Lattice Simulation of the Magnetic Monopole Mass

Imperial College London

Department of Physics

Jose Moreno

Supervisor:
Prof. Arttu Rajantie

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

September 17, 2014

Abstract

Using lattice methods, the mass of the 't Hooft-Polyakov monopole in the 3+1 dimension Georgi-Glashow model is calculated non-perturbatively as the difference in energy between the vacuum lattice configuration and a configuration with one magnetic monopole. These two types of configurations are created by using C-periodic and twisted boundary conditions respectively. The mass calculation is carried out using several lattice sizes.

To Ana,
your patience and understanding
made this adventure possible.

Acknowledgements

The are probably too many people, to whom I am grateful, that have generously pushed me along the particular world line that has taken me here today. I would like to thank my parents for getting me started in this wonderful path of learning and for always putting their children's interests above their own. Jenson and Angy, for always forgiving me when missing out on play time with them. And Ana, for her continuous encouragement and support, but also for taking care of 'everything else' in life while I enjoyed my studies. Finally, I would like to thank my supervisor, Prof. Arttu Rajantie, for his excellent AQFT lectures and for his guidance and tutoring during the dissertation.

Contents

1 Introduction **5**

2 The Georgi-Glashow model **8**

3 The Metropolis method **12**

3.1 Error estimation - the bootstrap method 16

4 The monopole mass **19**

4.1 The mass calculation method 22

5 On the lattice **25**

5.1 Lattice implementation 27

5.2 Discretising the Georgi-Glashow model 30

5.3 Boundary conditions 34

6 Results **36**

6.1 Error calculations 41

6.2 Other simulations 43

7 Conclusions and further study **46**

A The simulation code **50**

A.1 Compilation and execution 50

A.2 Field representation 51

A.3 Annotated code 52

B Bootstrap code **66**

Chapter 1

Introduction

Even though the discovery of particles that carry just one type of electric charge has turned out to be relatively straightforward, this has not been the case with regards to magnetic charges. Magnetic monopoles, that is, stable particles carrying magnetic charges have remained experimentally elusive to this day.

However, there is strong theoretical evidence that allows us to remain hopeful that magnetic monopoles may one day be experimentally confirmed. Already in the early days of Quantum Mechanics, Dirac [1] showed, in 1931, that it was compatible with their existence and he even went on to claim that "*... one would be surprised if Nature had made no use of it*".

Much more generally, in 1974, 't Hooft [2] and Polyakov [3] found that any Grand Unified Theory (GUT) that, within their laws, included the subgroup, U(1), responsible for electromagnetism would also inevitably lead to the existence of magnetic monopoles.

The fact that, so far, only theory backs up the existence of magnetic monopoles has in no way made the effort devoted to their investigation less

worthwhile. On the contrary, their study has not only led to different discoveries along the way but has also helped in shaping the way other theories have been developed.

For example, Dirac's [1] investigation provided an explanation for the quantisation of electric charge while 't Hooft and Polyakov's work led the way for a much deeper understanding of gauge field theories. There is also research that points to magnetic monopoles as a possible explanation for QCD confinement [4].

However, the theories above predict that magnetic monopoles are quite heavy and so unlikely to be directly produced in current particle accelerators. Nonetheless, if they do exist, events in the Universe like the Big Bang should have been able to produce them in sufficient amounts to make their detection possible. Our failure to detect them so far then impose some limits to their number in today's Universe and this has forced us to review some of the theories about the Big Bang and the expansion of the Universe. This led to the proposal of the theory of *inflation* [5] in 1981 by Alan Guth which has been successful in explaining the monopole problem as well as the flatness and horizon problems in Cosmology. So we see here as well, that the no observation of monopoles has however been useful in suggesting other theories whose predictions have been successfully tested experimentally.

The purpose of this document is to use computer simulations to estimate the mass of the magnetic monopole as defined by 't Hooft [2] and Polyakov [3] under the Georgi-Glashow [6] model in 3+1 dimensions. Section 2 contains a brief description of the Georgi-Glashow model and how it can give rise to magnetic monopoles. Then section 3 introduces *Metropolis*, a simple but very successful general purpose algorithm that we use in our computer simulations.

The theoretical approach to the calculation of the magnetic monopole mass is described in section 4 while the actual calculation, which in practice is done numerically by discretising the theory on a lattice is explained in section 5. Finally the results of the simulations are presented in section 6 and suggestions for further investigation in section 7. The appendices contain a trimmed version of the code that was used in the simulation with some comments on the implementation.

For the interested reader, [7] provides an excellent theoretical and historical review of magnetic monopoles. Another accessible review worth looking at is provided in [8].

Chapter 2

The Georgi-Glashow model

As we have seen in the introduction there are many different theories which lead to magnetic monopoles. We will focus our interest in the 't Hooft-Polyakov monopole, which appears as the monopole solutions in Grand Unified Theories (GUTs). And in particular, one of the simplest theories in which we get these type of monopole solutions is the one based on the Georgi-Glashow model [6].

The Georgi-Glashow model in 3+1 dimensions consists of an SU(2) gauge field A_μ and a scalar Higgs field ϕ in the adjoint representation. The lagrangian of this theory can be written as:

$$\mathcal{L} = -\frac{1}{2}\,\mathrm{Tr}\,F_{\mu\nu}F^{\mu\nu} + \mathrm{Tr}[D_\mu,\phi][D^\mu,\phi] - m^2\,\mathrm{Tr}\,\phi^2 - \lambda(\mathrm{Tr}\,\phi^2 - \nu^2)^2,$$

where

$$D_\mu = \partial_\mu + igA_\mu$$

$$F_{\mu\nu} = [D_\mu, D_\nu]/ig = \partial_\mu A_\nu - \partial_\nu A_\mu + ig[A_\mu, A_\nu].$$

The adjoint scalar field ϕ in SU(2) is a 3-component field. That means that we can represent ϕ as a 3-component vector, obviously not in real space

but in some internal vector space, which we can visualise as an arrow in this internal space.

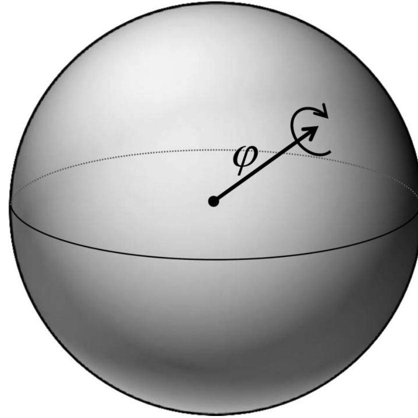


Figure 2.1: The possible vacuum states of the Higgs field have equal length (within an internal three dimensional space), forming an S^2 topological space. Each vacuum state on the surface is equally probable, but once one particular vacuum state is chosen the SU(2) symmetry of the system is broken by the Higgs mechanism and only the U(1) symmetry corresponding to electromagnetism survives.
Image Source: [7]

In the vacuum, the Higgs field ϕ has a non-zero value with length $|\phi| = \nu$ and using the 3-component representation this looks like a sphere of possible vacuum solutions (Figure 2.1). The actual vacuum however needs to choose one of these solutions and when this occurs the SU(2) symmetry of the system is spontaneously broken to an U(1) symmetry which corresponds to the familiar one in electromagnetism. This occurs when m^2 is negative and in that case we have

$$\mathrm{Tr}\,\phi^2 = \nu^2 \equiv \frac{-m^2}{2\lambda}.$$

Although the vacuum expectation value of the Higgs field fixes its length to $\nu = |\phi|$, its direction can vary from point to point. The 't Hooft-Polyakov solution corresponds to the case where, at every point, ϕ points away from the origin and, as this configuration looks like spikes coming out of the origin (Figure 2.2), for this reason it is also called the hedgehog solution [7].

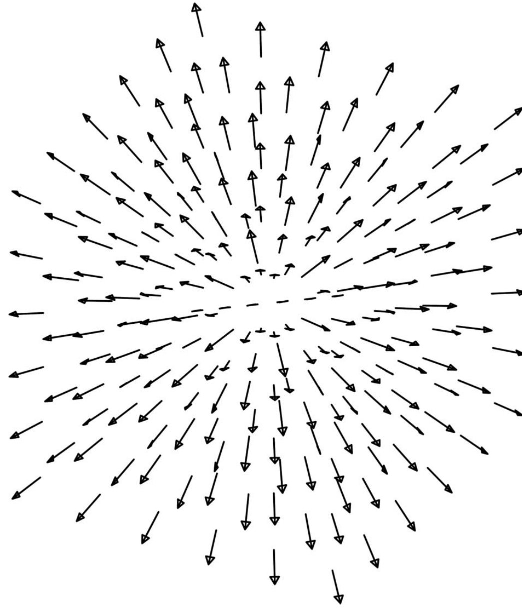


Figure 2.2: The hedgehog configuration of the 't Hooft-Polyakov monopole solution. Image Source: [7]

What we find is that the 't Hooft-Polyakov solution has finite energy, it is smooth at the origin and its energy density falls off exponentially so if we integrate over space it has finite energy and appears localised at the origin. As it carries finite energy and is localised in a small volume around a point, we can think of it as a point-like particle. In addition, as in the broken phase we have U(1) electrodynamics symmetry, we find that the magnetic charge of the hedgehog solution is not zero and hence the point-like particle corresponds to a magnetic monopole.

It is worth mentioning that the SU(2) theory we are considering is the simplest where we obtain these 't Hooft-Polyakov monopole solutions. However one can show by topological arguments that any GUT will always have these monopole solutions and therefore these are a generic prediction of Grand Unification. This is one reason why many scientists expect magnetic monopoles to exist. The problem is that the energy of the solution is at the GUT

scale (around 10^{16} GeV) and so the magnetic monopoles are predicted to be extremely heavy and we would not expect to find them in current particle accelerator experiments. Still, as we do here, it is possible to study some of its properties from a theoretical point of view, for example its mass.

Chapter 3

The Metropolis method

The Metropolis algorithm [9, 10, 11] is a Monte Carlo based method that is very popular and successful when solving simulation problems where there are no exact analytic or numerical solutions available.

Roughly speaking, the Metropolis algorithm generates a sequence of random field configurations with a desired probability distribution. To generate this sequence, it starts from an initial configuration $\phi(x)$ which we are free to choose. For example this initial configuration could be the vacuum or some random field configuration. It then changes $\phi(x)$ by some arbitrary amount $\Delta\phi(x)$ in order to obtain a new candidate configuration $\phi'(x)$. This process is called a *Metropolis update*,

$$\phi \to \phi' = \phi + \Delta\phi.$$

The amount $\Delta\phi(x)$ needs to be generated in such a way that the transformation is reversible and symmetric, in the sense that the probability of going from configuration $\phi(x)$ to $\phi'(x)$ should be the same as the probability of going from $\phi'(x)$ to $\phi(x)$.

We refer to the new configuration as a *candidate* configuration as it still needs to be accepted or rejected. In the case that it is accepted the candidate configuration becomes the next configuration in the sequence, otherwise the new configuration is rejected and the next configuration in the sequence is set to be equal to the previous configuration.

As we move from one configuration to the next the action of the system changes by an amount $\Delta S[\phi]$ which we need to calculate as part of the simulation as this value will determine whether the new candidate configuration is accepted or rejected:

$$\Delta S[\phi] = S[\phi'] - S[\phi].$$

If the amount $e^{-\Delta S[\phi]}$ is greater than an uniformly generated random number between zero and one then the candidate configuration is accepted, otherwise rejected. This implies that if the action decreases, that is, $\Delta S[\phi]$ is negative, then the candidate configuration will always be accepted as the exponential will in this case be greater than one and so greater than the random number.

In practice, we want the sequence of configurations to explore as much of the phase space as possible. Hence we need to strike a balance. If $\Delta\phi(x)$ is too small then the sequence of configurations are very close to each other and we may not explore enough of the phase space. On the other hand if $\Delta\phi(x)$ is too large then chances are that the probability of acceptance is low and many of the configurations will be rejected, wasting computer resources. Heuristically we tune $\Delta\phi(x)$ in such a way that the probability of acceptance is around 60–70%.

Repeatedly using the above algorithm leads to a sequence of field con-

figurations whose probability distribution, in the limit, is an attractive fixed point. This implies that after enough updates have been carried out the distribution would have converged to this fixed point and then if $\phi(x)$ has this probability distribution so will $\phi'(x)$. In essence, this means that we need to perform a number of updates, N_{therm}, before we can start taking measurements, in order to ensure that the system has reached an *equilibrium*. This procedure of updating the system until we reach the desired probability distribution is called the *thermalisation* process. How long the thermalisation process takes will depend on the parameters of the algorithm and needs to be estimated by performing some test runs.

Once the system has been thermalised we can start taking measurements. However, it is not convenient to take measurements on every single configuration update as consecutive configurations will be too close together and so highly correlated. Hence we want to leave a number of updates, N_{cor}, between measurements so that each measurement is independent of the previous one. In general, the thermalisation process is much longer than the number of updates between measurements, for example, $N_{therm} \approx 5N_{cor}$–$10N_{cor}$.

At each measurement, we calculate the value of the observable we are interested in. This could be the value of the field at any given point in the lattice or more typically an average. In our case, and because the system is translation invariant, we are never interested in the value of the field at a given point but instead in some kind of volume average on the whole lattice.

Given the standard expression for the expectation value of some operator Θ which depends on the fields $\phi(x)$ and $U(x)$

$$\langle \Theta[\phi, U] \rangle = \frac{\int \mathcal{D}\phi \mathcal{D}\mathcal{U}\, \Theta[\phi, U] e^{-S_E}}{\int \mathcal{D}\phi \mathcal{D}\mathcal{U}\, e^{-S_E}},$$

we can estimate the path integrals by following the Metropolis algorithm [12]

whose basic steps are shown below:

1. Choose an arbitrary initial configuration for the fields ϕ and U.
2. Update the configurations N_{therm} times to thermalise the system.
3. Update the configurations N_{cor} times and measure $\Theta[\phi, U]$.
4. Repeat step 3 N times.
5. Calculate the average of the N measurements to obtain an estimate for $\langle\Theta\rangle$.

The algorithm to update the field $\phi(x)$ and obtain a new configuration $\phi'(x)$ is as follows:

1. Choose a lattice site x_i and update $\phi(x_i)$ by a random number δ to obtain $\phi'(x_i)$.
2. Metropolis step:
   Calculate the change in the action, ΔS, due to the change in step 1.
   If $\Delta S < 0$, accept the new value $\phi'(x_i)$.
   If $\Delta S > 0$, accept $\phi'(x_i)$ with probability $e^{-\Delta S}$, that is, generate a random number ϵ between 0 and 1 and accept if $e^{-\Delta S} > \epsilon$.
   Otherwise, reject the change and make $\phi'(x_i) = \phi(x_i)$.
3. Repeat steps 1 and 2 for every site in the lattice (*Metropolis sweep*).
   At the end of the sweep we have a new configuration $\phi'(x)$.

We would update $U(x)$ following a similar algorithm, although note that, as we will see, $U \in \mathrm{SU}(2)$, then we would need to ensure that the new configuration U' remains an element of the SU(2) group.

3.1 Error estimation - the bootstrap method

The Metropolis algorithm described in the previous section gives us a sequence of configurations $\phi^{(k)}(x)$ with, $k = 1 \dots N$. On each of those configurations (and ignoring the field U for now) we take a measurement $\Theta^{(k)} \equiv \Theta[\phi^{(k)}(x)]$. And finally we calculate the mean of those N measurement values as an approximation to the expectation value $\langle\Theta\rangle$.

What we would like next is to have an estimate of the error, i.e. an estimate of the variability (standard deviation) in $\langle\Theta\rangle$. One way to do this would be to repeat the whole Metropolis algorithm above multiple times, say M. With each one we obtain a value for the expectation $\langle\Theta\rangle$ so that at the end of the process we end up with a sequence of values $\langle\Theta\rangle^{(b)}$, with $b = 1 \dots M$. Then, to estimate the error in $\langle\Theta\rangle$, we could just calculate the usual statistical standard deviation:

$$\Delta\langle\Theta\rangle = \sqrt{\frac{1}{M-1}\sum_{b=1}^{M}\left(\langle\Theta\rangle^{(b)} - \overline{\langle\Theta\rangle}\right)^2},$$

where

$$\overline{\langle\Theta\rangle} = \frac{1}{M}\sum_{b=1}^{M}\langle\Theta\rangle^{(b)}$$

is the mean value of the M Metropolis results.

The problem with this approach is that each iteration of the Metropolis algorithm to calculate $\langle\Theta\rangle$ is computationally very expensive and we would need to repeat it M times with M typically large, in the order of hundreds or thousands.

An alternative to this is the *Bootstrap* method [13], which is the one we have used in our calculations.

The Bootstrap method is very useful in situations where either the underlying distribution is unknown or is such that it is difficult to extract samples from. It is based on the idea that the statistics of the unknown (or real distribution) is the same or close to the statistics in a sample of it and one of its main strengths is its simplicity. It is known as a *resampling* method as in order to infer those statistics it uses new sample datasets that it creates by *re-sampling* the original sample dataset (allowing repetition).

In our particular case, rather than repeating the costly Metropolis algorithm over and over to generate new samples, we can use the bootstrap method to create new sets of measurements $\Theta^{(k,b)}$ by re-sampling the original set $\Theta^{(k)}$, $k = 1 \dots N$. In this way we could cheaply create M datasets, calculate the average of each one of them to obtain $\langle \Theta \rangle^{(b)}$, $b = 1 \dots M$ and then calculate the standard deviation of these in the same way as above.

In summary, the bootstrap method consists of the following steps:

1. Start from an original set of measurements $\{m_1, \dots, m_N\}$.
2. Re-sample the original set, with repetition, to obtain a new set $\{m_1^{(1)}, \dots, m_N^{(1)}\}$.
3. Calculate the average of the new set as

$$\langle m \rangle^{(1)} = \frac{1}{N} \sum_{i=1}^{N} m_i^{(1)}.$$

4. Repeat steps 2 and 3 M times to obtain a series of averages $\langle m \rangle^{(1)}, \dots, \langle m \rangle^{(M)}$.
5. Compute the standard error of the set in step 4 as

$$\Delta \langle m \rangle = \sqrt{\frac{1}{M-1} \sum_{b=1}^{M} \left(\langle m \rangle^{(b)} - \overline{\langle m \rangle} \right)^2},$$

where

$$\overline{\langle m \rangle} = \frac{1}{M}\sum_{b=1}^{M}\langle m \rangle^{(b)}.$$

Appendix B contains the implementation of the bootstrap algorithm used in the error analysis calculations.

Chapter 4

The monopole mass

When calculating the mass associated to particles of a scalar field we traditionally use the two point correlation function in momentum space which, for a free field, is given by the propagator, for example:

$$\frac{1}{k^2 - m^2}.$$

In the free theory, the field operator creates a particle. In an interacting theory however the field operator not only creates the particle but also a superposition of states. Therefore, to calculate the mass we look at the two point correlator at long distances, as then the lowest state corresponding to the particle dominates and the two point correlation function in Euclidean space goes like e^{-mx}. Hence the way we would measure the mass of the particle would be to start from the correlation function at long distances and look at the exponent in the exponential decay.

The above method works well for the scalar field, QED, etc. However we can not use this method for the magnetic monopole theory as we do not have any local creation operator that carry magnetic charge and hence we can not

construct a correlator for it[1].

Therefore here we use an alternative approach to calculate the mass of the magnetic monopole based on the free energy difference between sectors with one magnetic monopole and zero magnetic monopoles.

Because the magnetic charge is a conserved quantity, the Hilbert space factorises in sectors corresponding to states with zero, one, two, ... magnetic charges. This means that if we start with a state which has n magnetic monopoles then the system can only evolve to other states with n magnetic charges, i.e. it can only remain within the same sector. Consequently, we can just concentrate on the partition functions Z_n for the individual topological sectors as the Hilbert space is then just given by the product of those sectors.

We can write the partition function, Z_n, of the sector with magnetic charge n in Euclidean space as:

$$Z_n = \int \mathcal{D}\phi \, e^{-S_E[\phi]}, \tag{4.1}$$

where the integral is only over the states with magnetic charge n, and the Euclidean action S_E is given by

$$S_E = \int d^4x \left\{ \frac{1}{2} \operatorname{Tr} F_{ij}F^{ij} + \operatorname{Tr}[D_i, \phi][D^i, \phi] + m^2 \operatorname{Tr} \phi^2 + \lambda(\operatorname{Tr} \phi^2 - \nu^2)^2 \right\}.$$

The case $n = 0$ corresponds then to the vacuum state, with zero magnetic charge, while $n = 1$ is the vacuum state plus one monopole particle.

The only difference between Z_1 and Z_0 is that in Z_1 we have a monopole in all of the system configurations. Hence in the path integral for Z_1 we therefore have a monopole worldline going through the lattice.

[1]This is not entirely accurate. There has been attempts [14, 15, 16] at building a creation operator for magnetic monopoles but hey have other complications and hence we do not consider them here.

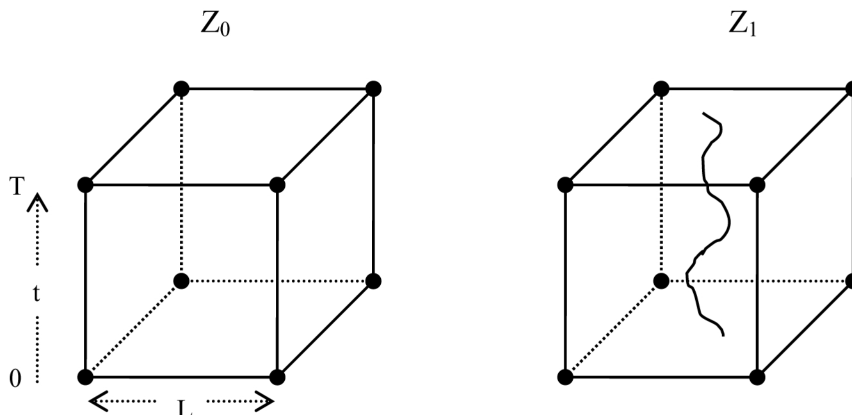


Figure 4.1: Example lattice configuration in the zero and one sectors.

Taking the ratio of the partition functions Z_1 and Z_0 will then give us just the contribution from the one magnetic monopole worldline.

When we put our system on the lattice we need to make it finite not only in spatial directions but also in the time direction. In that case one can write the partition function as the trace of the time evolution operator $\hat{U}(T)$ and expand in some complete set of energy eigenstates

$$Z = \mathrm{Tr}\,\hat{U}(T) = \sum_n \langle n|\hat{U}(T)|n\rangle,$$

where $\hat{U}(t) = e^{-i\hat{H}t}$ in Minkowski spacetime but becomes $\hat{U}(t) = e^{-\hat{H}t}$ under Euclidean time (after a Wick's rotation, $t \to -it$). Then the partition function becomes

$$Z = \sum_n \langle n|e^{-\hat{H}T}|n\rangle = \sum_n e^{-E_n T}$$

and E_n is the eigenvalue associated to the orthonormal state $|n\rangle$.

When T is large, $T \gg L$, then $Z \approx e^{-E_0 T}$. In the topological sector one, the ground state is when the monopole is stationary and has energy $E_0 = M$, while for the sector zero, which corresponds to the vacuum, the energy is zero. Hence we can write the ratio of these partition functions to

leading order as

$$\frac{Z_1}{Z_0} = e^{-MT},$$

where M is the monopole mass and T is the length of the system in the time direction.

Note however that the above expression is valid when $T \to \infty$, but in practice, in the lattice T is usually similar to L so the derivation is a bit more complicated. In this case (see [17]) there is an adjustment factor K at leading order so that the expression is instead

$$\frac{Z_1}{Z_0} = Ke^{-MT} \qquad \text{where } K = 2\left(\frac{mL^2}{2\pi T}\right)^{3/2}.$$

The main idea is then to calculate the ratio of these partition functions using the computer simulations and then once this ratio is known we can work out M as

$$M = K' - \frac{1}{T}\ln\frac{Z_1}{Z_0} \qquad \text{where } K' = \frac{1}{T}\ln K. \tag{4.2}$$

4.1 The mass calculation method

The problem with the calculation of M in expression (4.2) however is that in a Monte Carlo simulation we are not able to calculate partition functions like Z but only expectation values. Hence we need to find a way to transform this formula into a function of some expectation values so that we can calculate it in our simulations.

Going back to the partition function in (4.1), we can work out its derivative with respect to the parameter m^2:

$$\frac{\partial Z}{\partial m^2} = -Z\langle \mathrm{Tr}\,\phi^2\rangle.$$

And this can be related to M by similarly differentiating (4.2) with respect to the same parameter m^2:

$$\frac{\partial M}{\partial m^2} = -\frac{1}{T}\left(\frac{1}{Z_1}\frac{\partial Z_1}{\partial m^2} - \frac{1}{Z_0}\frac{\partial Z_0}{\partial m^2}\right) = \frac{1}{T}\left(\langle \mathrm{Tr}\,\phi^2\rangle_1 - \langle \mathrm{Tr}\,\phi^2\rangle_0\right), \tag{4.3}$$

where $\langle\cdot\rangle_1$ and $\langle\cdot\rangle_0$ denote expectation values in topological sectors one and zero respectively.

The expression in (4.3) is now a quantity that is expressed in terms of expectation values so it can be calculated using Monte Carlo simulations. For this, we need to calculate the expectation value of the trace of the scalar field squared twice, once for configurations with a single monopole (sector 1) and another for vacuum configurations (sector 0), and subtracting one from the other.

Finally to calculate M we need to integrate (4.3). We can do this by repeating the whole calculation for different values of m^2 and, by collecting enough data in this way, we can then integrate by expressing the derivative as a finite difference.

Therefore in practice, we measure both sets of data (sector 1 and sector 0) for different values of m^2. For large values of m^2, in the symmetric phase of the theory, the monopole does not exist and so in this phase both sets of data coincide and are quite small and consequently we have that the mass of the monopole is $M = 0$.

Then, there is a critical value, m_c^2, below which the symmetry is spontaneously broken and $\mathrm{Tr}\,\phi^2$ starts to increase. This is when we get a magnetic monopole. In this broken phase there is a slight difference between the expectation values in sectors one and zero. What the integration of (4.3) does is calculate the area between the sector 1 and sector 0 curves, and this is the

value of M we are trying to calculate.

Chapter 5

On the lattice

The objective of our simulations will be to calculate the mass of the 't Hooft-Polyakov monopole under the Georgi-Glashow theory introduced in previous sections. Typically, observables of the system are expressed as expectation values of some function of the fields in the theory, for instance, $\langle\phi(x)\rangle$, $\langle\phi(x)\phi(y)\rangle$, etc. and hence we would like to end up with an expression for the mass in terms of expectation values.

In general, for an operator $\Theta[\phi]$ which is a function of the field ϕ, its expectation value is given by the path integral:

$$\langle\Theta[\phi]\rangle = \frac{\int\mathcal{D}\phi\,\Theta[\phi]e^{-iS[\phi]}}{\int\mathcal{D}\phi\,e^{-iS[\phi]}}, \tag{5.1}$$

where the integrals are over all possible field configurations and $S[\phi]$ is the action of the theory.

Under weak coupling we can use perturbation methods to calculate the above expression by expanding around the (small) coupling constants of the theory. In this document however we will use a *non-perturbative* method based on discretising the theory on a lattice and using a Monte Carlo based algorithm, *Metropolis*, to calculate the integrals. This would allow us to

continue to use the same method even for the case in which the coupling g increases and the theory becomes strongly coupled. Another advantage of this method is that it lends itself nicely to computer simulations as we are replacing the continuous theory with a discrete one.

It is useful when working on lattice simulations to perform a *Wick's rotation* on the action, i.e. take $t \to -it$. This results in replacing the Minkowski path integrals with Euclidean path integrals which are better suited to numerical work as the integrands do not oscillate wildly in sign and have better convergence. Hence the expectation values in (5.1) are calculated instead as:

$$\langle \Theta[\phi] \rangle = \frac{\int \mathcal{D}\phi \, \Theta[\phi] e^{-S_E[\phi]}}{\int \mathcal{D}\phi \, e^{-S_E[\phi]}}, \tag{5.2}$$

where $S_E[\phi]$ is the Euclidean action.

We notice now that by defining $Z = \int \mathcal{D}\phi \, e^{-S_E[\phi]}$, the above path integral for the expectation value has the same form as the mean value of the operator under a probability density function given by

$$P[\phi] = \frac{e^{-S_E[\phi]}}{Z}.$$

Using this fact, a typical Monte Carlo simulation that aims to calculate the expectation value $\langle \Theta[\phi] \rangle$ would consist of two steps. In the first one a series of field configurations are generated with the correct probability distribution $P[\phi]$. In the second step the observables are calculated by taking measurements of the operator in those configurations and averaging these measurements. In this way we can approximate the expectation value by the average or mean of the measurements, that is

$$\langle \Theta[\phi] \rangle \approx \overline{\Theta[\phi]} = \lim_{N \to \infty} \frac{1}{N} \sum_{i=1}^{N} \Theta[\phi].$$

Each configuration consists of a value of the field at each point in the lattice. The question is then how to generate these field configurations such that they have the correct probability distribution $P[\phi]$. The algorithm we use here, one of the most general and simplest, is called *Metropolis* [9] and was described in chapter 3.

5.1 Lattice implementation

In order to perform the simulations on the computer we can only have a finite number of points on which to evaluate our theory so therefore we have to transform our continuous theory to an equivalent discrete theory.

We achieve this by considering a 4-dimensional Euclidean space-time hypercube of finite volume $V = TL^3$ where T and L are the lengths of the hypercube in the time and spatial dimensions respectively. In addition, we make the 4-coordinate x discrete by allowing only those values that fall on a lattice with nodes separated by a distance a, that is, the points on the lattice have coordinates $x^\mu = an^\mu$, with $n^\mu \in \mathbb{Z}^4$. The continuum physics can then be recovered by taking the limits $V \to \infty$ and $a \to 0$.

Next, we need to define our action on this lattice and this involves evaluating the gauge and Higgs fields on the nodes of the lattice. However, in doing so, we have to be very careful not to break any of the symmetries which are fundamental to our theory, in particular we want to keep the action locally gauge invariant.

When we discretise the field $\phi(x)$ on a lattice we define values for the field at each node of the lattice, $\phi(an^\mu)$. However the case of the gauge field $A_\mu(x)$ is more complicated. In principle we could try to consider the gauge

field as four scalar fields that we define at each node. The problem with this approach is that expressions written in terms of the gauge field are not gauge invariant. This is because the gauge field transforms under an SU(2) transformation $\Lambda(x)$ as

$$A_\mu \to \Lambda A_\mu \Lambda^\dagger - \frac{i}{g}\Lambda\partial_\mu\Lambda^\dagger,$$

which depends on derivatives of the gauge transformation $\Lambda(x)$. On the lattice, this derivative would be converted to a finite difference involving the values of Λ at two neighbouring nodes, which do not necessarily obey all properties of derivatives. This suggests that we should not define the gauge field at each point in the lattice but instead we should think of it as living in the links between two nodes.

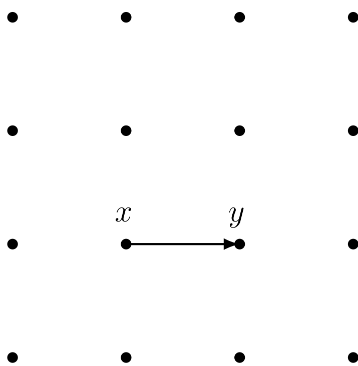


Figure 5.1: The way the gauge field transforms at point x depends on how it is transforming at point y. This means we should think of the gauge field as living between the two points.

In practice what this means is that instead of using $A_\mu(x)$ at each site, we represent the gauge field by link variables $U_\mu(x)$ defined as a path-ordered integral:

$$U(x,y) \equiv U_\mu(x) \equiv e^{ig\int_x^y dx\cdot A} = e^{ig\int_x^{x+a\hat{\mu}} dx\cdot A},$$

where $\hat{\mu}$ is a unit vector in the direction from x to y and $y = x + a\hat{\mu}$.

The link variables $U_\mu(x)$ now transforms in a much simpler way under a

gauge transformation Λ

$$U_\mu(x) \to \Lambda(x) U_\mu(x) \Lambda^\dagger(x + a\hat{\mu}).$$

However, this expression means that it does not transform like a local field but instead gets contributions from two points. Nevertheless, we will soon see how we can still use these link variables to discretise our action.

Graphically, we can depict [12] the link variable $U_\mu(x)$ by a directed line from node x to node $x + a\hat{\mu}$ representing the integration path in the line integral in the exponent of $U_\mu(x)$

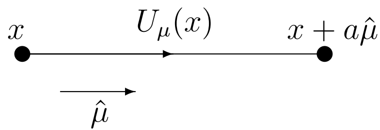


Figure 5.2: We write our theory in terms of the link variables U, which live at the links between two points in the lattice, rather than in terms of the variables A_μ which would live at the nodes.

A more geometrical view of the link variable U is as a parallel transporter. In order to compare the value of the fields at two different points x and y (for instance when calculating a derivative or finite difference) we need to parallel transport one of them to the location of the other. In our case, $U_\mu(x)$ is a matrix, living at the link between x and $x + a\hat{\mu}$, which can be used to parallel transport $\phi(x + a\hat{\mu})$ to x. Similarly, we can use its hermitian conjugate to parallel transport $\phi(x)$ to node $x + a\hat{\mu}$. That is:

$$\begin{array}{ll} U_\mu(x)\phi(x + a\hat{\mu}) & \text{lives at x,} \\ U_\mu^\dagger(x)\phi(x) & \text{lives at } x + a\hat{\mu}. \end{array}$$

Ken Wilson [18] came up, in 1974, with the idea of using these link variables to rewrite the action and hence making it suitable for lattice calculations.

In our case, the continuous action has the form

$$S = \int d^4x \left\{ -\frac{1}{2} \operatorname{Tr} F_{\mu\nu} F^{\mu\nu} + \operatorname{Tr}(D_\mu \phi D^\mu \phi) + V(\phi) \right\},$$

and so we need to find an alternative expression that is written in terms of link variables in such a way that preserves the gauge invariance and is local.

5.2 Discretising the Georgi-Glashow model

If we multiply two consecutive link variables $U(x, y)$ and $U(y, z)$ together and take the trace then the result will now depend on the two end nodes x and z.

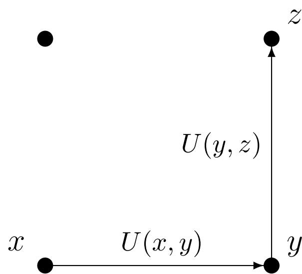


Figure 5.3: The trace of the product $U(x, y)U(y, z)$ depends on x and z but is not gauge invariant or local.

If we continue in this way and take the product of consecutive link variables that form a loop (called a *Wilson loop*) then the final expression is not only local, i.e. depends only on x, but in addition, after taking the trace, it will also be gauge invariant. The simplest loop we can form in this way is what is called the *Wilson plaquette* and expands a single lattice cell.

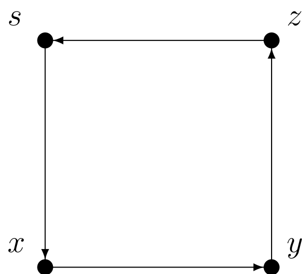


Figure 5.4: The *Wilson plaquette*, $U(x, y)U(y, z)U(z, s)U(s, x)$ now depends just on x and its trace is gauge invariant.

If we denote by $\hat{\mu}$ the unit vector in the direction from point x to point y and by $\hat{\nu}$ the unit vector in the direction from point y to z then we can write the Wilson plaquette $P_{\mu\nu}$ in its more traditional form:

$$P_{\mu\nu} = \mathrm{Tr}\, U_\mu(x) U_\nu(x + a\hat{\mu}) U_\mu^\dagger(x + a\hat{\nu}) U_\nu^\dagger(x),$$

where $U_\mu(x) \equiv U(x, x + a\hat{\mu})$.

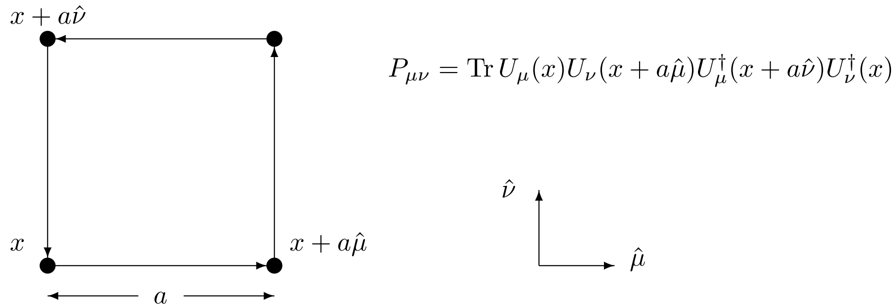


Figure 5.5: The Wilson plaquette.

Note that in this case we denote a link pointing in the opposite direction by $U_\mu^\dagger(x) \equiv U(x + a\hat{\mu}, x)$.

For smooth and slowly varying gauge fields we can approximate the link variable as

$$U_\mu(x) \approx e^{iagA_\mu(x)} \approx 1 + iagA_\mu(x) + O(a^2).$$

With this approximation we can Taylor expand the Wilson plaquette in powers of a and it turns out that for the case of SU(2) it is related to the continuous gauge field $A_\mu(x)$ in the following way:

$$P_{\mu\nu} \approx 2 - \frac{a^4 g^2}{2}\, \mathrm{Tr}\, F_{\mu\nu} F^{\mu\nu} + O(a^6),$$

where a is the lattice spacing.

We can now replace the kinetic term of the action with an expression in terms of the link variables, in particular the plaquette, i.e.

$$\frac{1}{2}\operatorname{Tr} F_{\mu\nu}F^{\mu\nu} \approx \frac{1}{a^4 g^2}(2 - P_{\mu\nu}).$$

Note that instead of the Wilson plaquette we could have chosen any other closed loop as these are also gauge invariant, however, this is the simplest choice. In fact, by a careful choice of the loop we can obtain expressions where the order a^4 term cancels out and therefore achieving greater accuracy.

Making explicit the sum over the μ and ν indices, and noticing that the term is anti-symmetric in those indices the corresponding Lagrangian term is:

$$\frac{1}{2}\operatorname{Tr} F_{\mu\nu}F^{\mu\nu} \approx \sum_{\mu,\nu} \frac{1}{a^4 g^2}(2 - P_{\mu\nu}) = \sum_{\mu<\nu} \frac{2}{a^4 g^2}(2 - P_{\mu\nu}),$$

where the extra factor of 2 in the last term comes from replacing μ, ν in the sum with $\mu < \nu$.

In a similar way as above, we can discretise the gradient term of the action in terms of the link variables by writing the covariant derivative of the scalar field as:

$$D_\mu \phi = \frac{U_\mu \phi(x + a\hat{\mu})U_\mu^\dagger - \phi(x)}{a}.$$

This expression parallel transports $\phi(x + a\hat{\mu})$ to position x and subtracts $\phi(x)$ to calculate the derivative in the μ direction.

Using this expression the gradient term in the lagrangian can then be written as

$$\operatorname{Tr}(D_\mu \phi D^\mu \phi) \approx \sum_\mu \frac{\operatorname{Tr}\phi^2(x + a\hat{\mu}) + \operatorname{Tr}\phi^2(x) - 2\operatorname{Tr}\phi(x)U_\mu \phi(x + a\hat{\mu})U_\mu^\dagger}{a^2}.$$

The potential term in the action does not need any special treatment as it is already correctly defined at the sites, so we have

$$V(\phi) = \lambda(\operatorname{Tr}\phi^2 - \nu^2)^2 = \lambda(\operatorname{Tr}\phi^2)^2 + \lambda\nu^4 - 2\lambda(\operatorname{Tr}\phi^2)\nu^2,$$

and as $\nu^2 \equiv -\frac{m^2}{2\lambda}$, the potential terms result in the following contribution to the lagrangian

$$V(\phi) = \lambda(\operatorname{Tr}\phi^2)^2 + \frac{(m^2)^2}{4\lambda} + m^2 \operatorname{Tr}\phi^2.$$

Putting all the terms together we end up with the following discretised Euclidean lagrangian:

$$\begin{aligned}
\mathcal{L}_E &= \frac{4}{a^4 g^2}\sum_{\mu<\nu}(1 - \frac{1}{2}P_{\mu\nu}) \\
&+ \frac{1}{a^2}\sum_{\mu}\Big(\operatorname{Tr}\phi^2(x+a\hat{\mu}) + \operatorname{Tr}\phi^2(x) - 2\operatorname{Tr}\phi(x)U_\mu(x)\phi(x+a\hat{\mu})U_\mu^\dagger(x)\Big) \\
&+ \lambda(\operatorname{Tr}\phi^2)^2 + \frac{(m^2)^2}{4\lambda} + m^2\operatorname{Tr}\phi^2.
\end{aligned}$$

Finally, to obtain the action we need to integrate over space-time

$$S_E = \int d^4x\, \mathcal{L}_E,$$

and this gets transformed on the lattice to

$$\begin{aligned}
S_E &= \sum_x a^4\,\mathcal{L}_E \\
&= \sum_x \Bigg\{\beta\sum_{\mu<\nu}\Big(1 - \frac{1}{2}P_{\mu\nu}\Big) \\
&+ a^2\sum_\mu\Big(2\operatorname{Tr}\phi^2(x) - 2\operatorname{Tr}\phi(x)U_\mu(x)\phi(x+a\hat{\mu})U_\mu^\dagger(x)\Big) \\
&+ \lambda a^4(\operatorname{Tr}\phi^2)^2 + a^4\frac{(m^2)^2}{4\lambda} + a^4 m^2\operatorname{Tr}\phi^2\Bigg\},
\end{aligned}$$

where as it is conventional in Lattice Field Theory, following an analogy with statistical physics, we have defined the coefficient β for the case of SU(2) as

$$\beta = \frac{4}{g^2},$$

and means that with this relation, weak coupling implies large β.

As it stands however, the parameter m and field ϕ in the above action are dimensionful quantities. We can make them dimensionless by absorbing an 'a' term into them so that we can re-write the action in terms of $a\phi(x)$ and am instead.

That is, with the following transformation

$$\begin{aligned} \phi(x) &\rightarrow a\phi(x) \\ m &\rightarrow am \end{aligned}$$

the lattice action becomes

$$\begin{aligned} S_E = \sum_x \Bigg\{ & \beta \sum_{\mu<\nu} \left(1 - \frac{1}{2} P_{\mu\nu}\right) \\ + & \sum_\mu \left(2 \operatorname{Tr} \phi^2(x) - 2 \operatorname{Tr} \phi(x) U_\mu(x) \phi(x + a\hat{\mu}) U_\mu^\dagger(x)\right) \\ + & \lambda (\operatorname{Tr} \phi^2)^2 + \frac{(m^2)^2}{4\lambda} + m^2 \operatorname{Tr} \phi^2 \Bigg\}, \end{aligned} \tag{5.3}$$

where now all the parameters are dimensionless.

5.3 Boundary conditions

In previous sections we have argued that we can calculate the mass of the magnetic monopole by Monte Carlo methods by generating configurations in the vacuum and configurations that contain one magnetic monopole (sector

zero and sector one respectively). The question remains as to how we can actually generate those configurations in such a way that the existence or not of a magnetic monopole is guaranteed.

The way this is achieved is by a clever use of the boundary conditions on the lattice. Here, we just briefly summarise what the boundary conditions are in each case. Further details can be obtained from references [19], [20]and [17].

The following *C-periodic* boundary conditions are chosen to generate vacuum configurations (sector zero):

$$\begin{aligned} U_\mu(x+N\hat{\jmath}) &= \sigma_2 \; U_\mu(x) \; \sigma_2 = U^*_\mu(x) \\ \phi(x+N\hat{\jmath}) &= -\sigma_2 \; \phi(x) \; \sigma_2 = \phi^*(x) \end{aligned}$$

while the *twisted* boundary conditions below generate configurations with one magnetic monopole (sector one)

$$\begin{aligned} U_\mu(x+N\hat{\jmath}) &= \sigma_j \; U_\mu(x) \; \sigma_j \\ \phi(x+N\hat{\jmath}) &= -\sigma_j \; \phi(x) \; \sigma_j. \end{aligned}$$

Chapter 6

Results

The simulations were based on the following discretised Euclidean action as derived in chapter 5:

$$
\begin{aligned}
S_E = \sum_x \Bigg\{ & \beta \sum_{\mu<\nu} \left(1 - \frac{1}{2} \operatorname{Tr} U_\mu(x) U_\nu(x+\hat{\mu}) U_\mu^\dagger(x+\hat{\nu}) U_\nu^\dagger(x)\right) \\
- & 2 \sum_\mu \left(\operatorname{Tr} \phi(x) U_\mu(x) \phi(x+\hat{\mu}) U_\mu^\dagger(x)\right) \\
+ & (8+m^2) \operatorname{Tr} \phi^2(x) + \lambda \left(\operatorname{Tr} \phi^2(x)\right)^2 + \frac{(m^2)^2}{4\lambda} \Bigg\},
\end{aligned}
$$

where all parameters are on lattice units with lattice distance $a = 1$. This expression is obtained from (5.3) by substituting

$$\sum_\mu 2 \operatorname{Tr} \phi^2(x) = 8 \operatorname{Tr} \phi^2(x)$$

and regrouping terms.

The dimensionless Lagrangian parameters λ and β were set to the following values through all simulations:

$$
\begin{aligned}
\lambda &= 0.1 \\
\beta &= 20 \qquad \text{(corresponding to } g = 1/\sqrt{5}\text{).}
\end{aligned}
$$

The calculation of the monopole mass was repeated for lattice sizes (TL^3) of 16^4, 24^4, 32^4, 48^4 and 64^4.

For each lattice size the parameter m^2 was varied from around a value of -0.200 to a value of -1.800. This was repeated twice, once for C-periodic boundary conditions corresponding to configurations in the vacuum, and another for twisted boundary conditions corresponding to configurations with one magnetic monopole. These two sets of values can be plotted against m^2 to obtain two graphs (see fig. 6.1).

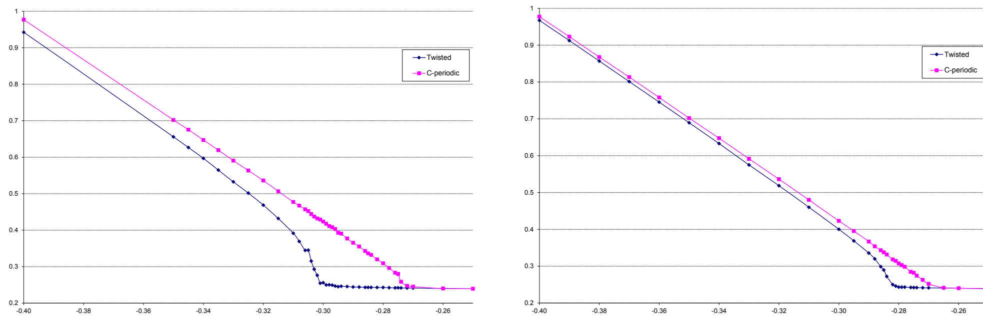


Figure 6.1: $\langle \mathrm{Tr}\, \phi^2 \rangle$ vs. m^2 for lattice sizes 16^4 (left) and 24^4 (right) near the critical mass m_c^2. The gap between the curves continues to reduce as we increase the lattice size, becoming almost undistinguishable for bigger lattices. It is the area between these two graphs that gives us the magnetic monopole mass M.

Each Metropolis simulation consisted of a total of 150,000 updates. The first $N_{therm} = 20,000$ updates were part of the initial thermalisation process. After this, measurements were then taken every $N_{cor} = 1,000$ updates, making a total of 130 measurements per simulation.

The acceptance/rejection rate, which is part of the Metropolis step, was tuned by the input parameters *rangeA* and *rangePhi*. These two parameters control how much the fields U and ϕ change between consecutive field

configurations. The following values were used for all simulations

$$\begin{aligned} rangeA &= 0.3 \\ rangePhi &= 0.2, \end{aligned}$$

which resulted in an acceptance rate of approximately 64% and 58% for U and ϕ respectively.

Therefore, for each combination of lattice size, mass m^2 and boundary conditions (c-periodic or twisted) we obtained an estimate for the expectation value density of the trace of ϕ^2, that is, $\langle \mathrm{Tr}\,\phi^2(x)\rangle/V$ where $V = TL^3$ is the volume of the lattice.

As the values calculated in the simulation are volume density quantities we need to adjust the expression for M obtained in (4.3) by multiplying by the volume V of the lattice, that is:

$$\frac{\partial M}{\partial m^2} = \frac{1}{T}\Big(\langle \mathrm{Tr}\,\phi^2\rangle_1 - \langle \mathrm{Tr}\,\phi^2\rangle_0\Big) = L^3\Big(\langle \mathrm{Tr}\,\phi^2\rangle_{tw} - \langle \mathrm{Tr}\,\phi^2\rangle_{per}\Big), \qquad (6.1)$$

where *tw* and *per* stand for twisted or periodic boundary conditions and are the quantities obtained in the simulation (volume densities), as opposed to the corresponding 1 and 0 values which are pure expectation values (not densities).

The integration of (6.1) is solved in terms of finite differences by approximating the above expression by

$$\frac{\Delta M}{\Delta m^2}(m^2) = L^3\Big(\langle \mathrm{Tr}\,\phi^2\rangle_{tw}(m^2) - \langle \mathrm{Tr}\,\phi^2\rangle_{per}(m^2)\Big)$$

and hence if m^2 takes values $m^2 = m_0^2 \dots m_{max}^2$ we then calculate M as follows:

$$\begin{aligned} M &= \sum_{k=1}^{max} \Delta M(m_k^2) \\ \Delta M(m_k^2) &= L^3\Big(\langle \mathrm{Tr}\,\phi^2\rangle_{tw}(m_k^2) - \langle \mathrm{Tr}\,\phi^2\rangle_{per}(m_k^2)\Big)\left[m_k^2 - m_{k-1}^2\right], \end{aligned}$$

where $\Delta M(m_0^2) = 0$, and for this to be the case we need to take $m_0^2 > m_c^2$ so that the system is in the unbroken phase and accordingly there is no magnetic monopole.

Figure 6.2 shows the value of the derivative of M with respect to m^2 for the different lattice sizes we have used in the simulations. This corresponds to the difference between the two curves in fig. 6.1 adjusted by a volume factor. Similarly to what we saw there, the lattice size has a big impact in the shape and size of the area between the curves, particularly for small latices.

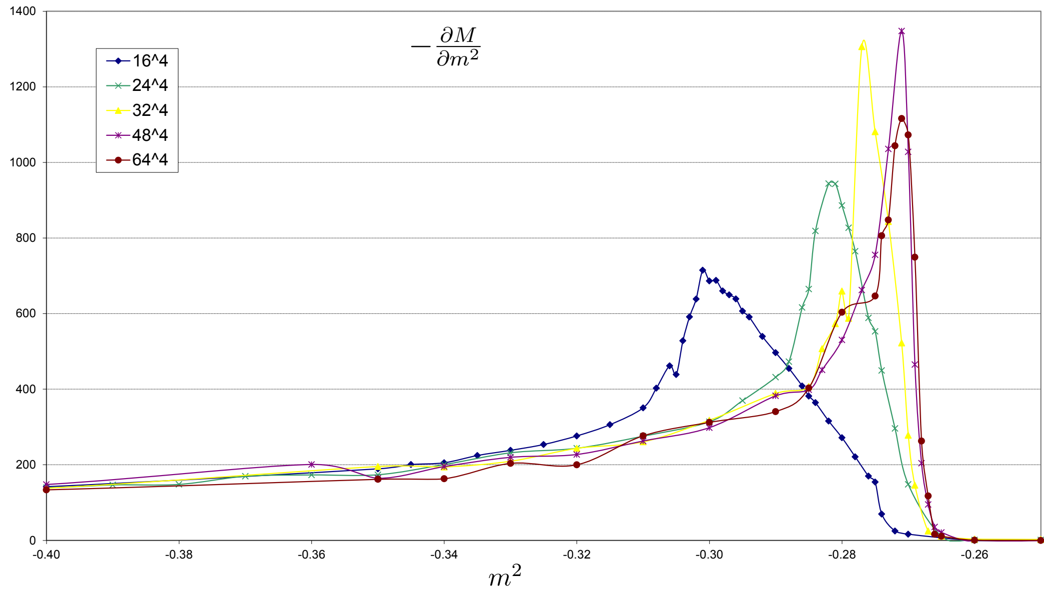


Figure 6.2: $\frac{\partial M}{\partial m^2}$ vs. m^2 for several lattice sizes.

The graph suggests a critical value of the mass parameter in lattice units of around $m_c^2 \approx 26.6$. It also shows how the peak of the graph moves to the right, i.e. to increasing values of m^2 as the lattice size increases.

The corresponding graph for M is shown in figure 6.3. We can observe

that the larger the lattice size, the greater the initial slope off the critical point m_c^2. This is due to the earlier contribution to M from the peak in the previous graph.

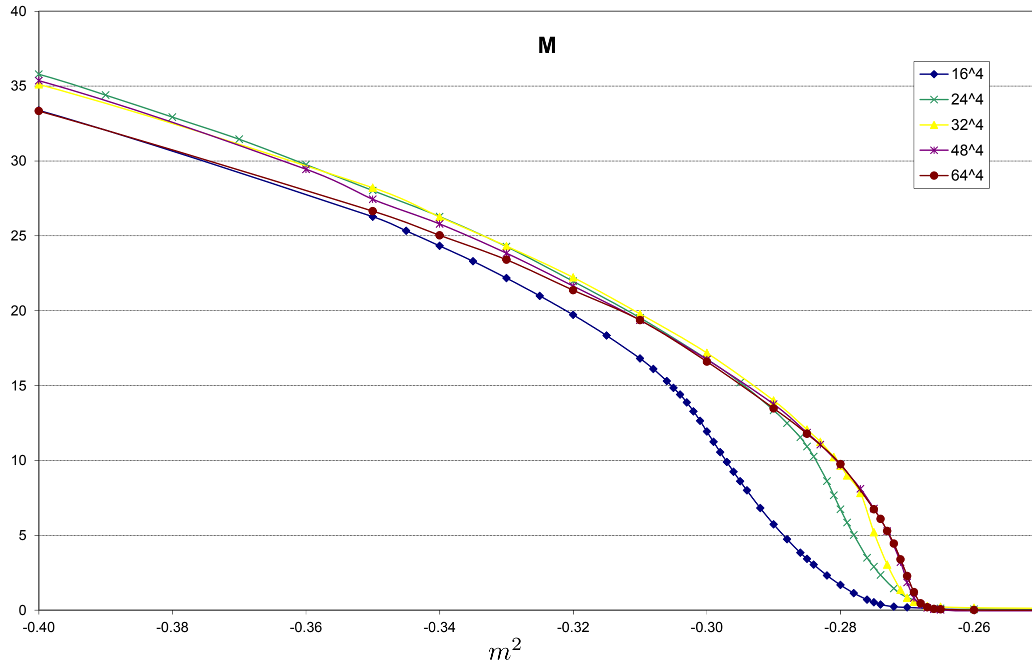


Figure 6.3: M vs. m^2 near the critical mass for several lattice sizes.

This can be appreciated better in the next graph, fig. 6.4, which provides a close up near the critical point.

At the same time, in general, at lower values of m^2, i.e. more negative, the slope tends to fall off quicker the larger the lattice, so that as we move further into the broken phase the curves seem to settle down to lower values of M the larger the lattice (see fig. 6.4). An exception to this is the case for lattice size 16^4 which has similar final values to 64^4 although this is probably due to the slow initial slope perhaps because of larger distortion of the monopole in smaller lattices. Lattice 16^4, 48^4 and 64^4 settle around a value of $M = 80-81$ while lattices 24^4 and 32^4 to a value around $M = 85 - 87$

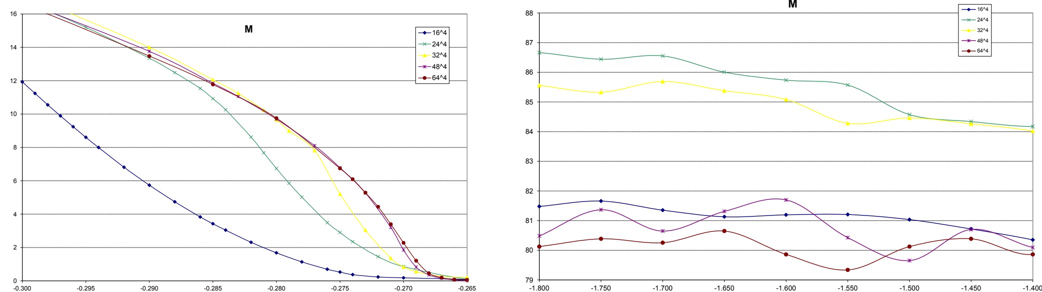


Figure 6.4: M vs. m^2 near the critical point (left) and deep in the broken phase (right).

Finally fig. 6.5 provides a full view of the calculated graph for M for all latice sizes.

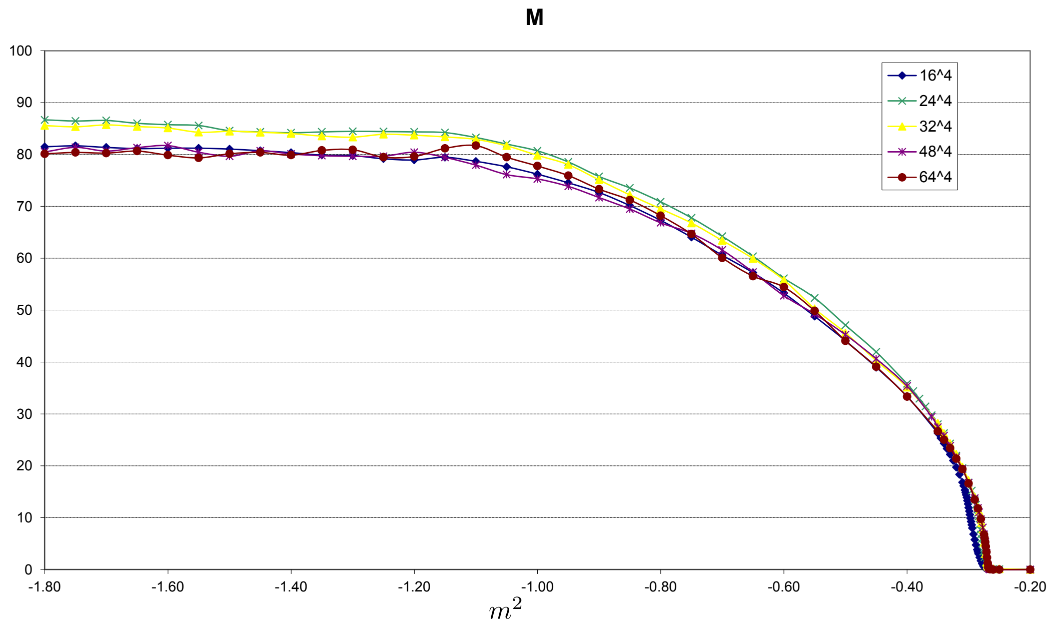


Figure 6.5: M vs. m^2. Zoom out view.

6.1 Error calculations

Each computer simulation (which took from 10 minutes for a 16^4 lattice to over two days for the 64^4 case) provided us with a point in the graph of figure 6.1 that was obtained by averaging a set of 130 measurements of $\mathrm{Tr}\,\phi^2$ on

the lattice. This set of measurements was used as input to the bootstrap algorithm, for it to re-sample from, and produce an estimate of the error in the result, i.e. the standard deviation in the value of $\langle \mathrm{Tr}\, \phi^2 \rangle$.

The error obtained in this way had a maximum value of under 0.10% for C-periodic boundary conditions and 0.22% for twisted boundary conditions.

These errors were then propagated to the calculation of $\frac{\partial M}{\partial m^2}$ and M. Figure 6.6 shows the curve for M with the corresponding error bars for lattice sizes 16^4 and 64^4.

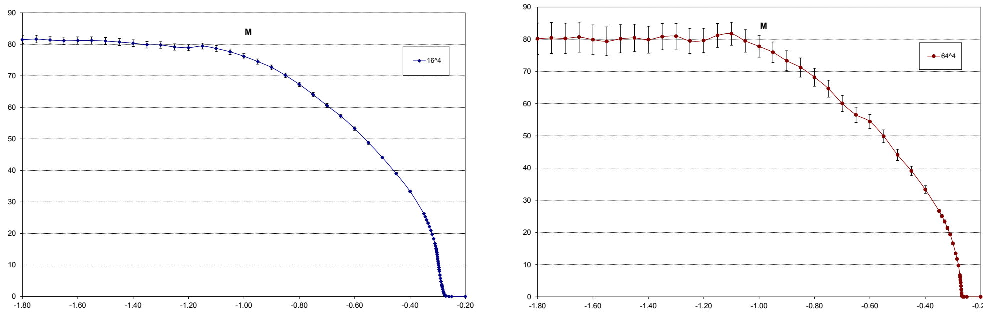


Figure 6.6: M vs. m^2 with error bars for a 16^4 (left) and 64^4 (right) lattice.

It is observed that the errors increase the further away from the critical mass point. This is probably largely due to the propagation of errors in the formulas as the number of terms in the sum increases in the calculation of M. Maybe, an alternative error calculation method could have been to use the additional data sets produced by the bootstrap method to not only calculate additional values of $\langle \mathrm{Tr}\, \phi^2 \rangle$ but also to calculate the corresponding values of $\frac{\partial M}{\partial m^2}$ and M and then combine these additional values of M to calculate its standard deviation.

The errors also increased with the lattice size.

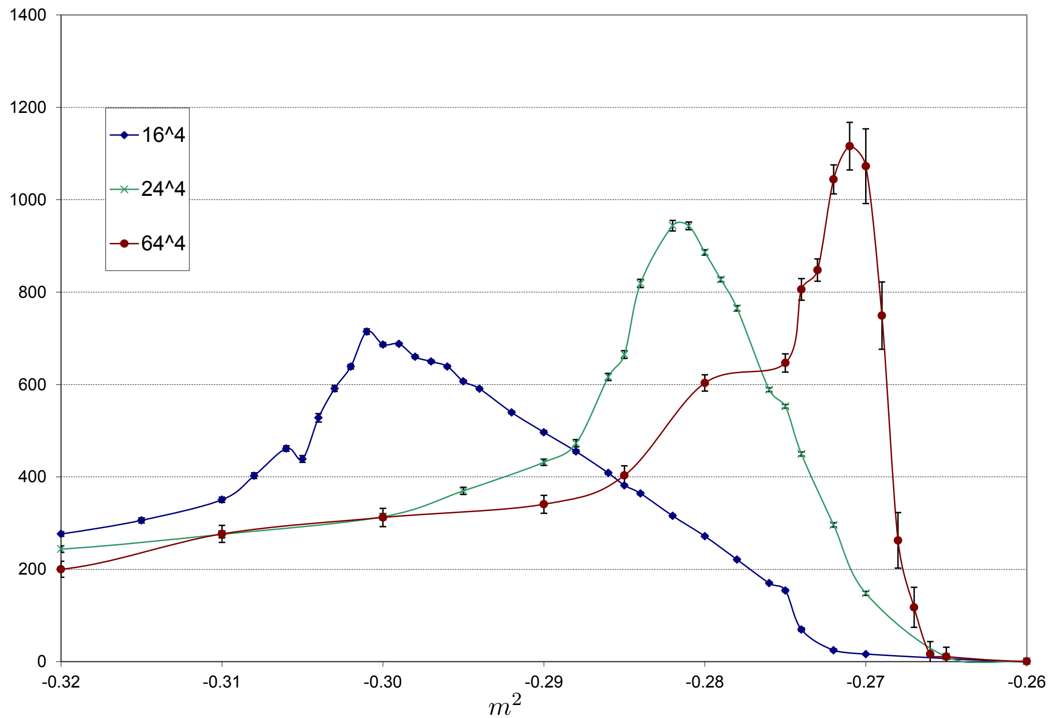


Figure 6.7: $\frac{\partial M}{\partial m^2}$ vs. m^2 with error bars. Some lattice sizes have been removed from the graph to avoid cluttering.

6.2 Other simulations

All the simulations in previous sections were carried out with a Lagrangian parameter value of $\beta = 20$. Here, we repeat the same simulations but instead using $\beta = 4$, which corresponds to a coupling constant of $g = 1$. This no longer corresponds to weak coupling.

Due to time constraints we limited the simulations to a lattice size of 16^4. The results are shown in figures 6.8 and 6.9.

We initially run simulations for widely separated values of m^2 in order to determine the location of the critical mass m_c^2 as this could potentially be different than for the weak coupling case, and indeed it was, with a value of $m_c^2 \approx 1$. Then we slowly increased the number of intermediate points around this value to achieve better accuracy.

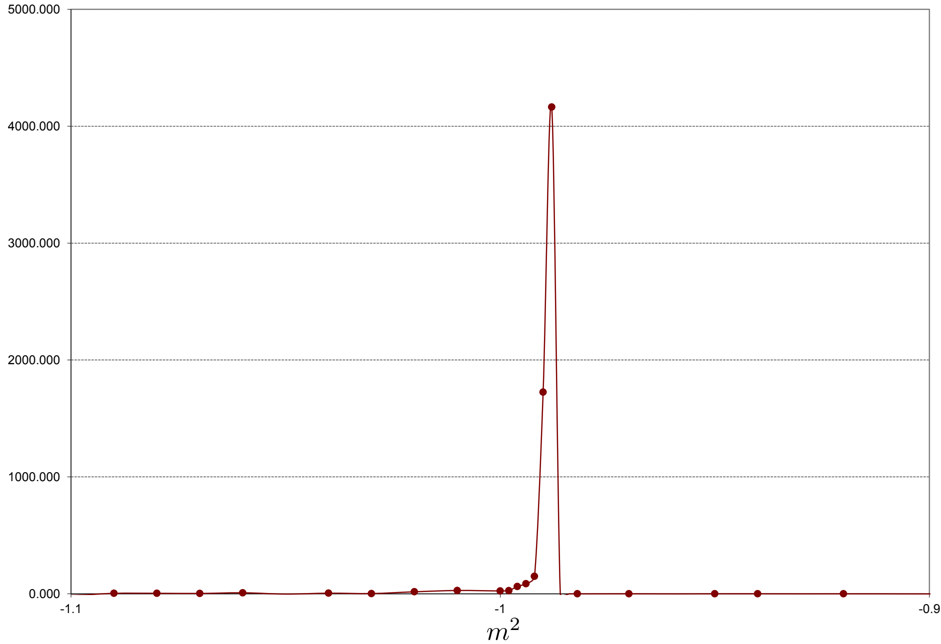


Figure 6.8: $\frac{\partial M}{\partial m^2}$ vs. m^2 for $\beta = 4$ and lattice size 16^4.

We see that the critical point has shifted considerably to the left compared to the weakly coupled case from earlier. Also the slope of the curves have become much more sharper. It would maybe be interesting to expand on this and study the behaviour of the critical point as we vary β.

Apart from the variation in β we also carried out simulations where we kept L fixed while we increased T, in order to investigate the effect of having a lattice that was longer in time than in the spatial dimensions. We collected results for the case of $L = 24$ while T took values 24, 48, 96 and 192, however we did not find major differences in the results, in particular close to the critical point. Further away, the value of M again seemed to take lower values the larger the size of the lattice, T in this case. However all results were within error tolerance of each other.

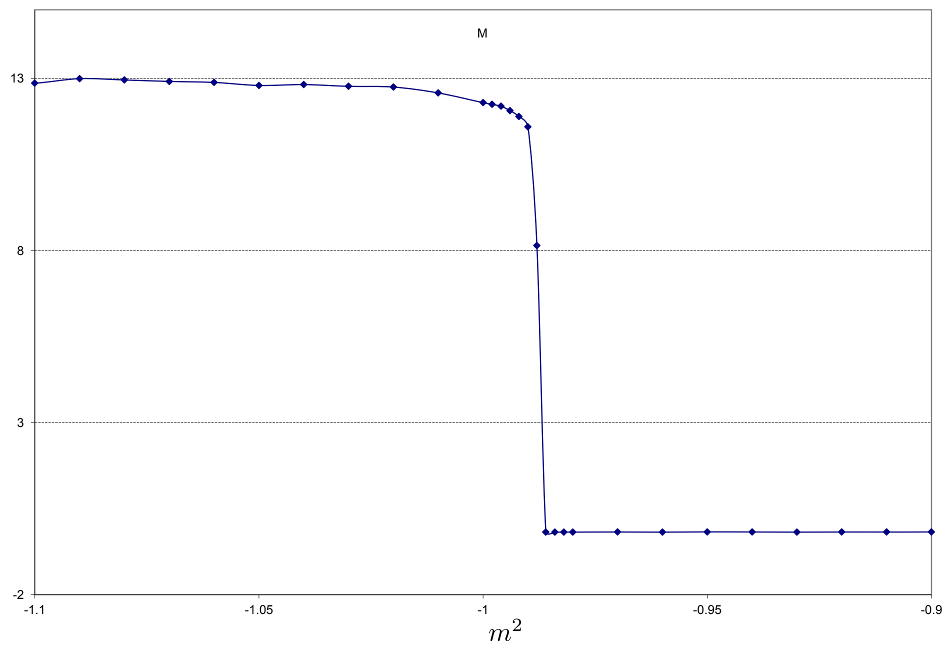


Figure 6.9: M vs. m^2 for $\beta = 4$ and lattice size 16^4.

Chapter 7

Conclusions and further study

In this document we have seen how to calculate the mass of the 't Hooft-Polyakov monopole for the Georgi-Glashow model in 3+1 dimensions. The calculations were carried out non-perturbatively by using a lattice on which to discretise the theory. The values obtained for different lattice sizes agree with the results observed in [20].

The Georgi-Glashow theory we have investigated has two coupling constants, λ and g. In this document we have looked at the case where these are weak ($\lambda = 0.1$ and $g = 1/\sqrt{5}$) but it would be interesting to understand how things work when we go to strong couplings.

The reason for using a weak λ is because in this case the theory is closer to classical monopoles, and then we know what to expect, so we can compare results. But now that we know that the method we have used works at weak coupling it would be interesting to look at the behaviour of the theory when we increase λ and its effects on the magnetic monopole mass.

As regards to the coupling g, if we think of the gauge field A_μ as in electromagnetism, i.e. under U(1) symmetry, then g would correspond to

the elementary electric charge e which we know to be small and hence in real physics we know that the coupling is weak in this case.

However, in the non-abelian case, g corresponds to the GUT scale coupling and in this case we do not know whether this is strong or weak. Therefore, it would be interesting to study what happens when g becomes large.

Apart from some numerical constant (which depends on the gauge group) g gives us roughly the electric charge, $g \approx e$, and Dirac showed [1] that if e is the elementary electric charge then in order for quantum mechanics to be consistent we must have that the elementary magnetic charge q satisfies the relation

$$q = \frac{2\pi}{e} \approx \frac{2\pi}{g}. \qquad \text{(Dirac condition)}$$

This means that as we increase g we make the magnetic charge q weaker and eventually we get to the regime where the magnetic monopoles are weakly coupled (q small) while the electric charge is strongly coupled. In this regime we can not do normal perturbation theory as the coupling e is strong, however, we can explore the possibility that there might be an approximate duality between electric and magnetic charges.

In supersymmetric theories, there are arguments that show a duality which allows to swap electric and magnetic charges and hence allows to reformulate the theory in terms of the weakly coupled magnetic charges degrees of freedom (instead of the strongly coupled electric ones). This then allows the use of the standard perturbation methods for the calculations. Strictly speaking though, this duality is only true in supersymmetric theories and the theory we are considering in this document is not supersymmetric. However, even though no exact duality then exists, we could still ask whether some element of duality is still present if we go to the strong coupling limit in g.

That is, we could check whether the behaviour of the system in that limit still actually looks the same as in weak coupling but with electric and magnetic fields interchanged. If this was the case then it would suggest that actually the physics that we would find would be very simple as it should then be similar to the weak coupling case except that it would be described in terms of magnetic degrees of freedom instead of electric ones.

A further piece of study that could be carried out is to compare how the masses of the electrically and magnetically charged particles behave. In the broken phase of the theory, there remains an unbroken U(1) symmetry, so there is a massless photon associated to it. In addition, there are another two components of the gauge field, which obtain a mass through the Higgs mechanism. These correspond to two charged bosons which we can call $W^{\pm}$. Finally, and apart from the magnetic monopole, there is a neutral massive Higgs scalar H.

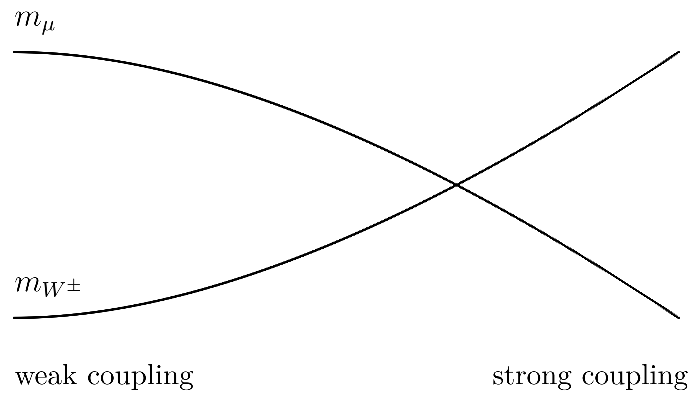
At weak coupling, where the semi-classical picture works, the mass of these particles are given, to leading order, by

$$\begin{aligned} m_{W^{\pm}} &= g\nu \\ m_H &= \sqrt{2\lambda}\nu. \end{aligned}$$

The mass of the classical magnetic monopole is, again to leading order,

$$m_{\mu} = \frac{\nu}{g}.$$

Looking at the masses $m_{W^{\pm}}$ and m_{μ} we see that under weak coupling, as g is small, the electrically charged particles $W^{\pm}$ are much lighter than the magnetically charged particles, that is, $m_{W^{\pm}} < m_{\mu}$.



When we go to strong coupling though, they swap places and the magnetically charged particles become the lighter ones. Of course, at the same time, the theory becomes non-perturbative and hence we can not trust the above formulas for the mass any more. But qualitatively we see that, at least initially, the magnetic monopoles become important degrees of freedom because they are the light particles while the electrically charged particles become heavy and therefore irrelevant. So an interesting study would be to monitor the masses of these particles as we increase g and check whether, as the formulas predict, their masses meet and cross at a point.

Appendix A

The simulation code

A.1 Compilation and execution

The simulation code[1] is written in Fortran and has been compiled and run using the facilities at *Imperial HPC services* [21].

The code is parameterised in order to generate several executables from the same code base. For the simulations two executables were created for each lattice size, one with C-periodic and another with twisted boundary conditions.

Four external variables can be defined at compile time with the following meaning:

tsize number of lattice nodes in the time direction.

size number of lattice nodes in each of the three spatial dimensions.

twist use twisted boundary conditions.

[1]The simulation code was generously provided by my supervisor, Prof. Arttu Rajantie. This was then customised to the task at hand by: deleting unused execution paths, adding input parameters to increase flexibility and avoid the need for multiple executables, modifying program output to aid in post-processing of results through linux shell scripts and Excel spreadsheets and annotating the code.

Cper use C-periodic boundary conditions.

Note though that only one of *twist* or *Cper* should be defined for any given compilation.

For example, for a 24^4 lattice size with *twisted* boundary conditions the following instructions were used:

```
ifort -Dtsize="24" -Dsize="24" -Dtwist -openmp -ip -O3
      -auto -Isource -fpp source/monopole.F
      -o bin/mono_tw24 -mkl=parallel -lmkl_core
```

When the program is executed it asks for six input parameters:

beta, m2, lambda lagrangian parameters in lattice units.

nMax total number of updates for the Metropolis algorithm. Measurements are taken every 1000 updates.

rangeA, rangePhi control by how much U and ϕ are changed at each Metropolis update. These can be used to tune the acceptance/rejection rate.

The output of the simulation code consists of a table with a measurement of $\mathrm{Tr}\,\phi^2(x)$ taken every 1000 Metropolis updates. These measurements can then be averaged to produce an estimate for $\langle \mathrm{Tr}\,\phi^2(x)\rangle$, although the first 20 entries from the table are ignored as these are part of the thermalisation process, corresponding to the first 2000 updates.

A.2 Field representation

The simulation code mainly revolves around the logic to update the scalar field $\phi(x)$ and the link variables $U(x)$.

The scalar field $\phi(x)$ is declared in code as a 3-component vector (1:3) for each space-time lattice node:

```
DOUBLE PRECISION, DIMENSION(:,:,:,:,:), ALLOCATABLE :: phi
ALLOCATE(phi(1:3,1:hx,1:hy,1:hz,1:ht))
```

This vector encodes a 2×2 traceless hermitian matrix as follows:

$$\phi = [\phi_1, \phi_2, \phi_3] \to \begin{bmatrix} \phi_1 & \phi_3 + i\phi_2 \\ \phi_3 - i\phi_2 & -\phi_1 \end{bmatrix}.$$

Similarly the link variable U is declared in code as a 4-component vector (0:3) in each of the four directions (1:4) per space-time point in the lattice:

```
DOUBLE PRECISION, DIMENSION(:,:,:,:,:,:), ALLOCATABLE :: u
ALLOCATE(u(0:3,1:4,1:hx,1:hy,1:hz,1:ht))
```

The link variable U is a member of the SU(2) group and its four components are encoded in a vector (0:3) as

$$U \in SU(2) = [u_0, u_1, u_2, u_3] \to \begin{bmatrix} u_0 + iu_1 & -u_2 + iu_3 \\ u_2 + iu_3 & u_0 - iu_1 \end{bmatrix} = \begin{bmatrix} \alpha & -\bar{\beta} \\ \beta & \bar{\alpha} \end{bmatrix}.$$

All code subroutines that manipulate matrices (e.g. `trmul`, `matmul3`, ...) assume this same representation for any SU(2) matrix.

A.3 Annotated code

The Fortran code used for the simulation is included below. The version included here has been trimmed down and simplified to avoid unnecessary cluttering while maintaining the core of the simulation algorithm.

In particular, the code does not include:

- code for logging of errors.

- code to save status files with information on input parameters, ratio of success of the Metropolis update (which was needed for the adjustment of input parameters *rangeA* and *rangePhi*), number of updates, ...
- routines for post-processing of results.

```
      include 'mkl_vsl.fi'

      USE mkl_vsl
      USE mkl_vsl_type
      IMPLICIT NONE

! ----------------------------------------------------------------
! ---------------- Data Declarations ------------------------------
! ----------------------------------------------------------------

      ! tsize and size defined externally
      INTEGER ht,hx,hy,hz
      PARAMETER(ht=tsize,hx=size,hy=size,hz=size)

      INTEGER dum,mx(-hx:2*hx),my(-hy:2*hy),mz(-hz:2*hz),mt(-ht:2*ht)
      PARAMETER(mx= (/ (MOD(dum+2*hx-1,hx)+1,dum=-hx,2*hx) /) )
      PARAMETER(my= (/ (MOD(dum+2*hy-1,hy)+1,dum=-hy,2*hy) /) )
      PARAMETER(mz= (/ (MOD(dum+2*hz-1,hz)+1,dum=-hz,2*hz) /) )
      PARAMETER(mt= (/ (MOD(dum+2*ht-1,ht)+1,dum=-ht,2*ht) /) )

      DOUBLE PRECISION beta,m2,lambda
      DOUBLE PRECISION, DIMENSION(:,:,:,:,:,:), ALLOCATABLE :: u
      DOUBLE PRECISION, DIMENSION(:,:,:,:,:), ALLOCATABLE :: phi
      REAL, DIMENSION(:,:,:,:,:,:), ALLOCATABLE :: rnd
      REAL, DIMENSION(:,:,:,:,:,:,:), ALLOCATABLE :: rndu

      INTEGER n,t,i,j,k,m,dir,nMax

      DOUBLE PRECISION r
      INTEGER seed,s(2)

      TYPE (VSL_STREAM_STATE) :: stream(ht)
      DOUBLE PRECISION rangeA,rangephi
      DOUBLE PRECISION action
      CHARACTER(100) datadir, execName
      INTEGER count0,count_rate,count_max
      INTEGER errcode

      CALL getarg(0,execName)
      CALL getarg(1,datadir)
      datadir = TRIM(datadir)//'/XXX_'

      ALLOCATE(u(0:3,1:4,1:hx,1:hy,1:hz,1:ht))
      ALLOCATE(phi(1:3,1:hx,1:hy,1:hz,1:ht))
      ALLOCATE(rnd(4,10,hx,hy,hz,ht))
      ALLOCATE(rndu(2,6,4,hx,hy,hz,ht))

! ----------------------------------------------------------------
! ---------------- Main Code --------------------------------------
! ----------------------------------------------------------------
```

```
      ! Initialise random number seed
      CALL SYSTEM_CLOCK(count0,count_rate,count_max)
      seed=count0
      s(1)=seed/65536
      s(2)=seed-65536*s(1)

      CALL RANDOM_SEED (PUT = S)

          ! Initialise random number streams
      DO t=1,ht
         CALL random_number(r)
         seed=NINT(65536*r)
         errcode=vslnewstream( stream(t), VSL_BRNG_MCG31, seed )
         WRITE(*,*) errcode
      ENDDO

          ! Read input parameters
      WRITE(*,*) 'm2, lambda, beta? (lattice units)'
      READ(*,*)m2,lambda,beta

      WRITE(*,*) 'nMax, rangeA, rangephi? '
      READ(*,*)nMax,rangeA,rangephi

      WRITE(*,*) 'm2=',m2,' lambda=',lambda,' beta=',beta
      WRITE(*,*) 'nMax=',nMax,' rangeA=',rangeA,' rangephi=',rangephi

      CALL initialise(SQRT(-m2),SQRT(-m2/lambda))

      OPEN(10,file=datadir//'conf',form='unformatted',err=10,
     +     status='old')
      READ(10)u
      READ(10)phi
      CLOSE(10)

 10

          ! Generate full set of random numbers but throw them away
      ! to avoid problem with first values

      DO t=1,ht
         errcode=vsrnguniform (0, stream(t), 2*6*4*hx*hy*hz,
     +        rndu(:,:,:,:,:,:,t), 0.0, 1.0)
         errcode=vsrnguniform (0, stream(t), 4*10*hx*hy*hz,
     +        rnd(:,:,:,:,:,t), 0.0, 1.0)
      ENDDO

          ! Perform time evolution (Metropolis updates)
          ! and take measurements every 100 updates
      DO n=1,nMax

!$OMP PARALLEL DO

                ! Generate full set of random numbers
                ! These will be used to update configurations
         DO t=1,ht
            errcode=vsrnguniform (0, stream(t), 2*6*4*hx*hy*hz,
     +           rndu(:,:,:,:,:,:,t), 0.0, 1.0)
            errcode=vsrnguniform (0, stream(t), 4*10*hx*hy*hz,
     +           rnd(:,:,:,:,:,t), 0.0, 1.0)
         ENDDO
```

```
                 ! Update link variables U
                 ! first odd lattice nodes then the even ones
          CALL update(0,beta,m2,lambda,rangeA)
          CALL update(1,beta,m2,lambda,rangeA)

                 ! Similarly update field phi (odd and even)
          CALL updatephi(0,m2,lambda,rangephi)
          CALL updatephi(1,m2,lambda,rangephi)

                 ! Take measurement of observables every 100 updates
          IF (MOD(n,100).EQ.0) THEN
             CALL measure(datadir,m2)
          ENDIF
       ENDDO

       CONTAINS

! ------------------------------------------------------------
! ---------------- Subroutines ------------------------------
! ------------------------------------------------------------
             ! ------------------------------------------------------
       ! Initialise phi depending on boundary conditions
             ! ------------------------------------------------------
       SUBROUTINE initialise(mass,vev)
       IMPLICIT NONE
       DOUBLE PRECISION mass,vev
       INTEGER i,j,k,t
       DOUBLE PRECISION r,dx,dy,dz

!$OMP PARALLEL DO
!$OMP+PRIVATE(i,j,k,dx,dy,dz,r)
       DO t=1,ht
          DO k=1,hz
             DO j=1,hy
                DO i=1,hx
#ifdef twist
                   dx=REAL(2*i-hx-1)/2.0
                   dy=REAL(2*j-hy-1)/2.0
                   dz=REAL(2*k-hz-1)/2.0
                   r=SQRT(dx**2+dy**2+dz**2)

                   phi(:,i,j,k,t)=vev*(1.0-EXP(-mass*r))/r*(/dx,dy,dz/)
#else
                   phi(:,i,j,k,t)=(/ vev,0d0,0d0 /)
#endif
                ENDDO
             ENDDO
          ENDDO
       ENDDO
       END SUBROUTINE initialise


             ! ------------------------------------------------------
       ! Update link variable U by
             ! performing 6 sweeps of the metropolis algorithm
             ! Updates only odd/even lattice nodes
             ! Node is odd if coords i+j+k+t odd (otherwise even)
             ! ------------------------------------------------------
       SUBROUTINE update(odd,beta,mass2,lambda,rangeA)
       IMPLICIT NONE

       INTEGER odd
```

```
      DOUBLE PRECISION beta,mass2,lambda
      DOUBLE PRECISION rangeA

      INTEGER t,i,j,k,dir,d2,n,pos(4)
      INTEGER posp(4),posp2(4),posm2(4),pospm2(4)
      DOUBLE PRECISION ,DIMENSION(3) :: phi1,phi2
      DOUBLE PRECISION ,DIMENSION(0:3) :: oldlink
      DOUBLE PRECISION ,DIMENSION(0:3) :: surr,newlink
      DOUBLE PRECISION oldact,r,datot
      DOUBLE PRECISION delact
      INTEGER accept,maxAccept
      REAL acceptRate

      accept=0
      datot=0.0
      DO dir=1,4

!$OMP PARALLEL DO
!$OMP+PRIVATE(i,j,k,d2,n,oldlink,newlink,delact,surr,r)
!$OMP+PRIVATE(oldact,phi1,phi2)
!$OMP+PRIVATE(pos,posp,posp2,pospm2,posm2)
!$OMP+REDUCTION(+:accept,datot)

        DO t=1,ht
          DO k=1,hz
            DO j=1,hy
              DO i=1+MOD(odd+j+k+t,2),hx,2
                pos=(/ i,j,k,t /)
                posp=posplus(dir,pos)
                oldlink=readu4(dir,pos)
                surr=(/ 0.0,0.0,0.0,0.0 /)
                DO d2=1,4
                  IF (d2.NE.dir) THEN
                    posp2=posplus(d2,pos)
                    pospm2=posminus(d2,posp)
                    posm2=posminus(d2,pos)
                    surr=surr+matmul3nhh(readu4(d2,posp),
     +                     readu4(dir,posp2),readu4(d2,pos))
                    surr=surr+matmul3hhn(readu4(d2,pospm2),
     +                     readu4(dir,posm2),readu4(d2,posm2))
                  ENDIF
                ENDDO

                                          ! Calculate OldAction
                phi1=readphi(pos)
                phi2=readphi(posp)
                oldact=-beta/2d0*trmul(oldlink,surr)
     +               -4d0*dotprod(phi1,mrot(oldlink,phi2))

                                          ! Update 6 times, each combination of directions
                                          ! mu, nu with mu < nu
                DO n=1,6
                  newlink=gennewlink(n,oldlink,rangeA,
     +                rndu(1,n,dir,i,j,k,t))

                                                    ! Calculate DeltaAction = NewAction -
                                                       OldAction
                                                    delact=-beta/2d0*trmul(newlink,surr)
     +               -4d0*dotprod(phi1,mrot(newlink,phi2))
     +               -oldact

                                             ! Metropolis step: accept/reject
```

```
                        IF (delact.GT.0.0) THEN
                           r=rndu(2,n,dir,i,j,k,t)
                           IF (r.LT.EXP(-delact)) THEN
                              oldlink=newlink
                              oldact=oldact+delact
                              accept=accept+1
                              datot=datot+delact
                           ENDIF
                        ENDIF

                        IF (delact.LE.0.0) THEN
                           oldlink=newlink
                           oldact=oldact+delact
                           accept=accept+1
                           datot=datot+delact
                        ENDIF
                     ENDDO

                                          ! Update U with new value
                     u(:,dir,i,j,k,t)=oldlink
                  ENDDO
               ENDDO
            ENDDO
         ENDDO
      ENDDO

          ! Calculate acceptance rate
      maxAccept = 6*2*hx*hy*hz*ht
          acceptRate = REAL(accept)/REAL(maxAccept)

c       WRITE(*,*)'U: Accepted ',accept,' / ', maxAccept,'=', acceptRate

      END SUBROUTINE update


          ! -----------------------------------------------
          ! Generate next configuration of field phi by
          ! performing 10 sweeps of the metropolis algorithm
          ! Updates only odd/even lattice nodes
          ! Node is odd if coords i+j+k+t odd (otherwise even)
          ! -----------------------------------------------
      SUBROUTINE updatephi(odd,m2,lambda,rangephi)
      IMPLICIT NONE

      INTEGER odd
      DOUBLE PRECISION m2,lambda
      DOUBLE PRECISION rangephi
      INTEGER i,j,k,t,dir
      DOUBLE PRECISION newphi(3),chphi(3),oldphi(3),surr(3)
      DOUBLE PRECISION r,delact,oldact,op2,np2,datot
      INTEGER pos(4),posp(4),posm(4)
      INTEGER n
      INTEGER accept,maxAccept
      REAL acceptRate

       datot=0.0
       accept=0
!$OMP PARALLEL DO PRIVATE(i,j,k,dir,surr,chphi,newphi,oldphi,n)
!$OMP+PRIVATE(delact,oldact,r)
!$OMP+PRIVATE(op2,np2)
!$OMP+PRIVATE(pos,posp,posm)
!$OMP+REDUCTION(+:accept,datot)
```

```
      DO t=1,ht
         DO k=1,hz
            DO j=1,hy
               DO i=1+MOD(odd+j+k+t,2),hx,2
                  surr=(/ 0.0,0.0,0.0 /)
                  pos=(/i,j,k,t/)

                                    ! For each point x=pos and direction mu=dir
                                    ! calculate -2*[U(x)*phi(x+mu)*U+(x) +
                                      U+(x-mu)*phi(x-mu)*U(x-mu)]
                                    ! i.e. contribution to the action of phi(x)
                                    ! To simplify this task we separate the updates of odd
                                      and even
                                    ! lattice nodes.
                  DO dir=1,4
                     posp=posplus(dir,pos)
                     surr=surr-2.0*mrot(readu4(dir,pos),readphi(posp))
                     posm=posminus(dir,pos)
                     surr=surr-2.0*mroth(readu4(dir,posm),readphi(posm))
                  ENDDO
                  oldphi=readphi(pos)

                                    ! Calculate old action
                  op2=2.0*squared(oldphi)
                  oldact=2.0*dotprod(oldphi,surr)
     +                   +(8.0+m2)*op2+lambda*op2**2

                                    ! Repeat update 10 times
                  DO n=1,10
                                             ! Calculate new phi(x)
                     chphi=2.0*rangephi*(rnd(1:3,n,i,j,k,t)-.5)
                     newphi=oldphi+chphi

                                             ! Calculate DeltaAction = NewAction - OldAction
                     np2=2.0*squared(newphi)
                     delact=2.0*dotprod(newphi,surr)
     +                      +(8.0+m2)*np2+lambda*np2**2
     +                      -oldact

                                             ! Metropolis step: accept/reject
                     IF (delact.GT.0.0) THEN
                        IF (LOG(rnd(4,n,i,j,k,t)).LT.-delact) THEN
                           accept=accept+1
                           oldact=oldact+delact
                           oldphi=newphi
                           datot=datot+delact
                        ENDIF
                     ENDIF

                     IF (delact.LE.0.0) THEN
                        accept=accept+1
                        oldact=oldact+delact
                        oldphi=newphi
                        datot=datot+delact
                     ENDIF
                  ENDDO

                                    ! Update phi
                  phi(:,i,j,k,t)=oldphi
               ENDDO
            ENDDO
         ENDDO
```

```
      ENDDO

         ! Calculate acceptance rate
         maxAccept = 10*hx*hy*hz*ht/2
         acceptRate = REAL(accept)/REAL(maxAccept)

c        WRITE(*,*)'Phi: Accepted ',accept,' / ',maxAccept,'=',acceptRate

      END SUBROUTINE updatephi


         ! ----------------------------------------------------
         ! Calculate <Tr phi^2>/Volume and append to
         ! output file "trace2"
         ! ----------------------------------------------------
      SUBROUTINE measure(path,m2)
      IMPLICIT NONE

      CHARACTER(100) path
      DOUBLE PRECISION m2
      INTEGER i,j,k,t,pos(4)
      DOUBLE PRECISION phi2, avgphi2

      avgphi2=0.0

!$OMP PARALLEL DO
!$OMP+PRIVATE(i,j,k,pos,phi2)
!$OMP+REDUCTION(+:,avgphi2)
      DO t=1,ht
         DO k=1,hz
            DO j=1,hy
               DO i=1,hx
                  pos=(/i,j,k,t/)
                                    ! phi2 = Tr phi(x)^2
                  phi2=2.0*squared(readphi(pos))
                  avgphi2=avgphi2+phi2
               ENDDO
            ENDDO
         ENDDO
      ENDDO

         ! Output total Tr phi^2 divided by volume of lattice
      OPEN(10,file=TRIM(path)//'trace2',position='APPEND')
      WRITE(10,"(F10.4,A1,F10.4)") REAL(avgphi2/hx/hy/hz/ht),' ',m2
      CLOSE(10)

      END SUBROUTINE measure

         ! ----------------------------------------------------
         ! plaq2(mu,nu,x)=(1/2)*Trace[Umu(x)Unu(x+mu)Umu+(x+nu)Unu+(x)]
         ! ----------------------------------------------------
      DOUBLE PRECISION FUNCTION plaq2(dir,d2,pos)
      IMPLICIT NONE
      INTEGER dir,d2,pos(4)
      DOUBLE PRECISION ,DIMENSION(0:3) :: m1,m2,u1,u2,u3,u4

      u1=readu4(dir,pos)
      u3=readu4(d2,pos)
      u2=readu4(d2,posplus(dir,pos))
      u4=readu4(dir,posplus(d2,pos))
      m1=matmul(u1,u2)
      m2=matmul(u3,u4)
```

```
      plaq2=trmulnh(m1,m2)/2.0
      END FUNCTION plaq2

          ! ------------------------------------------------
          ! link2 (mu, x) = Trace[phi(x)U(x)phi(x+mu)U+(x)]
          ! ------------------------------------------------
      DOUBLE PRECISION FUNCTION link2(dir,pos)
      IMPLICIT NONE
      INTEGER dir,pos(4)
      DOUBLE PRECISION, DIMENSION(3) :: ph1,ph2
      DOUBLE PRECISION, DIMENSION(0:3) :: u1

      ph1=readphi(pos)
      u1=readu4(dir,pos)
      ph2=readphi(posplus(dir,pos))
      link2=2.0*dotprod(ph1,mrot(u1,ph2))

      END FUNCTION link2

          ! ------------------------------------------------
          ! Returns the value of the SU(2) link variable U
          ! at position pos and direction dir.
          ! Enforces boundary conditions when pos outside
          ! bounds of lattice
          ! ------------------------------------------------
      FUNCTION readu4(dir,pos)
      IMPLICIT NONE
      DOUBLE PRECISION readu4(0:3),u1(0:3)
      INTEGER dir,i,j,k,t,pos(4)
      INTEGER i0,j0,k0,t0
      i0=mx(pos(1))
      j0=my(pos(2))
      k0=mz(pos(3))
      t0=mt(pos(4))
      u1=u(:,dir,i0,j0,k0,t0)
#ifdef Cper
      IF (pos(1).NE.i0) THEN
         u1(1)=-u1(1)
         u1(3)=-u1(3)
      ENDIF
      IF (pos(2).NE.j0) THEN
         u1(1)=-u1(1)
         u1(3)=-u1(3)
      ENDIF
      IF (pos(3).NE.k0) THEN
         u1(1)=-u1(1)
         u1(3)=-u1(3)
      ENDIF
#endif Cper
#ifdef twist
      IF (pos(1).NE.i0) THEN
         u1(2)=-u1(2)
         u1(3)=-u1(3)
      ENDIF
      IF (pos(2).NE.j0) THEN
         u1(1)=-u1(1)
         u1(3)=-u1(3)
      ENDIF
      IF (pos(3).NE.k0) THEN
         u1(1)=-u1(1)
         u1(2)=-u1(2)
      ENDIF
```

```
#endif twist
       readu4=u1
       END FUNCTION readu4

            ! ------------------------------------------------------
            ! Returns the value of the 3-component vector phi
            ! at position pos.
            ! Enforces boundary conditions when pos outside
            ! bounds of lattice
            ! ------------------------------------------------------
       FUNCTION readphi(pos)
       IMPLICIT NONE

       DOUBLE PRECISION readphi(3),p1(3)
       INTEGER i,j,k,t,pos(4)
       INTEGER i0,j0,k0,t0

       i0=mx(pos(1))
       j0=my(pos(2))
       k0=mz(pos(3))
       t0=mt(pos(4))
       p1=phi(:,i0,j0,k0,t0)

#ifdef Cper
       IF (pos(1).NE.i0) p1(2)=-p1(2)
       IF (pos(2).NE.j0) p1(2)=-p1(2)
       IF (pos(3).NE.k0) p1(2)=-p1(2)
#endif Cper

#ifdef twist
       IF (pos(1).NE.i0) p1(1)=-p1(1)
       IF (pos(2).NE.j0) p1(2)=-p1(2)
       IF (pos(3).NE.k0) p1(3)=-p1(3)
#endif twist
       readphi=p1
       END FUNCTION readphi

            ! ------------------------------------------------------
            ! herm(a) = a+, i.e. hermitian of matrix a
            ! a element of SU(2)
            ! ------------------------------------------------------
       FUNCTION herm(a)
       IMPLICIT NONE
       DOUBLE PRECISION, DIMENSION(0:3) :: herm,a
       herm(0)=a(0)
       herm(1)=-a(1)
       herm(2)=-a(2)
       herm(3)=-a(3)
       END FUNCTION herm

            ! ------------------------------------------------------
            ! matmul(a, b) = a*b
            ! a,b elements of SU(2)
            ! ------------------------------------------------------
       FUNCTION matmul(a,b)
       IMPLICIT NONE
       DOUBLE PRECISION, DIMENSION(0:3) :: matmul,a,b
       matmul(0)=a(0)*b(0) - a(1)*b(1) - a(2)*b(2) - a(3)*b(3)
       matmul(1)=a(1)*b(0) + a(0)*b(1) + a(3)*b(2) - a(2)*b(3)
       matmul(2)=a(2)*b(0) - a(3)*b(1) + a(0)*b(2) + a(1)*b(3)
       matmul(3)=a(3)*b(0) + a(2)*b(1) - a(1)*b(2) + a(0)*b(3)
       END FUNCTION matmul
```

```
      ! ----------------------------------------------------
      ! dotprod(phi1, phi2) = (1/2)*Trace(phi1*phi2)
      ! phi1, phi2 traceless hermitian 2x2 matrices
      ! ----------------------------------------------------
DOUBLE PRECISION FUNCTION dotprod(p1,p2)
IMPLICIT NONE
DOUBLE PRECISION p1(3),p2(3)
dotprod=p1(1)*p2(1)+p1(2)*p2(2)+p1(3)*p2(3)
END FUNCTION dotprod

      ! ----------------------------------------------------
      ! squared(phi) = (1/2)*Trace(phi^2)
      ! phi traceless hermitian 2x2 matrix
      ! ----------------------------------------------------
DOUBLE PRECISION FUNCTION squared(p)
IMPLICIT NONE
DOUBLE PRECISION p(3)
squared=dotprod(p,p)
END FUNCTION squared

      ! ----------------------------------------------------
      ! Generates a new link from an old one
      ! ensuring new link still is a member of SU(2)
      ! rangeA controls the size of the change
      ! r is a random number
      ! ----------------------------------------------------
FUNCTION gennewlink(n,oldlink,rangeA,r)
IMPLICIT NONE
DOUBLE PRECISION, DIMENSION(0:3)::gennewlink,oldlink
REAL r
TYPE (VSL_STREAM_STATE) :: stream
DOUBLE PRECISION rangeA,sr,cr,x2
INTEGER n,i,j,errcode
INTEGER pair1(6),pair2(6)
PARAMETER(pair1=(/0,0,0,1,1,2/))
PARAMETER(pair2=(/1,2,3,2,3,3/))

gennewlink=oldlink
i=pair1(n)
j=pair2(n)
sr=(2.0*r-1.0)*rangeA
cr=SQRT(1.0-sr**2)

      ! Matrix C = [cr sr; -sr cr]
      ! gennewlink = C * oldlink
gennewlink(i)=oldlink(i)*cr+oldlink(j)*sr
gennewlink(j)=-oldlink(i)*sr+oldlink(j)*cr

      ! x2 = 1 / |gennewlink|
x2=0.0
DO i=0,3
   x2=x2+gennewlink(i)**2
ENDDO

      ! normalise new link
      ! gennewlink = gennewlink / |gennewlink|
x2=1.0/SQRT(x2)
DO i=0,3
   gennewlink(i)=gennewlink(i)*x2
ENDDO
END FUNCTION
```

```
      ! ----------------------------------------------------------------
      ! posplus(mu, x) returns next node to x in the
      !                 mu direction
      ! ----------------------------------------------------------------
 FUNCTION posplus(dir,pos)
 IMPLICIT NONE
 INTEGER dir,pos(4),posplus(4)
 posplus=pos
 posplus(dir)=posplus(dir)+1
 END FUNCTION posplus

      ! ----------------------------------------------------------------
      ! posminus(mu, x) returns previous node to x in the
 !                                  mu direction
      ! ----------------------------------------------------------------
 FUNCTION posminus(dir,pos)
 IMPLICIT NONE
 INTEGER dir,pos(4),posminus(4)
 posminus=pos
 posminus(dir)=posminus(dir)-1
 END FUNCTION posminus

      ! ----------------------------------------------------------------
 ! mrot(a,phi) = a*phi*(a+)
      ! a element of SU(2)
      ! phi hermitian traceless 2x2 matrix
      ! ----------------------------------------------------------------
 FUNCTION mrot(a,p)
 IMPLICIT NONE
 DOUBLE PRECISION mrot(3)
 DOUBLE PRECISION a(0:3),p(3)
 mrot(1)= a(0)**2*p(1) + a(1)**2*p(1) - a(2)**2*p(1) -
-  a(3)**2*p(1) + 2*a(1)*a(2)*p(2) +
-  2*a(0)*a(3)*p(2) - 2*a(0)*a(2)*p(3) +
-  2*a(1)*a(3)*p(3)
 mrot(2)=    2*a(1)*a(2)*p(1) - 2*a(0)*a(3)*p(1) +
-  a(0)**2*p(2) - a(1)**2*p(2) + a(2)**2*p(2) -
-  a(3)**2*p(2) + 2*a(0)*a(1)*p(3) +
-  2*a(2)*a(3)*p(3)
 mrot(3)=    2*a(0)*a(2)*p(1) + 2*a(1)*a(3)*p(1) -
-  2*a(0)*a(1)*p(2) + 2*a(2)*a(3)*p(2) +
-  a(0)**2*p(3) - a(1)**2*p(3) - a(2)**2*p(3) +
-  a(3)**2*p(3)
 END FUNCTION mrot

      ! ----------------------------------------------------------------
 ! mroth(a,phi) = (a+)*phi*a
      ! a element of SU(2)
      ! phi hermitian traceless 2x2 matrix
      ! ----------------------------------------------------------------
 FUNCTION mroth(a,p)
 IMPLICIT NONE
 DOUBLE PRECISION mroth(3)
 DOUBLE PRECISION a(0:3),p(3)
 mroth(1)=  a(0)**2*p(1) + a(1)**2*p(1) - a(2)**2*p(1) -
-  a(3)**2*p(1) + 2*a(1)*a(2)*p(2) -
-  2*a(0)*a(3)*p(2) + 2*a(0)*a(2)*p(3) +
-  2*a(1)*a(3)*p(3)
 mroth(2)=  2*a(1)*a(2)*p(1) + 2*a(0)*a(3)*p(1) +
-  a(0)**2*p(2) - a(1)**2*p(2) + a(2)**2*p(2) -
```

```
-  a(3)**2*p(2) - 2*a(0)*a(1)*p(3) +
-  2*a(2)*a(3)*p(3)
 mroth(3)=    -2*a(0)*a(2)*p(1) + 2*a(1)*a(3)*p(1) +
-  2*a(0)*a(1)*p(2) + 2*a(2)*a(3)*p(2) +
-  a(0)**2*p(3) - a(1)**2*p(3) - a(2)**2*p(3) +
-  a(3)**2*p(3)
 END FUNCTION mroth

     ! -----------------------------------------------
 ! matmul3nhh(a,b,c) = a*(b+)*(c+)
     ! a,b,c elements of SU(2)
     ! a+ is the hermitian conjugate of a
     ! -----------------------------------------------
 FUNCTION matmul3nhh(a,b,c)
 DOUBLE PRECISION ,DIMENSION(0:3) :: matmul3nhh,out,a,b,c
 out(0)=
+ a(0)*b(0)*c(0) + a(1)*b(1)*c(0) +
-  a(2)*b(2)*c(0) + a(3)*b(3)*c(0) +
-  a(1)*b(0)*c(1) - a(0)*b(1)*c(1) -
-  a(3)*b(2)*c(1) + a(2)*b(3)*c(1) +
-  a(2)*b(0)*c(2) + a(3)*b(1)*c(2) -
-  a(0)*b(2)*c(2) - a(1)*b(3)*c(2) +
-  a(3)*b(0)*c(3) - a(2)*b(1)*c(3) +
-  a(1)*b(2)*c(3) - a(0)*b(3)*c(3)
 out(1)=
+      a(1)*b(0)*c(0) - a(0)*b(1)*c(0) -
-  a(3)*b(2)*c(0) + a(2)*b(3)*c(0) -
-  a(0)*b(0)*c(1) - a(1)*b(1)*c(1) -
-  a(2)*b(2)*c(1) - a(3)*b(3)*c(1) -
-  a(3)*b(0)*c(2) + a(2)*b(1)*c(2) -
-  a(1)*b(2)*c(2) + a(0)*b(3)*c(2) +
-  a(2)*b(0)*c(3) + a(3)*b(1)*c(3) -
-  a(0)*b(2)*c(3) - a(1)*b(3)*c(3)
 out(2)=
+    a(2)*b(0)*c(0) + a(3)*b(1)*c(0) -
-  a(0)*b(2)*c(0) - a(1)*b(3)*c(0) +
-  a(3)*b(0)*c(1) - a(2)*b(1)*c(1) +
-  a(1)*b(2)*c(1) - a(0)*b(3)*c(1) -
-  a(0)*b(0)*c(2) - a(1)*b(1)*c(2) -
-  a(2)*b(2)*c(2) - a(3)*b(3)*c(2) -
-  a(1)*b(0)*c(3) + a(0)*b(1)*c(3) +
-  a(3)*b(2)*c(3) - a(2)*b(3)*c(3)
 out(3)=
+   a(3)*b(0)*c(0) - a(2)*b(1)*c(0) +
-  a(1)*b(2)*c(0) - a(0)*b(3)*c(0) -
-  a(2)*b(0)*c(1) - a(3)*b(1)*c(1) +
-  a(0)*b(2)*c(1) + a(1)*b(3)*c(1) +
-  a(1)*b(0)*c(2) - a(0)*b(1)*c(2) -
-  a(3)*b(2)*c(2) + a(2)*b(3)*c(2) -
-  a(0)*b(0)*c(3) - a(1)*b(1)*c(3) -
-  a(2)*b(2)*c(3) - a(3)*b(3)*c(3)
 matmul3nhh=out
 END FUNCTION matmul3nhh

     ! -----------------------------------------------
 ! matmul3hhn(a,b,c) = (a+)*(b+)*c
     ! a,b,c elements of SU(2)
     ! a+ is the hermitian conjugate of a
     ! -----------------------------------------------
 FUNCTION matmul3hhn(a,b,c)
 DOUBLE PRECISION ,DIMENSION(0:3) :: matmul3hhn,out,a,b,c
 out(0)=
```

```
+  a(0)*b(0)*c(0) - a(1)*b(1)*c(0) -
-  a(2)*b(2)*c(0) - a(3)*b(3)*c(0) +
-  a(1)*b(0)*c(1) + a(0)*b(1)*c(1) -
-  a(3)*b(2)*c(1) + a(2)*b(3)*c(1) +
-  a(2)*b(0)*c(2) + a(3)*b(1)*c(2) +
-  a(0)*b(2)*c(2) - a(1)*b(3)*c(2) +
-  a(3)*b(0)*c(3) - a(2)*b(1)*c(3) +
-  a(1)*b(2)*c(3) + a(0)*b(3)*c(3)
 out(1)=
+ -(a(1)*b(0)*c(0)) - a(0)*b(1)*c(0) +
-  a(3)*b(2)*c(0) - a(2)*b(3)*c(0) +
-  a(0)*b(0)*c(1) - a(1)*b(1)*c(1) -
-  a(2)*b(2)*c(1) - a(3)*b(3)*c(1) -
-  a(3)*b(0)*c(2) + a(2)*b(1)*c(2) -
-  a(1)*b(2)*c(2) - a(0)*b(3)*c(2) +
-  a(2)*b(0)*c(3) + a(3)*b(1)*c(3) +
-  a(0)*b(2)*c(3) - a(1)*b(3)*c(3)
 out(2)=
+   -(a(2)*b(0)*c(0)) - a(3)*b(1)*c(0) -
-  a(0)*b(2)*c(0) + a(1)*b(3)*c(0) +
-  a(3)*b(0)*c(1) - a(2)*b(1)*c(1) +
-  a(1)*b(2)*c(1) + a(0)*b(3)*c(1) +
-  a(0)*b(0)*c(2) - a(1)*b(1)*c(2) -
-  a(2)*b(2)*c(2) - a(3)*b(3)*c(2) -
-  a(1)*b(0)*c(3) - a(0)*b(1)*c(3) +
-  a(3)*b(2)*c(3) - a(2)*b(3)*c(3)
 out(3)=
+ -(a(3)*b(0)*c(0)) + a(2)*b(1)*c(0) -
-  a(1)*b(2)*c(0) - a(0)*b(3)*c(0) -
-  a(2)*b(0)*c(1) - a(3)*b(1)*c(1) -
-  a(0)*b(2)*c(1) + a(1)*b(3)*c(1) +
-  a(1)*b(0)*c(2) + a(0)*b(1)*c(2) -
-  a(3)*b(2)*c(2) + a(2)*b(3)*c(2) +
-  a(0)*b(0)*c(3) - a(1)*b(1)*c(3) -
-  a(2)*b(2)*c(3) - a(3)*b(3)*c(3)
 matmul3hhn=out
 END FUNCTION matmul3hhn

     ! ------------------------------------------------
 ! trmul(a,b) = trace(a*b)
     ! a,b elements of SU(2)
     ! ------------------------------------------------
 DOUBLE PRECISION FUNCTION trmul(a,b)
 DOUBLE PRECISION a(0:3),b(0:3)
 trmul=2.0*(a(0)*b(0) - a(1)*b(1) - a(2)*b(2) - a(3)*b(3))
 END FUNCTION trmul

     ! ------------------------------------------------
 ! trmulnh(a,b) = trace(a*b+)
     ! a,b elements of SU(2)
     ! b+ is the hermitian conjugate of b
     ! ------------------------------------------------
 DOUBLE PRECISION FUNCTION trmulnh(a,b)
 IMPLICIT NONE
 DOUBLE PRECISION a(0:3),b(0:3)
 trmulnh=2.0*(a(0)*b(0) + a(1)*b(1) + a(2)*b(2) + a(3)*b(3))
 END FUNCTION trmulnh

 END
```

Appendix B

Bootstrap code

```
      PROGRAM bootstrap
      IMPLICIT NONE

      INTEGER,PARAMETER::MAXSAMPLE=500
      CHARACTER(200) inputfile
      REAL, DIMENSION(200):: avgdata
      INTEGER N
      REAL avgphiN,m2,stderr

! ----------------------------------------------------------------
! ------------------- Main Code ----------------------------------
! ----------------------------------------------------------------
          ! Read input file name
          CALL getarg(1,inputfile)

          ! Extract avg data from input file
          N = ReadAvgPhi(inputfile,avgdata,m2)

          ! Calculate average
          avgphiN = sum(avgdata(1:N))/N

          ! Calculate standard deviation using bootstrap
      stderr = CalcStdError(avgdata(1:N),MAXSAMPLE)

          ! Output to file
          OPEN(unit=11,file=TRIM(inputfile)//'_boot',action='WRITE', status='REPLACE')
      WRITE(11,"(I4,A1,F10.4,A1,F12.6,A1,F12.6,A1,F12.6)") &
                N,' ',m2,' ',avgphiN,' ',stderr,' ', stderr/avgphiN*100
      CLOSE(11)

      CONTAINS

! ----------------------------------------------------------------
! ------------------- Read input file ----------------------------
! ----------------------------------------------------------------
      INTEGER FUNCTION ReadAvgPhi(filename,avgdata,m2)
      IMPLICIT NONE
      CHARACTER(*),INTENT(in):: filename
      REAL,DIMENSION(:),INTENT(out):: avgdata
      REAL,INTENT(OUT)::m2
```

```
      INTEGER linecount
      REAL tmp, avgphi

      avgdata=0
      OPEN(unit=10,file=TRIM(filename),action='READ',status='OLD')

          linecount=0
      DO
         READ(10,*,end=100,err=90)   tmp,avgphi,m2
         linecount=linecount+1
         avgdata(linecount)=avgphi
      END DO

90    WRITE(*,*) "ERROR"
100   CLOSE(unit=10)

          ReadAvgPhi=linecount
      END FUNCTION ReadAvgPhi

! -----------------------------------------------------------------
! --------------- Bootstrap method --------------------------------
! -----------------------------------------------------------------
      REAL FUNCTION CalcStdError(indata,maxsample)
      IMPLICIT NONE
      REAL,DIMENSION(:),INTENT(in)::indata
      INTEGER,INTENT(in)::maxsample

      INTEGER::N,i,j
      REAL r,bootmean
      REAL,DIMENSION(size(indata))::newdata
      REAL,DIMENSION(maxsample)::newavgs

          N=size(indata)

          ! Initialise random numbers
      CALL RANDOM_SEED

      ! Generate maxsample sets of data
      DO i=1,maxsample
         ! Create new set of data by sampling old one with repetition
         DO j=1,N
            CALL RANDOM_NUMBER(r)
            newdata(j)=indata(INT(r*N)+1)
         ENDDO

         ! Calculate average of new data and store it
         newavgs(i) = sum(newdata)/N
      ENDDO

      ! Calculate mean of averages
      bootmean = sum(newavgs)/maxsample

      ! Calculate standard deviation of boot samples
      CalcStdError = sqrt(sum((newavgs-bootmean)**2)/(maxsample-1))

      END FUNCTION CalcStdError

      END PROGRAM bootstrap
```

Bibliography

[1] Paul AM Dirac. Quantised singularities in the electromagnetic field. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, pages 60–72, 1931.

[2] Gerard Hooft. Magnetic monopoles in unified gauge theories. *Nuclear Physics B*, 79(2):276–284, 1974.

[3] Alexander M Polyakov. Particle spectrum in the quantum field theory. Technical report, SIS-74-4677, 1974.

[4] MN Chernodub, FV Gubarev, MI Polikarpov, and Valentin I Zakharov. Monopoles and confining strings in qcd. *arXiv preprint hep-lat/0103033*, 2001.

[5] Alan H Guth. Inflationary universe: A possible solution to the horizon and flatness problems. *Physical Review D*, 23(2):347, 1981.

[6] Howard Georgi and SL Glashow. Unity of all elementary-particle forces. *Physical Review Letters*, 32(8):438, 1974.

[7] Arttu Rajantie. Introduction to magnetic monopoles. *Contemporary Physics*, 53(3):195–211, 2012.

[8] John Preskill. Magnetic monopoles. *Annual Review of Nuclear and Particle Science*, 34(1):461–530, 1984.

[9] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[10] Michael Creutz. Quarks, gluons and lattices. 1985.

[11] Isabel Beichl and Francis Sullivan. The metropolis algorithm. *Computing in Science & Engineering*, 2(1):65–69, 2000.

[12] G Peter Lepage. Lattice qcd for novices. *arXiv preprint hep-lat/0506036*, 2005.

[13] B Efron and RJ Tibshirani. An introduction to the bootstrap. *Monographs on statistics and applied probability (57)*, 1993.

[14] J Froehlich and PA Marchetti. Gauge-invariant charged, monopole and dyon fields in gauge theories. *Nuclear Physics B*, 551(3):770–812, 1999.

[15] VA Belavin, MN Chernodub, and MI Polikarpov. Monopole creation operators as confinement–deconfinement order parameters. *Physics Letters B*, 554(3):146–154, 2003.

[16] Arsen Khvedelidze, David McMullan, and Alex Kovner. Magnetic monopoles in 4d: a perturbative calculation. *Journal of High Energy Physics*, 2006(01):145, 2006.

[17] Arttu Rajantie and David J Weir. Nonperturbative study of theï£ · t hooft-polyakov monopole form factors. *Physical Review D*, 85(2):025003, 2012.

[18] Kenneth G Wilson. Confinement of quarks. *Physical Review D*, 10(8):2445, 1974.

[19] Anne C Davis, Tom WB Kibble, Arttu Rajantie, and Hugh P Shanahan. Topological defects in lattice gauge theories. *Journal of High Energy Physics*, 2000(11):010, 2000.

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

[21] Imperial college high performance computing service.