Topology of the $O(3)$ non-linear sigma model under the gradient flow

A thesis submitted in partial fulfillment of the requirement
for the degree of Bachelor of Science with Honors in
Physics from the College of William and Mary in Virginia,

by

Stuart Thomas

Advisor: Prof. Christopher J. Monahan

Prof. Todd Averett

Prof. Andreas Stathopoulos

Williamsburg, Virginia
May 2021

Contents

Acknowledgments

I would like to thank my advisor, Professor Christopher Monahan. His constant helpfulness has made this research possible and enjoyable. I would also like to thank the entire faculty of the William & Mary Physics Department. The amount of knowledge and wisdom they have given me has been beyond my imagination.

I would like to acknowledge the help of the High Performance Computing group at William & Mary, whose cluster performed many of these simulations. I also want to acknowledge the financial help of the 1693 Scholars Program which funded preliminary research over the summer.

Finally, I would like to thank my friends and family who helped me reach this point in my academic and personal journey.

List of Figures

List of Tables

Abstract

Quantum field theory is an extraordinarily successful framework that describes phenomena in particle physics and condensed matter. The $O(3)$ non-linear sigma model (NLSM) is a specific theory used in both of these fields, describing ferromagnets and acting as a prototype for the strong nuclear force. It features topologically stable configurations known as instantons which cannot continuously evolve to the ground state. The topological susceptibility is a parameter that describes this stability and is predicted to vanish in physical theories, however numerical simulations find that the topological susceptibility diverges in the continuum limit [1]. This issue has motivated the application of the "gradient flow", a smoothing of high-frequency modes. We study the effect of the gradient flow on this divergence, finding that it persists as a logarithmic divergence. This result supports a previous study [2] and indicates that either the definition of topological charge is problematic or the NLSM has no well-defined continuum limit. We also study the nontrivial field theory by introducing a θ-term into the action and analyze the topological charge as a function of θ under the gradient flow.

Chapter 1

Introduction

Quantum field theory (QFT) is a framework used to describe a range of physical phenomena to a remarkable degree of accuracy. Paired with the Standard Model, QFT provides the prevailing basis for all small-scale physics (that is, where general relativity does not apply) and is the fundamental tool for studying particle physics. QFT also plays a critical role in condensed matter physics through effective field theories which model emergent phenomena such as phonons and quasiparticles. Compared to experiment, QFT is remarkably accurate, famously predicting the electron g-factor to eleven significant figures [3], arguably the most accurate prediction in all of science.

However this power comes at a cost: the study of quantum fields is rife with infinities. A naïve treatment of quantum field theory produces divergent values for physical quantities, a clearly impossible result. Since the 1950s, this issue has been resolved for a large number of models— most notably quantum electrodynamics— through perturbation theory and the renormalization group. This counter-intuitive technique exploits the freedom of parameters such as mass and electric charge. Since these constants cannot be directly measured, renormalization allows them to become infinite, thereby cancelling the infinities in physical results. Unfortunately, not all theories are perturbatively renormalizable.

One such example is the *non-linear sigma model* (NLSM), a prototypical theory

in both condensed matter and particle physics. In solid-state systems, this model describes Heisenberg ferromagnets [4] and in nuclear physics, it acts as a prototype for quantum chromodynamics (QCD), the gauge theory that describes the strong nuclear force. In general, the NLSM shares key features with non-Abelian gauge theories such as QCD, including a mass gap and asymptotic freedom [5]. Therefore, the NLSM is a useful model for exploring the effect of these properties in a simpler system.

In this study, we specifically consider the $O(3)$ NLSM in 1+1 dimensions (one dimension of space, one dimension of time). This theory exhibits topological properties such as *instantons*, or classical field solutions at local minima of the action in Euclidean space. Solutions such as these are "topologically protected", meaning they cannot evolve into the vacuum state via small fluctuations. Due to this property, topology is critically important to quantum field theories in cosmology and high energy physics [6]. Additionally, topological stability may become a key tool for fault-tolerant quantum computers [7]. In these devices, topology protects the delicate quantum states necessary for information processing.

Since the NLSM in not perturbatively renormalizable, we require nonperturbative techniques to study topological effects. A solution is to place the field on a discretized Euclidean lattice, a technique originally used for quantum chromodynamics [8]. After this transformation, fields become numerically calculable using modern computers. This process introduces the lattice spacing as a length scale a where the *continuum limit* is defined as taking a to zero. We expect physical observables to converge in the continuum limit, however this is not always the case. As an example, states of definite angular momentum mix when discretized on a rectangular lattice due to a breaking of continuous rotational symmetry. This causes some operators that depend on angular momentum to suffer divergences.

In this study, we focus on one observable that diverges in the continuum: the topological susceptibility. The topological susceptibility in the 1+1 $O(3)$ NLSM has been the subject of debate for the past four decades [2] since it is unclear if a convergent solution exists. While the some analytical arguments argue the topological susceptibility should approach zero in the continuum limit, numerical results predict infinities [1]. In QCD, mathematical techniques proved that the susceptibility vanishes in the continuum limit [9], a fact supported by numerical calculation [10].

To remedy this issue, we consider the gradient flow, a technique designed to remove divergences in operators. By dampening high-frequency fluctuations, the gradient flow reduces terms that scale with the inverse lattice spacing, making some observables finite on the lattice [11]. In QCD, the gradient flow successfully makes the topological susceptibility finite in numerical calculations [10], corroborating the analytical result in [9]. This success has motivated the usage of the gradient flow to calculate the topological susceptibility in the 1+1 $O(3)$ NLSM, though recent studies demonstrate that the observable still diverges in the continuum limit [2].

A second perspective on the topological susceptibility arises from the introduction of a θ-term into the field Lagrangian. This term drives the vacuum state into a topological phase [12]. Differentiating the field's partition function with respect to θ yields a value proportional to the topological susceptibility. The effect of nonzero θ on the theory therefore should reflect the divergence in the continuum limit.

In this work we verify the divergence of the topological susceptibility and develop a clearer picture of how the θ-term affects the topology of the 1+1 $O(3)$ NLSM.

1.1 Method Overview

To numerically study the topological qualities of the NLSM, we first implement a Markov chain Monte Carlo simulation. We initially construct a proof-of-concept

Python program that models the simpler ϕ^4 model (see Sec. 2.1.2). After comparing with existing literature, we transition to a C++ simulation for efficiency, implementing the NLSM on larger lattices. Since the gradient flow has no exact solution in the NLSM we implement a numerical solution using a fourth-order Runge-Kutta approximation with automatic step sizing. By applying the gradient flow to every configuration in the sample, we can measure its effect on the topological charge and susceptibility.

1.2 Summer Research

A portion of this work began during the summer of 2020 using funding from the 1693 Scholars Program. This preliminary research consisted of literature review and numerical tests with an Ising model simulation as well as an initial implementation of the ϕ^4 model. The ϕ^4 calculation performed in this study and the entirety of the NLSM portion occurred during the academic year as part of the PHYS 495/496 Honors course.

1.3 Conventions

- Throughout this paper, we use natural units, i.e. $\hbar = 1$ and $c = 1$.
- We use Einstein summation notation, an implicit sum over repeated spacetime indices. For example, if x^μ is a four-vector in Minkowski spacetime and x_μ is its covariant form, the term

$$\begin{aligned} x^\mu x_\mu &= \sum_{\mu=0}^{4} x^\mu x_\mu \\ &= x_0^2 - x_1^2 - x_2^2 - x_3^2. \end{aligned}$$

Chapter 2

Theory

This thesis incorporates two main bodies of knowledge: quantum field theory and statistical simulation. Through the path integral formulation of quantum field theory, we are able to describe the physics of the former with the established mathematics of the latter.

2.1 Quantum Field Theory

In this section we outline a rough description of quantum field theory. A full introduction is beyond the scope of this paper, however we do assume knowledge of nonrelativistic quantum mechanics and classical field theory.

The fundamental hypothesis of quantum field theory (QFT) describes particles as discrete packets of energy on a quantum field. But what is a quantum field? Like in classical mechanics, a field is a function of spacetime with some mathematical object assigned to each point in space and time. In the case of the electric field, this object is a three-dimensional vector, while the electric potential is a scalar field. Classical and quantum fields have Lagrangians which define how they evolve in space and time. What differentiates a quantum field from a classical field is superposition: where classical fields have a definite configuration, quantum fields exist in a superposition of all possible configurations. It is possible—though nontrivial and outside the scope of this

description— to motivate the appearance of discrete particles from this superposition (see [13]). This formulation of QFT is known as the "path integral formulation", which differs from the "second quantization" used by many textbooks.

This general description allows QFT to easily incorporate special relativity. By ensuring that the Lagrangian of a theory is invariant under Lorentz transformations, we can ensure that the theory itself is Lorentz invariant. This is a necessary condition of physical theories.

2.1.1 Path Integral Formulation

We can model a quantum field as a superposition of all possible classical fields. Like single-particle quantum mechanics, each configuration has a complex probability amplitude. To measure expectation values of observables, we simply take an average over all configurations weighted by this complex amplitude. We can formalize this notion using the fundamental formula[1]

$$\langle \hat{O} \rangle = \frac{1}{Z} \int \mathcal{D}\phi \, \hat{O}[\phi] \, e^{iS[\phi]} \tag{2.1}$$

where $\langle \hat{O} \rangle$ is the expectation value of an arbitrary operator $\hat{O}$; Z is a normalization constant; S is the action functional, defined from the theory's Lagrangian; and $\int \mathcal{D}\phi$ represents the eponymous path integral. Though it is possible to define this integral rigorously, for our purposes we can equate it to a sum over all possible configurations. This form is the quantum analog of the classical principle of least action and reduces to such for large values of the action. For a more pedagogical explanation, see Richard Feynman's lectures on physics [14].

At first glance, Eq. 2.1 is remarkably similar to the statistics of the canonical ensemble. Through this similarity, we will be able to use mathematical tools from statistical mechanics to study quantum field theories. However, the factor of i in

[1] Since this study concerns vacua, we do not include a source term.

the exponent currently prohibits us from making this jump. To remedy this issue, we perform a "Wick rotation" which shifts spacetime into Euclidean coordinates. In physical spacetime, defined by the Minkowski metric, the Lorentz-invariant distance is given as

$$s^2 = x_0^2 - x_1^2 - x_3^2 - x_3^2 \tag{2.2}$$

where $x_0 = ct$ and $\vec{x} = (x_1, x_2, x_3)^T$. By redefining the time coordinate of a spacetime point x to be $x_4 = ix_0$, we find that the quantity

$$s_E^2 = x_1^2 + x_3^2 + x_3^2 + x_4^2, \tag{2.3}$$

is invariant under $SO(4)$ transformations and is therefore representative of a four-dimensional Euclidean space. Furthermore, we find that

$$\begin{aligned} d^4x_E &= d^3\vec{x}dx_4 \\ &= id^3\vec{x}dx_0 \\ &= id^4x. \end{aligned} \tag{2.4}$$

We can use this transformation to redefine the Lagrangian $\mathcal{L}$ in Euclidean space as $\mathcal{L}_E$, replacing all x_0 with $-ix_4$ and flipping the overall sign. Since a Lorentz-invariant Lagrangian must only include even powers and derivatives of x, the Euclidean Lagrangian remains real. Subsequently, we can define a Euclidean action based on the differential in Eq. 2.4:

$$\begin{aligned} S_E &= \int d^4x_E \mathcal{L}_E \\ &= i \int d^4x \, (-\mathcal{L}) \\ &= -iS \end{aligned} \tag{2.5}$$

allowing us to redefine the path integral as

$$\langle \hat{O} \rangle = \frac{1}{Z} \int \mathcal{D}\phi \, \hat{O}[\phi] \, e^{-S_E[\phi]}. \tag{2.6}$$

By replacing the Minkowski action S with a Euclidean action S_E, we have transformed the amplitude e^{iS} to a statistical Boltzmann factor e^{-S_E}. This new form will allow us to use statistical techniques to simulate quantum fields.

2.1.2 ϕ^4 Model

One of the simplest interacting field theories is known as the ϕ^4 model. This theory describes a spin-0 boson and consists of a real scalar field given by the four-dimensional Minkowski action

$$S[\phi] = \int d^4x \left[\frac{1}{2}\partial^\mu\phi\partial_\mu\phi - \frac{1}{2}m_0^2\phi^2 - \frac{\lambda}{4}\phi^4\right]. \tag{2.7}$$

The first two terms describe a free relativistic particle of mass m_0 while the last term describes an interaction with strength λ. Per Einstein summation notation, there is an implicit sum over spacetime dimensions $\mu \in \{0, 1, 2, 3\}$ indexing the derivative vectors[2]

$$\partial^\mu = \left(\frac{\partial}{\partial t}, \frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right) \tag{2.8}$$

$$\partial_\mu = \left(\frac{\partial}{\partial t}, -\frac{\partial}{\partial x}, -\frac{\partial}{\partial y}, -\frac{\partial}{\partial z}\right). \tag{2.9}$$

In this study, we specifically consider fields in $1+1$ spacetime dimensions. Following Sec. 2.1.1, we convert Eq. 2.7 from 3+1 Minkowski spacetime to $1+1$ Euclidean spacetime, yielding

$$S_E[\phi] = \int d^2x_E \left[\frac{1}{2}\left(\partial_t\phi\right)^2 + \frac{1}{2}\left(\partial_x\phi\right)^2 + \frac{1}{2}m_0^2\phi^2 + \frac{\lambda}{4}\phi^4\right]. \tag{2.10}$$

where ∂_t and ∂_x are the two partial derivatives in $1+1$ Euclidean spacetime.

This field features spontaneous symmetry breaking at a critical value of m_0^2 [15]. This property causes the field to spontaneously align, similarly to spins aligning in a

[2]This canonical representation of the kinetic term $\frac{1}{2}\partial^\mu\phi\partial_\mu\phi$ is equivalent to $\frac{1}{2}\dot{\phi}^2 - \frac{1}{2}\left(\nabla\phi\right)^2$.

ferromagnet. The name "symmetry breaking" refers to the transformation $\phi \to -\phi$, which changes the values of observables in the aligned regime but not the disordered regime. These two phases are known as the "broken" and "symmetric" phases and their transition is well understood.

2.1.3 Non-linear Sigma Model

The non-linear sigma model (NLSM) is a prototypical theory for a variety of physical phenomena including applications in string theory [4] and ferromagnetism [5]. As a simple nonperturbative model, it also provides an ideal starting point for lattice QCD studies. Specifically, the NLSM exhibits many properties shared by Yang-Mills gauge theories, such as a mass gap, asymptotic freedom and $O(2)$ renormalizability [5].

Unlike the ϕ^4 model, which consists of a real value at each point in spacetime, the $O(3)$ NLSM consists of a 3D unit vector at each point. For this reason, every transformation of the field must be norm preserving. Per its name, the $O(3)$ NLSM features a global symmetry under the 3D orthogonal group $O(3)$; in other words, the theory remains the same if all vectors are rotated equivalently. To differentiate it from the ϕ^4 model, we denote the NLSM field as $\vec{e}(x)$.

The theory is defined by the $1+1$ dimensional Euclidean action

$$S_E = \frac{\beta}{2} \int d^2x \ \left[(\partial_t \vec{e})^2 + (\partial_x \vec{e})^2 \right] \tag{2.11}$$

subject to the constraint that $\vec{e} \cdot \vec{e} = 1$. Here, β is the inverse coupling.

2.2 Markov Chain Monte Carlo

To accomplish a statistical analysis of quantum fields, we use a Monte Carlo simulation which produces a large number of configurations and calculates statistics on the sample. A brute-force calculation over all possible configurations, as Eq. 2.6 suggests,

is clearly infinite and computationally infeasible. However, the exponential nature of the Boltzmann factor dictates that only configurations near the action minima contribute to observable statistics. Therefore, by selecting a sample of configurations near this minimum, we are able to extract meaningful results with a finite computation.

2.2.1 The Markov Chain

In order to determine this subset of configurations, we use a Markov chain. This method identities field configurations that minimize the action using a random walk through phase space. Essentially, we begin with a predetermined configuration and then make small adjustments, gradually lowering the action. By measuring states after a certain amount of time has passed, called the "thermalization", we can form a set of configurations near the action minima and approximate the observables of the system.

Each step consists of two parts: proposing a change and accepting the new configuration. The proposal creates a new configuration ϕ_b based on the current configuration ϕ_a, a change that we accept with probability

$$P(\phi_a \to \phi_b). \tag{2.12}$$

This probability determines if ϕ_b should be added to the Markov chain and depends on the change in action, stochastically ensuring that the chain will seek the action minima.

There are four requirements that this function must obey to produce a Boltzmann distribution of samples:

1. $P(\phi_a \to \phi_b)$ must depend only on the configurations ϕ_a and ϕ_b.

2. The probability must be properly normalized, i.e. $\sum_\phi P(\phi_a \to \phi) = 1$.

3. Every configuration must be reachable in a finite number of steps. In other words, the chain must be ergodic.

4. In order to reach equilibrium, the chain must be reversible. In other words, the probability of a $\phi_a \to \phi_b$ transition must be equal to the probability of a $\phi_b \to \phi_a$ transition. Mathematically, this condition takes the form of a "detailed balance equation":

$$P(\phi_a)\, P(\phi_a \to \phi_b) = P(\phi_b)\, P(\phi_b \to \phi_a), \tag{2.13}$$

where $P(\phi)$ is the probability of a system existing in state ϕ.

This final condition will allow us to explicitly define the transition probability using the action. From the Boltzmann distribution, we know

$$P(\phi) = \frac{1}{Z} e^{-S_E[\phi]}. \tag{2.14}$$

Therefore, by rearranging Eq. 2.13, we find

$$\frac{P(\phi_a \to \phi_b)}{P(\phi_b \to \phi_a)} = e^{S_E[\phi_a] - S_E[\phi_b]}. \tag{2.15}$$

This formula will provide the explicit transition probabilities for the Metropolis and Wolff algorithms.

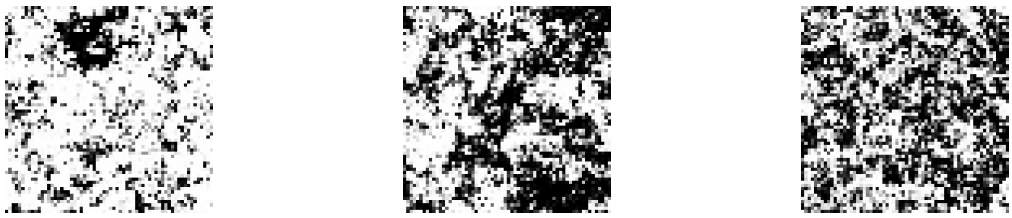
2.3 Observables

To extract physics from Monte Carlo simulations, we define a set of "observables". These quantities manifest as expectation values of operators, calculated using the Euclidean path integral formula (Eq. 2.6). We can classify these observables into two categories: primary and secondary observables. Primary observables are calculated as expectation values of global operators while secondary observables are derived from these quantities.

2.3.1 Primary Observables

Each primary observable is defined on each configuration independently, meaning they do not encode ensemble statistics of the Markov chain. In the ϕ^4 model, we can develop an intuition around these quantities by visualizing the symmetric and broken phases. Fig 2.1 and Tab. 2.1 show examples of these quantities in three different configurations: one in the broken phase, one in the symmetric phase, and one at the transition.

There are two potential points of confusion here. The first lies in the definition of "broken" phase. Though the symmetric phase more closely resembles a pane of broken glass, it leaves $\phi \to -\phi$ symmetry *un*-broken, thereby giving the title "broken" to the more visually uniform configuration. An additional potential pitfall is the distinction between the *lattice average* and the *ensemble average*. The first is a mean over all lattice sites while the second is a mean over all configurations in the Markov Chain.



(a) broken ($m_0^2 = -1.0$)

(b) transition ($m_0^2 = -0.7$)

(c) symmetric ($m_0^2 = -0.4$)

Figure 2.1: Visualization of broken phase, symmetric phase and transition. Simulation run on 64×64 lattice, plotted after 1000 sweep thermalization (see Sec 3.3.4), $\lambda = 0.5$.

Average Magnetization

The average magnetization quantifies the total alignment of the field. In both the ϕ^4 model and the NLSM, a value of zero indicates the symmetric phase while a nonzero value indicates broken symmetry. In the NLSM, a magnitude of one represents total

	broken	transition	symmetric
$\lvert\bar{\phi}\rvert$	0.56	0.07	0.02
S_E/L^2	0.29	0.40	0.44

Table 2.1: Average magnetization with average action per site corresponding to the particular configurations in Fig. 2.1.

alignment.

Due to the $\phi \to -\phi$ symmetry of the ϕ^4 model, the ensemble mean of the average magnetization $\langle\bar{\phi}\rangle$ is 0. Likewise, the $O(3)$ symmetry in the NLSM enforces $\langle\bar{\vec{e}}\rangle = 0$. To measure the alignment, we therefore use the magnitude of this quantity, defined in the 1+1 ϕ^4 model as

$$|\bar{\phi}| \equiv \frac{1}{V}\left|\int d^2x\ \phi(x)\right|. \tag{2.16}$$

and in the NLSM as

$$|\bar{\vec{e}}| \equiv \frac{1}{V}\left|\int d^2x\ \vec{e}(x)\right|. \tag{2.17}$$

In the broken phase, both $\langle|\bar{\vec{e}}|\rangle$ and $\langle|\bar{\phi}|\rangle$ are nonzero but vanish in the symmetric phase.

Internal Energy

The internal energy is defined as [1]

$$E = \frac{2}{\beta V}\langle S\rangle \tag{2.18}$$

in the NLSM. We use this metric to compare our simulation with existing literature.[3]

2.3.2 Secondary Observables

Unlike primary observables, secondary observables are defined for each *ensemble*, not each configuration. We define three secondary observables: the magnetic susceptibility, the Binder cumulant and the bimodality. Since the Binder cumulant and

[3]The internal energy is not part of the ϕ^4 portion of this work.

the bimodality primarily provide different perspectives on phase transitions, we will restrict our usage of these observables to the ϕ^4 model.

Magnetic Susceptibility

Though the magnitude of the average magnetization is the main phase transition indicator, the values become smooth around the critical point as the lattice spacing increases. This behaviour on the lattice makes the critical point difficult to identify. An alternative metric is the magnetic susceptibility. The magnetic susceptibility is proportional to the variance of the magnetization and features a peak at the critical point. This peak is more identifiable than the smooth transition of the magnetization.

Mathematically, this value is defined as

$$\chi_m \equiv V\big(\langle \bar{\vec{e}}^2 \rangle - \langle \bar{\vec{e}} \rangle^2\big) \tag{2.19}$$

in the NLSM and

$$\chi_m \equiv V\big(\langle \bar{\phi}^2 \rangle - \langle \bar{\phi} \rangle^2\big) \tag{2.20}$$

in the ϕ^4 model. Following the symmetry argument in Sec. 2.3.1, the second terms vanish and the susceptibility becomes

$$\chi_m = V\langle \bar{\vec{e}}^2 \rangle \tag{2.21}$$

in the NLSM and

$$\chi_m = V\langle \bar{\phi}^2 \rangle \tag{2.22}$$

in the ϕ^4 model.

Binder Cumulant

We define the Binder cumulant U as [16]

$$U \equiv 1 - \frac{\langle \bar{\phi}^4 \rangle}{3\langle \bar{\phi}^2 \rangle^2}. \tag{2.23}$$

Similar to the magnitude of the average magnetization, this formula yields 0 in the symmetric phase and a nonzero value in the broken phase. The Binder cumulant of the broken phase exhibits a universal value of $U = 2/3$. The advantage of this metric is a fixed point of the scaling transformation which corresponds with the critical point of the phase transition [17]. In other words, the Binder cumulants for different length scales should all intersect at the critical point. In our study, we use the Binder cumulant as further evidence of a phase transition in the ϕ^4 model.

2.3.3 Bimodality

The final phase transition indicator that we use is the bimodality. In the symmetric phase, the average magnetization $\bar{\phi}$ centers around 0 while in the broken phase, these values cluster around two peaks. This metric quantitatively measures the separation of these peaks.

To calculate the bimodality, we begin by measuring $\bar{\phi}$ for each configuration. We separate each value into an odd number number of bins, ensuring that there is a bin centered at $\bar{\phi} = 0$. We then calculate the number of configurations n_0 in the center bin and the number of configurations n_{max} in the fullest bin. The bimodality is then calculated as

$$B = 1 - \frac{n_0}{n_{max}}. \tag{2.24}$$

When the configurations are centered around $\bar{\phi} = 0$, i.e. in the symmetric phase, this value is $B = 0$. When the configurations are aligned such that $\bar{\phi} \neq 0$, i.e. in the broken phase, this value becomes $B = 1$. In this study, we separate ϕ into bins of width $\Delta\phi = 0.1$.

2.3.4 Jackknife Method

Though the uncertainties of primary observables are simple to calculate, this process is more complicated for secondary observables. While we could propagate the uncertainty of the Binder cumulant, such a process is not clear for the bimodality. Therefore, we utilize a method known as Jackknife resampling to calculate statistical errors of secondary observables.

We begin by calculating the expectation value O of some observable $\hat{O}$ on an ensemble of N configurations. Then, for each configuration i, we calculate the expectation value of $\hat{O}$ while excluding said configuration, calling this quantity O_i. This leaves us with a set of N expectation values O_i. Assuming independent measurements, we can calculate the variance of $\hat{O}$ as

$$\mathrm{Var}(\hat{O}) = \sum_i \left(O_i - O\right)^2 . \tag{2.25}$$

We use this formula to calculate all statistical errors in this study.

2.4 Topological Observables

The 1+1 $O(3)$ NLSM features topological features originating from two properties:

1. At $x \to \infty$, the field must become uniform since the Lagrangian must vanish. This allows us to model $x \to \infty$ as a single point on the field, forming a Riemann sphere with a two-dimensional surface.

2. The elements of the $O(3)$ NLSM are three-dimensional unit vectors, and thus also exist on a sphere.

With these two properties, we can view the field as a continuous mapping between two three-dimensional spheres, each denoted as S^2, and associate an integer number of wrappings to each mapping from S^2 to S^2. We can envision a tangible metaphor

for this wrapping with a balloon and a baseball: by simply inserting the baseball into the balloon, we have established a mapping from every point on the balloon to every point on the baseball. We can create an equally valid map by twisting the balloon's mouth and wrapping the baseball again. In a purely mathematical world, we perform this process an infinite number of times, thereby associating every possible mapping with an integer. The group of integers is known as the *homotopy group* of the $1+1$ $O(3)$ NLSM. We associate every field configuration with an element of this group, known as the *topological charge*, which we denote as Q. Configurations with $|Q| = 1$ are known as instantons.

Following this quantity, we can define a topological susceptibility χ_t

$$\chi_t \equiv \frac{1}{L^2}\Big(\langle Q^2\rangle - \langle Q\rangle^2\Big). \tag{2.26}$$

This quantity relates to the stability of topological phases and we expect it to approach zero in the continuum limit [1].

2.4.1 NLSM θ term

The NLSM is invariant under the transformation $\vec{e} \to -\vec{e}$, a change that flips the sign of the topological charge. This implies that $\langle Q\rangle$ disappears and therefore, that the NLSM vacuum is topologically trivial. We can construct a nontrivial vacuum by introducing a θ term into the action:

$$S[\vec{e}] \to S[\vec{e}] - i\theta Q[\vec{e}]. \tag{2.27}$$

With this topological action, $\langle Q\rangle = 0$ if and only if $\theta = 0$.

We can also redefine the topological susceptibility for the $\theta = 0$ case as

$$\chi_t = \frac{\langle Q^2\rangle}{L^2}. \tag{2.28}$$

2.5 Ultraviolet Divergences

In Section 2.1.1, we defined a fundamental equation of quantum fields using a path integral which encompasses an uncountably infinite configuration space. However, we said nothing of the integral's convergence. In fact, many fundamental processes in QFT have divergent amplitudes, yielding nonsensical results. The most common type of divergence stems from high-momentum states, giving them the name "ultraviolet divergences". The remedy to this catastrophe is unintuitive. Essentially, we adopt infinite values for the parameters of the Lagrangian (m_0^2 and λ in ϕ^4 theory and β in the NLSM). Since neither of these two quantities is ever measured directly, we do not have to assume that their values are finite. In practice, this technique consists of two steps: regularization and renormalization.

2.5.1 Regularization

Regularization is a process which introduces a new parameter into calculations. One example is a momentum cutoff. This technique transforms infinite momentum integrals as follows:

$$\int_0^\infty dk \to \int_0^\Lambda dk,$$

introducing Λ as a regularization parameter. This process makes results Λ-dependent, but finite. Another example is dimensional regularization, which calculates results in terms of the spacetime dimension D and analytically continues this parameter from the integers into the real numbers.

In this study, we employ a third technique: lattice regularization. This process discretizes the field, modeling $\phi(x)$ as a lattice $\phi(x_i)$ where i indexes lattice sites. The inherent parameter in this case is the lattice spacing a which measures the width of each lattice chunk. This discretization effectively imposes a hard momentum cutoff of $k = \pi/a$. According to Bloch's theorem, any mode above this cutoff is equivalent

to a lower-momentum mode since the high frequency information disappears on a discrete lattice. We can view this cutoff in momentum space by considering a square with side length $2\pi/a$ centered at the origin. On the lattice, any mode outside this zone contains no more information than a corresponding mode inside. This area is known as a "Brillouin zone" and contains all possible momentum modes on the lattice, effectively imposing a hard cutoff.

One of the main strengths of lattice regularization is preservation of gauge invariance, a property that makes lattice methods useful for QCD.

2.5.2 Renormalization

After regularization, we redefine the Lagrangian parameters in terms of the new parameter using a handful of boundary conditions. Following [2], we require that L/ξ remains constant, where L is the side length of the system and ξ is the coherence length. In perturbation theory, the renormalization process is arduous and includes the introduction of counter-terms into the Lagrangian. In the case of the NLSM it is impossible using counter-terms but can be performed numerically. In this study, we use the second moment of the correlation length, notated ξ_2, which more precisely estimates the ξ on the lattice [2]. The specific values of β and ξ_2 used in this work are taken from [2].

To achieve a physical theory, we take the limit as the regularization parameters approach their physical values. With a momentum cutoff, we take $\Lambda \to \infty$ and with dimensional regularization we usually take $D \to 4$. With lattice regularization, we approach the continuum, taking the lattice spacing $a \to 0$.

At this point, we have surely eliminated all divergences, right? Unfortunately, this is not always the case. External operators may also diverge due to regularization procedures. As we decrease the width of each lattice site, high frequency modes

become more significant, leading to ultraviolet divergences. A prototypical example is the angular momentum operator on the lattice. Since a square lattice breaks continuous rotational symmetry, orthogonal angular momentum operators mix on the lattice and can cause divergences [11].

The topological susceptibility χ_t is one such value that diverges in the continuum limit, though the reasons for this divergence are not fully understood [1]. This question is central to this work.

2.5.3 The Gradient Flow

To remove this ultraviolet divergence, we adopt a technique known as "smearing", a local averaging of the field [18]. Specifically, we use a technique known as the "gradient flow" [19] which introduces a new half-dimension[4] called "flow time". The flow time τ parameterizes the smearing such that an evolution in flow time corresponds to suppressing ultraviolet divergences.

Specifically, the gradient flow pushes field configurations toward classical minima of the action. Additionally, renormalized correlation functions remain renormalized at nonzero flow time for gauge theories such as QCD [20]. In 2D ϕ^4 scalar field theory, the gradient flow is defined by the differential equation

$$\frac{\partial \rho(\tau, x)}{\partial \tau} = \partial^2 \rho(\tau, x) \tag{2.29}$$

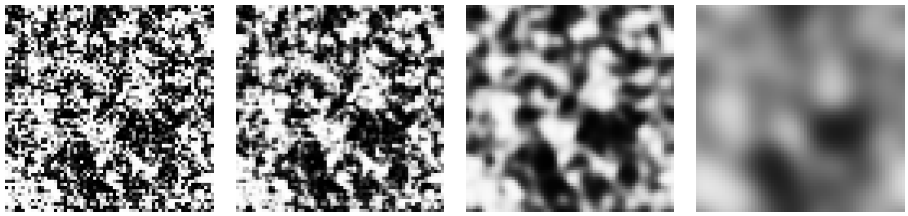
where ∂^2 is the Laplacian in 2D Euclidean spacetime[5] and τ is the flow time. Here, ρ is the field flowed into a nonzero flow time, bounded by the condition $\rho(\tau = 0, x) = \phi(x)$. In the ϕ^4 theory, we can solve this equation exactly to find [11]

$$\rho(\tau, x) = \frac{1}{4\pi\tau} \int d^2y \; e^{-(x-y)^2/4\tau} \phi(y). \tag{2.30}$$

[4]The term "half-dimension" indicates that the flow time is exclusively positive.

[5]Explicitly, $\partial^2 = \frac{\partial^2}{\partial t^2} + \nabla^2$

This function forms a Gaussian, smoothly dampening high-momentum modes and removing ultraviolet divergences from evolved correlation functions [21]. We can visualize this by plotting the ϕ field, shown in Fig. 2.2. These plots demonstrate the reduction of high momentum modes.



(a) $\tau = 0$ (b) $\tau = 0.001$ (c) $\tau = 0.01$ (d) $\tau = 0.1$

Figure 2.2: Effect of flow time evolution on a random lattice in the symmetric phase. White represents positive values of ϕ while black represents negative.

Generally, we can choose any flow time equation that drives the field towards a classical minimum. Following [2], we can define the gradient flow for the NLSM via the differential equation

$$\partial_\tau \vec{e}(\tau, x) = \left(1 - \vec{e}(\tau, x)\vec{e}(\tau, x)^T\right) \partial^2 \vec{e}(\tau, x). \tag{2.31}$$

We solve this equation numerically using the boundary condition $\vec{e}(\tau = 0, x) = \vec{e}(x)$, a process described in Sec. 3.4.

Chapter 3

Methods

Our study of the gradient flow in the NLSM is based on a computational system that simulates quantum fields numerically. We begin by implementing a numerical Monte Carlo method to simulate the lattice in two dimensions and verify our program with the well-studied ϕ^4 scalar field theory. We then generalize our model to a vector field to simulate the NLSM and implement a numerical solution to the gradient flow. We use the data produced by this program to study the topological charge and susceptibility under the gradient flow.

We first implement the ϕ^4 model in Python. Afterwards, we transition to the NLSM, using C++ for increased efficiency . To compile the C++ simulation, we use the `gcc` compiler with the highest level of optimization.

3.1 Fields on the Lattice

To implement the lattice regularization technique, we must redefine the field action in terms of discrete positions, a process known as "discretization". We transition from x as a continuous vector in $\mathbb{R}^2$ to $x_{i,j}$ where

$$x_{i,j} = ia\hat{t} + ja\hat{x}. \tag{3.1}$$

Here, a is the lattice constant and $\hat{t}$ and $\hat{x}$ are unit vectors. This change effectively shifts the domain of the field from $\mathbb{R}^2$, which is uncountably infinite, to $\mathbb{Z}^2$, which is countably infinite. To achieve a finite domain, we impose periodic boundary conditions, such that

$$\phi\left(x_{i+L,j}\right) = \phi\left(x_{i,j+L}\right) = \phi\left(x_{i,j}\right) \tag{3.2}$$

where L is the side length (in units of the lattice spacing a) of the system. In this study we focus solely on square geometries and thus the side length L is unambiguous.

In the ϕ^4 model, we can specify a discrete action using the Euclidean action from Eq. 2.10. We begin by redefining the derivative operator as a finite difference:

$$\partial_\mu \phi = \frac{\phi\left(x + a\hat{\mu}\right) - \phi(x)}{a}. \tag{3.3}$$

We can then define the kinetic term

$$\begin{aligned}
\frac{1}{2}\left(\partial_t \phi\right)^2 + \frac{1}{2}\left(\partial_x \phi\right)^2 \rightarrow & \frac{1}{2a^2}\left[\left(\phi(x + a\hat{t}\,) - \phi(x)\right)^2 + \left(\phi(x + a\hat{x}) - \phi(x)\right)^2\right] \\
\rightarrow & \frac{1}{2a^2}\left[\phi^2(x + a\hat{t}\,) + \phi^2(x + a\hat{x}) + 2\phi^2(x)\right. \\
& \left. -2\phi(x + a\hat{t}\,)\phi(x) - 2\phi(x + a\hat{x})\phi(x)\right]
\end{aligned} \tag{3.4}$$

Since we will eventually sum over all sites x, the periodic boundary conditions imply that an overall shift in x does not effect the final action. Therefore, we can combine the first two terms with the third term to produce

$$\frac{1}{2}\left(\partial_t \phi\right)^2 + \frac{1}{2}\left(\partial_x \phi\right)^2 \rightarrow \frac{1}{a^2}\left[2\phi^2(x) - \phi(x + a\hat{t}\,)\phi(x) - \phi(x + a\hat{x})\phi(x)\right] \tag{3.5}$$

Unlike the kinetic term, the mass and interaction terms remain unchanged under the discretization procedure. The only remaining change is a shift from an integral to a discrete sum. This transformation takes the form

$$\int dt dx \rightarrow a^2 \sum_i \tag{3.6}$$

such that the final discretized action becomes

$$S_{\text{lat}}[\phi] = \sum_i \left[-\phi(x_i + a\hat{t}\,)\phi(x_i) - \phi(x_i + a\hat{x})\phi(x_i) + \left(2 + \frac{1}{2}m_0^2\right)\phi^2(x_i) + \frac{1}{4}\lambda\phi^4(x_i) \right] \tag{3.7}$$

Likewise, we can discretize the NLSM. In this case, the derivative term becomes

$$\frac{1}{2}\left(\partial_t \vec{e}\right)^2 + \frac{1}{2}\left(\partial_x \vec{e}\right)^2 \to \frac{1}{a^2}\left[2 - \vec{e}(x + a\hat{t}\,)\cdot\vec{e}(x) - \vec{e}(x + a\hat{x})\cdot\vec{e}(x)\right]. \tag{3.8}$$

Note that we have used the identity $\vec{e}\cdot\vec{e} = 1$. Inserting this into Eq. 2.11 yields the discretized action

$$S_{\text{lat}}[\vec{e}] = \sum_i \left[2 - \vec{e}(x + a\hat{t}\,)\cdot\vec{e}(x) - \vec{e}(x + a\hat{x})\cdot\vec{e}(x)\right]. \tag{3.9}$$

Finally, we redefine the gradient flow in on the lattice. Since the gradient flow is solved exactly in the ϕ^4 model, we rely on a discrete Fourier transform. This method isolates the momentum modes of the field and dampens them by a factor of $e^{-\tau p^2}$ where τ is the flow time and p is the momentum. In the NLSM, the definition of the gradient flow (Eq. 2.31) becomes

$$\partial_\tau \vec{e}(\tau, x) = \left(1 - \vec{e}(\tau, x)\vec{e}(\tau, x)^T\right)\partial^2\vec{e}(\tau, x), \tag{3.10}$$

where the Laplacian operator ∂^2 is now defined as

$$\partial^2\vec{e}(\tau, x) = \vec{e}(\tau, x + a\hat{t}) + \vec{e}(\tau, x - a\hat{t}) + \vec{e}(\tau, x + a\hat{x}) + \vec{e}(\tau, x - a\hat{x}) - 4\vec{e}(t, x).$$

Unlike the ϕ^4 gradient flow, this differential equation has no analytic solution. Therefore, we numerically solve for the gradient flow using a fourth-order Runge-Kutta approximation (see Sec. 3.4).

3.1.1 Discretized Observables

Following the definitions in Sec. 2.3.1, we redefine the primary and secondary observables on the discretized lattice. Using the discretization definition in Eq. 3.6, we

express the average magnetization as

$$|\bar{\phi}| \equiv \frac{1}{L^2}\left|\sum_i^{L^2} \phi(x_i)\right|. \tag{3.11}$$

in the ϕ^4 model and

$$|\bar{\vec{e}}| \equiv \frac{1}{L^2}\left|\sum_i^{L^2} \vec{e}(x_i)\right|. \tag{3.12}$$

in the NLSM, where $L^2 = V/a^2$.

We can also rewrite the expressions for the magnetic susceptibility, originally defined in Eqs. 2.21 and 2.22. In the NLSM, the susceptibility becomes

$$\chi_m = L^2 \left\langle \left(\sum_x \vec{e}(x)\right)^2 \right\rangle \tag{3.13}$$

$$= L^2 \left\langle \sum_{x,y} \vec{e}(x)\cdot\vec{e}(y) \right\rangle. \tag{3.14}$$

Due to translational invariance from the periodic boundary conditions, we simplify this expression to be

$$\chi_m = \left\langle \sum_x \vec{e}(0)\cdot\vec{e}(x) \right\rangle. \tag{3.15}$$

Following the same logic, we derive

$$\chi_m = \left\langle \sum_x \phi(0)\phi(x) \right\rangle \tag{3.16}$$

for the ϕ^4 model.

The definitions for the internal energy, Binder cumulant and bimodality remain unchanged on the lattice.

3.2 Defining the Topological Charge

On the lattice, the topological charge is nontrivial to calculate. Primarily, there are multiple possible mappings between field configurations and the integers. In this work, we use the definition found in [1]. To begin, we define a local topological charge

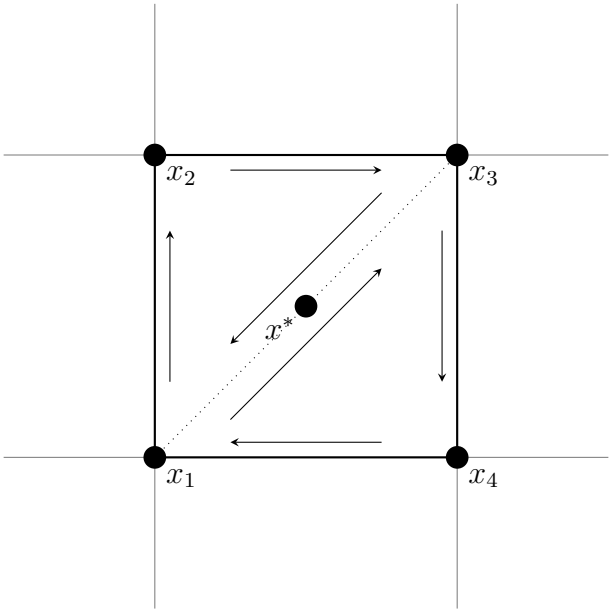


Figure 3.1: Visualization of plaquette x^*. The dotted line separates the plaquette into two signed areas which are used to define the topological charge density $q(x^*)$. Arrows represent order of signed area.

density q, defined for each square of adjacent lattice points. This square, known as a *plaquette*, is denoted x^*. The global charge Q is the sum of all local charges:

$$Q \equiv \sum_{x^*} q(x^*). \tag{3.17}$$

As a function of x^*, the charge density is a function of the field on the plaquette vertices, an idea visualized in Fig. 3.1. In the NLSM, the field at each of these vertices is a point on the sphere S^2. Therefore, there is a signed area A on the sphere associated with each triplet of points, as shown in Fig. 3.2 (the sign changes with odd permutations of the ordering). We follow the derivation in [1] and split the plaquette into two triangles, as shown in Fig. 3.1, with the ordering determining the sign. The topological charge density is defined using this signed area as

$$q(x^*) = \frac{1}{4\pi}\left[A\Big(\vec{e}(x_1), \vec{e}(x_2), \vec{e}(x_3)\Big) + A\Big(\vec{e}(x_1), \vec{e}(x_3), \vec{e}(x_4)\Big)\right]. \tag{3.18}$$

This value is defined if $A \neq 0, 2\pi$, or in other words, as long as the three points on the sphere are distinct and do not form a hemisphere. In numerical calculations, these

points can be ignored. Therefore, we impose that the signed area is defined on the smallest spherical triangle, or mathematically

$$-2\pi < A < 2\pi. \tag{3.19}$$

Following [1], this yields an expression for the signed area

$$A(\vec{e}_1, \vec{e}_2, \vec{e}_3) = 2\arg\Big(1 + \vec{e}_1 \cdot \vec{e}_2 + \vec{e}_2 \cdot \vec{e}_3 + \vec{e}_3 \cdot \vec{e}_1 + i\vec{e}_1 \cdot (\vec{e}_2 \times \vec{e}_3)\Big). \tag{3.20}$$

Under periodic boundary conditions, these triangles on the sphere necessarily wrap S^2 an integer number of times, ensuring Q is an integer. In the continuum limit, field must be uniform at $x \to \infty$ and therefore effectively forms a sphere (see Sec. 2.4). In

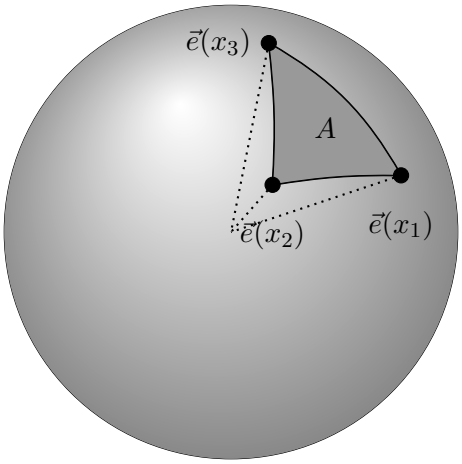


Figure 3.2: Visualization of signed area A on the sphere S^2 traced out by field at points x_1, x_2 and x_3.

the NLSM described by Eq. 3.9, the values of Q are roughly Gaussian around $Q = 0$, as shown in Fig. 3.3.

Following this quantity, we can define a topological susceptibility χ_t

$$\chi_t \equiv \frac{1}{L^2}\Big(\langle Q^2\rangle - \langle Q\rangle^2\Big). \tag{3.21}$$

In the trivial case, $\langle Q\rangle$ is equal to 0 and

$$\chi_t = \frac{1}{L^2}\sum_{x^*, y^*}\langle q(x^*)q(y^*)\rangle. \tag{3.22}$$

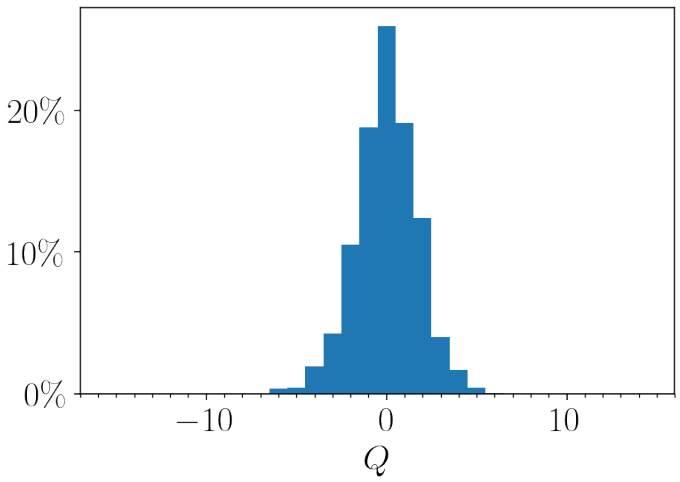


Figure 3.3: Histogram of topological charge values Q for trivial NLSM. $L = 404$, 10,000 measurements, measurements very 50 sweeps, 1,000 sweep thermalization, $\tau = 0$

Assuming periodic boundary conditions and therefore translational symmetry, this expression simplifies to

$$\chi_t = \frac{1}{L^2} \sum_{x^*} \langle q(x^*) q(0) \rangle. \tag{3.23}$$

On the lattice, this quantity is known to diverge in the continuum limit owing solely to the $x^* = 0$ term [2]. This divergence exists in QCD as well [10].

3.3 Monte Carlo Simulations

We implement a Markov Chain Monte Carlo method following Schaich's thesis [22]. This implementation utilizes a "random walk," i.e. a set of random steps through phase space, to determine statistical values such as correlation functions across the lattice. By the definition of the Markov chain, the probability of adoption of each state, and therefore its inclusion in the Monte Carlo calculation, depends only on the current state and the proposed state. This probability is denoted as $P(\phi_a \to \phi_b)$ where ϕ_a and ϕ_b are the existing and proposed lattice configurations respectively.

We begin this random walk with a so-called "hot start" where each field value at

each lattice site is randomly selected. As an alternative to the hot start, we could use a "cold start" where the field begins completely aligned. With an appropriate thermalization, these initial configurations have no effect on the Monte Carlo statistics, a fact we verify in Sec. 3.3.4.

3.3.1 Metropolis Algorithm

We primarily use the Metropolis algorithm for the calculation of new Markov chain configurations. The method begins with proposing a new value for a single lattice point, which is accepted with a probability

$$P(\phi_a \to \phi_b) = \begin{cases} e^{S[\phi_a]-S[\phi_b]} & S[\phi_b] > S[\phi_a] \\ 1 & \text{otherwise} \end{cases} \tag{3.24}$$

where ϕ_a is the initial configuration and ϕ_b is the proposed configuration. This process is performed for each point on the lattice, making up a "sweep". Repeating this sweep multiple times pushes the lattice toward the action minimum.

3.3.2 Wolff Cluster Algorithm

Though the Metropolis algorithm will slowly find the absolute minimum of the theory, the presence of local minima can greatly prolong the convergence. Both the ϕ^4 model and the NLSM feature gradients of the field and therefore large regions of aligned sites can cause metastable states. One method of removing these clusters involves identifying all clusters on the lattice and probabilistically flipping each, a technique known as the Swendsen-Wang algorithm [23].

A more efficient approach is the Wolff algorithm [24], which *grows* one cluster probabilistically and flips it unconditionally. In the case of ϕ^4 theory, this flipping takes the form of a simple sign change. In the NLSM we choose a random unit vector $\vec{r}$ and consider the projection of the field on this vector. When the cluster flips, each site is flipped along this direction. To identify the cluster, a recursive algorithm

adds new sites with a given probability, growing the cluster from a single randomly selected "seed". Starting with the seed, the probability of adding each neighboring site is given by the source site x and the proposed site x'. Wolff defines this probability for arbitrary sigma models as

$$P_{add}(\vec{e}(x), \vec{e}(x')) = \begin{cases} 1 - e^{-2\beta[\vec{r}\cdot\vec{e}(x)][\vec{r}\cdot\vec{e}(x')]} & \text{sgn}[\vec{r}\cdot\vec{e}(x)] = \text{sgn}[\vec{r}\cdot\vec{e}(x')] \\ 0 & \text{otherwise} \end{cases} \tag{3.25}$$

This expression is designed to preserve the detailed balance equations. We can demonstrate this quality, and motivate an equivalent expression for the ϕ^4 model, by considering the probability $P(\phi \to f_C(\phi))$ of flipping some cluster C. Generally,

$$P\big(\phi \to f_C(\phi)\big) \propto \prod_{\langle x,x'\rangle \in \partial C} \Big[1 - P_{add}\big(\vec{e}(x), \vec{e}(x')\big)\Big] \tag{3.26}$$

where ∂C is the set of pairs of sites on the boundary of C. Since $P_{add} = 0$ for unaligned sites, these pairs contribute nothing to the value. We can also find the probability $P(f_C(\phi) \to \phi)$ with the same expression:

$$P\big(f_C(\phi) \to \phi\big) \propto \prod_{\langle x,x'\rangle \in \partial C} \Big[1 - P_{add}\big(R\,\vec{e}(x), \vec{e}(x')\big)\Big], \tag{3.27}$$

where the matrix R is the reflection matrix along the vector $\vec{r}$.

From the discretized action of the NLSM model (Eq. 3.9) and the detailed balance equation (Eq. 2.15), we derive

$$\prod_{\langle x,x'\rangle \in \partial C} \frac{1 - P_{add}\big(\vec{e}(x), \vec{e}(x')\big)}{1 - P_{add}\big(R\,\vec{e}(x), \vec{e}(x')\big)} = \exp\left\{-\beta \sum_{\langle x,x'\rangle \in \partial C} [\vec{e}(x) - R\,\vec{e}(x)] \cdot \vec{e}(x')\right\}. \tag{3.28}$$

Note that all the pairs within and outside the cluster cancel in the fraction on the left and the difference on the right. Using the definition of the reflection matrix

$$R\,\vec{e} = \vec{e} - 2(\vec{e}\cdot\vec{r})\vec{e}, \tag{3.29}$$

we can simplify the equation to be

$$\prod_{\langle x,x'\rangle \in \partial C} \frac{1 - P_{add}\big(\vec{e}(x), \vec{e}(x')\big)}{1 - P_{add}\big(R\,\vec{e}(x), \vec{e}(x')\big)} = \prod_{\langle x,x'\rangle \in \partial C} \exp\big\{-2\beta[r\cdot\vec{e}(x)][r\cdot\vec{e}(x')]\big\}. \tag{3.30}$$

By substituting Eq. 3.25 for P_{add}, it is clear to see this equation is satisfied.

Using this same reasoning, we can deduce an expression for P_{add} in the ϕ^4 model. Since this model is one dimensional, the reflection matrix R is equivalent to -1. Adapting the NLSM detailed balance equation for the ϕ field, we find

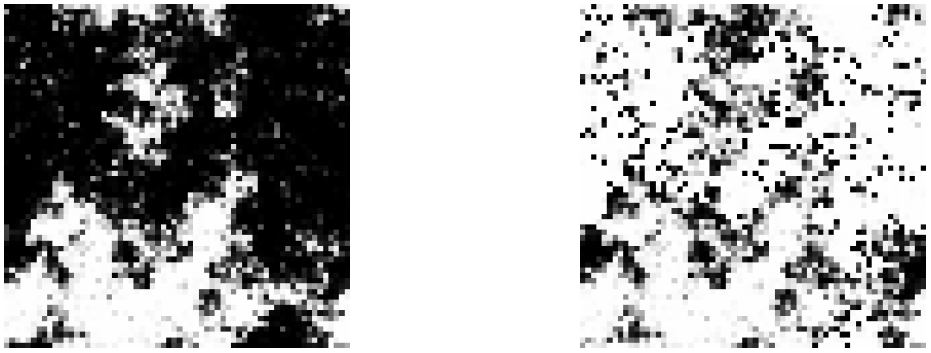
$$\prod_{\langle x,x'\rangle\in\partial C} \frac{1-P_{add}\big(\phi(x),\phi(x')\big)}{1-P_{add}\big(\phi(x),-\phi(x')\big)} = \prod_{\langle x,x'\rangle\in\partial C} \exp\{-2\phi(x)\phi(x')\}. \tag{3.31}$$

This equation is satisfied by the ansatz

$$P_{add}(\phi(x),\phi(x')) = \begin{cases} 1-e^{-2\phi(x)\phi(x')} & \mathrm{sgn}[\phi(x)] = \mathrm{sgn}[\phi(x')] \\ 0 & \text{otherwise} \end{cases}. \tag{3.32}$$

We use this expression in the computational implementation of this algorithm.

Fig. 3.4 shows a real demonstration of this process. We can see a large cluster of negative field values becoming positive. Note that periodic boundary conditions apply, so the small peninsula of black at the bottom is actually part of the larger cluster. Furthermore, this visualization demonstrates the probabilistic nature of the Wolff algorithm. Since states are added probabilistically, there are some small holes in the cluster. These will be removed by following Metropolis sweeps.



(a) before cluster flip (b) after cluster flip

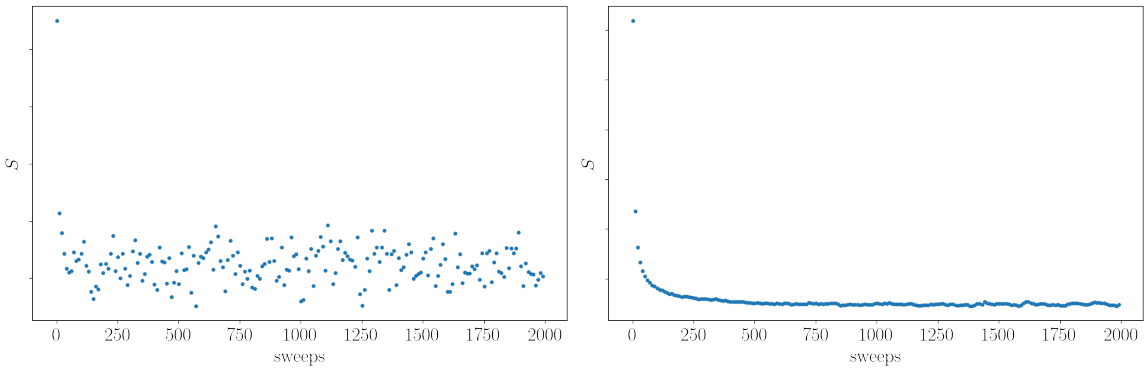
Figure 3.4: An example of the Wolff cluster algorithm in the ϕ^4 model. White represents positive values of ϕ while black represents negative. $\lambda = 0.5$, $m_0^2 = -0.9$.

3.3.3 Checkerboard Algorithm

In order to parallelize the Metropolis algorithm, we use a checkerboard algorithm. We begin by splitting the lattice into "white" sites and "black" sites, like the tiles on a checkerboard. Since the Lagrangian density at each site does not depend on diagonal neighbors, each white site is independent of every other white site and likewise for black sites. Therefore, we can split the sites of each color into separate parallel processing nodes and independently run the Metropolis algorithm, ensuring that no site affects the Lagrangian density at any other site. We use this method to parallelize our program through the Message Passing Interface (MPI).

3.3.4 Thermalization

In order to determine the necessary thermalization, we plot the action as a function of Metropolis sweeps in Fig. 3.5. Based on this plot, we determine that 1000 sweeps



(a) $L = 24$ (b) $L = 404$

Figure 3.5: Plots of the action as a function of Monte Carlo time, starting with a random NLSM lattice.

will give sufficient time for the system to reach the classical action minimum. We use this value for the remainder of this study.

We also compare the hot and cold starts by plotting a histogram of the actions in

Fig. 3.6. This plot qualitatively demonstrates the irrelevance of the initial configuration.

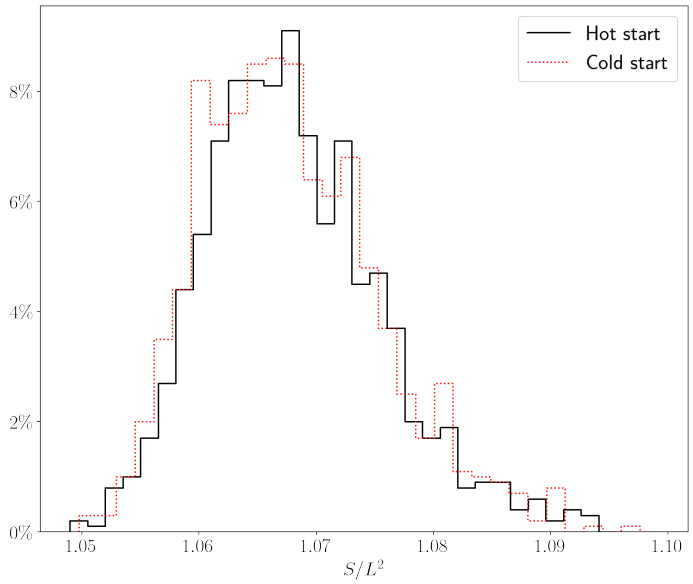


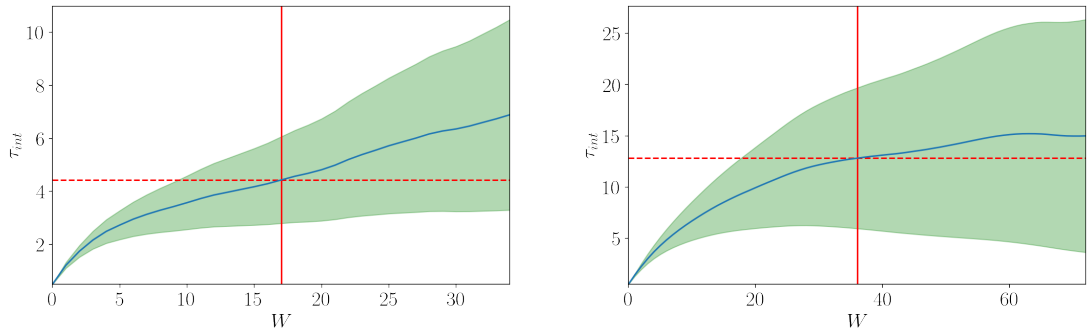
Figure 3.6: Histogram of lattice-averaged actions S/L^2 with hot and cold starts. 1,000 sweep thermalization in $L = 404$ lattice, 1,000 measurements taken every 50 sweeps.

3.3.5 Autocorrelation

Due to the nature of the Markov chain, each member of the ensemble is correlated to every other. Since each configuration is based on previous configurations, each pair of members in the Markov chain has a correlation which decreases exponentially based on the number of steps between. This value is known as the "autocorrelation" and scales as $e^{-t/\tau_{int}}$ where t is the number of steps between configurations and τ_{int} is the autocorrelation time.[1] When performing simulations of the lattice, the number of sweeps between measurements should be much larger than τ_{int} since Monte Carlo methods generally assume independent observations.

[1]Though it is called a time, τ_{int} is in units of Markov Chain steps.

We use Wolff's automatic windowing procedure [25] and the magnetic susceptibility χ_m to estimate the autocorrelation. Using Wolff's public MatLab code[2], we estimate the autocorrelation time for $L = 24$ and $L = 404$ lattices. This algorithm identifies the optimal window size with which to calculate the autocorrelation time. We perform this process on $L = 24$ and $L = 404$ lattices using a thermalization of 1000 sweeps and 500 total measurements. Note that this calculation includes a Wolff cluster algorithm every 5 sweeps. The result of this calculation is shown in Fig. 3.7.



(a) $L = 24$, $\tau_{int} = 4.43$ (b) $L = 404$, $\tau_{int} = 12.81$

Figure 3.7: Plots of automatic windowing procedure used to calculate τ_{int} for the NLSM model. W is summation window size.

Based on these two values for τ_{int}, we decide to measure every 50 sweeps for each simulation. This value will ensure that each measurement is effectively independent.

3.4 Runge-Kutta Algorithm

In order to calculate the gradient flow in the NLSM, we numerically solve the ordinary differential equation in Eq. 3.10 using a fourth-order Runga-Kutta approximation. This algorithm refines the Euler method

$$\vec{e}(\tau + h, x) \approx \vec{e}(\tau, x) + hf(\vec{e}(\tau, x))$$

[2] `https://www.physik.hu-berlin.de/de/com/ALPHAsoft`

where $f(\vec{e})$ is defined for convenience as

$$f(\vec{e}) = \partial_\tau \vec{e}(\tau, x) = \left(1 - \vec{e}(\tau, x)\vec{e}(\tau, x)^T\right) \partial^2 \vec{e}(\tau, x), \tag{3.33}$$

following from Eq. 3.10. To the fourth order, this approximation becomes

$$k_1 = hf\left(\vec{e}\,(\tau, x)\right) \tag{3.34}$$

$$k_2 = hf\left(\vec{e}\,(\tau, x) + \frac{k_1}{2}\right) \tag{3.35}$$

$$k_3 = hf\left(\vec{e}\,(\tau, x) + \frac{k_2}{2}\right) \tag{3.36}$$

$$k_4 = hf\left(\vec{e}\,(\tau, x) + k_3\right) \tag{3.37}$$

$$\vec{e}(\tau + h, x) = \vec{e}(\tau, x) + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5). \tag{3.38}$$

This method is usually superior to Euler's method and the midpoint method in accuracy [26].

To increase the efficiency of this algorithm, we implement the step-doubling algorithm to adaptively adjust h. If the error of a Runge-Kutta step is greater than the tolerance, the same step is repeated with half the step size. Alternatively, if the error is less than half of the tolerance, the step size is doubled for the next calculation. Finally, if the step size is greater than the distance to the next measurement, that distance is used as the step size, using the normal value afterwards. Otherwise, the algorithm proceeds with the consistent step size.

To calculate the error, we compare one lattice $\vec{e}_1$ produced using a step of size $2h$ with another lattice $\vec{e}_2$ produced via two steps of size h. The error Δ can be estimated to up the fifth order of h as [26]

$$\Delta = \frac{1}{15}\sqrt{\sum_x \left|\vec{e}_2(x) - \vec{e}_1(x)\right|} \tag{3.39}$$

The tolerance used in this work is $\Delta_{max} = 0.01$.

3.5 Topological Charge with a θ-term

In Sec. 2.4, we discussed the introduction of a θ term into the action. This change makes the theory "topological", pushing $\langle Q \rangle$ away from zero. In order to calculate $\langle Q \rangle$ as a function of Q, we consider the path integral

$$\langle Q \rangle_\theta = \int \mathcal{D}\vec{e}\, Q[\vec{e}] e^{-S[\vec{e}]+i\theta Q[\vec{e}]} \tag{3.40}$$

$$= \int \mathcal{D}\vec{e}\, \left(Q[\vec{e}] e^{i\theta Q[\vec{e}]} \right) e^{-S[\vec{e}]} \tag{3.41}$$

$$= \langle Q e^{i\theta Q} \rangle_{\theta=0}. \tag{3.42}$$

Therefore, we can calculate $\langle Q \rangle_\theta$ for arbitrary θ using the same simulation framework as the $\theta = 0$ case.

We can also relate this function to the topological susceptibility. By expanding the exponent as a Taylor series around $\theta = 0$, we find that

$$\langle Q \rangle_\theta = \langle Q \rangle_{\theta=0} + i\theta \langle Q^2 \rangle_{\theta=0} + O(\theta^2), \tag{3.43}$$

such that

$$\mathrm{Im}\left[\frac{\partial}{\partial\theta}\Big|_{\theta=0} \langle Q \rangle_\theta \right] = \langle Q^2 \rangle_{\theta=0} \tag{3.44}$$

$$= \chi_t L^2. \tag{3.45}$$

Since χ_t diverges at $\theta = 0$, we expect the plot of $\mathrm{Im}\langle Q \rangle_\theta$ to approach a vertical line in the in continuum limit. The numerical demonstration of this hypothesis is a goal of this work.

Chapter 4

Results

Using the Monte Carlo procedure outlined in Sec. 3, we run statistical simulations on the ϕ^4 model and NLSM.

4.1 ϕ^4 Results

We initially implement the ϕ^4 model to verify the results of our system. According to previous studies [11, 15, 22], the ϕ^4 model exhibits a symmetric and broken phase depending on its parameters m_0^2 and λ, transitioning at $m_0^2 = -0.72$ when $\lambda = 0.5$. We verify this result by plotting four observables: the lattice average $|\langle\bar{\phi}\rangle|$, the magnetic susceptibility χ_m, the Binder cumulant U and the bimodality B in Fig. 4.1. This figure confirms a phase transition near $m_0^2 = -0.7$ and supports the accuracy of our model.

4.2 Non-linear Sigma Model Results

After confirming the phase transition in the ϕ^4 model, we turn to the NLSM. In order to confirm the accuracy of the model, we compare results with existing literature. We first consider the results of Berg & Lüscher [1], specifically the internal energy and magnetic susceptibility.

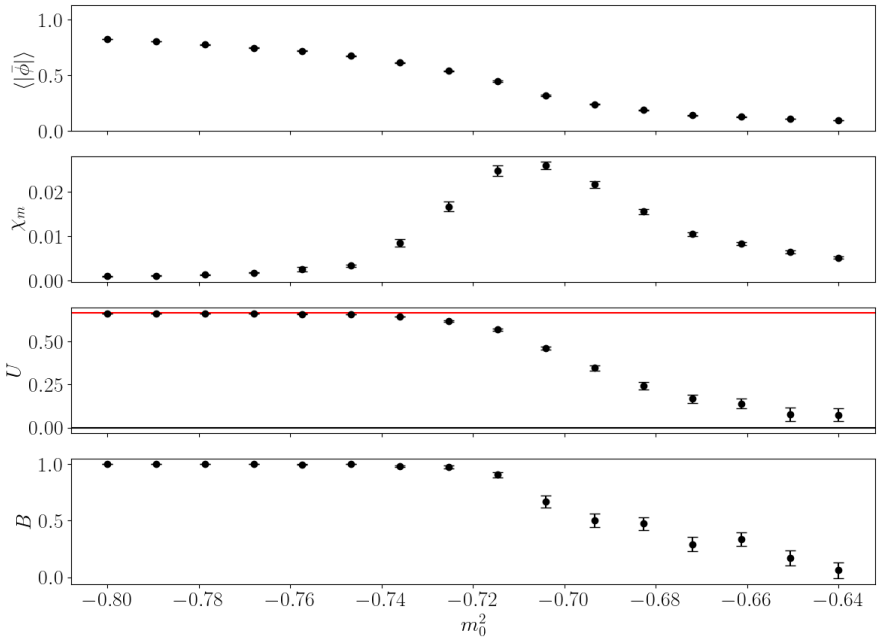


Figure 4.1: The lattice average $|\langle\bar{\phi}\rangle|$, the magnetic susceptibility χ_m, the Binder cumulant U and the bimodality B plotted as functions of m_0^2. $L = 64$, $\lambda = 0.5$. The lattice was thermalized from a hot start for 1000 sweeps. Afterwards, 1000 measurements were taken with 50 sweeps between each. The red horizontal line indicates $U = 2/3$, the broken phase limit of the Binder cumulant.

4.2.1 Comparison with Existing Literature

Following [1], we approximate the internal energy in the strong ($\beta < 1$) and weak ($\beta > 2$) regimes as

$$E \approx \begin{cases} 4 - 4y - 8y^3 - \frac{48}{5}y^5 & \beta < 1 \\ \frac{2}{\beta} + \frac{4}{\beta^2} + 0.156\frac{1}{\beta^3} & \beta > 2 \end{cases} \tag{4.1}$$

where

$$y = \coth\beta - \frac{1}{\beta}. \tag{4.2}$$

We compare this analytical result and simulated values of χ_m with the Monte Carlo simulation in Fig. 4.2. These two charts show a high degree of agreement with the current literature however there is a slight discrepancy, prehaps arising from different

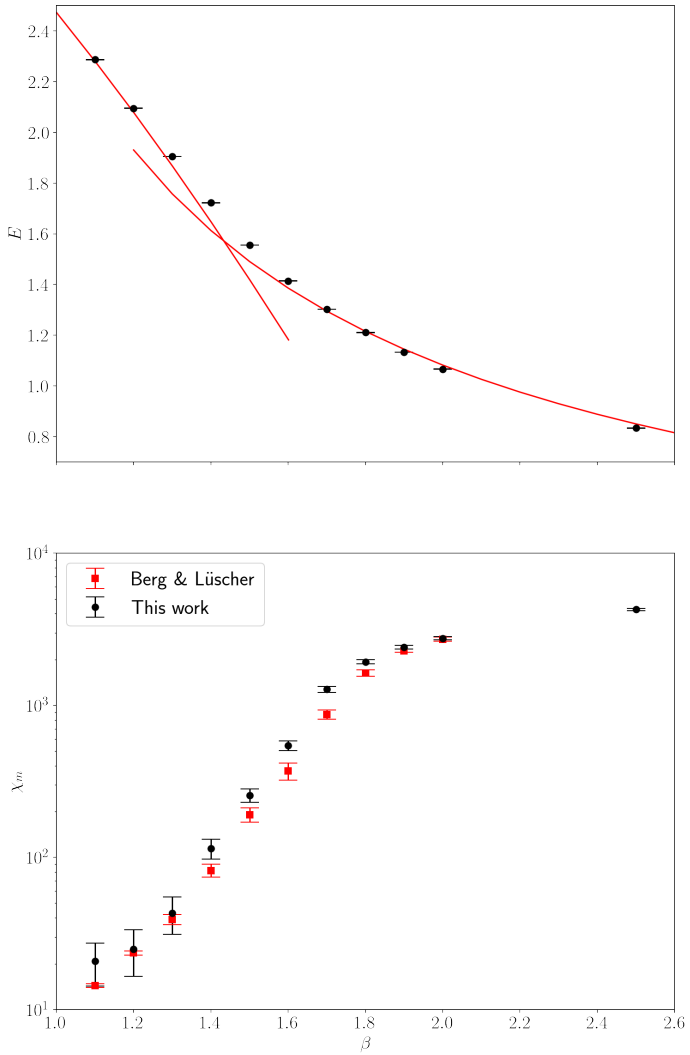


Figure 4.2: Comparison with [1]. First panel: internal energy compared with analytic energy (Eq. 4.1). Second panel: magnetic susceptibility compared with literature values.

Monte Carlo methods.

We also seek to confirm the results from Bietenholz et al. [2]. Specifically, we show the topological susceptibility χ_t diverges in the continuum limit even at finite flow time. Since χ_t is in units of inverse distance squared, we multiply by ξ_2^2, the square of the second moment correlation length, to achieve a scale-invariant value $\chi_t \xi_2^2$. Additionally, we use a parameter t_0 to scale the flow time such that $t_0 \sim L^2$.

In our Monte Carlo simulation, we utilize the same values as [2] for ξ_2, β and t_0.

To begin the comparison, we plot $\chi_t \xi_2^2$ as a function of flow time τ, shown in Fig. 4.3. We find that the flow time effectively decreases the topological susceptibility by dampening high-momentum modes. To analyze the divergence of χ_t in the continuum limit, we plot $\chi_t \xi_2^2$ as a function of lattice size L. We perform this simulation at flow times $\tau = 0$ and $\tau = 5t_0$ (Fig. 4.4). We fit the data with two options: a log fit

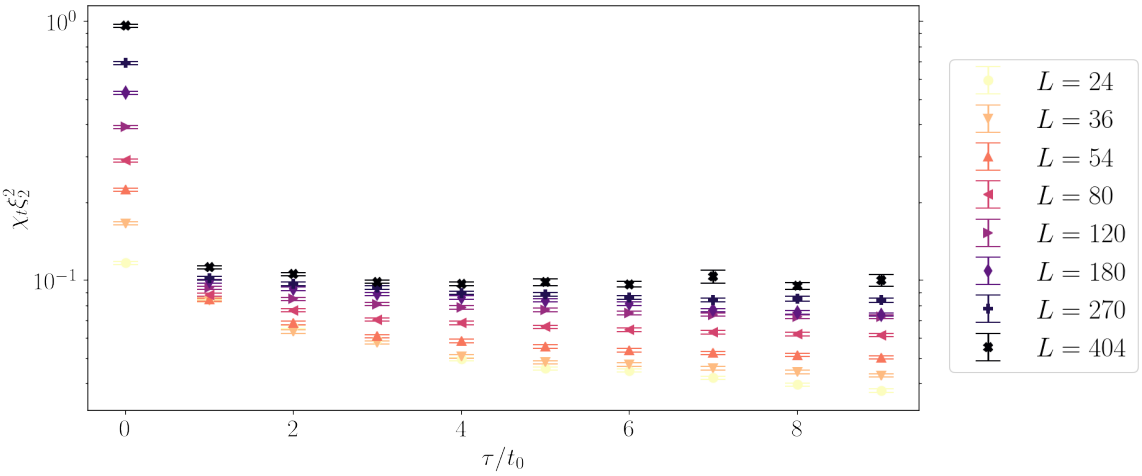


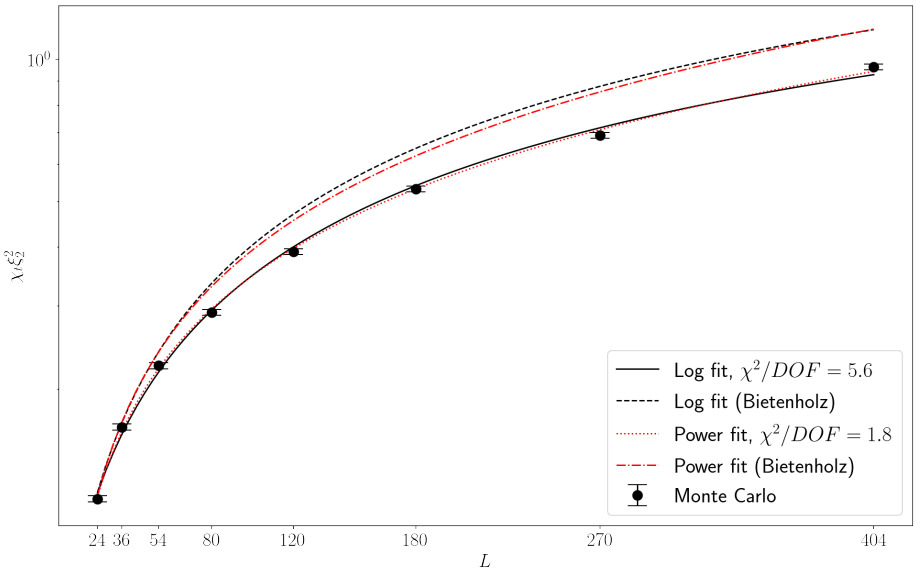
Figure 4.3: $\chi_t L^2$ as a function of flow time τ. Simulation run with 10,000 measurements every 50 sweeps, 1,000 sweep thermalization.

$$\chi_t \xi_2^2 = a\log(bL + c) \tag{4.3}$$

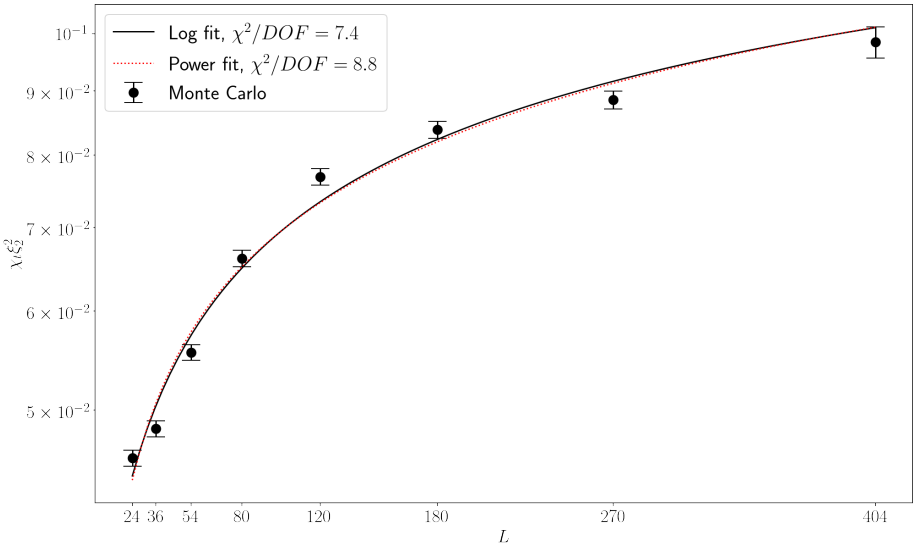
and a power law fit

$$\chi_t \xi_2^2 = aL^b + c. \tag{4.4}$$

We calculate the parameters to these functions using the `curve_fit` tool in the `scipy` Python package [27]. When $\tau = 5t_0$, the data fits these functions with χ^2/DOF of 7.4 and 8.8 respectively, indicating errors were underestimated. Both of these functions diverge as $L \to \infty$, indicating that the topological susceptibility also diverges in the continuum limit. Though there is a clear difference between the quantitative fit from



(a) $\tau = 0$



(b) $\tau = 5t_0$

Figure 4.4: $\chi_t\xi_2^2$ as a function of L. We fit the data with both a logarithmic and power fit. Simulation run with 10,000 measurements, once every 50 sweeps, 1,000 sweep thermalization. In the $\tau = 0$ case, we have compared our result with the curve fit found in [2].

[2] and the fit calculated in this work, both demonstrate divergent behavior. This result supports the inherent divergence of χ_t in the continuum limit.

4.2.2 Topological Charge when $\theta \neq 0$

Following the method explained in Sec. 3.5, we calculate the imaginary part of $\langle Q \rangle$ for arbitrary θ. We perform this calculation for three values of the flow time τ, shown in Fig 4.5. These plots demonstrate the divergence of the continuum limit in the $\tau = 0$

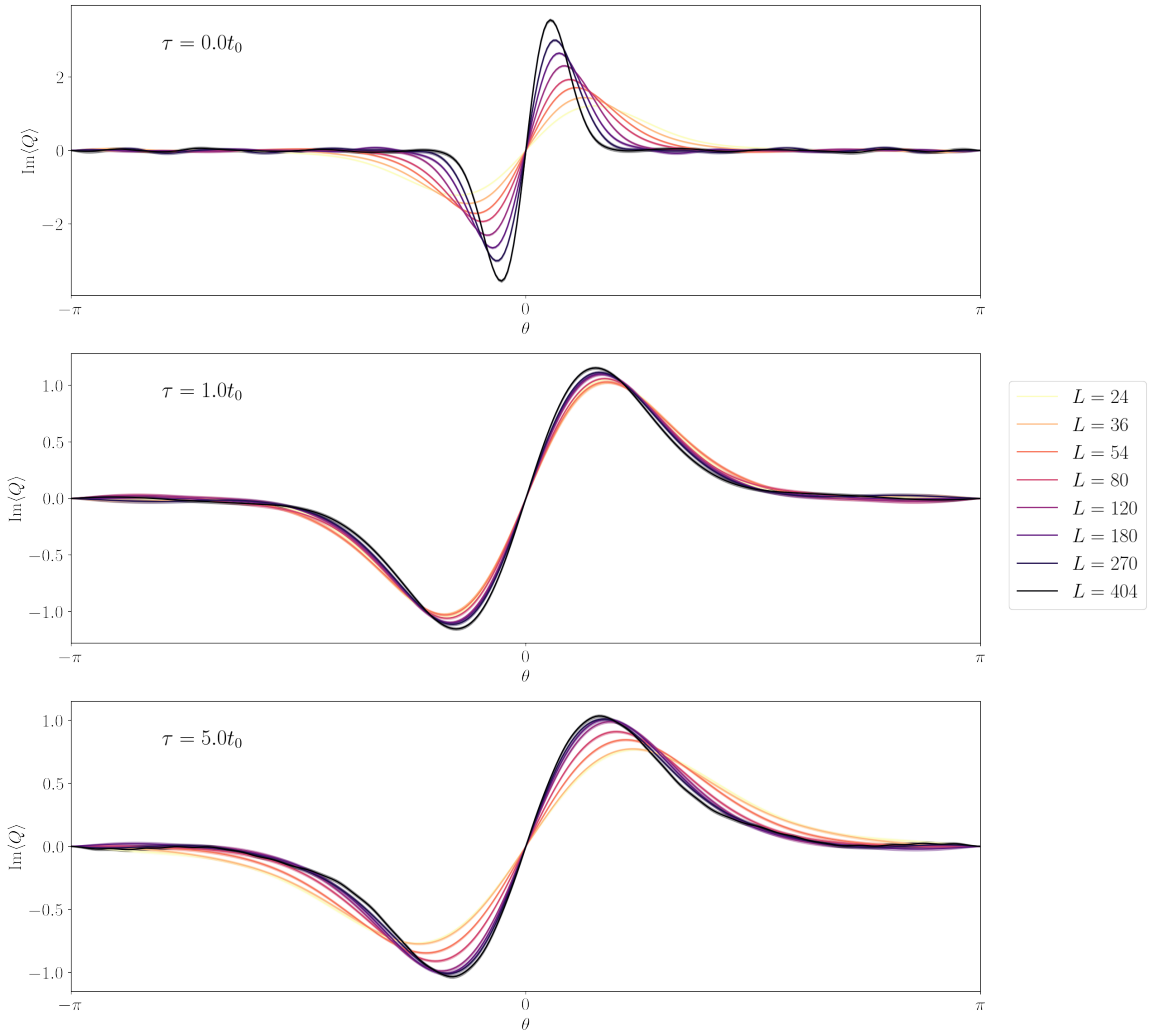


Figure 4.5: Imaginary part of $\langle Q \rangle$ as a function of θ. Simulation run with 10,000 measurements, measurements very 50 sweeps, 1,000 sweep thermalization. Note the different scaling of the y-axis.

and the flowed regimes. In the $\tau = 0$ case, the slope increases sharply, reflecting the rapid divergence of χ_t. However in the flowed regime, this divergence is much slower,

reflecting the decreased values of χ_t shown in Fig. 4.3.

4.3 Implications

Berg & Lüscher [1] originally illuminated this discrepancy between the renormalization group hypothesis (that $\chi_t \to 0$) and the numerical results, providing three possible causes:

1. The definition of the topological charge does not scale to the continuum.

2. There are ultraviolet divergences.

3. There is no reasonable continuum limit.

Since the gradient flow suppresses ultraviolet fluctuations, the persistence of a divergent topological susceptibility under the gradient flow undermines the second option, thereby supporting the other two.

Chapter 5

Conclusion & Outlook

Using a Monte Carlo simulation, we have analyzed two main quantum field theories in 1+1 spacetime dimensions: the ϕ^4 model and the $O(3)$ non-linear sigma model. We used the path integral formulation of QFT to simulate these quantum fields as statistical systems on a Euclidean lattice and extract information about the phase transitions and topology. To perform these calculations, we run a series of Metropolis sweeps interspersed with Wolff cluster steps, creating a Markov chain of samples. We ensure that each measurement is effectively independent by measuring the autocorrelation and thermalize the lattice to keep all measurements near the action minima.

In the ϕ^4 model, we verified a phase transition at $m_0^2 = -0.7$ using the magnitude of the average magnetization, the magnetic susceptibility, the Binder cumulant and the bimodality. This process verified the effectiveness of the Monte Carlo computational system. We then generalized our system to the $O(3)$ non-linear sigma model and transitioned to a C++ code base. We confirmed our calculations with known results by measuring the internal energy and magnetic susceptibility. We calculated the topological susceptibility χ_t to analyze its divergence in the continuum limit, confirming that this is indeed the case even at finite flow time. Specifically, the topological susceptibility under the gradient flow follows either a power law relationship or a logarithmic relationship as a function of lattice size, both of which diverge as

$L \to \infty$. At flow time $\tau = 5t_0$, the χ^2/DOF goodness of fit values are of 8.8 and 7.4 respectively.

Finally, we demonstrated the relationship between the topological charge and the θ in the topological action. This plot demonstrates quantitatively the different rates of divergence in the topological susceptibility. In the $\tau = 0$ case, we see the slope rapidly approach positive infinity at $\theta = 0$. While this transition occurs more slowly with the application of the gradient flow, the slope continues to increase.

Though the logarithmic and power-law functions visually fit the susceptibility data well, the χ^2 values are high. This imprecision can be attributed to underestimated errors. Since the $\tau = 0$ case features a more acceptable fit ($\chi^2/DOF = 1.8$ for the power-law), the gradient flow seemingly contributes to this error. While the Jackknife method accurately estimates statistical errors of the sample, we did not incorporate any systematic errors arising from the gradient flow calculation. Future work could reduce the tolerance of the adaptive step size algorithm to make this calculation more accurate, though this change increases computational requirements substantially. Furthermore, a larger number of lattice sizes may provide a more complete picture of the continuum limit.

Procedurally, we found that the MPI parallelization and checkerboard algorithm were unnecessary in this calculation. The computation time of the Runge-Kutta algorithm for computing the gradient flow far outweighed that of the Monte Carlo simulation. A possible improvement could be a parallelization of the Runge-Kutta algorithm or a more accurate approximation to the gradient flow. Additionally, the performance of the Python simulation was slower by up to two orders of magnitude. Future studies should therefore rely solely on an efficient programming language for Monte Carlo techniques.

In the context of the long-standing question of topological suscpetibility in the

NLSM, this study has diminished the plausibility that ultraviolet fluctuations cause the divergence. Instead, we are left to consider the other two options outlined in [1]: that the definition of the topological charge density is problematic or that the NLSM does not have a decent continuum limit. Future work could include similar numerical calculations using a different definition of the topological charge.

Generally, this study has implications in both condensed matter and nuclear physics. Though the calculation of the χ_t divergence confirms existing literature, the relationship between the topological charge and θ in the flow time was previously unexplored. Beyond the convergence of χ_t, this relationship has applications in condensed matter where different values of θ can describe spin-chains of either fermions or bosons [28]. Furthermore, the mass gap has a strong relationship with the θ-term in the topological NLSM, featuring massive and massless regimes [12].

Appendix A

C++ NLSM Monte Carlo Program

A.1 sweep.h

```
#ifndef SWEEP_H
#define SWEEP_H
#include <vector>
#include <tuple>
#include <stack>
#include <unordered_set>
#include <limits>
#include <array>
#include <stdexcept>
#include <math.h>
#include <iomanip>
#include <ostream>
#include <memory>
#include <string>

#include "constants.h"
#include "mpi.h"
#include "phi.h"
#include "lattice.h"


using namespace std;

enum ClusterAlgorithm { NONE, WOLFF};
typedef int site;

class BaseObservable {
    public:
        virtual double operator()(const Lattice2D& lat) const = 0;
        virtual const string name() const = 0;
        virtual ~BaseObservable() = default;
};

class Recorder {
    vector<double> measurements;
    vector<BaseObservable*> observables;
    public:
        Recorder(const vector<BaseObservable*> some_observables);
        void reserve(size_t size);
        int size();
        void record(const Lattice2D& lat, double x);
        void write(const string filename);
```

```
        ~Recorder();
};


struct sweep_args {

    int sweeps;
    int thermalization;
    int record_rate;
    ClusterAlgorithm cluster_algorithm;
    vector<double> ts;
    int cluster_rate;
    bool progress;
};

class Sweeper {
    const int process_Rank;
    const int size_Of_Cluster;
    const int sites_per_node;

    enum COLOR {
        black,
        white
    };

    vector<vector<site>*> mpi_assignments;

    private:
        static int get_rank(MPI_Comm c) {
            int rank;
            MPI_Comm_rank(c, &rank);
            return rank;
        }
        static int get_size(MPI_Comm c) {
            int size;
            MPI_Comm_size(c, &size);
            return size;
        }

        auto static constexpr gif_filename = "lattice.gif";
        const int gif_delay = 10;
        vector<site> full_neighbors(site aSite);
        Plaquette plaquette(site aSite);
        vector<Phi> dphis;
        Lattice2D flowed_lat;
        Lattice2D prev_flowed_lat;
        Lattice2D flowed_lat_2;
        Lattice2D k1, k2, k3, k4;

    public:
        const int DIM;
        Lattice2D lat;

        Sweeper();
        ~Sweeper();
        Sweeper(int DIM, MPI_Comm c);

        double full_action();
        void assert_action(double tol=0.001);
        int wrap(site c);
        double rand_dist(double r);
        Phi new_value(Phi old_phi);
        Phi proj_vec();
        Phi random_phi();
```

```
        void full_sweep(Recorder* recorder, const sweep_args& args);
        double sweep(COLOR color);
        void wolff();

        void broadcast_lattice();
        void collect_changes(double dS, COLOR color);

        double Padd(Phi dphi, Phi phi_b);
        unordered_set<int> generate_cluster(int seed, Phi r, bool accept_all);

        void runge_kutta(double t_, double h, Lattice2D& l, bool recycle_k1=false);
        void flow(vector<double> ts, Recorder* recorder=nullptr);

};

double randf();
int randint(int n);
Phi sign(Phi x);
double sign(double x);

#endif
```

A.2 sweep.cpp

```
#include <iostream>
#include <stdexcept>
#include "sweep.h"
#include <algorithm>
#include "gif.h"
#include "progress.cpp"
#include <assert.h>
#include <stdlib.h>
#include <math.h>
#include <fstream>

//#define VERIFY_ACTION
//#define GIF
//#define ACTIONFLOW
#define ADAPTIVESTEP

using namespace std;

Recorder::Recorder(const vector<BaseObservable*> some_observables) {
    observables = some_observables;
}

void Recorder::record(const Lattice2D& lat, double x=0) {
    measurements.push_back( x );
    for (const BaseObservable* f : observables) {
        measurements.push_back( (*f)(lat) );
    };
}

void Recorder::reserve(size_t size) {
    return measurements.reserve((observables.size()+1) * size);
}

int Recorder::size() {
    return measurements.size() / (observables.size() + 1);
}

void Recorder::write(const string filename) {
    ofstream outputfile;
```

```
    outputfile.open(filename);
    outputfile << "tau";
    for (const auto* obs : observables) {
        outputfile << "," << obs->name();
    }
    outputfile << endl;


    for (int i=0; i<measurements.size(); i++) {
        if ( (i+1)%(observables.size()+1) != 0 ) {
            outputfile << measurements[i] << ",";
        } else {
            outputfile << measurements[i] << endl;
        };
    }
    outputfile.close();
}

Recorder::~Recorder() {
    for (const BaseObservable* f : observables) {
        delete f;
    };

}


Sweeper::Sweeper (int DIM, MPI_Comm c) :
    process_Rank(get_rank(c)),
    size_Of_Cluster(get_size(c)),
    sites_per_node((DIM*DIM)/2 / get_size(c)),
    DIM{DIM}
{
    // generate lattice
    array<double, N> zero_array = {};
    Phi zero_phi(zero_array);

    dphis.resize(sites_per_node);

    mpi_assignments.push_back(new vector<site>);
    mpi_assignments.push_back(new vector<site>);

    site s;
    for (int i = 0; i<DIM; i++){
        for (int j = 0; j<DIM; j++) {
            s = i*DIM + j;
            lat[s] = new_value(zero_phi);

            lat.neighbor_map.push_back(full_neighbors(s));
            lat.plaquette_map.push_back(plaquette(s));

            // generate MPI assignments
            if (s / (2*sites_per_node) == process_Rank) { // Factor of 2 for
                ↪ checkerboard
                if ( (i+j)%2 ) {
                    mpi_assignments[COLOR::black]->push_back(s);
                } else {
                    mpi_assignments[COLOR::white]->push_back(s);
                }
            }
        }
    }

    lat.action = full_action();
    int offset = process_Rank *  sites_per_node;
}
```

```
Sweeper::~Sweeper() {
    delete mpi_assignments[COLOR::black];
    delete mpi_assignments[COLOR::white];

}

void print(std::vector <Phi> const &a) {
   for(int i=0; i < a.size(); i++)
   std::cout << a.at(i) << ' ';
   cout << endl;
}

double Sweeper::full_action() {
    site s;
    int i;
    Phi forward_nphi_sum;
    vector<site> neighbors;
    double S = 0;
    // initial action
    for (int s=0; s<DIM*DIM; s++) {
        neighbors = lat.neighbor_map[s];
        forward_nphi_sum  = lat[neighbors[2]] + lat[neighbors[3]];

        S += lat.lagrangian(lat[s], forward_nphi_sum);
    }
    return S;

}

int Sweeper::wrap( int c) {
    int mod = c % DIM;
    if (mod < 0) {
        mod += DIM;
    }
    return mod;

}

vector<site> Sweeper::full_neighbors(site aSite) {
    int x = aSite % DIM;
    int y = aSite / DIM;
    vector<site> neighbors;
    neighbors.push_back(wrap(x-1) + DIM * y);
    neighbors.push_back(x + DIM * wrap(y-1));
    neighbors.push_back(wrap(x+1) + DIM * y);
    neighbors.push_back(x + DIM * wrap(y+1));
    return neighbors;
}

Plaquette Sweeper::plaquette(site aSite) {
    int x = aSite % DIM;
    int y = aSite / DIM;
    return make_tuple(  x + DIM * y,
                        wrap(x+1) + DIM * y,
                        wrap(x+1) + DIM * wrap(y+1),
                        x + DIM * wrap(y+1)
                     );
}


double Sweeper::rand_dist(double r) {
    return 3 * r - 1.5;
}
```

```
Phi Sweeper::new_value(Phi old_phi) {
    return random_phi();
}

Phi Sweeper::proj_vec () {
    return Phi();
}

Phi Sweeper::random_phi() {
    Phi new_phi;
    for (int i=0; i<N; i++) {
        new_phi[i] = 2 * randf() - 1;
        if (abs(new_phi[i]) > 1e+10) {
            cout << "overflow" << endl;
            exit(1);

        }
    }
    return new_phi * (1/sqrt(new_phi.norm_sq()));

}

void write_gif_frame(Lattice2D& lat, GifWriter* gif_writer, int delay, double rate=3)
    ↪ {
    Phi aPhi;
    auto DIM = lat.L;
    vector<uint8_t> vec(4*DIM*DIM);

    for (int i=0; i<DIM*DIM; i++) {
        aPhi = lat[i];
        for (int j=0; j<N; j++) {
            vec[4*i + j] = static_cast<uint8_t>((aPhi[j]+1)*128);
        }
        vec[4*i + 3] = 255;
    }
    GifWriteFrame(gif_writer, vec.data(), DIM, DIM, delay);
}

void Sweeper::assert_action(double tol) {
    double fa = full_action();
    if (abs(fa-lat.action)>tol) {
        cout << "ASSERT␣ACTION␣FAILED␣(" << lat.action << "␣!=␣" << fa << ")\n";
        exit(1);
    } else {
        cout << "assert␣action␣passed␣(" << lat.action << "␣==␣" << fa << ")\n";

    }
}


void Sweeper::full_sweep(Recorder* recorder, const sweep_args& args = sweep_args()) {


    shared_ptr<progress_bar> progress;
    progress = args.progress ? make_shared<progress_bar>(cout, 70u, "Working") :
        ↪ nullptr;

    double dS;
    Phi phibar;
    double chi_m;

    int s;


    vector<uint8_t> white_vec(DIM*DIM*4,255);
```

```
    double norm_factor = 1 / (double) (DIM*DIM);
    #ifdef GIF
        GifWriter gif_writer;
        GifBegin(&gif_writer, gif_filename, DIM, DIM, gif_delay);
        write_gif_frame(lat, &gif_writer, gif_delay);
    #endif
    COLOR colors[2] = {COLOR::white, COLOR::black};
    for (int i=0; i<args.sweeps; i++) {
        for (const auto &color : colors) {
            if (progress != nullptr) {
                progress->write((double)i/args.sweeps);

            }
            broadcast_lattice();

            dS = sweep(color);
            collect_changes(dS, color);
        }

        if (i%args.record_rate==0 && i>=args.thermalization) {
            flow(args.ts, recorder);
            #ifdef GIF
                write_gif_frame(lat, &gif_writer, gif_delay);
            #endif
        }

        if (i%args.cluster_rate==0 && args.cluster_algorithm == WOLFF) {
            #ifdef GIF
                write_gif_frame(lat, &gif_writer, gif_delay);
            #endif
            wolff();
            #ifdef GIF
                write_gif_frame(lat, &gif_writer, gif_delay);
            #endif

        }

    }
    #ifdef GIF
        GifEnd(&gif_writer);
    #endif

}


double Sweeper::sweep(COLOR color){


    double tot_dS = 0;
    double dS, new_L, old_L, A, r;
    Phi newphi, dphi, phi, backward_nphi_sum, forward_nphi_sum;
    int i;
    site s;
    vector<site> neighbors;

    array<double, N> zero_array = {};
    Phi zero_phi(zero_array);

    for (int i = 0; i<sites_per_node; i++){
        s = mpi_assignments[color]->at(i);

        phi = lat[s];

        neighbors = lat.neighbor_map[s];
```

```
        backward_nphi_sum = lat[neighbors[0]] + lat[neighbors[1]];
        forward_nphi_sum  = lat[neighbors[2]] + lat[neighbors[3]];

        newphi = new_value(phi);

        old_L = lat.lagrangian( phi, forward_nphi_sum);
        new_L = lat.lagrangian( newphi, forward_nphi_sum);

        dphi = newphi - phi;
        dS = (new_L - old_L) - lat.beta * backward_nphi_sum * dphi;

        A = exp(-dS);
        r = randf();

        if (dS < 0 || r <= A) {
            dphis[i] = dphi;
            tot_dS += dS;
        } else {
            dphis[i] = zero_phi;
        }
    }
    return tot_dS;
}



double randf() {
    return (double)rand() / RAND_MAX;
}

int randint(int n) {
    return rand() % n;
}


double Sweeper::Padd(Phi dphi, Phi phi_b){

    double dS = -lat.beta * (dphi * phi_b);
    return 1 - exp(-dS);
}

unordered_set <int> Sweeper::generate_cluster(int seed, Phi r, bool accept_all) {
    int s, c, i;
    Phi phi_a, phi_b, dphi;

    double cumsum_dS = 0;

    double Padd_val;
    stack <tuple<int, double>> to_test; // (site, previous r_proj)
    unordered_set <int> cluster;

    phi_a = lat[seed];

    double r_proj_a = phi_a * r;
    double r_proj_b;
    double proj_sign = sign(r_proj_a);

    to_test.push(make_tuple(seed, 0. )); // site and r_projection. r_projection is
        ↪ overridden for seed
    bool first = true;
    while (to_test.size()>0) {
        tie(s, r_proj_a) = to_test.top();
        dphi = -2 * r_proj_a * r;
        to_test.pop();
```

```
        if (cluster.find(s)!=cluster.end()) {
            continue;
        }

        phi_b = lat[s];
        r_proj_b = phi_b * r;
        if (sign(r_proj_b) == proj_sign) {
            if (accept_all || first || randf() < Padd(dphi, phi_b)) {
                cluster.insert(s);
                for (const int n : lat.neighbor_map[s]){
                    to_test.push( make_tuple(n, r_proj_b) );
                }
            }

        }
        if (first) first = !first;
    }
    return cluster;
}

void Sweeper::wolff() {

    int seed = randint(pow(DIM,2));
    unordered_set <int>  cluster;
    int neighbors[4];
    int n, i, c;

    Phi r = random_phi();
    cluster = generate_cluster(seed, r, false);

    double dS = 0;
    Phi phi, dphi;


    for (const int c : cluster) {
        phi = lat[c];
        dphi = -2 * (phi * r) * r;
        lat[c] = phi + dphi;
        for (const int n : lat.neighbor_map[c]) {
            dS -= lat.beta * (lat[n] * dphi);
        }
    }

    lat.action+=dS;
#ifdef VERIFY_ACTION
    assert_action();
#endif

}

void Sweeper::broadcast_lattice() {
    if (size_Of_Cluster>1) {
        const int raw_data_len = DIM*DIM*N;
        double raw_data[raw_data_len];

        if (process_Rank == MASTER) {
            for (int i = 0; i < raw_data_len; i++) {
                raw_data[i] = lat[i/N][i%N];
            };
        }


        MPI_Bcast(&raw_data, raw_data_len, MPI_DOUBLE, MASTER, MPI_COMM_WORLD);
        MPI_Bcast(&lat.action, 1, MPI_INT, MASTER, MPI_COMM_WORLD);
```

```
        if (process_Rank != MASTER) {
            for (int i = 0; i < raw_data_len; i++) {
                lat[i/N] [i%N] = raw_data[i];
            };
        }
    }

}

void Sweeper::collect_changes(double dS, COLOR color){
    const int recv_data_size = N*DIM*DIM/2;
    double send_data[N*sites_per_node];

    for (int i=0; i<sites_per_node; i++) {
        for (int j = 0; j<N; j++) {
            send_data[i*N+j] = dphis[i][j] ;
        };
    };

    int recv_sites[DIM*DIM/2];
    double recv_data[recv_data_size];
    double recv_actions[size_Of_Cluster];

    MPI_Gather(mpi_assignments[color]->data(), sites_per_node, MPI_INT, &recv_sites,
        ↪ sites_per_node, MPI_INT, MASTER, MPI_COMM_WORLD);
    MPI_Gather(&send_data, N*sites_per_node, MPI_DOUBLE, &recv_data, N*sites_per_node
        ↪ , MPI_DOUBLE, MASTER, MPI_COMM_WORLD);
    MPI_Gather(&dS, 1, MPI_DOUBLE, &recv_actions, 1, MPI_DOUBLE, MASTER,
        ↪ MPI_COMM_WORLD);

    if (process_Rank == MASTER) {
        for (int i = 0; i<DIM*DIM/2; i++){
            for (int j = 0; j<N; j++) {
                lat[recv_sites[i]][j] += recv_data[i*N + j];
            }
        }

        for (int i = 0; i<size_Of_Cluster; i++) {
            lat.action += recv_actions[i];
        }
    }
}

Phi sign(Phi x){
    Phi phi_sign;
    phi_sign[0] = (x[0] > 0) - (x[0] < 0);
    return phi_sign;
}

double sign(double x){
    return (x>0) - (x<0);
}

inline void deriv(Lattice2D& f, double t, const Lattice2D& yn, double h, const
    ↪ Lattice2D* k = nullptr) {

    Phi neighbor_sum;
    Phi dte;
    Phi e;
    double Pij;
    double laplacianj;
    int L = yn.L;

    for (site s=0; s<L*L; s++) {
```

```
        neighbor_sum.init_as_zero();

        if (k) {
            e = yn[s] + k->at(s);
            for (site n : f.neighbor_map[s])
                neighbor_sum += yn[n] + k->at(n);
        } else {
            e = yn[s];
            for (site n : f.neighbor_map[s])
                neighbor_sum += yn[n];
        }

        for (int i=0; i<N; i++) {
            dte[i] = 0;
            for (int j=0; j<N; j++) {
                Pij = (i==j) - e[i] * e[j];
                laplacianj = neighbor_sum[j] - 2*D*e[j];
                dte[i] += Pij * laplacianj;
            }
        }

        f[s] = h*dte;
    }
}

void Sweeper::runge_kutta(double t_, double h, Lattice2D& l, bool recycle_k1) {

        // Runge Kutta (see http://www.foo.be/docs-free/Numerical_Recipe_In_C/c16-1.
            ↪ pdf)
        // Slight changes for efficency:
        //   - deriv(t, y, h, k) := h * f(t, y + k);
        //   - k1 => k1/2; k2 => k2/2

        if (t_>0) {
            if (recycle_k1) {
                k1 /= 2;
            } else {
                deriv(k1, t_,     l, h/2);
            }
            deriv(k2, t_+h/2, l, h/2, &k1);
            deriv(k3, t_+h/2, l, h,   &k2);
            deriv(k4, t_+h,   l, h,   &k3);

            k1 /= 3;
            k4 /= 6;

            l += k1;
            l += k2;
            l += k3;
            l += k4;

            // Normalize phi
            for (Phi& phi : l) {
                phi /= sqrt(phi.norm_sq());
            }
        }
}

void Sweeper::flow(vector<double> ts, Recorder* recorder) {
    // ts must be in ascending order
    double h = 0.01; // aka dt
    double t_ = 0;
    double chi_m;
    double S;
```

```
    double error;

    auto measurement_iter = ts.begin();
    double measurement_t = *measurement_iter;

    flowed_lat = lat;

    const auto gif_filename = "flow.gif";
    const int gif_delay = 10;
#ifdef GIF
        GifWriter gif_writer;
        GifBegin(&gif_writer, gif_filename, DIM, DIM, gif_delay);
#endif

#ifdef GIF
        int counter = 0;
#endif

    bool rerun=false;

    while (true) {

        if (t_ + h > measurement_t) {
            runge_kutta(t_, measurement_t-t_, flowed_lat);
            recorder->record(flowed_lat, measurement_t);
            t_ = measurement_t;

            measurement_iter++;
            if (measurement_iter == ts.end()) break;
            measurement_t = *(measurement_iter);

        } else {
#ifdef ADAPTIVESTEP
            if (!rerun){
                prev_flowed_lat = flowed_lat;
                flowed_lat_2 = flowed_lat;
                runge_kutta(t_, h, flowed_lat);
            } else {
                flowed_lat = flowed_lat_2;
            }

            runge_kutta(t_, h/2, flowed_lat_2, true);
            runge_kutta(t_+h/2, h/2, flowed_lat_2);

            error = 0;
            for (site i=0; i<flowed_lat.size(); i++) {
                error += (flowed_lat[i] - flowed_lat_2[i]).norm_sq();
            }

            error = sqrt(error)/15;
            if (error>MAXERROR) {
                rerun=true;
                h /= 2;
            } else if (error<MAXERROR/2) {
                rerun=false;
                h *=2;
                t_ += h;
            } else {
                rerun=false;
                t_ += h;
            }

#else
            runge_kutta(t_, h, flowed_lat);
            t_ += h;
```

```
#endif
        }
#ifdef ACTIONFLOW
        flowed_lat.action = flowed_lat.full_action();
#endif


#ifdef GIF
            if (counter % 10 == 0) write_gif_frame(flowed_lat, &gif_writer, gif_delay
                ↪ );
            counter++;
#endif
    }

    #ifdef GIF
        GifEnd(&gif_writer);
        system("gifsicle --colors 256 --resize 512x512 flow.gif -o flow.gif");
    #endif
}
```

A.3 lattice.h

```
#ifndef LATTICE_H
#define LATTICE_H


#include "phi.h"
#include <vector>
#include <tuple>

typedef tuple<int,int,int,int> Plaquette;

using namespace std;

class Lattice2D {
    typedef vector<Phi> datatype;
    datatype data;
    public:
        static int L;
        static double beta;
        static vector<vector<int>> neighbor_map;
        static vector<Plaquette> plaquette_map;
        double action;

        vector<Phi> vec() const;
        size_t size() const;

        Lattice2D();
        Lattice2D(const Lattice2D& other);

        Phi operator[] (int i) const;
        Phi& operator[] (int i);
        Phi at(int i) const;

        Lattice2D& operator+=(const Lattice2D& other);
        Lattice2D& operator*=(const double & factor);
        Lattice2D& operator/=(const double & factor);
        Lattice2D operator+ (const Lattice2D & other) const;

        // Iterators
        typedef datatype::iterator iterator;
        typedef datatype::const_iterator const_iterator;
```

```
        iterator begin();
        const_iterator cbegin() const;
        iterator end();
        const_iterator cend() const;

        static double lagrangian(const Phi phi, const Phi nphi_sum);
        double full_action();

};
#endif
```

A.4 lattice.cpp

```
#include <iostream>
#include "lattice.h"
#include <algorithm>

using namespace std;

int Lattice2D::L;
double Lattice2D::beta;

vector<vector<int>> Lattice2D::neighbor_map;
vector<Plaquette> Lattice2D::plaquette_map;

Lattice2D::Lattice2D() {
    data.resize(L*L);
}

Lattice2D::Lattice2D(const Lattice2D& other) {
    data = other.vec();
    action = other.action;
}

vector<Phi> Lattice2D::vec() const { return data; }
size_t Lattice2D::size() const { return data.size(); }

Phi Lattice2D::operator[] (int i) const { return data[i]; };
Phi& Lattice2D::operator[] (int i) { return data[i]; };
Phi Lattice2D::at(int i) const { return data.at(i); };

Lattice2D& Lattice2D::operator+=(const Lattice2D& other) {
    auto iter = other.cbegin();
    for_each(data.begin(), data.end(), [&iter](Phi &phi){phi += *(iter++); } );
    return *this;
};


Lattice2D& Lattice2D::operator*=(const double & factor) {
    for_each(data.begin(), data.end(), [factor](Phi &phi){phi *= factor; } );
    return *this;
};

Lattice2D& Lattice2D::operator/=(const double & factor) {
    for_each(data.begin(), data.end(), [factor](Phi &phi){phi /= factor; } );
    return *this;
};

Lattice2D Lattice2D::operator+ (const Lattice2D & other) const {
    Lattice2D new_lat(*this);
    new_lat += other;
    return new_lat;
};
```

```
Lattice2D::iterator Lattice2D::begin() { return data.begin(); }
Lattice2D::const_iterator Lattice2D::cbegin() const { return data.cbegin(); }
Lattice2D::iterator Lattice2D::end() { return data.end(); }
Lattice2D::const_iterator Lattice2D::cend() const { return data.cend(); }


double Lattice2D::full_action() {
    int site, i;
    Phi forward_nphi_sum;
    vector<int> neighbors;
    double S = 0;
    // initial action
    for (int s=0; s<L*L; s++) {
        neighbors = neighbor_map[s];
        forward_nphi_sum  = data[neighbors[2]] + data[neighbors[3]];
        S += lagrangian(data[s], forward_nphi_sum);
    }

    return S;

}


double Lattice2D::lagrangian(const Phi phi, const Phi nphi_sum) {
    return beta * (D - phi * nphi_sum); // note that the sum over dimension has
        ↪ already been made
}
```

A.5 phi.h

```
#ifndef PHI_H // include guard
#define PHI_H

#include "constants.h"
#include <array>
#include <ostream>

using namespace std;

class Phi {
    array<double, N> phi;

    public:
        Phi();
        Phi(array<double, N> phi);
        void init_as_zero();
        double norm_sq() const;
        Phi& operator+=(const Phi& other);
        Phi& operator*=(const double & a);
        Phi& operator&=(const double & a);
        Phi& operator/=(const double & a);

        Phi operator+ (const Phi & phi) const;
        Phi operator- (const Phi & phi) const;
        Phi operator- () const;
        double operator* (const Phi & phi) const; // Dot product
        Phi operator& (const Phi & phi) const; // Cross product
        Phi operator* (const double & a) const;
        friend ostream& operator<< (ostream& os, const Phi & aPhi);
        double operator[] (int i) const;
```

```
        double& operator[] (int i);
        bool operator== (const Phi & phi) const;
};

Phi operator*(double a, const Phi& b);


#endif
```

A.6 phi.cpp

```
#include "phi.h"

using namespace std;

Phi::Phi() {};

Phi::Phi(array<double, N> phi){
    this->phi = phi;
};

void Phi::init_as_zero(){
    phi = {0,0,0};
}

double Phi::norm_sq() const {
    double cumsum = 0;
    for (int i=0; i<N; i++) {
        cumsum += phi[i] * phi[i];
    }
    return cumsum;

}

Phi& Phi::operator+=(const Phi& other) {
    for (int i=0; i<N; i++) {
        phi[i] += other[i];
    }
    return *this;
}

Phi& Phi::operator*=(const double & a) {
    for (int i=0; i<N; i++) {
        phi[i] *= a;
    }
    return *this;
}

Phi& Phi::operator/=(const double & a) {
    for (int i=0; i<N; i++) {
        phi[i] /= a;
    }
    return *this;
}

Phi Phi::operator+ (const Phi & aPhi) const {
    Phi new_phi(phi);
    new_phi += aPhi;
    return new_phi;
}

Phi Phi::operator- (const Phi & aPhi) const {
    return *this + (-aPhi);
```

```
}
Phi Phi::operator- () const {
    Phi new_phi;
    for (int i=0; i<N; i++) {
        new_phi[i] = -phi[i];
    }
    return new_phi;
}

Phi Phi::operator& (const Phi & other) const {
    Phi new_phi;
    new_phi[0] = phi[1] * other[2] - phi[2] * other[1];
    new_phi[1] = phi[2] * other[0] - phi[0] * other[2];
    new_phi[2] = phi[0] * other[1] - phi[1] * other[0];
    return new_phi;


}

double Phi::operator* (const Phi & aPhi) const{
    //dot product
    double dot = 0;
    for (int i=0; i<N; i++) {
        dot += phi[i] * aPhi[i];
    }
    return dot;
}

Phi Phi::operator* (const double & a) const{
    Phi new_phi(phi);
    new_phi *= a;
    return new_phi;
}

ostream& operator<<(ostream& os, const Phi & aPhi)
{
    os << "(";
    for (int i=0; i<N; i++) {
        os << aPhi[i];
        if (i<N-1) { os<<",␣"; }
    }
    os << ")";
    return os;
}

double Phi::operator[] (int i) const {
    return phi[i];
}
double & Phi::operator[] (int i) {
    return phi[i];
}

bool Phi::operator== (const Phi & aPhi) const {
    for (int i=0; i<N; i++) {
        if (phi[i] != aPhi[i]) {
            return false;
        }
    }
    return true;
}

Phi operator*(double a, const Phi& b)
{
    return b*a;
}
```

A.7 observables.h

```
#include <math.h>
#include <string>


namespace observables {


    class action : public BaseObservable {
        public:
            double operator()(const Lattice2D& lat) const {
                return lat.action;
            };
            const string name() const { return "S"; };
    };

    class beta : public BaseObservable {
        public:
            double operator()(const Lattice2D& lat) const {
                return lat.beta;
            };
            const string name() const { return "beta"; };
    };

    class L : public BaseObservable {
        public:
            double operator()(const Lattice2D& lat) const {
                return lat.L;
            };
            const string name() const { return "L"; };
    };

    class chi_m : public BaseObservable {
        public:
            double operator()(const Lattice2D& lat) const {
                double val = 0;

                for (auto itx = lat.cbegin(); itx!=lat.cend(); ++itx) {
                    for (auto ity = lat.cbegin(); ity!=lat.cend(); ++ity) {
                        val += (*itx)*(*ity);
                    }
                }
                return val;
            };
            const string name() const { return "chi_m"; };
    };

    class F : public BaseObservable {
        public:
            double operator()(const Lattice2D& lat) const {
                double val = 0;

                int x=0;
                int y=0;
                const double two_pi_L = 2*M_PI*lat.L;

                //for (const Phi& phi : as_const(lat)) {
                for (auto itx = lat.cbegin(); itx!=lat.cend(); ++itx) {
                    for (auto ity = lat.cbegin(); ity!=lat.cend(); ++ity) {
                        val += (*itx)*(*ity) * cos( two_pi_L * (x - y));
                        x++;
                        if (x==lat.L) x=0; // This ensures that x is the Euclidean
                            ↪ space dimension
```

```
                    }
                    if (y==lat.L) y=0;
                }
                return val;
            };
            const string name() const { return "F"; };
    };

    class Q : public BaseObservable {
        private:
             double angle(double re, double im) const {
                double arctan = atan(im / re);
                if (re> 0) {
                    return arctan;
                } else if (im> 0) {
                    return M_PI + arctan;
                } else {
                    return (-M_PI + arctan);
                }
            }
            double sigma_A(Phi s1, Phi s2, Phi s3) const {
                // Returns values (-2pi,2pi)
                double real_part = 1 + s1 * s2 + s2 * s3 + s3 * s1;
                double imag_part = s1 * (s2 & s3);
                return 2*angle(real_part, imag_part);
            }

            double q(int x, const Lattice2D& lat, bool reversed=false) const {
                int x1, x2, x3, x4;
                tie(x1, x2, x3, x4) = lat.plaquette_map[x];
                Phi s1 = lat[x1], s2 = lat[x2], s3 = lat[x3], s4 = lat[x4];
                if (reversed) {
                    return sigma_A(s1,s2,s4) + sigma_A(s2,s3,s4);
                } else {
                    return sigma_A(s1,s2,s3) + sigma_A(s1,s3,s4);
                }
            }
        public:
            double operator()(const Lattice2D& lat) const{
                double Q = 0;
                for (int i=0; i<lat.L*lat.L; i++) {
                    Q += q(i, lat);
                }
                Q /= (4 * M_PI);
                return Q;
            };
            const string name() const { return "Q"; };
    };

};
```

A.8 constants.h

```
#define N 3
#define MASTER 0
#define PROG_CHAR "#"
#define D 2
#define MAXERROR 0.01
```

References

[1] B. Berg and M. Lüscher, "Definition and statistical distributions of a topological number in the lattice $O(3)$ σ-model", Nuclear Physics B **190**, 412–424 (1981).

[2] W. Bietenholz et al., "Topological Susceptibility of the 2d O(3) Model under Gradient Flow", Phys. Rev. D **98**, 114501 (2018).

[3] B. Odom et al., "New measurement of the electron magnetic moment using a one-electron quantum cyclotron", Phys. Rev. Lett. **97**, 030801 (2006).

[4] C. Callan et al., "Strings in background fields", Nuclear Physics B **262**, 593–609 (1985).

[5] A. Polyakov, "Interaction of goldstone particles in two dimensions. Applications to ferromagnets and massive Yang-Mills fields", Physics Letters B **59**, 79–81 (1975).

[6] P. Goddard and P. Mansfield, "Topological structures in field theories", Rep. Prog. Phys. **49**, 725–781 (1986).

[7] A. Y. Kitaev, "Quantum computations: algorithms and error correction", Russ. Math. Surv. **52**, 1191–1249 (1997).

[8] K. G. Wilson, "Confinement of quarks", Phys. Rev. D **10**, 2445–2459 (1974).

[9] L. Giusti, G. Rossi, and M. Testa, "Topological susceptibility in full QCD with Ginsparg–Wilson fermions", Physics Letters B **587**, 157–166 (2004).

[10] M. Bruno et al., "Topological susceptibility and the sampling of field space in $N_f = 2$ lattice QCD simulations", Journal of High Energy Physics **2014**, 150 (2014).

[11] C. Monahan, "The gradient flow in simple field theories", in Proceedings of The 33rd International Symposium on Lattice Field Theory — PoS(LATTICE 2015) (July 15, 2016), p. 052.

[12] B. Alles Salom and A. Papa, "Numerical Study of the mass spectrum in the 2D O(3) sigma model with a theta term", in Proceedings of The XXV International Symposium on Lattice Field Theory — PoS(LATTICE 2007) (Mar. 21, 2008), p. 287.

[13] A. Zee, *Quantum field theory in a nutshell*, Vol. 7 (Princeton University Press, 2010).

[14] R. P. Feynman, R. B. Leighton, and M. L. Sands, "The Principle of Least Action", in *The Feynman Lectures on Physics*, Vol. II, World Student Series (Addison-Wesley Pub. Co., Reading, Mass., 1963).

[15] S.-J. Chang, "Existence of a second-order phase transition in a two-dimensional φ^4 field theory", Phys. Rev. D **13**, 2778–2788 (1976).

[16] K. Binder, "Finite size scaling analysis of ising model block distribution functions", Zeitschrift für Physik B Condensed Matter **43**, 119–140 (1981).

[17] D. P. Landau and K. Binder, *A guide to monte carlo simulations in statistical physics* (Cambridge University Press, Cambridge, 2000).

[18] S. Solbrig et al., "Quantitative comparison of filtering methods in lattice QCD", PoS **LATTICE 2007**, 334 (2008).

[19] C. Monahan and K. Orginos, "Locally smeared operator product expansions in scalar field theory", Phys. Rev. D **91**, 074513 (2015).

[20] M. Lüscher, "Chiral symmetry and the Yang–Mills gradient flow", J. High Energ. Phys. **2013**, 123 (2013).

[21] H. Makino and H. Suzuki, "Renormalizability of the gradient flow in the 2D $O(N)$ non-linear sigma model", Progress of Theoretical and Experimental Physics **2015**, 33B08– (2015).

[22] D. A. Schaich and W. Loinaz, "Lattice Simulations of Nonperturbative Quantum Field Theories", Senior Thesis (unpublished) (Amherst College, May 12, 2006).

[23] R. H. Swendsen and J.-S. Wang, "Nonuniversal critical dynamics in Monte Carlo simulations", Phys. Rev. Lett. **58**, 86–88 (1987).

[24] U. Wolff, "Collective monte carlo updating for spin systems", Phys. Rev. Lett. **62**, 361–364 (1989).

[25] U. Wolff, "Monte Carlo errors with less errors", Computer Physics Communications **176**, 383 (2007).

[26] W. T. Vetterling et al., *Numerical recipes: Example book C* (Cambridge University Press, 1992).

[27] P. Virtanen et al., "SciPy 1.0: Fundamental algorithms for scientific computing in python", Nature Methods **17**, 261–272 (2020).

[28] M. Bögli et al., "Non-trivial θ-vacuum effects in the 2-d O(3) model", J. High Energ. Phys. **2012**, 117 (2012).