Imperial College London
Department of Physics

Numerical Computation of the Mass of a Quantum Sine-Gordon Soliton in (1+1) dimensions

Author:
Linas Bitkevičius

Supervisor:
Dr. Arttu Rajantie

Submitted in partial fulfilment of the requirements for the degree of Master of Science of Imperial College London

September 23, 2011

Contents

1 Introduction

Solitons were first observed on water in 1834 by John Scott Russel [1], who described them as "waves of translation". He dedicated some time to find out some properties of those waves and deduced that they are stable and can travel over very large distances and that the speed of those waves depend on their height and width. He also observed that the waves will not merge together, but rather, smaller ones will always be overtaken by larger ones. Theoretically these waves, called solitons, were explained by Diederik Korteweg and Gustav de Vries in 1895. A solution to an equation that is named after them now describes these solitary waves precisely. Since then people got interested in these kind of objects and the theory is now being used to describe various theoretical phenomena ranging from dualities between theories, as in this case, to solitons in DNA [2].

It turns out that certain equations have those kinds of "solitary wave" solutions, and sine-Gordon is one of them. They have been studied extensively in [3, 4, 5, 2, 6, 7, 8, 9, 10, 11]. Sine-Gordon field theory is a scalar field theory that has bosons as their fundamental particles that also admits topological soliton solutions, which fit into the multiple expectation values of the field. It follows that sine-Gordon equation describes soliton-like waves, which are called kinks. They are jumps in the value of the field that correspond to energy density bumps in the space-time the field is on. It is particularly interesting to look at the theory in $(1 + 1)$ dimensions, since then the field is one-dimensional in space and solitons in that space can be interpreted as particles moving along this one-dimensional line in time. It turns out that these solitons are fundamentally fermions - there is another theory, the massive Thirring model, that describes those same kind of fermions and the two theories are dual [3]. Hence, study of the sine-Gordon system is twice as valuable, since it describes two theories at once. In particular, the mass dependence of the fundamental fermion of the Thirring model can be described via the parameters in the sine-Gordon theory.

One could be interested in calculating the mass of the kink in order to find the exact relation of the theories. The mass of a quantum sine-Gordon soliton can be obtained by applying the weak-coupling approximation [11, 7, 12]. It is not an exact result and fails at specific values of coupling. It is therefore important to measure the mass numerically, where possible, to check how analytical results agree with non-perturbative calculations. In particular, it is important where the weak coupling expansion fails. However, we will see later that calculation at these values is tricky.

Measuring the quantum sine-Gordon kink mass numerically is a challenge, since one has to simulate the field. Fortunately, quantum mechanical path integrals look much like partition functions in statistical mechanics. Using this, one can build a relation between a quantum field theory and a statistical-mechanical system. This relation then enables us to simulate the field theory and measure the results.

Numerical computations also introduce other challenges, such as fundamental computational discreteness. One has to write a computer program in order to simulate a statistical process, which is never going to fully correspond to the real one. The difference is that in order to get real results, one has to be able to perform an infinite amount of calculations, which is impossible to do. One can, however, perform Monte Carlo simulations on the thermodynamical system in order to sample only those configurations that are probable according to their corresponding Boltzmann weights. This way the simulation does not have to be run for an infinite amount of time and reasonable estimations can be drawn.

To measure the mass of a quantum sine-Gordon kink one has to either measure the correlation functions of the field and use a spectral expansion with the assumption that kink states are ground states [13], or try to simulate the field in the kink state and the ground state and measure the free energy difference in order to find the mass increment [14, 15]. This work is concentrated about the latter method. In order to simulate the kink state, one has to apply anti-periodic boundary conditions to the lattice he is putting his field on [16]. This is achieved by twisting the field values at the boundary of the lattice with a period of the ground states. The twist then forces the field to consider the kink state as it's ground state in the Metropolis algorithm and enables us to measure the required field values. The increments of mass as functions of increments of the parameter in the action are then integrated to find the approximate dependence of mass on that parameter.

In the sine-Gordon case, the mass of the soliton can be obtained by a mathematical calculation, and it would seem that numerical computations are more or less redundant. In fact, most of quantum field theories do not boast exact results, therefore it is important to develop and understand alternative techniques that we can use to find the relevant parameters such as masses of fundamental or other particles.

2 Lattice Quantum Field Theory

2.1 Sine-Gordon theory

Sine-Gordon field theory is a field theory that can be studied both at classical level, and at quantum level. Its name is a reference to the more basic Klein-Gordon field theory, but since the potential for this theory includes a sine (or a cosine) instead of the usual ϕ^2, it is rather unimaginatively renamed. Analyzing sine-Gordon theory in $(1+1)$ dimensions is rather interesting, since the corresponding kinks are then 1-dimensional particles in space-time, rather than strings or walls. Many choose a resemblance to the $\lambda\phi^4$ model [12, 11] and use different parameters, however, here we define the sine-Gordon Lagrangian like this [3, 10]:

$$\mathcal{L}[\phi] = \frac{1}{2}\partial_\mu\phi\partial^\mu\phi - V[\phi], \tag{2.1}$$

$$V[\phi] = -\frac{\alpha}{\beta^2}\cos\beta\phi - \gamma. \tag{2.2}$$

The relevant parameter in the potential is α, which is the "bare mass" of the theory and can be renormalized [17] to produce the result for the mass of the fundamental boson. The parameter is also used to determine the classical and quantum mass of the soliton particles that this theory describes. However, it can be shown, that this boson is not the only particle the theory describes [3].

As mentioned, the theory is a generalization of the $\lambda\phi^4$ Lagrangian, and this can be seen in the case where β is large and higher terms in the Taylor expansion of the cosine in the potential are negligible:

$$\frac{\alpha}{\beta^2}\cos\beta\phi = \frac{\alpha}{\beta^2} - \frac{\alpha}{2}\phi^2 + \frac{\alpha\beta^2}{4!}\phi^4 + ... \tag{2.3}$$

This expansion also fixes γ - it must be chosen to be equal to $-\alpha/\beta^2$, so the energy

$$E = \int_t\int_x dtdx\left(\frac{1}{2}\left(\frac{\partial\phi}{\partial t}\right)^2 + \frac{1}{2}\left(\frac{\partial\phi}{\partial x}\right)^2 + \frac{\alpha}{\beta^2}\left(1-\cos\beta\phi\right)\right) \tag{2.4}$$

of the field minima

$$\phi_n = \frac{2\pi n}{\beta}. \tag{2.5}$$

limit to zero. These zeroes are the infinitely many vacua that the theory has.

Lagrangian (2.1) has the property of symmetry. We are free to shift the field values $\phi \to \phi + 2\pi n/\beta$. Also, the transition from $\alpha \to -\alpha$ corresponds to the transition $\phi \to -\phi$. This means that we can only consider positive α and not care about the negative values.

The theory has another interpretation: at $\phi(x = \pm\infty, t)$ it can assume expectation values which must be one of those minima, if one wants the total energy to be finite and conserved. These minima must not necessarily be the same at both infinities. It follows that there must be a region in space that has a twist from one vacuum expectation value to the other. These twists account for energy density bumps, which can be interpreted as particles that we call a solitons, or kinks. In space-time they are extended objects, strings, that can be interpreted as world-lines of those kinks. Depending on the vacuum expectation values on both sides of the twist, the particle might be called a kink or an anti-kink.

The conservation and finiteness of energy also lets us define topological charge for different field regions on space and which must be conserved, and is derived for the field symmetries of the vacuum expectation values [11]:

$$Q = \frac{\beta}{2\pi} \int_{-\infty}^{+\infty} dx \frac{\partial \phi}{\partial x}. \tag{2.6}$$

The transition from one expectation value to another increases or decreases Q by ± 1. So kinks divide fields to topological sectors that are labelled by charge.

The actual kink and anti-kink solutions are given by the Bogomol'nyi equation for the soliton [11]:

$$\frac{\partial \phi}{\partial x} = \pm \sqrt{2V(\phi)}, \tag{2.7}$$

which gives (the kink solution)

$$\phi_{\text{kink}} = \frac{4}{\beta} \arctan e^{\sqrt{\alpha}(x-x_0)} = -\phi_{\text{antikink}}, \tag{2.8}$$

assuming that the kink is at x_0. This solution shows that if $x \to -\infty$ then $\phi \to \phi_0$ and as $x \to +\infty$ then $\phi \to \phi_0 + 2\pi$.

In fact, excluding the described solitons and anti-solitons, the sine-Gordon particle spectrum also includes other solutions, namely, the breather, which is a kink and antikink bound state, as well as scattering states.

Since kinks are energy bumps, they are felt if these kind of fields are present. The energy accounts for a mass, which classically can be obtained by doing the (2.4) integral for the soliton field solutions (2.8) [11]. This is the mass

$$M_{\text{classical kink}} = \frac{8\sqrt{\alpha}}{\beta^2}. \tag{2.9}$$

It is a different story in the quantum case. One has to include approximations and renormalize the theory in order to get the result [12]. Quantum kink mass can be obtained by expanding the potential energy around the kink vacuum expectation value. The resulting expression can then be approximated to the second order (weak coupling limit) to reduce to a set of simple harmonic oscillators. Normal mode frequencies are then obtained, and the quantum corrections to the classical kink mass are then just the sum of the zero-point excitations. The corrections to this result are obtained by perturbation theory. The quantum kink mass then has to be renormalized. Renormalization is performed by normal-ordering the Hamiltonian. Counterterms are chosen so that divergences can be cancelled and the final expression for the quantum kink mass is obtained:

$$M_{\text{quantum kink}} = \frac{8\sqrt{\alpha}}{\beta^2} - \frac{\sqrt{\alpha}}{\pi} + \mathcal{O}(\beta^2). \tag{2.10}$$

Although simple, this result is not entirely exact, but for small β it should hold.

There is an interesting correspondence of the sine-Gordon theory with the massive Thirring model [3], which is surprising, since it is a theory whose fundamental particles are fermions. The Lagrangian of this model looks like this [4]:

$$\mathcal{L}_{\text{MT}}[\psi] = \bar{\psi} i \gamma_\mu \partial^\mu \psi - \frac{\lambda}{2} j_\mu j^\mu - M \bar{\psi}\psi, \tag{2.11}$$

where

$$j^\mu = \bar{\psi}\gamma^\mu\psi. \tag{2.12}$$

The two theories are dual with these identifications:

$$M = \frac{8\sqrt{\alpha}}{\beta^2}\left(1 - \frac{\beta^2}{8\pi}\right), \quad 1 + \frac{\lambda}{\pi} = \frac{4\pi}{\beta^2}, \tag{2.13}$$

where we see that the fermion mass is the same as the mass of the quantum kink. Actually, the fermions and antifermions of the Thirring model are dual to solitons and antisolitons in the sine-Gordon theory, and fermion bound states correspond to quantized breathers. This duality also restricts the sine-Gordon theory to only have soliton and anti-soliton solutions when $\beta^2 < 8\pi$.

2.2 Euclidean action correspondence with statistical mechanics

This section is widely discussed in [18]. The action for the sine-Gordon theory (2.1) is

$$iS[\phi] = i\int_t\int_x dtdx\left[\frac{1}{2}\partial_\mu\phi\partial^\mu\phi + \frac{\alpha}{\beta^2}\left(\cos\beta\phi - 1\right)\right]. \tag{2.14}$$

In order to be able to do some numerical calculations, one is forced to do a Wick rotation on the action, i.e., make the transformation $t \to -it$. This is also sometimes called analytical continuation, and it makes sense, since provides an invertible map between the given Minkowskian action and a Euclidean one (although statistical errors in simulations using Euclidean action tend to produce infinities in the Minkowskian picture). It is especially useful in discretized theories with potentials that are not bounded from above, such as the $\lambda\phi^4$ model. Furthermore, after this transformation, field theories generalize to statistical mechanical systems that can be simulated and whose macroscopic values can be measured using numerical Monte Carlo techniques. After the rotation, $iS[\phi]$, which appears in the quantum path integral and the Minkowskian partition function

$$Z = \int \mathcal{D}\phi e^{-iS[\phi]}, \tag{2.15}$$

now turns into S_E, the Euclidean action:

$$S_E[\phi] = \int_t\int_x dtdx\left[\frac{1}{2}\left(\frac{\partial\phi}{\partial t}\right)^2 + \frac{1}{2}\left(\frac{\partial\phi}{\partial x}\right)^2 + \frac{\alpha}{\beta^2}\left(1 - \cos\beta\phi\right)\right], \tag{2.16}$$

where the Euclideanization process included an extra minus sign for convenience. Note how the Euclidean action is purely real and appears to look like a Hamiltonian of a 3-dimensional statistical system. In fact, the minimum of this action for any given field value does not go lower than zero and it really does correspond to a total energy. The partition function correspondingly changes to

$$Z = \int_{pbc} \mathcal{D}\phi e^{-S_E[\phi]}. \tag{2.17}$$

where "pbc" indicates that the integration is now with periodic boundary conditions, which are later introduced on the discretized lattice.

2.3 Discretization

Computational simulations are never going to be able to reproduce exact phenomena of the real world, but they can get very close. The problem is that computers do not understand infinities. Getting the exact numerical value of an integral would require an infinite amount of time. The solution is to discretize the continuous space and time and periodicize them. The field is then put on such a lattice and is discretized as well. As a consequence, computation of continuum quantities involving integrals over all space-time, such as the action of a given configuration, is errorneous, but yields manageable results.

For space-time, discretization is done like this [18]:

$$x_\mu \to xa, \quad x = 0, 1, ..., N-1, \tag{2.18}$$

where in this case $x_\mu = (t, x)$ and a is the spacing between two lattice points, that is usually set to some value to produce best results, and N is the number of points the dimension of the lattice has. The size of the two-dimensional lattice becomes $L^2 = (Na)^2$. With this latticization, the field is no longer continuous, and becomes discretized:

$$\phi(t, x) \to \phi_{tx}. \tag{2.19}$$

Integrals also get discretized and turn into sums over the lattice points. This is usually done as follows:

$$\int_0^L dx_\mu \to a \sum_{x=0}^{N-1}. \tag{2.20}$$

The sums are now over all points in the dimension and are multiplied by the distance between those points. Derivatives are changed to differences between two points of the field on the lattice according to the continuum definition of the derivative of a function:

$$\partial_\mu \phi = \frac{1}{a} \left(\phi_{x+a\hat{\mu}} - \phi_x \right). \tag{2.21}$$

Here the hat denotes a unit vector in a given direction of x or t. If we choose to restore continuity, we have such a possibility:

$$a \to 0, \quad \partial_\mu \phi \to \frac{\partial \phi}{\partial x_\mu}. \tag{2.22}$$

In fact, we periodicize the field in order to be closer to the continuum limit, so that the (1+1)-dimensional field lies on a 2-torus:

$$\phi_{x+L} = \phi_x. \tag{2.23}$$

The field is now effectively continuous, albeit periodic and discrete. We can control this "discreet continuity" by tuning the parameters L and N. As $N \to \infty$, then $a = L/N \to 0$, and this accounts for the continuum limit of the theory. This analytic possibility always remains open throughout the simulation.

It is now possible to rewrite the Euclidean action (2.16) in terms of the latticisized field variables:

$$S_E[\phi(x)] \to S_E[\phi_{tx}] = \sum_{tx} \left[\frac{a^2}{2} \partial_\mu \phi_{tx} \partial_\mu \phi_{tx} + \frac{\alpha a^2}{\beta^2} \left(1 - \cos \beta \phi_{tx}\right) \right]. \tag{2.24}$$

The action of a given configuration can now be calculated by summing the derivatives (2.21) and the terms in the potential over all points in the lattice. It follows that this action also obeys the classical continuum limit, although it is important to note that different discretization techniques produce different errors in the statistical simulation of the theory and it is a delicate matter.

The partition function effectively turns into

$$Z = \mathrm{Tr} e^{-H(t_+ - t_-)} = \sum_n e^{-TS_E[\phi_{tx}]}, \tag{2.25}$$

where n is the number of states, a very large number, H is the full Hamiltonian of the system, which is the Euclidean action, and $(t_+ - t_-)$ is the periodic time difference, which thermodynamically means that the temperature of the system is finite.

2.4 Mass measurement

The aim of this thesis is to give a numerical result of the sine-Gordon kink mass and to check if it fits the analytic result provided in the section (2.1). Since we turned our field theory into a thermodynamical system, it is possible to measure expectation values of the field directly after sampling a grand ensemble of fields in the Boltzmann distribution using the Metropolis algorithm. The calculation of mass is then widely discussed in [16, 15, 14, 13]. These methods include using twisted boundary conditions on the spatial direction of the lattice:

$$\phi_{tx} = \phi_{t(x+L)} + \frac{2\pi}{\beta}. \tag{2.26}$$

A twist turns the ground state of the field to the kink state, or makes "the field like to be twisted". It is performed by equating the $(x+L)$'th element of the field to $x + 2\pi/\beta$, the period of the theory. These terms only affect the value of the action through the kinetic derivative terms, since at the boundary of the lattice in the x direction the action minimizes only when the points $x = 0$ and $x = L - 1$ differ by $2\pi/\beta$. While at any time the field with periodic boundary conditions may contain an even number of kinks, this effectively generates an odd number of kinks on the lattice. The kink - anti-kink pairs decay, and after some time only one kink remains. The Metropolis algorithm, described later, now generates a grand ensemble of field states around the kink state, which we can sample to produce the expectation value of the twisted field. The mass of the kink is can be obtained by measuring the difference of the free energies at the twisted and periodic cases respectively divided by the lattice time length. This is obtained by noting that the topological charge is quantized and in practice it's difference can be controlled by using twisted boundary conditions. This means that the partition function of the theory is just the sum of the partition functions at different topological charges Q of equation (2.6). In particular, we can write [15, 14, 13]

$$Z_1 = Z_0 e^{-M_{\text{kink}} T}. \tag{2.27}$$

Z_1, the partition function for the states with $Q = 1$, is Z_{tw}, since we twist the boundary conditions to generate the one-kink state, whereas Z_0 is the partition function for the untwisted pbc $Q = 0$ state Z_{p}. From this we can obtain the mass for the kink, which is:

$$M_{\text{kink}} = \frac{1}{T} \ln \frac{Z_{\text{tw}}}{Z_{\text{p}}} = \frac{1}{T} \left(F_{\text{tw}} - F_{\text{p}} \right). \tag{2.28}$$

Although the expression is really simple, it is impossible to measure the partition functions directly, they are only sampled. But since the partition functions are integrals of the exponentials of the Euclidean action, (2.17), it is possible to take the derivative of the mass with respect to a term in the action and then integrate it from a value that is known, such as the free theory, where the mass is zero:

$$\frac{\partial M_{\text{kink}}}{\partial \rho} = \frac{1}{T} \left[\left\langle \frac{\partial S_E[\phi_{tx}]}{\partial \rho} \right\rangle_{\text{tw}} - \left\langle \frac{\partial S_E[\phi_{tx}]}{\partial \rho} \right\rangle_{\text{p}} \right]. \tag{2.29}$$

In the sine-Gordon theory, the bare mass is α, so we choose $\rho = \alpha$ and get

$$\frac{\partial M_{\text{kink}}}{\partial \alpha} = \frac{aL}{\beta} \left[\langle \cos \beta \phi_{tx} \rangle_{\text{p}} - \langle \cos \beta \phi_{tx} \rangle_{\text{tw}} \right]. \tag{2.30}$$

We know that at $\alpha = 0$ the theory is a free scalar field and has no topological defects, since there is no potential, the kink at this limit is non-existent. Having this in mind, it is possible to use Simpson's integration, or any other similar method to integrate this from $\alpha = 0$ to get the final result at the desired α. Unfortunately, this method has linear error propagation, which increases the error with increasing the number of measurement points and makes the estimation of the value so much more difficult. It is unreliable. In practice, it is better to use the finite difference method, which has better error control, since it is a logarithm, to measure it. The change in the soliton mass is given by the change in partition functions (2.25) expressed by the change of the actions of the twisted and periodic cases in (2.28) [15, 14, 13]:

$$\Delta M_{\text{kink}} = -\frac{1}{T}\ln\frac{\Delta Z_{\text{tw}}}{\Delta Z_{\text{p}}} = -\frac{1}{T}\ln\frac{\sum_n e^{-T\Delta S_E[\phi_{tx}]_{tw}}}{\sum_n e^{-T\Delta S_E[\phi_{tx}]_{p}}} = -\frac{1}{T}\ln\frac{\langle e^{-T\Delta S_E[\phi_{tx}]}\rangle_{\text{tw}}}{\langle e^{-T\Delta S_E[\phi_{tx}]}\rangle_{\text{p}}} \tag{2.31}$$

Finally, mass increment can be rewritten in terms of $\Delta\alpha$ and (2.24):

$$M_{\text{kink}}(\alpha_2) - M_{\text{kink}}(\alpha_1) = -\frac{1}{T}\left(\ln\frac{\left\langle \exp\left(\frac{(\alpha_2-\alpha_1)a^2}{\beta^2}\sum_{tx}(\cos\beta\phi - 1)\right)\right\rangle_{\alpha_{1,\text{tw}}}}{\left\langle \exp\left(\frac{(\alpha_2-\alpha_1)a^2}{\beta^2}\sum_{tx}(\cos\beta\phi - 1)\right)\right\rangle_{\alpha_{1,\text{p}}}}\right). \tag{2.32}$$

The mass difference can be calculated by taking the logarithm of the changes of the partition functions measured at α_1. This formula can also be used to reduce the error, since it is actually smaller when the number of measurements is higher compared to the naive method (2.30) where the opposite is true. If one also measures the expectation values at α_2 and finds the same negative increment, it may be checked that they agree to determine if the system was properly equilibrated. Note that the measurements measure the same thing and are essentially the same. They both use the same principle - change in the action due to the change of the parameter in the action. The difference is then that they are measured differently. But if both methods are fulfilled correctly - they should in principle produce same results.

3 Monte Carlo method

Since the sine-Gordon field theory has been turned into a thermodynamical system, it is time to make use of that by making simulations of the theory. Monte Carlo method is discussed in [19, 20]. The whole point is about estimating the partition function of the theory. This is done by associating the probabilities of states appearing in the system with weights which those states would have in real systems. Achieving this requires us to set the transition probabilities from one state to another during the simulation in exactly this manner. By choosing those transition probabilities in such a way, that the equilibrium solution is the Boltzmann distribution. Then the samples that are taken from these generated states are also in Boltzmann distribution, and provide a rough estimate for the partition function. The advantage of this technique is that one does not need to sample a large bit of the partition function in order to get accurate estimates of the properties of the system. The disadvantage is that one cannot get rid of the statistical errors using this method. The measured partition function is not smooth and if one calculates expectation values by taking the derivative of the partition function, he does not get accurate results, since non-smooth functions have statistically random derivatives. Instead one calculates as many values as one can, and then the estimates get better.

3.1 The estimator

The usual goal in Monte Carlo simulations, as is the case in this thesis, is to compute the expectation value $\langle E\rangle$ of some observable E, such as the cosine of the field. The best way to calculate an expectation value is to average the desired quantity over all states in the system each of them weighted by the Boltzmann weight [19]:

$$\langle E\rangle = \frac{\sum_s E_s e^{-\beta H_s}}{\sum_s e^{-\beta H_s}}, \tag{3.1}$$

where H_s is the total energy in the state s and β is the inverse temperature of the system. This result is only good for small systems, where the state space is small enough for a computer to handle. In the case where the state space is large, or infinite, we can only choose a subset $\{s_1, s_2, ..., s_N\}$ of those values to produce only the estimate of the expectation value. This estimator would then look like this:

$$E_N = \frac{\sum_{i=1}^{N} E_{s_i} e^{-\beta H_i} / w_{s_i}}{\sum_{j=1}^{N} e^{-\beta H_j} / w_{s_j}}. \tag{3.2}$$

E_N is called the estimator of E. If we set $N \to \infty$, then the estimator has the property that $E_N \to \langle E \rangle$, or, as we increase the number of sampled states, the estimation of E comes closer and closer to $\langle E \rangle$. Now if we pick both probability distributions w_{s_i} to be equal, the expression simplifies, but can we can get an accurate estimate of $\langle E \rangle$ using this method? We have

$$E_N = \frac{\sum_{i=1}^{N} E_{s_i} e^{-\beta H_i}}{\sum_{j=1}^{N} e^{-\beta H_j}}. \tag{3.3}$$

The problem with this expression is that whatever set of samples we take, the probability that these samples contribute the most to the partition function is very small. Consider a 2-dimensional 64^2 lattice with the field value on each point. The state space is then $\mathbb{R}^{64^2}$, which is an infinite number of states (well, not really infinite, since again, the computer does not understand infinities, so the range of values the field can take is actually finite). The problem is that for most thermodynamical systems, the states that contribute the most to the partition function are very very rare, sometimes even only the ground state, among the infinitely many states, is the dominant contribution, if the system is unlikely to jump to the next energy state. This sampling method of taking any state with some given probability distribution is therefore very time-consuming, since we need to take as many samples, as to include the relevant information about the partition function. Hence, the real problem is sampling the states that are contributing the most to the partition function. This is called Importance sampling.

Importance sampling is a technique used to determine the expectation value of a desired variable. It allows only the most contributing states and then the expectation value can be obtained by taking an average of the values corresponding to those states.

If we try to take the samples in the system in which the state appearance is proportional to its Boltzmann weight, we can mimic nature, which naturally puts thermodynamical systems in such a distribution to get the best results. The plan is then not to set the probability distributions in (3.2) to be equal to one another, but rather set them to the Boltzmann distribution of the system, that is, set $w_{s_i} = \frac{e^{-\beta H_i}}{Z}$. Then (3.2) becomes

$$E_N = \frac{1}{N} \sum_{i=1}^{N} E_{s_i}, \tag{3.4}$$

a neat and tidy expression, only the average of the measured values of an observable. The nice property remains, that as $N \to \infty$, $E_N \to \langle E \rangle$. And the more values one takes, the more accurate is the result. Now the measurement accuracy only mostly depends on the time one spends sampling the states that are generated. The relative frequency of sampling different states is proportional to the time that the system spends in those states, and so in this way one can mimic a thermal system. The way one picks states from the Boltzmann probability distribution is by using the Markov process.

3.2 Markov chain

Having established the way to measure expectation values in the theory, it is important to be able to generate such states with the probability that correspond to the Boltzmann distribution of the system. One might think that a remedy to such a process would be to generate any state and then accept it in such a way that it fits the distribution. In fact, one would reject almost every state, since the state space is practically infinite. One must rely on a Markov process to be able to generate states that are most likely in the distribution. A Markov chain is a thermodynamical system in which the transition probability of to every new generated state depends only on the original state and on the new generated state, and also, transition probabilities must not vary over time [19]. One can define the transition probability of the system changing from the state i to the state j as $P(i \to j)$. This means that the state j will always have the same probability to be generated given that the current state of the system is i even including every other factor. $P(i \to j)$ also satisfies

$$\sum_j P(i \to j) = 1. \tag{3.5}$$

The probability that a state will be generated, given that the current state is i, is equal to one. This Markov process generates a Markov chain of states and is chosen is such a way, that the states that appear are given by the probabilities that are in turn given by their respective Boltzmann weights. When run for a substantial amount of time, the system equilibrates, and starts to generate the required chain of states. The drawback of this chain of states is that every state is correlated to every other state. This inevitably accounts to an error in the overall calculation of the expectation values of any given variable that depends on the field on the lattice. The conditions that one must impose on the process are ergodicity and detailed balance.

3.3 Ergodicity and detailed balance

Ergodicity in the Markov process is a condition that requires every state to be reachable from every other state [19]. If this would not be the case, then some states that might be very probable in the system might have the transition probability zero, and hence, be unreachable. The system would effectively be a ”sum” of some separate systems. In practice, we can always set some probabilities to zero, or they can occur naturally, but there must always be a finite path that enables to reach every state from any other state. The algorithm that is used to simulate the sine-Gordon theory also satisfies the ergodicity condition.

Another condition is detailed balance, which is ensures that the states that are being generated are in Boltzmann distribution. When the system is at the equilibrium, the rate at which the system goes from one state to another is equal to the inverse probability for the other state to go back to the first one:

$$\sum_j w_i P(i \to j) = \sum_j w_j P(j \to i). \tag{3.6}$$

Using (3.5), this immediately turns into

$$w_i = \sum_j w_j P(j \to i). \tag{3.7}$$

A set that has transition probabilities that satisfy this equation and corresponds to the distribution p_i, is an equilibrium of the Markov process.

However, this still does not guarantee that the probability distribution will tend to the chosen one. An additional constraint must be imposed in order to get the needed Boltzmann distribution:

$$w_i P(i \to j) = w_j P(j \to i). \tag{3.8}$$

This constraint is called detailed balance. If it is satisfied, then (3.6) is also satisfied, since it is just a sum over i on both sides. This condition straightforwardly implies that the transition probability from state i to state j is equal to the transition probability for the opposite transition and that the rates at which the system goes from one state to another and vice versa balance out. Now, after equlibration, the probability distribution tends exponentially towards the one that we want to impose. Since we want the simulation to tend to the states that are given by the Boltzmann distribution, the detailed balance equation (3.8) tells us

$$\frac{P(i \to j)}{P(j \to i)} = \frac{w_j}{w_i} = e^{-\beta(H_j - H_i)}. \tag{3.9}$$

The equilibrium of states of the Markov process is now in the Boltzmann distribution. One only has to write a computer program to simulate the process.

3.4 Acceptance ratios

In general, it is difficult to find the best transition probabilities for the Markov process in order to be able to generate the required states efficiently. This problem can be solved by introducing something called the acceptance ratio. In principle, one can make the system to always stay in one place by setting the transition probability $P(i \to i) = 1$. Then the detailed balance equation (3.8) trivially states $1 = 1$. This means that one can choose whatever value for $P(i \to i)$ for it to satisfy that equation. In turn, this allows for space to adjust the value of $P(i \to j)$ to keep the sum rule (3.6) satisfied by keeping balance throughout the sums. So one can keep changing the transition probabilities for the state i to change to state j as long as one keeps track that the transition probability for staying in the state i is kept within the range $[0, 1]$ (because it is still a probability) and then the detailed balance is still satisfied. Furthermore, as one changes $P(i \to j)$, one can also change $P(j \to i)$ respectively to keep (3.9) satisfied, so that the ratio is preserved. This means that one can control the set of transition probabilities just by controlling $P(i \to i)$.

In order to illustrate this, one can break up the transition probability into two parts. When simulating the Markov process, one constantly generates new states given an old state. Then transition probability actually reads

$$P(i \to j) = S(i \to j)A(i \to j). \tag{3.10}$$

In other words, transition probability can be broken up to the probability that the state is going to be generated times the probability that the generated state is going to be accepted. The quantity $S(i \to j)$ is called the selection probability. It is the probability that the algorithm, given a state i, will generate a new state j. The quantity $A(i \to j)$ is the probability that the new state is going to be accepted, and is called the acceptance ratio. When a new state is generated, it should be accepted with the probability $A(i \to j)$, and rejected with probability $(1 - A(i \to j))$, and the system should stay in the same state i. (3.9) then reads

$$\frac{P(i \to j)}{P(j \to i)} = \frac{S(i \to j)A(i \to j)}{S(j \to i)A(j \to i)}. \tag{3.11}$$

A subtle balance arises here in the simulation, since the acceptance ratio is controlled by the difference in the thermodynamical energies of the new and old states, as seen in (3.9), and since the ratio $A(i \to j)/A(j \to i)$ can take any value from zero to infinity, the programmer can adjust the values of the selection probabilities any way he likes. The challenge is then to find the way to maximize the acceptance ratio while minimizing the time the algorithm takes to generate the required states. For instance, if the acceptance ratio is low (this might happen if the energies of the two different states differ by a large amount), then the algorithm is going to take a while to "walk out" all of the states that happen to contribute the most to the partition function, and is going to be very inefficient. It is very frustrating to find out after 12 hours of runtime that only a couple of relevant states have been reached.

A solution to this problem might be to notice that both acceptance ratios can be multiplied by the same constant. Since the detailed balance equation only needs the ratio of those ratios, the constants cancel out. One has to watch out, though, that the acceptance ratios don't step out of the range $[0, 1]$. The solution then is to make the larger acceptance ratio to be equal to one, and adjust the other so that (3.9) is satisfied. This is the heart the Metropolis algorithm.

3.5 Metropolis algorithm

The Metropolis algorithm was first proposed by Metropolis and his colleagues in 1953 in a paper about hard sphere gasses [21, 19]. It makes use of a specific acceptance ratio that optimizes the transition probabilities for the generation of states in the Boltzmann distribution of the Markov chain.

One has to choose a set of probabilities $S(i \to j)$ for every possible transition $i \to j$, and then choose the acceptance ratios $A(i \to j)$ for each of these states such that (3.8) is satisfied. Then the algorithm accepts the new state randomly according to the chosen acceptance ratio. Otherwise, it is left at the same state it was before. The process is then repeated again and again until the system equlibrates and the new generated states j are being generated with the probabilities of their corresponding Boltzmann weights.

The algorithm makes use of "single-spin-flip dynamics" (name taken from the Ising model). In general, one can change the values of the lattice at every point simultaneously and then check whether to accept or reject the state. However, since the energy of the state generated in this way will most probably differ by a large value from the last state, the acceptance ratio is therefore going to be very small (in fact, as $n \to \infty$, where n is the number of values that get changed simultaneously, $A(i \to j) \to 0$). Hence, the algorithm is going to be very inefficient. The most efficient way to get the most subtle changes in energy and, therefore, high acceptance ratios, is to only change one value of the lattice at a time and then check the energy difference to accept or reject the new state. Also, using this kind of dynamics ensures that ergodicity is satisfied - it is obvious that every possible state can be reached from any given state in a finite number of lattice updates.

In the Metropolis algorithm one wants to set $S(i \to j) = S(j \to i)$, which means that then in (3.9) they cancel out, leaving only the ratio of acceptance ratios for the ratio of the transition probabilities. In fact, when one puts a field on the lattice (as is the case in lattice quantum field theories), the field is left a possibility to take ranges from $-\infty$ to $+\infty$. This, however, is never the case (nor is it going to happen naturally), since, again, computers don't have an infinite amount of memory. Also, due to the limited precision of calculation (for the same reason), the values of the field are going to be incremented. This means that the set of states j that can be generated from the set of states i by adding a random number to a value of a field on a lattice is finite. Hence, the selection probabilities

$$S(i \to j) = S(j \to i) = 1/m, \tag{3.12}$$

where m is the number of possible states j, a very large number. When considered in (3.11), this means

$$\frac{P(i \to j)}{P(j \to i)} = \frac{S(i \to j)A(i \to j)}{S(j \to i)A(j \to i)} = \frac{A(i \to j)}{A(j \to i)} = e^{-\beta(H_j - H_i)}. \tag{3.13}$$

That is, selection probabilities can be completely get rid of, and the programmer can set them to whatever value he wants to. One usually sets them to unity to get the fastest result, since we always want to be generating new states.

The only thing left to do, is to introduce a constant in front of A to adjust the acceptance ratios optimally. In principle, those ratios could be set so that the highest value they can take is 1, and the lowest - zero. Although this is a possibility, the control over ratios that have such a functional form is problematic and the algorithm can become inefficient really fast. In fact, the detailed balance equation doesn't constrain the functional form of $A(i \to j)$ to be the exponential. It could be changed in whatever way one wants to, as long as the (3.7) and (3.9) are satisfied and the bounds of the function resemble that it is a probability. So in order to maximize the acceptance ratios, the way to do it is to set the higher one to unity - the highest value possible -, and adjust the other in such a way that (3.9) is not violated. In other words, if the energy the Markov process generates a state j and it has a higher energy than the state i, which was the starting state, then naturally, the higher acceptance ratio would be $A(j \to i)$, so we set it to 1. Then this means that $A(i \to j)$ must be $e^{-\beta(H_j - H_i)}$, or in other words, the Metropolis algorithm accepts the newly generated states that have a lower energy without any conditions, but if the energy of the newly generated state is higher, then it is accepted randomly with the described probability. The same subtle balance, mentioned before, arises here: if the energy difference of the two states is large, then the number of accepted states is going to be low, and the algorithm is not going to move a lot. Otherwise, if the energy difference is small enough, such as for the states that are near the equilibrium, the number of accepted states is going to be large, and the algorithm is going to "walk out" much of the surrounding area of the potential minimum. Furthermore, these states are going to be generated and accepted as likely as the Boltzmann distribution tells them to be. It might seem unnatural at first that the lower states are always unconditionally accepted. It might be the case, but it is the most efficient way to do Monte Carlo simulations for thermodynamical systems that describe natural phenomena, such as the magnetization of a ferromagnetic material, or configuration of gasses at a given temperature in a chamber, or the simulation of a quantum field theory (given that the path integral has a thermodynamical realization).

3.6 Implementation of the Metropolis algorithm

Although everything looks so nice and shiny, implementation of the Metropolis algorithm introduces several challenges of its own. First of all, the system that we are simulating is nowhere near to being the same as the real one, it is only an approximation, since, again, computers only have limited amounts of memory. one needs a lattice, which for a 2-dimensional theory is set to some number $N \times N$. On this lattice there must be a theory that can be limited to continuity. This means adding periodic boundary conditions. Essentially, it means setting the last element of the lattice to be the neighbour of the first one and vice versa for each row and column. This makes the theory to be "taken out" as a slice of the big infinite lattice, or effectively just makes the theory periodic over the distance of the length of the lattice. One must also decide on the temperature the theory is being simulated at, and also the initial state of the system. This is closely related to the phenomena of equlibration (this can be reduced by choosing "good" initial states) and thermalization.

Equilibration time is the time the algorithm takes to go from the initial state to the state that is probable according to its Boltzmann weight, or the one that lies around the minimum of the potential of the theory [19]. In other words, when the system has equilibrated, then the algorithm starts generating the Markov chain of states that are in the Boltzmann distribution of the system. It is important to understand when the system has equilibrated before measurements are started, since the results are not going to be the ones that are expected - the system will not be at the equilibrium.

An easy and unreliable method of measuring equilibiration time is just by looking at the lattice and comparing it to the expected results, i.e., knowledge of the ground state of the system. Then one can visually confirm the amount of updates, or sweeps (a sweep is a whole run through the lattice) it takes to equilibrate the system. This requires a person sitting in front of the simulations and measuring the quantity.

Another more exact way of determining equilibration time is to plot some expectation value against time (which is measured in sweeps). For example, if one is simulating a quantum field theory, then one could measure the expectation value of the square of the field, $\langle \phi \rangle$, and plot it against time to see when it more or less starts to wobble around a value. Then it is a strong suggestion that the system has equilibrated. This doesn't get rid of the problem that the system might go down to a local minimum, but then one can run several different simulations with different parameters to find the time in this way, or one can expect a value by simply considering calculating the approximate potential minimum by hand.

Equilibration time is usually quite large if one considers that the system must go to the minimum from a certain predefined configuration. There are several initial configurations that might be considered: the zero-temperature initial configuration and the high-temperature initial configuration. The first one is setting the system to the ground state where the corresponding energy is zero. This might be a good idea, but if the theory has several ground states, then one is essentially choosing the ground state for the system to be in, which should be avoided. The second choice is generating the lattice randomly such that the corresponding energy is high, and then the system thermalizes to some state it wants to be in. this way a preferred ground state is not specified, and the system looks more like a natural process.

Interestingly enough, one usually wants to measure the system over a range of values of a certain parameter, such as the temperature or the bare mass of a fundamental particle of a quantum field theory, say, m. The experimenter would equilibrate the system, measure the expectation values, get some results for one value of m, and increment it to the next value. In such case, it would be unwise to set the lattice to the initial state again. The reason for that is that we expect the ground state of the theory not to differ that much at two very slightly differing m_1 and m_2. We then use the last configuration of the system at m_1 as the starting configuration of the system at m_2 to save as much time as we can. There is still some time the system has to be left for to equilibrate, although that time is much smaller than the equilibration time and the simulation therefore is much quicker if one uses this trick.

3.7 Autocorrelation time

Another annoying thing about the Markov chain is that every new state j depends on the last state i, and i depends on g, g on h, and so on. This means that the states that have been generated are basically correlated with all of the other states that were generated before. This is a problem, since we want to measure just the states in the ensemble and we do not want them to be correlated with each other. Therefore, there is a fundamental error in the calculation of expectation values which we must include in our considerations. Autocorrelation time of a set of values of a sample can be determined using the autocorrelation function [19]:

$$C(t) = \int d\tau \left(x(\tau) - \langle x \rangle\right)\left(x(\tau + t) - \langle x \rangle\right) = \int d\tau \left(x(\tau)x(\tau + t) - \langle x \rangle^2\right). \tag{3.14}$$

This function measures how the value of a continuous function at time t is correlated with the value of the same function at time τ. It is expected to drop exponentially with respect to time:

$$C(t) \sim e^{-t/\tau}. \tag{3.15}$$

Two states are said to be uncorrelated (or rather, correlated as little as to produce a small error and which one can sample) when $t = 2\tau$. Notice that $C(0)$ is just the standard deviation of the data set, so one can normalize the time by dividing it by the standard deviation to get $C(0) = 1$. It means that when one computes the $C(t)$ curve and finds the value of t at which $C(t) = e^{-\frac{1}{2}}$, one determines the autocorrelation time τ. It is normal to make measurements at smaller times than the autocorrelation time, since we usually don't know the autocorrelation time before we start the simulation. Once the measurements are taken, the time can be measured and the relevant values can be sampled. If the values taken are correlated, then the error increases, but if the values are not correlated, then less measurements are taken and in this way, the error increases again.

In practice, drawing autocorrelation time functions usually takes N^2 amount of time, where N is the time taken to process one data element. This happens since the discreet formula for (3.14) is

$$C(t) = \frac{1}{N-t} \sum_{\tau=0}^{N-t} x(\tau)x(\tau+t) - \frac{1}{N-t}^{2} \sum_{\tau=0}^{N-t} x(\tau) \sum_{\tau=0}^{N-t} x(\tau+t), \tag{3.16}$$

which is $\sim N^2$, since one has to do $(N-t)$ sums of N elements. A drawback of this method is also that the errors due to this summation method add up quite fast. In reality, though, it is better if one determines the autocorrelation time that is larger than the real one - this produces a bigger error due to a bigger factor in the error, but the values are not correlated, and that is usually the bigger factor.

Since it takes roughly N^2 amount of time to measure the autocorrelation time for each data set (of whose it could be many), in practice it is quicker and less messier to use another method to make this measurement. One could notice that it is possible to perform a Fourier transform on the integral (3.14):

$$\begin{aligned}C(t) &= \int d\tau \left(x(\tau) - \langle x\rangle\right)\left(x(\tau + t) - \langle x\rangle\right) = \int d\tau \tilde{x}(\tau)\tilde{x}(\tau + t) \\ &= \int dw' \int dw \int d\tau \tilde{x}(w)\tilde{x}(w')e^{-iw'\tau}e^{-iw(\tau+t)} \\ &= \int dw' \int dw \tilde{x}(w)\tilde{x}(w')\delta(w' + w)e^{-iw't} = \int dw \tilde{x}(w)\tilde{x}(-w)e^{iwt} \\ &= \int dw |x(t) - \langle x\rangle|^2 e^{iwt}. \end{aligned} \tag{3.17}$$

Autocorrelation time function is the Fourier transform of the square of the modulus of the data set less its average. Usually (at least in C++) there are large precision number libraries that also include the Fast Fourier Transform algorithms which can be utilized to do computations such as this one. This is relevant, since, compared to the computation time of the function (3.14), the mentioned algorithms only take about $N \ln N$ amount time. In the case of N being a large number, this is a significant increase in the speed of the calculation of autocorrelation time. In principle, one can take a data set, perform a Fourier transform on it, set the first element to zero, since it is just the average (we want to subtract the average, take the modulus squared of the output, and then perform an inverse Fourier transform on the resulting data set. This is going to result to a symmetric result, but if the autocorrelation time is less than a half of the length of the data set, it can be measured in the same way as with (3.14), i.e., $t = 2\tau$ when $C(t) = e^{-\frac{1}{2}}$.

4 Simulation

Once the grounds for the sine-Gordon quantum field theory and Monte Carlo simulations are set, it is time to write the code to simulate the theory to measure the mass of the sine-Gordon kink. In a sense, we are lucky that quantum field theories so easily correspond to thermodynamical systems. Without this "duality", it would be impossible to do numerical calculations and one could only resort to perturbation theory, which is not exact, sometimes so much, that it is effectively nonsense. Then there is an issue of renormalizability, which is not always possible due to the internal terms of the Lagrangian that defines the theory. If the properties of the quanta cannot be obtained like this, it doesn't mean that they don't exist or are unphysical. Numerical calculations give a way to reveal the phenomena of quantum field theories and predict those that perturbation theory cannot solve exactly or approximately.

The program consists of two big pieces: the first one - "simulation.cpp" - simulates the theory and measures the required values, and the second one - "mass.cpp" - measures the expectation values themselves and outputs the mass which is computed using the naive derivative method and the finite derivative method, given the data output from the first program. Two methods of integrating the mass provide a good way to determine if the results are correct.

4.1 Application of theory

The aim of this thesis is to measure the quantum-mechanical mass of the soliton of the sine-Gordon theory. The theory can be simulated using the formulae developed in the previous two sections. Once the field has been put on the lattice, it is possible to measure the mass using (2.30) and (2.30). To measure the mass, one has to generate the field, or if the same simulation is being done the second time, it is possible to load the field from the last simulation. This saves equilibration time, even though it is not as time-expensive as the whole simulation.

As discussed above, Euclidean action corresponds to the negative Hamiltonian of the thermodynamical system corresponding to the quantum field theory being simulated. In turn, equations that are used in the Monte Carlo simulation that were discussed in the previous sections use $H = S_E$ and $\beta = T$. Note that T here is not temperature. It is the lattice length in the time direction, which corresponds to the inverse temperature in the thermodynamical system [18].

The next thing is to do the actual Monte Carlo simulation. In simulation.cpp this is realized in the function "metropolis". This function is also used to perform checkerboard updates (which are going to be explained shortly), equilibrate the system and check whether it was already done so, thermalize between adjacent measuring points of the parameter, write out the estimators for the expectation values and their errors into files (that later get accessed by mass.cpp) and draw the current field (so the simulation can be watched live - if one wants to extremely save computation time, he should rather not include this). It is a big function that does everything and gets called when the main function loads the settings (these include the lattice length, spacing, highest value of α, beta, step size, the number of measurements, and so on) and generates or loads the field.

"Metropolis" starts with a loop that is used to do checkerboard style updates of the field. The reason for such updating process is that one does not want to recalculate the derivatives that are in the discretized action (using (2.24) and (2.21))

$$S_E[\phi_{tx}] = \sum_{tx} \left[\frac{1}{2} \left(\phi_{(t+1)x} - \phi_{tx} \right)^2 + \frac{1}{2} \left(\phi_{t(x+1)} - \phi_{tx} \right)^2 + \frac{\alpha a^2}{\beta^2} \left(1 - \cos \beta \phi_{tx} \right) \right] \tag{4.1}$$

twice while doing only one sweep of updates. This is because the updating process should be as smooth as possible to not to interrupt with the correct Metropolis state generation.

The actual Metropolis is realized in the next step. This function has the value for the current action as an input. That is, the main function passes the value of the action of the generated field, which is computed using "get_action", to function "metropolis", that updates the field and computes the updated action. This step is done in the function "update". The difference between these two action calculation functions is that the first one takes (4.1) and does the sum. This is a long and boring process, which can be avoided, given that the old action is already known. "Update" is more intelligent than "get_action", since it notices that if one updates the field value ϕ_{ij}, then the terms in the action that change are just the neighbouring derivatives and the potential term. So what it does is it subtracts the old neighbouring derivatives and the potential term, and adds the new ones to get the new action:

$$\begin{aligned}
S_E[\phi_{tx}]_{\text{updated}} = S_E[\phi_{tx}] &- \frac{1}{2}\left(\phi_{(i+1)j} - \phi_{ij}\right)^2 - \frac{1}{2}\left(\phi_{i(j+1)} - \phi_{ij} + \delta_{xL}\frac{2\pi\Delta Q}{\beta}\right)^2 \\
&- \frac{1}{2}\left(\phi_{ij} - \phi_{(i-1)j}\right)^2 - \frac{1}{2}\left(\phi_{ij} - \phi_{(i(j-1)} + \delta_{x1}\frac{2\pi\Delta Q}{\beta}\right)^2 \\
&+ \frac{1}{2}\left(\tilde{\phi}_{(i+1)j} - \tilde{\phi}_{ij}\right)^2 + \frac{1}{2}\left(\tilde{\phi}_{i(j+1)} - \tilde{\phi}_{ij} + \delta_{xL}\frac{2\pi\Delta Q}{\beta}\right)^2 \\
&+ \frac{1}{2}\left(\tilde{\phi}_{ij} - \tilde{\phi}_{(i-1)j}\right)^2 + \frac{1}{2}\left(\tilde{\phi}_{ij} - \tilde{\phi}_{(i(j-1)} + \delta_{x1}\frac{2\pi\Delta Q}{\beta}\right)^2 \\
&+ \frac{\alpha a^2}{\beta^2}\left(\cos\beta\phi_{ij} - \cos\beta\tilde{\phi}_{ij}\right),
\end{aligned} \tag{4.2}$$

where δ_{xt} is the Kronecker delta function, ΔQ is the topological charge (2.6) difference between two sectors that are being simulated and $\tilde{\phi}_{ij}$ marks the new updated field value at ij against the usual old field value. This is a much faster function, since it only needs to process five terms in the action, whereas "get_action" has to sum every one of L^2 potential terms and the corresponding $4L^2$ derivative terms. In addition, the action has to be recalculated at the end of every change of $\Delta\alpha$ when calculating the estimators for the mass, since the action changes due to the change in the potential when α changes. For this, unfortunately, one must resort to the more time-consuming method, since there is no previous action which could be used by the faster function.

In the Metropolis algorithm one has to compare the new and old actions. In it's heart there is the acceptance ratio. Metropolis updates the field using the functions described above, gets the new action, and then compares it to the old one. If the action is lower, then it is accepted with no conditions. But if the newly generated action is larger than the old one, it gets accepted randomly with the probability

$$P = e^{T(S_E[\phi_{tx}] - S_E[\phi_{tx}]_{\text{upd.}})}. \tag{4.3}$$

The randomness gets realized by using a random number generator for C/C++ "ranlxd" by Martin Luescher. Note how if the updated action is much higher than the old one,

P is very small and the new field has a very low chance of being accepted. In contrast, if the updated action is only slightly larger than the previous one, then the exponential is also large, and the chance of acceptance is much higher. This allows for a "walk" around the lowest state, but not too far away. In fact, the acceptance ratio of a new state corresponds to its Boltzmann weight.

Action calculation functions also use the periodic boundary conditions in the t direction, and twisted boundary conditions in the x direction. This makes the field effectively periodic and infinite, as discussed in section (2.3). Twisting the boundary conditions to make the algorithm generate kink states using Metropolis is not that straightforward. In this case one has to calculate the comparing action differently. Note that condition (2.26) only applies to the derivatives, not the cosine potential term, since it is the field difference between two adjacent lattice cells. In fact, the twist only applies if the updated cell is ϕ_{t1} or ϕ_{tL}, since "update" has to recalculate all four derivatives around the point. To make the algorithm generate one-kink states, one has to set ΔQ to ± 1, as opposed to just the flat periodic case, where $\Delta Q = 0$. When the twisted action is calculated, it is checked against the previous action. The same apply - reject method is applied as for the periodic case, but if the state is accepted, then the action that is accepted has to be recalculated without the twist. The anti-periodic action is just used for the acceptance ratio shift, so Metropolis generates the kink state.

In order to make mass calculations, one has to run the simulation for both the periodic and twisted cases. The next steps in the function are bits that determine if the system has equilibrated or thermalized. This is done by letting the simulation run without changing anything for some time, which is determined by the experimenter before the simulation starts. Once the system has equilibrated, "metropolis" calculates the estimators for the naive method, (2.30) and the finite differences method (2.32).

The latter method can be expanded and error propagation can be more or less controlled. The program is written in such a way that the simulation takes a value of α and does a calculation of the expectation values for the mass over a certain number of sweeps on the lattice. Once the calculations with the value are done, α is slightly incremented, thermalization for the system starts, and once it is finished, calculations for the expectation values for other value of α start, and so on. Equation (2.32) provides a way to reduce the statistical error by increasing the number of increments taken between the first and last values of α. Since the mass difference depends on the difference of the partition function as a function of the parameter α, one could notice that [15, 14, 13]

$$\left\langle \exp\left(\frac{(\alpha_2-\alpha_1)a^2}{\beta^2}\sum_{tx}(\cos\beta\phi-1)\right)\right\rangle_1 = -\left\langle \exp\left(\frac{(\alpha_1-\alpha_2)a^2}{\beta^2}\sum_{tx}(\cos\beta\phi-1)\right)\right\rangle_2 . \tag{4.4}$$

That is, the change in the expectation value as the change in α measured at α_1 is minus the change in the expectation value as the negative change in α measured at α_2. This means that the change in mass can be measured in two directions. The two values can be then averaged to give a more precise measurement. The errors then also depend on the difference of those two measured values. If one now defines

$$f_1 = -\frac{1}{T}\ln\left\langle \exp\left(\frac{(\alpha_2-\alpha_1)a^2}{\beta^2}\sum_{tx}(\cos\beta\phi-1)\right)\right\rangle_1 \tag{4.5}$$

and

$$f_2 = \frac{1}{T}\ln\left\langle \exp\left(\frac{(\alpha_1-\alpha_2)a^2}{\beta^2}\sum_{tx}(\cos\beta\phi-1)\right)\right\rangle_2 , \tag{4.6}$$

the change in kink mass can be written as

$$M_{\text{kink}}(\alpha_2) - M_{\text{kink}}(\alpha_1) = \frac{1}{2}\left(f_1^{tw} + f_2^{tw} - f_1^{p} - f_2^{p}\right). \tag{4.7}$$

The function measures the estimators for the exponentials in $f_{1,2}$ and outputs them into a file which can then be processed by the other program "mass.cpp". "Metropolis" breaks the perpetual loop calculations and outputs the settings file which contains the information fo the calculations once all of the values for every increment of α are calculated and output into separate files.

"Mass.cpp" is a program that measures the mass from the files that are output by "simulation.cpp". It's first task is to load the settings that come with all the data the first program produces. The main function has to call the "expectation" functions for the finite differences and the naive methods to get the expectation value and error increments which it can then integrate. These functions compute expectation values from their estimators in the data files. "Simulation.cpp" is written in such a way as to take measurements once a sweep, so "mass.cpp" has to process an experimenter's defined amount of files with a certain amount of data entries in them.

The expectation functions also use the correlator method (section (3.7)) to compute autocorrelation times of the data sets, which they then use to adjust the error they return. In general, for bigger lattice sizes the autocorrelation time is not that big, since the measurements are taken once a sweep, which is 4098 for $L = 64$. A Fast Fourier Transform library is used to perform the Fourier transforms in order to deduce the autocorrelation times for the data sets. For the finite difference method the program uses the "MPFR" high precision library. This is needed, because sometimes, especially for low values of α, the numbers in (4.4) are very high, since the kink is wide and doesn't it in the lattice, and the L^2 sum gives numbers that are too high for the usual C++ data types to handle, and the program thinks that they are infinite, which does not provide a lot of information.

Once the calculations are done, the results are output in a format which is good for depicting the mass curves in gnuplot.

4.2 Error Analysis

Autocorrelation time is used to either choose the estimators for the expectation values once every τ, or choose every estimator value, but include τ in the error calculation. The data for errors is also measured in "simulation.cpp". Those measurements are being done parallelly to the mass estimator measurements and are being output into the same files, so that then "mass.cpp" could analyze everything at once. The error for every value of

(2.30), since it it is a sum of two expectation values, is [19]

$$\Delta M_{\text{naive}} = \sqrt{\left(\sigma^2 \frac{\tau}{N}\right)_1 + \left(\sigma^2 \frac{\tau}{N}\right)_2 + \frac{\alpha(\Delta\alpha)^4}{180} \max_{\xi\in[0,\alpha]} |M^4_{\text{kink}}(\xi)|}, \tag{4.8}$$

where the last term is due to the Simpson's method of integration, σ is the usual standard deviation, and N is the number of measurements. The error for (4.7) can be computed in a different way [15, 14, 13]. Since it is a logarithm, the error is

$$\Delta M_{\text{f.d.}} = \sqrt{\frac{1}{2T} \left(\Delta f^2_{1,\,\text{tw}} + \Delta f^2_{2,\,\text{tw}} + \Delta f^2_{1,\,\text{p}} + \Delta f^2_{2,\,\text{p}} + (f_{1,\,\text{p}} - f_{2,\,\text{p}})^2 + (f_{1,\,\text{tw}} - f_{2,\,\text{tw}})^2\right)}. \tag{4.9}$$

The last two terms result for the difference (4.5). The errors for f are

$$\Delta f = \frac{\Delta \left\langle \exp\left(\frac{\Delta\alpha a^2}{\beta^2} \sum_{tx} (\cos\beta\phi - 1)\right)\right\rangle}{\left\langle \exp\left(\frac{\Delta\alpha a^2}{\beta^2} \sum_{tx} (\cos\beta\phi - 1)\right)\right\rangle}, \tag{4.10}$$

since it is a logarithm, and where $\Delta\alpha$ is not an error, but rather the corresponding difference of the parameter in the action. The error increments can then be summed up for every value to get the total error.

5 Results and their discussion

A number of simulations were performed for various values of β to compute the quantum kink mass dependence on α. At least 501 increments of α were computed for the graphs and for each increment $\Delta\alpha$ at least 100 measurements were taken. These numbers were enough to reduce the errors to considerable values. The measurements were done on lattices of size 16^2 and 64^2 from $\alpha = [0, 3]$ for purely numerical calculation (although the results in this case are still inaccurate due to effects discussed shortly). The time taken to equilibrate the system before running the simulations, after testing, was set to 10000 sweeps and between every change of α it was set to 250 sweeps again by testing.

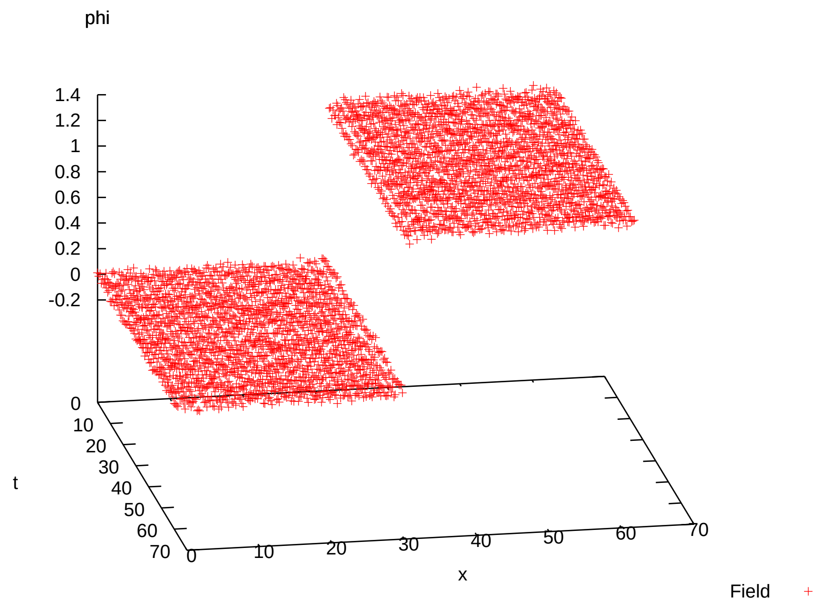

Figure 1: Narrow Sine-Gordon kink with $\beta = \sqrt{8\pi}$ and $\alpha = 25.117$ on a lattice $L = 64$.

Rejection rate of the Metropolis algorithm was controlled dynamically by increasing or decreasing the field update size in order to keep it around 40%.

Lattice spacing a was set to 0.8. The reason for that is due to the effect that at large α kinks are very narrow. This means that the lattice can be not discreet enough in order for the kink to be fully "seen", as depicted in (fig. 1). High α effectively turns the kink into $\alpha = \infty$ on the lattice and the mass increments that one gets in (2.32) and (2.30) by incrementing the parameter limits to 0 as the parameter increases. We then witness a saturation of the measured mass and the measured and analytical masses start to diverge. The fact that the continuum limit is unreachable therefore means that the mass of the quantum kink cannot be computed for every value of α correctly. This can be more or less avoided, however, and for a convenient $a = 0.8$ the mass can still be measured quite accurately for up to about $\alpha = 2$. This discussion, however, does not set the lower bound for a and if one decreases it, then the "lattice is too small" effect (fig. 2) kicks in, as described in section (2.3). For small α the kinks are too wide for the lattice they are on, and they don't fit in. Equation (2.28) ceases to work here, as the free energy on that lattice is no longer the full free energy of the kink. The same happens for a change in the action.

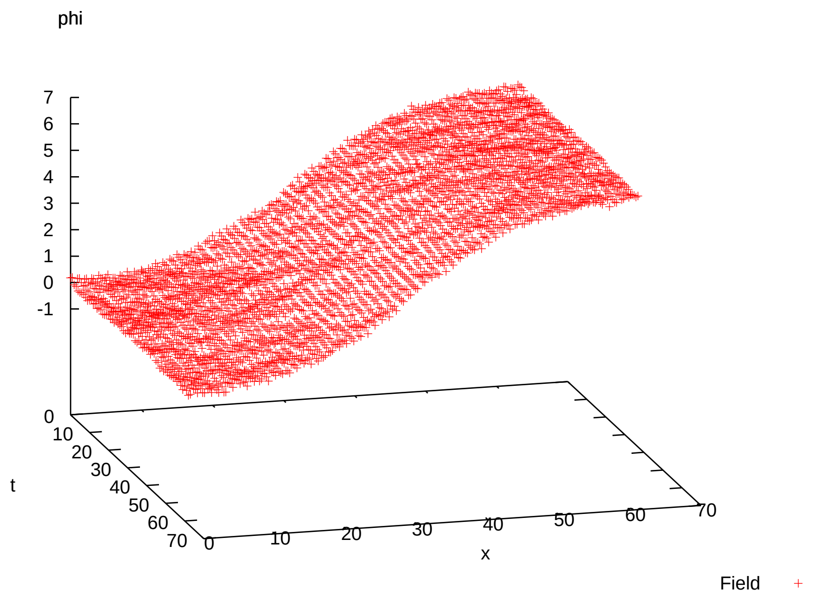

Figure 2: Wide Sine-Gordon kink with $\beta = 1$ and $\alpha = 0.013$ on a lattice $L = 64$.

This discrepancy introduces a large offset in the calculation of mass of the kink and at the beginning of the trend mass measurement cannot be interpreted as a good measurement. Since the mass is measured by integrating every value, this effect introduces a constant difference between the analytically and numerically measured masses throughout every α. This can also be avoided by increasing the size of the lattice and decreasing the number of increments of α, so that the measurement at $\alpha = \Delta\alpha$ is good, and the number of increments is large enough for the error to be not too large. Increasing lattice size, however, increases calculation times exponentially and is inconvenient. Another way to avoid this problem of small α is to start the integration at another point than zero. $\alpha = 0$ is a free scalar field theory and topological defects are impossible there, since there is no potential. We know that the mass of the kink is zero in that limit and that's why we start the integration from there. For results that are not obtained purely numerically one could rely on the analytical expression of mass (2.10) and start the integration from a point that is not 0, say, 0.2, in order to avoid the wide kink problem, although this would not be possible for a theory where the analytic expression of the parameter that is being measured is not known.

Actually, there is another limit. The original Coleman's paper [3] explains that for $\beta^2 > 8\pi$ the theory has no lower bound for energy and that the kink mass falls to zero. It would be nice to test this idea, but there is another problem, the high β problem. In this limit, the relative kink mass changes, since of the term α/β^2 in the potential (2.2), although only slightly, since only the first term in the expansion (2.3) does not depend on β. In particular, $\beta = \sqrt{8\pi}$ gives rise to a small lattice effect. For this reason, it is really tricky to compute the zero kink mass limit.

With $a = 0.8$ we make a series of measurements for $\beta \in [0.2, 1]$ on a 16^2 lattice and also make the same series of measurements on 64^2 starting at $\alpha = 0$ and make comparisons with the corresponding analytical expressions.

The 16^2 measurements were quite surprising, since the results were actually quite accurate, even if the field at a point of time is expressed over only 16 points. As we can see in figures (8 - 12), the trends for the results agree up to $\alpha = 1$, after which point they start to diverge from the trends analytical expressions. The error for the naive calculation is large, and it actually gets larger if one increases the number of measurements, while the error for the finite difference method is small and gets smaller with increase of number of measurements, as discussed in section (4.2). Despite that, and the flawed integration due to the wide kink effect, the two results agree nicely throughout the plots.

The 64^2 measurements take longer, but produce more accurate results by reducing both small kink calculation discrepancies and finite lattice size effect. The results start to diverge at $\alpha = 1$. The slight increase of values with respect to the analytical result in the interval $\alpha \in [0, 2]$ is due to a finite number of increments $\Delta\alpha$ and can be abolished by performing longer computations. As we can see, the point at which the two lines start to diverge increases with the increase of β, which is expected, since β changes the relative kink mass and keeps the kink wider, so it can be properly measured for slightly longer intervals of α.

The following is a collection of obtained results for various β on a $L = 64$ and 16.

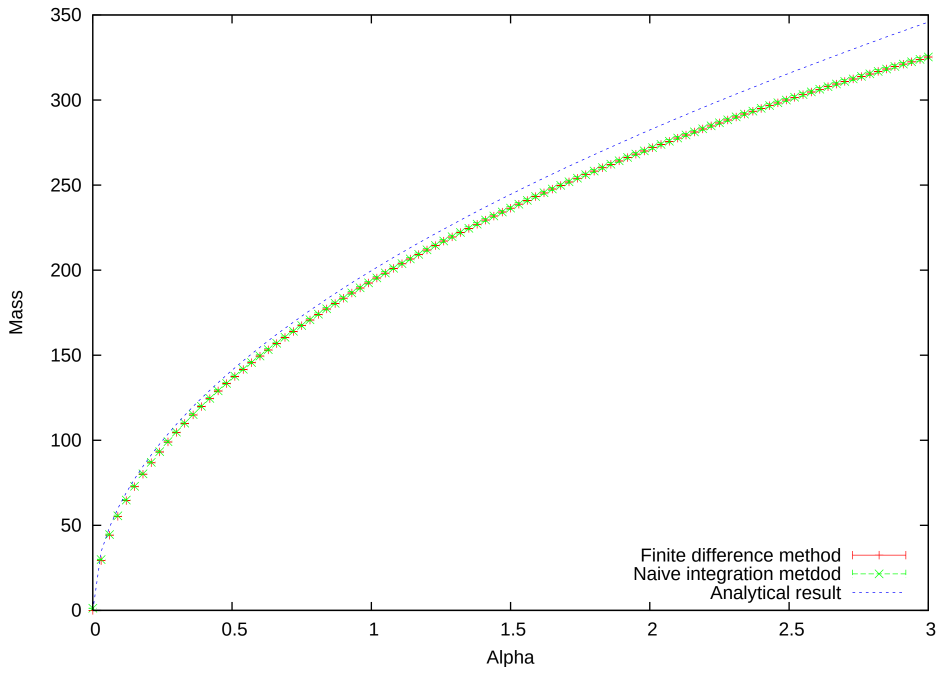

Figure 3: Sine-Gordon kink mass measured at $\beta = 0.2$ on a lattice $L = 64$.

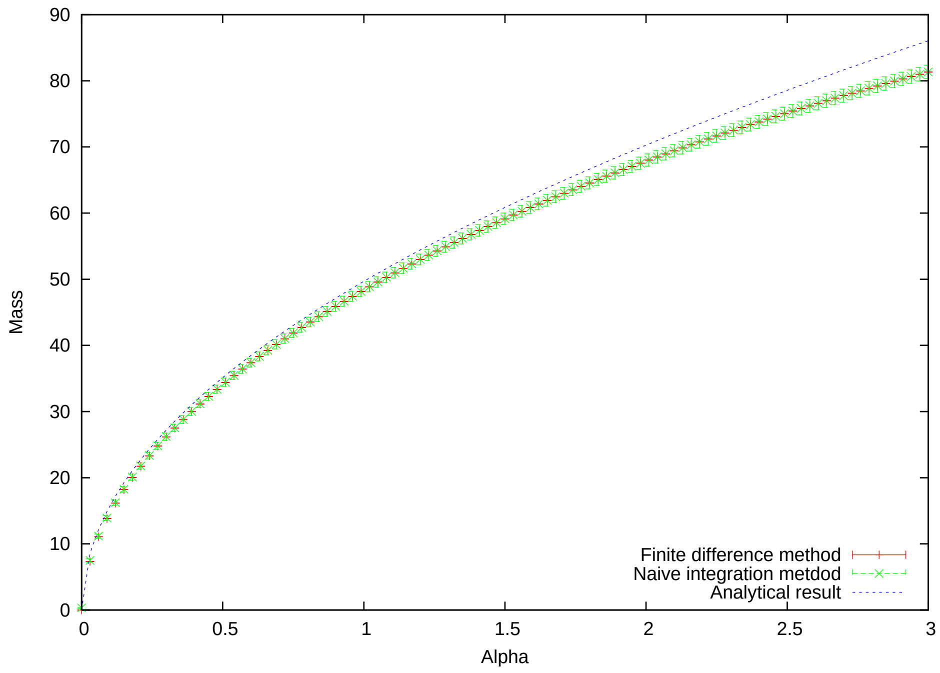

Figure 4: Sine-Gordon kink mass measured at $\beta = 0.4$ on a lattice $L = 64$.

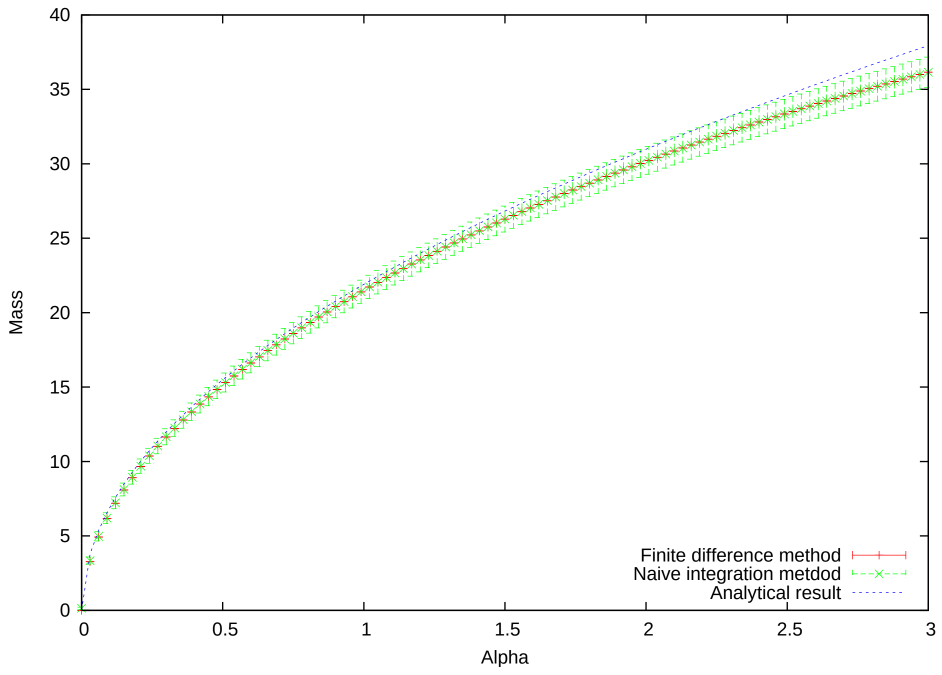

Figure 5: Sine-Gordon kink mass measured at $\beta = 0.6$ on a lattice $L = 64$.

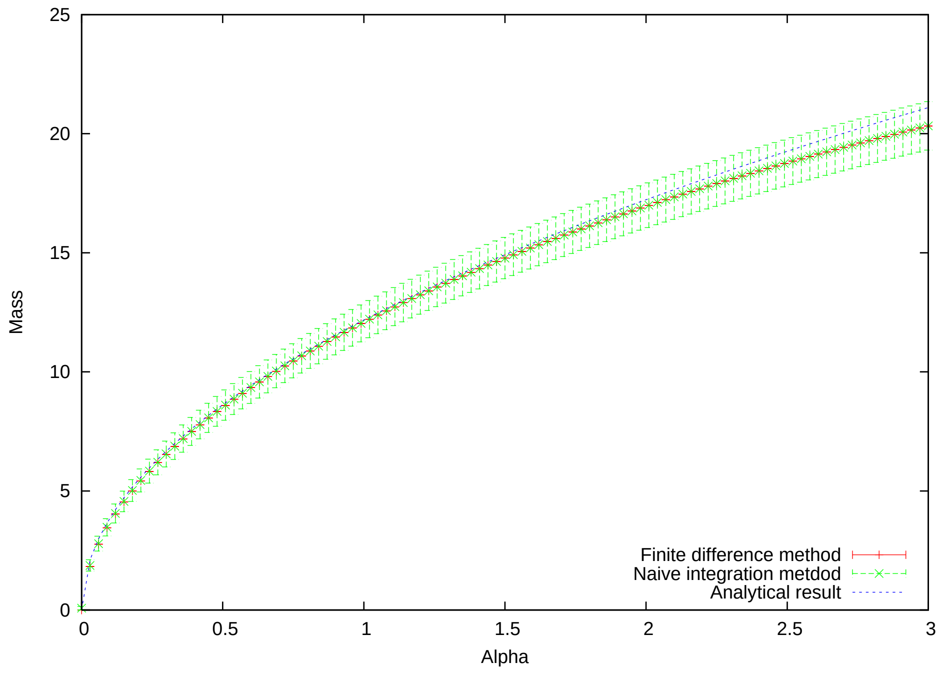

Figure 6: Sine-Gordon kink mass measured at $\beta = 0.8$ on a lattice $L = 64$.

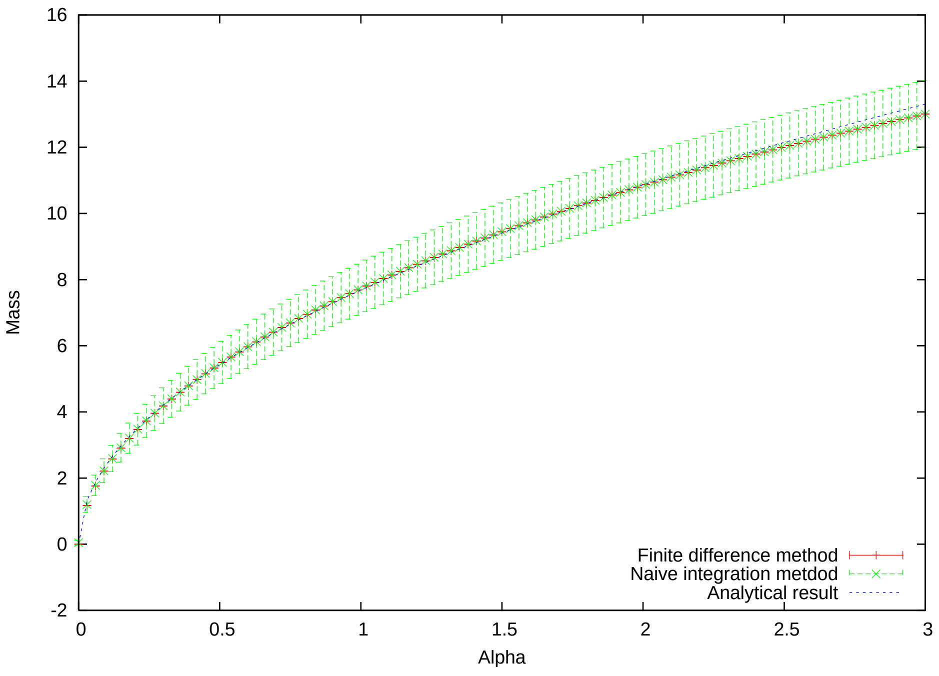

Figure 7: Sine-Gordon kink mass measured at $\beta = 1$ on a lattice $L = 64$.

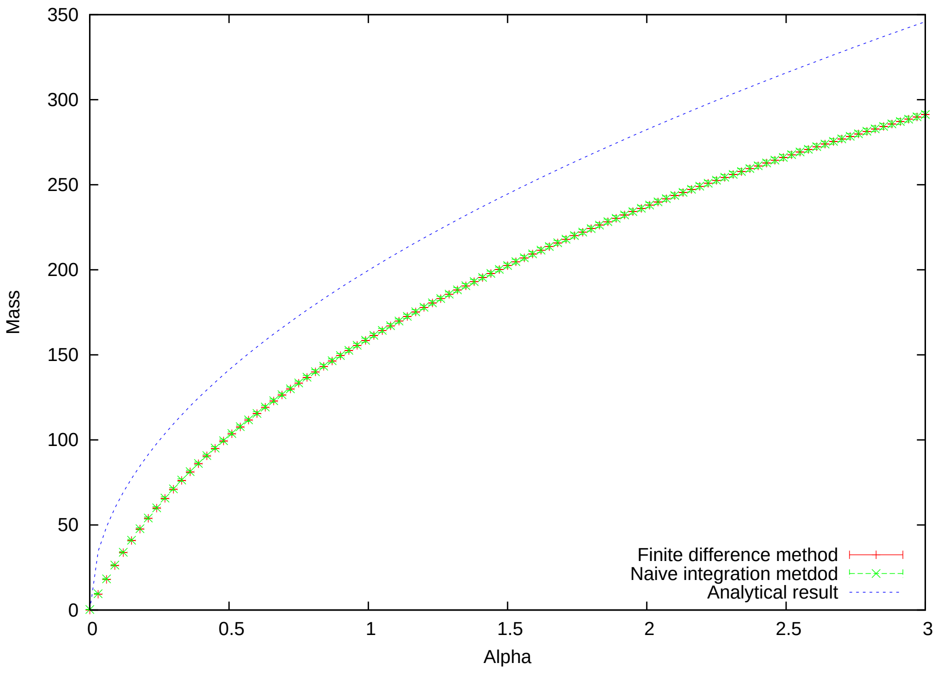

Figure 8: Sine-Gordon kink mass measured at $\beta = 0.2$ on a lattice $L = 16$.

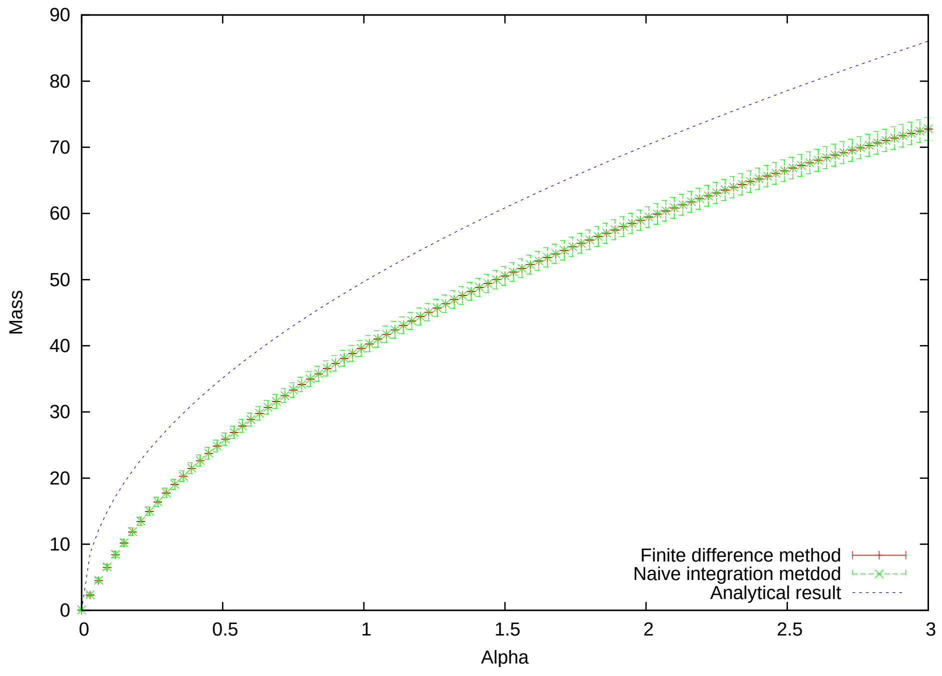

Figure 9: Sine-Gordon kink mass measured at $\beta = 0.4$ on a lattice $L = 16$.

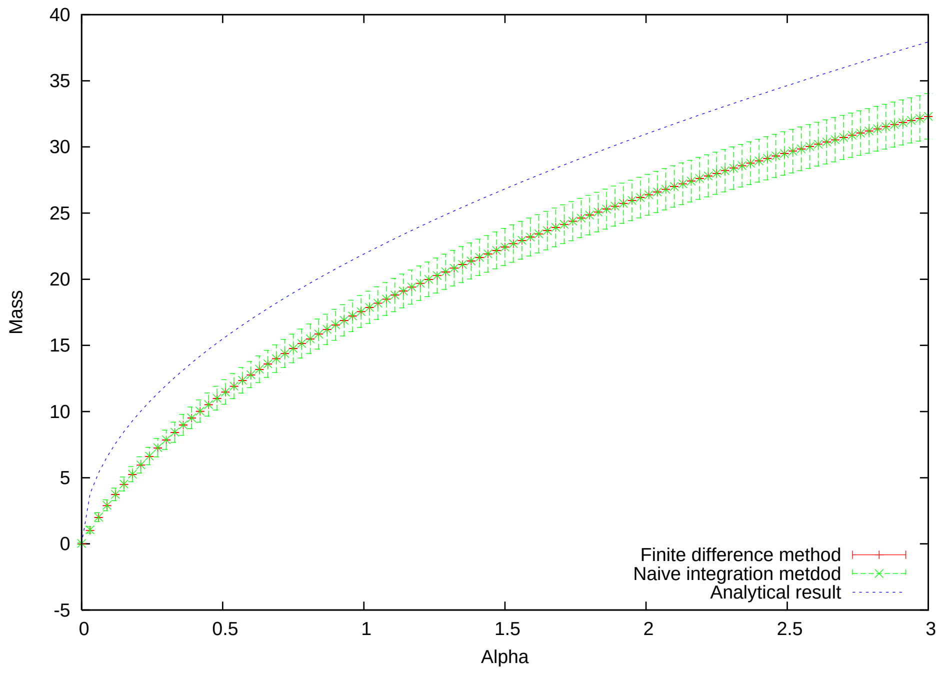

Figure 10: Sine-Gordon kink mass measured at $\beta = 0.6$ on a lattice $L = 16$.

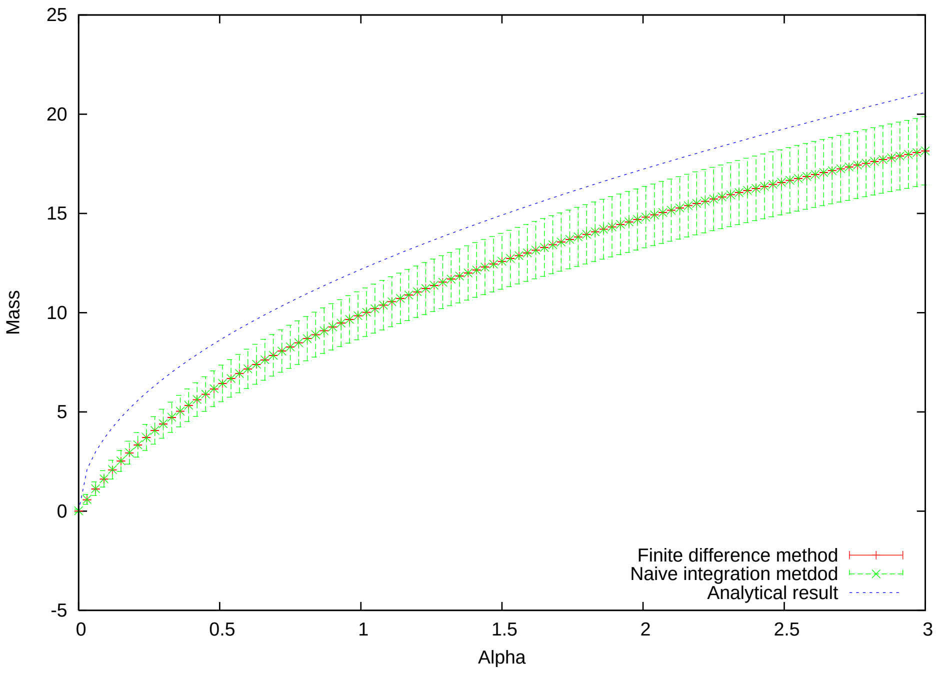

Figure 11: Sine-Gordon kink mass measured at $\beta = 0.8$ on a lattice $L = 16$.

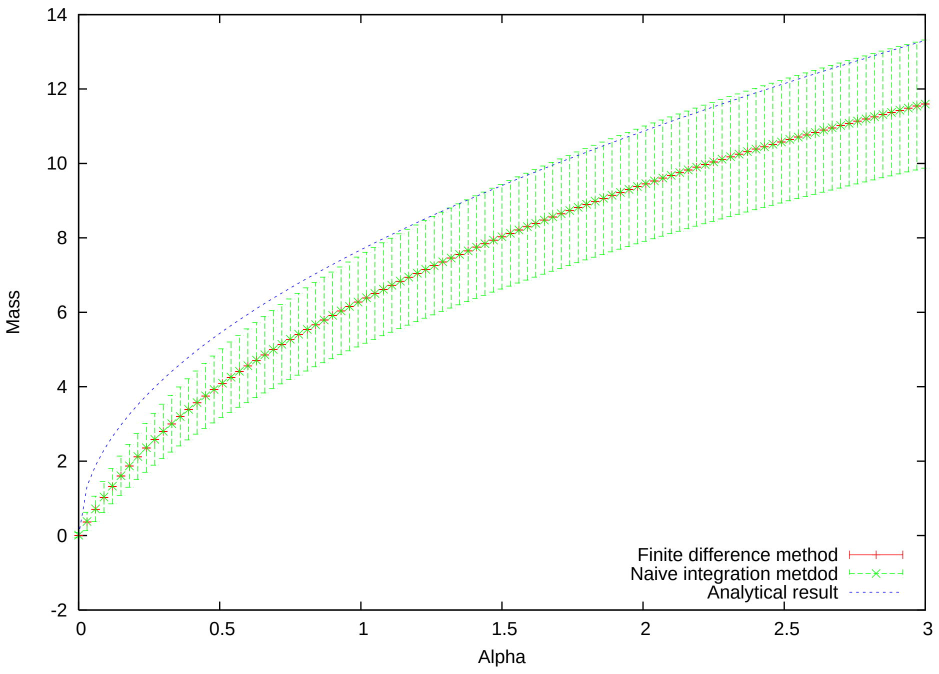

Figure 12: Sine-Gordon kink mass measured at $\beta = 1$ on a lattice $L = 16$.

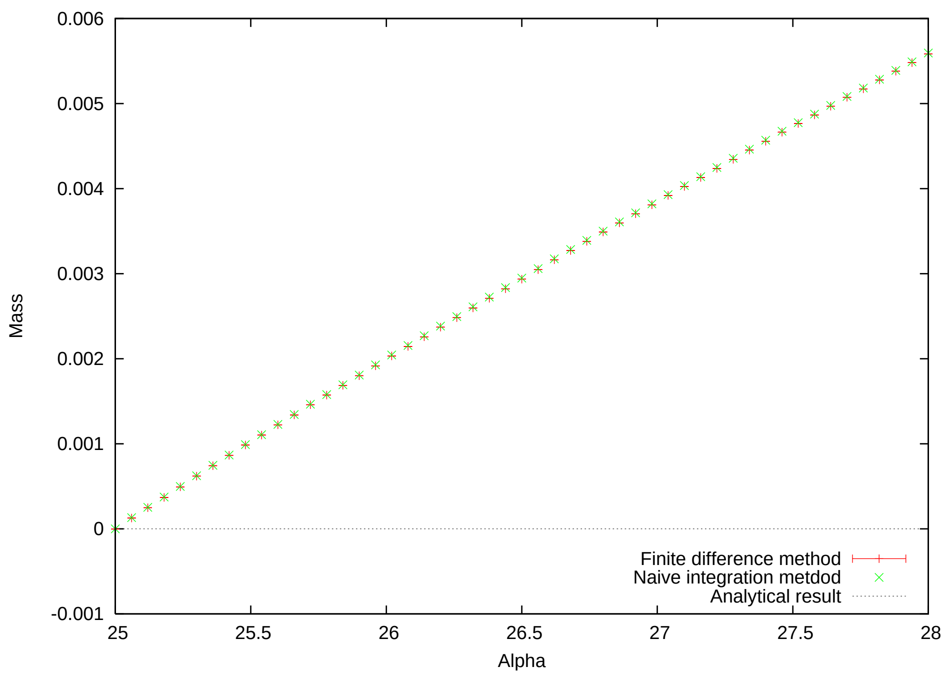

Figure 13: Sine-Gordon kink mass measured at $\beta = \sqrt{(8\pi)}$ and $\alpha \in [25, 28]$ on a lattice $L = 64$, where the integration was started at the analytical $M_{\text{kink}} = 0$.

An attempt at measuring the zero kink mass limit is worth mentioning as well. Since $\beta = \sqrt{(8\pi)}$ is about 25, we try starting the computation at $\alpha = 25$ to be as close as possible to β^2, and use the analytical expression, which is 0, to start the integration from, so we can at least see the trend. Unfortunately, for high β, the interpretable α region increases only very slightly slightly, even if we set $\beta^2 = \alpha$, for the reasons discussed earlier. Also, the results should not agree due to the fact that the analytical mass (2.10) is given up to the order β^2, in other words - the weak coupling approximation fails. The graph of that measurement is given in (fig. 13), and cannot be interpreted as a good result.

6 Conclusion

It should be emphasized that the analytical result for quantum sine-Gordon kink mass was confirmed numerically. The differences between the two results were understood and explained carefully enough to understand which bits can be interpreted as good results, and which ones as bad. Although there were not many types of simulations performed (and not much, other than the mass trend, expected), the main technique for non-perturbative calculations was developed and used successfully. Application of the Metropolis algorithm was understood and performed well. One of the most important things, I think, is that the programs written can be further developed to include more or different measurements and used to simulate other theories as well.

The mass of a sine-Gordon kink, in principle, can also be measured by using the correlation function measurements and then performing a spectral expansion with conjunction to determining that only the first term contributes to the mass of the kink. The correlation functions could then be fitted to find the mass values. This method could also be applied to the written programs quite easily and it would be encouraging to compare the obtained results.

Acknowledgements

I would like to thank my supervisor Dr. Arttu Rajantie and the very helpful Dr. David Weir at Imperial College London who guided me throughout my project. I would also like to thank my colleague students in the Quantum Fields and Fundamental Forces course, 2010/2011, at the Theoretical Physics Group at Imperial College London for useful discussions and insights on the problems I had to solve during my project, and the the college security as well, who gave me night access to the Blackett Laboratory.

A Simulation.cpp

This is the code that was written in C++ to perform the simulations.

```
/*-------------------------------------------------------------------------------------------
MONTE CARLO SIMULATION OF THE SINE-GORDON QUANTUM FIELD THEORY

AUTHOR: LINAS BITKEVICIUS
IMPERIAL COLLEGE LONDON
DEPARTMENT OF THEORETICAL PHYSICS

Settings can be changed in the "Settings" file. The format is as follows:

[lattice size]
[a]
[beta]
```

```
[step size]    (the formula is (random[0;1) - 0.5)/step_size)
[number of measurements per value (N)]
[equilibrium/(L*L)]    (before taking measurements - equilibriate the system)
[last_alpha]
[number of values to be measured]    (must be odd number for simpson integration to work well in mass.
    cpp)
[thermalization/(L*L)]    (between every measurement)
[classical?] = 1 is classical, 0 is quantum

-----------------------------------------------------------------------------------------------------*/
#include <iostream>
#include <cmath>
#include <fstream>
#include <cstdio>
#include <cstring>
#include "ranlxd.h"

using namespace std;

void montecarlo(double *field, double action);
double get_action(double *field);
double update(double *field, double action, int x, double *naujas, double M);

double PI = 3.141592;

//field + lattice
double *field;
int L = 64;
//action params
double a = 0.8;
double starting_point = 0;
double alpha = starting_point;
double beta = 1;
//just the usual constatnt here
double con = a*a/(beta*beta);
//increment
double delta_alpha = 0.01;
 //highest value of alpha
double last_alpha = 1;
double inc;

double step = 0.5;//usually 0.5

//make N measurements with one delta alpha value and then change to another delta alpha (if
    sample_size = L*L*N)
int N = 100;
long int sample_size = L*L*N; //USUALLY L*L*N

//if T = thermalisation, then we can continue with lattice writing
int thermalisation = L*L;
int T = 0;

//number of update sweeps to reach the quilibrium USUALLY 10000
long int equilibrium = L*L*10000;
long int E = 0;

//multipliers in front of PI/beta in the boundary derivatives contributing to the action. 0 -
    untwisted. Any other - twisted B.C's
//period of the theory: M1 = M2 = +-2
double M1 = 0;
double M2 = 0;

//rejection rate calculation
double total = 0;
double rejected = 0;

bool classical;
double untwisted_action;
double Mcl = 0;

//constantly outputs the field
ofstream out_field;
//expectation value streams. I don't close them, since I want to constantly update the file.
ofstream phout; //higher one out
ofstream phsqout; //lower one out
ofstream nout; //naive exp. values out
//lattice loading stream
ifstream fin;
//settings stream
ifstream setin;
ofstream setout;
//classical mass stream
ofstream massout;

int main()
{
  int point = 0;
  double z; //dumps the x and t variables from the results file

  //////////////////SETTINGS//////////////////
  cout << "Loading_settings..." << endl;
  setin.open("Settings");
  setin >> L >> a >> beta >> step >> N >> equilibrium >> last_alpha >> delta_alpha >> thermalisation
      >> classical;
  setin.close();
  cout << "Lattice␣size␣is␣=␣" << L << ";␣a␣=␣" << a << endl;
  cout << "beta␣=␣" << beta << endl;
  cout << "N␣=␣" << N << endl;
  cout << "highest␣alpha␣=␣" << last_alpha << endl;
  cout << "Number␣of␣values␣that␣will␣be␣written␣=␣" << delta_alpha << ";␣delta_alpha,␣or␣increment␣
      size␣is␣=␣" << (last_alpha - alpha)/(delta_alpha - 1) << endl;
  cout << "step␣=␣" << step << endl;
  cout << "thermalisation␣=␣" << thermalisation << "*" << L << "*" << L << endl;
  cout << "sample_size␣=␣" << N << "*" << L << "*" << L << endl;
  cout << "equilibrium␣=␣" << equilibrium << "*" << L << "*" << L << endl;
  thermalisation *= L*L;
  inc = delta_alpha;
  delta_alpha = (last_alpha - alpha)/(delta_alpha - 1);
  sample_size = L*L*N;
  equilibrium *= L*L;
  con = a*a/(beta*beta);
  cout << "Enter␣the␣twist␣parameter.␣+-2␣for␣twisted␣boundary␣conditions,␣0␣for␣periodic.␣";
  cin >> M1;
  M2 = M1;
  if (M1 != 0 && classical == true)
    {
      massout.open("results/CMASS");
    }
  ///////////////////////////////////////////////


  //load the lattice or generate a random field
  field = new double [L*L];
  fin.open("usually␣here␣is␣the␣file␣name");
  if (!fin.fail())
    {
      cout << "Lattice␣loaded␣succesfully!" << endl;
      for (int i = 0; i < L; i++)
        {
          for (int j = 0; j < L; j++)
            {
              //write field to file
              fin >> z >> z >> *(field + point);
              point++;
            }
        }
      fin.close();
    }
  else
    {
      cout << "Loading␣the␣lattice␣failed,␣generating␣a␣new␣random␣lattice." << endl;
      ranlxd(field, L*L);
      for (int i = 0; i < (L*L); i++)
        {
              *(field + i) -= 0.5;
              *(field + i) = 2*PI*(i%L)/(beta*(L-1)) - PI/beta;
              /*              if (i%L < L/4)
                field[i] = 0;
              else if (i%L > 3*L/4)
                field[i] = 0;
              else
              field[i] = 2*PI/beta;*/
              if (i%L < L/2)
                field[i] = 0;
              else
              field[i] = 2*PI/beta;
        }
    }
  //run montecarlo
  montecarlo(field, get_action(field));
  return 0;
}



//does checkerboard and comparison
void montecarlo(double *field, double starting_action)
{
  //monte carlo variables
  int point = 0;
  double action;
  double updated_action;
  int written = 0;
  double *random;
  random = new double;
  double *naujas;
  naujas = new double;

  //expectation value measurement variables
  double evplus = 0; //constant minus sum of cosines
  double evminus = 0;
  double nevsq = 0; //expectation values of squares
  double evsq = 0;
  char fnameplus[50]; //the whole file name
  char fnameminus[50];
  char fnamenaive[50];
  int blabla; //holds the value of sprintf

  //start function
```

```
//----------------set up the file name and streams------------------
if (M1 != 0 && M2 != 0)
  {
    blabla = sprintf(fnameplus, "results/t_u_%f", alpha);
    blabla = sprintf(fnameminus, "results/t_d_%f", alpha);
    blabla = sprintf(fnamenaive, "results/t_n_%f", alpha);
  }
else
  {
    blabla = sprintf(fnameplus, "results/p_u_%f", alpha);
    blabla = sprintf(fnameminus, "results/p_d_%f", alpha);
    blabla = sprintf(fnamenaive, "results/p_n_%f", alpha);
  }
phout.open(fnameplus); //writes out evplus
nout.open(fnamenaive);

//-----------------------------main for loop of the monte carlo algorithm--------------------------
action = starting_action;
for (int x = 0; x < L*L; x = x + 2)
  {

    ranlxd(random, 1);
    updated_action = update(field, action, x, naujas, M1);
    if(updated_action <= action)
      {
        *(field + x) = *naujas;
        action = untwisted_action;
        total++;
      }
    else if (*random < exp(L*(action - updated_action)) && classical == false)
      {
        *(field + x) = *naujas;
        action = untwisted_action;
        total++;
      }
    else
      {
        total++;
        rejected++;
      }

    //-------------------------expectation value computation and writing-----------------------

    //equilibriation of the system
    if (E < equilibrium)
      {
        E++;
        if (E%(L*L*100) == 0)
          {
            cout << "Equilibriating field. " << 100*E/equilibrium << "% done" << endl;

            //MECHANISM TO WRITE THE FIELD TO A FILE SO IT CAN BE VIEWED
            if (M1 == 0)
              {
                out_field.open("resp");
              }
            else
              {
                out_field.open("restw");
              }
            for (int i = 0; i < L; i++)
              {
                for (int j = 0; j < L; j++)
                  {
                    out_field << i << " " << j << " " << *(field + point) << endl;
                    point++;
                  }
              }
            point = 0;
            out_field.close();
          }
        if (E == equilibrium)
          {
            cout << "Starting expectation value computation." << endl;
          }
      }

    //implements thermalisation time
    if (T == thermalisation && E == equilibrium)
      {
        written++;
      }
    else if (T != thermalisation)
      {
        T++;
      }

    //every full lattice run compute the value in the
    if (x == L*L - 2 && E == equilibrium && T == thermalisation)
      //if (E == equilibrium && T == thermalisation)
      {
        for (int i = 0; i < L*L; i++)
          {
            evplus += cos(beta * field[i]);
            evminus += cos(beta * field[i]);
            nevsq += pow(cos(beta * field[i]), 2);
            evsq += pow(delta_alpha*(1/(beta*beta))*a*(cos(beta * field[i]) - 1), 2);
          }
        nout << evplus/(L*L) << " " << nevsq/(L*L) << endl;
        if (alpha != last_alpha)
          {
            phout << delta_alpha*a*(evplus - L*L)/(beta*beta) << " " << evsq << endl;
          }
        if (alpha != 0)
          {
            phsqout << delta_alpha*a*((L*L) - evminus)/(beta*beta) << " " << evsq << endl;
          }
        evplus = 0;
        evminus = 0;
        nevsq = 0;
        evsq = 0;
      }
    //end expectation value computation

    //field file writing
    if (written == static_cast<int>(sample_size))
      {
        //MECHANISM TO WRITE THE FIELD TO A FILE SO IT CAN BE VIEWED
        if (M1 == 0)
          {
            out_field.open("resp");
          }
        else
          {
            out_field.open("restw");
          }
        for (int i = 0; i < L; i++)
          {
            for (int j = 0; j < L; j++)
              {
                out_field << i << " " << j << " " << *(field + point) << endl;
                point++;
              }
          }
          out_field.close();

        if (E == equilibrium)
          {
            //close phout stream, change alpha, change fname, open new phout stream
            phout.close();
            phsqout.close();
            nout.close();

            //automatic step size adjustment. Want rejection rate to be exactly 40%
            if (rejected/total < 0.395)
              {
                step = step - 0.05;
              }
            if (rejected/total > 0.405)
              {
                step = step + 0.05;
              }

            //write out the classical mass
            if (M1 != 0 && classical == true)
              {
                massout << alpha << " " << Mcl << endl;
                Mcl += action/T;
              }

            cout << 100*(alpha - starting_point)/(last_alpha + delta_alpha - starting_point) << "% 
                done. Computed alpha " << alpha << ". Rejection rate is " << rejected*100/total << "
                %, step is " << step << "..." << endl;

            //check if program has to end
            if (alpha > last_alpha -0.0001 && alpha < last_alpha + 0.0001)
              {
                setout.open("results/Settings");
                setout << L << endl;
                setout << a << endl;
                setout << beta << endl;
                setout << N << endl;
                setout << last_alpha << endl;
                setout << inc << endl;
                setout.close();
                cout << "Computation finished. Exiting program now." << endl;
                exit(1);
              }
            alpha += delta_alpha;
            if (M1 != 0 && M2 != 0)
              {
                blabla = sprintf(fnameplus, "results/t_u_%f", alpha);
                blabla = sprintf(fnameminus, "results/t_d_%f", alpha);
                blabla = sprintf(fnamenaive, "results/t_n_%f", alpha);
              }
            else
              {
                blabla = sprintf(fnameplus, "results/p_u_%f", alpha);
                blabla = sprintf(fnameminus, "results/p_d_%f", alpha);
                blabla = sprintf(fnamenaive, "results/p_n_%f", alpha);
```

```
                }
              if (alpha != last_alpha && T == thermalisation)
                {
                  phout.open(fnameplus);
                }
              if (T == thermalisation)
                {
                  phsqout.open(fnameminus);
                  nout.open(fnamenaive);
                }
            }
          written = 0;
          T = 0;
          rejected = 0;
          total = 0;
          point = 0;
          action = get_action(field);
        }



      ///////////////////////////////////////////////////////////
      //for even lattices/implementation of checkerboard update
      if (L%2 == 0 && x!= L*L - 1 && x != L*L - 2)
        {
          //if x is before the last one in the row, push it by one
          if (x%L == L - 2)
            {
              x++;
            }
          //else if it is the last one in the row, push it back
          else if (x%L == L - 1)
            {
              x--;
            }
        }
      //infinite for mechanism
      if (x == (L*L - 2))
        {
          x = -2;
        }
      if (x == (L*L - 1))
        {
          x = -1;
        }
    }
}


//updates the action and returns the updated action and also changes the new field value
double update(double *field, double action, int x, double *naujas, double M)
{
  double xdl;
  double xdr;
  double tdd;
  double tdu;

  double xdlzero;
  double xdrzero;

  //generate the new value of the field at x
  ranlxd(naujas, 1);
  *naujas = (*naujas - 0.5)/step + *(field + x);

  //x derivatives
  //points on the left of the lattice
  if (x%L == 0)
    {
      xdl = pow((*naujas - *(field + x - 1 + L) + M*PI/beta) ,2) - pow((*(field + x) - *(field + x - 1
           + L)+ M*PI/beta) ,2);
      xdr = pow((*(field + x + 1) - *naujas) ,2) - pow((field[x+1] - *(field + x)) ,2);
      xdlzero = pow((*naujas - *(field + x - 1 + L)) ,2) - pow((*(field + x) - *(field + x - 1 + L))
           ,2);
      xdrzero = pow((*(field + x + 1) - *naujas) ,2) - pow((field[x+1] - *(field + x)) ,2);
    }
  //points on the right
  else if ((x - L + 1)%L == 0)
    {
      xdr = pow((*(field + x + 1 - L) - *naujas + M*PI/beta) ,2) - pow((*(field + x + 1 - L) - *(field
           + x) + M*PI/beta) ,2);
      xdl = pow((*naujas - *(field + x - 1)) ,2) - pow((*(field + x) - *(field + x - 1)) ,2);
      xdrzero = pow((*(field + x + 1 - L) - *naujas) ,2) - pow((*(field + x + 1 - L) - *(field + x))
           ,2);
      xdlzero = pow((*naujas - *(field + x - 1)) ,2) - pow((*(field + x) - *(field + x - 1)) ,2);
    }
  else
    {
      xdl = pow((*naujas - *(field + x - 1)) ,2) - pow((*(field + x) - *(field + x - 1)) ,2);
      xdr = pow((*(field + x + 1) - *naujas) ,2) - pow((*(field + x + 1) - *(field + x)) ,2);
      xdlzero = pow((*naujas - *(field + x - 1)) ,2) - pow((*(field + x) - *(field + x - 1)) ,2);
      xdrzero = pow((*(field + x + 1) - *naujas) ,2) - pow((*(field + x + 1) - *(field + x)) ,2);
    }

  //t derivatives
  //points at the bottom
  if (x >= L*(L - 1))
    {
      tdu = pow((*naujas - *(field + x - L)) ,2) - pow((*(field + x) - *(field + x - L)) ,2);
      tdd = pow((*(field + x - L*(L - 1)) - *naujas) ,2) - pow((*(field + x - L*(L - 1)) - *(field + x
           )) ,2);
    }
  //points on top
  else if (x < L)
    {
      tdu = pow((*naujas - *(field + x + L*(L - 1))) ,2) - pow((*(field + x) - *(field + x + L*(L - 1)
           )) ,2);
      tdd = pow((*(field + x + L) - *naujas) ,2) - pow((*(field + x + L) - *(field + x)) ,2);
    }
  else
    {
      tdu = pow((*naujas - *(field + x - L)) ,2) - pow((*(field + x) - *(field + x - L)) ,2);
      tdd = pow((*(field + x + L) - *naujas) ,2) - pow((*(field + x + L) - *(field + x)) ,2);
    }
  untwisted_action = (action + 0.5*(xdlzero + xdrzero + tdu + tdd) + (alpha*a*a/(beta*beta))*(cos(beta
       * *(field + x)) - cos(beta * *naujas)));
  return (action + 0.5*(xdl + xdr + tdu + tdd) + (alpha*a*a/(beta*beta))*(cos(beta* *(field + x)) -
       cos(beta * *naujas)));
}


//returns the action for a given field
double get_action(double *field)
{
  double td = 0;
  double xd = 0;
  double V = 0;

  for (int x = 0; x < L*L; x++)
    {
      //x derivatives
      //points on the right
      if ((x - L + 1)%L == 0)
        {
          xd += pow((*(field + x + 1 - L) - *(field + x)), 2);
        }
      else
        {
          xd += pow((*(field + x + 1) - *(field + x)), 2);
        }
      //t derivatives
      //points at the bottom
      if (x >= L*(L - 1))
        {
          td += pow((*(field + x - L*(L - 1)) - *(field + x)), 2);
        }
      else
        {
          td += pow((*(field + x + L) - *(field + x)), 2);
        }
      //potential
      V += cos(beta* *(field + x));
    }
  return 0.5*(td + xd) - (alpha*a*a/(beta*beta))*(V-L*L);
}
```

B Mass.cpp

```
/*------------------------------------------------------------------------------------------------
MASS COMPUTATION FOR THE SINE-GORDON QUANTUM FIELD THEORY KINK

AUTHOR: LINAS BITKEVICIUS
IMPERIAL COLLEGE LONDON
DEPARTMENT OF THEORETICAL PHYSICS

This program works with the results that are output by simulation.cpp.
It computes the mass of the sine-Gordon kink.

Settings file is "results/Settings". It gets updated after running simulation.cpp. The format is as
     follows:

[Lattice size]
[a]
[beta]
[number of measurements per value (N)]
[last alpha]
[# of values]

------------------------------------------------------------------------------------------------*/

#include <iostream>
#include <fstream>
```

```
#include <cstdio>
#include <cmath>
#include <mpfr.h>
#include <cstdlib>
#include <fftw3.h>

using namespace std;

int autocorrelation();
double expectation(char *fname);
double naive_expectation(char *fname);
//double mass(double expectation);

double a = 1;
double beta = 1;
double starting_point = 0;
double alpha = starting_point;
double last_alpha;
int increments;
double delta_alpha;
double T = 64;
double N;
double dump;
double error = 0;//the total error

//mpfr variables//

mpfr_t mnum;
mpfr_t mresult;
mpfr_t mexpon;
mpfr_t mvalue;
mpfr_t mavesq;
mpfr_t mave;
mpfr_t mhold;
mpfr_t mN;
mpfr_t merror;
mpfr_t mtime;
mpfr_t mconst;

//--------------//

// FFT variables //

double *input;
fftw_complex *output;
fftw_plan plan_r2c;
fftw_plan plan_c2r;
int padding = 0; //USE THIS ONLY FOR IN-PLACE TRANSFORMS?

//----------------//

int main()
{
  //mpfr variables init and set//
  mpfr_init(mnum);
  mpfr_init(mresult);
  mpfr_init_set_d(mexpon, 0.0, GMP_RNDZ);
  mpfr_init(mvalue);
  mpfr_init_set_d(mavesq, 0.0, GMP_RNDZ);
  mpfr_init(mave);
  mpfr_init_set_d(mhold, 0.0, GMP_RNDZ);
  mpfr_init(mN);
  mpfr_init_set_d(merror, 0.0, GMP_RNDZ);
  mpfr_init(mtime);
  mpfr_init_set_d(mconst, 1.0, GMP_RNDZ);

  //----------------------------//

  //file names init//
  char twisted2[50];
  char periodic2[50];
  char twisted1[50];
  char periodic1[50];
  char twistednaive[50];
  char periodicnaive[50];
  char nmassname[50];
  char fmassname[50];
  //----------------//

  //other//
  double tn;//used in naive
  double pn;//--
  int bla;//holder for sprintf
  double M = 0;//fmass
  double nM = 0;//nmass
  double nerror1 = 0;//all the error variables
  double nerror2 = 0;
  double ntotalerror = 0;
  double simpsonerror = 0;
  double ferrorp1 = 0;
  double ferrorp2 = 0;
  double ferrort1 = 0;
  double ferrort2 = 0;
  double ftotalerror = 0;
  double multiplier;//simpson multiplier
  double ftw2;//components of fmass calculation
  double ftw1;
  double fp2;
  double fp1;//--
  bool even = false;//simpson only even
  //-----//

  //streams//
  ofstream fout;
  ofstream nout;
  ifstream setin;
  setin.open("results/Settings");
  //-------//

  //load settings from "results/Settings"//
  cout << "Loading results/Settings." << endl;
  setin >> T >> a >> beta >> N >> last_alpha >> increments;
  setin.close();
  cout << "Lattice size = " << T << endl;
  cout << "a = " << a << endl;
  cout << "beta = " << beta << endl;
  cout << "Number of measurements per value (N) = " << N << endl;
  cout << "Last value of alpha = " << last_alpha << endl;
  cout << "Number of values = " << increments << endl;
  mpfr_set_d(mN, N, GMP_RNDZ);
  bla = sprintf(nmassname, "massresults/NMASS_L%d_a%d_b%f_n%d", static_cast<int>(T), static_cast<int>(
      last_alpha), beta, increments);
  bla = sprintf(fmassname, "massresults/FMASS_L%d_a%d_b%f_n%d", static_cast<int>(T), static_cast<int>(
      last_alpha), beta, increments);
  fout.open(fmassname);
  nout.open(nmassname);

  //set starting values of masses (if needed) [SET TO ZERO BY DEFAULT]
  M = 8*sqrt(starting_point)/(beta*beta) - sqrt(starting_point)/3.141592;
  nM = 8*sqrt(starting_point)/(beta*beta) - sqrt(starting_point)/3.141592;

  delta_alpha = (last_alpha - alpha)/(static_cast<double>(increments) - 1);

  //initialize FFT arrays//
  input = (double*) fftw_malloc(sizeof(double)*static_cast<int>(N) + padding);
  output = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*static_cast<int>(N) + padding);
  //-------------------------------------//

  //FINITE DIFFERENCE METHOD//
  cout << "Starting the computation of FMASS." << endl;
  for (double i = alpha; i <= last_alpha + 0.0001; i = i + delta_alpha)
    {
      bla = sprintf(twisted1, "results/t_u_%f", i - delta_alpha);
      bla = sprintf(periodic1, "results/p_u_%f", i - delta_alpha);
      bla = sprintf(twisted2, "results/t_d_%f", i);
      bla = sprintf(periodic2, "results/p_d_%f", i);
      cout << "FMASS " << ((i - alpha)/(last_alpha - alpha))*100 << "% done" << endl;
      if (i == alpha)
        {
          M += 0;
          ftotalerror += 0;
        }
      else if (i == last_alpha)
        {
          ftw2 = expectation(twisted2);
          ferrort2 = error;
          fp2 = expectation(periodic2);
          ferrorp2 = error;
          M += (ftw2 - fp2);
          ftotalerror = (1/T)*0.25*sqrt(pow(ferrorp2, 2) + pow(ferrort2, 2) + 4*T*pow(ftotalerror, 2))
              ; //cia
        }
      else
        {
          ftw2 = expectation(twisted2);
          ferrort2 = error; //cia
          fp2 = expectation(periodic2);
          ferrorp2 = error; //cia
          ftw1 = -expectation(twisted1);
          ferrort1 = error; //cia
          fp1 = -expectation(periodic1);
          ferrorp1 = error; //cia
          M += 0.5*(ftw2 + ftw1 - fp2 - fp1);
          ftotalerror = (1/T)*0.25*sqrt(pow(ferrorp2, 2) + pow(ferrort2, 2) + pow(ferrort1, 2) + pow(
              ferrorp1, 2) + pow((ftw1 - ftw2) ,2) + pow((fp1 - fp2) ,2) + 4*T*pow(ftotalerror, 2));
              //cia
        }
      fout << i << " " << M << " " << ftotalerror << endl;
    }
  cout << "FMASS computation finished." << endl;
  //-----------------------//

  //NAIVE EXPECTATION VALUE CALCULATION//
  cout << "Starting the computation of NMASS." << endl;
  for  (double k = alpha; k <= last_alpha + 0.0001; k = k + delta_alpha)
    {
      //do simpson's integration
      cout << "NMASS " << 100*(k - alpha)/(last_alpha - alpha) << "% done" << endl;
      if (even == true)
        {
              even = false;
            }
          else
            {
              even = true;
            }
          bla = sprintf(twistednaive, "results/t_n_%f", k);
          bla = sprintf(periodicnaive, "results/p_n_%f", k);
          tn = naive_expectation(twistednaive);
          nerror1 = error;
          pn = naive_expectation(periodicnaive);
          nerror2 = error;
          if (k == alpha || k == last_alpha)//first one or last one
            {
              multiplier = 1;
            }
          else if (even == true)//even
            {
              multiplier = 2;
            }
          else if (even == false)//odd
            {
              multiplier = 4;
            }
          nM += multiplier*(T*delta_alpha*a/(beta*beta*3))*(pn - tn);
          //determine the error due to Simpson's integration
          if (simpsonerror < (pow(delta_alpha, 4)/180)*k*pow(nM, 4))
            {
              simpsonerror = (pow(delta_alpha, 4)/180)*k*pow(nM, 4);
            }
          ntotalerror = sqrt(pow(nerror1, 2) + pow(nerror2, 2) + pow(simpsonerror, 2) + pow(ntotalerror,
               2));
          if (even == true) //write out only the even terms (Simpson's rule only works for even)
            {
              nout << k << "␣" << nM << "␣" << ntotalerror << endl;
            }
        }
      cout << "NMASS␣computation␣finished." << endl;
      //-----------------------------------//

      //clear mpfr variables//
      mpfr_clear(mnum);
      mpfr_clear(mresult);
      mpfr_clear(mexpon);
      mpfr_clear(mvalue);
      mpfr_clear(mave);
      mpfr_clear(mavesq);
      mpfr_clear(mhold);
      mpfr_clear(mN);
      mpfr_clear(merror);
      mpfr_clear(mtime);
      mpfr_clear(mconst);

      //clear FFT variables//
      cout << "CLEARING␣FFT␣VARIABLES." << endl;
      fftw_destroy_plan(plan_r2c);
      fftw_destroy_plan(plan_c2r);
      fftw_free(input);
      fftw_free(output);
      //------------------//
      cout << "DONE!" << endl;
      cout << "FILE␣NAMES␣ARE:␣" << endl;
      cout << fmassname << endl;
      cout << nmassname << endl;
      return 0;
  }

  double expectation(char *fname)
  {
      ifstream fin;
      double value = 0;
      double expectation = 0; //will act as a holder
      double average = 0;
      double esq;

      double number = 0;
      int count = 0;
      int time = 1;

      //input for FFT
      fin.open(fname);
      if(fin.fail())
        {
          cout << "ERROR␣LOADING␣FILE.␣" << fname << endl;
          exit(1);
        }
      while (count < N)
        {
          fin >> input[count] >> dump;
          average += input[count];
          count++;
        }
      fin.close();//closes the stream
      average /= N;
```

```
//FFT

plan_r2c = fftw_plan_dft_r2c_1d(static_cast<int>(N), input, output, FFTW_ESTIMATE);
fftw_execute(plan_r2c);

//normalize + mod squared

for (int i = 0; i < N; i++)
  {
    output[i][0] /= sqrt(N);
    output[i][1] /= sqrt(N);
    output[i][0] = sqrt(pow(output[i][0], 2) + pow(output[i][1], 2));
    output[i][1] = 0;
  }

//IFFT and normalize

plan_c2r = fftw_plan_dft_c2r_1d(static_cast<int>(N), output, input, FFTW_ESTIMATE);
fftw_execute(plan_c2r);
for (int i = 0; i < N; i++)
  {
    input[i] /= sqrt(N);
  }

//find the autocorrelation time
for (int i = 0; i < N; i++)
  {
    input[i] -= average;
    if (input[i] < 1/exp(1))
      {
        time = i;
      }
    if (i == N)
      time = N;
    if (input[i] < 1/exp(1))
      break;
  }
if (time == 0)
  {
    time = 1;
  }
mpfr_set_d(mtime, static_cast<double>(time), GMP_RNDZ);

count = 0;

//compute the actual expectation value
mpfr_set_d(mexpon, 0.0, GMP_RNDZ);
fin.open(fname);
if(fin.fail())
  {
    cout << "ERROR_LOADING_FILE._" << fname << endl;
    exit(1);
  }
while (count < N)
  {
    fin >> value >> esq;
    mpfr_set_d(mhold, esq, GMP_RNDZ);
    mpfr_set_d(mave, (value/N), GMP_RNDZ);
    mpfr_add(mave, mave, mave, GMP_RNDZ);
    mpfr_div(mhold, mhold, mN, GMP_RNDZ);//get the average right here, since we know N
    mpfr_add(mavesq, mavesq, mhold, GMP_RNDZ);

    //get the expectation value
    mpfr_set_d(mvalue, value, GMP_RNDZ);//reads a value
    mpfr_exp(mvalue, mvalue, GMP_RNDZ);//exponentiates it
    mpfr_add(mexpon, mexpon, mvalue, GMP_RNDZ);//adds the exponential to the basket CHECK THIS OUT
         !!! MIGHT BE ERROR
    number++;
    count++;
  }
fin.close();//closes the stream
mpfr_set_d(mnum, number, GMP_RNDZ);//converts number to mnum
mpfr_div(mresult, mexpon, mnum, GMP_RNDZ);//gets the average (expectation value)


//get the error
mpfr_sqr(mave, mave, GMP_RNDZ);
mpfr_sub(mavesq, mavesq, mave, GMP_RNDZ);
mpfr_sub(mave, mN, mconst, GMP_RNDZ);
mpfr_mul(mave, mave, mN, GMP_RNDZ);
mpfr_div(merror, mavesq, mave, GMP_RNDZ);
mpfr_mul(merror, merror, mtime, GMP_RNDZ);
mpfr_sqrt(merror, merror, GMP_RNDZ);
mpfr_div(merror, merror, mresult, GMP_RNDZ);

//----

mpfr_log(mresult, mresult, GMP_RNDZ);//takes the log of it
expectation = mpfr_get_d(mresult, GMP_RNDZ);

mpfr_set_d(mave, 0.0, GMP_RNDZ);
mpfr_set_d(mavesq, 0.0, GMP_RNDZ);
mpfr_set_d(merror, 0.0, GMP_RNDZ);
```

```
  //return the average to get the expectation value
  return (1/T)*(expectation);
}

double naive_expectation(char *fname)
{
  ifstream fin;
  double number = 0;
  double value = 0;
  double expectation = 0;
  int count = 0;
  double average = 0;
  double averageofsq = 0;
  double esq;
  int time;
  fin.open(fname);
  if(fin.fail())
    {
      cout << "ERROR_LOADING_FILE._" << fname << endl;
      exit(1);
    }
  while (count < N)
    {
      fin >> input[count] >> esq;
      average += input[count];
      averageofsq += esq;
      count++;
    }
  average /= N;
  averageofsq /= N;
  fin.close();

  //do FFT here

  plan_r2c = fftw_plan_dft_r2c_1d(static_cast<int>(N), input, output, FFTW_ESTIMATE);//set up plan
  fftw_execute(plan_r2c);//execute the plan
  //for every argument in the output, get the modulus squared and put it back in the same array
  //normalization and mod squared
  for (int i = 0; i < N; i++)
    {
      output[i][0] /= sqrt(N);
      output[i][1] /= sqrt(N);
      output[i][0] = sqrt(pow(output[i][0], 2) + pow(output[i][1], 2));
      output[i][1] = 0;
    }

  //IFFT NOW

  plan_c2r = fftw_plan_dft_c2r_1d(static_cast<int>(N), output, input, FFTW_ESTIMATE);
  fftw_execute(plan_c2r);
  //normalization
  for (int i = 0; i < N; i++)
    {
      input[i] /= sqrt(N);
    }

  //find the autocorrelation time
  for (int i = 0; i < N; i++)
    {
      input[i] -= average; //get the real autocorrelation function
      if (input[i] < 1/exp(1))
        {
          time = i;
        }
      if (input[i] < 1/exp(1))
        break;
      if (i == N)
        time = N;
    }
  if (time == 0)
    {
      time = 1;
    }

  //compute the error
  error = sqrt(static_cast<double>(time)*(averageofsq - pow(average, 2))/(N*(N - 1)));

  count = 0;
  average = 0;

  //get the actual expectation value for the return. It is not the average, since we are now taking
       the autocorrelation time into account
  fin.open(fname);
  if(fin.fail())
    {
      cout << "ERROR_LOADING_FILE._" << fname << endl;
      exit(1);
    }
  while (count < N)
    {
      fin >> value >> dump;
      expectation += value;
      number++;
      count++;
    }
  fin.close();
  //return the average to get the expectation value
  return expectation/number;
}


int autocorrelation()
{
  return 1;
}
```

References

[1] J. Scott Russel. Report on waves. *Fourteenth meeting of the British Association for the Advancement of Science*, 1844.

[2] A. Kundu. Changing Solitons in Classical a Quantum Integrable Defect and Variable Mass Sine-Gordon Model. *Journal of Nonlinear Mathematical Physics*, 15:237–+, 2008.

[3] Sidney Coleman. Quantum sine-Gordon equation as the massive thirring model. *Physical Review D*, 11(8):2088–2097, April 1975.

[4] S. N. M. Ruijsenaars. Sine-Gordon Solitons Vs. Relativistic Calogero-Moser Particles. In *Proceedings of Integrable structures of exactly solvable two-dimensional models of quantum field theory 2000*, volume 35 of *NATO Science Series II Mathematics Physics Chemistry*, pages 273 – 292. Kluwer Dordrecht, 2001.

[5] A. B. Zamolodchikov. Exact two-particle s-matrix of quantum sine-gordon solitons. *Communications in Mathematical Physics*, 55:183–186, 1977. 10.1007/BF01626520.

[6] D. J. Bergman, E. Ben-Jacob, Y. Imry, and K. Maki. Sine-gordon solitons: Particles obeying relativistic dynamics. *Phys. Rev. A*, 27:3345–3348, Jun 1983.

[7] L. D. Faddeev V. E. Korepin. Quantization of solitons. *Teoret. Mat. Fiz.*, 25:147–163, Nov 1975.

[8] Kazumi Maki and Hajime Takayama. Quantum-statistical mechanics of extended objects. i. kinks in the one-dimensional sine-gordon system. *Phys. Rev. B*, 20:3223–3232, Oct 1979.

[9] G. Benfatto, P. Falco, and V. Mastropietro. Massless sine-gordon and massive thirring models: Proof of colemans equivalence. *Communications in Mathematical Physics*, 285:713–762, 2009. 10.1007/s00220-008-0619-x.

[10] C. dos Santos and D. Rubiera-Garcia. Generalized sine-Gordon solitons. 2011.

[11] R. Rajaraman. *Solitons and instantons: an introduction to solitons and instantons in quantum field theory.* North-Holland personal library. North-Holland Pub. Co., 1982.

[12] Roger F. Dashen, Brosl Hasslacher, and André Neveu. Particle spectrum in model field theories from semiclassical functional integral techniques. *Phys. Rev. D*, 11:3424–3450, Jun 1975.

[13] Arttu Rajantie and David J. Weir. Quantum kink and its excitations. *Journal of High Energy Physics*, 2009(04):068, 2009.

[14] Arttu Rajantie and David J. Weir. Nonperturbative study of the 't Hooft-Polyakov monopole form factors. 2011.

[15] Arttu Rajantie. Mass of a quantum 't hooft-polyakov monopole. *Journal of High Energy Physics*, 2006(01):088, 2006.

[16] J Groeneveld, J Jurkiewicz, and C P Korthals Altes. Twist as a probe for phase structure. *Physica Scripta*, 23(5B):1022, 1981.

[17] S. Nagy, I. Nándori, J. Polonyi, and K. Sailer. Functional renormalization group approach to the sine-gordon model. *Phys. Rev. Lett.*, 102:241603, Jun 2009.

[18] J. Smit. *Introduction to quantum fields on a lattice*, volume 15. 2002.

[19] M E J Newman and G T Barkema. *Monte Carlo Methods in Statistical Physics*, volume 96. Oxford University Press, 1999.

[20] C. Rebbi. *Lattice gauge theories and Monte Carlo simulations.* World Scientific, 1983.

[21] M. Richey. The evolution of markov chain monte carlo methods. 117:383–413, May 2010.