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 **iv**

List of Figures **v**

List of Tables **vi**

Abstract **v**

1 Introduction **1**

1.1 Method Overview 2

1.2 Conventions 3

2 Theory **4**

2.1 Quantum Field Theory 4

2.1.1 Path Integral Formulation 5

2.1.2 ϕ^4 model 7

2.1.3 Non-linear sigma model 7

2.1.4 NLSM θ term 8

2.2 Markov Chain Monte Carlo 8

2.2.1 The Markov Chain 9

2.3 Observables 10

2.3.1 Primary Observables 10

2.3.2 Secondary Observables 12
2.3.3 Bimodality 12
2.3.4 Jackknife Method 12
2.4 Topological Observables 12
2.5 Ultraviolet Divergences 14
2.5.1 Regularization: fields on the lattice 15
2.5.2 Renormalization 16
2.5.3 The Gradient Flow 17

3 Methods **19**
3.1 Fields on the Lattice 19
3.2 Monte Carlo Simulations 21
3.2.1 Metropolis Algorithm 22
3.2.2 Wolff Cluster Algorithm 22
3.2.3 Checkerboard algorithm 24
3.2.4 Autocorrelation times and thermalization 25
3.2.5 Runge-Kutta Algorithm 25

4 Results **27**
4.1 ϕ^4 model 27

5 Statistical Analysis **29**

6 Conclusion & Outlook **30**

A C++ NLSM Monte Carlo Program **31**
A.1 `sweep.h` 31
A.2 `sweep.cpp` 33

A.3 `lattice.h` 44
A.4 `lattice.cpp` 45
A.5 `phi.h` 46
A.6 `phi.cpp` 47
A.7 `observables.h` 49
A.8 `constants.h` 51

Acknowledgments

I would like to thank lots of people, and this is where I will do it...

List of Figures

2.1 Visualization of broken phase, symmetric phase and transition. Simulation run on 64×64 lattice, plotted after 1000 sweep thermalization. 11

2.2 Visualization of plaquette x^*. Dotted line separated plaquette into two signed areas which are used to define the topological charge density $q(x^*)$. Arrows represent order of signed area. 13

2.3 Visualization of signed area A on the sphere S^2 traced out by field at points x_1, x_2 and x_3. 15

2.4 Effect of flow time evolution on a random lattice in the symmetric phase. Red and blue indicate positive and negative values of the field in 2D spacetime. 18

3.1 An example of the Wolff cluster algorithm in the ϕ^4 model. White represents negative values of ϕ while black represents positive. $\lambda = 0.5$, $m_0^2 = -0.9$ 24

4.1 Simulation of the phase transition in the ϕ^4 model: Magnetization $\langle|\bar{\phi}|\rangle$, magnetic susceptibility χ_m, Binder cumulant U and bimodality B for various values of the mass squared m_0^2. 28

List of Tables

2.1 Average magnetization with average action per site corresponding to the particular configurations in Fig. 2.1. 11

Abstract

Abstract

Chapter 1

Introduction

Quantum field theory (QFT) is an extraordinarily successful framework which can be applied to a range of physical phenomena. 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. In condensed matter physics, effective field theories model emergent effects such as phonons and quasiparticles. Compared to experiment, QFT is remarkably accurate, famously matching the experimental value for the electron g-factor to eight significant figures.[1]

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 so-called *renormalization group*. However, the technique fails with perturbatively nonrenormalizable theories.

One such example is the *non-linear sigma model*, a prototypical theory in both condensed matter and particle physics. In solid-state systems, this model describes Heisenberg ferromagnets and in nuclear physics, it acts as a prototype for quantum chromodynamics (QCD), exhibiting characteristic features such as a mass gap and

asymptotic freedom. [1]

In this study, we specifically consider the $O(3)$ non-linear sigma model 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.

Since the non-linear sigma model cannot be renormalized perturbatively, we cannot study these topological effects with normal perturbative techniques. An alternative solution is placing the field on a discretized lattice, a technique originally used for quantum chromodynamics. In this scenario, field configurations become computationally calculable. This process introduces an nonphysical length scale a, the lattice spacing. Therefore, we expect any physical result to converge in the continuum limit, i.e. when $a \to 0$. However, this is not always the case as observables mix on the lattice, leading to divergences. As an example, states of definite angular momentum mix when discretized, a clear violation of the angular momentum commutation relations.

The gradient flow is a technique designed to remove these divergences. By dampening high-momentum fluctuations, the gradient flow reduces power-divergent mixing and make observables finite on the lattice.[2] In quantum chromodynamics, previous studies have verified the ability of the gradient flow to make observables finite[2]. Due to its success in QCD, there has been interest in using the gradient flow to finitize the topological susceptibility in the 1+1 $O(3)$ non-linear sigma model.[3].

1.1 Method Overview

To numerically study the topological qualities of the non-linear sigma model, we first implement a Markov Chain Monte Carlo simulation. We initially construct a proof-

[1] *citation needed*
[2] *citation needed*

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 non-linear sigma model in larger lattices. Since the gradient flow has no exact solution in the non-linear sigma model, we implement a numerical solution using a fourth-order Runge-Kutta approximation. By applying the gradient flow to every configuration in the sample, we can measure its effect on the topological charge and susceptibility.

1.2 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 spacetime four-vector 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 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.

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 is physical.

Though most introductions to QFT use the "second quantization," we will use a alternative formulation known as the path integral formulation (for a similar introduction see Zee's textbook [4]).

2.1.1 Path Integral Formulation

As stated above, we can model a quantum field theory as a superposition of all possible classical fields. Like single-particle quantum mechanics, each configuration has a 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 field'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.[5]

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 normal 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 equivalently Lorentz invariant, which is representative of a four-dimensional Euclidean space. Furthermore, we find that

$$d^4x_E = d^3\vec{x}dx_4 \tag{2.4}$$

$$= id^3\vec{x}dx_0 \tag{2.5}$$

$$= id^4x. \tag{2.6}$$

We can use this transformation to redefine the Lagrangian $\mathcal{L}$ in Euclidean space as $\mathcal{L}_E$, replacing all x_0 with ix_0. Since a Lorentz-invariant Lagrangian must include only even powers and derivatives of x, the Euclidean Lagrangian remains real. Subsequently, we can define a Euclidean action based on the differential in Eq. 2.6:

$$S_E = \int d^4x_E \mathcal{L}_E \tag{2.7}$$

$$= i \int d^4x \mathcal{L}_E, \tag{2.8}$$

allowing us to redefine the path integral as

$$\langle \hat{O} \rangle = \frac{1}{Z} \int \mathcal{D}\phi \, \hat{O}(\phi) \, e^{-S_E[\phi]}. \tag{2.9}$$

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

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

$$S[\phi] = \int d^D x \left[\frac{1}{2} \partial^\mu \phi \partial_\mu \phi - \frac{1}{2} m_0^2 \phi^2 - \frac{\lambda}{4} \phi^4 \right]. \tag{2.10}$$

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

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

Following Sec. 2.1.1, we calculate the Euclidean action in 1+1 dimensions as

$$S_E[\phi] = \int d^2 x_E \left[\frac{1}{2} \left(\partial_t \phi \right) + \frac{1}{2} \left(\partial_x \phi \right) + \frac{1}{2} m_0^2 \phi^2 + \frac{\lambda}{4} \phi^4 \right]. \tag{2.11}$$

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

This field features spontaneous symmetry breaking at a critical value of m_0^2. In practice this property causes the field to spontaneously align, similarly to spins aligning in a 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 many physical phenomena, including applications in string theory. As a simple nonperturbative model,

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

it 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. Furthermore, it has an exact application to Heisenberg ferromagnets in condensed matter.

Unlike the phi^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 non-linear sigma model features a global symmetry under the 3D orthogonal group $O(3)$. In other words, the theory is invariant under rotating all vectors the same way. To differentiate it from the ϕ^4 model, we denote the NLSM field as $\vec{e}(x)$.

The theory is defined by the Euclidean action

$$S_E = \frac{\beta}{2} \int d^D x \partial^\mu \vec{e} \cdot \partial_\mu \vec{e} \tag{2.12}$$

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

2.1.4 NLSM θ term

In order to make the non-linear sigma model topologically nontrivial, we introduce a θ term into the action such that

$$S[\vec{e}] \to S[\vec{e}] - i\theta Q[\phi], \tag{2.13}$$

where Q is the topological charge (see Sec. 2.4).

2.2 Markov Chain Monte Carlo

To accomplish a statistical analysis of quantum fields, we use a Monte Carlo simulation. This method amounts to producing a large number of configurations and calculating statistics on the sample. A brute-force calculation over all possible configurations— as Eq. 2.9 suggests— is clearly infinite and computationally infeasible. However, the

exponential nature of the Boltzmann factor dictates that only configurations near the action minimum 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

The Markov chain is an effective method to identity these low-action states. Essentially, we begin with a random configuration and then make small adjustments, gradually lowering the action. Starting with a field configuration ϕ_a, we propose a new field ϕ_b. The probability of accepting this change, thereby adding the configuration to the chain, is given by the function

$$P(\phi_a \to \phi_b).$$

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 *detailed balance equations*:

$$P(\phi_a)\, P(\phi_a \to \phi_b) = P(\phi_b)\, P(\phi_b \to \phi_a), \tag{2.14}$$

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

$$P(\phi) = \frac{1}{Z} e^{-S_E[\phi]}. \tag{2.15}$$

Therefore, by rearranging Eq. 2.14, we find

$$\frac{P(\phi_a \to \phi_b)}{P(\phi_b \to \phi_a)} = e^{S_E[\phi_a] - S_E[\phi_b]}. \tag{2.16}$$

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

2.3 Observables

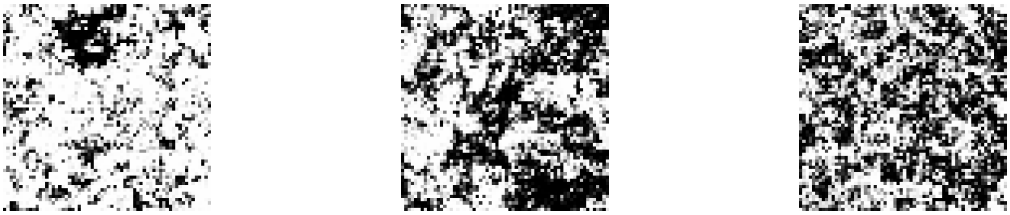
To extract physics from Monte Carlo simulations, we define a set of "observables". These quantities manifest as expectation values of operators, calculated using the Euclidean path integral formula (Eq. 2.9). We can classify these observables into two categories: primary and secondary observables. Primary 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.[3] 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.

[3]One slight exception is the magnetic susceptibility, which can be defined both ways.

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 phase ($m_0^2 = -1.0$) (b) transition ($m_0^2 = -0.7$) (c) symmetric phase ($m_0^2 = -0.4$)

Figure 2.1: Visualization of broken phase, symmetric phase and transition. Simulation run on 64×64 lattice, plotted after 1000 sweep thermalization.

	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.

Average Magnetization

The average magnetization quantifies the total alignment of the field. In the ϕ^4 theory, this value is a scalar and indicates the spontaneous symmetry breaking of the field. In the NLSM, this value is a vector and also represents a spontaneous symmetry breaking, though this quality is outside the scope of this study. In both theories, a value of zero indicates the symmetric phase while a nonzero value indicates broken symmetry. In the NLSM, a magnitude of one represents total alignment.

Mathematically, we can define this quantity in the ϕ^4 model as

$$|\bar{\phi}| \equiv \frac{1}{L^2} \sum_{i}^{L^2} \phi(x_i). \tag{2.17}$$

In the NLSM, the equivalent value is defined as

$$|\bar{\vec{e}}| \equiv \frac{1}{L^2} \sum_{i}^{L^2} \vec{e}(x_i). \tag{2.18}$$

Magnetic Susceptibility

2.3.2 Secondary Observables

Binder Cumulant

2.3.3 Bimodality

2.3.4 Jackknife Method

2.4 Topological Observables

The $O(3)$ non-linear sigma model model features topological properties 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 in three dimensions.

2. The elements of the $O(3)$ non-linear sigma model are three-dimensional unit vectors, thereby existing on a three dimensional unit sphere.

With these two properties, we can view the field as a continuous mapping between two 3D spheres, 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.

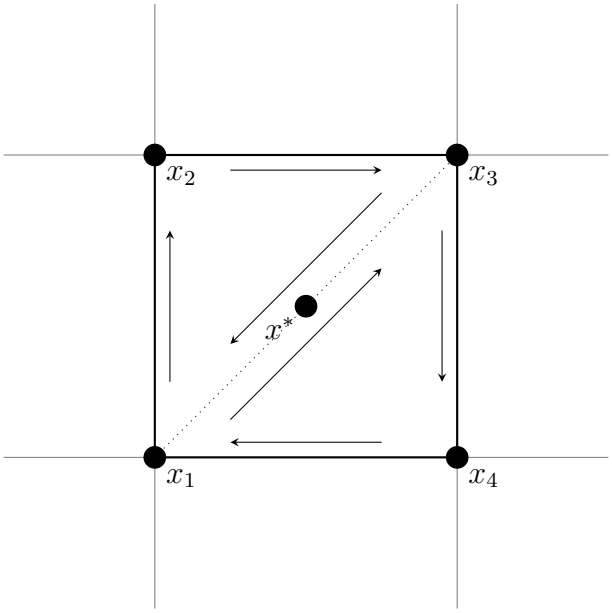


Figure 2.2: Visualization of plaquette x^*. Dotted line separated plaquette into two signed areas which are used to define the topological charge density $q(x^*)$. Arrows represent order of signed area.

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 non-linear sigma model. We associate every field configuration with an element of this group, known as the *topological charge*, which we denote as Q.

In practice, this quantity is nontrivial to calculate. To begin, we define a local topological charge 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:

$$Q \equiv \sum_{x^*} q(x^*). \tag{2.19}$$

As a function of x^*, the charge density is a function of the field on the plaquette vertices, an idea visualized in Fig. 2.2. 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. 2.3 (the sign changes with odd permutations of the ordering). We follow the derivation in [6] and split the plaquette into two triangles, as shown in Fig. 2.2, with the ordering determining the sign. The topological charge density is defined using this signed area as

$$q(x^*) = \frac{1}{4\pi}\Big[A\Big(\vec{e}(x_1), \vec{e}(x_3), \vec{e}(x_3)\Big) + A\Big(\vec{e}(x_1), \vec{e}(x_3), \vec{e}(x_4)\Big)\Big]. \tag{2.20}$$

This sign 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

$$-2\pi < A < 2\pi. \tag{2.21}$$

Following [6], this yields an expression for the signed area

$$A(\vec{e}_1, \vec{e}_2, \vec{e}_1) = 2\arg\Big(1 + \vec{e}_1 \cdot \vec{e}_2 + \vec{e}_2 \cdot \vec{e}_3 + \vec{e}_3 \cdot \vec{e}_1 + i\vec{e}_1 \cdot (\vec{e}_2 \times \vec{e}_3)\Big). \tag{2.22}$$

Under periodic boundary conditions, these triangles on the sphere necessarily wrap S^2 an integer number of times, ensuring Q is an integer. Following this quantity, we can define a topological susceptibility χ_t

$$\chi_t \equiv \frac{1}{L^2}\Big(\langle Q^2\rangle - \langle Q\rangle^2\Big). \tag{2.23}$$

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 type of

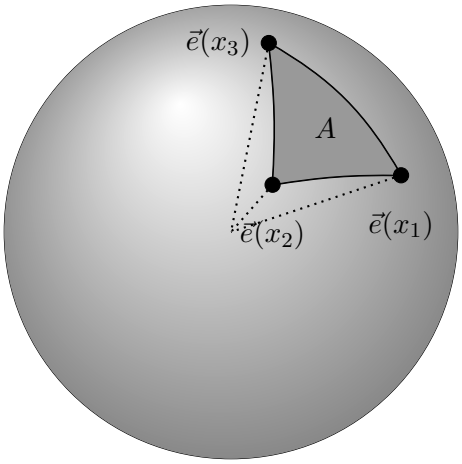


Figure 2.3: Visualization of signed area A on the sphere S^2 traced out by field at points x_1, x_2 and x_3.

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). 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 is involved and consists of two steps: regularization and renormalization.

2.5.1 Regularization: fields on the lattice

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 into the real numbers.

In this study, we employ lattice regularization. This process discretizes the field, modeling the field $\phi(x)$ as a lattice ϕ_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.

2.5.2 Renormalization

After regularization, we redefine the Lagrangian parameters in terms of the regularization parameter, following a handful of boundary conditions. In this study, we assert that L/ξ remains constant, where L is the side length of the system and ξ is the coherence length. In perturbation theory, this process is arduous and includes the introduction of counter-terms into the Lagrangian. In the case of the non-linear sigma model, it is impossible using counterterms[4]but can be performed numerically. In this study, we use predetermined values from [3].

At this point, we can calculate observables as functions of regularization parameters. To achieve physical values, 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. The topological susceptibility χ_t is one such value that diverges in the continuum limit. As we decrease the width of each lattice site, high frequency modes become more significant, leading to an ultraviolet divergence in the operator.

[4] *citation needed*

2.5.3 The Gradient Flow

To remove this ultraviolet divergence, we adopt a technique is "smearing", a local averaging of the field.[7] Specifically, we use a technique known as the "gradient flow" [8] which introduces a new half-dimension called "flow time", or τ.[5] The flow time parameterizes the smearing such that an evolution in flow time corresponds to suppressing ultraviolet divergences.

Specifically, the gradient flow pushes field configuration toward classical minima of the action. Additionally, renormalized correlation functions remain renormalized at nonzero flow time.[9] In 2D ϕ^4 scalar field theory, the gradient flow is defined by the differential equation

$$\frac{\partial \rho(\tau, x)}{\partial \tau} = \partial^2 \rho(\tau, x) \tag{2.24}$$

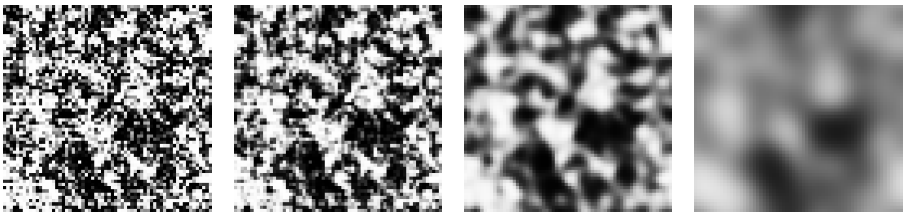
where ∂^2 is the Laplacian in 4-D Euclidean spacetime and τ is the flow time. Here, ρ is the field evolved into a nonzero flow time, bounded by the condition $\rho(\tau = 0, x) = \phi(x)$. In the ϕ^4 theory, we can solve this equation exactly to find [2]

$$\rho(\tau, x) = \frac{1}{4\pi\tau} \int d^2y e^{-(x-y)^2/4\tau} \phi(y). \tag{2.25}$$

This function forms a Gaussian, smoothly dampening high-momentum modes and removing ultraviolet divergences from evolved correlation functions.[10] We can visualize this by plotting the ϕ field, shown in Fig. 2.4. These plots demonstrate the reduction of high momentum modes.

Generally, we can choose any flow time equation that drives the field towards a classical minimum. Beyond the ϕ^4 model, we need flow equations that incorporate different types of fields. In the non-linear sigma model, it has been shown that an appropriate manifestation of the flow time can resolve divergences [10]. Therefore, it is possible to define a different flow equation for this model as well. Following [3], we

[5] The term "half-dimension" indicates that $\tau > 0$.



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

Figure 2.4: Effect of flow time evolution on a random lattice in the symmetric phase. Red and blue indicate positive and negative values of the field in 2D spacetime.

can define the gradient flow in this model via the differential equation

$$\partial_\tau \vec{\rho}(\tau, x) = \left(1 - \vec{\rho}(\tau, x)\vec{\rho}(\tau, x)^T\right) \partial^2 \vec{\rho}(\tau, x), \tag{2.26}$$

where ∂^2 is the Laplacian operator in Euclidean space[6]. We solve this equation numerically using the boundary condition $\vec{\rho}(0, x) = \vec{e}(x)$.

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

Chapter 3

Methods

Our study of the gradient flow in the non-linear sigma model consists of a computational part and an analytical part. We begin by outlining a numerical Monte Carlo method to simulate the lattice in two and three dimensions. We verify our program with the well-studied ϕ^4 scalar field theory. We then generalize our model to a vector field to simulate the non-linear sigma model. This simulation system provides data on vacuum states with which we study the gradient flow's effect on topology.

We implement these two algorithms first in Python for the ϕ^4 model. Afterwards, we transition to C/C++ code due to the increased speed for more complicated theories.

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

$$x_{i,j} = ia\hat{\mu}_t + ja\hat{\mu}_x. \tag{3.1}$$

Here, a is the lattice constant and $\hat{\mu}_0$ and $\hat{\mu}_1$ 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 constant 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. 2.11. We begin by redefining the derivative operator as a 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

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

$$\begin{aligned}\rightarrow \frac{1}{2a^2}\big[&\phi^2(x + a\hat{t}) + \phi^2(x + a\hat{x}) + 2\phi^2(x) \\ &-2\phi(x + a\hat{t})\phi(x) - 2\phi(x + a\hat{x})\phi(x)\big]\end{aligned} \tag{3.5}$$

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

$$\frac{1}{2}\left(\partial_t \phi\right)^2 + \frac{1}{2}\left(\partial_x \phi\right)^2 \rightarrow \frac{1}{a^2}\left[2\phi^2(x) - \phi(x + a\hat{t})\phi(x) - \phi(x + a\hat{x})\phi(x)\right] \tag{3.6}$$

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 sum. This takes the form

$$\int dt dx \rightarrow a^2 \sum_i \tag{3.7}$$

such that the final discretized action becomes

$$S_{\text{lat}}[\phi] = \sum_i \left[-\phi(x_i + a\hat{t})\phi(x_i) - \phi(x_i + a\hat{x})\phi(x_i) + \left(2 + \frac{1}{2}m_0^2\right)\phi^2(x_i) + \frac{1}{4}\lambda\phi^4(x_i)\right] \tag{3.8}$$

Likewise, we can discretize the non-linear sigma model. In this case, the derivative term becomes

$$\frac{1}{2}\left(\partial_t \vec{e}\right)^2 + \frac{1}{2}\left(\partial_x \vec{e}\right)^2 \to \frac{1}{a^2}\left[2 - \vec{e}(x + a\hat{t}) \cdot \vec{e}(x) - \vec{e}(x + a\hat{x}) \cdot \vec{e}(x)\right]. \tag{3.9}$$

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

$$S_{\text{lat}}[\vec{e}] = \sum_i \left[2 - \vec{e}(x + a\hat{t}) \cdot \vec{e}(x) - \vec{e}(x + a\hat{x}) \cdot \vec{e}(x)\right]. \tag{3.10}$$

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 Fast Fourier Transform. *TODO: explicit expression for this* In the NLSM, the definition of the gradient flow (Eq. 2.26) becomes

$$\partial_\tau \vec{e}(\tau, x) = \left(1 - \vec{e}(\tau, x)\vec{e}(\tau, x)^T\right) \partial^2 \vec{e}(\tau, x), \tag{3.11}$$

where the Laplacian operator ∂^2 is defined as

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

3.2 Monte Carlo Simulations

We implement a Markov Chain Monte Carlo method following Schaich's thesis [11]. 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 $P(\mu \to \nu)$ where μ and ν are the existing and proposed lattice configurations respectively. We use a combination of the Metropolis and Wolff algorithms to determine this value.

3.2.1 Metropolis Algorithm

We primarily use the Metropolis algorithm for the calculation of new Markov chain configurations. We begin with a so-called "hot start," where each field value at each lattice site is randomly selected. Then we propose a new value for a single lattice point, which is accepted with a probability

$$P(\phi_a \to \phi_b) = \begin{cases} e^{-(S[\phi_b]-S_[\phi_a])} & S[\phi_b] < S_[\phi_a] \\ 1 & \text{otherwise} \end{cases} \tag{3.12}$$

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 many times pushes the lattice toward the action minimum.

3.2.2 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 non-linear sigma model feature "kinetic" terms with gradients of ϕ. Therefore, the presence of large similarly-valued regions in the lattice can lead to a local minimum. The Wolff algorithm helps reduce the presence of these clusters through two steps: identifying a cluster and flipping it along some arbitrary vector. In the case of ϕ^4 theory, this flipping takes the form of a simple sign change. In the non-linear sigma model 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.[1]To identify the cluster, the algorithm uses a recursive algorithm defined by the probability of adding a new site.

We define a cluster C as a continuous set of aligned sites which is generated probabilistically. In ϕ^4 theory, we define "aligned" as having the same sign. In the

[1] *citation needed*

non-linear sigma model, we choose a random unit vector $\vec{r}$ and use it to project the field vector. The sign of this quantity defines "aligned" in the $O(3)$ NLSM. In order to find the probability p_i of adding a site x_i to the cluster C, we use the detailed balance equation (Eq. 2.16) which relates the Markov chain probability $P(\phi_a \to \phi_b)$ to the actions of each configuration. In the context of the Wolff cluster algorithm, we will notate this probability as $P(\phi \to f_C(\phi))$ and express it as

$$P(\phi \to f_C(\phi)) = \left(\prod_{i \in C} p_i\right)\left(\prod_{j \in N}(1-p_j)\right) \tag{3.13}$$

where N is the set of aligned neighboring sites. Since $p_i = 0$ for sites that are not aligned with the cluster, they are not included in this expression. Now if we consider the probability of flipping the cluster back, i.e. $P(f_C(\phi) \to \phi)$, we achieve a similar expression:

$$P(f_C(\phi) \to \phi) = \left(\prod_{i \in C} p_i\right)\left(\prod_{j \in M}(1-p_j)\right) \tag{3.14}$$

where M is the set of initially unaligned sites (they become aligned after the first flip).

Using the detailed balance equations, we can now relate these quantities with the actions $S[\phi]$ and $S[f_C(\phi)]$:

$$\frac{\prod_{j \in N}(1-p_j)}{\prod_{j \in M}(1-p_j)} = e^{S[\phi]-S[f_C(\phi)]}. \tag{3.15}$$

To simplify the exponent, we follow the discretized ϕ^4 action in Eq. 3.8. We see that the mass and interaction terms are invariant under a sign change of ϕ and therefore disappear in the difference. In fact, the only change in the action results from the neighboring sites $N \cup M$. The first two terms of Eq. 3.8 can be viewed as a sum over all pairs of neighboring lattice points x_a and x_b, such that a change

$$\phi(x_a) \to -\phi(x_a) \tag{3.16}$$

$$\phi(x_b) \to \phi(x_b) \tag{3.17}$$

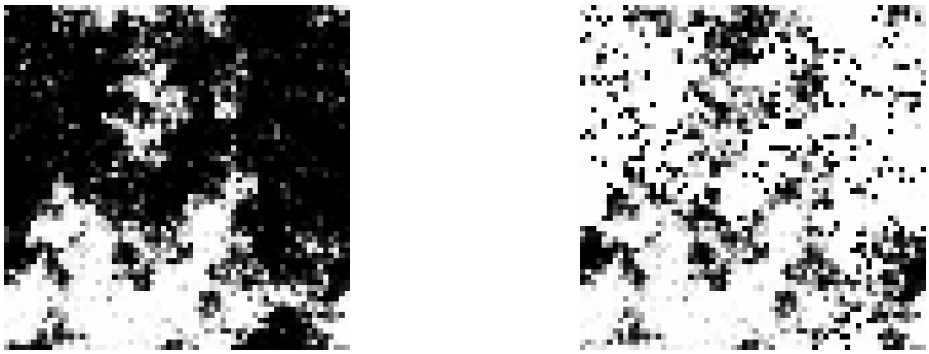
implies

$$-\phi(x_b)\phi(x_a) \to \phi(x_b)\phi(x_a),$$

leading to a term -2$\phi(x_a)\phi(x_b)$ in the exponent. Likewise,

$$p_i = 1 - e^{-\beta \vec{e}_a \cdot \vec{e}_b}. \tag{3.18}$$

Fig. 3.1 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 island 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 3.1: An example of the Wolff cluster algorithm in the ϕ^4 model. White represents negative values of ϕ while black represents positive. $\lambda = 0.5$, $m_0^2 = -0.9$

3.2.3 Checkerboard algorithm

In order to parallelize the Metropolis algorithm, we use a checkerboard algorithm. Since the Lagrangian density at each site does not depend on any sites diagonal to it, the lattice can be split into "white" sites and "black" sites, like the tiles on a

checkerboard. Each white site is independent of every other white site and likewise with 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 the code through the Message Passing Interface (MPI).

3.2.4 Autocorrelation times and thermalization

One important aspect is the correlation between different states of the Markov chain. Ideally, each sample of the lattice would be completely independent, but this is not the case. Therefore, we must sweep the lattice sufficiently to produce an effective Monte Carlo result. We use Wolff's automatic windowing procedure [12] and the magnetic susceptibility χ_m to estimate the autocorrelation and determine an appropriate number of sweep between measurements for each theory. Furthermore, the initial values of our Markov chain will be highly correlated with the initial random start. Therefore, we also wait a set number of sweeps before taking any measurements so that the lattice can approach a minimum of the action.

3.2.5 Runge-Kutta Algorithm

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

$$\vec{e}(\tau + h)_x \approx \vec{e}(\tau)_x + hf(\vec{e}(\tau)_x).$$

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

$$f(\vec{e}) = \partial_\tau \vec{e}(\tau)_x = \left(1 - \vec{e}(\tau)_x \vec{e}(\tau)_x^T\right) \partial^2 \vec{e}(\tau)_x, \tag{3.19}$$

following from Eq. 3.11. To the fourth order, this approximation becomes

$$k_1 = hf\left(\vec{e}(\tau)_x\right) \tag{3.20}$$

$$k_2 = hf\left(\vec{e}(\tau)_x + \frac{k_1}{2}\right) \tag{3.21}$$

$$k_3 = hf\left(\vec{e}(\tau)_x + \frac{k_2}{2}\right) \tag{3.22}$$

$$k_4 = hf\left(\vec{e}(\tau)_x + k_3\right) \tag{3.23}$$

$$\vec{e}(\tau + h)_x = \vec{e}(\tau)_x + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5). \tag{3.24}$$

This method is usually superior to Euler's method and the midpoint method [13].

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.

Chapter 4

Results

4.1 ϕ^4 model

We initially implemented the ϕ^4 model using the Monte Carlo method described in Sec. 3 to verify the results of our system. According to previous studies ([2], [11]), the ϕ^4 model exhibits a symmetric and broken phase depending on its parameters m_0^2 and λ, specifically occurring at $m_0^2 = -0.72$ for $\lambda = 0.5$. We verify this result by plotting four observables: the lattice average $|\langle\bar{\phi}\rangle|$, the lattice variance χ, the Binder cumulant U and the bimodality B in Fig. 4.1.

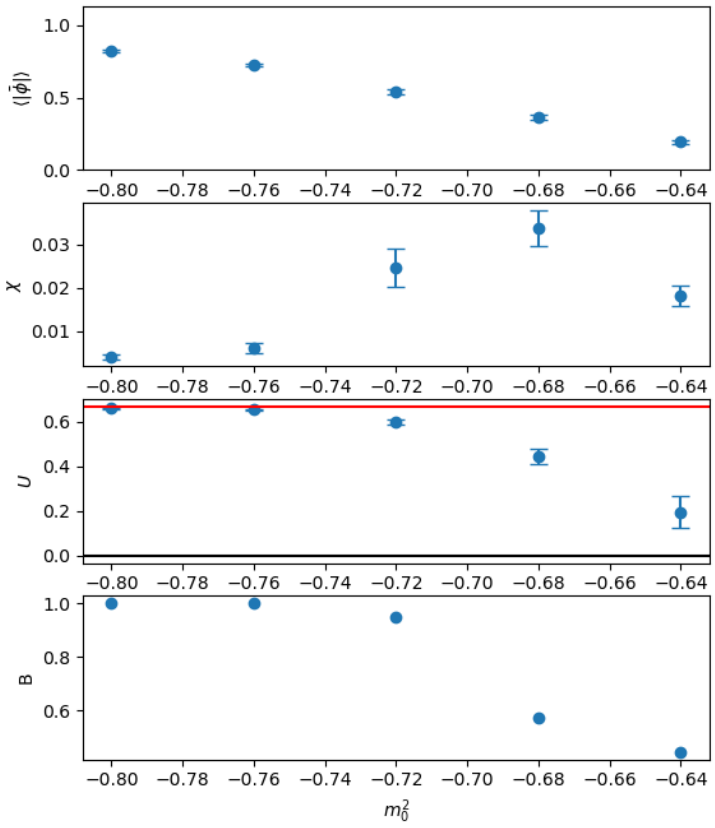


Figure 4.1: Simulation of the phase transition in the ϕ^4 model: Magnetization $\langle|\bar{\phi}|\rangle$, magnetic susceptibility χ_m, Binder cumulant U and bimodality B for various values of the mass squared m_0^2.

Chapter 5

Statistical Analysis

Chapter 6

Conclusion & Outlook

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) {*/


    //Phi dphi;
    //double sum_of_squares = 0;
    //for (int i = 0; i<N-1; i++) {
        //dphi[i] = rand_dist(randf());

        //sum_of_squares += pow(old_phi[i] + dphi[i], 2);
    //}
    //dphi[N-1] = (2*randint(2) - 1) * sqrt(1-sum_of_squares) - old_phi[N-1]; //the
        ↪ first term randomizes the sign

    //// test?
    //cout << "Dot: " << (old_phi + dphi) * (old_phi + dphi) << endl;
    //return dphi;
/*}*/

Phi Sweeper::new_value(Phi old_phi) {
    //Phi dphi;
    //dphi[0] = rand_dist(randf());
    //return dphi;
    return random_phi();
}

Phi Sweeper::proj_vec () {
    //for (int i = 0; i<N-1; i++) {
        //r[i] =
    //}
    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;
        //cout << vec[0] <<" "<< vec[1] <<" "<< vec[2] <<" "<< vec[3] <<" "<< endl;
    }
```

```
    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()) {

    //if (args.cluster_algorithm != WOLFF) throw invalid_argument("Currently, Wolff
        ↪ is the only allowed algorithm");

    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();

            //cout << "Met: " << action << " vs " << full_action() << endl;
            dS = sweep(color);
            collect_changes(dS, color);
        }
        //cout << "Met: " << action << " vs " << full_action() << endl;

        //if (i==args.thermalization && gif)
            //GifWriteFrame(&gif_writer, white_vec.data(), DIM, DIM, gif_delay); //
                ↪ add one white frame after thermalization

        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 (gif) write_gif_frame(lat, &gif_writer, gif_delay);
        }
```

```
        if (i%args.cluster_rate==0 && args.cluster_algorithm == WOLFF) {
            //cout << "Wolff: " << action << " vs " << full_action() << endl;
            #ifdef GIF
                write_gif_frame(lat, &gif_writer, gif_delay);
            #endif
            wolff();
            #ifdef GIF
                write_gif_frame(lat, &gif_writer, gif_delay);
            #endif
            //if (gif) write_gif_frame(this, &gif_writer, gif_delay);
            //cout << "Wolff: " << action << " vs " << full_action() << endl;

        }

    }
    #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;

        //cout << old_L << " " << new_L << " " << backward_nphi_sum << endl;
        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; // may want to replace this with something better later
}


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

    double dS = -lat.beta * (dphi * phi_b);
    return 1 - exp(-dS); // Schaich Eq. 7.17, promoted for vectors
}

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, phi_a * numeric_limits<double>::max() )); // site
        ↪  and phi value, using infinity to ensure Padd=1 for first addition
    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 (s == seed) {
                    //cout << "SEED" << endl;
                    //cumsum_dS += 2 * beta * (lat[seed] * phi_b);
                //} else {
                    //cumsum_dS += 2 * beta * (phi_a * phi_b);
                //}
            }
```

```
        }
        if (first) first = !first;
    }
    //cout << "cumsum_dS: " << cumsum_dS << endl;
    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;
    //cout << "dS: " << dS << endl;
#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){ // temporary, should be eventually replaced with proj_vec
    Phi phi_sign;
    phi_sign[0] = (x[0] > 0) - (x[0] < 0);
    return phi_sign;
}

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


// GF
//
//

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];
        }

        //if (s==0) cout << "Site: " << e << endl;

        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];
                //if (s==0 && i==0 && j==0) cout << "Laplacian:    " << laplacianj <<
                    ↪  endl;
                //if (s==0 && i==0 && j==0) cout << "Pij:          " << Pij << endl;
                //if (s==0 && i==0 && j==0) cout << "Neighbor sum: " << neighbor_sum
                    ↪ << endl;
                dte[i] += Pij * laplacianj;
            }
        }
        //if (s==0) cout << "deriv " << dte << endl;
        //if (s==0) cout << "deriv " << h*dte << endl;

        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);
            //cout << "took measurement at " << t_ << endl;
            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;
            //cout << "t_: "<<t_<< "\terror: " << error << "\th: " << h << "\
                ↪ tmax_error: "<<MAXERROR<<endl;
            if (error>MAXERROR) {
                //cout << "RERUN" << endl;
                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;
        //Lattice2D operator+ (const Lattice2D & other) const;

        // Iterator stuff
        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

```
// 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) {
    // S[phi] = beta/2 int (dphi . dphi)
    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};
    //for (int i=0; i<N; i++) {
        //phi[i] = 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);
    //for (int i=0; i<N; i++) {
        //new_phi[i] = phi[i] + aPhi[i];
    //}
    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 C : public BaseObservable {
    //public:
        //double operator()(const Lattice2D& lat) const {
            //double val = 0;
            //const Phi phi0 = lat[0];
            ////for (const Phi& phi : as_const(lat)) {
            //for (auto it = lat.cbegin(); it!=lat.cend(); ++it) {
                //val += (*it)*phi0;
            //}
            //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;

            //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( 2pi_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
```

Bibliography

[1] Victor F. Weisskopf. Development of field theory in the last 50 years. *Physics today*, 34(11):69, 1981.

[2] Christopher Monahan. The gradient flow in simple field theories. In *Proceedings of The 33rd International Symposium on Lattice Field Theory — PoS(LATTICE 2015)*, page 052, Kobe International Conference Center, Kobe, Japan, July 2016. Sissa Medialab.

[3] Wolfgang Bietenholz, Philippe de Forcrand, Urs Gerber, Héctor Mejía-Díaz, and Ilya O. Sandoval. Topological Susceptibility of the 2d O(3) Model under Gradient Flow. *Phys. Rev. D*, 98(11):114501, December 2018.

[4] Anthony Zee. *Quantum Field Theory in a Nutshell*, volume 7. Princeton university press, 2010.

[5] Richard P. Feynman, Robert B. Leighton, and Matthew L. Sands. The Principle of Least Action. In *The Feynman Lectures on Physics*, volume II of *World Student Series*. Addison-Wesley Pub. Co., Reading, Mass., 1963.

[6] B Berg and Martin Lüscher. Definition and statistical distributions of a topological number in the lattice $O(3)$ σ-model. *Nuclear Physics B*, 190(2):412–424, 1981.

[7] Stefan Solbrig, Falk Bruckmann, Christof Gattringer, Ernst-Michael Ilgenfritz, Michael Müller-Preussker, and Andreas Schäfer. Smearing and filtering methods in lattice QCD - a quantitative comparison. *arXiv:0710.0480 [hep-lat]*, October 2007.

[8] Christopher Monahan and Kostas Orginos. Locally smeared operator product expansions in scalar field theory. *Phys. Rev. D*, 91(7):074513, April 2015.

[9] Martin Lüscher. Chiral symmetry and the Yang–Mills gradient flow. *J. High Energ. Phys.*, 2013(4):123, April 2013.

[10] Hiroki Makino and Hiroshi Suzuki. Renormalizability of the gradient flow in the 2D $O(N)$ non-linear sigma model. *Progress of Theoretical and Experimental Physics*, 2015(3):33B08–0, March 2015.

[11] David A Schaich and William Loinaz. *Lattice Simulations of Nonperturbative Quantum Field Theories*. PhD thesis, Amherst College, May 2006.

[12] Ulli Wolff. Monte Carlo errors with less errors. *Computer Physics Communications*, 176(5):383, March 2007.

[13] William T Vetterling, William T Vetterling, William H Press, William H Press, Saul A Teukolsky, Brian P Flannery, and Brian P Flannery. *Numerical Recipes: Example Book c*. Cambridge University Press, 1992.