects between the "stl" and "gml" types of lattices;
// The file directory "fdirectory" must have the shape "C:\\Users\\User\\" and
// the input lattice must be saved in this direcotry under the name
// "LxL.stl.txt", where L denotes the lattice size.
// The stl file must be saved in the same directory as the gml file.
string Fixed_Lattice_Input(int lattice_size, string lattice_type,
string fdirectory, int ID){

    string input_name = fdirectory;
    string aux = to_string(lattice_size);
    input_name += aux;
    input_name += "x";
    input_name += aux;
    input_name += ".";

    input_name += lattice_type;
    input_name += ".";
    input_name += to_string(ID);
    input_name += ".txt";

    return input_name;
}

void Output(int ind, int rr, MatrixVector** spl,
            GaugeMatrices** gm_aux, int p_start, int p_step_size,
            int interval , int t, string fdirectory, int run_count){

    string ID = to_string(run_count);

    string fname2 = FileNameGen(ID,fdirectory);
    ofstream out_2(fname2.c_str());

    int kmax = rr+1; //number of random restarts
    int i = 1;
    while (i < kmax){

        // this is initializing an auxiliary array with the
        // gauge matrices (gm)
        GaugeMatrices** gm1 = FillGMVector(gm_aux);
        // this is initializing an auxiliary lattice with the
        // gauge transformed matrices
        MatrixVector** u_gauge1 = FillGaugeTransformedLattice(spl,gm1);
        long double script_e =
                ScriptEDescentVP(ind,spl,u_gauge1,gm1,p_start,
                                 p_step_size,interval,t).script_e;

        out_2 << "Run:␣" << run_count << "␣RR:␣" << i << "␣min_vp:␣" <<
              setprecision(14) << script_e << endl;

        i++;
    }

    ifstream in("vp_result.txt");
    out_2.close();
}

// This function applies the multi-start approach to the SPM mehod.
// The file directory "fdirectory" must have the shape "C:\\Users\\User\\"
void Final_Output(int kN_x, string fdirectory, int p_step_size, int t_max,
                  int ind, int rr, int ID, int interval){

    // get starting time stamp
    auto t = std::time(nullptr);
    auto tm = *std::localtime(&t);
    cout << "Time␣stamp,␣Start:␣" << std::put_time(&tm, "%d-%m-%Y␣%H-%M-%S")
         << endl;

    string stl_input_name = Fixed_Lattice_Input(kN_x, "stl", fdirectory);
    string gml_input_name = Fixed_Lattice_Input(kN_x, "gml", fdirectory);

    MatrixVector **p_spl = CreateLatticePointer();
    ReadInSTL(p_spl, stl_input_name);

    GaugeMatrices **p_gm_aux = CreateGMVectorPointer();
    ReadInGML(p_gm_aux, gml_input_name);

    Output(ind, rr, p_spl, p_gm_aux, p_start, p_step_size,
           interval, t_max, fdirectory, ID);

    t = std::time(nullptr);
    tm = *std::localtime(&t);
    cout << "Run␣" << ID << "␣completed:␣" <<
         std::put_time(&tm, "%d-%m-%Y␣%H-%M-%S") << endl;

    //getting ending time stamp
    auto t_end = std::time(nullptr);
    auto tm_end = *std::localtime(&t_end);
    cout << "Time␣stamp,␣End:␣" <<
         std::put_time(&tm_end, "%d-%m-%Y␣%H-%M-%S") << endl;
}
```

B.12 Main Function

Below is the main function that can be executed, after implementing all packages above, to calculate the minimum of $\mathcal{E}(t)$ according to the SPM method. Before executing the function, the file

directory, where the results will be saved in a text file, must still be specified. This is marked with the phrase **"C:**
Users
User
...
" in the main function. Furthermore, all parameters of the calculation can be specified, some of them are part of the main function and some can be found in the Parameters and Structs Package, see Section B.2. The lattice, for which the minimum of $\mathcal{E}(t)$ should be calculated, must be saved in the same directory, as is used for the output, and must be named as follows **"LxL.stl.txt"**, where the **integer** L represents the lattice size.

```
int main() {

    string fdirectory = "C:\\Users\\User\\...\\";
    int p_step_size = 1;
    int t_max = 150;

    int ind = 5; // number of sweeps, over which we take the average of the
                 // ascent rate to compare it to the prediction
                 // parameter;
    int rr = 5; // number of random restarts

    int interval = 10; // interval of sweeps, used to get the
                       // prediction parameter;

    // this is initializing an auxiliary array with the initial
    // gauge matrices (gm)
    GaugeMatrices** p_gm_aux = FillInitialGML();
    GiveOutGML(p_gm_aux,fdirectory,kN_x, ID);

    Final_Output(kN_x,fdirectory,p_step_size,t_max,ind,rr,ID,interval);
    return 0;
}
```