Lattice Simulations of Nonperturbative Quantum Field Theories [1]

David A. Schaich
Prof. William Loinaz, Faculty Advisor

12 May 2006

[1] Submitted to the Department of Physics of Amherst College in partial fulfillment of the requirements for the degree of Bachelor of Arts with Distinction.

Acknowledgements

First I thank my parents, without whose continuous assistance, encouragement and moral and material support over the past twenty-odd years I would not have been able to undertake this work, let alone complete it.

My advisor, Prof. Will Loinaz also has my deep gratitude – for his assistance, encouragement, deep knowledge, and for his willingness and ability to assist his students in any way necessary.

I also thank Prof. Kaplan of the Department of Mathematics and Computer Science for his work building and maintaining Amherst's computing cluster, as well as promptly and competently assisting with the related questions and problems.

Finally, I would like to acknowledge the rest of the Amherst College Department of Physics, my teachers, friends, and constant companions over the past four years.

Software used extensively in this work includes Condor cluster management software, Mathematica, ROOT, and the GNU Scientific Library.

This work was supported by National Science Foundation grant 0521169 and a Faculty Research Award Program grant from Amherst College.

Abstract

We give a brief pedagogical introduction to Markov chain Monte Carlo simulations of statistical systems and the basics of the ϕ^4 quantum field theory. We show how to treat ϕ^4 theory as a statistical system in Euclidean space and perform lattice simulations using both local and cluster Monte Carlo algorithms. We then present the details and results of simulations of ϕ^4 theory on the lattice in which we chart the phase transition line of the theory in two and four dimensions and calculate an accurate value of the continuum critical coupling constant in two dimensions. Finally we discuss lattice simulations that calculate the masses of ϕ^4 solitons in two dimensions, nonperturbative phenomena that result from the nonlinearity of the theory.

The two-dimensional critical coupling constant we obtain is $[\lambda/\mu^2]_{crit} = 10.85^{+.03}_{-.08}$, which disagrees by over 7σ with the value of $[\lambda/\mu^2]_{crit} = 10.26^{+.08}_{-.04}$ reported by Loinaz and Willey [32]. We show that this disagreement is the result of higher-order dependence of $[\lambda/\mu^2]_{crit}$ on λ, which is only revealed by our more extensive calculations and resulting increase in precision; our data is actually largely consistent with theirs. Our calculations of ϕ^4 soliton masses improve upon the previous study by Ciria and Tarancón [15]; although our simulation data appears to agree with theirs we note and correct an error in their analysis.

Contents

List of Figures

List of Tables

List of Code Snippets

Chapter 1

Introduction

The basic goal of elementary particle physics is to determine the nature of fundamental particles and their interactions. This is easier said than done.

As far as physicists have been able to determine, particles and interactions are best described by quantum field theory, which combines quantum mechanics and special relativity. Quantum field theories are often studied perturbatively: the particles (such as photons and electrons) are first considered by themselves and then the interactions are added in, beginning with the strongest interactions and continuing with weaker ones until a point is reached at which adding further interactions won't significantly affect the result and can be neglected.

This approach is not always valid, particularly when the system under consideration cannot be described by applying a small perturbation to a simpler, solvable system. For example, this is the case for low-energy quantum chromodynamics, collective phenomena such as solitons, and indeed for nonlinear quantum field theories. Performing numerical simulations on computers is a powerful tool for studying such nonperturbative quantum field theories. This approach, lattice quantum field theory, has made major strides as computing power has increased in recent years.

In this work we use lattice simulations to investigate nonperturbative features of the relatively simple ϕ^4 quantum field theory. After a brief pedagogical introduction to Markov chain Monte Carlo simulations of statistical systems and the basics of the ϕ^4 quantum field theory, we show how to treat the ϕ^4 theory as a statistical system in Euclidean space and perform lattice simulations using both local and cluster Monte Carlo algorithms. We then present the details and results of simulations of ϕ^4 theory on the lattice in which we chart the phase transition line of the theory in two and four dimensions and calculate an accurate value of the continuum critical coupling constant in two dimensions. Finally we discuss lattice simulations that calculate the masses of ϕ^4 solitons in two dimensions, nonperturbative phenomena that result from the nonlinearity of the theory.

The two-dimensional critical coupling constant we obtain is $\left[\lambda/\mu^2\right]_{crit} = 10.85^{+.03}_{-.08}$, which disagrees by over 7σ with the value of $\left[\lambda/\mu^2\right]_{crit} = 10.26^{+.08}_{-.04}$ reported by Loinaz

and Willey [32]. We show that this disagreement is the result of higher-order dependence of $[\lambda/\mu^2]_{crit}$ on λ, which is only revealed by our more extensive calculations and resulting increase in precision; our data is actually largely consistent with theirs. Our calculations of the ϕ^4 soliton mass improve upon the previous study by Ciria and Tarancón [15]; although our simulation data appears to agree with theirs we note and correct an error in their analysis.

1.1 The Purposes of This Work

This document has two chief purposes. First, it is a report on the work we have done for Physics 77 and Physics 78D on lattice simulations of nonperturbative features of simple quantum field theories and related topics. However, it is designed in large part as a pedagogical introduction to these subjects (including lattice simulations, Monte Carlo methods, simple scalar quantum field theory and solitons). Our discussions are targeted at intermediate to advanced undergraduates, especially those who may conduct similar thesis projects in coming years. As a result, calculations and derivations are generally worked out in more detail than would otherwise be expected.

We assume the reader has taken introductory courses on statistical mechanics or thermodynamics, classical mechanics and quantum mechanics, and has some knowledge of related mathematics, including aspects of complex analysis such as contour integration and the residue calculus. We will also assume some basic familiarity with programming, though useful tools such as **gdb** and **gprof** as well as data structures such as hash tables will be introduced in an appendix.

1.2 Outline

We begin with a brief review of relevant statistical concepts and formulae in Chapter 2 before introducing in Chapter 3 basic concepts of computer simulations such as pseudorandom number generators as well as simple applications such as simulations of random walks. This introduction is followed in Chapter 4 by a more rigorous consideration of Markov chain Monte Carlo methods. We then explore in Chapter 5 particular Monte Carlo algorithms such as the Metropolis algorithm and Wolff cluster algorithm in the context of the Ising model of a ferromagnet. We also use the Ising model to introduce phases, phase transitions, and spontaneous symmetry breaking.

We then turn to quantum field theory in Chapter 6, in particular the simple self-interacting scalar theory known as ϕ^4 theory. While a full introduction to quantum field theory is far beyond the scope of this work, we briefly explore some of the central issues in the context of ϕ^4 theory, including the interpretation of ϕ in terms of particle creation and annihilation; the existence of ϕ^4 solitons as a result of the nonlinearity of the theory; Feynman rules and perturbation expansions in terms of Feynman diagrams; as well as the resulting divergences and ways to address them through renormalization.

In Chapter 7 we show how this quantum field theory can be treated as a statistical system and discretized for simulation on the lattice. We discuss the details and results of our lattice simulations of ϕ^4 theory, including analyses of the phase transitions of the two- and four-dimensional theory and determination of the continuum critical coupling constant in two dimensions. We also present the results of our calculations of ϕ^4 soliton masses in two dimensions.

The qualified reader may wish to turn directly to Chapter 7, which focuses most directly on our simulations and their results. We recommend that those interested in our investigation of phase transitions in the ϕ^4 theory be familiar with the contents of Sections 5.1 through 5.5, which in turn depend on the ideas developed in Chapter 4. Those interested in our simulations of solitons would do well to review the material in Sections 6.3 and 6.4 as well. The information in the other chapters may be neglected to a first approximation.

We also include some appendices discussing more distantly related matters: Amherst College's scientific computing cluster, on which our computations were performed (Appendix A); habits, tricks and tools helpful for efficient programming (Appendix B); the full C++ code of some of our simulations (Appendix C); and the full results of the regressions used to obtain our final critical coupling constant results (Appendix D).

1.3 Note on Units

Following the standard conventions of current research, we will work exclusively in 'natural units' (where $c = 1$ and $\hbar = 1$) when discussing quantum field theory in the context of high energy physics and 'energy units' (where the Boltzmann factor $k_B = 1$) when considering thermodynamics and condensed matter physics more generally. In natural units, energy, momentum and mass all have the same dimension $mass$, and length and time both have dimension $mass^{-1}$. These 'mass dimensions' (or 'energy dimensions' or 'canonical dimensions') will be expressed in the convenient shorthand $[x] = k$, signifying that quantity x has mass dimension $mass^k$.

Chapter 2

Statistical Background

Analyses of statistical systems and simulations obviously rely on statistics and considerations of statistical uncertainty in addition to systematic effects. In this chapter we will briefly review basic concept of statistics and error analysis that should be familiar (variance, standard deviation, error propagation), introduce some more specialized methods, and address complications introduced by correlated measurements. Taylor [60] is a useful reference.

2.1 Familiar Concepts

The central concepts are the mean and standard deviation of a series of measurements of some observable Q. For N uncorrelated measurements of equal weight, the mean and standard deviation take the familiar forms

$$\langle Q \rangle = \frac{1}{N} \sum_i^N Q_i, \tag{2.1}$$

$$\sigma = \sqrt{\frac{1}{N-1} \frac{1}{N} \sum_i^N \left(Q_i - \langle Q \rangle \right)^2} = \sqrt{\frac{1}{N-1} \left(\langle Q^2 \rangle - \langle Q \rangle^2 \right)}. \tag{2.2}$$

Slightly more generally, if measurement Q_i has weight w_i, the mean becomes

$$\langle Q \rangle = \frac{\sum_i^N w_i Q_i}{\sum_i^N w_i}, \tag{2.3}$$

which reduces to Eqn. 2.1 when all measurements have equal weight $w_i = 1/N$. The denominator is required for normalization; it is often set to $\sum_i^N w_i = 1$, but this is not necessary. The proper form of weights for measurements with a given uncertainty is a complication that will be addressed in the next section.

Another familiar result is the propagation of uncertainties. For a function F of independent variables x_1, $x_2 \ldots x_N$, each with independent uncertainty δx_1, $\delta x_2 \ldots \delta x_N$,

the overall uncertainty δF in F is given by

$$\delta F = \sqrt{\left(\frac{\partial F}{\partial x_1}\delta x_1\right)^2 + \left(\frac{\partial F}{\partial x_2}\delta x_2\right)^2 + \cdots + \left(\frac{\partial F}{\partial x_N}\delta x_N\right)^2}. \tag{2.4}$$

For example, the uncertainty in the critical coupling constant $[\lambda/\mu^2]$, where λ is fixed and μ^2 has uncertainty $\delta\mu^2$ is given by

$$\delta\frac{\lambda}{\mu^2} = \left|\frac{d}{d\mu^2}\frac{\lambda}{\mu^2}\right|\delta\mu^2 = \frac{\lambda\delta\mu^2}{(\mu^2)^2}. \tag{2.5}$$

The inverse dependence on $(\mu^2)^2$ makes it difficult to keep the uncertainty from increasing dramatically as we consider the continuum limit $\mu^2 \to 0$. This is unfortunate, since the simulations with the smallest μ^2 are the most useful for determining the continuum limit.

2.2 Complications: Weights and Correlations

As we will see in later sections, measurements of systems being simulated with Monte Carlo methods are generally partially correlated and not completely independent. This means that the familiar formula for the standard deviation, Eqn. 2.2, cannot be directly applied since it assumes uncorrelated measurements. Newman and Barkema [40, Chap. 3] present the following result for the standard deviation of N correlated measurements:

$$\sigma = \sqrt{\frac{1+2\tau}{N-1}\left(\langle Q^2\rangle - \langle Q\rangle^2\right)}. \tag{2.6}$$

τ is the autocorrelation time and is a measurement of how long (how many measurements) it takes for the system to produce a state which is statistically independent of the initial state. Eqn. 2.6 clearly approaches Eqn. 2.2 as $\tau \to 0$. Typically in our work, however, both $N \gg 1$ and $\tau \gg 1$, in which case we have

$$\sigma \approx \sqrt{\frac{2\tau}{N-1}\left(\langle Q^2\rangle - \langle Q\rangle^2\right)} \approx \sqrt{\frac{2\tau}{N}\left(\langle Q^2\rangle - \langle Q\rangle^2\right)} \tag{2.7}$$

As Eqn. 2.7 suggests, measurements are typically held to be fully independent if $\tau \leq \frac{1}{2}$, that is, if each measurement is taken two or more autocorrelation times after the previous one.

The autocorrelation time is more rigorously defined as the typical time-scale on which the time-displaced autocorrelation function $\chi(t)$ falls off. The "time-displaced" autocorrelation function measures correlation across a series of measurements, in contrast to "spatially"-displaced autocorrelation functions, which measure correlation between different observables or features of the system at a single measurement and produce autocorrelation lengths ξ corresponding to the typical length-scale on which they fall off. $\chi(t)$ is defined as

$$\chi(t) = \frac{1}{t_{max}-t}\left[\sum_{t'=0}^{t_{max}-t} Q(t')Q(t'+t)\right] - \langle Q\rangle^2. \tag{2.8}$$

'Time' in this context refers to a series of measurements of some observable Q (typically the magnetization or vacuum expectation value of the system being studied), with each instant in time a separate partly-correlated measurement. t_{max} is the total number of measurements made over the course of the simulation. Now let us show a simple way to extract τ from $\chi(t)$. We explicitly assume that correlation (and thus $\chi(t)$) decreases exponentially as t increases, $\chi(t) \propto e^{-t/\tau}$. If the system being simulated is in equilibrium, this is a reasonable assumption that has been empirically verified in our work as well as that of many others. Such exponential dependence is clear in Fig. 2.1, which plots the scaled autocorrelation function $\chi(t)/\chi(0)$ over time for one of our Ising model simulations on a 32^2 lattice at $kT = 2.2$ with each measurement t representing five sweeps of the lattice with the Metropolis algorithm – see Chapter 5 for more information. Nonequilibrium simulations require different methods, and will not be considered in this thesis despite being an active field of research.

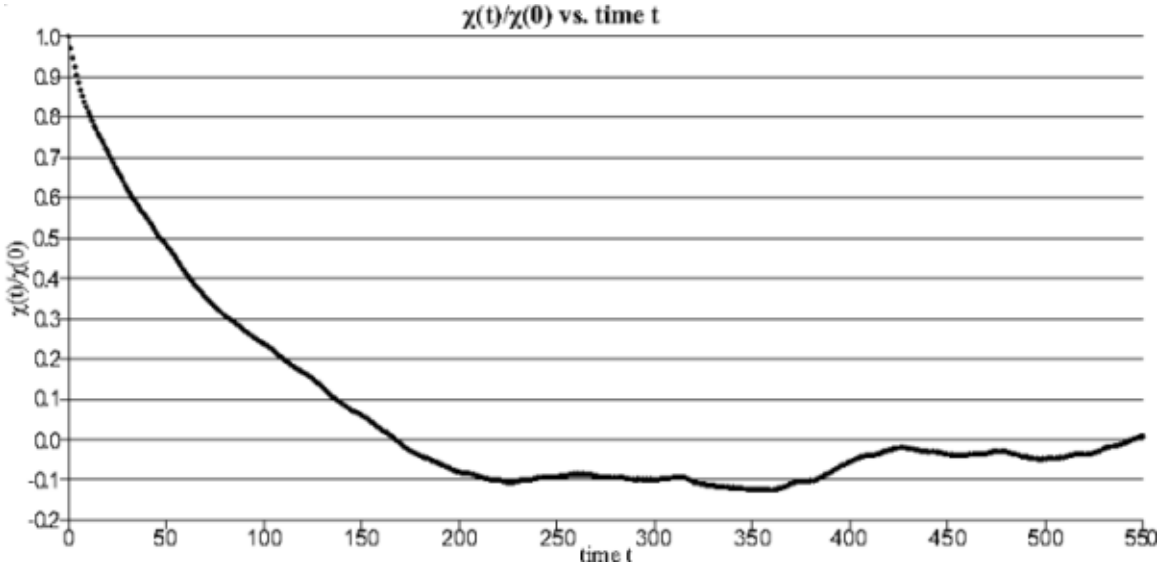

Figure 2.1: Scaled autocorrelation function over time (for Ising model on a 32^2 lattice at $kT = 2.2$)

We can extract the autocorrelation time τ from $\chi(t) = Ae^{-t/\tau}$ by scaling the function by $\chi(0) = A$. We then have

$$\frac{\chi(t)}{\chi(0)} = e^{-t/\tau} \rightarrow \tau = \frac{-t}{\log\left(\frac{\chi(t)}{\chi(0)}\right)}. \tag{2.9}$$

C++ code performing this calculation in the context of lattice simulations of ϕ^4 theory can be found in Code Snippet C.9 in Appendix C. We proceeded by calculating $\chi(t)$ until it deviated from exponential decay, either flattening out such that $\chi(t) > \chi(t-1)$ or dropping below zero $\chi(t) < 0$. We then calculated the autocorrelation time τ by extracting a τ_t from each of the $\chi(t)$ values for $t > 0$ and taking the average.

Simply taking the average suffices for calculating the autocorrelation time τ from the various τ_t values since we give them all the same weight. How would we determine the appropriate weights to give to measurements if we didn't have them beforehand? For

example, suppose we had several measurements of the same observable with different uncertainties. These uncertainties reflect our beliefs in the precision of the various measurements, and we want to take them into account when determining the best estimate. There is a straightforward way to determine the proper relationship between uncertainties and the weights, which we sketch out below. Taylor [60, Chap. 7] presents a more detailed derivation.

If we make the reasonable assumption that the uncertainties σ on the measurements are governed by a normal (Gaussian) distribution, we have

$$P(x) \propto \frac{1}{\sigma} \exp\left(\frac{-(x-X)^2}{2\sigma^2}\right), \tag{2.10}$$

where X is the unknown "true" value and x is the value measured. If we have two measurements of the same value X, x_a and x_b, the probability of measuring both values is the product of the probabilities of measuring each value individually,

$$P(x_a, x_b) = P(x_a)P(x_b) \propto \frac{1}{\sigma_a \sigma_b} \exp\left(\frac{-\chi^2}{2}\right);$$
$$\chi^2 = \left(\frac{x_a - X}{\sigma_a}\right)^2 + \left(\frac{x_b - X}{\sigma_b}\right)^2.$$

The best estimate is the value of X which minimizes χ^2, maximizing the probability. We can find the best estimate by setting the derivative of $P(x_a, x_b)$ with respect to X equal to zero:

$$2\frac{x_a - x_{best}}{\sigma_a^2} + 2\frac{x_b - x_{best}}{\sigma_b^2} = 0. \tag{2.11}$$

(We replace X with x_{best} since we are determining only the best estimate based on our measurements, not the unknown true value itself.) Eqn. 2.11 can easily be solved for x_{best} to give

$$x_{best} = \frac{x_a/\sigma_a^2 + x_b/\sigma_b^2}{1/\sigma_a^2 + 1/\sigma_b^2}.$$

Note that if we define $w_i = 1/\sigma_i^2$, this becomes

$$x_{best} = \frac{w_a x_a + w_b x_b}{w_a + w_b}, \tag{2.12}$$

which is simply a special case of Eqn. 2.3. Since this derivation can easily be generalized to N estimates, we see that all we need to do to determine the best estimate is set the weights w_i equal to the inverse square of the corresponding uncertainties σ_i.

As discussed in Baird [2], this is also the case for weighted regressions such as least squares fits: the weight to assign to each data point is the inverse square of its uncertainty. Programs such as Mathematica can handle this easily. It is worth noting that since the weights are proportional to the inverse square of the uncertainties (as opposed to, say, the inverse or inverse square root), data with relatively large uncertainties will have very small effects on the final result; the best estimates will be dominated by the most precise data.

2.3 Bootstrap and Jackknife Methods

Unfortunately, we can't easily apply the statistical methods developed above to certain quantities, in particular those that aren't repeatedly measured many times over the course of the simulation but rather depend on the measured quantities in some more complicated way. For instance, the specific heat is proportional to the variation in the energy over all the measurements, $C \propto \langle E^2 \rangle - \langle E \rangle^2$. The uncertainties in $\langle E^2 \rangle$ and $\langle E \rangle^2$ are correlated, so we can't just glibly propagate them using Eqn. 2.4. So what is to be done?

One intuitive approach would be to separate our measurements of E and E^2 into several groups, calculate $\langle E^2 \rangle_i$ and $\langle E \rangle_i^2$ for each group i, and use the standard deviation of those values as the statistical uncertainty in C. This method is generally reasonable, but the results it gives depend on the number of groups used, which should not be the case: statistical uncertainty should depend on the data, not the analysis. There are two related but more rigorous methods – known as the bootstrap and jackknife methods – that are more reliable than this simple approach.

Suppose we have N measurements of the energy and want to determine the specific heat. The bootstrap method orders us to select N of these measurements at random (note that the same measurement can be selected twice!) and calculate the specific heat using them. It has been shown that after this process has been repeated several (say, n) times, the total uncertainty in the specific heat is given by

$$\sigma = \sqrt{\langle C^2 \rangle - \langle C \rangle^2} \tag{2.13}$$

with $\langle C^2 \rangle$ and $\langle C \rangle^2$ calculated from the n different samples taken. Note that there is no factor of $1/(n-1)$ in this expression, since that would imply that σ could be made arbitrarily small simply by resampling the data enough times. Also, σ doesn't depend on the autocorrelation time τ, since under the bootstrap scheme correlated measurements can effectively be considered duplicates.

The bootstrap method generally gives good estimates of the uncertainty, which become exact as the amount of data available increases. The similar jackknife method is even more reliable, though it is not independent of the autocorrelation time. It works by first calculating an estimate for the specific heat C based on all N measurements. It then ignores the first measurement and recomputes C_1 based on the remaining $N-1$, repeating this process to calculate all possible C_i with the single i^{th} measurement ignored. The uncertainty in C itself is then given by

$$\sigma = \sqrt{2\tau \sum_{i=1}^{N} (C_i - C)^2}. \tag{2.14}$$

If the measurements are all independent, $\tau \leq \frac{1}{2}$, this becomes simply

$$\sigma = \sqrt{\sum_{i=1}^{N} (C_i - C)^2}. \tag{2.15}$$

Chapter 3

Basics of Computer Simulations

In this chapter we will introduce some of the basic issues underlying computer simulations, such as random number generation, lattices, and boundary conditions. In addition, we will explore random walks as simple applications of computational methods that will introduce concepts of algorithms and data structures.

3.1 Pseudorandom Numbers and Generators

Pseudorandom numbers produced by pseudorandom number generators (PRNGs) are used heavily in Monte Carlo simulations. PRNGs are deterministic programs that that produce sequences of numbers that are 'random enough' for the purposes of the simulation in a sense that will be clarified below. These generators are generally periodic, but this is not a major problem since the periods of modern PRNGs are large enough that programming in nonperiodicity is unnecessary in practice.[3.1] For example, the period of the Mersenne Twister algorithm used in this work is $2^{19937} - 1$ (about 10^{6000}). Beginning with an initial 'seed' state, each number in the sequence is generated from the previous 'state', which could depend on multiple preceding numbers. This allows for the sequence of numbers generated to be reproduced by running the same algorithm with the same seed. This reproducibility is important in scientific programming and simulations.

The sequences of numbers produced by PRNGs should "look random" by being uniformly distributed over the available range, and should have a long period and minimal correlation between subsequent values. If a PRNG obeys these properties, it is difficult (perhaps impossible) to distinguish its output from perfect random noise without knowing the seed. In the future, I will refer simply to "random numbers", omitting the implicit "pseudo".

This work uses exclusively the Mersenne Twister algorithm invented by Matsumoto and Nishimura [37] and implemented in the GNU Scientific Library (GSL) [20]. This

[3.1] Nonperiodic PRNGs have the disadvantage that the memory required to keep track of the preceding sequence and prevent repetition grows over time.

algorithm was designed specifically for Monte Carlo simulations, has a long period of $2^{19937} - 1$ and is equi-distributed in 623 dimensions. It is also very fast (in some architectures faster than the basic generators in system libraries, which are unsuitable for serious simulations) and has a very small memory footprint (requiring only 624 words of memory).

3.2 Random Walks

Random walks are a familiar concept from statistical mechanics with applications in diffusion, Brownian motion, protein folding, electrical networks, population genetics, and even mathematical analysis (see for instance Spitzer [56]). They can easily be implemented by simple algorithms, providing a gentle introduction to computer simulation. In this section we'll explore the basic random walk as well as some simple variants that produce interesting effects. We'll walk through a limited theoretical discussion, present the results of our simulations, and compare them to results obtained by other researchers.

3.2.1 Basic Random Walk

The basic random walk pictures a walker taking a single step in a randomly chosen direction (restricted for simplicity to the $2d$ Cartesian directions in d dimensions – left/right, up/down, etc.). The process then repeats. This is a simple example of an algorithm, a step-by-step procedure that is often, though not necessarily, repetitive or recursive.

An interesting feature of the basic random walk is the simple relationship between the number of steps N (of length l) taken by the walker and her average squared distance from the walk's origin: $\langle x^2 \rangle = Nl^2$. The one-dimensional case is (as one would expect) especially easy to analyze. After N steps $s_i = \pm l$ have been taken in the two possible directions, the squared distance is

$$
\begin{aligned}
x^2 &= \left(\sum_i^N s_i \right)^2 = (s_1 + s_2 + \ldots s_N)(s_1 + s_2 + \ldots s_N) \\
x^2 &= \left(s_1^2 + s_2^2 + \ldots s_N^2 \right) + s_1 (s_2 + s_3 + \ldots s_N) + \cdots + s_N (s_1 + s_2 + \ldots s_{N-1}) \\
x^2 &= \sum_1^N s_i^2 + \sum_{i \neq j}^N s_i s_j .
\end{aligned}
$$

Since $s_i s_j = \pm l$ with equal probability, the second sum averages out to zero, while the first is simply Nl^2. Thus in one dimension

$$\langle x^2 \rangle = Nl^2. \tag{3.1}$$

In fact, this relationship holds for arbitrary dimensions, though the summations become steadily more cumbersome as the dimensionality of the walk increases. Table 3.1

Table 3.1: Predictions and results for basic random walks

Dimension	Predicted Relationship	Results
2	$\langle x^2 \rangle = N$	$\langle x^2 \rangle = 0.9998(5)N$
3	$\langle x^2 \rangle = N$	$\langle x^2 \rangle = 1.0010(4)N$
4	$\langle x^2 \rangle = N$	$\langle x^2 \rangle = 0.9994(4)N$

and Fig. 3.1 present the results of simulations (using code included in Code Snippet C.1) of basic random walks in two, three and four dimensions. The walks used step length $l = 1$ and performed 10^5 walks of lengths $N = 1 \ldots 150$. For simplicity, we will set $l = 1$ from now on.

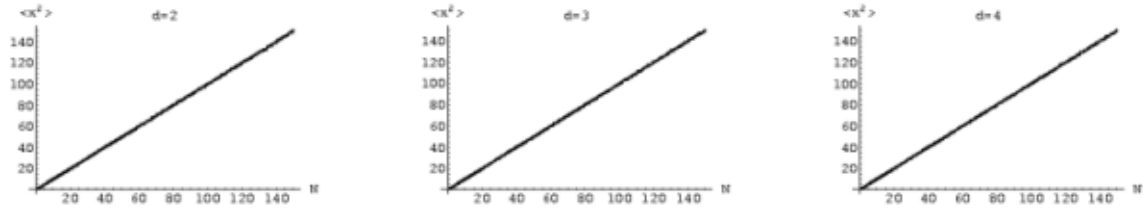

Figure 3.1: Basic random walks in two, three and four dimensions

3.2.2 Constrained Random Walks

Adding some simple constraints to the basic random walk can produce interesting effects. In a nonreversal random walk, the walker cannot reverse course and immediately return to the site just vacated. In a self-avoiding random walk, the walker can never return to any site she has already visited. If all the sites accessible from the walker's current position have already been visited, the walk is aborted and restarted (with a *different* sequence of random numbers determining the steps taken) and no data is collected. Clearly in one dimension $\langle x^2 \rangle = N^2$ for both of these algorithms – once the walker takes one step in a particular direction, she can never turn around and must continue travelling in that direction for all N steps.

A simple heuristic argument suggests that in higher dimensions the distance traveled by nonreversal random walks should be slightly more than that travelled in basic random walks, but still proportional to N. When a random walker reverses a step (which will happen with probability $(2d)^{-1}$), the two steps effectively cancel out: the walker ends up the same distance away from her starting point, with two fewer steps left to take. Forbidding reversal would prevent this from occurring, and thus should produce a basic random walk of

$$N + N \sum_i \left(\frac{2}{2d} \right)^i = N \left(1 + \sum_i \frac{1}{d^i} \right)$$

Table 3.2: Predictions and results for nonreversal random walks

Dimension	Predicted Relationship	Results
2	$\langle x^2 \rangle = 2N$	$\langle x^2 \rangle = 1.9993(10)N$
3	$\langle x^2 \rangle = (3/2)N$	$\langle x^2 \rangle = 1.5018(6)N$
4	$\langle x^2 \rangle = (4/3)N$	$\langle x^2 \rangle = 1.3331(5)N$

steps.[3.2] So the average squared distance would simply be

$$\langle x^2 \rangle = N \left(1 + \sum_i \frac{1}{d^i} \right). \tag{3.2}$$

Results presented in Table 3.2 verify this prediction in two, three and four dimensions. Though the three-dimensional result is a few σ from the prediction, they are generally in excellent agreement.

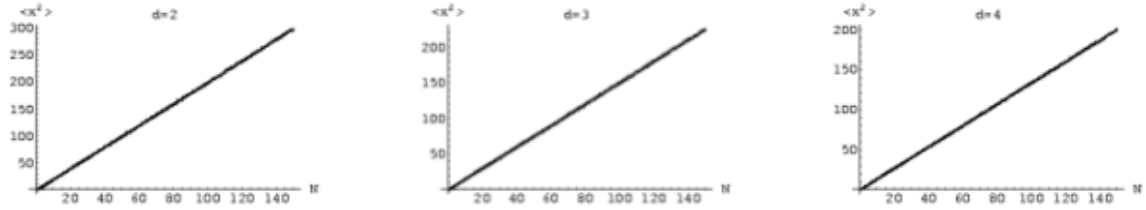

Figure 3.2: Nonreversal random walks in two, three and four dimensions

The most interesting complication presented by self-avoiding random walk simulations is the chance that the walk might have to be aborted and restarted should the walker 'paint herself into a corner'. This is most common in two dimensions, since the line traced out by the walk divides the plane and severely constrains the future path of the walk. Such dead-end walks can still occur in three- and four-dimensional simulations as well, though they are much less common since it is generally possible to go around lines in three or four dimensions – the walker would need to be boxed in across all dimensions.

Self-avoiding random walks also pose an interesting programming problem, since at each potential step the history of the walk must be consulted to ensure the walker avoids her past path. There are two naïve ways to do this, both of which perform poorly. The actual history of the walk can be stored in an array or vector and searched for each potential step, but this scheme has the obvious drawback that all these searches would waste a lot of time, especially as the length of the walk increases. Alternately, a d-dimensional lattice of Boolean values could record whether or not a site has already been visited. This seems more reasonable, but starting or restarting a walk requires initializing or clearing the lattice, which can take a significant amount of time, especially as the dimensionality of the lattice increases (the number of sites in the lattice increases like $(2N)^d$). A better solution is to use a hash table, which allows for constant-time operations on average. Using this hash table in place of a Boolean lattice decreased the

[3.2] It might seem that only N/d additional steps would be required. Note, however, that each of these additional steps has a $(2d)^{-1}$ chance of being a reversal, so more steps need to be added, producing the sum.

Table 3.3: Results for self-avoiding random walks

Dimension	$\langle x^2 \rangle = N^a$ model	Variance	$\langle x^2 \rangle = aN^b$ model	Variance
2	$\langle x^2 \rangle = N^{1.3201(2)}$	17.10	$\langle x^2 \rangle = 0.853(7)N^{1.3535(16)}$	4.35
3	$\langle x^2 \rangle = N^{1.1259(4)}$	16.40	$\langle x^2 \rangle = 1.461(6)N^{1.0456(9)}$	0.29
4	$\langle x^2 \rangle = N^{1.0774(4)}$	6.44	$\langle x^2 \rangle = 1.338(4)N^{1.0157(7)}$	0.11

running time of our four-dimensional self-avoiding random walk simulation by nearly four orders of magnitude. Source code for the hash table and a more detailed discussion of the issue can be found in Section B.3, below.

An additional technical consideration is whether or not to select a direction and check it or to check all potential steps and choose randomly only among those that do not violate the self-avoidance requirement. As shown below in Code Snippet C.3, we proceeded by checking all the potential destinations of a step from the current site, and then randomly selecting only among those that had not been visited previously. Even though it investigates every neighboring site, this approach is on average much more efficient than the alternative. This is because it avoids repeatedly visiting sites that may be forbidden and also makes it immediately apparent whether or not the walk has reached a dead-end.

The average squared distance travelled by an N-step self-avoiding random walk is more difficult to determine analytically than in the basic and non-reversal cases, since the whole history of the walk affects its future development. In general, the results presented in Table 3.3 and Figure 3.3 show that there is a nonlinear relationship $\langle x^2 \rangle \sim N^b$ that becomes less pronounced as the dimensionality of the walk increases. In fact, as the number of dimensions increases, the self-avoiding walk approaches the linear relation characteristic of the basic and nonreversal walks. This is sensible: as the number of dimensions increases, the chances that the walker will get trapped in a 'corner' (or even stumble across her past path) decrease, and the relation should gradually approach that of the nonreversal random walk. This suggests that there should be a multiplicative factor (so $\langle x^2 \rangle \sim aN^b$) that will gradually approach that of the nonreversal random walk, $\left(1 + \frac{1}{d}\right)$, as d increases. As shown in Table 3.3, adding this multiplicative factor dramatically reduces the variance per degree of freedom, though it remains quite high in the two-dimensional case.

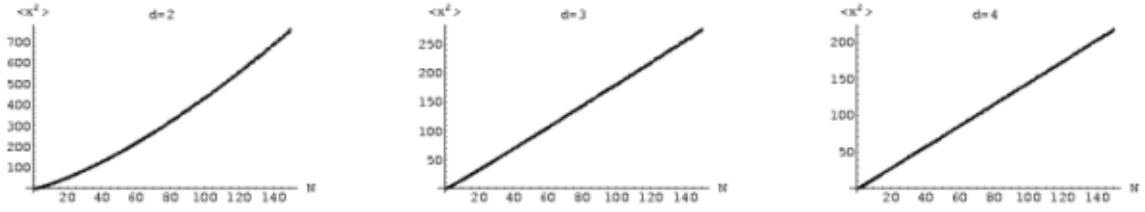

Figure 3.3: Self-avoiding random walks in two, three and four dimensions

3.2.3 Comparison to Others' Results

A massive amount of research has been performed on random walks of all sorts and their applications to various systems and disciplines. Although we only derived the relationship between N and $\langle x^2 \rangle$ in one dimension and presented a heuristic argument for those of nonreversal walks in arbitrary dimensions, analytical results have been derived and other simulations performed for all of these scenarios (and many others besides). For basic random walks, the generalization that $\langle x^2 \rangle = N$ in higher dimensions is well-known. A proof for the two dimensional case is performed by Weisstein [67]. It is similar to that in one dimension, only with s_i represented as a phasor instead of ± 1.

Constrained random walks are considerably more complex. The general expression

$$\langle x^2 \rangle = AN^{\nu} \left(1 + BN^{-\Delta} + CN^{-1} + \cdots\right) \tag{3.3}$$

appears to be generally accepted (Δ is the leading correction-to-scaling exponent), though Caracciolo *et al.* [10] present somewhat more complicated expressions. Shannon, Choy and Fleming [53] claim $\nu = \frac{3}{2}$ and $\Delta = -\frac{1}{2}$ in two dimensions, and Sokal [55] as well as Izyumov and Samokhin [25] report $\nu \approx 1.18$ in three dimensions. The older paper by Lawler [30], which predates Eqn. 3.3, lists $\nu = \frac{3}{2}$ in two dimensions, $\nu = \frac{6}{5}$ in three dimensions and actually claims $\langle x^2 \rangle = AN \log N$ in four dimensions. While these values for ν don't agree particularly well with our numerical results[3.3], they are all of the same order and the qualitative features of power-law scaling with exponents decreasing as the dimensionality of the walk increases are the same. Since our analysis did not attempt to take into account correction-to-scaling or fit to a series of multiple terms, we would not necessarily expect to duplicate these more detailed results.

In addition, Chris Bednarzyk '01 studied random walks in two dimensions as part of his thesis, [5]. He presents analytical results (without citation or explanation) for all three varieties, predicting $\langle x^2 \rangle = N^{4/3}$ for two-dimensional self-avoiding walks, and $\langle x^2 \rangle = 3N$ for two-dimensional nonreversal walks. While the former result is not too far off from our results (and Chris's own simulations found $\langle x^2 \rangle = N^{1.314}$, which he judges consistent with the prediction), the latter disagrees by a factor of 2 with both our simulations and heuristic argument.

3.3 Lattices and Boundary Conditions

In our later simulations, we will typically study statistical systems or field theories by simulating them on discrete lattices on the computer. Each site in these lattices will represent a point in spacetime identified with either a field value or a component of the thermodynamic system. The site will interact with neighboring sites through dynamics determined by the Lagrangian of the theory and (if applicable) the discretization process used to convert from the continuum theory to the discrete version simulated on the lattice.

[3.3] And a fit of our data to Lawler's prediction of $\langle x^2 \rangle = AN \log N$ produces a variance per degree of freedom of over 68.

These issues will all be discussed in greater detail in later chapters. However, before the lattice can be glibly manipulated, certain of its basic features need to be clarified. In particular, since all our simulations are carried out on finite lattices, the boundary conditions applied to the lattice take on considerable importance. In the interest of treating all lattice sites the same and creating the illusion that the lattice doesn't actually have an edge, periodic boundary conditions are commonly used. Periodic boundary conditions identify opposing sides of the lattice (in two dimensions creating a torus, for example), as illustrated in Fig. 3.4.[3.4] In the figure, only one nine-site lattice in one 3x3 corner of the illustrated array actually exists in the computer's memory, but the boundary conditions make it seem as though the lattice periodically stretches off arbitrarily far in all directions.

1	**2**	**3**	*1*	*2*	*3*
4	**5**	**6**	*4*	*5*	*6*
7	**8**	**9**	*7*	*8*	*9*
1	*2*	*3*	*1*	*2*	*3*
4	*5*	*6*	*4*	*5*	*6*
7	*8*	*9*	*7*	*8*	*9*

1	**2**	**3**	*4*	*5*	*6*
4	**5**	**6**	*7*	*8*	*9*
7	**8**	**9**	*1*	*2*	*3*
1	*2*	*3*	*4*	*5*	*6*
4	*5*	*6*	*7*	*8*	*9*
7	*8*	*9*	*1*	*2*	*3*

Figure 3.4: Periodic (left) and helical (right) boundary conditions

An alternate scheme is the "helical" setup proposed by Newman and Barkema [40] and compared to periodic boundary conditions in Fig. 3.4. In this arrangement, the left- and right-hand neighbors of a given site n are $n \pm 1$ (modulo the size of the lattice), even if this involves wrapping around the last element on one row to be the neighbor of the first element on the next row. Although this is slightly more complex to visualize than simple periodic boundary conditions (especially in higher dimensions), it has the compensating virtues of being both more general and easier to implement.

The increased generality is due to the fact that the total number of sites in the lattice no longer has to be equal to the product of the dimensions of the lattice – that is, the lattice no longer needs to be rectangular. Fig. 3.5 shows a lattice with a prime number of sites using helical boundary conditions. In practice, lattices are usually chosen to be squares, cubes and four-cubes, but the added generality does not hurt. Helical boundary conditions are slightly simpler to implement since neighboring sites to the left and right are always separated by 1 (modulo the size of the lattice), and those above and below separated by the width of the lattice modulo its size. For periodic boundary conditions, on the other hand, these relationships depend on the position of the site within the lattice (specifically, whether or not it's on an edge).

Even though helical boundary conditions thus offer some advantages and were the scheme we used for most of our simulations, periodic boundary conditions are so widely used that it makes sense to adopt them nonetheless. After all, in our simulations we used square lattices and set up data structures containing all the neighbors of all the lattice

[3.4] Fig. 3.4 shows a two-dimensional lattice stored in a one-dimensional array, the same scheme we used in our simulations. (The figure counts from 1 for purely aesthetic reasons.) It is also possible to use a multi-dimensional array, but this introduces the unnecessary complication of having to generate multiple random numbers to identify a site.

1	**2**	**3**	*4*	*5*	*6*
4	**5**	**6**	*7*	*1*	*2*
7	*1*	*2*	*3*	*4*	*5*
3	*4*	*5*	*6*	*7*	*1*
6	*7*	*1*	*2*	*3*	*4*
2	*3*	*4*	*5*	*6*	*7*

Figure 3.5: Helical boundary conditions on a seven-site lattice

sites while initializing the lattice (see the code in Appendix C, below), making the choice of boundary condition essentially irrelevant to the overall running time of the program. Under these conditions, it is best to adopt the boundary conditions most widely known and trusted, regardless of their relative simplicity or efficiency. Since we used helical boundary conditions for many of our programs, we felt it advisable to verify (at least for small lattices up to 128^2) that our ϕ^4 theory simulations give the same results regardless of which boundary conditions we used. This is indeed the case; typical results are shown in Fig. 3.6, in which all data points agree well within uncertainty (Fig. 3.6 includes error bars, but they are too small to be readily visible).

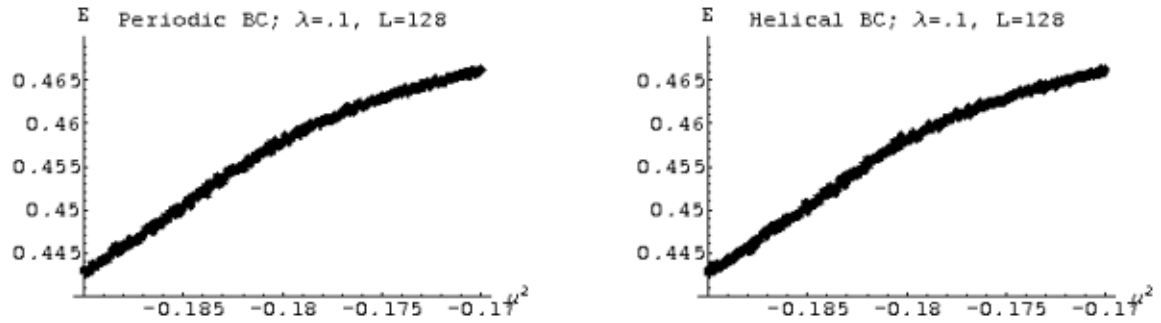

Figure 3.6: Energy of ϕ^4 simulations using periodic (left) and helical (right) boundary conditions at $\lambda = .1$ and $L = 128$

+	+	-	-	+	+	-	-
+	+	-	-	+	+	-	-
+	+	-	-	+	+	-	-
+	+	-	-	+	+	-	-
+	+	-	-	+	+	-	-

Figure 3.7: Schematic illustration of antiperiodic boundary conditions

Simulations for measuring the mass of the soliton need to use both periodic and 'antiperiodic' boundary conditions, as described below in Section 7.6. Antiperiodic boundary conditions identify the same points as periodic boundary conditions, but negate the value of the lattice site being wrapped around. Fig. 3.7 illustrates this concept for a lattice using periodic boundary conditions on its vertical (time) axis and antiperiodic boundary conditions on its horizontal (space) axis. For simplicity, the figure shows only the apparent signs of the lattice values as seen by neighboring sites that have been wrapped around. The actual sign of the lattice value does not change. In addition, the patterns

of identification do not change – the underlying topology is still that of a torus, but one in which the field values are negated after each complete circuit of the spatial axis. See Section C.4 for details of the implementation.

Chapter 4

Markov Chain Monte Carlo

Monte Carlo methods are stochastic processes that use random numbers to simulate the behavior of physical systems. Monte Carlo simulations are performed using the importance sampling technique of Markov processes (or Markov chains). As we will see, Markov chains allow for the simulation to efficiently reproduce the Boltzmann distribution that characterize the physical systems under consideration.

After briefly illustrating why stochastic processes must be used to perform efficient simulations, we will introduce the concept of importance sampling and show how to reproduce the Boltzmann distribution using Markov chains.

4.1 Motivation

You will pay the price for your lack of vision. – Emperor Palpatine

Suppose we wanted to simulate a statistical mechanical system in equilibrium on a computer. How would we? Well, the simplest thing to do would be to set up each possible state of the system, measure the quantities of interest for that state, weighted by the state's Boltzmann factor (the probability of the system being in the state):

$$\langle Q \rangle = \frac{\sum_\mu Q_\mu e^{-E_\mu/kT}}{\sum_\mu e^{-E_\mu/kT}} = \frac{\sum_\mu Q_\mu e^{-\beta E_\mu}}{\sum_\mu e^{-\beta E_\mu}} \tag{4.1}$$

This is simply the definition of the expectation value $\langle Q \rangle$.

Although this approach is simple, it pays the price for its lack of vision, since interesting systems typically have a large number of states that would need to be simulated. Consider, for example, a 16x16 lattice of points, each of which can take one of only two values. This tiny system has roughly $2^{256} \sim 10^{77}$ configurations which would need around 10^{66} years to fully simulate.[4.1] A 512x512 lattice would take more like $10^{78,900}$ years.

[4.1] Though the number can be reduced slightly by symmetry considerations – the system possesses reflection symmetries, inversion symmetry, and four-fold rotation symmetry.

The situation gets even worse for small thermodynamic systems. For instance, a mole of gas room temperature and atmospheric pressure has roughly 10^{23} molecules, each with a de Broglie wavelength of roughly $10^{-10}m$. Fully simulating every state of this system will require roughly $10^{10^{23}}$ years, which will take around $10^{10^{23}}$ times the age of the Universe.

Since that's a bit longer than most of us are willing to wait for results, we have to conclude that this straightforward approach is unacceptable. Equally unsuitable would be just randomly sampling the few states that could be simulated in a reasonable amount of time. Perhaps 10^9 states of the 16x16 two-value lattice could be considered in a reasonable time by a modern computer, representing a minuscule 10^{-68th} of the total number of states, almost all with vanishingly small Boltzmann factors. Any results calculated from such a small random subset of states would be meaningless.

4.2 Importance Sampling

The resolution to this dilemma is easy to see with a little vision: instead of sampling states randomly, only sample the 'important' states with relatively large Boltzmann factors. This is known as importance sampling and is a key principle of Monte Carlo simulations. In a sense, this is obvious: it is well-known that in statistical systems generally only a *very* small proportion of states actually matter, due to the exponential nature of the Boltzmann factor. Physically, systems will only ever inhabit a few of their possible configurations, so it is entirely reasonable to simulate such systems by sampling only this very small but actually important set of states.

In fact, looking at Eqn. 4.1, we see that a clever thing to do would be to try to select states with a probability proportional to their Boltzmann factors, $p_\mu = Z^{-1}e^{-\beta E_\mu}$, where $Z = \sum_\mu e^{-\beta E_\mu}$ is the system's partition function. If this is done then the estimator for observable Q becomes

$$Q_M = \frac{\sum_i^M Q_{\mu_i} {p_{\mu_i}}^{-1} e^{-\beta E_\mu i}}{\sum_i^M {p_{\mu_i}}^{-1} e^{-\beta E_{\mu_i}}} = \frac{\sum_i^M Q_{\mu_i} Z}{\sum_i^M Z} = \frac{1}{M}\sum_i^M Q_{\mu_i}. \tag{4.2}$$

The estimator Q_M is an approximation to the expectation value $\langle Q \rangle$ based on sampling M states. Clearly $Q_M \to \langle Q \rangle$ as M increases.

The question is how to reproduce the Boltzmann distribution – in effect, to find out which states actually do matter – without determining the whole partition function Z, which would require simulating all the states to calculate their Boltzmann factors, a plan we have already considered and abandoned. It turns out that by requiring a stochastic process to obey a simple set of conditions, the Boltzmann distribution can be rapidly and faithfully reproduced. Processes that obey these conditions are known as Markov processes (after Russian mathematician Andrey Markov, 1856–1922), of which discrete-time Markov chains are the simplest and most common type.

4.3 Markov Chains

A Markov chain is a process in which the probability $P(\mu \rightarrow \nu)$ of making a transition from state μ to state ν depends only on those two states (hence the idea of a 'chain' of states). In particular, $P(\mu \rightarrow \nu)$ must therefore remain constant over time. Since each transition has to end up in some state, $\sum_\nu P(\mu \rightarrow \nu) = 1$. In addition, there can be a nonzero probability of not altering the state, $P(\mu \rightarrow \mu) \neq 0$. Clearly the basic random walk discussed above satisfies this condition, as does the nonreversal random walk (if its 'states' are defined to include the previous as well as the current site) and self-avoiding random walk (whose 'states' need to include information about the entire history of the walk).

In addition to the requirement that $P(\mu \rightarrow \nu)$ depend only on μ and ν, two further conditions are required to guarantee that the Markov chain reproduces the Boltzmann distribution.

First, it must be possible to reach any state from any other state through a finite number of transitions. This condition is known as **ergodicity** and clearly must be satisfied by any attempt to reproduce the Boltzmann distribution: Boltzmann factors, due to their exponential dependence on the energy of the system, may be very small but are always nonzero. It is theoretically possible to reach all physical states in the original system, so this must also be the case in the simulation hoping to model that system.

Note that ergodicity does not require a nonzero transition probability from any particular state to *all* other states. In practice, the vast majority of transition probabilities are set to zero by any efficient Monte Carlo simulation. In a d-dimensional basic random walk, for instance, all but the $2d$ transition probabilities for taking a step to a neighboring site are zero. However, it is easily possible to get from any one site to any other through the appropriate *series* of steps. Thus the basic random walk fulfills the requirement of ergodicity (as does the nonreversal random walk, but not the self-avoiding random walk).

The final condition, **detailed balance**, is both the most subtle and the most important, since it actually guarantees that the Boltzmann distribution – as opposed to any other probability distribution – will be the one produced by the simulation after it has come to equilibrium. If the system is in equilibrium, then this means that the rate at which the system transitions into any particular state must equal the rate at which it transitions out of that state. Mathematically,

$$\sum_\nu p_\mu P(\mu \rightarrow \nu) = \sum_\nu p_\nu P(\nu \rightarrow \mu). \tag{4.3}$$

Eqn. 4.3 is the condition of balance.[4.2] However, this condition is not quite rigid enough to guarantee a static equilibrium. Dynamic equilibrium, in which the probability distribution circles through a limit cycle, is also possible. Thus even though the system

[4.2] Since $\sum_\nu P(\mu \rightarrow \nu) = 1$, Eqn. 4.3 can simplified slightly to $p_\mu = \sum_\nu p_\nu P(\nu \rightarrow \mu)$.

obeys Eqn. 4.3, it need not necessarily reach a stable, static probability distribution and thus may not reproduce the Boltzmann distribution.

Instead a slightly more stringent condition is required to eliminate the possibility of limit cycles. This is the condition of detailed balance, which requires that the rate at which the system transitions into any particular state μ from state ν must equal the rate at which it transitions out of μ and back to ν. That is,

$$p_\mu P(\mu \to \nu) = p_\nu P(\nu \to \mu), \tag{4.4}$$

which clearly will satisfy Eqn. 4.3. If detailed balance is satisfied, the ratio of transition probabilities P equals the ratio of the probabilities p of actually being in any particular state, which we require be set by the Boltzmann distribution:

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \frac{p_\nu}{p_\mu} = e^{-\beta(E_\nu - E_\mu)} \tag{4.5}$$

Now, our goal is to make our importance sampling as efficient as possible. To assist in this effort, it is useful to break up the transition probability $P(\mu \to \nu)$ into the product of a 'generation probability' $g(\mu \to \nu)$ of producing any particular state ν from the current state μ and an 'acceptance probability' $A(\mu \to \nu)$ of adopting the new state after it has been generated.

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \frac{g(\mu \to \nu)A(\mu \to \nu)}{g(\nu \to \mu)A(\nu \to \mu)} \tag{4.6}$$

We want to avoid wasting computing time by generating and then rejecting large numbers of potential new states. Our goal should be to make the acceptance probabilities A as large as possible – ideally unity. We can attempt this by putting as much information as possible about $P(\mu \to \nu)$ into $g(\mu \to \nu)$, and also by appropriately scaling the As – since it is their ratio that matters in Eqn. 4.6, we can set the larger of the two to one and scale the other so that it satisfies Eqn. 4.5 for the particular generation probabilities used.

In the next chapter, we will give concrete examples of these issues by exploring some popular Markov chain Monte Carlo algorithms – in particular the well-known Metropolis algorithm and the Wolff cluster algorithm – in the context of the Ising model, a simple model of a ferromagnet.

Chapter 5

The Ising Model and Monte Carlo Algorithms

The Ising model is a simple and well-known statistical system, introduced by Ernst Ising as a model of ferromagnetic materials in 1925 [24] and studied extensively since then.[5.1] Ising defined the model and analytically solved the one-dimensional case in his 1924 doctoral thesis; the two-dimensional case was solved twenty years later by Lars Onsager [41]. Despite the model's simplicity, it was recently proven to be NP-complete in higher ($d \geq 3$) dimensions – see Cipra [14] for more details. No analytic solution has yet been found and it may not be possible to find one.

Because the Ising model has been the subject of so much study, there exists a large body of knowledge (both theoretical and computational) against which simulations can be checked. This makes the model an attractive one to study in order to gain some experience with Monte Carlo simulations. In addition, the Ising model is also related to the statistical formulation of the ϕ^4 quantum field theory. The Ising model and ϕ^4 theory are in the same universality class (see De *et al.* [16]) and the Ising model corresponds to the infinite-coupling limit of the ϕ^4 theory.

In this chapter we will introduce the Ising model and discuss how to simulate it on the lattice using Monte Carlo algorithms such as the Metropolis algorithm and the Wolff cluster algorithm.

5.1 The Ising Model

The Ising model pictures a magnet as a lattice of 'spins', each of which can take one of the values ± 1 (which can be thought of as up-pointing or down-pointing magnetic

[5.1] Ising's initial work was actually wrong, but his errors were quickly corrected in the ensuing flurry of study: over 12,000 papers involving the Ising model were published between 1969 and 1997 alone.

dipoles of unit magnitude). The Hamiltonian for the model is

$$H = -\sum_{\langle ij \rangle} J_{ij} s_i s_j - \sum_i B_i s_i, \tag{5.1}$$

where $\sum_{\langle ij \rangle}$ indicates a summation over all pairs of neighboring spins, J_{ij} is the strength of the interaction between those neighboring spins, and B is an external magnetic field that may vary from site to site. Since the model is being used as an introduction to Monte Carlo simulations, we can gain some traction by making J and B constant throughout the lattice. The simplest case is to set $B = 0$ and $J = 1$. For $B = 0$ and J constant, the Hamiltonian becomes

$$H = -J \sum_{\langle ij \rangle} s_i s_j. \tag{5.2}$$

If $J > 0$, the model describes a ferromagnet, in which it is energetically favorable for the spins to align (i.e., all have the same value). $J < 0$ models an antiferromagnet, in which antiparallel spins are favored.

At temperature T the Ising model has a partition function

$$Z = \sum_\nu e^{-\beta H_\nu}, \tag{5.3}$$

where $\beta = (kT)^{-1} = T^{-1}$ since we work in energy units with $k = 1$, and the sum is taken over all possible configurations ν of the lattice. As a thermodynamic system, the Ising model is governed by Boltzmann statistics, which decree that the probability $P(\mu)$ that the system is in a state μ with energy H_μ is

$$P(\mu) = \frac{e^{-\beta H_\mu}}{\sum_\nu e^{-\beta H_\nu}} = \frac{1}{Z} e^{-\beta H_\mu}, \tag{5.4}$$

At high temperatures, β is small, so the lattice can be in many different states with roughly equal probability. In this situation, considerations of entropy tell us that the lattice will tend to be in a disordered state with spins pointing in different directions almost randomly, since the multiplicity of such states is greater than that of ordered states in which all the spins are aligned. However, we saw above that the ordered states are actually energetically favored, so at low temperatures where β is large, ordered or nearly-ordered states will be the only states the lattice can inhabit with significant probability.

Observables of interest in the Ising model include the energy of the lattice of spins (given by Eqn. 5.2), the magnetization of the lattice (that is, the average spin across it), and the specific heat and susceptibility, quantities proportional to the variances of the energy and magnetization, respectively. We will typically deal with the volume-average energy, magnetization, specific heat and susceptibility per lattice site (e, m, c and χ) as opposed to their total values over the entire N-site lattice (E, M, C and X, where $E = Ne$, etc.). Expressions for the energy and magnetization are very simple,

$$\langle e \rangle = \frac{\left\langle -\sum_{\langle ij \rangle} s_i s_j \right\rangle}{N} \qquad\qquad \langle m \rangle = \frac{\langle \sum_i s_i \rangle}{N}, \tag{5.5}$$

and those for specific heat and susceptibility take the expected form of variances:

$$c = N\beta^2 \left(\langle e^2 \rangle - \langle e \rangle^2 \right) \qquad \chi = N\beta \left(\langle m^2 \rangle - \langle m \rangle^2 \right). \tag{5.6}$$

Measurements of these four observables on a square 32^2 lattice over a range of temperatures are shown in Figs. 5.1 through 5.4. These figures confirm our above discussion about behavior of the Ising model at various temperatures. At low temperatures we see that the energy is low and magnetization high, indicating that the system is in an ordered state; at high temperatures the energy is relatively high and magnetization is near zero, signifying that the system is in a disordered state. A profitable approach to studying such behavior is the concept of phases and phase transitions.

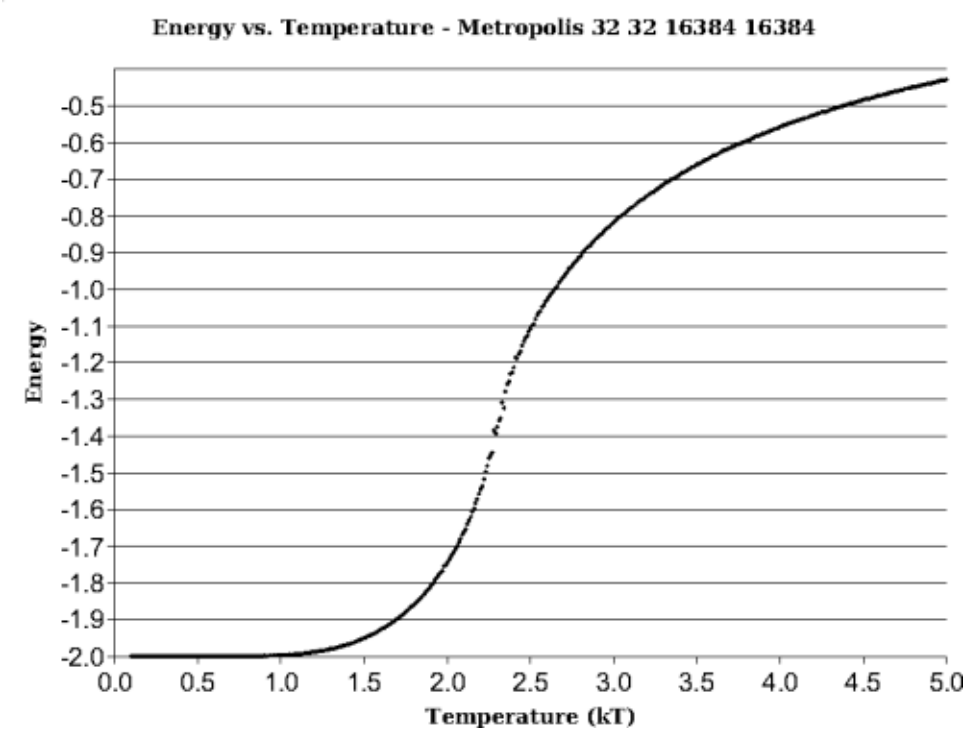

Figure 5.1: Energy vs. temperature for Ising model simulations on a 32^2 lattice.

5.2 Phases, Phase Transitions, and Spontaneous Symmetry Breaking

We can describe Figs. 5.1 through 5.4 by saying that the Ising model possesses two phases – a disordered high-temperature phase characterized by high energy and low magnetization along with an ordered low-temperature phase characterized by low energy and high magnetization. The disordered phase is known as the symmetric phase, since it possesses a simple up-down symmetry. That is, an Ising system in the symmetric phase looks the same if all the spins are flipped to point in the opposite direction. This is illustrated in Fig. 5.5, in which black and white pixels represent spins aligned in opposite directions. The system on the right in Fig. 5.5 was created by flipping every single spin

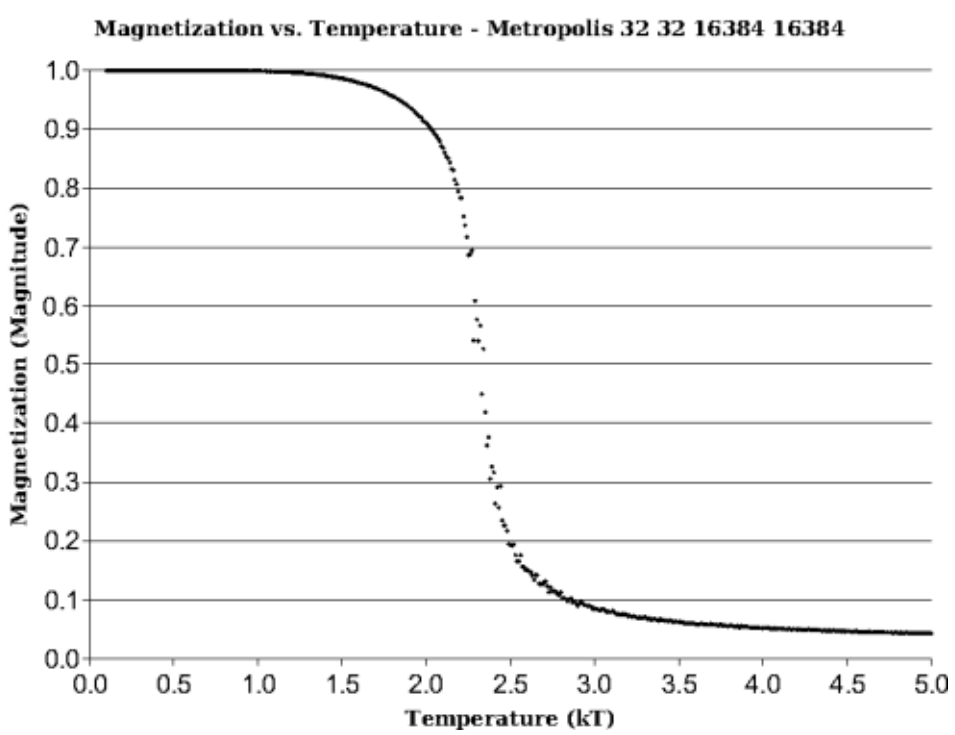

Figure 5.2: Magnetization vs. temperature for Ising model simulations on a 32^2 lattice

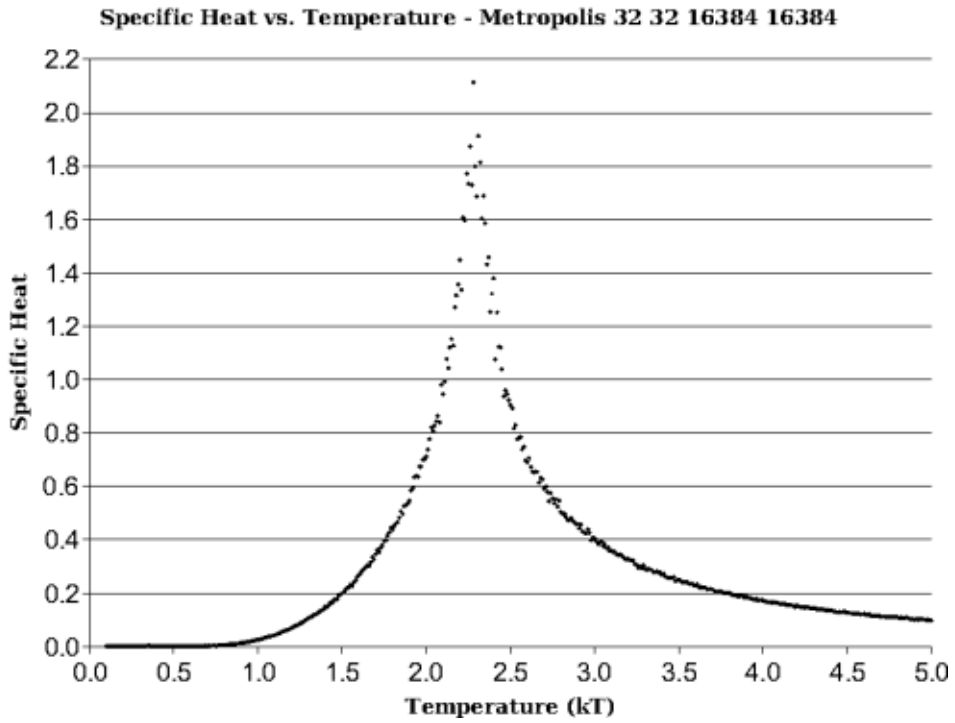

Figure 5.3: Specific heat vs. temperature for Ising model simulations on a 32^2 lattice

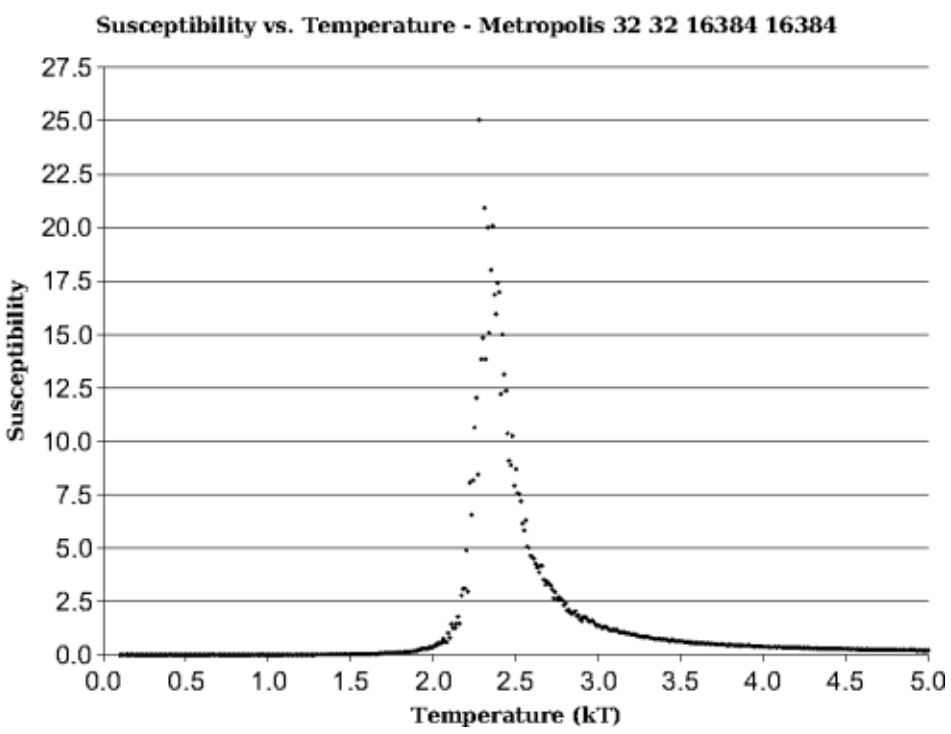

Figure 5.4: Susceptibility vs. temperature for Ising model simulations on a 32^2 lattice

in the system on the left (up $\rightarrow$ down and *vice versa*); the two are in that sense opposites but nevertheless look much alike.

Figure 5.5: Sample Ising model states in the symmetric phase on a 128^2 lattice. In this and following figures, black and white pixels represent spins aligned in opposite directions.

In contrast, at low temperatures the ordered phase no longer has up-down symmetry. The two systems in Fig. 5.6 are mirror images, just like those in Fig. 5.5; however, it is now very easy to tell the two states apart. The ordered phase that exists at low temperatures is therefore known as the broken symmetry (or simply "broken") phase, since the alignment of the spins along a single direction "breaks" the high-temperature symmetry.

The Ising model can also exist temporarily in metastable states such as those illustrated in Fig. 5.7. It is easy to see that although the energy of horizontal-band

Figure 5.6: Sample Ising model states in the broken phase on a 128^2 lattice. (Frame added to the left-hand system for clarity.)

metastable state is higher than that of the completely-aligned ground state, if any single spin were to reverse direction the energy of the system would increase since the spin would now have three or four oppositely-aligned neighbors. Similarly, the energy of the diagonal-stripe metastable state is higher than that of the ground state, but the energy cannot be lowered by flipping a single spin. Such metastable states can only exist until thermal fluctuations cause them to tunnel through the energy barrier and fall into the ground state. Since they are thus only temporary, though perhaps long-lasting, they are not technically considered phases.

Figure 5.7: Sample metastable states of the Ising model on a 128^2 lattice. (Frames added for clarity.)

Since the Ising model inhabits a symmetric phase at high temperatures and a broken phase at low temperatures, there clearly must be some sort of transition from one phase to the other as the temperature changes. This can clearly been seen from the graphs of Ising model observables over temperature, especially Figs. 5.3 and 5.4, which show peaks in the specific heat and susceptibility around the region where the transition occurs. This transition becomes sharper if the system is simulated on larger lattices, by which we mean that the peaks become narrower and their maxima increase. This is illustrated for the susceptibility of ϕ^4 theory (Chapter 6) by Figs. 5.8 and 5.9; since ϕ^4 theory and the Ising model are in the same universality class, the Ising model's behavior will be the same. We have presented this data in two charts since the change in the heights of the peaks from 32^2 to 1024^2 lattices is so dramatic that the peaks for smaller lattices simply look like flat lines in Fig. 5.9.

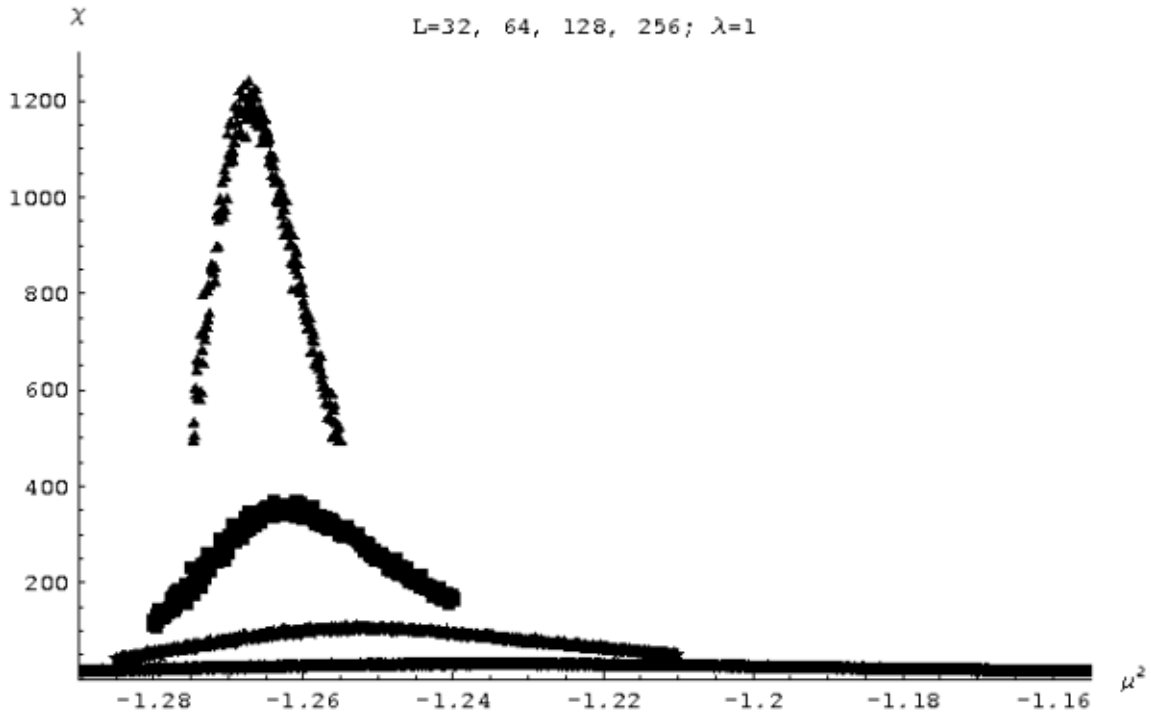

Figure 5.8: Susceptibility vs. μ_l^2 for ϕ^4 theory simulated at $\lambda = 1$ on square lattices of size $L = 32$, 64, 128 and 256

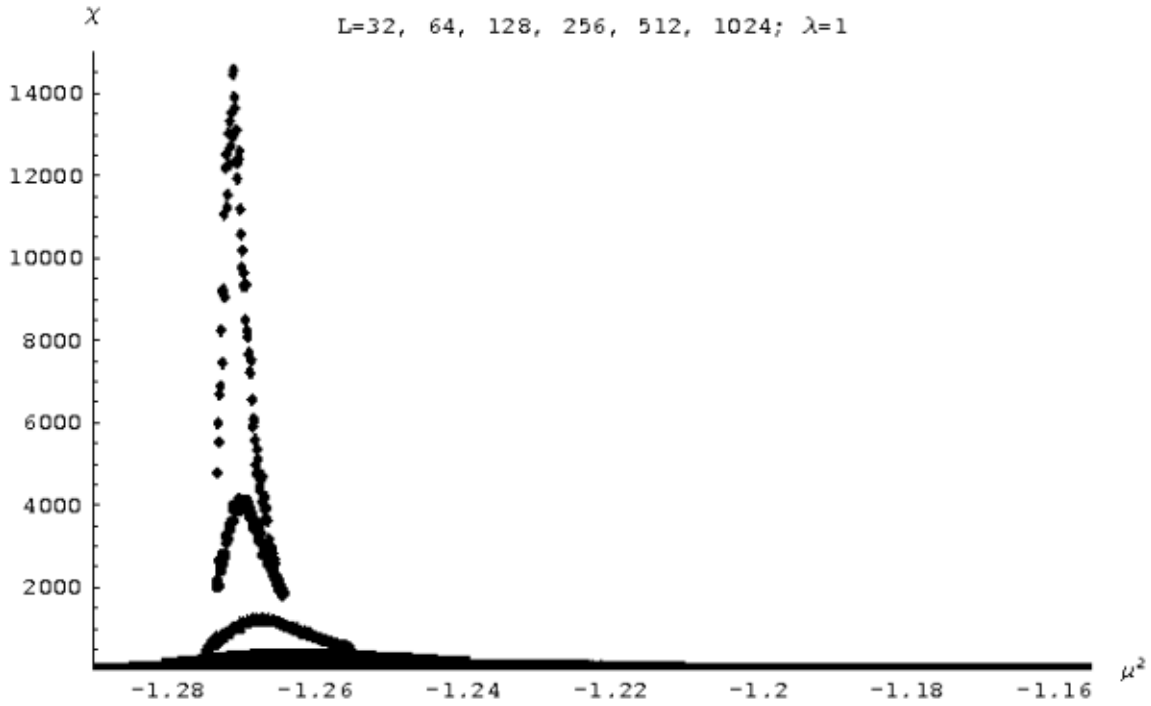

Figure 5.9: Susceptibility vs. μ_l^2 for ϕ^4 theory simulated at $\lambda = 1$ and $L = 32$, 64, 128, 256, 512 and 1024

In the infinite-volume limit, the specific heat and susceptibility of the Ising model actually diverge at a point known as the critical temperature. Since the two-dimensional Ising model has been solved analytically, we know that its critical temperature is exactly

$$T_c = \frac{2J}{\log\left(1+\sqrt{2}\right)} \approx 2.269J \rightarrow \beta_c \approx \frac{0.4407}{J}. \tag{5.7}$$

We call a transition such as that of the Ising model in which the energy and magnetization are continuous but their variances diverge "second-order"; in a "first-order" transition the magnetization itself is discontinuous. It is also possible to define higher-order phase transitions, but since this classification scheme runs into some difficulties, all phase transitions that are not first-order are often called simply "continuous".

Researchers are typically most interested in the critical regions and phase transitions of thermodynamics systems. In part this is because there's simply more going on in those regions of the phase space: at low temperatures the spins are all aligned into a boring ordered state, and at high temperatures they're more or less randomly distributed, but in between the system somehow needs to get from one uninteresting extreme to the other, a potentially interesting process. Even more important, however, is the fact that the quantities of interest of the Ising model and other systems – magnetization, specific heat and susceptibility among others – are characterized by 'critical exponents' near the critical point. Values for the critical exponents for the Ising model are

$$m \sim |t|^{\beta} \qquad \beta = \frac{1}{8} \tag{5.8}$$

$$c \sim |t|^{\alpha} \qquad \alpha = 0 \tag{5.9}$$

$$\chi \sim |t|^{-\gamma} \qquad \gamma = \frac{7}{4}, \tag{5.10}$$

where t is the reduced temperature, $t = \frac{T-T_c}{T_c}$. ($\alpha = 0$ simply means the specific heat diverges logarithmically, not as a power of $|t|$.)

The critical exponents are important because they are believed to be properties of the most basic features of the Ising model, independent of such details as the value of J or the geometry of the lattice. Even more remarkably, different system often have the same critical exponents, a property known as universality. Systems with the same critical exponents are said to be in the same universality class, and behave similarly in their critical regions. This is the case for the two-dimensional Ising model and ϕ^4 theory, the focus of our work.

We studied the Ising model and its phase transition using computer programs to simulate it on rectangular lattices, which can be found below in the Section C.2. These programs use two different Monte Carlo algorithms, the Metropolis algorithm and the Wolff cluster algorithm, to which we now turn.

5.3 Metropolis Algorithm

The Metropolis algorithm, first described by Nicholas Metropolis *et al.* [38] in 1953, is one of the simplest, best-known, and widely-used Monte Carlo algorithms in existence. It works by making a random change to the lattice and then accepting the newly generated state with a probability based on the resulting change in the energy of the system. Note that this explicitly breaks up the transition probability into the product of a generation probability and an acceptance probability, $P(\mu \to \nu) = g(\mu \to \nu)A(\mu \to \nu)$, as discussed above in Section 4.3

For simplicity, we will consider only single-site Metropolis, in which only one spin is flipped at a time in order to generate the new state. Although this is the most common approach (and the one used in all our programs), single-site updating is not a necessary feature of the Metropolis algorithm. However, it greatly simplifies the verification of detailed balance, since when the algorithm is performed on an N-site lattice there are only N possible new states than can be (randomly) generated. So for each of the N possible destination states, $g(\mu \to \nu) = 1/N = g(\nu \to \mu)$.

Looking back to Eqn. 4.5, we have

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \frac{g(\mu \to \nu)A(\mu \to \nu)}{g(\nu \to \mu)A(\nu \to \mu)} = \frac{A(\mu \to \nu)}{A(\nu \to \mu)} = e^{-\beta(E_\nu - E_\mu)}. \tag{5.11}$$

The relative transition probabilities are now determined by the acceptance ratios. Our goal is to maximize the acceptance probability in order to keep from repeatedly generating and then rejecting new states. Since the Boltzmann distribution favors states with lower energies, we should always accept any transition that brings the lattice into a lower energy state. Since the acceptance ratio must obey Eqn. 5.11, we see that the probabilities should be given by

$$A(\mu \to \nu) = \begin{cases} 1 & \text{if } E_\nu \le E_\mu, \\ e^{-\beta(E_\nu - E_\mu)} = e^{-\beta\Delta E} & \text{if } E_\nu > E_\mu, \end{cases} \tag{5.12}$$

That is, we always accept any state that doesn't increase the energy of the system, and conditionally accept states that increase the energy with probability $e^{-\beta\Delta E} < 1$.[5.2] Note that the transition probabilities depend only on the current state μ and the newly generated state ν, and that they have been chosen to reproduce the Boltzmann distribution. Eqn. 5.12 defines the Metropolis algorithm, even in cases where multiple spins are flipped at once.

However, using single-site updating makes it easy to verify that the algorithm is ergodic. In order to get from one state to another, it is only necessary to flip, one-by-one, all of the sites at which the two states differ. Since the transition probabilities for each individual change are all greater than zero, so is the total product. In addition, using single-site updating means that (in models such as the Ising model and ϕ^4 theory that do not involve long-distance interactions) ΔE can be calculated very efficiently by considering only the nearest neighbors of the affected site.

[5.2] The alternative $A(\mu \to \nu) = e^{-\frac{1}{2}\beta(E_\nu - E_\mu)}$ may seem logical but is actually far less efficient, as made clear by Newman and Barkema [40, Section 3.1].

Finally, since the condition of detailed balance (Eqn. 4.5) was used to generate Eqn. 5.12, it is almost trivially satisfied, as can be verified by substituting the probabilities back into the detailed balance condition. Thus the Metropolis algorithm is a Markov chain Monte Carlo algorithm, and we can rely on its ability to reproduce the Boltzmann distribution.

5.4 Critical Slowing Down

Although the Metropolis algorithm is widely used and trusted, it has the disadvantage of producing relatively large autocorrelation times (defined above in Section 2.2), especially near the critical point. This is illustrated in Fig. 5.10, which shows how the autocorrelation time τ for ϕ^4 theory (measured in units of five sweeps of the lattice with the Metropolis algorithm) peaks near the critical point. As discussed in Chapter 2, this means that in the critical region we must either run our simulations for longer times or accept larger uncertainties in our measurements. Both of these alternatives are unattractive, especially since the critical region is often the very portion of phase space we're most interested in studying, both to look at phase transitions as well as calculate the critical exponents discussed in Section 5.1. In order to address the problem of critical slowing down, it will be helpful to gain at least a qualitative understanding of its cause.

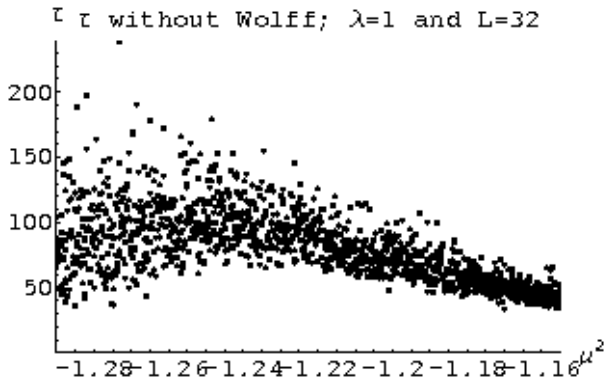

Figure 5.10: Critical slowing down for Metropolis algorithm simulation of the ϕ^4 model (Chapter 6) on a 32^2 lattice at $\lambda = 1$

Consider what must happen as the system cools from the symmetric phase into the broken phase. As the temperature decreases, the spins start to align, at first in small groups that we'll call 'clusters' of correlated spins. These clusters must expand as the temperature decreases, eventually encompassing the whole system and growing to infinite size as the system reaches the critical point. Thus we can define another critical exponent governing the behavior of the autocorrelation length ξ of clusters of aligned spins near the critical point,

$$\xi \sim |t|^{-\nu}. \tag{5.13}$$

On finite lattices of size L, the autocorrelation length cannot be longer than L, and so is cut off at $\xi \sim L$ near the critical point.

The reason we introduce the autocorrelation length is that the Metropolis algorithm has a difficult time dealing with multiple clusters, and the larger the autocorrelation length, the greater the difficulty. This is because the Metropolis algorithm is a local algorithm. Any site fully enclosed within a cluster sees only spins in a particular direction, and so there would be a significant gain in the energy of the system if it were flipped. The Metropolis algorithm will thus tend to leave spins within clusters unflipped, and take action only on boundaries between clusters. Therefore if we're using a local algorithm like the Metropolis algorithm, the degree of correlation between subsequent measurements depends on the size of the clusters (that is, the autocorrelation length), and therefore diverges at the critical point. We can now express the autocorrelation time for the Metropolis algorithm as a power of reduced temperature $|t|$. Since the exponent is specific to the Metropolis algorithm, it is not a critical exponent. Instead we note the dependence of the autocorrelation time on the autocorrelation length and write,

$$\tau \sim |t|^{-z\nu} \sim \xi^z, \tag{5.14}$$

where ν is the critical exponent of the autocorrelation length and z is called the 'dynamic' exponent. All the algorithm-specific properties of the divergence of τ have been placed into z.

Now the most pernicious aspect of critical slowing down is clear: since we have $\xi \sim L$ for finite lattices,

$$\tau \sim L^z. \tag{5.15}$$

Unless z is small, the autocorrelation time will increase intolerably as the lattice gets larger. For the Metropolis algorithm applied to the Ising model, z has been measured to be $z = 2.167(1)$, so the accuracy of the algorithm will decrease dramatically as lattice size increases. This is doubly unfortunate, since larger lattices not only take more computing resources to simulate in the first place, but are of special interest since they more accurately model the continuum behavior of the system.

Fortunately, there are other algorithms that specifically address and reduce the problem of critical slowing down. In our simulations, we used the Wolff cluster algorithm, which has $z = 0.25(1)$, an order of magnitude smaller than that of the Ising model. Let's see how it works.

5.5 Wolff Cluster Algorithm

Since we have seen that critical slowing down is an effect of the division of the lattice into clusters near the critical point, we suspect that it could be addressed by somehow incorporating these clusters into the algorithm used to generate the Boltzmann distribution. A class of Monte Carlo algorithms known as cluster algorithms takes just this approach. The most popular cluster algorithm at the moment is the Wolff cluster algorithm, which we used extensively in our simulations and will explain in this section.

Ulli Wolff introduced the algorithm that bears his name in 1989 [69], as an improved version of the earlier Swendsen-Wang cluster algorithm [59]. The Swendsen-Wang

algorithm partitions the entire lattice into a number of disjoint (non-overlapping) clusters, each cluster containing only spins aligned in a particular direction. Each of these clusters is then individually flipped with probability $\frac{1}{2}$. The Wolff algorithm, in contrast, creates only a single cluster of aligned spins that generally does not include the whole lattice.[5.3]

The algorithm proceeds by randomly selecting a single initial 'seed' spin that forms the kernel of the cluster, then probabilistically adding to the cluster all of the seed spin's neighbors that have the same spin, recursively considering all of *their* neighbors, and so on until all the properly-aligned neighboring sites have been tested. The cluster is then flipped to create the new state; in contrast to the Metropolis algorithm, this new state is always accepted. The Wolff cluster algorithm thus moves the probabilistic aspects of the Monte Carlo simulation from the acceptance probability to the selection probability, specifically, to the act of adding sites to the cluster. Let us now determine how this act must be performed in order to guarantee that the Wolff cluster algorithm is a proper Markov chain Monte Carlo method that can be trusted to reproduce the Boltzmann distribution.

Consider again the condition of detailed balance, Eqn. 4.5. Since $A(\mu \to \nu) = 1 = A(\nu \to \mu)$, we have

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \frac{g(\mu \to \nu)A(\mu \to \nu)}{g(\nu \to \mu)A(\nu \to \mu)} = \frac{g(\mu \to \nu)}{g(\nu \to \mu)} = e^{-\beta(E_\nu - E_\mu)}. \tag{5.16}$$

Let's think about what happens when a cluster is flipped. All the spins in the cluster will be aligned both before and after it is flipped, so the only change in energy will come from the sites neighboring the cluster. The n neighboring spins that are currently aligned with the cluster will point opposite to it after it is flipped, while the m neighboring spins that are currently pointing opposite to the cluster will be aligned with it.[5.4] Now, each of the n neighboring spins with the same alignment as the cluster could have been added to the cluster when it was being formed, as described above. Suppose each independently had the same probability P_{add} of being added to the cluster, $1 - P_{add}$ of being left out. The total probability of these n sites being left out is then clearly $(1 - P_{add})^n$. But since the selection probability is entirely dependent on which sites are added to the cluster and which are left out, it must be proportional to this value,

$$g(\mu \to \nu) \propto (1 - P_{add})^n .$$

Similarly, $g(\nu \to \mu) \propto (1 - P_{add})^m$, since going in the 'reverse direction' from ν to μ will simply reverse the definitions of m and n. Therefore

$$\frac{g(\mu \to \nu)}{g(\nu \to \mu)} = (1 - P_{add})^{n-m} = e^{-\beta(E_\nu - E_\mu)}.$$

Finally, we can express $\Delta E = E_\nu - E_\mu$ in terms of n and m. For the Ising model with constant coupling J and no external field, the energy will rise by $2J$ for each of the n

[5.3] Adding every site in the lattice to the cluster is possible only in the broken phase, when all the spins are aligned.

[5.4] When counting m and n, we need to count the same site multiple times if it has multiple neighbors in the cluster.

sites that used to be aligned with the cluster but no longer are, while it will fall by the same amount for each of the other m that are now aligned with the cluster in the new state. So $\Delta E = 2J(n-m)$ and we have

$$(1 - P_{add})^{n-m} = \left(e^{-2\beta J}\right)^{n-m} \rightarrow P_{add} = 1 - e^{-2\beta J} \tag{5.17}$$

So just using this simple, constant probability $P_{add} = 1 - e^{-2\beta J}$ to determine which (appropriately-aligned) sites are added to the cluster guarantees that the Wolff cluster algorithm will satisfy detailed balance. It is also easy to see that it will satisfy ergodicity, using the simple trick of noting that for any finite βJ there is a nonzero chance of creating a single-site cluster. Thus there is also a nonzero probability of forming and flipping such a single-site cluster at each site where any two potential states differ, making it possible to reach any state from any other in a finite number of steps. Since the Wolff cluster algorithm clearly does not rely on any memory of previous states, it is a *bona fide* Markov chain Monte Carlo algorithm that will reproduce the Boltzmann distribution, just like the better-known Metropolis algorithm.

But does the Wolff cluster algorithm help with the critical slowing down problem, the reason we started thinking about it in the first place? Clearly we wouldn't have bothered to write all this if it didn't. A large number of studies have verified that cluster algorithms greatly reduce critical slowing down when compared to local algorithms such as the Metropolis algorithm. Of particular interest for our later work are papers by Brower and Tamayo [9], De *et al.* [16], and Loinaz and Willey [32], which use mixtures of the Metropolis and Wolff algorithms (actually, Brower and Tamayo use the Swendsen-Wang cluster algorithm, but the general idea is the same). Specifically, they sweep the lattice several (generally 5-10) times with the Metropolis algorithm before performing a single Wolff cluster flip to deal with any clusters that may have formed.[5.5]

This approach is especially useful for theories such as ϕ^4 theory that have more complicated structures than the simple spin-up/spin-down on the Ising model on which the Wolff cluster algorithm operates. It is often possible to embed Wolff clusters into these system, but typically simply performing the Wolff cluster algorithm on these embedded clusters does not satisfy ergodicity. Sweeping the lattice with the Metropolis algorithm in between Wolff cluster flips is a way around this problem for ϕ^4 theory, as we will explain in more detail in Section 7.5, after introducing ϕ^4 theory itself.

We ourselves were able to verify that mixing the Wolff cluster algorithm and Metropolis algorithm in this fashion reduces critical slowing down. We performed our small-lattice ($L \leq 128$) ϕ^4 simulations twice, once using just the Metropolis algorithm and once using this Metropolis/Wolff mixture. Although our motivation for doing so was to verify that the mixture gives the same results as the tried-and-true Metropolis algorithm by itself (it does – see Section 7.5), our data could also be used to explore the effects of the Wolff cluster algorithm on critical slowing down and autocorrelation times.

Fig. 5.11 shows that although the autocorrelation time τ (now measured in units of five sweeps of the lattice with the Metropolis algorithm followed by one Wolff cluster

[5.5] It might be an interesting exercise to see how critical slowing down and autocorrelation times depend on the ratio of Metropolis sweeps to Wolff cluster flips. We did not perform such an analysis.

flip) still peaks around the critical point, it is roughly an order of magnitude less than it was when the same simulation was performed using only the Metropolis algorithm (Fig. 5.10). This order of magnitude reduction in τ is typical for the two-dimensional ϕ^4 model on these small lattices – given the relation between lattice size and autocorrelation length, Eqn. 5.15, we suspect the benefits are even greater for simulations using larger lattices. These results also provide some empirical support for our earlier explanation of critical slowing down through clusters; by operating in terms of clusters, the Wolff cluster algorithm and other similar cluster algorithms are better able to address this issue and reduce critical slowing down.

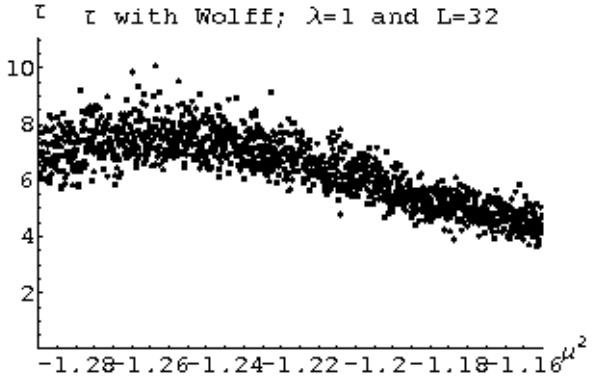

Figure 5.11: Critical slowing down for mixed Metropolis/Wolff simulation of the ϕ^4 model (Chapter 6) on a 32^2 lattice at $\lambda = 1$

We should emphasize that using cluster algorithms doesn't eliminate critical slowing down entirely – the autocorrelation time still peaks near the critical point in Fig. 5.11. However, they do make the problem more manageable and the simulations much more efficient – at least around the critical point, the region in which we're generally most interested (and all that's shown in Figs. 5.10 and 5.11). A little thought will show that the Wolff cluster algorithm is often very inefficient in other regimes, particularly very high and very low temperatures. At very high temperatures, few spins will be aligned, so there will be few sizable clusters that can be formed in the first place. Moreover, as the temperature increases, β decreases, so $P_{add} = 1 - e^{-2\beta J} \to 0$. This means that few if any sites will be added to the cluster, and the "cluster flip" will consist of the useless flipping back and forth of randomly chosen sites. The Metropolis algorithm performs much more efficiently in this regime.

The Metropolis algorithm is also much more efficient at low temperatures. At high β, all the spins tend to be aligned and $P_{add} \to 1$, so many sites will be added to the cluster. Often at very low temperatures all spins are aligned (the lattice is 'frozen') and all are added to the cluster. So in this regime, the Wolff cluster algorithm simply flips the entire lattice, accomplishing nothing. The Metropolis algorithm will not have much effect in this regime either, but at least it won't visit every site on the lattice during each iteration. The moral is that cluster algorithms such as the Wolff cluster algorithm are most useful when there are several sizable clusters for them to work with, a state of affairs most common around the critical point.

5.6 Related Statistical Systems

The Ising model, though by far the best-known statistical model of ferromagnetic phase transitions (and other analogous phenomena), is far from the only one. There are several generalizations to the Ising model that allow the 'spins' to move beyond simple spin-up/spin-down and take on more values, or even vary continuously. In the following subsections we will briefly introduce some of the most important of them, the Potts model and $O(n)$ σ models where $n = 2$ (the XY model) and $n = 3$ (the Heisenberg model).[5.6]

More extensive discussions of all three models (and more besides) can be found in Berg [6] and Newman and Barkema [40].

5.6.1 Potts Model

The q-state Potts model is a simple generalization of the Ising model, introduced in 1952 by R. B. Potts [46]. Like the Ising model, it consists of sites containing 'spins' of unit length that interact only with their nearest neighbors. However, instead of pointing either up or down, each Potts site can be in one of q states $1, ..., q$. The general Hamiltonian for the Potts model on a 2-dimensional lattice is

$$H = -\sum_{\langle ij \rangle} J_{ij}\delta(q_i, q_j) - \sum_i B_i s_i. \tag{5.18}$$

$\sum_{\langle ij \rangle}$, J_{ij} and B_i are all the same as in the Ising model discussed above in Section 5.1. Note that in contrast to the Ising model, the δ-function ensures that neighboring spins only directly interact if they are in the same state.

However, it is still possible to reproduce the Ising model from this more general scenario. Again setting $B = 0$ and $J = 1$ to gain some traction, the Hamiltonian becomes

$$H = -J\sum_{\langle ij \rangle} \delta(q_i, q_j) = -\frac{1}{2}J\sum_{\langle ij \rangle} 2\left(\delta(q_i, q_j) - \frac{1}{2}\right) - \sum_{\langle ij \rangle} \frac{1}{2}J. \tag{5.19}$$

In the case $q = 2$, this is equivalent (up to an additive constant) to the Ising model with $J \to \frac{1}{2}J$, which accounts for the Ising interaction between oppositely-aligned spins. Like the Ising model, the Potts model has been analytically solved in two dimensions, and has a broken symmetry ground state in which all the spins spontaneously fall into the same q_i. As suggested by Eqn. 5.19, the critical temperature for the $q = 2$ Potts model is simply half that of the Ising model,

$$T_c^{Potts} = \frac{T_c}{2} = \frac{J}{\log\left(1 + \sqrt{2}\right)} \approx 1.135J \to \beta_c \approx \frac{0.8814}{J}.$$

The generalization to the q-state Potts model is (perhaps surprisingly) simple:

$$T_c^{Potts} = \frac{J}{\log\left(1 + \sqrt{q}\right)} \to \beta_c = \frac{\log\left(1 + \sqrt{q}\right)}{J}.$$

[5.6] The $O(1)$ σ model is actually the Ising model itself.

The phase transition is known to be second order for $q \leq 4$ and first order for $q \geq 5$.

The Metropolis and Wolff algorithms can both be applied to the Potts model. The Wolff algorithm now builds its cluster out of sites all having the same value of q_i, and both algorithms need to change the value of $q_{current}$ at the site(s) under consideration into a new, randomly-chosen q_{new}. In general, q_{new} can be the same as $q_{current}$. Though we wrote some $q = 10$ Potts model simulations using the Metropolis and Wolff algorithms early in the course of our work, we then moved on to other topics without obtaining noteworthy results.

5.6.2 XY Model

The Potts model is peculiar in that neighboring sites only directly interact if they are in the same state q_i. This makes the Potts model difficult to describe physically – the noninteracting neighboring sites can't easily be visualized as spins pointing in different directions, except in the $q = 2$ model that reproduces the Ising model. A further generalization known as the XY model produces precisely these kinds of spins, which can point in any direction in a plane; unlike the Ising and Potts models, the XY model is a continuous spin model. Like those models, however, XY spins are all set to unit length, so they can be written $\vec{s} = (s_1, s_2)$, where $s_1^2 + s_2^2 = 1$.

Technically, the XY model is the $O(2)$ σ model, so named because its spins are confined to the unit circle in the two-dimensional plane, which the mathematically-inclined reader will recognize as corresponding to the two-parameter special orthogonal (or rotation) group $SO(2)$, the group of rotations in two dimensions that preserve distance from the origin. It's intuitively obvious that the two-component spins (s_1, s_2) can be specified with a single parameter, namely the angle θ at which the spin is aligned relative to some fixed axis. This observation is a reflection of the well-known mapping between $SO(2)$ and the unitary ('circle') group $U(1)$.

The Hamiltonian for the XY model is similar to that of the Ising model, though now that the spins have multiple components we need to take their dot products instead of simply multiplying them:

$$H = -J \sum_{\langle ij \rangle} \vec{s}_i \cdot \vec{s}_j \tag{5.20}$$

(As we did above, we are setting all external fields B to zero and requiring constant interaction strength J.) The ground state of the system is, like that of the Ising and Potts models, one in which all the spins are aligned. However, now the spins can align in any direction in the plane, giving the system a continuous broken symmetry, rotation around the plane. In quantum field theory, such spontaneous breaking of a continuous symmetry entails the existence of massless particles known as Goldstone bosons – see Ryder [51, Chap. 8].

Since the XY model is a continuous spin model, we need to generalize the partition

function and Boltzmann distribution from the discrete case; we now have

$$Z = \int e^{-\beta E} \rho(E) dE$$
$$p(E)dE = \frac{\int e^{-\beta E} \rho(E) dE}{Z}$$

where $p(E)dE$ is the probability of finding the system in a state with energy between E and $E + dE$ and $\rho(E)$ is the density of states, defined such that $p(E)dE$ is the number of states in the interval E to $E + dE$. The question of how a system with a continuous spectrum of states could have a particular number of states in an energy interval is difficult to address classically.

5.6.3 Heisenberg Model

The Heisenberg model, the final statistical model we will introduce, is quite similar to the XY model – a simple extension that nevertheless features some striking new phenomena. The Heisenberg model is the $O(3)$ σ model, which involves three-component normalized vectors $\vec{s} = (s_1, s_2, s_3)$, $s_1^2 + s_2^2 + s_3^2 = 1$. The Hamiltonian for the Heisenberg model is precisely that of the XY model, Eqn. 5.20, with the spins $\vec{s}_i \cdot \vec{s}_j$ now understood to possess three components.

The Heisenberg model is of special interest because it is one of the simplest possible 'nonabelian' statistical systems. Nonabelian theories or models are simply those with nonabelian symmetry groups. The Heisenberg model's symmetry group is $SO(3)$ (which corresponds to $U(2)$ just as $SO(2)$ corresponds to $U(1)$), which is clearly nonabelian – all that signifies is that rotations in three dimensions do not commute, a well-known and easily verifiable fact.

Nonabelian groups are especially important in quantum field theory, since they include gauge theories such as quantum chromodynamics (QCD). Yang-Mills theories are an important class of gauge theories, and were actually the first nonabelian gauge theories to be developed, in the 1950s. One of the most remarkable features of four-dimensional Yang-Mills theory, and the reason we mention it here, is that it possesses strong analogies to the two-dimensional Heisenberg model. Thus the Heisenberg model can serve as a relatively gentle introduction to the world of gauge theory and QCD.

It is possible to simulate continuous spin models such as the XY and Heisenberg models using the Metropolis algorithm and Wolff cluster algorithm, though in forms slightly modified from those introduced above in the context of the Ising model. However, it is generally considerably more efficient to use other algorithms, such as the heat bath algorithm, that are better suited to continuous models. We'll now take a quick look at the heat bath algorithm along with some other noteworthy algorithms.

5.7 Other Noteworthy Algorithms

Just as the Ising model is not the only interesting model we could consider, the Metropolis and Wolff algorithms are not the only methods we could use to simulate them. For example, we have already introduced the Swendsen-Wang cluster algorithm, off of which the Wolff cluster algorithm was based.

In the following subsections, we will briefly explore some other algorithms that are of particular interest. First we will consider in some detail the invaded cluster algorithm, which uses an interesting method to efficiently determine the critical point of the Ising model and related systems. Next we will turn to the heat bath algorithm designed to increase the efficiency of simulations of Potts and σ models in which each spin can take on many potential values. Finally we will briefly introduce multigrid methods and worm algorithms, both of which attempt to reduce critical slowing to a lower level than that attained by standard cluster algorithms, but use quite different approaches.

5.7.1 Invaded Cluster Algorithms

The invaded cluster algorithm, relatively recently developed by Machta and others [17, 35, 36], is the most interesting algorithm that we encountered but did not use in our work. It is applicable to the Ising model and possibly other systems, but works quite differently than both the Metropolis and Wolff algorithms. Instead of reproducing the Boltzmann distribution for a given temperature, the invaded cluster algorithm modifies the temperature in order to directly find the critical point. Should we be interested in determining the critical point (and we are), we can do so much more efficiently by using the invaded cluster algorithm than by manually analyzing observables such as the susceptibility. The invaded cluster algorithm can also be used for other purposes (such as determining critical exponents mentioned above), but we need to keep in mind that the states it generates do not obey Boltzmann statistics.

The invaded cluster algorithm for the Ising model takes advantage of the pattern of cluster formation that characterizes the phase transition. As we noted above, the clusters that form as the temperature is cooled from the symmetric phase to the broken phase reach the size of the lattice at the critical point. We define a "percolating cluster" as a cluster that spans the lattice. That is, its length in at least one direction is equal to the dimension L of the lattice in that direction, or it runs into itself as a result of periodic boundary conditions. Each iteration of the invaded cluster algorithm attempts to create a percolating cluster on the lattice. It then adjusts the temperature of the simulation in order to find the maximum possible temperature at which a percolating cluster forms – clearly this maximum temperature is the critical temperature.

We can accomplish this by assigning a random number between zero and one to every pair of neighboring spins on the lattice that are aligned in the same direction. We can think of each number as sitting on a potential link between the two spins. We sort the random numbers and run through them from smallest to largest, establishing links

between the corresponding pairs of spins. A set of spins which are all linked in this manner can be considered a cluster. We continue making links until a percolating cluster is formed, or all parallel-spin pairs have been linked, whichever comes first. Each cluster is separately flipped with probability $\frac{1}{2}$, just as in the Swendsen-Wang algorithm.

Next we calculate the fraction of all possible links that had to be established in order to form the percolating cluster. This fraction is equal to the minimum possible P_{add} that would have created a percolating cluster under the Swendsen-Wang algorithm (if $P_{add} = 1$ then no percolating cluster could have been created). However, for the Swendsen-Wang algorithm, exactly like the Wolff algorithm, P_{add} has to depend on temperature to satisfy detailed balance. That is,

$$P_{add} = 1 - e^{-2\beta J} = 1 - \exp\left[\frac{-2J}{T}\right] \rightarrow T = \frac{-2J}{\log(1 - P_{add})}. \tag{5.21}$$

So by flipping the clusters as they existed when the percolating cluster was formed, we effectively carried out an iteration of the Swendsen-Wang algorithm at this temperature.

Now, let's see how to extract the critical temperature from that. If the system is in the broken phase, then more sites will be aligned in clusters than would be the case precisely at the critical temperature, so in order to form a percolating cluster P_{add} does not need to be as large as it would have to be at the critical temperature. But this means that the temperature from Eqn. 5.21 is greater than the critical temperature. In the extreme case where every single spin in the cluster is aligned, $P_{add} = \frac{1}{2}$, which corresponds to $T = 2.89 > 2.269 = T_c$. Conversely, if the system is in the symmetric phase, then fewer sites will be aligned than would be the case precisely at the critical temperature, so in order to form a percolating cluster P_{add} needs to be larger than it would have to be at the critical temperature. Eqn. 5.21 then produces a temperature less than the critical temperature. In the extreme case where the lattice is so disordered no percolating cluster can be formed, $P_{add} = 1$ (its maximum), which corresponds to $T = 0 < T_c$.

So we see that when the system has a temperature $T_0 < T_c$, the invaded cluster algorithm will perform Monte Carlo steps at a temperature $T > T_c$, effectively raising the temperature of the system, and *vice versa*. Thus the algorithm drives the cluster toward the critical temperature, which can be measured directly. Empirical studies have shown that the invaded cluster algorithm is very efficient, reaching equilibrium times 20 or more times faster than the Swendsen-Wang algorithm at the critical point.

Although Dukovski, Machta and Chayes [17] have used the invaded cluster algorithm to study the XY model in addition to the Ising model, it has not yet been applied to ϕ^4 theory, the primary focus of this work. If it is possible to apply the invaded cluster algorithm to ϕ^4 theory, our results could have been obtained much more quickly. However, it is not clear that this is possible. We will show in Chapter 7 that when the Wolff cluster algorithm is applied to ϕ^4 theory, P_{add} is not constant over the whole lattice, and, moreover, doesn't even directly depend on μ_0^2, the variable whose critical value we want to determine. However, we have not yet shown that it is impossible to apply the invaded cluster algorithm to ϕ^4 theory. The fact that the algorithm can be made to work for the XY model, which has variable P_{add}, gives us some encouragement.

5.7.2 Heat Bath Algorithm

The heat bath algorithm is a very popular and widely-used algorithm designed for systems such as the Potts model in which a spin can take on more than two values. To motivate the algorithm, we can consider the extreme case of the $q = 100$ Potts model. Suppose we are using the Metropolis method to update a single site whose neighbors all have different q_i. Since the new state for the site is chosen randomly, the site has a 4% chance of ending up in one of the energetically-favored states. If it already in one of those four states, then if the temperature is low it could have a rejection rate of up to 96%. That is, when the Metropolis algorithm is applied to the Potts model it wastes a lot of time selecting and rejecting states, even for more reasonable numbers of states q_i.

The logical way to approach this problem is to bias the generation of states, so that they are no longer all selected with exactly the same probability. Instead we want to select the energetically-favored states with greater probability. Thus the selected state is more likely to be accepted, so less time is wasted rejecting large numbers of energetically-unfavorable updates.

This is precisely what the heat bath algorithm does. For the q-state Potts model it generates states according to the Boltzmann distribution, selecting state $1 \leq n \leq q$ with normalized probability

$$p_n = \frac{e^{-\beta E_n}}{\sum_{m=1}^{q} e^{-\beta E_m}}, \tag{5.22}$$

values drawn from the "heat bath". The new state generated in this manner is always accepted. Note that unlike the Metropolis algorithm these probabilities depend only on the final state, not the current one.

It is clear that the heat bath algorithm satisfies ergodicity, since there is a nonzero probability of each site in the system being assigned each possible state. The proof that the system satisfies detailed balance is also simple. From Eqn. 4.5 we have

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \frac{g(\mu \to \nu)A(\nu \to \mu)}{g(\nu \to \mu)A(\nu \to \mu)} = \frac{g(\mu \to \nu)}{g(\nu \to \mu)} = e^{-\beta(E_\nu - E_\mu)},$$

since $A(\mu \to \nu) = 1 = A(\nu \to \mu)$. But $g(\mu \to \nu)$ is simply p_n, giving

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \left(\frac{e^{-\beta E_\nu}}{\sum_{\kappa=1}^{q} e^{-\beta E_\kappa}}\right)\left(\frac{\sum_{\kappa=1}^{q} e^{-\beta E_\kappa}}{e^{-\beta E_\mu}}\right) = e^{-\beta(E_\nu - E_\mu)},$$

as required.

Though we did not do so ourselves, the heat bath algorithm can be applied to ϕ^4 theory with little difficulty. For instance, Sun [58] uses it to study aspects of ϕ^4 theory related to Bose-Einstein condensation phase transitions.

5.7.3 Multigrid Methods

The class of Monte Carlo algorithms collectively known as multigrid methods are of interest because they reduce critical slowing down to an even greater extent than do cluster algorithms. For example, the simplest multigrid method, which we consider below, has a dynamic exponent of $z \approx 0.2$ for the two-dimensional Ising model, slightly lower than that of the Wolff cluster algorithm.

However, multigrid methods are considerably more complicated to implement than cluster algorithms (to say nothing of proving ergodicity and detailed balance). Since their dynamic exponents are only slightly smaller than that of the Wolff cluster algorithm, multigrid methods will only present an appreciable advantage on very large lattices. As a result of this fact and their overall complexity, multigrid methods have not been widely used, though Sun [58] applies them to aspects of ϕ^4 theory related to Bose-Einstein condensation phase transitions.

We'll briefly outline the simplest multigrid method for the two-dimensional Ising model, which is presented in more detail by Newman and Barkema [40, Chap. 4]. We pick a spin on the lattice and look at all of its neighbors in turn. If a pair of spins are pointing in opposite directions, we erase the Ising interaction between them, so that their relative orientations can change with no effect on the energy. Otherwise the two spins are pointing in the same direction, in which case they are frozen together with the same probability $P_{add} = 1 - e^{-2\beta J}$ used by the Wolff and Swendsen-Wang cluster algorithms. Spins that are frozen together must remain aligned. Those spins that are not frozen but whose interactions have not been erased interact with the normal Ising interaction.

We repeat the process for all the other spins in the lattice, with the complication that if two spins are frozen, we no longer look at their neighbors – they retain the Ising interaction regardless of what direction they're pointing in, unless we have already erased the interaction in a previous step. Similarly, if two parallel spins were not frozen when the second was considered as a neighbor of the first, they must retain the Ising interaction; they cannot be locked together if the first is ever considered as a neighbor of the second.

After we have determined the interaction between every pair of spins on the lattice – whether it is erased, frozen, or retained as the standard Ising interaction – we have effectively divided the lattice into clusters consisting of either one or two spins. We then treat each cluster as an individual spin and perform a few sweeps of the lattice with the Metropolis algorithm.

Next we repeat the whole process, treating the clusters as spins and joining them together into larger clusters of up to four of the original spins, using the same rules as given above. We perform a few more Metropolis sweeps (again with the new clusters treated as individual spins). This process of merging spins and sweeping the resulting lattice is repeated until the clusters reach the size of the lattice. Then we reverse the process by taking the composite clusters apart into the spins (really, lower-level clusters) which made them and sweeping the lattice with the Metropolis algorithm, until all clusters have been dismantled. Alternately, iterations of constructing and dismantling clusters could be intermixed. Either way, the algorithm has the effect of flipping blocks

of spins of all sizes, from single spins to the size of the whole lattice.

As might be expected, the proof that this algorithm and other more complicated multigrid methods obey detailed balance and ergodicity is nontrivial. We will not include it here.

5.7.4 Worm Algorithms

Worm algorithms are another recent innovation, developed primarily by Prokof'ev and Svistunov [47, 48]. The goal of worm algorithms is to reduce critical slowing down to the levels obtained by cluster algorithms while remaining essentially local. They accomplish this by working with bond states, in which several bonds are formed between the various sites of the lattice. They consider the configuration space consisting of all bond states which have either an even number of bonds attached to every site or exactly two sites with an odd number of bonds attached. In the first case, all the paths (connected series of bonds) must be closed; in the second case the two special sites form the endpoints of the only open path.

The algorithms then update the bond states by moving the endpoints of the open path to neighboring sites – if there is no open path, then the endpoints can be taken to intersect at any site in any of the closed paths in the state. Note that these are local updates. Using a relatively technical high-temperature series expansion of the partition function and general state, which we will not reproduce here, Prokof'ev and Svistunov are able to show that their algorithms are ergodic and satisfy detailed balance.

Perhaps more importantly, they present results showing $Z \approx 0.25$ for the two-dimensional Ising model. Thus worm algorithms, though analytically complicated, are local-update algorithms that reduce critical slowing down to the same level as the Wolff cluster algorithm. It should also be possible to apply them to ϕ^4 theory. Although Prokof'ev and Svistunov do not explicitly discuss ϕ^4 theory, they do test their algorithms on a number of different systems, including two- and three-dimensional Ising and XY models, as well as the three-dimensional Gaussian model and the $q = 3$ Potts model. Moreover, they also partially apply them to the $|\psi|^4$ model, whose Hamiltonian,

$$-H = \sum_{\langle ij \rangle} \left(\psi_i \psi_j^* \right) + \sum_i \left[\mu |\psi_i|^2 - U |\psi_i|^4 \right], \tag{5.23}$$

very closely resembles that of the discretized ϕ^4 model, Eqn. 7.14.

Chapter 6

ϕ^4 Theory

ϕ^4 theory is one of the simplest possible nonlinear field theories. It describes a scalar field whose self-interactions make possible nonperturbative phenomena such as solitons. Despite this, its phase structure resembles that of the Ising model (they are in fact in the same universality class – see De *et al.* [16]), also exhibiting spontaneous symmetry breaking during a critical transition from a symmetric to a broken phase.

In this chapter we will introduce ϕ^4 theory as a quantum field theory, treating ϕ as an operator related to the creation and annihilation of interacting scalar particles. We will briefly survey relevant topics such as Feynman rules and renormalization, and will present a renormalization calculation in two dimensions. We will assume some qualitative knowledge of the path-integral formulation of nonrelativistic quantum mechanics, an introduction to which can be found in Townsend [62, Chap. 8].

In the next chapter we will show how to transform the ϕ^4 quantum field theory into an equivalent Euclidean statistical system, discretize it, and simulate it on the lattice. We will also wait until the next chapter to discuss the specific calculations we performed and present our results.

We begin our study of ϕ^4 theory by considering some more general aspects of relativistic quantum field theory. We'll briefly discuss the relatively simple Klein-Gordon equation, the relativistic analog of Schrödinger's equation, and show that it only really makes sense if it is held to operate on a scalar field as opposed to the probability amplitude of a single particle. We'll call this field $\phi(x)$, where x is a point in Minkowski space, and show how we can quantize it by treating it as a (Hermitian) quantum-mechanical operator. The resulting theory leads to the interpretation of ϕ as an operator that creates and annihilates scalar particles.

6.1 The Klein-Gordon Equation

The Schrödinger equation for a free particle can be obtained by treating the energy and momentum in the relation $E = \frac{p^2}{2m}$ as operators, $E \to i\frac{\partial}{\partial t}$ and $p \to i\vec{\nabla}$:

$$E = \frac{p^2}{2m} \to i\frac{\partial}{\partial t}|\psi\rangle = -\frac{1}{2m}\vec{\nabla}^2|\psi\rangle.$$

Applying the same procedure to the relativistic generalization $E^2 - p^2 = m^2$ produces the free-particle Klein-Gordon equation,

$$-\frac{\partial^2}{\partial t^2}\phi + \frac{1}{2m}\vec{\nabla}^2\phi = m^2\phi \to \left(\partial^2 + m^2\right)\phi = 0, \tag{6.1}$$

where $\partial^2 = \frac{\partial^2}{\partial t^2} - \vec{\nabla}^2$ and ϕ must be a Lorentz scalar in order for the equation to be Lorentz covariant.

Although the Klein-Gordon equation is a simple and elegant extension of the Schrödinger equation, it is second-order in the time variable, which causes some difficulties. First of all, since it is E^2 and not E itself that appears in Eqn. 6.1, there will be a negative-energy solution for each positive-energy solution. $\exp\left[i(\vec{k}\cdot\vec{x} + \omega t)\right]$ is a simple example with energy $E = -\sqrt{\vec{k}^2 + m^2}$. This presents some obvious interpretive difficulties, but even beyond that, we see that if the particle were to interact with its environment, it would be able to emit an infinite amount of energy by continually transitioning to ever-lower energy levels. Fortunately (for us, if not for the Klein-Gordon equation), this does not occur in nature. If we tried to get around this by redefining the Klein-Gordon equation as the operator form of positive square root of the energy $E = \sqrt{p^2 + m^2}$, we would either have to define the square root of a differential operator or use a series expansion, which wipes out locality. Both options are unattractive and unacceptable.

Perhaps even more seriously, the fact that the Klein-Gordon equation is second-order in the time variable makes it possible for ϕ to have negative probability density when interpreted as a probability amplitude for a single particle. We can define the conserved probability current of the Klein-Gordon equation as the Minkowski-space extension of the nonrelativistic probability current $\vec{j} = -\frac{i}{2m}\left(\psi^*\vec{\nabla}\psi - \psi\vec{\nabla}\psi^*\right)$:

$$j_\mu = -\frac{i}{2m}\left(\phi^*\partial_\mu\phi - \phi\partial_\mu\phi^*\right) \tag{6.2}$$

The first component is the probability density $\rho = -\frac{i}{2m}\left(\phi^*\frac{\partial}{\partial t}\phi - \phi\frac{\partial}{\partial t}\phi^*\right)$. Since the Klein-Gordon equation is second-order in the time variable, both ϕ and $\frac{\partial}{\partial t}\phi$ can be arbitrarily fixed at a particular time so as to produce $\rho < 0$. For instance, the negative-energy solution mentioned above, $\exp\left[i(\vec{k}\cdot\vec{x} + \omega t)\right]$, also results in a negative probability density $\rho = -\frac{1}{m}\sqrt{\vec{k}^2 + m^2}$. This is nonsense, of course, and means that if we were to take the Klein-Gordon equation seriously as a single-particle equation, we would have to abandon the interpretation of ρ as a probability density.

The solution to these problems is to treat ϕ as a field (postponing considerations of the physical meaning of m in this context). A standard result is that the energy of a field ϕ is given by $H = \int \Theta^{00} d^3x$, where $\Theta^{\mu\nu}$ is the energy-momentum tensor, defined as $\Theta^{\mu}_{\nu} = (\partial_{\partial_\mu \phi}\mathcal{L})\partial_\nu \phi - \delta^{\mu}_{\nu}\mathcal{L}$, analogous to the more familiar $H = p_i \dot{q}_i - L$.[6.1] For the Lagrangian[6.2] $\mathcal{L}$, we use

$$\mathcal{L} = \frac{1}{2}\left[(\partial_\mu \phi)\,(\partial^\mu \phi) - m^2\phi^2\right], \tag{6.3}$$

since the Euler-Lagrange equation for ϕ resulting from this Lagrangian is exactly the Klein-Gordon equation. Plugging this into our expressions for Θ and H, we find

$$H = \frac{1}{2}\int \left[(\partial_0 \phi^*)\,(\partial_0 \phi) + \left(\vec{\nabla}\phi^*\right)\left(\vec{\nabla}\phi\right) + m^2\phi^*\phi\right] d^3x, \tag{6.4}$$

which can never be negative.

The above result holds even if ϕ is treated as a classical field. The relationship between ϕ and probability, however, is entirely quantum-mechanical, so in order to address it, we need to quantize ϕ itself, a nontrivial task to which we devote the next subsection.

6.2 The Meaning of ϕ

Suppose we have a scalar field $\phi(x)$ in Minkowski space that obeys the Klein-Gordon equation. We can quantize it by treating $\phi(x)$ as a (Hermitian) quantum-mechanical operator, which can be written as a Fourier transform of momentum-space operators $a(k)$ and $a^\dagger(k)$:

$$\phi(x) = \int \frac{d^4k}{(2\pi)^4}\left[a(k)e^{-ik\cdot x} + a^\dagger(k)e^{ik\cdot x}\right], \tag{6.5}$$

where $k \cdot x = k_0 x_0 - \vec{k}\cdot\vec{x}$. We can apply a couple of conditions to ensure that the energy k_0 is positive and to satisfy the 'mass-shell' condition $k^2 = k_0^2 - \vec{k}^2 = m^2$ imposed by the Klein-Gordon equation. The first condition can be applied using the Heaviside step function

$$\theta(k_0) = \begin{cases} 1 & \text{if } k_0 > 0, \\ 0 & \text{if } k_0 < 0 \end{cases}$$

and the second by adding a (normalized) factor of $2\pi\delta(k^2 - m^2)$. So our actual measure, with those conditions explicitly stated, is

$$\int \frac{d^4k}{(2\pi)^4} 2\pi\delta(k^2 - m^2)\theta(k_0).$$

[6.1] This is shown, for example, in Physics 52 at Amherst, which typically uses either or both of Jackson [26] and Landau [29] as texts.

[6.2] Technically Eqn. 6.3 describes the Lagrangian *density*; the actual Lagrangian or Hamiltonian ('energy') is the integral over spatial dimensions of the corresponding density. However, the "density" is almost universally omitted, a convention we adopt despite its initial potential for confusion.

We can simplify this somewhat by defining $\omega_k = \sqrt{\vec{k}^2 + m^2} > 0$, which means $\delta(k^2 - m^2) = \delta(k_0^2 - \omega_k^2)$. Using the identity

$$\delta(f(x)) = \sum_i \frac{\delta(x - x_i)}{\frac{df}{dx}}$$

(where the sum is taken over all the roots of $f(x)$) to break up the δ-function gives

$$\delta(k_0^2 - \omega_k^2) = \frac{k_0 - \omega_k}{2k_0} + \frac{k_0 + \omega_k}{2k_0}$$

since $k_0 = \pm\omega_k$ are the two roots of $f(k_0) = k_0^2 - \omega_k^2$.

So we have

$$\int \frac{d^4k}{(2\pi)^4} 2\pi\delta(k^2 - m^2)\theta(k_0) = \int \frac{d^4k}{(2\pi)^3} \frac{1}{2k_0} (\delta(k_0 - \omega_k) + \delta(k_0 + \omega_k))\theta(k_0). \tag{6.6}$$

$\theta(k_0)$ ensures that $k_0 + \omega_k > 0$, so $\delta(k_0 + \omega_k) = 0$. Integrating over k_0 plucks out the ω_k from the remaining δ-function (the only effect it has on the exponentials is to replace $k_0 \to \omega_k$) and we are left with

$$\int \frac{d^4k}{(2\pi)^4} 2\pi\delta(k^2 - m^2)\theta(k_0) = \int \frac{d^3k}{(2\pi)^3 2\omega_k}. \tag{6.7}$$

So we can write Eqn. 6.5 as

$$\phi(x) = \int \frac{d^3k}{(2\pi)^3 2\omega_k} \left[a(k)e^{-ik\cdot x} + a^\dagger(k)e^{ik\cdot x}\right], \tag{6.8}$$

which is simpler for some calculations.

We have introduced the Fourier transform, Eqn. 6.8, because we wish to show that $a(k)$ and $a^\dagger(k)$ are analogous to the standard creation and annihilation operators familiar from nonrelativistic quantum mechanics (see Townsend [62] for an introduction). The derivation is similar to that used in the context of the harmonic potential and can be found in more detail in Hatfield [23, Chap. 3] and with more rigor in Ryder [51, Chap. 4].

First we define the momentum field $\pi(\vec{x}, t)$ conjugate to $\phi(\vec{x}, t)$ in the intuitive way:

$$\pi(\vec{x}, t) = \frac{\partial \mathcal{L}}{\partial \dot{\phi}(\vec{x}, t)}, \tag{6.9}$$

where $\dot{\phi}(\vec{x}, t) = \frac{\partial}{\partial t}\phi(\vec{x}, t)$. $\phi(\vec{x}, t)$ and $\pi(\vec{x}, t)$ obey the same equal-time commutation relations (ETCRs) as the nonrelativistic position and momentum operators:

$$[\phi(\vec{x}, t), \phi(\vec{x}', t)] = 0 = [\pi(\vec{x}, t), \pi(\vec{x}', t)] \tag{6.10}$$

$$[\phi(\vec{x}, t), \pi(\vec{x}', t)] = i\delta\left(\vec{x} - \vec{x}'\right). \tag{6.11}$$

Since we can write the Lagrangian $\mathcal{L}$ (Eqn. 6.26) as

$$\mathcal{L} = \frac{1}{2}\left(\dot{\phi}^2 - \left(\vec{\nabla}\phi\right)^2 - \mu_0^2\phi^2 - \frac{1}{2}\lambda\phi^4\right)$$

we see that Eqn. 6.9 reduces to

$$\pi(\vec{x},t) = \dot{\phi}(\vec{x},t) = \int \frac{d^3k}{(2\pi)^3}\frac{1}{2\omega_k}\left[-i\omega_k a(k)e^{-ik\cdot x} + i\omega_k a^\dagger(k)e^{ik\cdot x}\right]. \tag{6.12}$$

We can now invert Eqns. 6.8 and 6.12 to express $a(k)$ and $a^\dagger(k)$ in terms of $\phi(x)$ and $\pi(x)$:

$$\begin{aligned}
&\int d^3x e^{-i\vec{k}'\cdot\vec{x}}\left[\omega_{k'}\phi(\vec{x},t) + i\pi(\vec{x},t)\right] \\
&= \int \frac{d^3x d^3k}{(2\pi)^3 2\omega_k}\left[a(k)\left(\omega_{k'}+\omega_k\right)e^{-i\vec{x}\cdot(\vec{k}'-\vec{k})}e^{-i\omega_k t} + a^\dagger(k)\left(\omega_{k'}-\omega_k\right)e^{-i\vec{x}\cdot(\vec{k}'+\vec{k})}e^{i\omega_k t}\right].
\end{aligned}$$

Integrating over x replaces the exponentials with δ-functions, $\int dx e^{ix} = 2\pi\delta(x)$, giving

$$= \int \frac{d^3k}{2\omega_k}\left[a(k)\delta\left(\vec{k}-\vec{k}'\right)\left(\omega_{k'}+\omega_k\right)e^{-i\omega_k t} + a^\dagger(k)\delta\left(\vec{k}+\vec{k}'\right)\left(\omega_{k'}-\omega_k\right)e^{-i\omega_k t}\right],$$

whose second term vanishes since $\vec{k}' = -\vec{k} \to \omega_{k'} - \omega_k = \sqrt{\vec{k}'^2+m^2} - \sqrt{\vec{k}^2+m^2} = 0$. Similarly, the δ-function in the first term extracts $\vec{k} = \vec{k}'$ and hence $\omega_k = \omega_{k'}$, so the final result is

$$\int d^3x e^{-i\vec{k}'\cdot\vec{x}}\left[\omega_{k'}\phi(\vec{x},t) + i\pi(\vec{x},t)\right] = a(k')e^{-i\omega_{k'}t}$$

$$\to a(k) = \int d^3x e^{ik\cdot x}\left[\omega_{k'}\phi(\vec{x},t) + i\pi(\vec{x},t)\right] \tag{6.13}$$

$$\to a^\dagger(k) = \int d^3x e^{-ik\cdot x}\left[\omega_{k'}\phi(\vec{x},t) - i\pi(\vec{x},t)\right]. \tag{6.14}$$

So now we can quickly derive the ETCRs for $a(k)$ and $a^\dagger(k)$ by using those for $\phi(\vec{x},t)$ and $\pi(\vec{x},t)$ (Eqn. 6.10):

$$\begin{aligned}
\left[a(k), a^\dagger(k')\right] &= \int d^3x d^3x' e^{ik\cdot x}e^{-ik'\cdot x'}\left[\omega_k\phi(\vec{x},t) + i\pi(\vec{x},t), \omega_{k'}\phi(\vec{x}',t) - i\pi(\vec{x}',t)\right] \\
&= \int d^3x d^3x' e^{ik\cdot x}e^{-ik'\cdot x'}\left(-i\omega_k\left[\phi(\vec{x},t),\pi(\vec{x}',t)\right] + i\omega_{k'}\left[\pi(\vec{x},t),\phi(\vec{x}',t)\right]\right) \\
&= \int d^3x d^3x' e^{ik\cdot x}e^{-ik'\cdot x'}\left(\omega_k+\omega_{k'}\right)\delta(\vec{x}-\vec{x}') \\
&= \int d^3x e^{ix(k-k')}\left(\omega_k+\omega_{k'}\right) = (2\pi)^3\left(\omega_k+\omega_{k'}\right)\delta(k-k').
\end{aligned}$$

We can use the first coordinate in the δ-function to require $\omega_k = \omega_{k'}$ (due to the mass-shell condition $k_0 = \omega_k$), leaving as a final result

$$\left[a(k), a^\dagger(k')\right] = (2\pi)^3 2\omega_k\delta^3(\vec{k}-\vec{k}'). \tag{6.15}$$

A similar calculation quickly gives

$$\begin{aligned}
\left[a(k), a^\dagger(k')\right] &= \int d^3x d^3x' e^{ik\cdot x}e^{ik'\cdot x'}\left[\omega_k\phi(\vec{x},t) + i\pi(\vec{x},t), \omega_{k'}\phi(\vec{x}',t) + i\pi(\vec{x}',t)\right] \\
&= \int d^3x d^3x' e^{ik\cdot x}e^{ik'\cdot x'}\left(\omega_{k'}-\omega_k\right)\delta(\vec{x}-\vec{x}'),
\end{aligned}$$

so

$$[a(k), a(k')] = 0 = \left[a^\dagger(k), a^\dagger(k')\right]. \tag{6.16}$$

We can now construct the suggestively-named operator

$$N(k) = a^\dagger(k)a(k) \tag{6.17}$$

and derive its ETCRs:

$$\begin{aligned}[N(k), N(k')] &= a^\dagger(k)\left[a(k), a^\dagger(k')\right]a(k') + a^\dagger(k')\left[a^\dagger(k), a(k')\right]a(k) \\ &\propto \left(a^\dagger(k)a(k') - a^\dagger(k')a(k)\right)\delta(\vec{k} - \vec{k}') \\ [N(k), N\,(k')] &= 0. \end{aligned} \tag{6.18}$$

This means that we can form a basis out of the eigenstates $|n(k)\rangle$ of $N(k)$, where $N(k)|n(k)\rangle = n(k)|n(k)\rangle$. To consider the action of $a(k)$ and $a^\dagger(k)$ on $|n(k)\rangle$, we write down the simple commutation relations

$$\left[N(k), a^\dagger(k)\right] = a^\dagger(k)\left[a(k), a^\dagger(k)\right] + a(k)\left[a^\dagger(k), a^\dagger(k)\right] = a^\dagger(k), \tag{6.19}$$

$$[N(k), a(k)] = a^\dagger(k)\left[a(k), a(k)\right] + a(k)\left[a^\dagger(k), a(k)\right] = -a(k), \tag{6.20}$$

and consider

$$N(k)a^\dagger(k)|n(k)\rangle = a^\dagger(k)N(k)|n(k)\rangle + a^\dagger(k)|n(k)\rangle = (n(k)+1)a^\dagger(k)|n(k)\rangle \tag{6.21}$$

$$N(k)a(k)|n(k)\rangle = a(k)N(k)|n(k)\rangle - a(k)|n(k)\rangle = (n(k)-1)a(k)|n(k)\rangle. \tag{6.22}$$

That is, $a^\dagger(k)$ and $a(k)$ are creation and annihilation operators, analogous to their familiar counterparts from the simple harmonic oscillator (SHO) in nonrelativistic quantum mechanics. $a^\dagger(k)$ creates a scalar particle of four-momentum k, transforming an n-particle state into an $(n+1)$-particle state. Similarly, $a(k)$ transforms an n-particle state into an $(n-1)$-particle state by destroying a pre-existing particle of four-momentum k. Should no such particle exist, we have $a(k)|0(k)\rangle = 0 \rightarrow N(k)|0(k)\rangle = a^\dagger a(k)|0(k)\rangle = 0$. The derivation that $a(k)|n(k)\rangle = \sqrt{n}|(n-1)(k)\rangle$ and $a^\dagger(k)|n(k)\rangle = \sqrt{n+1}|(n+1)(k)\rangle$ is the same as in the case of the SHO.

$N(k)$ may be interpreted as the number operator for particles with momentum k, so long as its eigenvalues are all non-negative integers. It is straightforward to see that this is the case, first by considering the norm of $a(k)|n(k)\rangle$, which must be non-negative.

$$(a(k)|n(k)\rangle)^\dagger(a(k)|n(k)\rangle) = \langle n(k)|a^\dagger(k)a(k)|n(k)\rangle = n(k)\langle n(k)|n(k)\rangle \geq 0, \tag{6.23}$$

so $n(k) \geq 0$ since the norm $\langle n(k)|n(k)\rangle$ is positive. To establish that $n(k)$ is an integer, we simply note that states are generated by repeated applications of $a^\dagger$ to the ground state $n(k) = 0$, each of which increases its value by 1.

With some rather tedious manipulation, we can substitute Eqn. 6.8 into Eqn. 6.4 to show that the Hamiltonian takes the same form as that of the SHO,

$$\mathcal{H} = k_0\left(N(k) + \frac{1}{2}\right). \tag{6.24}$$

Actually, since $\mathcal{H}$ is really the Hamiltonian density, we see that the total Hamiltonian H is equivalent to an infinite sum of harmonic oscillators:

$$H = \int \frac{d^3k}{(2\pi)^3 2k_0} k_0 \left(N(k) + \frac{1}{2} \right). \tag{6.25}$$

This presents the slight complication that the nonzero ground state energies of these infinitely-many oscillators makes H infinite. Fortunately, since experiments measure only energy differences, we can remove this constant term from our theory with no observable effects (at least, so far as we can tell – cf. Peskin and Schroeder [43, Chap. 22]).

6.3 Lagrangian, Hamiltonian and Equation of Motion

The ϕ^4 theory is described by the Lagrangian (in Minkowski space),

$$\mathcal{L} = \frac{1}{2}\partial_\mu\phi\partial^\mu\phi - \frac{1}{2}\mu_0^2\phi^2 - \frac{\lambda}{4}\phi^4. \tag{6.26}$$

The corresponding Hamiltonian is clearly

$$\mathcal{H} = \frac{1}{2}\partial_\mu\phi\partial^\mu\phi + \frac{1}{2}\mu_0^2\phi^2 + \frac{\lambda}{4}\phi^4. \tag{6.27}$$

Note that both these equations are invariant under reflection symmetry $\phi \to -\phi$. They can be parameterized in several different ways, as long as they include two independent parameters. We choose this μ_0^2–λ formulation because it allows us to qualitatively interpret μ_0^2 as related to the mass of the scalar particles described by the theory and λ as the strength of the self-coupling interaction. See Balog *et al.* [3, 4] and Parsa [42] for examples of alternate parameterizations.

To gain some traction, we'll work in only two dimensions, one spatial dimension and one time dimension. We'll note any important features of the four-dimensional (three space + 1 time) case, though typically derivations will be analogous, simply replacing x with $\vec{x}$, $\frac{\partial}{\partial x}$ with $\vec{\nabla}$, and so on.

It is easy to see that in two dimensions our independent parameters μ_0^2 and λ both need to have dimension $mass^2$ (that is, $[\mu_0^2] = [\lambda] = 2$ – see Section 1.3). Since the action $S = \int d^2x\mathcal{L}$ appears as the argument of an exponential in the path-integral formulation, e^{iS}, it must be dimensionless.[6.3] Therefore, $[\mathcal{L}] = 2$ in order to cancel out the two factors of $mass^{-1}$ that come from d^2x. The partial derivatives ∂_μ in the first term of the Lagrangian, Eqn. 6.26, each have $[\partial] = 1$, meaning that ϕ must be dimensionless. Therefore, in order for all three terms to have the same dimensions, $[\mu_0^2] = [\lambda] = 2$. The four-dimensional case (analyzed in Zee [70, Chap. III.2]) is completely analogous: we

[6.3] See Townsend [62, Chap. 8] for a brief introduction to path integrals. It is also worth consulting the volume their inventor wrote with Hibbs [19].

simply have $[\mathcal{L}] = 4$, which requires $[\phi] = 1$, $[\mu_0^2] = 2$ and $[\lambda] = 0$ (i.e., a dimensionless coupling).

The equation of motion for the ϕ^4 theory is given by the Euler-Lagrange equation, which works out in a straightforward manner:

$$\partial_\mu \left(\frac{\partial \mathcal{L}}{\partial(\partial_\mu \phi)} \right) = \frac{\partial \mathcal{L}}{\partial \phi}$$

$$\partial^2 \phi = \frac{\partial^2 \phi}{\partial t^2} - \frac{\partial^2 \phi}{\partial x^2} = -\left(\mu_0^2 \phi + \lambda \phi^3 \right). \tag{6.28}$$

We can rearrange Eqn. 6.28 into a form reminiscent of the Klein-Gordon equation, Eqn. 6.1:

$$\left(\partial^2 + \mu_0^2 \right) \phi = -\lambda \phi^3. \tag{6.29}$$

Note that the self-interaction λ term makes the equation of motion nonlinear, which allows for the appearance of nonperturbative phenomena such as solitons.

Like the Ising model, the ϕ^4 potential $V = \frac{1}{2}\mu_0^2\phi^2 + \frac{\lambda}{4}\phi^4$ possesses two distinct phases if $\lambda > 0$. If $\lambda < 0$, the theory has no stable minima, though a metastable state is possible if $\mu_0^2 > 0$ (Fig. 6.1) – the state can persist until it tunnels through the energy barrier and falls off to negative infinite energy. The two stable phases with $\lambda > 0$ correspond to positive μ_0^2 (a symmetric phase with $\langle \phi \rangle = 0$) and negative μ_0^2 (a broken symmetry phase with minima at $\pm\sqrt{\frac{-\mu_0^2}{\lambda}}$), shown below in Figs. 6.2 and 6.3, respectively.

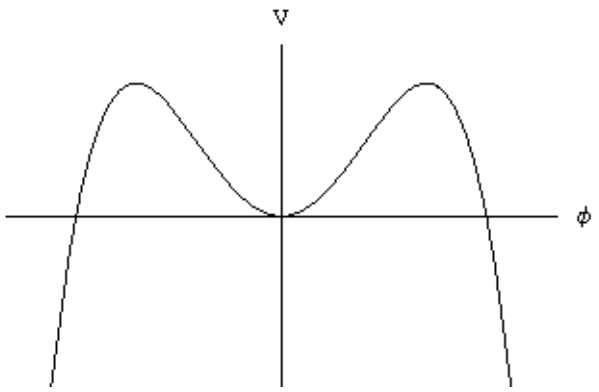

Figure 6.1: Metastable ϕ^4 potential: $\mu_0^2 > 0$, $\lambda < 0$; $V = \frac{1}{2}\mu_0^2\phi^2 + \frac{\lambda}{4}\phi^4$

As we expect from our study of the Ising model, the existence of symmetric and broken phases implies that ϕ^4 theory exhibits a spontaneous symmetry-breaking phase transition. As a matter of fact, the parallel with the Ising model is much deeper. As first shown thirty years ago by Chang [13], the ϕ^4 phase transition in two dimensions is second order, as is that of the Ising model. In fact, ϕ^4 theory in two dimensions is in the same universality class as the two-dimensional Ising model, which means that both systems have the same critical exponents governing their behavior near their respective critical points.

Calculating the critical points of ϕ^4 theory on two- and four-dimensional lattices was one of our main projects, which we will discuss in detail in Section 7.5. Before we begin that discussion, however, we will introduce solitons, another focus of our work.

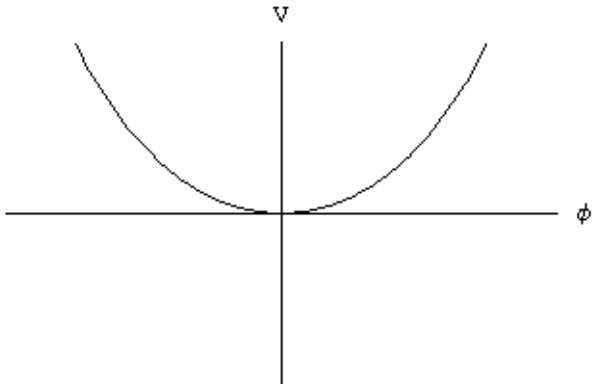

Figure 6.2: ϕ^4 potential in the symmetric phase: $\mu_0^2 > 0$, $\lambda > 0$; $V = \frac{1}{2}\mu_0^2\phi^2 + \frac{\lambda}{4}\phi^4$

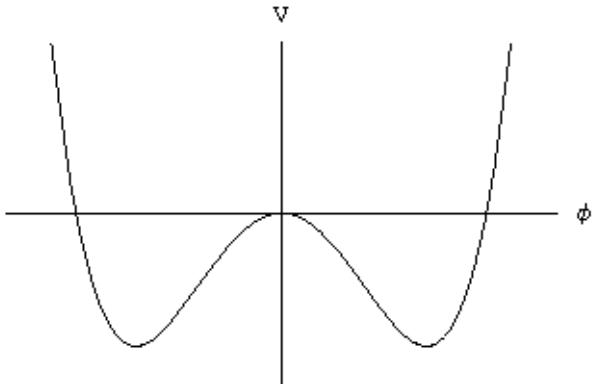

Figure 6.3: ϕ^4 potential in the broken phase: $\mu_0^2 < 0$, $\lambda > 0$; $V = \frac{1}{2}\mu_0^2\phi^2 + \frac{\lambda}{4}\phi^4$

6.4 Constant and Soliton Solutions

We can easily see from Eqn. 6.28 and Figs. 6.3 and 6.2 that the ϕ^4 theory equation of motion has constant solutions ($\frac{\partial}{\partial t}\phi = 0 = \frac{\partial}{\partial x}\phi$)[6.4] in both the symmetric ($\phi(x,t) = 0$) and broken phases ($\phi(x,t) = \pm\sqrt{-\mu_0^2/\lambda}$). Although the existence in the broken phase of two degenerate vacua that transform into one another under reflection $\phi \to -\phi$ is worth mentioning, none of these constant solutions themselves are particularly interesting. The energy density of $\phi(x,t) = 0$ is simply flat zero, and since the energy density of $\phi(x,t) = \pm\sqrt{-\mu_0^2/\lambda}$ is constant and finite, it could be legitimately set to zero simply by adding a constant to the Lagrangian, Eqn. 6.26. This is done, for example, by Parsa [42].

Fortunately, there exist somewhat more interesting solutions to Eqn. 6.28, known as kink solutions. To gain some traction, we'll consider the static (time-independent) kink given by

$$\phi_\pm(x,t) = \pm\sqrt{\frac{-\mu_0^2}{\lambda}}\tanh\left[\sqrt{\frac{-\mu_0^2}{2}}x\right]. \tag{6.30}$$

Technically, the positive solution is the kink solution, and the negative is called the 'antikink' (since they clearly interfere destructively through trivial cancellation). To obtain a time-dependent kink, a Lorentz transformation can be applied to Eqn. 6.30. The kink can also be localized around some point x_0 by replacing $x \to x - x_0$ in the argument of the hyperbolic tangent.

It is a straightforward calculation to verify that Eqn. 6.30 satisfies the equation of motion Eqn. 6.28:

$$\frac{\partial\phi}{\partial x} = \pm\frac{-\mu_0^2}{\sqrt{2\lambda}}\mathrm{sech}^2\left[\sqrt{\frac{-\mu_0^2}{2}}x\right]$$

$$\frac{\partial^2\phi}{\partial t^2} - \frac{\partial^2\phi}{\partial x^2} = -\frac{\partial^2\phi}{\partial x^2} = \mp\mu_0^2\sqrt{\frac{-\mu_0^2}{\lambda}}\mathrm{sech}^2\left[\sqrt{\frac{-\mu_0^2}{2}}x\right]\tanh\left[\sqrt{\frac{-\mu_0^2}{2}}x\right],$$

and

$$\mu_0^2\phi + \lambda\phi^3 = \phi\left(\mu_0^2 + \lambda\frac{-\mu_0^2}{\lambda}\tanh^2\left[\sqrt{\frac{-\mu_0^2}{2}}x\right]\right) = \phi\mu_0^2\mathrm{sech}^2\left[\sqrt{\frac{-\mu_0^2}{2}}x\right]$$

$$-\left(\mu_0^2\phi + \lambda\phi^3\right) = \mp\mu_0^2\sqrt{\frac{-\mu_0^2}{\lambda}}\mathrm{sech}^2\left[\sqrt{\frac{-\mu_0^2}{2}}x\right]\tanh\left[\sqrt{\frac{-\mu_0^2}{2}}x\right],$$

as needed. What the kink solution does, qualitatively, is continuously connect the two degenerate vacua. As shown in Fig. 6.4, the solution starts in one well of the potential corresponding to one of $\phi(x,t) = \pm\sqrt{-\mu_0^2/\lambda}$ (Fig. 6.3) as $x \to -\infty$ and crosses over to the other ($\phi \to -\phi$) as $x \to \infty$.

The kink solution is commonly called a *soliton*, a term that has evolved somewhat over time, as outlined by Parsa [42]. Originally, "soliton" denoted special solutions to

[6.4] We continue to work in only two dimensions.

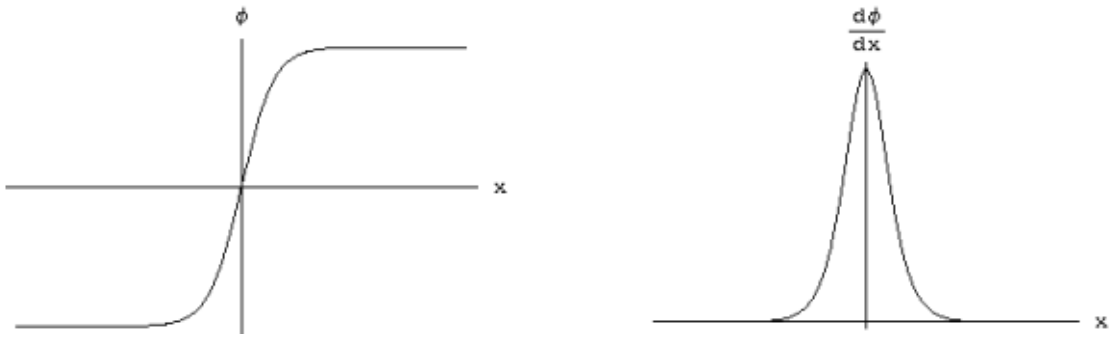

Figure 6.4: Kink solution of ϕ^4 equation of motion

nonlinear wave equations with the property that when two solitons interacted or scattered with each other, they would retain their initial shapes and velocities. This is not actually the case with the ϕ^4 kink introduced above. Instead, the kink obeys a less stringent condition that has largely replaced the original definition of a soliton. Now a solution to nonlinear field equations is considered a soliton if it has finite nonzero energy and is confined to a finite region – that is, it doesn't dissipate. Let's check these conditions for the ϕ^4 kink, starting by analogy with a more familiar soliton.

The standard example of a soliton is the kink solution to the Sine-Gordon equation, a nonlinear cousin of the Klein-Gordon equation:

$$\partial^2\phi = \sin(\phi), \tag{6.31}$$

which is the Euler-Lagrange equation corresponding to the Lagrangian

$$\mathcal{L} = \frac{1}{2}\partial_\mu\phi\partial^\mu\phi - \cos(\phi). \tag{6.32}$$

The simplest soliton (and antisoliton) solution to the Sine-Gordon equation is

$$\psi_\pm(x,t) = 4\tan^{-1}\left\{\pm\exp\left[\frac{x-vt}{\sqrt{1-v^2}}\right]\right\}, \tag{6.33}$$

which describes a soliton in the old-fashioned sense of travelling waves that emerge from scattering processes with their initial shapes and velocities. The Sine-Gordon soliton is shown in Fig. 6.5; it clearly resembles the ϕ^4 soliton in Fig. 6.4, though the exponential in the arctangent keeps ψ positive. In fact, as illustrated by Ramond [50], the ϕ^4 Lagrangian can actually be considered an approximation to an appropriately-parameterized Sine-Gordon Lagrangian. Expanding the cosine term in this reparameterized Sine-Gordon Lagrangian gives

$$\mathcal{L} = \frac{1}{2}\partial_\mu\phi\partial^\mu\phi + \frac{\mu_0^4}{\lambda}\left(\cos\left[\frac{\phi\sqrt{\lambda}}{\mu_0}\right] - 1\right) = \frac{1}{2}\partial_\mu\phi\partial^\mu\phi + \frac{\mu_0^4}{\lambda}\left(\frac{-\lambda\phi^2}{2\mu_0^2} + \frac{\lambda^2\phi^4}{4!\mu_0^4}\right) + \mathcal{O}(\phi^6)$$

$$\mathcal{L} = \frac{1}{2}\partial_\mu\phi\partial^\mu\phi - \frac{1}{2}\mu_0^2\phi^2 + \frac{\lambda}{4!}\phi^4 + \mathcal{O}(\phi^6),$$

which is precisely the ϕ^4 Lagrangian, up to fourth order in ϕ and a trivial redefinition of the parameter λ.

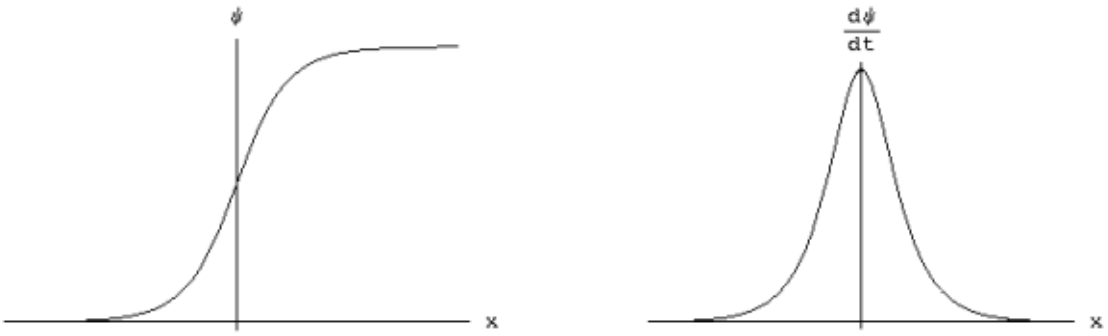

Figure 6.5: Kink solution of Sine-Gordon equation

We are interested in the relationship between the ϕ^4 kink and the Sine-Gordon soliton since it allows us to apply qualitative results for the one to the other. In particular, Ryder [51, Chap. 10] has shown that the Sine-Gordon soliton can be visualized as an infinite horizontal string with pegs attached to it at equally spaced intervals, each peg connected to its neighbor with a small spring and acted upon by gravity. The ground state corresponds to every peg hanging vertically, while the soliton corresponds to the situation shown in Fig. 6.6, in which the pegs transition through a kink from pointing up to pointing down. This demonstrates the stability of the kink: a transition into the ground state would require turning over a (semi-)infinite number of pegs, requiring a (semi-)infinite amount of energy.

Figure 6.6: The Sine-Gordon soliton as an infinite line of interacting pegs, from Ryder [51]

The stability of the kink makes it possible to define a conserved integer charge corresponding to it. Note that this charge is not the result of an invariance in $\mathcal{L}$ under some symmetry transformation; it is not a Noether charge. Instead it is a discrete topological charge, a result of the fact that $\phi(\infty) = -\phi(-\infty)$. In other words, the stability of the kink is a topological consequence of the boundary conditions.

We have shown by analogy with the Sine-Gordon soliton that the ϕ^4 kink is stable and so doesn't dissipate. It is a direct calculation to check that the energy for the ϕ^4 kink (and antikink) is finite and nonzero. We start off with

$$\mathcal{H} = \frac{1}{2}\left(\frac{\partial\phi}{\partial t}\right)^2 + \frac{1}{2}\left(\frac{\partial\phi}{\partial x}\right)^2 + \frac{1}{2}\mu_0^2\phi^2 + \frac{\lambda}{4}\phi^4 \qquad \frac{\partial\phi}{\partial x} = \mp\frac{\mu_0^2}{\sqrt{2\lambda}}\mathrm{sech}^2(\beta x)$$

$$\phi = \pm\sqrt{\frac{-\mu_0^2}{\lambda}}\tanh\left[\sqrt{\frac{-\mu_0^2}{2}}x\right] = \pm\sqrt{\frac{-\mu_0^2}{\lambda}}\tanh(\beta x) \qquad \frac{\partial\phi}{\partial t} = 0$$

$$\rightarrow \mathcal{H} = \frac{\mu_0^4}{4\lambda}\left(\mathrm{sech}^4(\beta x) + \tanh^4(\beta x) - 2\tanh^2(\beta x)\right)$$

$$\mathcal{H} = \frac{\mu_0^4}{4\lambda}(2\mathrm{sech}^4(\beta x) - 1) \tag{6.34}$$

Now to find the energy of the kink itself, we need to subtract the ground state (vacuum) energy $\mathcal{H}_V$ from that expression. It's trivial to show that for either degenerate vacuum

$\mathcal{H}_V = -\mu_0^4/4\lambda$. So the total energy of the kink is just

$$H = \frac{\mu_0^4}{2\lambda}\int \operatorname{sech}^4(\beta x) = \frac{\mu_0^4}{2\lambda\beta}\frac{1}{3}[2+\operatorname{sech}^2(\beta x)]\tanh(\beta x)|_{x\to-\infty}^{x\to\infty}$$
$$= \frac{\mu_0^4}{2\lambda\beta}\frac{1}{3}(4) = \frac{2\sqrt{2}}{3}\frac{\left(\sqrt{-\mu_0^2}\right)^3}{\lambda}.$$

The energy of the ϕ^4 kink is finite and nonzero, establishing that it is indeed a soliton. Note that classically the soliton's mass is just this energy,

$$M_{cl} = \frac{2\sqrt{2}}{3}\frac{\left(\sqrt{-\mu_0^2}\right)^3}{\lambda}. \tag{6.35}$$

We'll return to solitons when we consider how to simulate them on the lattice in Section 7.6. Those interested in additional analytic details of solitons would do well to consult Rajaraman [49], a comprehensive and invaluable resource on solitons and their cousins instantons.

6.5 Feynman Diagrams

Perturbation theory means Feynman diagrams. – M. Veltman [63]

Feynman diagrams are among the most famous tools of high-energy physics; the vast majority of physicists and students of physics have seen Feynman diagrams at some point, even if they don't necessarily know what they mean. In this section we present a basic outline of the motivation for and machinery of these common beasts.

Essentially, Feynman diagrams are only tools to visualize and simplify perturbative calculations. The perturbation expansion is carried out in terms of the interaction Hamiltonian, which we define by separating our original Hamiltonian $\mathcal{H}$ (and analogously for the Lagrangian $\mathcal{L}$) into two pieces – the free-field Hamiltonian $\mathcal{H}_0$ and the interaction Hamiltonian $\mathcal{H}_{int}$.

$$\mathcal{H} = \mathcal{H}_0 + \mathcal{H}_{int} \tag{6.36}$$

All that we have done is to separate the free-particle portion of the Hamiltonian from any non-quadratic terms. In ϕ^4 theory, for instance, $\mathcal{H}_0 = \frac{1}{2}\partial_\mu\phi\partial^\mu\phi + \frac{1}{2}\mu_0^2\phi^2$ and $\mathcal{H}_{int} = \frac{\lambda}{4}\phi^4$ (cf. Eqn. 6.27). Typically $\mathcal{H}_{int}$ depends on some coupling, $\lambda/4$ in the case of ϕ^4 theory. Thus the perturbation expansion in $\mathcal{H}_{int}$ can also be considered an expansion in terms of the coupling.[6.5]

By splitting up the Hamiltonian in this manner, we are effectively moving into the 'interaction picture' of quantum mechanics. Townsend [62, Chap. 15] presents a basic

[6.5] Although we've set $\hbar = 1$, Ryder [51, Chap. 9] shows that an expansion in the interaction Hamiltonian is equivalent to an expansion in $\hbar$.

introduction to the interaction picture and shows that in it the time-evolution operator $U(t, t_0)$ satisfies the equation

$$i\frac{\partial}{\partial t}U(t, t_0) = H_{int}U(t, t_0), \tag{6.37}$$

which can be solved iteratively to give U as a function of $H_{int} = \int d^3x\mathcal{H}_{int}$:

$$\begin{aligned} U(t, t_0) = 1 + (-i)\int_{t_0}^{t} dt_1 H_{int}(t_1) + (-i)^2\int_{t_0}^{t} dt_1\int_{t_0}^{t_1} dt_2 H_{int}(t_1)H_{int}(t_2) \\ + (-i)^3\int_{t_0}^{t} dt_1\int_{t_0}^{t_1} dt_2\int_{t_0}^{t_2} dt_3 H_{int}(t_1)H_{int}(t_2)H_{int}(t_3) + \cdots \end{aligned} \tag{6.38}$$

We can simplify that by using the identity

$$\begin{aligned} \int_{t_0}^{t} dt_1\cdots\int_{t_0}^{t_{n-1}} dt_n H_{int}(t_1)\cdots H_{int}(t_n) \\ = \frac{1}{n!}\int_{t_0}^{t} dt_1\cdots dt_n T\{H_{int}(t_1)\cdots H_{int}(t_n)\} \end{aligned} \tag{6.39}$$

where T is the time-ordering operator,

$$T\{A(t_1)B(t_2)\} = \begin{cases} A(t_1)B(t_2) & \text{if } t_1 > t_2, \\ B(t_2)A(t_1) & \text{if } t_2 > t_1 \end{cases}$$

This result is intuitive in the second-order term, and generalizes to higher powers. $\int_{t_0}^{t} dt_1\int_{t_0}^{t_1} dt_2$ integrates over the triangle in the t_1t_2-plane that forms the lower half of the square $\int_{t_0}^{t} dt_1 dt_2$ (cf. Peskin [43, Chap. 4]). The time-ordering operator guarantees that the integrand is symmetric around the $t_1 = t_2$ line, so only a factor of $\frac{1}{2} = \frac{1}{n!}$ is needed to make the integrals equal.

We define the S-matrix by taking $t_0 \to -\infty$, $t \to \infty$ in Eqn. 6.38. An element $\mathcal{M} = \langle q|S|p\rangle$ in the S-matrix thus gives the amplitude $\mathcal{M}$ for some initial state $|p\rangle$ to evolve into some final state $|q\rangle$ through some interaction. Converting from the Hamiltonians $H(t)$ to the Hamiltonian densities $\mathcal{H}(t, \vec{x}) = \mathcal{H}(x)$ gives a reasonably simple expression for the S-matrix:

$$S = 1 + \sum_{n=1}^{\infty}\frac{(-i)^n}{n!}\int d^4x_1\cdots d^4x_n T\{\mathcal{H}_{int}(x_1)\cdots\mathcal{H}_{int}(x_n)\}. \tag{6.40}$$

Let's perform some calculations in ϕ^4 theory to see what this means. We'll work in four dimensions in this section, but it will be easy to convert our results to two dimensions for our later work.

To zeroth order in the interaction Hamiltonian, we simply have $\langle q|S|p\rangle = \langle q|p\rangle = \delta(q-p)$, which makes intuitive sense: if there is no interaction, the states do not change. Let's now add in a first-order interaction that alters the states. For definiteness, suppose the initial state $|p\rangle = |p_1, p_2\rangle$ consists of two ϕ particles with momenta p_1 and p_2, while in the final state $|q\rangle = |p_3, p_4\rangle$ the two particles have different momenta p_3 and p_4.

This situation is illustrated diagrammatically in Fig. 6.7, our first Feynman diagram. ϕ particles (those created or annihilated by the ϕ operators) are represented by solid lines. Interactions are represented by a dot at a vertex where four particles meet. (It must always be four since there are four factors of ϕ in the interaction Hamiltonian.) Fig. 6.7 shows two particles coming in from the left, interacting at a vertex at spacetime point x, and then heading off to the right with different momenta.

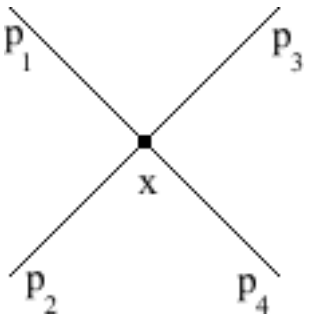

Figure 6.7: Feynman diagram for four-point tree-level interaction

The integral corresponding to this diagram is

$$-i\int d^4x\langle p_3,p_4|\mathcal{H}_{int}(x)|p_1,p_2\rangle = \frac{-i\lambda}{4}\int d^4x\langle p_3,p_4|\phi(x)\phi(x)\phi(x)\phi(x)|p_1,p_2\rangle. \quad (6.41)$$

Recall from Eqn. 6.5 that each $\phi(x)$ will either create or annihilate a particle at x. Thus one of the four ϕ operators will annihilate the existing p_1-momentum particle, one of the three remaining factors will annihilate p_2, while the remaining two will create p_3 and p_4. There are $4! = 24$ different ways the interaction could occur, so this "symmetry factor" needs to be included in the result. Each ϕ has Fourier transform

$$\phi(x) = \int \frac{d^4k}{(2\pi)^4}\left[a(k)e^{-ik\cdot x} + a^\dagger(k)e^{ik\cdot x}\right],$$

where $a^\dagger(k)$ and $a(k)$ are the creation and annihilation operators, respectively. So to create a particle of momentum p, we perform the integration

$$\langle p|\phi(x)|0\rangle = \int \frac{d^4k}{(2\pi)^4}\langle p|a(k)|0\rangle e^{-ik\cdot x} + \int \frac{d^4k}{(2\pi)^4}\langle p|a^\dagger(k)|0\rangle e^{ik\cdot x}$$
$$\langle p|\phi(x)|0\rangle = \int \frac{d^4k}{(2\pi)^4}e^{ik\cdot x}\delta(k-p) = e^{ip\cdot x}$$

($n(p) = 0$). Similarly, annihilating a particle of momentum p gives a factor of $e^{-ip\cdot x}$. Eqn. 6.41 therefore becomes

$$\frac{-i\lambda 4!}{4}\int d^4x e^{-ip_1\cdot x}e^{-ip_2\cdot x}e^{ip_3\cdot x}e^{ip_4\cdot x} = -6i\lambda\int d^4x e^{-ix\cdot(p_1+p_2-p_3-p_4)}$$
$$= -6i\lambda(2\pi)^4\delta^4(p_1+p_2-p_3-p_4).$$

The overall δ-function simply enforces conservation of momentum, and is commonly separated from the amplitude itself. In fact, we typically write (cf. Peskin [43, Pg. 112])

$$i\mathcal{M}(2\pi)^4\delta^4(p_1+p_2-p_3-p_4) = -6i\lambda(2\pi)^4\delta^4(p_1+p_2-p_3-p_4) \rightarrow \mathcal{M} = -6\lambda. \quad (6.42)$$

This normalization has the advantage that we receive the same result in any number of dimensions (with the ϕ integrals normalized appropriately). In two dimensions, for instance, we would have

$$i\mathcal{M}(2\pi)^2\delta^2(p_1+p_2-p_3-p_4) = -6i\lambda(2\pi)^2\delta^2(p_1+p_2-p_3-p_4) \rightarrow \mathcal{M} = -6\lambda. \quad (6.43)$$

Looking over the above calculation, we see it also seems possible to use two factors of ϕ to create and then annihilate a particle at point x. We would initially have only one particle $|p_1\rangle$ and would end up with $|p_2\rangle$, as illustrated in Fig. 6.8, which shows how creating and annihilating a particle at a single interaction results in a closed loop in the diagram. The corresponding integral is

$$-i\int d^4x\langle p_2|\mathcal{H}_{int}(x)|p_1\rangle = \frac{-i\lambda}{4}\int d^4x\langle p_2|\phi(x)\phi(x)\phi(x)\phi(x)|p_1\rangle. \quad (6.44)$$

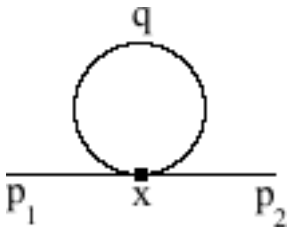

Figure 6.8: Feynman diagram for two-point single-loop "leaf" interaction

Eqn. 6.44 will only have a symmetry factor of $4 \cdot 3 = 12$ since there are only two observable particles to be dealt with. The remaining factors of ϕ will be 'contracted' with each other to create and annihilate a particle with momentum q, which raises some difficulties. First of all, since we only have one factor of the interaction Hamiltonian, there's only one interaction, so everything needs to take place at the same point in spacetime. However, the particle has to be created before it can be annihilated. This prompts us to have it created and annihilated at two different points, x and y, by having one of the remaining ϕ act at point x and the other at point y. We'll then take the limit $y \rightarrow x$. The final complication is that either of the two remaining ϕ could create this particle, just as any one of the four could annihilate $|p_1\rangle$, so we'll need to take both possibilities $x_0 < y_0$ and $x_0 > y_0$ into account. Finally, note that although the momentum with which this particle will be created is arbitrary (forcing us to retain the momentum integration from Eqn. 6.5), the momentum at which it is to be annihilated is completely determined. These considerations lead us to write

$$\frac{-12i\lambda}{4}\int d^4x e^{ix\cdot(p_2-p_1)}\left[\theta(x-y)\int\frac{d^4q}{(2\pi)^4}e^{iq\cdot(x-y)} + \theta(y-x)\int\frac{d^4q}{(2\pi)^4}e^{iq\cdot(y-x)}\right] \quad (6.45)$$

Although that expression doesn't appear particularly tractable, the term in square brackets is a common beast (cf. Peskin and Schroeder [43], Veltman [63, Chap. 3] and Ryder [51, Chap. 6]) known as the **Feynman propagator** $\Delta_F(x-y)$, which we can

define in terms of Wightman functions Δ^+ and Δ^-:

$$\Delta^+(x-y) = \int \frac{d^4q}{(2\pi)^4} e^{iq\cdot(x-y)}\theta(q_0)\delta(q^2-m^2)$$
$$\Delta^-(x-y) = \int \frac{d^4q}{(2\pi)^4} e^{iq\cdot(y-x)}\theta(q_0)\delta(q^2-m^2)$$
$$\Delta_F(x-y) = \theta(x-y)\Delta^+(x-y) + \theta(y-x)\Delta^-(x-y).$$

(As discussed in Section 6.2, the factor of $\theta(q_0)\delta(q^2+m^2)$ is implicit in all of our four-momentum integrals. We explicitly note it here to make our definitions more rigorous.) It is a nontrivial and not particularly interesting exercise to simplify the expression into the standard form

$$\Delta_F(x-y) = \int \frac{d^4q}{(2\pi)^4} \frac{ie^{-iq\cdot(x-y)}}{q^2-\mu_0^2+i\epsilon}, \tag{6.46}$$

where ϵ is infinitesimal and will be taken to zero after it has served its purpose of simplifying later calculations. As we can see by recalling the steps that brought us to this point, the Feynman propagator is a quantity that describes the creation, propagation, and annihilation of a particle. To substitute Eqn. 6.46 back into Eqn. 6.45, we first take the limit $y \to x$, which gives us

$$\Delta_F(0) = \int \frac{d^4q}{(2\pi)^4} \frac{i}{q^2-\mu_0^2+i\epsilon}$$
$$\to 3\lambda \int d^4x e^{ix\cdot(p_2-p_1)} \int \frac{d^4q}{(2\pi)^4} \frac{1}{q^2-\mu_0^2+i\epsilon}.$$

Since the momentum integral is now independent of x, we can evaluate the position integral very easily,

$$3\lambda(2\pi)^4\delta^4(p_1-p_2) \int \frac{d^4q}{(2\pi)^4} \frac{1}{q^2-\mu_0^2+i\epsilon}.$$

Setting that equal to $i\mathcal{M}(2\pi)^4\delta^4(p_1-p_2)$ we find

$$\mathcal{M} = -3i\lambda \int \frac{d^4q}{(2\pi)^4} \frac{1}{q^2-\mu_0^2+i\epsilon}. \tag{6.47}$$

So far we've just mentioned in passing the Feynman diagrams corresponding to these interactions. They don't seem to have had any role to play. So why are they so famous? What is their purpose?

The key that makes Feynman diagrams so useful is that the results of calculations like those above can be generated by applying a small set of **Feynman rules** to the Feynman diagram corresponding to the given situation. Our calculations above certainly suggest that some things will come out the same for every diagram – vertices will result in factors of λ, loops in propagators Δ_F. While two examples may seem a small base from which to establish general rules that hold for every conceivable situation in ϕ^4 theory, we will proceed nonetheless; the reader is welcome to perform more calculations.

Here are our Feynman rules for ϕ^4 theory diagrams in a D-dimensional space:

- Add a factor of $-i\lambda/4$ for each vertex.
- Add a factor of $(2\pi)^D\delta(p_1+\cdots+p_4)$ to conserve momentum at each vertex.
- Add a factor of $i(q^2-\mu_0^2+i\epsilon)^{-1}$ for each "internal" line with momentum q
- Integrate $\int d^Dq(2\pi)^{-D}$ over each "internal" momentum q that is not determined by conservation of momentum.
- Multiply by the symmetry factor.

We claim that by applying these simple rules to any Feynman diagram, we'll obtain the final result of the corresponding integral, equal to $i\mathcal{M}(2\pi)^4\delta(p_1+\cdots+p_n)$, where this delta function describes overall momentum-conservation.

It's easy to apply these rules to the two cases worked out above. In the four-point tree-level diagram, Fig. 6.7, we have $D=4$, one vertex (factor of $-i\lambda(2\pi)^4\delta(p_1+p_2-p_3+p_4)/4$), no internal lines or undetermined momenta, and a symmetry factor of 4!. So we claim

$$i\mathcal{M}(2\pi)^4\delta(p_1+p_2-p_3-p_4)=-6i\lambda(2\pi)^4\delta(p_1+p_2-p_3-p_4)\rightarrow\mathcal{M}=-6\lambda, \quad (6.48)$$

which is exactly what we found above with somewhat more effort. Similarly, for the two-point leaf diagram, Fig. 6.8, we have $D=4$, one vertex (factor of $-i\lambda(2\pi)^4\delta(p_1-p_2)/4$), one internal line (factor of $i(q^2-\mu_0^2+i\epsilon)^{-1}$), one undetermined momentum q (integrate over $\int d^4q(2\pi)^{-4}$) and a symmetry factor of 12. So we claim

$$\begin{aligned} i\mathcal{M}(2\pi)^4\delta(p_1-p_2)&=-3i\lambda(2\pi)^4\delta(p_1-p_2)\int\frac{d^4q}{(2\pi)^4}\frac{i}{q^2-\mu_0^2+i\epsilon}\\ \rightarrow\mathcal{M}&=-3i\lambda\int\frac{d^4q}{(2\pi)^4}\frac{1}{q^2-\mu_0^2+i\epsilon}, \end{aligned} \quad (6.49)$$

which is again precisely our result. Comparing this last paragraph to the preceding calculations should convince anyone of the value of Feynman diagrams as computational aids.

6.6 Renormalization

If the doors of perception were cleansed, everything would appear to man as it is, infinite. – William Blake

Now that we have an efficient means of performing calculations based on Feynman diagrams, we will quickly find that the results of such calculations are often infinite – unfortunately unphysical. Renormalization is the process by which we "hide the infinities" and force our calculations to produce meaningful results; it is thus one of the most important concepts of quantum field theory.[6.6] Although renormalization was initially greeted with a certain amount of skepticism, it is now firmly established as a legitimate procedure, one that is essential to making sense of quantum field theory.

[6.6] Renormalization can also be performed in classical field theory to address similar problems, though this is less common. See Thirring [61, Chap. 8] for an example.

The first step of renormalization is to perform a 'regularization' procedure that separates the infinite and finite portions of the result, making it easy to see what is causing problems and how they can be fixed. Renormalization itself then analogously separates the Lagrangian itself into renormalized and "counterterm" parts. There are several different ways to perform both regularization (e.g., cut-off regularization, Pauli-Villars regularization, dimensional regularization and lattice regularization) and renormalization (e.g., minimal subtraction, modified subtraction and recursive subtraction schemes, which can be interpreted through the equivalent means of counterterms or renormalization factors).

Not all quantum field theories are renormalizable, though renormalizability is necessary for a theory to be physically meaningful. Fortunately ϕ^4 theory is renormalizable in two and four dimensions. In the following sections, we will go into greater depth about the problem of divergences in quantum field theory and briefly introduce various regularization schemes before presenting sample calculations in which we use dimensional regularization and the minimal subtraction scheme to renormalize ϕ^4 theory up to one loop in two dimensions.

6.6.1 A Divergent Diagram in ϕ^4 Theory

Your theory is getting sicker and sicker. – M. Veltman

Let's consider the (deceptively) simple leaf diagram introduced in Section 6.5, Fig. 6.8. Applying our Feynman rules (in two dimensions, this time) results in a symmetry factor of 12, a factor of $-i\lambda(2\pi)^2\delta(p_1-p_2)/4$ from the single vertex, a factor of $i(q^2-m^2+i\epsilon)^{-1}$ from the internal line of the loop, and finally an integration $\int d^2q(2\pi)^{-2}$ over the undetermined momentum q. So

$$\begin{aligned} i\mathcal{M}(2\pi)^2\delta(p_1-p_2) &= -3i\lambda(2\pi)^2\delta(p_1-p_2)\int\frac{d^2q}{(2\pi)^2}\frac{i}{q^2-\mu_0^2+i\epsilon} \\ \rightarrow\mathcal{M} &= -3i\lambda\int\frac{d^2q}{(2\pi)^2}\frac{1}{q^2-\mu_0^2+i\epsilon}. \end{aligned} \tag{6.50}$$

Unfortunately, the integral diverges as $q\rightarrow\infty$, not exactly what we would expect from a physical process.

The easiest way to spot this divergence is to break up d^2q up into a radial and angular part, $d^2q\sim qdqd\Omega$. Unfortunately, this cannot be done in Minkowski space, since $q^2=q_0^2-\vec{q}^2=q_0^2-q_1^2$ is not the square of the radius. There's no guarantee it's even positive! We can get around this by performing a "Wick rotation", contour integration in the complex q_0 plane.[6.7] This is the reason we have kept the small factor of $i\epsilon$ in the denominator. This factor leads to poles in the complex q_0 plane at points

$$q^2-\mu_0^2+i\epsilon=q_0^2-\vec{q}^2-\mu_0^2+i\epsilon=0\rightarrow q_0=\pm\sqrt{\vec{q}^2+\mu_0^2-i\epsilon}\approx\pm(\sqrt{\vec{q}^2+\mu_0^2}-i\delta),$$

where

$$\delta=\frac{\epsilon}{2\sqrt{\vec{q}^2+\mu_0^2}}$$

[6.7] See Gamelin [21, Chap. VII] for an introduction to the residue calculus and contour integration.

As shown in Fig. 6.9, the poles are δ away from the real q_0 axis, which means that the "Figure-8" contour shown contains no poles. We see that the role of $i\epsilon$ is to shift the poles slightly off of the real axis to make the contour integration simpler; it is also possible to construct equivalent contours with the poles on the axis (cf. Peskin [43, Chap. 2]) but they are considerably more complicated. As $R \to \infty$, the integrand goes like R^{-2} but the arc lengths only increases linearly in R, so the contribution of the integral over those legs is zero. Thus the integral over real q_0 from $-\infty$ to ∞ is equal to the integral over imaginary q_0 from $-i\infty$ to $i\infty$,

$$\int_{-\infty}^{\infty} \frac{dq_0}{2\pi} \frac{1}{q_0^2 - \vec{q}^2 - \mu_0^2 + i\epsilon} = \int_{-i\infty}^{i\infty} \frac{dq_0}{2\pi} \frac{1}{q_0^2 - \vec{q}^2 - \mu_0^2 + i\epsilon}. \tag{6.51}$$

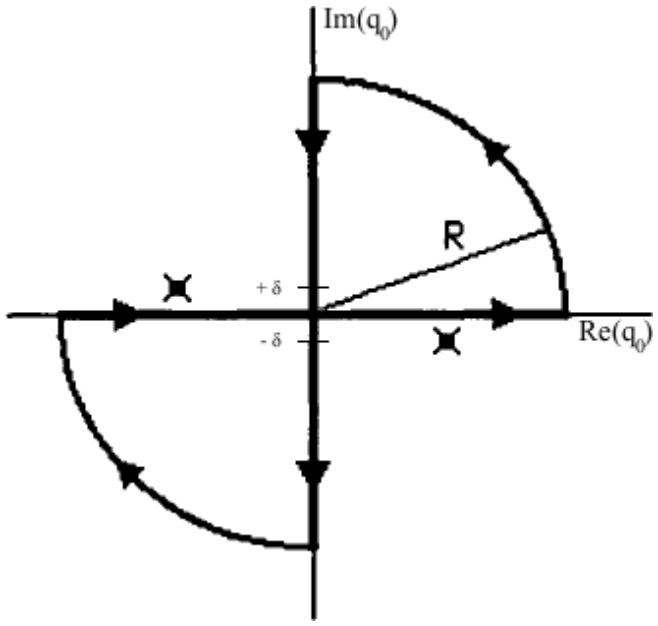

Figure 6.9: "Figure-8" contour from Klaus and Griffiths [28]

We define $q_2 = -iq_0$, which gives

$$\int_{-i\infty}^{i\infty} \frac{dq_0}{2\pi} \frac{1}{q_0^2 - \vec{q}^2 - \mu_0^2 + i\epsilon} = \int_{-\infty}^{\infty} \frac{idq_2}{2\pi} \frac{1}{-q_2^2 - \vec{q}^2 - \mu_0^2 + i\epsilon} = \int_{-\infty}^{\infty} \frac{-idq_2}{2\pi} \frac{1}{\bar{q}^2 + \mu_0^2},$$

where in the last step we defined $\bar{q}^2 = q_1^2 + q_2^2 = \vec{q}^2 + q_2$ and finally took the limit $\epsilon \to 0$ since ϵ has served its purpose of simplifying the contour. Note that we have effectively "rotated" from Minkowski to Euclidean space. So now the whole integral is

$$-3i\lambda \int \frac{d^2q}{(2\pi)^2} \frac{1}{q^2 - \mu_0^2 + i\epsilon} = -3\lambda \int \frac{d^2\bar{q}}{(2\pi)^2} \frac{1}{\bar{q}^2 + \mu_0^2}. \tag{6.52}$$

Clearly the four-dimensional case will be entirely analogous since all we have altered is q_0.

So now we're able to break up $\bar{q}$ into a radial and angular part, $d^2\bar{q} = \bar{q}d\bar{q}d\Omega$. For high $\bar{q}$, $\bar{q}^2 + \mu_0^2 \approx \bar{q}^2$, so the integral itself is proportional to $\int \bar{q}^{-1} d\bar{=} \log(\bar{q})$, which

diverges as $\bar{q} \to \infty$ (ultraviolet divergence).[6.8] In four dimensions, there are another two factors of $\bar{q}$ in the numerator, so the integral is proportional to $\bar{q}^2$ and diverges quadratically as $\bar{q} \to \infty$. In the next subsection we will show how to isolate these ultraviolet divergences through the method of dimensional regularization, in preparation for eliminating them through renormalization. First, however, we'll mention some of the various ways regularization can be performed.

Essentially, ultraviolet divergence can be considered an effect of our ignorance of the true short-distance (high-energy) physics. The most direct way to remove the divergence from this calculation is to restrict it to lower energies where it is valid by imposing a finite upper bound $\Lambda < \infty$ on the integral. A slightly more elegant way to accomplish the same result is to multiply the integrand by $\frac{\Lambda^2}{\Lambda^2 - \bar{q}^2}$. In either case we can use the regularized result to identify and remove any terms that diverge as $\Lambda \to \infty$, obtaining a finite renormalized result. Although this "cut-off regularization" is the simplest approach, it destroys the gauge invariance of theories to which it is applied. Pauli-Villars regularization, in which a fictitious field of mass M plays a role analogous to the cut-off Λ, suffers from this same problem, while lattice regularization (in which the finite spacing between lattice points imposes a finite upper bound on $\bar{q}$) interferes with translational and rotational invariance.

Alternately we could note that the integral in Eqn. 6.50 is just barely divergent since it goes like $\int \bar{q}^{-1} d\bar{q}$. If we could somehow decrease the power of $\bar{q}$ in that integral to $-1-\epsilon < -1$, then it would be perfectly convergent in the ultraviolet regime. Analytic regularization does exactly this, replacing $\bar{q}^{-n}$ with $\bar{q}^{-zn}$, where $z \in \mathbb{C}$ with $\mathrm{Re}(z)$ is large enough to make the integral converge. (Thus analytic regularization is actually applicable even in cases where the divergence in the integral is worse than logarithmic.) The result is then "analytically continued" to near the physical value $z = 1$ using the methods of complex analysis, and any divergences that appear as $z \to 1$ can be removed to complete renormalization.

Dimensional regularization, one of the most common approaches and the primary one used we used, follows a similar procedure. However, instead of altering the power of momentum in the denominator, dimensional regularization instead performs the calculation in $d = 2 - \epsilon$ (or $d = 4 - \epsilon$) dimensions, which reduces the power of $d\bar{q}$ in the integral to produce an analytic result.[6.9] This result typically contains terms that diverge as $\epsilon \to 0$, which can be removed to complete renormalization. The key to the popularity of dimensional regularization is that it preserves the symmetries of the original theory, in particular gauge symmetry.[6.10] Although the procedure was originally developed for logarithmic divergence, it is still possible to use it to regularize high-order divergences.

Before carrying out a dimensional regularization of ϕ^4 theory in detail, we will introduce a simple test that allows us to determine whether or not a given Feynman

[6.8] Although $\log(\bar{q})$ itself diverges as $\bar{q} \to 0$ (infrared divergence), the 'mass' μ_0^2 in the denominator means that infrared divergences will only be a potential problem for quantum fields describing massless particles. This is fortunate, since infrared divergences are generally much more difficult to address than ultraviolet divergences, and will not be considered here.

[6.9] Technically, $\epsilon \in \mathbb{C}$.

[6.10] Kleinert and Schulte-Frohlinde [27, Chap. 8] discuss some additional technical advantages.

diagram actually requires renormalization: calculation of the "superficial degree of divergence".

6.6.2 Superficial Degree of Divergence

The superficial degree of divergence D is defined as the difference between the number of factors of momentum in the numerator and denominator of the integral corresponding to a Feynman diagram that contains one or more loops. Using the rules for Feynman diagrams presented above in Section 6.5, the superficial degree of divergence can easily be expressed in terms of properties of the diagrams. In d dimensions, each internal line in a diagram contributes two factors of momentum to the denominator of the integral through the propagator, while each vertex adds d factors to the numerator, along with a δ function to conserve momentum. Applying the δ functions leads to the result that the number of independent momenta over which (d-dimensional) integration takes place is equal to the number of loops in the diagram, L. (If there are no loops, the δ functions eliminate the integration, so there can be no divergence.) Thus we have

$$D = dL - 2I,$$

where I is the number of internal lines.

In ϕ^4 theory this can be reformulated by noting that each vertex involves four lines, so writing the number of external lines as E and the number of vertices as n, we have $4n = 2I + E$, since internal lines connect two vertices and so need to be counted twice. In a similar vein, although there are I internal momenta, momentum conservation imposes $n-1$ relations between them – there is momentum conservation at each of the n vertices, but overall momentum conservation makes one of these constraints redundant. The number of undetermined loop momenta is therefore $L = I - n + 1 = n - \frac{E}{2} + 1$, giving

$$D = d - \left(\frac{d}{2} - 1\right) E + n(d-4) \tag{6.53}$$

which reduces to $D = 2 - 2n$ in two dimensions and $D = 4 - E$ in four.

Clearly the integral will diverge if $D \geq 0$. Note that in two dimensions the superficial degree of divergence of ϕ^4 theory decreases with increasing n, guaranteeing a small (finite) number of divergent diagrams. Theories that have this feature are known as "super-renormalizable", while theories for which the superficial degree of divergence increases with increasing n (implying infinitely many ever-more divergent diagrams) are nonrenormalizable. In four dimensional ϕ^4 theory, D doesn't depend on n at all; the theory turns out to be renormalizable.

With a bit more manipulation (done in Peskin and Schroeder [43, Chap. 10], for instance), we can use Eqn. 6.53 to establish a convenient relation between the mass dimension of the coupling constant and the renormalizability of the theory. If the coupling constant has positive mass dimension, then the theory is super-renormalizable. The theory is precisely renormalizable if the coupling constant is dimensionless and nonrenormalizable if the mass dimension of the coupling constant is negative. This result holds

for a number of quantum field theories, including quantum electrodynamics (QED) and ϕ^4 theory. Recall that for ϕ^4 theory the dimension of the coupling constant is $[\lambda] = 2$ in the super-renormalizable two-dimensional theory and $[\lambda] = 0$ in the renormalizable four-dimensional theory, in agreement with this result.

For the leaf diagram above, we have $D = 0$ in two dimensions (logarithmic divergence) and $D = 2$ in four dimensions (quadratic divergence), as expected. Conveniently, it turns out that not only does $D \geq 0$ imply divergence, but the converse is almost true as well. Specifically, the Weinberg-Dyson convergence theorem (proposed in 1949 by Dyson [18] and fully proven by Weinberg [66] in 1960) establishes that a Feynman diagram converges if its superficial degree of divergence *and those of all its subgraphs* are negative. A subgraph of a diagram is just a portion of it that can be cleanly cut from its parent diagram to produce a sensible loop diagram of its own. In Figure 6.10, subgraphs of a number of Feynman diagrams (clearly of a different quantum field theory than ϕ^4) have been boxed off: 6.10.a has two subgraphs, while 6.10.b and 6.10.c each have one and 6.10.d has none. We will not attempt to prove Weinberg's Theorem here.

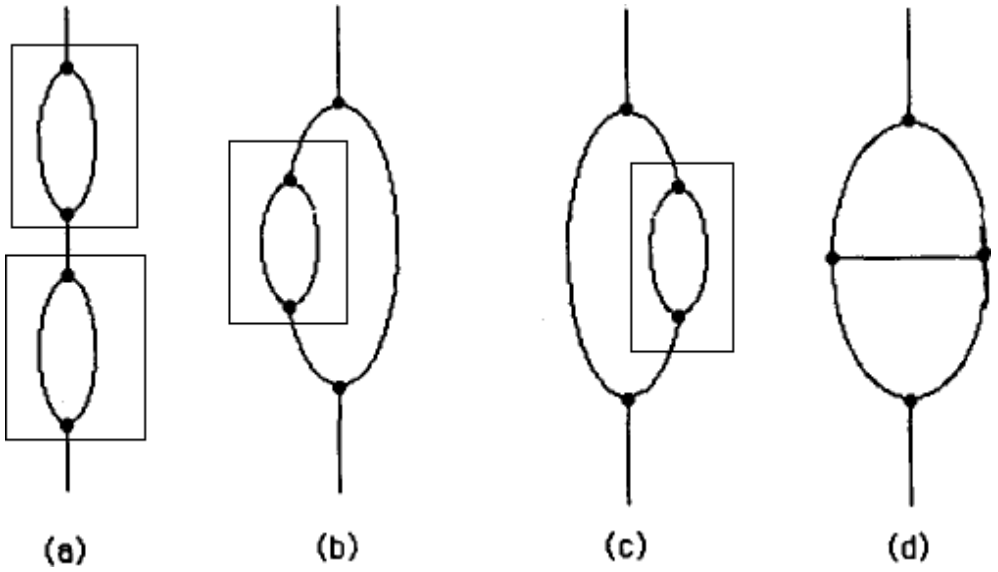

Figure 6.10: Subgraphs of Feynman diagrams from Kraus and Griffiths [28]

6.6.3 Dimensional Regularization of ϕ^4 Theory

So the superficial degree of divergence for ϕ^4 theory in two dimensions is $D = 2-2n$, which clearly shows that the leaf diagram is the only divergent diagram, since it is the only diagram with one vertex and $D \geq 0$. Thus the only integral which requires regularization is Eqn. 6.50, which we rewrite here in Euclidean form (for convenience, we have dropped the $^-$ from $\bar{q}$):

$$-3\lambda \int \frac{d^2q}{(2\pi)^2} \frac{1}{q^2 + \mu_0^2}. \tag{6.54}$$

We'll use dimensional regulation to separate the finite and infinite parts of that integral. As we will see, dimensional regulation is typically the most computationally tedious step

of renormalization. Recall that the general idea is to evaluate this integral in $2-\epsilon$ dimensions to see what portions of the result diverge as $\epsilon \to 0$. Varying the dimension will change the dimensionality of the coupling λ; to keep track of this we separate it into a dimensionless coupling constant λ_0 multiplied by a dimensionful $M^{2-\epsilon}$. Moving from 2 to $2-\epsilon$ dimensions thus entails

$$\frac{d^2q}{(2\pi)^2} = \frac{qdqd\Omega}{(2\pi)^2} \to \frac{d^{2-\epsilon}q}{(2\pi)^{2-\epsilon}} = \frac{q^{1-\epsilon}dqd^{1-\epsilon}\Omega}{(2\pi)^{2-\epsilon}}$$
$$-3\lambda \to -3\lambda_0 M^{2-\epsilon}.$$

The integral becomes

$$-3\lambda_0 \left(\frac{M}{2\pi}\right)^{2-\epsilon} \int_0^\infty \frac{q^{1-\epsilon}dq}{q^2+\mu_0^2} \int d^{1-\epsilon}\Omega. \tag{6.55}$$

General expressions for each of those two integrals are well-known:

$$\int d^{n-1}\Omega = \frac{2\pi^{n/2}}{\Gamma(n/2)}$$
$$\int_0^\infty \frac{x^k dx}{(x^n+a^n)^r} = \frac{(-1)^{r-1}\pi a^{k+1-nr}\Gamma\left(\frac{k+1}{n}\right)}{n\sin\left(\frac{k+1}{n}\pi\right)\Gamma\left(\frac{k+1}{n}-r+1\right)(r-1)!}.$$

Setting $n = 2-\epsilon$ in the first equation gives

$$\int d^{1-\epsilon}\Omega = \frac{2\pi^{1-\epsilon/2}}{\Gamma(1-\epsilon/2)}, \tag{6.56}$$

while setting $k = 1-\epsilon$, $n = 2$, $a = \mu_0$ and $r = 1$ makes the Γ-functions in the second equation cancel out, leaving

$$\int_0^\infty \frac{q^{1-\epsilon}dq}{q^2+\mu_0^2} = \frac{\pi m^{-\epsilon}}{2\sin\left(\frac{2-\epsilon}{2}\pi\right)}. \tag{6.57}$$

Putting everything together produces

$$-3\lambda_0 \left(\frac{M}{2\pi}\right)^{2-\epsilon} \frac{2\pi^{1-\epsilon/2}}{\Gamma(1-\epsilon/2)} \frac{\pi}{2\mu_0^\epsilon \sin\left(\frac{2-\epsilon}{2}\pi\right)}. \tag{6.58}$$

This can be simplified considerably through the identity

$$\frac{\pi}{\sin(p\pi)} = \Gamma(p)\Gamma(1-p) \to \frac{\pi}{\sin\left(\frac{2-\epsilon}{2}\pi\right)} = \Gamma(\epsilon/2)\Gamma(1-\epsilon/2).$$

Plugging that in, cancelling out the $\Gamma(1-\epsilon/2)$ and collecting everything raised to the ϵ power produces

$$\frac{-3\lambda_0 M^2}{4\pi}\Gamma(\epsilon/2)\left(\frac{2\sqrt{\pi}}{M\mu_0}\right)^\epsilon. \tag{6.59}$$

Finally we apply two more identities:

$$\Gamma(2/\epsilon) = 2/\epsilon - \gamma + \mathcal{O}(\epsilon)$$
$$z^\epsilon = 1 + \epsilon\log z + \mathcal{O}(\epsilon^2),$$

where $\gamma = 0.577216\ldots$ is the Euler-Mascheroni constant. Our final result is

$$\frac{-3\lambda_0 M^2}{4\pi}\left(\frac{2}{\epsilon} - \gamma + \log\left(\frac{4\pi}{M^2\mu_0^2}\right) + \mathcal{O}(\epsilon)\right). \tag{6.60}$$

There is only one term in that result that will diverge as $\epsilon \to 0$:

$$\frac{-3\lambda_0 M^2}{2\pi\epsilon} + \text{ finite.}$$

So there is only one term we have to remove through renormalization.

6.6.4 Renormalization of ϕ^4 Theory

Now that the divergent portion of the result has been separated from the finite quantities, we need to see how to reconfigure the Lagrangian in order to eliminate the divergent part without altering the convergent terms. There are two ways to proceed. First we could scale each of the variables in the Lagrangian by some 'renormalization factor' Z. An alternate approach more suited to diagrammatic analysis is to use 'counterterms', extra terms added to the Lagrangian to produce additional diagrams that cancel out the divergences in the original diagram(s). We will show how to move from renormalization factors to counterterms. There are various 'schemes' one could use to construct these counterterms. We illustrate the principle here with the simplest, the "minimal-subtraction scheme".

First of all, we rewrite our Lagrangian explicitly in terms of the "bare" field and parameters, ϕ_0, μ_0^2 and λ_0:

$$\mathcal{L} = \frac{1}{2}\left(\partial_\mu\phi_0\right)\left(\partial^\mu\phi_0\right) - \frac{1}{2}\mu_0^2\phi_0^2 - \frac{\lambda_0}{4}\phi_0^4. \tag{6.61}$$

We now express the bare quantities as products of the renormalized quantities and corresponding renormalization factors,

$$\phi_0 = \sqrt{Z_\phi}\phi_R \qquad \mu_0^2 = Z_{\mu^2}\mu_R^2 \qquad \lambda_0 = Z_\lambda\lambda_R,$$

which turns the Lagrangian into

$$\mathcal{L} = \frac{1}{2}Z_\phi\left(\partial_\mu\phi_R\right)\left(\partial^\mu\phi_R\right) - \frac{1}{2}Z_{\mu^2}Z_\phi\mu_R^2\phi_R^2 - Z_\phi^2 Z_\lambda\frac{\lambda_R}{4}\phi_R^4.$$

Next we rewrite all of the renormalization factors in the following form

$$Z_\phi = 1 + \delta Z_\phi \qquad Z_{\mu^2} = 1 + \delta Z_{\mu^2} \qquad Z_\lambda = 1 + \delta Z_\lambda. \tag{6.62}$$

Though we use "δ", we are not implying that any of the δZ are small quantities; indeed, they may be infinitely large. Inserting those expansions into the Lagrangian and retaining only terms to first order is δZ produces

$$\mathcal{L} = \mathcal{L}_R + \frac{1}{2}\delta Z_\phi\left(\partial_\mu\phi\right)\left(\partial^\mu\phi\right) - \frac{1}{2}(\delta Z_{\mu^2} + \delta Z_\phi)\mu^2\phi^2 - (\delta Z_\lambda + 2\delta Z_\phi)\frac{\lambda}{4}\phi^4. \tag{6.63}$$

(We have dropped the subscripts on ϕ_R, μ_R^2 and λ_R.) The first term has exactly the same form as the original Lagrangian, but with the bare field and bare parameters replaced by their renormalized counterparts.

We can interpret the other terms as additional diagrams, or 'counterterms'. There seem to be quite a few of them. However, it turns out that we can renormalize the leaf diagram even if we set $\delta Z_\phi = 0$ and $\delta Z_\lambda = 0$.[6.11] This is the simplest approach and the one we adopt, giving

$$\mathcal{L} = \mathcal{L}_R - \frac{1}{2}\delta Z_{\mu^2}\mu^2\phi^2 = \mathcal{L}_R - \frac{1}{2}\delta\mu^2\phi^2,$$

$$\mathcal{L} = \frac{1}{2}\left(\partial_\mu\phi\right)\left(\partial^\mu\phi\right) - \frac{1}{2}(\mu^2 + \delta\mu^2)\phi^2 - \frac{\lambda}{4}\phi^4. \tag{6.64}$$

The Lagrangian now looks exactly the same as the original, except that μ_0^2 has been written as $\mu^2 + \delta\mu^2$.

We want to choose $\delta\mu^2$ to make this expression finite. Loinaz and Willey [32] show that the final result is $\delta\mu^2 = 3\lambda A_{\mu^2}$, where

$$A_{\mu^2} = \int \frac{d^2p}{(2\pi)^2}\frac{1}{p^2+\mu^2} = \int_0^\infty e^{-\mu^2 t}\left[e^{-2t}I_0(2t)\right]^2 dt,$$

and I_0 is a modified Bessel function of the first kind. Loinaz and Willey actually use lattice regularization on a two-dimensional rectangular lattice to calculate their result, not dimensional regularization. Considering how we will apply this renormalization to two-dimensional lattice simulations in the next chapter, this is perhaps appropriate.

In sum, we find that in two dimensions neither ϕ nor λ need to be renormalized, while the renormalized μ^2 is given by

$$\mu^2 = \mu_0^2 + 3\lambda\int_0^\infty e^{-\mu^2 t}\left[e^{-2t}I_0(2t)\right]^2 dt. \tag{6.65}$$

There is only one "mass counterterm" $\delta\mu^2$ to cancel out the infinite part of the leaf diagram, which we illustrate in Fig. 6.11.

Figure 6.11: Mass renormalization of the leaf diagram in ϕ^4 theory

[6.11] This is the reason we have previously used μ_0^2 but not λ_0 or ϕ_0. We'll just show this result is consistent; to derive it from first principles we would have to introduce $\Sigma(p^n)$ functions, sums of the amplitudes of all "n-particle irreducible" diagrams.

6.6.5 Complications in the Broken Phase

Complicated as the calculations in the past few sections may have seemed, we actually made an unstated assumption to simplify things.[6.12] We assumed that the value of ϕ corresponding to the minimum-energy ground state (the "vacuum expectation value") was zero, $\mathcal{V} = \langle\phi\rangle = 0$. This assumption let up perform the perturbation expansion around zero: $\phi_0 = \mathcal{V} + \hat{\phi}_0 = \hat{\phi}_0$, where $\langle\hat{\phi}_0\rangle = 0$. This does not hold in the broken phase, $\mathcal{V} \neq 0$, resulting in extra factors of $\hat{\phi}_0^3$:

$$(\mathcal{V} + \hat{\phi}_0)^4 = \hat{\phi}_0^4 + 4\hat{\phi}_0^3\mathcal{V} + \cdots + \mathcal{V}^4$$

When $\mathcal{V} = 0$, of course, we just have $(\mathcal{V} + \hat{\phi}_0)^4 = \hat{\phi}_0^4$.

The stray $\hat{\phi}_0^3$ terms result in diagrams with vertices at which only three lines meet, as opposed to the four-line vertices we normally have in ϕ^4 theory. Of particular concern are "tadpole" diagrams like that shown in Fig. 6.12. By a calculation entirely analogous to that in Section 6.6.2, we can show that the two-dimensional superficial degree of divergence for ϕ^3 diagrams is still $D = 2 - 2n$ (as it is for all ϕ^n, in fact), meaning that the single-vertex tadpole diagram is divergent and needs to be renormalized. The other terms are less interesting since they are either finite or can be combined with the mass term. (In four dimensions the two-point tadpole diagram, Fig. 6.13 is also divergent; we'll only consider two dimensions for the moment.)

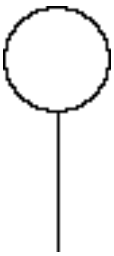

Figure 6.12: One-point "tadpole" diagram in the ϕ^4 broken phase

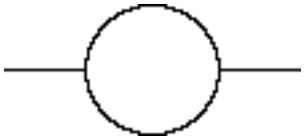

Figure 6.13: Two-point tadpole diagram in the ϕ^4 broken phase (not divergent in two dimensions)

Fortunately, now that we've seen the idea behind renormalization, this prospect is not necessarily so daunting. All we need to do, really, is specify an additional renormalization condition, namely that the one-point function vanishes. As illustrated diagrammatically in Fig. 6.14, this just means we need to add another counterterm that fully cancels out the tadpole diagram. We'll start by renormalizing ϕ_0 just as we did in the previous section,

$$\phi_0 = \sqrt{Z_\phi}\phi_R = \sqrt{Z_\phi}V + \sqrt{Z_\phi}\hat{\phi}_R,$$

[6.12] This assumption is (explicitly or silently) made by all of the references cited so far. Bochkarev and Willey [8] explore broken phase complications in the electroweak theory.

which leads to the relations

$$\phi_0 = \sqrt{Z_\phi}\phi_R \qquad \hat{\phi}_0 = \sqrt{Z_\phi}\hat{\phi}_R \qquad \mathcal{V} = \sqrt{Z_\phi}V$$

where $Z_\phi = 1 + \delta Z_\phi$, as usual.

Substituting those relations into the Lagrangian, we end up with terms proportional to V and containing no δZ:

$$\mathcal{L} = \frac{1}{2}(1+\delta Z_\phi)\partial_\mu\hat{\phi}\partial^\mu\hat{\phi} - \frac{1}{2}(1+\delta Z_{\mu^2}+\delta Z_\phi)\mu^2(V+\hat{\phi})^2 - \frac{\lambda}{4}(1+\delta Z_\lambda+2\delta Z_\phi)(V+\hat{\phi})^4. \quad (6.66)$$

(V is constant, so $\partial_\mu(V+\hat{\phi}) = \partial_\mu\hat{\phi}$.) Since we're just working to one loop, our calculation will be especially simple. We can write the renormalized $V = (1+\delta\xi)v$, where v is fixed by the zeroth order contribution to be simply the classical expectation value in the broken phase,

$$v = \sqrt{\frac{-\mu^2}{\lambda}}. \quad (6.67)$$

The additional counterterm proportional to $\delta\xi$ can be made to cancel out the tadpole diagram, as illustrated in Fig. 6.14. We won't actually evaluate the corresponding integrals to find a definite expression for $\delta\xi$; we'll simply note that by using Feynman diagrams and Feynman rules such a calculation is made reasonably straightforward.

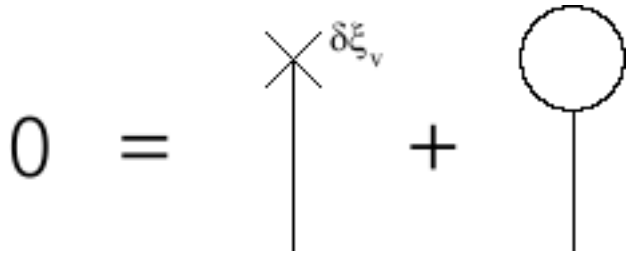

Figure 6.14: Tadpole renormalization in the ϕ^4 broken phase

6.6.6 Further reading

The danger already exists that the mathematicians have made a covenant with the devil to darken the spirit and to confine man in the bonds of Hell. – St. Augustine

Since renormalization is of such central importance to quantum field theory, many different authors have attempted to explain it in a variety of ways. Watson's [64, Pgs. 62–90] semi-popular account of quantum chromodynamics includes a good qualitative introduction to dimensional regularization and renormalization. Lepage [31] is another qualitative account of the essential features of renormalization, addressed to those familiar with the process but unclear about its physical meaning.

Kraus and Griffiths [28] gently introduce the mathematics of dimensional regularization and renormalization in the context of a simple model quantum field theory constructed for that purpose. Veltman [63] develops Feynman diagrams in detail using

the canonical operator approach and also briefly considers renormalization. Ryder [51, Chap. 9] is a standard textbook that explores renormalization at considerable depth for a variety of theories, including ϕ^4 theory, primarily using dimensional regularization and the functional-integral approach. Kleinert and Schulte-Frohlinde [27, Chap. 8–9] take a similar approach, but since ϕ^4 theory is the sole topic of the volume, their treatment of renormalization is more detailed but less general than Ryder's.

There are also texts that explain and use the other approaches to regularization. In particular, Zee [70, Chap. III] concentrates on cut-offs, while Hatfield [23, Chap. 17–18] uses the Pauli-Villars regularization procedure. Books and articles about lattice quantum field theory often discuss lattice regularization. See for instance Montvay and Munster [39, Chap. 3], Smit [54, Chap. 2], and Pierro [44, Chap. 2], [45, Chap. 2].

Chapter 7

ϕ^4 Theory on the Lattice

In this chapter, we will outline our simulations of ϕ^4 theory on the lattice. We begin by exploring the parallels between quantum field theories and classical statistical systems. Following this discussion we will walk through the procedure used to discretize ϕ^4 theory in order to treat it computationally.

Finally, we will summarize our actual simulations of phase transitions in two- and four-dimensional ϕ^4 theory and solitons in two dimensions. We will present the results of our calculations of the critical coupling constant $[\lambda/\mu^2]_{crit}$ and soliton masses in two dimensions and compare them with the current literature.

7.1 From Quantum Field Theory to Classical Statistical System

We have now seen both a detailed discussion of how to efficiently simulate classical statistical systems on the lattice as well as an introduction to the basics of relativistic quantum field theory. In this section we will combine the two strands of our earlier discussions by showing how we can rigorously translate the ϕ^4 quantum field theory into a statistical system similar to those we introduced above. We will assume only that the system is in equilibrium; the simple derivation we will present does not hold if this is not the case.

The easiest way to present this derivation is to use the path-integral formulation of quantum mechanics. We will not develop path integrals here (they are introduced in Townsend [62, Chap. 8] and fleshed out in Ryder [51, Chap. 5–6]). Instead we will simply cite the result that for a field theory governed by a Lagrangian $\mathcal{L}$, the generating functional of correlation functions (which produces the Green's functions that actually describe physical processes) is

$$Z[J] = \int \mathcal{D}\phi e^{iS} = \int \mathcal{D}\phi \exp\left[i \int d^4x (\mathcal{L} + J\phi)\right], \tag{7.1}$$

where $J(x)$ is an external source that couples to the field ϕ and $S = \int d^4x(\mathcal{L} + J\phi)$ is the resulting action. $Z[J]$ is reminiscent of the partition function from statistical mechanics, in that it has the general structure of a sum (integral) of an exponential weight over all possible configurations of the system.

We can make that analogy precise by transforming from Minkowski space to Euclidean space through a Wick rotation similar to that done in Subsection 6.6.1. We define $x_4 = ix_0$, which allows us to define the Euclidean quantities x_E^2, d^4x_E and $\partial_{E\mu}$ as

$$x^2 = x_0^2 - \vec{x}^2 \to -\vec{x}^2 - x_4^2 = -x_E^2 \tag{7.2}$$

$$d^4x = dx_0 d^3x \to -id^3x dx_4 = -id^4x_E \tag{7.3}$$

$$\begin{aligned} \partial_\mu\phi\partial^\mu\phi &= \left(\frac{\partial\phi}{\partial x_0}\right)^2 - \left(\vec{\nabla}\phi\right)^2 \\ &\to \left(i\frac{\partial\phi}{\partial x_4}\right)^2 - \left(\vec{\nabla}\phi\right)^2 = -\left(\vec{\nabla}\phi\right)^2 - \left(\frac{\partial\phi}{\partial x_4}\right)^2 = -\left(\partial_{E\mu}\phi\right)^2 \end{aligned} \tag{7.4}$$

These quantities are exactly what one would expect to find in a four-dimensional Euclidean space: $x_E^2 = x_1^2 + x_2^2 + x_3^2 + x_4^2$) and so on. (Clearly the results in two dimensions are completely analogous.) x_4 behaves like a fourth spatial dimension as opposed to a time dimension. We can apply the Wick rotation to the ϕ^4 Lagrangian to find

$$\mathcal{L} = \frac{1}{2}\left(\partial_\mu\phi\partial^\mu\phi\right) - \frac{1}{2}\mu_0^2\phi^2 - \frac{\lambda}{4}\phi^4 \to -\frac{1}{2}\left(\partial_{E\mu}\phi\right)^2 - \frac{1}{2}\mu_0^2\phi^2 - \frac{\lambda}{4}\phi^4 = -\mathcal{L}_E,$$

where we define the Euclidean Lagrangian as

$$\mathcal{L}_E = \frac{1}{2}\left(\partial_{E\mu}\phi\right)^2 + \frac{1}{2}\mu_0^2\phi^2 + \frac{\lambda}{4}\phi^4. \tag{7.5}$$

Note that the Euclidean Lagrangian actually has the form of an energy density (Hamiltonian): it is non-negative and becomes large when the field ϕ has large amplitude or large gradients.

Now we have all we need to apply the Wick rotation to the generating functional. For simplicity we'll turn off the external source $J(x) = 0$ for the moment.

$$\begin{aligned} Z[0] = \int \mathcal{D}\phi \exp\left[i\int d^4x\mathcal{L}\right] &\to \int \mathcal{D}\phi \exp\left[i\int \left(-id^4x_E\right)\left(-\mathcal{L}_E\right)\right] \\ &= \int \mathcal{D}\phi \exp\left[-\int d^4x_E\mathcal{L}_E\right] = \int \mathcal{D}\phi e^{-S_E} = Z. \end{aligned} \tag{7.6}$$

Since the Euclidean action $S_E = \int d^4x_E L_E$ has the form of an energy, Z is precisely the partition function describing the statistical mechanics of some macroscopic system (approximated by treating its fluctuating variable as a continuous field). We see that J is analogous to the external magnetic field which we earlier set to zero in the context of the Ising model.

Thus performing a Wick rotation turns the e^{iS} from the Feynman path integral into a Boltzmann factor e^{-S_E}. We can therefore treat ϕ^4 theory as a statistical theory in

four-dimensional Euclidean space with energy given by its Euclidean action

$$S_E = \int d^4x_E \mathcal{L}_E = \int d^4x_E \left[\frac{1}{2}\left(\partial_{E\mu}\phi\right)^2 + \frac{1}{2}\mu_0^2\phi^2 + \frac{\lambda}{4}\phi^4\right]. \tag{7.7}$$

Therefore we can apply the Monte Carlo methods developed in earlier chapters to ϕ^4 theory, once we discretize it so it can be simulated on a lattice.

7.2 The Discretization Procedure

In order to simulate ϕ^4 theory on the lattice, we need to discretize Eqn. 7.7 to turn it into a "lattice action" (with the form of an energy) on a finite lattice with periodic boundary conditions. We'll work in two dimensions for simplicity, and extend our final result to four dimensions only at the end of our calculation.

First we'll write Eqn. 7.7 in two dimensions, and in a slightly more transparent form. Even though we'll think of our lattice as having one spatial dimension and one time dimension (i.e., we'll use notation like $\int dxdt$), we emphasize that we're working in a Euclidean space that behaves mathematically like two spatial dimensions. We have

$$S_E = \int dxdt \left[\frac{1}{2}\left(\frac{\partial\phi}{\partial t}\right)^2 + \left(\frac{\partial\phi}{\partial x}\right)^2 + \frac{1}{2}\mu_0^2\phi^2 + \frac{\lambda}{4}\phi^4\right]. \tag{7.8}$$

There are three steps to the discretization procedure: the discretization of the field ϕ; of the partial derivatives; and of the integral itself. The first is the simplest: instead of being a continuous function $\phi(x,t)$, ϕ_n $(0 \le n \le N)$ is now defined only at N lattice sites separated by lattice spacing a.

The question of discretizing the partial derivatives is somewhat more vexing. The simplest approach is to express them in terms of differences between nearest-neighbor lattice sites:

$$\frac{\partial\phi}{\partial x} = \frac{\phi(x+\frac{a}{2},t) - \phi(x-\frac{a}{2},t)}{a} \tag{7.9}$$

$$\left(\frac{\partial\phi}{\partial x}\right)^2 = \frac{\phi^2(x+\frac{a}{2},t) + \phi^2(x-\frac{a}{2},t) - 2\phi(x+\frac{a}{2},t)\phi(x-\frac{a}{2},t)}{a^2}. \tag{7.10}$$

Thus the kinetic term becomes

$$\begin{aligned}
&\frac{1}{2}\int dxdt \left[\left(\frac{\partial\phi}{\partial t}\right)^2 + \left(\frac{\partial\phi}{\partial x}\right)^2\right] \\
&\quad = \frac{1}{2a^2}\int dxdt \left[\phi^2\left(x,t+\frac{a}{2}\right) + \phi^2\left(x,t-\frac{a}{2}\right) + \phi^2\left(x+\frac{a}{2},t\right)\right. \\
&\quad \left. + \phi^2\left(x-\frac{a}{2},t\right) - 2\phi\left(x+\frac{a}{2},t\right)\phi\left(x-\frac{a}{2},t\right) - 2\phi\left(x,t+\frac{a}{2}\right)\phi\left(x,t-\frac{a}{2}\right)\right]
\end{aligned}$$

Since we're integrating over the whole lattice and use periodic boundary conditions, all of the squared terms will give the same result, so we can rewrite them in a slightly simpler form,

$$\frac{1}{2}\int dxdt\left[\left(\frac{\partial\phi}{\partial t}\right)^2+\left(\frac{\partial\phi}{\partial x}\right)^2\right]=\frac{1}{2a^2}\int dxdt\left[4\phi^2(x,t)-2\phi\left(x+\frac{a}{2},t\right)\phi\left(x-\frac{a}{2},t\right)\right.$$
$$\left.-2\phi\left(x,t+\frac{a}{2}\right)\phi\left(x,t-\frac{a}{2}\right)\right].$$

Though this nearest-neighbors approximation is the simplest discretization approach and the one used in our programs, it has its disadvantages. In particular, by ignoring more distant sites, such as the diagonals $\phi\left(x\pm\frac{a}{2},t\pm\frac{a}{2}\right)$, we introduce systematic errors about which we will have more to say in Section 7.5. For the moment we assume that the systematic effects of this approximation are negligible.

The final step is to discretize the integral itself by taking $dxdt\to a^2$ and $x_i\to an_x$, giving

$$\frac{1}{2}\int dxdt\left[\left(\frac{\partial\phi}{\partial t}\right)^2+\left(\frac{\partial\phi}{\partial x}\right)^2\right]\to\frac{1}{2}\sum_{n_x,n_y}\left[4\phi^2(an_x,an_t)-2\phi(a(n_x+1),an_t)\phi(an_x,an_t)\right.$$
$$\left.-2\phi(an_x,a(n_t+1))\phi(an_x,an_t)\right].$$

We can simplify the notation by summing over lattice sites n instead of worrying about the lattice spacing a. We'll also define i and j such that $\phi(a(n_x+1),an_t)\to\phi(n+i)$ and $\phi(an_x,a(n_t+1))\to\phi(n+j)$:

$$\frac{1}{2}\int dxdt\left[\left(\frac{\partial\phi}{\partial t}\right)^2+\left(\frac{\partial\phi}{\partial x}\right)^2\right]\to\sum_n\frac{1}{2}\left[4\phi^2(n)-2\phi(n)\phi(n+i)-2\phi(n)\phi(n+j)\right].$$

The final trick is to apply periodic boundary conditions again, writing

$$\begin{aligned}&4\phi^2(n)=2\phi^2(n)+\phi^2(n+i)+\phi^2(n+j),\\&\frac{1}{2}\int dxdt\left[\left(\frac{\partial\phi}{\partial t}\right)^2+\left(\frac{\partial\phi}{\partial x}\right)^2\right]\\&\quad\to\sum_n\frac{1}{2}\left[(\phi(n+i)-\phi(n))^2+(\phi(n+j)-\phi(n))^2\right].\end{aligned}\tag{7.11}$$

Note that the two squared terms in Eqn. 7.11 correspond to the two dimensions of the lattice. It is not hard to see, reviewing the steps in the discretization process that have led here, that this result can be generalized to d dimensions by writing

$$\frac{1}{2}\int dxdt\left[\left(\frac{\partial\phi}{\partial t}\right)^2+\left(\frac{\partial\phi}{\partial x}\right)^2\right]\to\sum_n\left\{\frac{1}{2}\sum_{\nu=1}^{d}\left(\phi(n+e_\nu)-\phi(n)\right)^2\right\},\tag{7.12}$$

where e_ν is the unit vector in the ν direction.

For the potential term, discretizing the integral is a bit simpler. Taking $dxdt \to a^2$ as before, we have

$$\int dxdt \left[\frac{1}{2}\mu_0^2\phi^2 + \frac{\lambda}{4}\phi^4\right] \to \sum_n \left[\frac{a^2\mu_0^2}{2}\phi^2(n) + \frac{a^2\lambda}{4}\phi^4(n)\right].$$

Note that $[a] = -1$ since a is a length. For two-dimensional ϕ^4 theory $[\mu_0^2] = [\lambda] = 2$, we can work with dimensionless "lattice parameters" by defining $\mu_L^2 = a^2\mu_0^2$ and $\lambda_L = a^2\lambda$.[7.1] However, this introduces a problem when we try to go to the continuum limit by taking the lattice spacing $a \to 0$. Since both μ_L^2 and λ_L depend directly on the square of the lattice spacing a^2, as $a \to 0$, both $\mu_L^2 \to 0$ and $\lambda_L \to 0$, which isn't interesting.

The solution to this problem is to consider the (still dimensionless) ratio $[\lambda/\mu^2]$. This single parameter, the critical coupling constant, characterizes the continuum ϕ^4 theory – taking the continuum limit reduces the number of independent dimensionless parameters from two to one. In Sections 7.4–7.5, we will discuss how we have used lattice simulations to calculate the critical coupling constant of continuum ϕ^4 theory in two dimensions.

In terms of the dimensionless lattice parameters μ_L^2 and λ_L, the potential term becomes

$$\frac{1}{2}\int dxdt \left[\phi^2\left(\mu_0^2 + \frac{\lambda}{2}\phi^2\right)\right] \to \sum_n \left[\frac{1}{2}\mu_L^2\phi^2(n) + \frac{\lambda_L}{4}\phi^4(n)\right]. \tag{7.13}$$

Adding the kinetic and potential terms, Eqns. 7.12 and 7.13, we obtain the discretized lattice action,

$$S_E = \sum_n \left\{\frac{1}{2}\sum_{\nu=1}^{d}\left(\phi(n+e_\nu) - \phi_n\right)^2 + \frac{1}{2}\mu_L^2\phi_n^2 + \frac{\lambda_L}{4}\phi_n^4\right\}, \tag{7.14}$$

where we have written $\phi(n) = \phi_n$. For lattice simulations, it is more convenient to return to one of the intermediate expressions above by expanding the first term. In two dimensions, $d = 2$, we'll write $e_1 = i$ and $e_2 = j$, which gives

$$\frac{1}{2}\sum_{\nu=1}^{2}\left(\phi(n+e_\nu) - \phi_n\right)^2 = \frac{1}{2}\sum_n \left(4\phi_n^2 - 2\left[\phi(n+i)\phi_n + \phi(n+j)\phi_n\right]\right).$$

It is easy to see that the second term is simply the same sort of sum over all pairs of neighboring sites $\sum_{\langle ij\rangle}$ that we encountered earlier in the Ising model (Section 5.1):

$$S_E^{(2)} = -\sum_{\langle ij\rangle}\phi_i\phi_j + \sum_n\left[\left(2 + \frac{1}{2}\mu_0^2\right)\phi_n^2 + \frac{1}{4}\lambda\phi_n^4\right] \tag{7.15}$$

in two dimensions. In four dimensions, the sum over nearest neighbors will still exactly equal the mixed terms in the sum $\sum_{\nu=1}^{d}$, while there will be twice as many ϕ_n^2. So the four-dimensional lattice action is

$$S_E^{(4)} = -\sum_{\langle ij\rangle}\phi_i\phi_j + \sum_n\left[\left(4 + \frac{1}{2}\mu_0^2\right)\phi_n^2 + \frac{1}{4}\lambda\phi_n^4\right]. \tag{7.16}$$

[7.1] We'll often omit the implicit $_L$ subscripts when discussing lattice simulations.

7.3 ϕ^4 Monte Carlo Algorithms

Now that we know it is theoretically possible to simulate ϕ^4 theory on the lattice like a classical statistical system, let's investigate the practical details of doing so. In our simulations we use the Metropolis algorithm and the Wolff cluster algorithm, which need to be modified slightly from the forms presented above in Chapter 5 in order to deal with the continuous variation of ϕ (as opposed to the discrete ± 1 values of Ising spins). For the Metropolis algorithm, we cannot change the acceptance probability, Eqn. 5.12, since it is what defines the algorithm (though the expression for the energy is different). However, instead of simply reversing the sign of ϕ at the site under consideration, we randomly select a new value ϕ_{new} from the range $\phi_{current} \pm 1.5$. This allows the magnitude of ϕ to vary continually as required by ergodicity. We chose the value 1.5 in an attempt to strike a balance between allowing ϕ to change meaningfully but not so much that we would end up with a large number of extreme changes likely to be rejected by the algorithm.[7.2]

To use the Wolff cluster algorithm on ϕ^4 theory, we simply applied it to the embedded Ising model obtained by treating all sites with positive ϕ as spin-up Ising spins and all those with negative ϕ as spin-down. That is, we form clusters of sites with either positive or negative ϕ, and then negate the values of ϕ at all the sites in the cluster (note that we do not change the magnitude of ϕ at any of the sites in the lattice). This technique was introduced (for the Swendsen-Wang cluster algorithm) by Brower ad Tamayo [9] and used for the Wolff algorithm specifically by Loinaz and Willey [32] as well as De *et al.* [16]. Considering only the signs of ϕ in this manner allows the Wolff cluster algorithm to proceed almost exactly as it does for the Ising model. However, since the lattice action of the lattice in ϕ^4 theory is more complex than the Hamiltonian of the Ising model, we need to alter the probability of adding a site to the cluster in order to make sure detailed balance is still satisfied.

We need to replay the analysis in Section 5.5 that resulted in Eqn 5.17 for the probability P_{add} of adding a site to the cluster. There we introduced m and n, which we can call the number of the number of "bonds" between parallel-spin pairs that will be created or broken by the flip, respectively, and therefore the number that will be broken or created by the reverse process. We denote by x_b a site bordering the cluster on the outside and by x_c denote the neighboring site in the cluster. If the sign of $\phi(x_b)$ is the same as that of the cluster, then x_b was considered for inclusion in the cluster but rejected with probability $\left(1 - P_{add}^{(x_b)}\right)$.

We can no longer assume P_{add} is constant at all sites, so instead of $g(\mu \to \nu) \propto (1 - P_{add})^n$ we have $g(\mu \to \nu) \propto \prod_{i=1}^{n} \left(1 - P_{add}^{(i)}\right)$ and

$$\frac{g(\mu \to \nu)}{g(\nu \to \mu)} = \frac{\prod_{i=1}^{n} \left(1 - P_{add}^{(i)}\right)}{\prod_{j=1}^{m} \left(1 - P_{add}^{(j)}\right)} = e^{-(E_\nu - E_\mu)} = e^{-\Delta E}.$$

We have placed the site indices i and j inside parentheses so they are not confused with

[7.2] It might be an interesting exercise to see how the overall efficiency of the simulation depends on the magnitude of potential changes allowed. We did not perform such an analysis.

exponents.

Although we no longer have the simple Ising expression for the energy difference $\Delta E = E_\nu - E_\mu$, since the magnitude of ϕ is not changed at any of the sites in the cluster all the quadratic and quartic terms in the expression for the energy, Eqn. 7.14, cancel out and we are left with only the contributions from nearest-neighbor interactions. From Eqn. 7.14, we see the energy will rise by $2\phi(x_b^{(i)})\phi(x_c^{(i)})$ for all n bonds that will be broken and will fall by $2\phi(x_b^{(j)})\phi(x_c^{(j)})$ for all m bonds that will be created. Thus

$$\Delta E = \sum_{i=1}^{n} 2\phi(x_b^{(i)})\phi(x_c^{(i)}) - \sum_{j=1}^{m} 2\phi(x_b^{(j)})\phi(x_c^{(j)})$$

$$\frac{\prod_{i=1}^{n}\left(1-P_{add}^{(i)}\right)}{\prod_{j=1}^{m}\left(1-P_{add}^{(j)}\right)} = e^{-\Delta E} = \exp\left[-\sum_{i=1}^{n} 2\phi(x_b^{(i)})\phi(x_c^{(i)}) + \sum_{j=1}^{m} 2\phi(x_b^{(j)})\phi(x_c^{(j)})\right]$$

$$\frac{\prod_{i=1}^{n}\left(1-P_{add}^{(i)}\right)}{\prod_{j=1}^{m}\left(1-P_{add}^{(j)}\right)} = \left(\prod_{i=1}^{n} e^{-2\phi(x_b^{(i)})\phi(x_c^{(i)})}\right)\left(\prod_{j=1}^{m} e^{2\phi(x_b^{(j)})\phi(x_c^{(j)})}\right)$$

$$\prod_{i=1}^{n}\frac{1-P_{add}^{(i)}}{e^{-2\phi(x_b^{(i)})\phi(x_c^{(i)})}} = \prod_{j=1}^{m}\frac{1-P_{add}^{(j)}}{e^{-2\phi(x_b^{(j)})\phi(x_c^{(j)})}}$$

It is clear that equality will hold if we set

$$P_{add} = 1 - e^{-2\phi(x_b)\phi(x_c)}. \tag{7.17}$$

P_{add} now depends on the values of ϕ at the particular sites under consideration, but its expression is still pleasingly simple.

Our simulations used a mixture of the Metropolis and Wolff algorithms in these forms. Specifically, each measurement of the observables was performed after sweeping through the lattice five times using the Metropolis algorithm and then performing a single Wolff cluster flip. In all our future discussions we will refer to this combination of five Metropolis sweeps plus one Wolff cluster flip as an 'iteration'. By 'Metropolis sweep' we simply mean that the single-spin Metropolis algorithm is performed one time *for* each site in the lattice, but not necessarily one time *at* each site in the lattice. We run each Metropolis update on a randomly-selected site and do not guarantee that every site on the lattice is selected at some point during each 'sweep'. The terminology may seem slightly deceptive, but is both simple and standard (cf. Newman and Barkema [40, Chap. 3]).

The ergodicity of this mixture of algorithms is guaranteed by the ability of the Metropolis algorithm to 'shake up' the values of ϕ at all the sites into any given configuration through enough intermediate steps. The addition of the Wolff cluster helps reduce the autocorrelation time (cf. Figs. 5.10 and 5.11) and break the system out of any metastable state it may fall into.[7.3]

[7.3]It might be an interesting exercise to see how critical slowing down and the autocorrelation time depend on the ratio of Metropolis sweeps to Wolff cluster flips. We did not perform such an analysis.

Since cluster algorithms are (despite their relatively common use and firm analytical foundations) not yet as well-trusted as the classic Metropolis algorithm, we performed our small-lattice ($L \leq 128$) ϕ^4 simulations twice, once using just the Metropolis algorithm and once using this Metropolis/Wolff mixture. We found that the results of both simulations agreed very well, though the autocorrelation times of those not using the Wolff cluster algorithm were typically at least an order of magnitude greater than those using the mixture (see Section 5.5). In Fig. 7.1 we show typical results from these small-lattice simulations, illustrating the agreement between them. All data points agree well within uncertainty, though it is clear that using the Wolff algorithm dramatically decreases the uncertainty; error bars on the mixed Metropolis/Wolff results are barely visible.

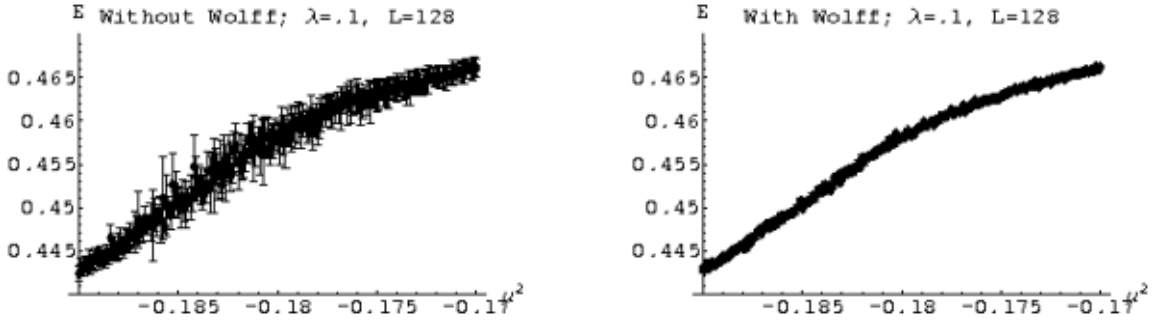

Figure 7.1: Energy for ϕ^4 simulations at $\lambda = 0.1$ and $L = 128$ using only Metropolis (left) and mixed Metropolis/Wolff (right)

Additionally, in all of our work exploring the ϕ^4 phase space and calculating the critical coupling constant, we used helical boundary conditions, for reasons discussed in Section 3.3. Since helical boundary conditions are not as standard as periodic boundary conditions, we also re-ran our small-lattice ($L \leq 128$) ϕ^4 simulations using periodic boundary conditions and verified that the results obtained with both schemes agreed, as shown above in Fig. 3.6.

7.4 Phase Transition Indicators

Our first lattice simulations of ϕ^4 theory investigated its phase transitions, which are similar to those of the previously-studied Ising model. The goal of these calculations was to verify the value of the critical coupling constant $[\lambda/\mu^2]_{crit}$ for two-dimensional continuum ϕ^4 theory published by Loinaz and Willey [32] several years ago. We used three indicators to identify the phase transition and critical point: the peak in the susceptibility mentioned above in the context of the Ising model; the bimodality, a measure of the brokenness of the system; and the fourth-order cumulant defined by Binder [7]. We will discuss each of these indicators in turn in the following three subsections.

7.4.1 Susceptibility

For one of our indicators, we use the susceptibility peak introduced in Section 5.1 above. The susceptibility of ϕ^4 theory behaves much like that of the Ising model, as expected since they are members of the same universality class with second-order phase transitions. As shown in Fig. 7.2, the ϕ^4 susceptibility has a peak at the critical point, which gets sharper as the size of the lattice on which the simulation takes place increases (Figs. 5.8 and 5.9). The specific heat also exhibits a similar peak, but since it diverges logarithmically (cf. Section 5.2) its peak is not as sharp as that of the susceptibility, so we did not use it as a phase transition indicator.

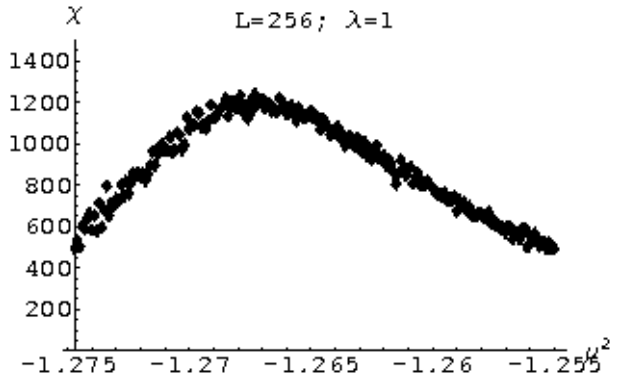

Figure 7.2: Susceptibility vs. temperature for ϕ^4 theory simulations on a 256^2 lattice at $\lambda = 1$

Such peaks are familiar beasts statistically. They can be modeled as Gaussians,

$$\chi \propto \exp\left[\frac{-(x-X)^2}{2\sigma^2}\right] \tag{7.18}$$

where X is the central value and σ is the standard deviation. It is trivial to find the maximum using a Mathematica script or similar means. σ can be extracted by considering the half-maxima $y_\pm$ of the peak, where

$$\frac{\chi(y)}{\chi(X)} = \exp\left[\frac{-(y_\pm - X)^2}{2\sigma^2}\right]\exp\left[\frac{(X-X)^2}{2\sigma^2}\right] = \exp\left[\frac{-(y_\pm - X)^2}{2\sigma^2}\right].$$

Simple rearrangement gives $y_\pm = X \pm \sigma\sqrt{2\log(2)}$. We define the full width at half-maximum FWHM as

$$FWHM = y_+ - y_- = 2\sigma\sqrt{2\log(2)} \approx 2.35\sigma. \tag{7.19}$$

Thus measuring the FWHM of the susceptibility peak immediately results in a measure of the standard deviation σ.

We actually overestimated a bit and defined our uncertainties on the susceptibility critical point estimates as exactly half of the FWHM. We also conservatively defined our half-maximum as one point beyond the point furthest from the peak with value greater than half the maximum, as opposed to the point closest to the peak with value less than

half the maximum. The specific details of our analysis are presented in its Mathematica implementation in Code Snippet C.13.

There is an additional complication relating to the susceptibility: when we calculate it, $\chi = N(\langle\bar{\phi}^2\rangle - \langle\bar{\phi}\rangle^2)$,[7.4] we actually need to use the absolute value of the volume-average of ϕ in the second term. That is,

$$\chi = N(\langle\bar{\phi}^2\rangle - \langle|\bar{\phi}|\rangle^2). \tag{7.20}$$

This is due to the fact that even in the broken phase, the sign of ϕ may change over the course of the simulation. In fact, this is almost guaranteed by our use of the Wolff cluster algorithm, which in the broken phase will generally flip nearly the entire cluster back and forth. Thus we end up with $\langle\bar{\phi}\rangle = 0$, even in the broken phase where we know $\langle|\bar{\phi}|\rangle \neq 0$.

7.4.2 Bimodality

Bimodality is an intuitive and relatively straightforward quantization of the visual picture of spontaneous symmetry breaking. We calculate it by binning all the values of $\bar{\phi}$ measured over the course of the simulation (after reaching equilibrium) into a histogram. The data is sorted into an odd number of bins so that there is a central bin corresponding to $\bar{\phi} \approx 0$. The bimodality is then defined as one minus the ratio of this central bin to the maximum value of the histogram,

$$B = 1 - \frac{n_0}{n_{max}}. \tag{7.21}$$

This definition is designed to make the bimodality a measure of the brokenness of the system. Simulations in the symmetric phase have a higher ratio and thus a lower bimodality (with a minimum of 0), while simulations in the broken phase result in a lower ratio and a correspondingly higher bimodality (with a maximum of 1).

Figures 7.3 and 7.4 show the $\bar{\phi}$ histograms for four different values of μ_L^2. In the left side of Fig. 7.3, the system is in the symmetric phase and the bimodality is zero, since the central bin is the maximum of the histogram. In the right side the system is strongly broken and the central bin is almost empty, leading to a bimodality near 1. Fig. 7.4 includes histograms for systems near the phase transition.

Although our definition of bimodality is pleasingly intuitive, its very simplicity presents some analytical difficulties. In particular, it is not immediately obvious what value of bimodality corresponds to the critical point itself, nor what values can be adopted as bounds on the uncertainty. There is no convenient central peak or FWHM that can be extracted from the bimodality curve shown in Fig. 7.5. We adopted the ad hoc rules that a bimodality of $B = 0.5$ indicates the critical point, while $B > 0.95$ and $B < 0.1$ serve as 2σ (95% confidence level) bounds. Code Snippet C.13 in Appendix C, below,

[7.4] We use $\bar{\phi}$ to denote the volume-average of ϕ over all lattice sites at a particular measurement, and $\langle\bar{\phi}\rangle$ to denote the time-average of $\bar{\phi}$ across all the measurements performed over the course of the simulation.

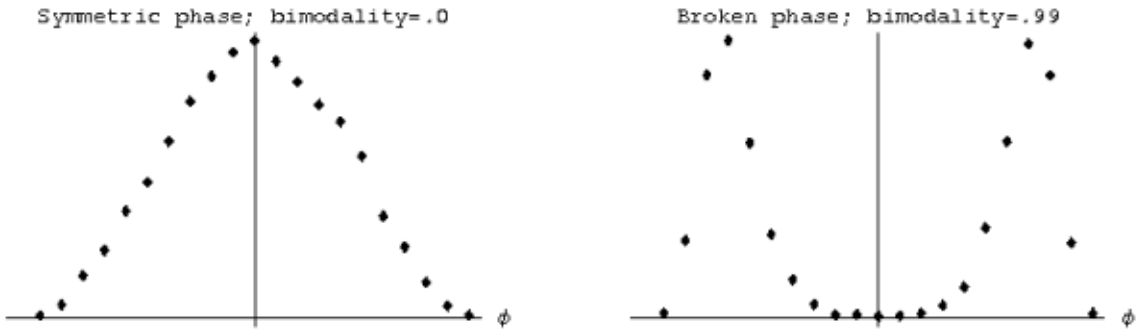

Figure 7.3: $\bar{\phi}$ histograms in symmetric and broken phases $L = 32$, $\lambda = .05$, $\mu_L^2 = -.075$ (left) and $\mu_L^2 = -.11$ (right)

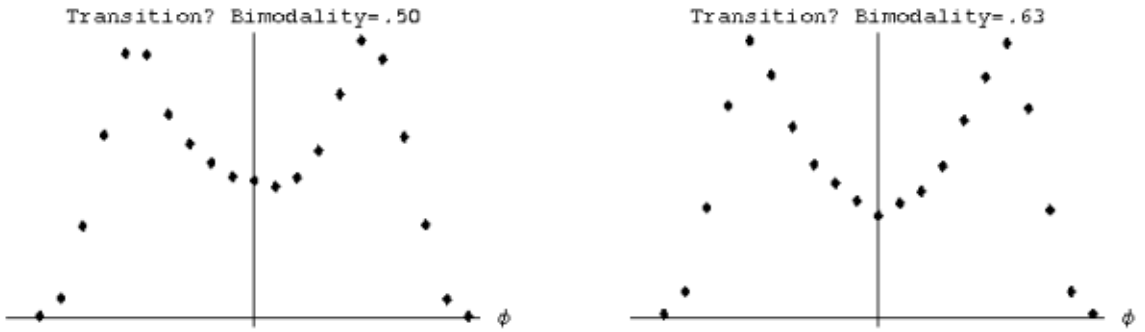

Figure 7.4: $\bar{\phi}$ histograms around the phase transition: $L = 32$, $\lambda = .05$, $\mu_L^2 = -.0928$ (left) and $\mu_L^2 = -.0957$ (right)

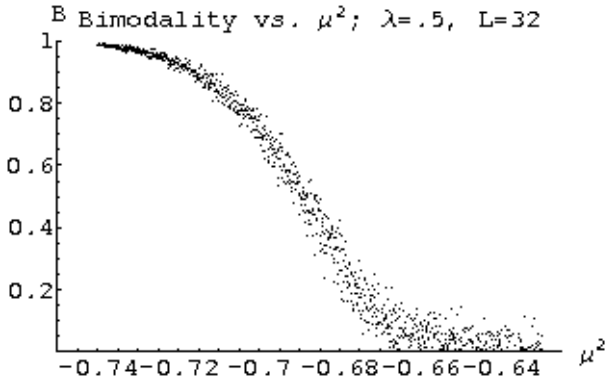

Figure 7.5: Bimodality vs. μ_L^2 for $L = 32$ and $\lambda = .5$

shows the Mathematica script we used to extract these estimates and bounds from our raw data.

As shown in Fig. 7.6, our rules led to identifications of the critical point that agreed within uncertainty with those made using the susceptibility peak. Note that the uncertainties in the points determined using bimodality are significantly smaller than the corresponding susceptibility uncertainties (the scales of both charts are identical). However, there is a general trend, not readily observable in Fig. 7.6, that the bimodality estimate is slightly higher than that of the susceptibility. It is easy to calculate the value of bimodality that corresponds to the susceptibility peak, and when we do so, we find out that $B = 0.5$ does not actually match the peak. Instead, the susceptibility peak occurs at $B = .63 \pm .11$ in two dimensions and $B = .78 \pm .18$ in four dimensions. Fig. 7.4 compares the histograms for two-dimensional systems with $B = .63$ and $B = .5$; it is not immediately obvious which better corresponds to the critical point.

Figure 7.6: Critical μ_L^2 determined from susceptibility (left) and bimodality (right) for $\lambda = .7$

While we could use this analysis as motivation to change our ad hoc rules, glibly doing so would raise a difficulty. It would make the bimodality observable dependent on the susceptibility observable, and no longer an independent indicator of the phase transition. Since the critical μ_L^2 values calculated using the bimodality agree with those calculated using the susceptibility peak – even though the critical bimodality $B = 0.5$ doesn't match the location of the susceptibility peak itself – we retained our initial rules.

An additional practical difficulty with the bimodality data is that it generally has more jitter than the other observables, especially on smaller lattices and especially in the broken phase. This can be seen clearly in Fig. 7.5, where the bimodality curve broadens considerably for higher μ_L^2, with neighboring data points often jumping from $B = 0$ as high as $B = 0.2$. The larger uncertainty in the value of bimodality corresponding to the susceptibility peak in four dimensions is likely due to the jitter caused by the smaller lattices that have to be used when working in four dimensions.

We addressed this problem by considering the running average of the bimodality instead of the pure data itself. Taking the three-term running average of the data in Fig. 7.5 produces the curve shown in Fig. 7.7, which has the same qualitative shape but is noticeably sharper, simplifying its analysis. The data presented above for the values of bimodality corresponding to the susceptibility peaks uses this smoothed bimodality, as does Fig. 7.6 and all further work.

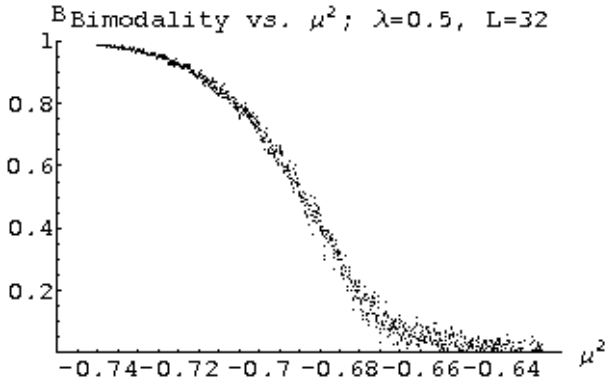

Figure 7.7: Smoothed bimodality vs. μ^2 for $L = 32$ and $\lambda = .5$

7.4.3 Fourth-order Cumulant

The final indicator we use to identify the critical point is the fourth-order cumulant, commonly known as a 'Binder cumulant'[7.5] due to its definition by Binder [7]:

$$U = 1 - \frac{\langle \bar{\phi}^4 \rangle}{3\langle \bar{\phi}^2 \rangle^2}, \tag{7.22}$$

with $\bar{\phi}$ defined as above. Since $\bar{\phi}^2$ and $\bar{\phi}^4$ are always positive, taking the absolute value isn't necessary.

U is useful since it has a fixed point at the critical point, as shown below in Fig. 7.8. In the broken phase, U asymptotically approaches $\frac{2}{3}$, while for systems in the symmetric phase, U approaches zero. Fig. 7.9 illustrates this behavior for both small $L = 32$ and large $L = 1024$ lattices at $\lambda = .5$. The key is that the cumulants calculated on larger lattices make a sharper transition, as can be very clearly seen from Fig. 7.10, which plots the same data as Fig. 7.9, but uses the same scale in each chart. As a result of these sharper transitions, there is a region on the broken side of the critical point where the larger lattices all have larger U than the smaller lattices, and a similar region on the symmetric side where the larger lattices all have smaller U. The symmetric case is easy to observe in Fig. 7.8; the broken case is obscured by the convergence toward $\frac{2}{3}$ but still holds. The point where all the cumulants cross is the critical point.

To analyze our data, we restricted our attention to the largest three lattices for simplicity (shown in Fig. 7.11). We defined the lower bound (one σ) on the critical point as the value of μ^2 below which simulations on larger lattices always have larger Binder cumulants and similarly defined the upper bound as the value above which simulations on larger lattices always have smaller cumulants (at least until the cumulants are all randomly fluctuating near zero). We then identified the critical point by selecting the point between the two bounds at which the separation between the cumulants is smallest. The fact that the critical point is constrained to be between the bounds is critical, since as all the cumulants approach $\frac{2}{3}$ in the broken phase the separation between them approaches zero (thus our estimate often ends up near the lower bound). The specific

[7.5] See for instance Sun [58].

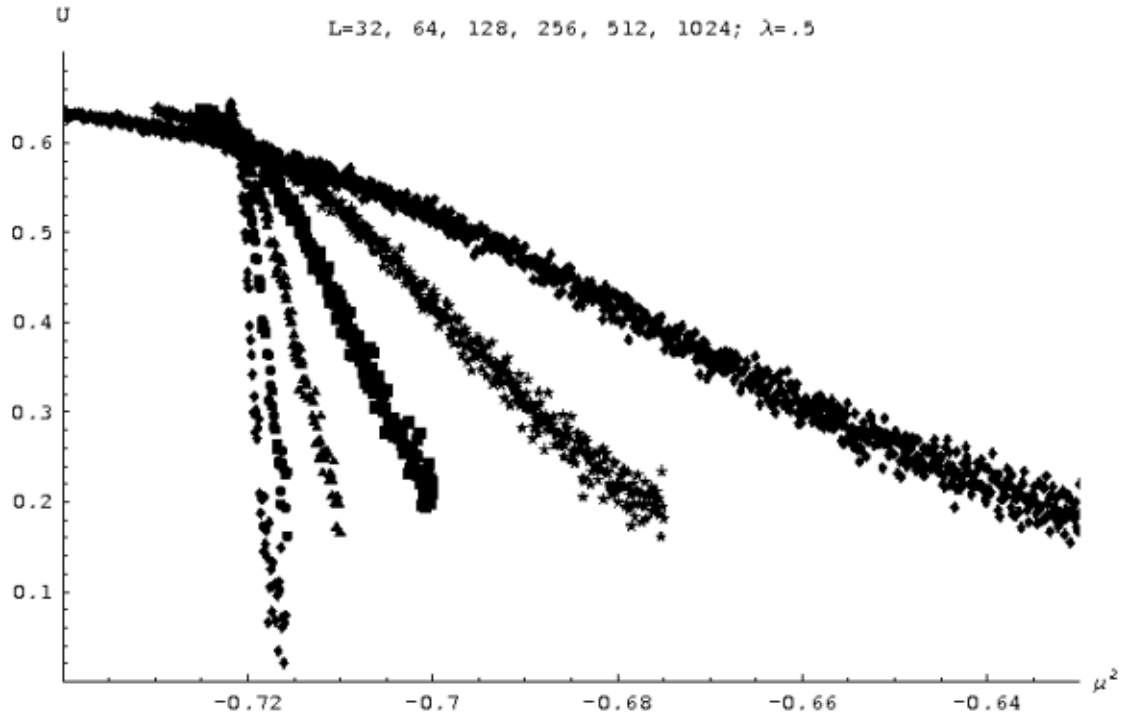

Figure 7.8: Binder cumulant vs. μ^2 for $\lambda = .5$ and $L = 32$, 64, 128, 256, 512 and 1024

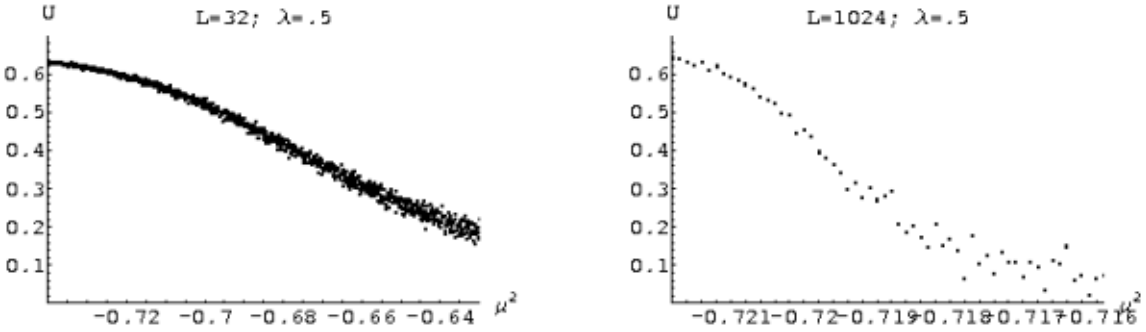

Figure 7.9: Binder cumulant vs. μ^2 for $\lambda = .5$ and $L = 32$ (left) and 1024 (right)

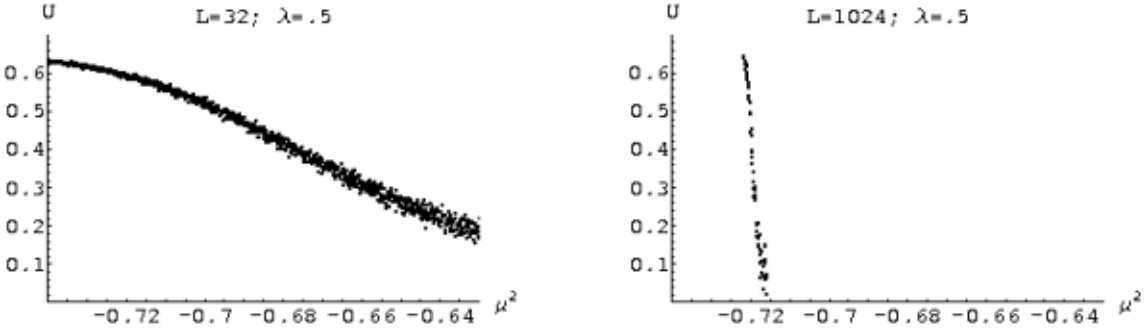

Figure 7.10: Binder cumulant vs. μ^2 for $\lambda = .5$ and $L = 32$ (left) and 1024 (right), with the same scale

details of this analysis can be found in its Mathematica implementation, included below as Code Snippet C.14.

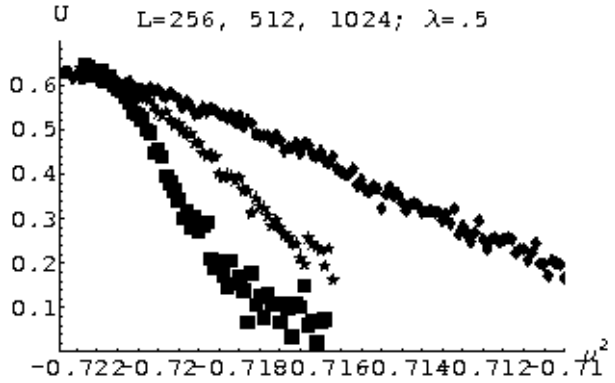

Figure 7.11: Binder cumulant vs. μ^2 for $\lambda = .5$ and $L = 256$, 512 and 1024

7.5 The Phase Transition Line and the Critical Coupling Constant

We can now discuss the details of our simulations investigating ϕ^4 theory and present our results. In this section we will explain our analyses of the phase transition line of the theory in two and four dimensions and our calculation of the two-dimensional critical coupling constant. In the next section we will turn to solitons and describe the simulations and analysis performed to calculate their masses in two-dimensional ϕ^4 theory.

7.5.1 Simulations and Results in Two Dimensions

We initially planned to focus our work to solitons in ϕ^4 theory, but before turning our attention on solitons themselves, we decided to gain familiarity with lattice simulations of ϕ^4 theory by replaying an earlier analysis found in the literature. To our surprise, our results, though consistent with those in the literature, revealed a subtlety that had previously been invisible. Though subtle, this effect had serious repercussions on our final results. In this subsection we will explain the initial motivation for our study, early developments that prompted further investigations, and finally the details of our simulations and results.

Motivation and Early Developments

We decided to investigate phase transitions in ϕ^4 theory mainly to gain familiarity with the theory itself and with methods for simulating it on the lattice. Our initial goal was to reproduce (perhaps with slightly increased precision) the results of Loinaz and

Willey [32], who used lattice methods to calculate a two-dimensional critical coupling constant of $[\lambda/\mu^2]_{crit} = 10.26^{+.08}_{-.04}$ in continuum ϕ^4 theory. Their results are shown below in Fig. 7.12.

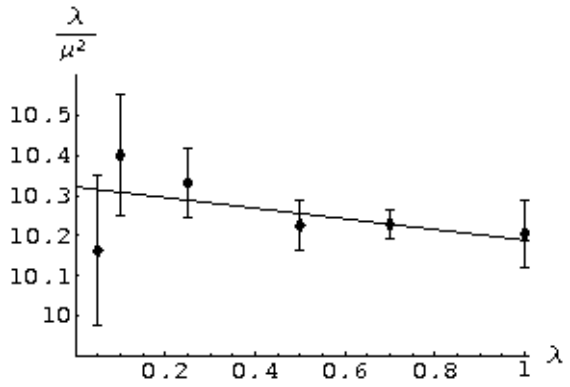

Figure 7.12: Two-dimensional critical coupling results from Loinaz and Willey [32]

Numerical methods are required to calculate the critical coupling constant because no satisfactory analytic approximations have yet been discovered. Loinaz and Willey compare their results to those of several potential approximation schemes. The simplest approach, considering the one-loop effective potential, predicts a critical coupling constant of $[\lambda/\mu^2]_{crit} = 6.6$. More complicated techniques make predictions scattered throughout the range $3.8 \leq [\lambda/\mu^2]_{crit} \leq 10.27$. Unfortunately, the Gaussian effective potential approach that agrees best with Loinaz and Willey's numerical result incorrectly predicts a first-order phase transition.

We imitated Loinaz and Willey by performing simulations at each of $\lambda = 1.0$, 0.7, 0.5, 0.25, 0.1 and 0.05 with square lattices of size $L = 32$, 64, 128, 256 and 512.[7.6] In addition, we expanded the simulations to include an additional data point, $\lambda = 0.01$, and took advantage of our modern computational resources to perform simulations on lattices of size $L = 1024$ for all λ. This greater amount of data and precision allowed us to reduce significantly the uncertainties on the $[\lambda/\mu^2]$ data points. Unfortunately, as shown in Fig. 7.13, this resulted in data that no longer agreed fit well along a linear regression.

There were a limited number of possibilities: either our simulations were flawed, the behavior of $[\lambda/\mu^2]$ was actually nonlinear in λ, or there was the vain hope that taking additional data would somehow wrench the data into a linear fit. The first possibility seemed unlikely. Our simulations had been performed completely independently of Loinaz and Willey's, yet still produced results that agreed very well with theirs except at low λ, as shown in Fig. 7.14. Since we did not have enough data to conclude that a linear model was completely ruled out, the only solution was to perform more simulations. So our study of solitons was postponed, and the initial exercise intended merely to familiarize us with ϕ^4 theory and lattice quantum field theory was no longer so simple.

[7.6] We will go further into the details of our simulations below.

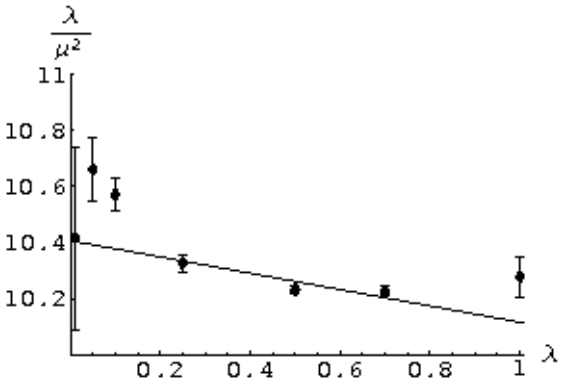

Figure 7.13: Preliminary two-dimensional critical coupling results as of December 2005

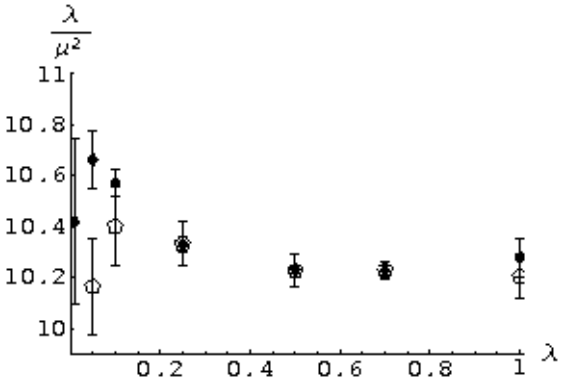

Figure 7.14: Comparison of preliminary results (solid points) to those of Loinaz and Willey [32] (hollow pentagons)

Simulations and Results

In order to improve upon these preliminary results, we added two more data points at $\lambda = 0.03$ and $\lambda = 0.02$ and performed additional large-lattice ($L = 600$, 1200) simulations at $\lambda = 0.05$, 0.03, 0.02 and 0.01, small values that have a strong impact on the continuum limit. In the end, then, our investigations of the ϕ^4 phase transition in two dimensions involved simulations of the system at nine different values of the coupling $\lambda = 1.0$, 0.7, 0.5, 0.25, 0.1, 0.05, 0.03, 0.02 and 0.01, performed on square lattices of length $L = 32$, 64, 128, 256, 512 and 1024, with additional simulations at $L = 600$, 1200 for $\lambda = 0.05$, 0.03, 0.02 and 0.01.

Starting from a random initial state, each simulation performed 16384 iterations for equilibration and an additional 16384 for statistics, except for the largest $L = 1024$ and 1200 lattices, on which simulations performed 8192 iterations for equilibration and an additional 8192 for statistics. We calculated the autocorrelation time τ for each simulation, using the method discussed above in Section 2.2 and implemented in the C++ code presented below in Code Snippet C.9. In general, the autocorrelation times were of order 10 iterations. Curiously, the autocorrelation times were largest for small lattices with low λ. The maximum autocorrelation times were of order 100 iterations, for $\lambda = 0.01$ on 32^2 and 64^2 lattices. Regardless, our simulations always equilibrated for at least 100 τ and took at least 100 statistically independent measurements. As a result statistical uncertainties are very small and overall uncertainty is dominated by systematic effects, in particular the effects of using finite lattices ("finite-size effects").

For each λ, we scanned through μ_L^2 in order to find the critical points and uncertainties predicted by both the bimodality and susceptibility, determining them as discussed above in Section 7.4. We used an iterative procedure to minimize the number of simulations that needed to be performed to locate the critical value of μ_L^2. After a broad initial scan on small 32^2 lattices we varied μ_L^2 on lattices with $L = 2^{n+1}$ only over the range given by the critical μ_L^2 and standard deviation calculated using $L = 2^n$ lattices. Although for some of the largest lattices we needed to perform additional simulations to locate the bounds on the critical point for either the susceptibility, bimodality or Binder cumulant, this method worked well for the most part.

After using the three indicators discussed above to determine critical points and uncertainties for each lattice size L, we performed linear regressions using Mathematica to determine the $L \to \infty$ ($L^{-1} \to 0$) limit. We did this separately for both the estimates from the susceptibility data and those from the bimodality data, as shown above in Fig. 7.6. The results of these regressions are shown in the first two columns of Table 7.1, with the results of the Binder cumulant calculations in the third column. Note that the data for the three indicators listed in Table 7.1 all agree within uncertainties, which were higher for the Binder cumulant since it was calculated with a single measurement as opposed to a regression to the infinite-volume limit.

The first column of Table 7.2 contains the unrenormalized ("bare") critical μ_L^2 for each value of λ – simply the weighted average of the critical values from Table 7.1. These values define the phase transition line of the bare theory, shown in Fig. 7.15 (with error

Table 7.1: Critical points determined from each phase transition indicator

λ	**Susceptibility**	**Bimodality**	**Cumulant**
1.00	-1.27233(16)	-1.27258(10)	-1.27260(45)
0.70	-0.95153(25)	-0.95152(7)	-0.95180(40)
0.50	-0.72080(11)	-0.72131(9)	-0.72130(30)
0.25	-0.40346(18)	-0.40373(6)	-0.40390(20)
0.10	-0.18424(11)	-0.18432(9)	-0.18430(20)
0.05	-0.10060(5)	-0.10071(4)	-0.10100(35)
0.03	-0.06410(4)	-0.06414(5)	-0.06420(15)
0.02	-0.04465(3)	-0.04468(5)	-0.04500(30)
0.01	-0.02397(6)	-0.02399(5)	-0.02410(10)

bars too small to be visible). We won't dwell on the bare phase transition line here, since we have renormalized the theory, as discussed above in Section 6.6. The renormalized critical μ^2 values in the second column of Table 7.2 were calculated using Eqn. 6.65, which we repeat here for convenience:

$$\mu^2 = \mu_0^2 + 3\lambda \int_0^\infty e^{-\mu^2 t} \left[e^{-2t} I_0(2t)\right]^2 dt, \tag{7.23}$$

where I_0 is the modified Bessel function of the first kind and our lattice parameter μ_L^2 plays the role of μ_0^2. Eqn. 7.23 was evaluated using Mathematica's `FindRoot` command; we then checked the result by substituting it back into the original expression. Because Eqn. 7.23 is highly nonlinear, we determined the uncertainties on the renormalized μ^2 by calculating the renormalized values corresponding to $\mu_L^2 \pm \sigma$ and treating them as upper and lower bounds on μ^2. Uncertainties on the critical coupling constants in the final column of Table 7.2 were calculated by simple error propagation.

Table 7.2: Points on the two-dimensional phase transition line for various λ_L

λ	**Critical** μ_L^2	**Critical** μ^2	$[\lambda/\mu^2]_{crit}$
1.00	-1.27251(16)	0.097321(46)	10.275(5)
0.70	-0.95152(16)	0.068464(45)	10.224(7)
0.50	-0.72111(11)	0.048887(32)	10.228(7)
0.25	-0.40371(9)	0.024179(26)	10.339(11)
0.10	-0.18428(8)	0.009477(23)	10.552(26)
0.05	-0.10067(12)	0.004679(33)	10.686(76)
0.03	-0.06412(5)	0.002794(15)	10.737(59)
0.02	-0.04466(10)	0.001870(28)	10.695(163)
0.01	-0.02400(4)	0.000931(12)	10.739(138)

Our results (Table 7.2 and Figs. 7.16 through 7.19) make the nonlinear relationship between $[\lambda/\mu^2]_{crit}$ and λ undeniable. Fig. 7.16 includes a clearly unacceptable linear regression; as shown in Table 7.3, the variance per degree of freedom of this fit is nearly 50.

Motivated by the general pattern in our data points, we performed nonlinear regressions in Mathematica to fit the data to expressions containing logarithmic terms.

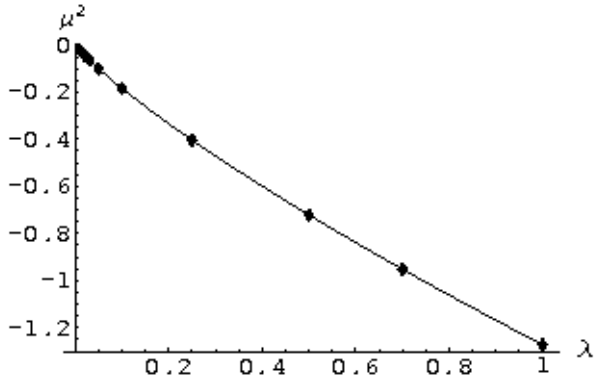

Figure 7.15: The bare ϕ^4 phase transition line in two dimensions: critical μ_L^2 vs. λ with nonlinear regression including terms up to $\lambda^2 \log[\lambda]$

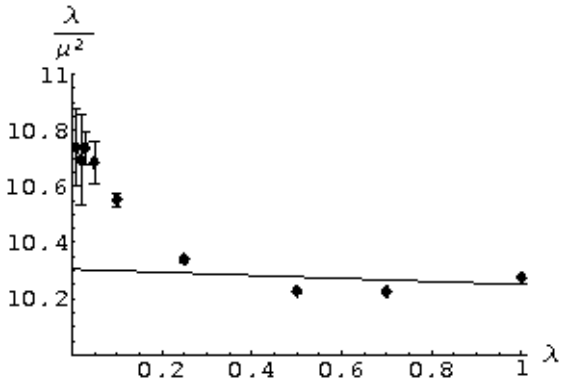

Figure 7.16: λ/μ^2 vs. λ with linear regression

The third column of Table 7.3 shows the results of fits involving a $\lambda \log[\lambda]$ term in addition to the constant and linear terms, while the fourth column adds a $\lambda^2 \log[\lambda]$ term, for a total of four terms. For each model, we performed fits for both the critical coupling constant $[\lambda/\mu^2]_{crit}$ in terms of λ (shown in Fig. 7.17) as well as μ^2_{crit} itself in terms of λ. All the μ^2_{crit} regressions look identical to the naked eye, so we only present one: Fig. 7.18 is the four-term regression (its error bars are too small to be visible). Fig. 7.15 also includes a fit of the bare critical μ_L^2 to the four-term regression; the nonlinearity in the bare phase transition line is even more pronounced than that in the renormalized line.

There was a slight complication: we received remarkably different results depending on whether we were performed the regressions in Mathematica 5.0 on Linux or in Mathematica 5.2 on Windows. In particular, in Linux the results for the $[\lambda/\mu^2]_{crit}$ and μ^2_{crit} regressions were different, while on Windows, both regressions produced nearly identical results. As shown in Table 7.3, we have adopted the results from Mathematica 5.2 on Windows. We trust this data more because it is from a newer version of the software operating under the operating system for which it was designed. We are not certain whether the difference in version or in operating system (or both) is the cause of this puzzling behavior. For reference we include the full Mathematica output for all of the regressions we performed for $[\lambda/\mu^2]_{crit}$ in Appendix D below.

Although the three-term regression given by Windows has a reasonable variance per degree of freedom of 1.26, the fit of the four-term regression is nearly perfect. Unfortunately, the two regressions produce rather different results of 10.774(31) and 10.874(17), respectively. Accordingly, we have adopted as our final result $[\lambda/\mu^2]_{crit} = 10.85^{+.03}_{-.08}$, which is consistent with both regressions.

Table 7.3: Continuum coupling constants from various fits of the two-dimensional data

	$c_1 + c_2\lambda$	$\cdots + c_3\lambda \log[\lambda]$	$\cdots + c_4\lambda^2 \log[\lambda]$
$[\lambda/\mu^2]_{crit}$ vs. λ	10.305(65) (Var = 47.8)	10.774(31) (Var = 1.26)	10.874(17) (Var = 0.13)
μ^2 vs. λ	10.243(19) (Var = 28.3)	10.775(31) (Var = 1.27)	10.874(17) (Var = 0.13)

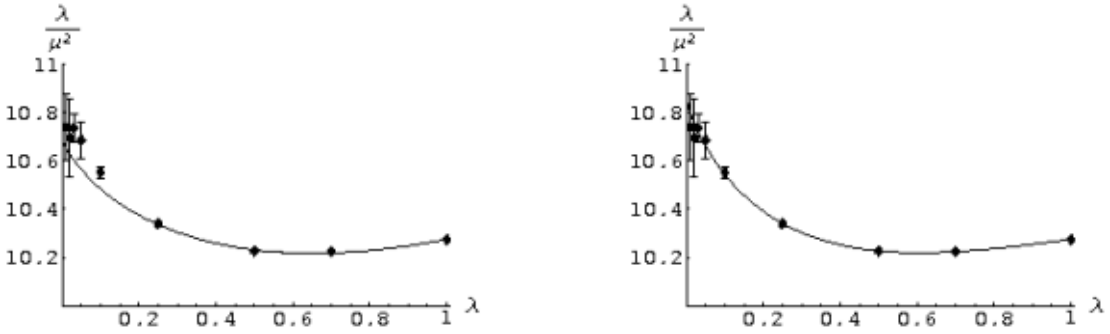

Figure 7.17: λ/μ^2 vs. λ with nonlinear regressions including $c_3\lambda \log[\lambda]$ (left) and $c_4\lambda^2 \log[\lambda]$ (right)

In a sense, this logarithmic dependency is not unexpected. Recall that when we discretized the action in Section 7.2, we neglected the contributions of all but nearest-

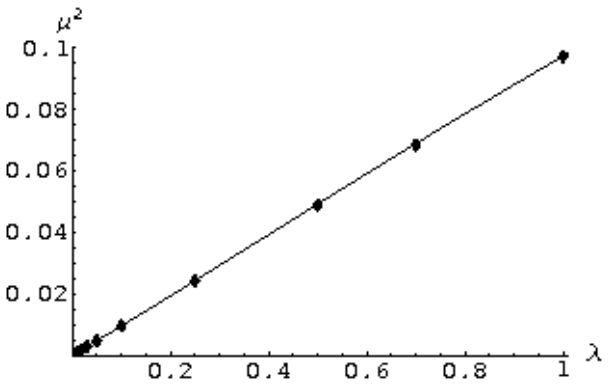

Figure 7.18: μ^2 vs. λ with nonlinear regression including $c_4\lambda^2 \log[\lambda]$

neighbors to the discretized form of the kinetic term $\left(\frac{\partial\phi}{\partial t}\right)^2 + \left(\frac{\partial\phi}{\partial x}\right)^2$. We remarked at the time that this approximation introduced systematic effects, but glibly assumed that they would be negligible. Our results suggest that this assumption is incorrect.

Unfortunately, it is very difficult to determine analytically these systematic effects. We are actively attempting such calculations, but we do not yet know whether they will prove tractable.

As a final coda, we note that even though our data decisively rules out a simple linear regression to the continuum limit, it is still largely consistent with Loinaz and Willey's results. Fig. 7.19 shows that all data points agree within uncertainty except for those at $\lambda = 0.05$, where the points are 2.75σ apart. Their linear fit agrees perfectly with their data, and it was only our additional data and resulting increase in precision that allowed us to observe these higher-order effects.

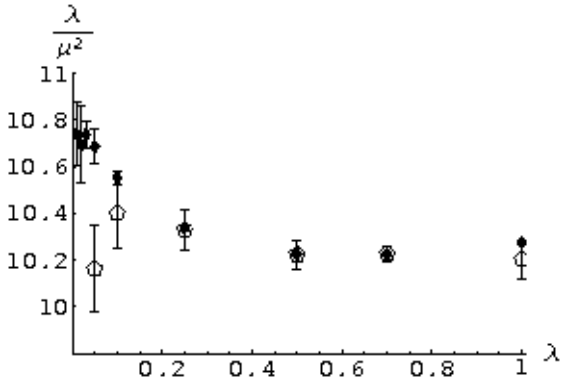

Figure 7.19: A comparison of our results (solid points) to those of Loinaz and Willey [32] (hollow pentagons)

7.5.2 Simulations and Results in Four Dimensions

Our exploration of the ϕ^4 theory in four dimensions followed a similar trajectory as our work in two dimensions. We began to study the four-dimensional theory expecting

to obtain a relatively straightforward result, but subsequently encountered complications which forced a reconsideration of our goals and methods. In this subsection we will explain the initial motivation for our work, early developments that prompted a change of course, and finally the details of our simulations and results.

Motivation and Early Developments

We were initially motivated to extend our studies of the phase structure of ϕ^4 theory into four dimensions by a recent disagreement in the literature between Balog *et al.* [3] and Stevenson [57]. Stevenson disputed Balog *et al.*'s conclusion that simulation data of four-dimensional ϕ^4 theory and the Ising model was "in perfect agreement with the conventional" wisdom obtained through analytic studies, a conclusion that disagrees with an earlier analysis by Stevenson's collaborators Cea, Consoli and Cosmai [11] (who themselves responded separately [12]). In particular, Stevenson claimed that "the change ['step'] in the wavefunction-renormalization constant $\hat{Z}_R$ across the phase transition is significantly greater than predicted." $\hat{Z}_R$ is similar to our field renormalization factor Z_ϕ introduced above in Section 6.6.4; using the action given in Eqn. 7.24, $\hat{Z}_R = 2\kappa Z_\phi$. Our hope was to resolve the conflict by directing our attention and that of Amherst's new scientific computing cluster toward the relatively limited issue under dispute.

We began to take data using the parameterization of the ϕ^4 Lagrangian favored by Balog *et al.*, which we present here in the form of the corresponding discretized lattice action:

$$S_E = \sum_x \left[-2\kappa \sum_{\mu=1}^{4} \phi(x)\phi(x+\mu) + \phi(x)^2 + \lambda(\phi(x)^2 - 1)^2 \right]. \tag{7.24}$$

The independent parameters are κ and λ, as opposed to μ_0^2 and (a differently-scaled) λ. Eqn. 7.24 also adds a constant λ to the Lagrangian. Unfortunately, our simulations of this parameterization of the theory tended to exhibit large fluctuations and autocorrelation times, as well as unrenormalized results for μ_0^2 that did not approach the continuum limit in a linear fashion.

Moreover, in January 2006, Balog, Niedermayer and Weisz [4] submitted another paper that performed much of the analysis we had planned. This development, coupled with our difficulties simulating Eqn. 7.24 and the unexpected results in our study of phase transitions in the two-dimensional theory, prompted a change of course. Instead of focusing on the disagreement between Stevenson and Balog *et al.*, we returned to our original Lagrangian, Eqn. 6.26, and set out to chart the phase transition line in four dimensions.

Simulations and Results

We ended up following a similar procedure in four dimensions as we did in two, using the same parameterization of the Lagrangian. Of course, we performed the simulations on lattices of different sizes, and the required renormalization was considerably

more complicated, but the overall approach was identical. Our lattices were now four-dimensional cubes with L^4 sites, where $L = 8$, 12, 16, 20, 24 and 36.[7.7] Using these lattice sizes, we again iteratively scanned μ_L^2 for $\lambda = 1.0$, 0.7, 0.5, 0.25, 0.1, 0.05, 0.03, 0.02 and 0.01, extracting the continuum limits of the critical μ_L^2 in the manner described above (Table 7.4).

Starting from a random initial state, each simulation performed 8192 iterations for equilibration and an additional 8192 for statistics, except for the largest $L = 36$ lattices, on which simulations performed 4096 iterations for equilibration and an additional 4096 for statistics. We calculated the autocorrelation time τ for each simulation, using the method discussed above in Section 2.2 and implemented in the C++ code presented below in Code Snippet C.9. The autocorrelation times depended on both λ and L; they were of order 10 iterations for large λ on small lattices and of order 100 iterations for both small λ at all lattice sizes as well as large λ on large lattices. A small number of autocorrelation times on the largest 36^4 lattices actually exceeded 1000 iterations. Thus although most simulations had dozens or hundreds of autocorrelation times in which to equilibrate and take measurements, a few had only a handful.

As shown in Fig. 7.20 and Table 7.4, although the statistical uncertainties on the data are small, uncertainties in the regressions to the continuum limit are larger than in the two-dimensional case, since L^{-1} is larger. We expect larger systematic influences from finite-size effects, since we had to use smaller lattices. Although these lattices have roughly the same number of sites as those used in the two-dimensional simulations, it is the lattice's linear length L that matters for purposes of finite size effects since this is the size at which periodic boundary conditions cause clusters to begin to run into themselves, demolishing the fiction of an effectively infinite lattice.

Table 7.4 was generated in almost exactly the same way as Table 7.2 above. The only significant difference is in the renormalization of the theory, which we were not able to perform. Renormalization appropriate for lattice simulations of ϕ^4 theory in four dimensions is difficult to calculate. The most noteworthy attempt is that of Lüscher and Weisz [33, 34], which is not directly applicable to our situation.

Instead we focused on the bare phase transition line, which is shown below in Fig. 7.21. Unlike in the two-dimensional case, the four-dimensional bare transition line does not exhibit significant nonlinearity. Fig. 7.21 includes a linear regression that re-

[7.7] There was, however, one complication. Completing what we thought were our final simulations, we discovered that our program crashes if asked to simulate ϕ^4 theory with $\lambda = 0.01$ on 36^4 lattices in the strongly broken regime $\mu_L^2 < -.0058$. We suspect that the recursive methods we used to construct the Wolff cluster in this code (see Section C.3) were called so many times that their stack frames overflowed the available stack space itself. This would explain why the error appeared only for large lattices far in the broken phase: since many sites were being added to the cluster, the recursive methods were being called many times. Performing a stack trace with **gdb** (see Section B.1) confirmed that the segmentation faults occurred after these recursive methods had been called at least 20,000 times (most likely many more).

We attempted to get around this problem by varying the seeds given to the random number generators, but this proved ineffective. Instead, we simply ran the handful of problematic simulations on 32^4 lattices instead of the original 36^4 ones. We believe that doing so introduced negligible systematic error. We plan to replace the recursive methods with a stack-based approach over the summer of 2006, while rewriting the code as a whole in Fortran.

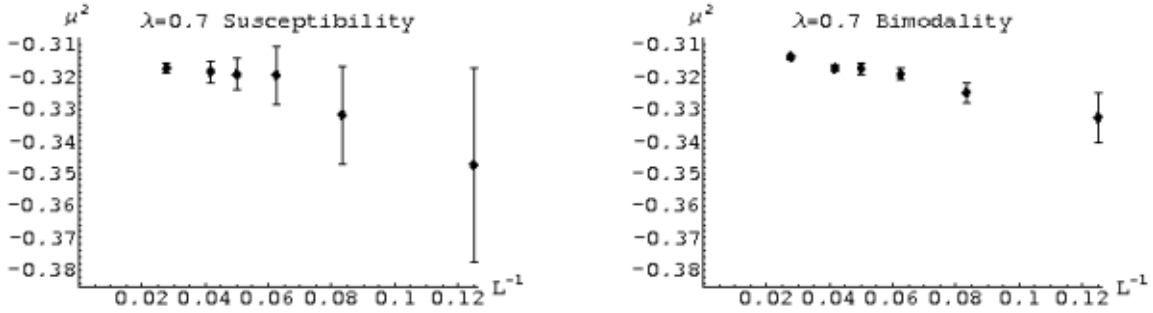

Figure 7.20: 4D critical μ_L^2 determined from susceptibility (left) and bimodality (right) for $\lambda = .7$

Table 7.4: Points on the four-dimensional bare phase transition line for various λ_L

λ	**Susceptibility**	**Bimodality**	**Cumulant**	**Critical μ_L^2**
1.00	-0.4432(15)	-0.4396(13)	-0.4449(25)	-0.4411(11)
0.70	-0.3129(19)	-0.3088(9)	-0.3161(27)	-0.3095(11)
0.50	-0.2237(9)	-0.2228(11)	-0.2259(32)	-0.2233(12)
0.25	-0.1109(6)	-0.1108(11)	-0.1167(16)	-0.1109(6)
0.10	-0.0438(7)	-0.0438(12)	-0.0477(8)	-0.0438(5)
0.05	-0.0225(8)	-0.0201(9)	-0.0241(9)	-0.0214(5)
0.03	-0.0125(3)	-0.0123(4)	-0.0143(7)	-0.0125(3)
0.02	-0.0078(8)	-0.0080(6)	-0.0097(2)	-0.0079(3)
0.01	-0.0051(3)	-0.0033(7)	-0.0059(11)	-0.0048(4)

sults in a variance per degree of freedom of 2.01. Adding a $\lambda \log[\lambda]$ term barely affects the variance, lowering it to 1.93, while adding a fourth $\lambda^2 \log[\lambda]$ term only reduced the variance to 1.63.[7.8]

However, these relationships will not necessarily hold after renormalization. We are actively attempting to determine the appropriate renormalization, but so far such calculations have proven intractable. We emphasize that in the our calculation of the bare phase transition line we have performed all necessary simulations and computations, a nontrivial contribution; only the analysis remains to be completed.

7.6 Soliton Mass Results

Though the unexpected developments in our investigations of ϕ^4 theory phase transitions and the two-dimensional critical coupling constant shifted our attention, we were able to perform an abbreviated foray into ϕ^4 solitons, originally the intended focus of our work. The first question to address is how to simulate solitons on the lattice; we are able to do so simply by altering the boundary conditions. Recall from Section 6.4 that soliton solutions continuously connect the two degenerate vacua of the broken phase,

[7.8]We performed these regressions only on Mathematica 5.2 on Windows, as a result of our earlier experiences with Mathematica 5.0 on Linux discussed above.

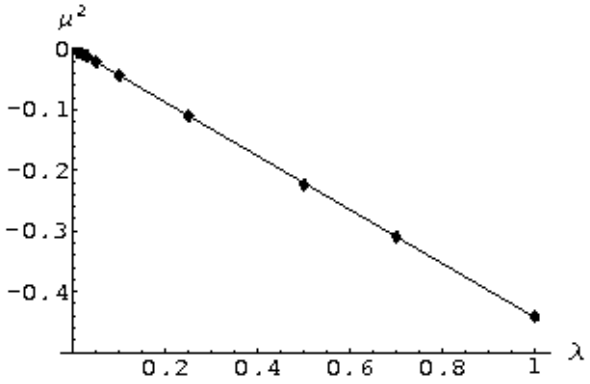

Figure 7.21: The bare ϕ^4 phase transition line in four dimensions: critical μ_L^2 vs. λ with linear regression

$\phi(x,t) = \pm\sqrt{-\mu_0^2/\lambda}$, with the result that $\phi(x \to \infty) = -\phi(x \to -\infty)$.

Now, we obviously do not have an infinite lattice, but we can encourage $\phi(0,t) = -\phi(L-1,t)$ on the spatial boundaries of the lattice by using antiperiodic boundary conditions (APBC), introduced above in Section 3.3. Briefly, APBC identify the same points as do periodic boundary conditions (PBC), but negate the value of ϕ at the site being 'wrapped around'. By using them on the spatial boundaries (but still using PBC on the temporal boundaries), we encourage (at all times t) the sites on opposing spatial boundaries to take on opposite values, since in that situation APBC will make them think they are aligned with their wrapped-around neighbors. Alignment is, recall, favored energetically due to the nearest-neighbors interaction in the lattice action, Eqn. 7.14.

Classically, we can calculate the mass (energy) of the soliton by substituting the soliton solution to the Euler-Lagrange equation back into the Hamiltonian and subtracting off the ground state energy. We did this back in Section 6.4 (Eqn. 6.35), finding

$$M_{cl} = \frac{2\sqrt{2}}{3}\frac{r^{3/2}}{\lambda}, \tag{7.25}$$

where for convenience we define $r = -\mu_0^2$, which is positive in the broken phase. Also for convenience, we will typically square M_{cl} and scale it by λ^2/r^3 in the figures below. When we quantize ϕ^4 theory, quantum corrections to the soliton mass diverge and need to be renormalized. This calculation is done in detail by Weidig [65] and Rajaraman [49, Section 5.4], who show that up to one loop (first order quantum corrections) the mass of the soliton is

$$M_{semi} = \frac{2\sqrt{2}}{3}\frac{r^{3/2}}{\lambda} + \sqrt{r}\left(\frac{1}{6}\sqrt{\frac{2}{3}} - \frac{3}{\pi\sqrt{2}}\right) + \mathcal{O}(\lambda). \tag{7.26}$$

As discussed by Rajaraman, the factor of $1/\lambda$ in this "semiclassical" result reveals the nonperturbative nature of the soliton and suggests this first order (in $\hbar$ and zeroth order in λ) approximation will only be valid in the classical weak-coupling limit $\lambda/r \ll 1$.

To calculate the soliton mass numerically, we need to perform simulations with both PBC and APBC. As discussed above, using APBC will produce solitons, while the simulations with PBC will fall into the ground state. By subtracting the ground state

energy from the energy of the simulations involving solitons, we can isolate the soliton mass. Actually, the situation is somewhat more complicated than this; as shown by Ciria and Tarancón [15], the mass of the soliton on the lattice is

$$M_{sol} = \frac{1}{T}\int_{\beta_c}^{\beta} \frac{\Delta S(\beta')}{\beta'} = \frac{1}{T}\int_{\beta_c}^{\beta} \frac{1}{\beta'}\left(\langle S_a\rangle - \langle S_p\rangle\right) \tag{7.27}$$

where S_a is the action calculated using APBC (corresponding to the soliton) while S_p is the action calculated using PBC (corresponding to the ground state). Both actions depend on $\beta = 1/\lambda$. β_c is the inverse of the critical λ_c for the fixed $r = -\mu_0^2$ at which the simulations are performed and T is the length of the lattice in the temporal direction (what we've been calling L).

We calculated $\Delta S(\beta)$ through programs similar to those used to investigate the phase transitions (most of the C++ source code is available in Code Snippets C.10 through C.12). The main difference was that these programs simultaneously simulated two separate lattices, one using PBC and the other using APBC. For three fixed values of $r = -\mu_0^2$ (1, 2.2 and 4, in imitation of Ardekani and Williams [1]), we scanned λ from near zero until we approached the phase transition into the symmetric phase, where the soliton vanishes. In hindsight it would have made more sense to vary β as opposed to λ. Our approach is still satisfactory but resulted in an excessive number of simulations performed near β_c, visible in the figures below. These excessive simulations were computationally costly, especially because we were not able to minimize the range of λ to scan as we did with μ_0^2 when investigating phase transitions.

As a result, we were only able to complete full scans of λ for simulations on lattices of size $L = 32$, 48, 64, 128 and 256. Each of our simulations again performed 16384 mixed Metropolis/Wolff iterations for equilibration and an additional 16384 for statistics. Although the PBC lattice was initialized in a random state, we found it necessary to initialize the APBC lattice in an ordered state mimicking the soliton. This is because the antiperiodic boundary conditions make it possible for the Wolff cluster algorithm to add to the cluster sites with opposite signs. In the strongly broken phase, this means that nearly all sites would be added, reducing the Wolff algorithm's ability to cope with metastable states (an APBC variant of the diagonal stripe state in Fig. 5.7 was especially persistent). By starting in an ordered state with lower energy than the metastable states, we guaranteed that they would not interfere with our calculations.

Even though we were able to eliminate the occurrence of metastable states by starting the APBC lattice in an ordered state, the autocorrelation times for the APBC lattices were still significantly longer than those for the PBC lattices. While the PBC lattices' autocorrelation times were all less than ten iterations, the APBC lattices' were typically two to four orders of magnitude greater, leading to statistical uncertainties on $\langle S_a\rangle$ roughly an order of magnitude greater than those on $\langle S_p\rangle$. The autocorrelation time typically decreased as lattice size increased under periodic boundary conditions but increased with L under antiperiodic boundary conditions.

This problem was noted by Ardekani and Williams, who identified its source as translational motion of the soliton. To address it, they imposed a constraint on the soliton, fixing its center to the center of the lattice by setting $\phi(L/2, t) = 0$ for all times

t and for the duration of the simulation. This halted the soliton's translational motion and made it more clearly defined, as shown in Fig. 7.22, which plots one measurement of the field value ϕ averaged over all times t for each "time slice" on a 48^2 lattice with $r = 1$ and $\lambda = 0.5$ ($f = r/\sqrt{\lambda} = \sqrt{2}$). Adding this constraint also dramatically decreased the autocorrelation times and uncertainties of the lattices with APBC, to the same level as the lattices with PBC, less than ten iterations. The decrease in uncertainties is illustrated by Fig. 7.23, which compares $\Delta S/T$ and M_{sol} for simulations using both unconstrained (left) and constrained (right) APBC lattices at $r = 4$.

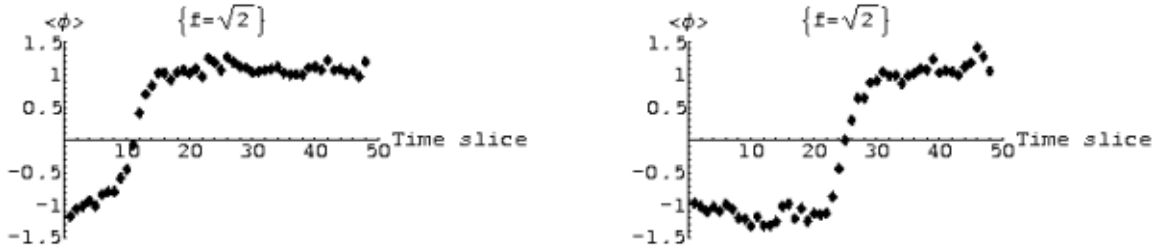

Figure 7.22: Unconstrained (left) and constrained (right) lattice solitons on a 48^2 lattice with $r = 1$ and $\lambda = 0.5$ ($f = r/\sqrt{\lambda} = \sqrt{2}$), at one measurement

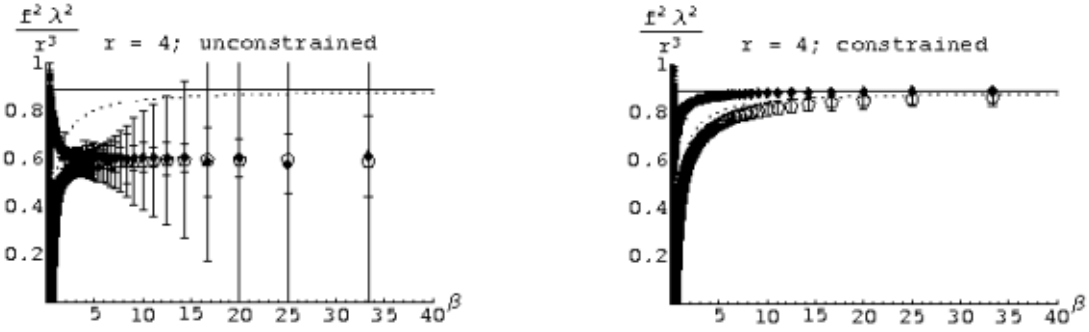

Figure 7.23: Soliton masses calculated at $r = 4$ on unconstrained (left) and constrained (right) lattices

Fig. 7.23 plots (against β) lattice data for $\Delta S/T$ (solid points) and M_{sol} (hollow diamonds) calculated from Eqn. 7.27 as well as M_{cl} (solid line) and M_{semi} (dashed line) from Eqns. 7.25 and 7.26. In each case the quantity being graphed has been squared and scaled by λ^2/r^3. We calculated the masses from Eqn. 7.27 through Mathematica, using its `ListInterpolation` command to calculate an integrable function from our discrete $\Delta S/T$ data points. We obtained the uncertainties in the masses by performing the same procedure using $\Delta S/T \pm \sigma_{\Delta S/T}$, with the result that the uncertainty on each value of $M_{sol}(\beta)$ is cumulative and includes contributions from the uncertainties of all $\Delta S/T$ points between β and the phase transition at β_c.

We constructed our graphs in this manner since our goal was to reproduce Fig. 7.24 from Ciria and Tarancón [15]. Since Ciria and Tarancón use $\lambda/4!$ in their Lagrangian instead of our $\lambda/4$, the vertical axis of Fig. 7.24 is 36 times what ours will be, while its horizontal axis is scaled by $\frac{1}{6}$ relative to ours. There are several reasons we expect our

results to be an improvement on theirs. First, we were able to use our phase transition results to determine highly accurate values of β_c (Table 7.5), which appears in Eqn. 7.27 as the lower bound of the integral. Ciria and Tarancón, in contrast, only report an estimate of $\beta_c = 0.4824(132)$ ($\lambda_c = 2.075(58)$) for $r = 2.2$, based solely on the position of the sharp transition in Fig. 7.24.

Table 7.5: Critical β_c calculated using the data from Section 7.5

	$r = 1$	$r = 2.2$	$r = 4$
β_c	1.34443(6)	1.97597(22)	4.13749(125)
λ_c	0.74381(3)	0.50634(6)	0.24169(7)

Additionally, Ciria and Tarancón performed their simulations only on 48^2 lattices. By using a series of lattices of increasing size $L = 32$, 48, 64, 128, 256 and taking the infinite-volume limit, our data should better take into account finite size effects and more accurately reflect the continuum theory. Finally, we simply took far more data than Ciria and Tarancón, partly due to our scans through λ as opposed to β. This results in smaller statistical uncertainties, the ability to extend our charts to $\beta > 2.4$, and more fully fleshed-out lines, especially for lower β.

Unfortunately, our initial results disagreed with Fig. 7.24, in particular regarding the semiclassical mass prediction. This disagreement remained even if we performed our calculations on unconstrained APBC lattices (Fig. 7.25, left). We were able to track the problem to an error made by Ciria and Tarancón in their statement of the semiclassical soliton mass, Eqn. 7.26, which places the factor of $\sqrt{2}$ in the numerator of the last term. Initially we assumed this was merely a typo, but we found that adopting their definition of the semiclassical mass produced results (Fig. 7.25, right) in agreement with theirs, indicating that they actually used this incorrect expression in their calculations. We are confident Ciria and Tarancón are incorrect because they do not present a derivation of the equation; instead they merely cite Rajaraman [49, Section 5.4], who actually places the $\sqrt{2}$ in the denominator (as does Weidig [65] independently).

Our full results are presented in Figs. 7.26 and 7.27. Each figure includes charts modeled on Fig. 7.24 for each of the three values of $r = -\mu_0^2$ at which we performed simulations. The charts on the left illustrate the behavior of the (squared and scaled) mass near the critical point ($\beta < 4$), while those on the right reveal its asymptotic behavior in the classical low-λ (high-β) limit ($\beta < 40$).

There are striking differences between the results obtained using the constraint proposed by Ardekani and Williams (Fig. 7.26) and those obtained without imposing the constraint (Fig. 7.27). While both sets of data behave similarly near the critical point, the constrained masses quickly approach (from below) the semiclassical values as either β or r increases. The perturbative calculation in turn approaches the classical value in the classical (weak coupling) limit $r/\lambda = r\beta \gg 1$. This disagrees with Ciria and Tarancón's conclusion that $\Delta S/T$ quickly approached the classical limit while M_{sol} slowly decreased to the perturbative value.

The behavior of the unconstrained masses is more striking. At low r and β it more closely resembles Ciria and Tarancón's results in Fig. 7.24, in that $\Delta S/T$ quickly

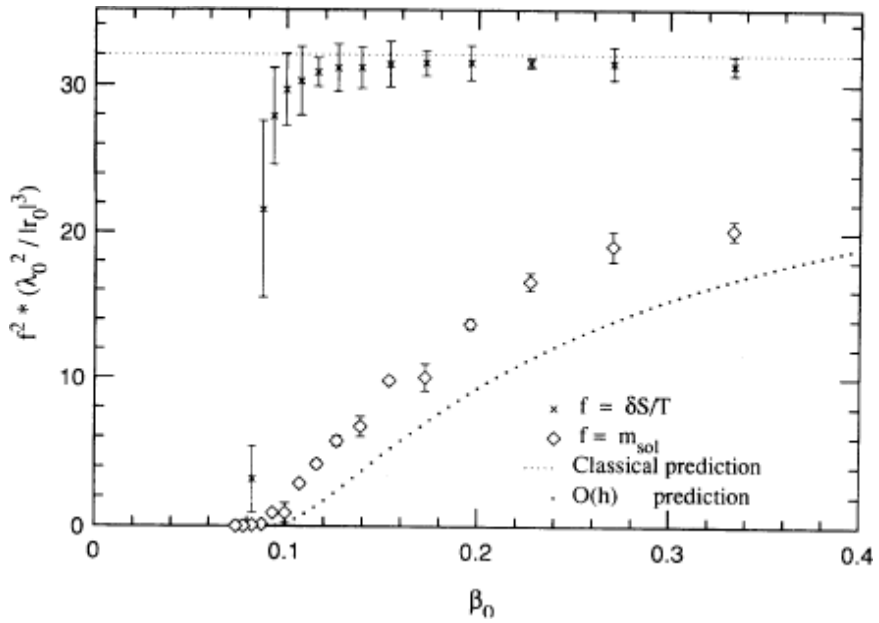

Figure 7.24: Results from Ciria and Tarancón [15]

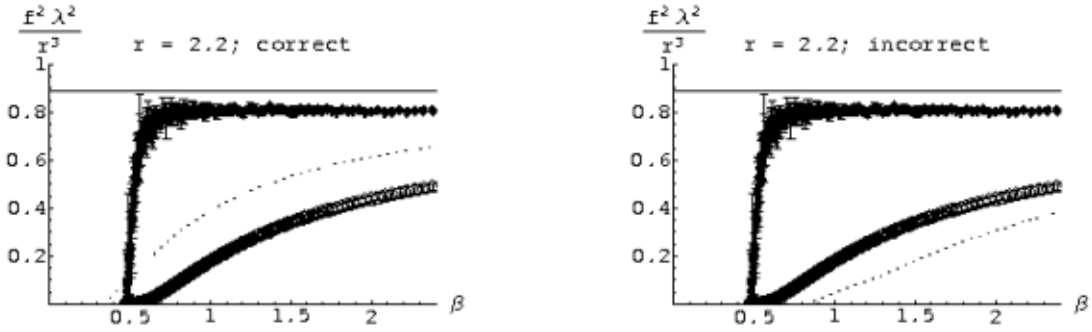

Figure 7.25: Our attempts to reproduce Fig. 7.24 using unconstrained lattices and two versions of Eqn. 7.26

approaches the classical prediction while M_{sol} approaches the semiclassical value more slowly. However, for higher r, we actually see $\Delta S/T$ begin to decrease until it falls below the perturbative result. The apparent peak in $\Delta S/T$ in the higher-r simulations is exaggerated by the larger error bars caused by critical slowing down; the central values vary smoothly. Note that although the constrained values of $\Delta S/T$ also cross the semiclassical line, they do so more dramatically at lower r and then approach it again, from below. The unconstrained masses appear to approach in the classical limit an asymptotic value significantly below the classical predictions, with the amount of disagreement seeming to increase as r increases. However, the greater uncertainties in the unconstrained values for $\Delta S/T$, which result in very large cumulative uncertainties on the masses, make it difficult to draw strong conclusions about the behavior of the unconstrained soliton masses in the classical limit.

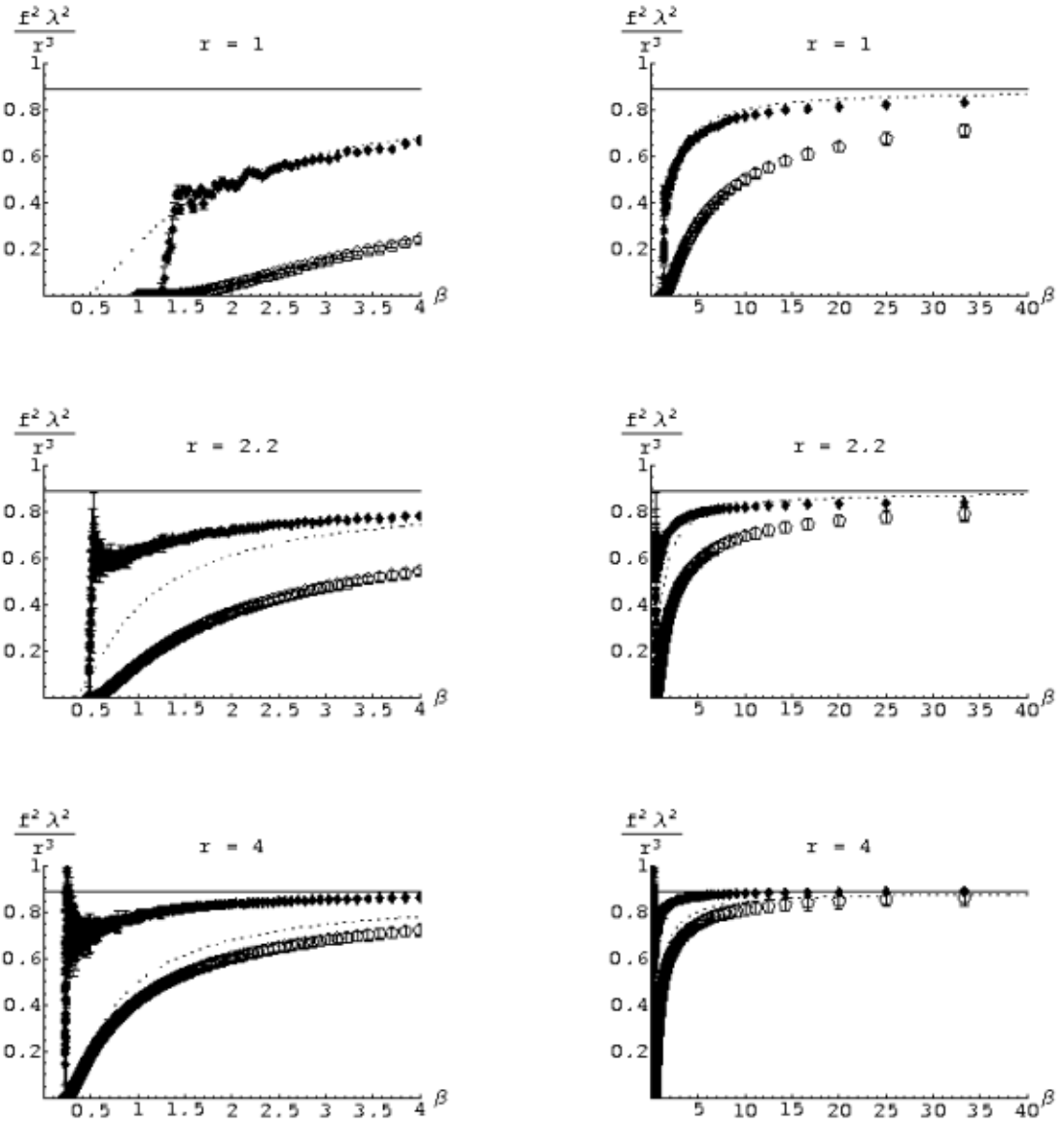

Figure 7.26: Soliton masses calculated on constrained lattices for $r = 1$, 2.2 and 4

We are inclined to believe that the results calculated on constrained lattices more closely reflect the pure soliton masses. As shown in Fig. 7.23, constrained solitons are

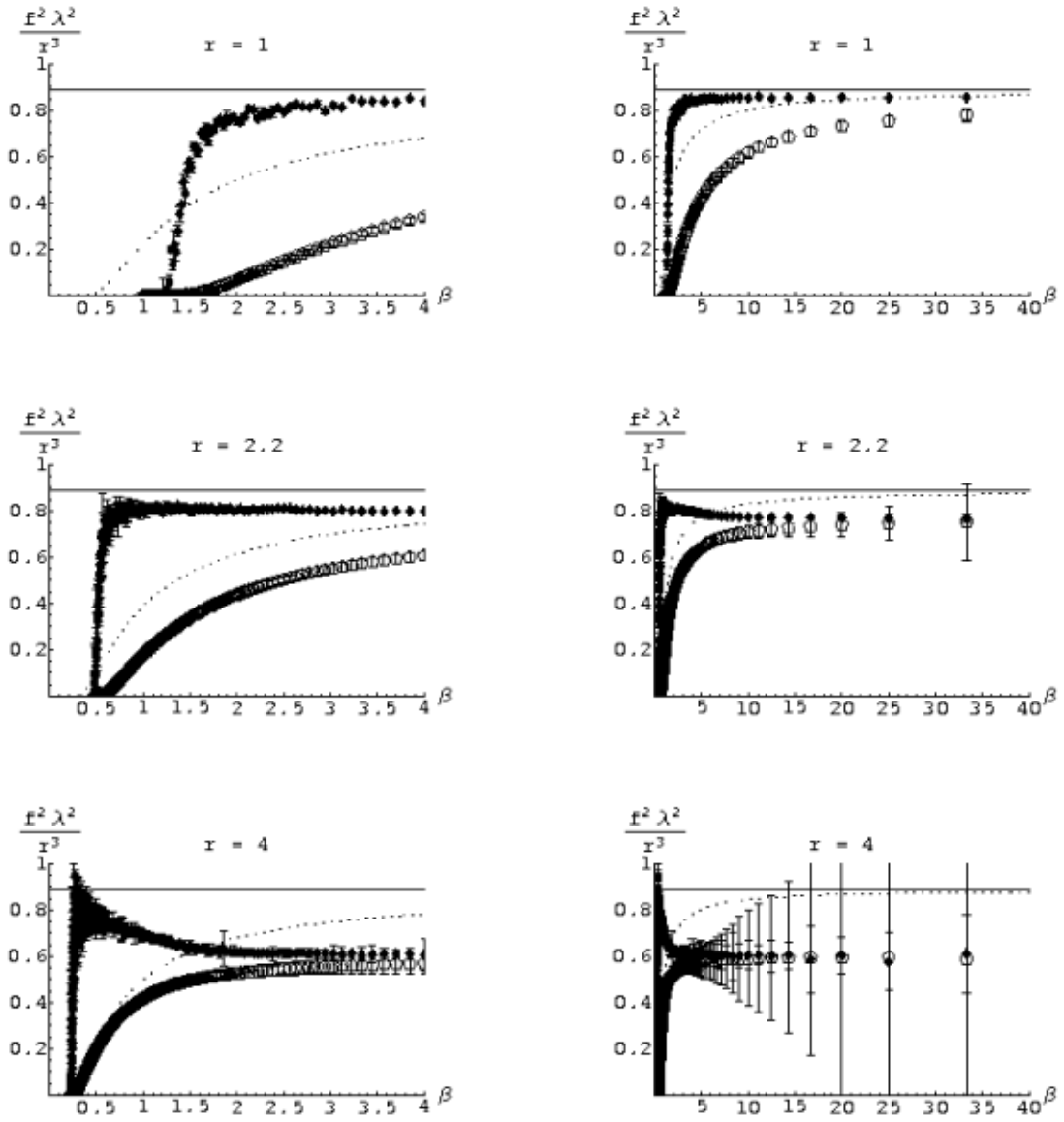

Figure 7.27: Soliton masses calculated on unconstrained lattices for $r = 1$, 2.2 and 4

more clearly defined than those simulated on unconstrained lattices. Moreover, the constrained masses correctly approach the classical and semiclassical predictions in the classical limit, while the unconstrained masses appear to stabilize at a significantly lower value (though with such extreme uncertainties for high r that making definite conclusions becomes troublesome).

Chapter 8

Conclusions and Directions for Future Research

In the preceding chapters we gave a brief pedagogical introduction to Markov chain Monte Carlo simulations of statistical systems and the basics of the ϕ^4 quantum field theory. We showed how to treat ϕ^4 theory as a statistical system in Euclidean space and perform lattice simulations using both local and cluster Monte Carlo algorithms. We then presented the details and results of simulations of ϕ^4 theory on the lattice in which we charted the phase transition line of the theory in two and four dimensions and calculated an accurate value of the continuum critical coupling constant in two dimensions theory. Finally we discussed lattice simulations that calculated the masses of ϕ^4 solitons in two dimensions, nonperturbative phenomena that resulted from the nonlinearity of the theory.

We obtained a critical coupling constant of $[\lambda/\mu^2]_{crit} = 10.85^{+.03}_{-.08}$ in two dimensions, which disagrees by over 7σ with that reported by Loinaz and Willey [32], $[\lambda/\mu^2]_{crit} = 10.26^{+.08}_{-.04}$. Though we showed that this disagreement is the result of higher-order dependence of $[\lambda/\mu^2]_{crit}$ on λ caused by approximations we made when discretizing the lattice, we have not yet been able to determine the analytic form of these corrections. Nor were we able to fully renormalize four-dimensional lattice ϕ^4 theory to obtain a critical coupling constant in four dimensions. Our work on both of these problems is continuing, however, and they are only two of a wide variety of issues worthy of further exploration.

While the analytic calculation of the corrections resulting from our nearest-neighbor approximation is somewhat intractable, there are other approaches that could be taken to analyze the issue further. For instance, our simulations and analysis could be replayed using an "improved action" that takes into account certain long-distance interactions, such as those between diagonal lattice sites, in addition to the nearest neighbors. There even exist schemes known as "perfect actions", which claim to largely eliminate such systematic effects of discretization while remaining effectively local.

There is also productive work to be done exploring algorithms, both those we used

in our and those we passed by. Although we suspect that the invaded cluster algorithm (Subsection 5.7.1) cannot be applied to ϕ^4 theory, its potential benefits to phase transition studies like ours are so great that it would be very valuable to establish conclusively whether or not the algorithm can be used. Some other tasks include tuning the algorithms we used in our simulations in order to increase their efficiency. In particular, we did not perform a systematic study of how the ratio of Metropolis sweeps to Wolff cluster flips affects critical slowing down and autocorrelation times (Section 5.5), nor did we investigate how the efficiency of ϕ^4 simulations depends on the amount which the Metropolis algorithm is allowed to change ϕ at any particular site during a single update (Section 7.5). While these projects may seem less glamorous than others, they are straightforward with potentially very valuable results – improving efficiency is critical to performing the most accurate calculations in the shortest amount of time.

There are also opportunities to extend our study of solitons in ϕ^4 theory. Our results for the soliton mass, though they are only preliminary, have already revealed errors in the existing literature. There are thus good reason to expand and refine our soliton mass calculations and perform more detailed analyses. Ardekani and Williams [1], for instance, perform an analysis of the zero-mode contribution to the soliton mass that could be improved by using our methods and data. In addition, there are also other features of solitons worth investigating on the lattice, such as the topological charge discussed by De *et al.* [16].

Finally, there are nearly unlimited possibilities for exploring other quantum field theories or statistical systems on the lattice. We mentioned a few such statistical systems in Section 5.6, such as the Potts model, XY model and Heisenberg model. As a nonabelian model, the Heisenberg model is especially interesting, not least due to its analogies with Yang-Mills quantum field theories and quantum chromodynamics (QCD). Lattice QCD in particular is one of the largest focuses of computational effort among professional researchers. In addition, researchers such as Sandvik [52] are performing active research modeling spin systems using the Heisenberg model.

Appendix A: Experimental Apparatus

The computations and simulations described above required significant computer resources. Fortunately, Amherst College recently established an interdisciplinary scientific computing cluster (Fig. A.1), which we used to perform all of our calculations.

In this appendix we will summarize the details of the cluster – its purpose, hardware and software – as well as present a brief guide to running projects on the cluster designed to get users off the ground and alert them to some potential pitfalls. We will also address parallel processing on the cluster and its relation to our work.

A.1 Amherst College's Interdisciplinary Scientific Computing Cluster

With funding from the National Science Foundation (NSF; grant #0512269) and Amherst College (a Faculty Research Award Program (FARP) grant), Prof. Kaplan of the Department of Mathematics and Computer Science has been developing and maintaining the new interdisciplinary scientific computing cluster at Amherst. Using the new funds from the NSF, the cluster was expanded considerably over the past year and has now reached a mature state possessing computing power sufficient to handle the simulations required by our work and the concurrent research pursued by others.

The cluster is interdisciplinary in that it is open to all Five-College[A.1] researchers who need large-scale computing resources, regardless of field. In the year when the cluster reached full-scale operation, the 2005-2006 academic year, it was used for computational projects in physics, computer science and biology. In addition, researchers in geology and economics are developing and considering projects that will make use of the cluster. Since the cluster was being constructed during this time, it was only available in a mature form for a few months and suffered from some of the difficulties that inevitably accompany such large-scale development projects. This suggests that in the near future more researchers representing an even broader array of fields will have an easier time taking advantage of

[A.1] The Five Colleges are Amherst College, Hampshire College, Mount Holyoke College, Smith College and the University of Massachusetts at Amherst.

the cluster's computational resources for their projects.

In the following sections we will briefly introduce the hardware of the cluster as well as the software that runs on the cluster and manages it before taking a deeper look at the practical matter of how to use this software to run projects on the cluster.

A.1.1 Hardware

Amherst's cluster is constructed exclusively from 'off-the-shelf' commercial equipment. We'll briefly state the technical specs of the computers in the cluster, without explanation, for those who may be interested.

The core of the cluster consists of 26 new HP DL145 G2's, each with two AMD Opteron 252 processors, 2 GB RAM, a 60 GB SATA hard disk and a 1 Gb NIC connection. Each AMD processor is 64-bit and single-core, and clocks at 2.6 GHz using a 1 MB L2 cache. In addition the cluster currently includes five 32-bit computers purchased earlier through the FRAP grant, each with 3 GHz Pentium 4 chips, 1 GB RAM, a 40 GB IDE disk and 1 Gb NIC connection. Earlier in the year there were also several other, older 32-bit computers in the cluster, which were gradually removed as the new HPs came online. The cluster also originally included the Macs used in Amherst's public computer labs when they were idle. OS upgrades have now made use of the public machines impossible, but we hope to reincorporate them back into the cluster in the future.

In coming years the NSF grant will be used to expand the cluster to at least 150 processors, possibly close to 200, pending resolution of cooling and power issues.

A.1.2 Software

All of the computers on the cluster run the Fedora Core 4 Linux operating system and are managed by Condor cluster management software, version 6.6.10. The compiler we used on the cluster is gcc version 4.0.0 20050519 (Red Hat 4.0.0-8). As shown in our sample Makefile (Code Snippet B.2), our default compiler flags were -Wall and -O3. -Wall orders the compiler to print all warnings, while -O3 has it optimize the code to reduce execution time as much as possible (increasing the time and memory needed to compile).

The most important program running on the cluster is condor, which runs the cluster. Condor manages and schedules the projects running on the cluster and must be used to submit (and in some cases, compile) the projects to be run. Since condor is of such importance, we have written a brief guide to running projects on the cluster using condor, to which we now turn.

Figure A.1: Amherst College's interdisciplinary scientific computing cluster, April 2006

A.2 A Brief Guide to Working on the Cluster

In order to run jobs on the condor cluster, all we have to do is run the command `condor_submit submit_description_file`, where the submit description file contains all the information condor needs to assign, manage, and run the jobs. In particular, it must specify the name of the program to run by stating `executable = <path>`, and must include a `queue` command after every job to be submitted. Additional options that are commonly used include

- any `arguments = <args>` the program needs to run;
- the self-explanatory `priority = <int>`;
- `output = <path>`, `error = <path>` and `log = <path>`, which tell condor where to write[A.2] results, error messages, and status information, respectively;
- `requirements = <args>` to specify any special needs of the program (for instance, whether it requires a certain operating system or architecture);
- and `universe = <arg>`, which specifies which universe to use (more about this shortly).

There are many more options described in the condor_submit documentation, which can be accessed by running `man condor_submit` on any computer in the cluster. A sample submit description file, with some explanatory comments added, is shown below in Code Snippet A.1; running `condor_submit` on Code Snippet A.1 will submit to the cluster two four-dimensional ϕ^4 simulations with $L = 36$, $\lambda = 0.01$, 4096 iterations for equilibration with another 4096 iterations for statistics, and $\mu_l^2 = -.01$, -.0101.

Code Snippet A.1: Sample submit description file for condor cluster

```
## Global job properties
universe =     vanilla  # Using the vanilla universe
notification = never    # Don't have condor email when job is done or error occurs
getenv =       true     # Get user environment -- $PATH, $LD_LIBRARY_PATH, etc
priority =     15
executable =   Simulation
image_size =   4036     # Make sure job isn't submitted to a machine without the resources to run it
requirements = (Arch=="x86_64") && (OpSys=="LINUX")

## Task properties
output = /mnt/store/daschaich/current/thesis/code/4DPhi4/results/36-1--100out
error = /mnt/store/daschaich/current/thesis/code/4DPhi4/results/36-1--100err
arguments = -100 1 36 36 36 36 4096 4096
queue

output = /mnt/store/daschaich/current/thesis/code/4DPhi4/results/36-1--101out
error = /mnt/store/daschaich/current/thesis/code/4DPhi4/results/36-1--101err
arguments = -101 1 36 36 36 36 4096 4096
queue
```

Typically we want to submit many jobs at once, up to hundreds or thousands. Manually writing the necessary submit description file would be idiotic. Instead we use scripts to generate an appropriate submit description file and submit it to the cluster. The script that generated Code Snippet A.1 is shown below in Code Snippet A.2.

[A.2] It is also possible to have results sent directly to a database, though we did not make use of this feature.

Code Snippet A.2: Automated script for submitting jobs to the cluster

```
#! /bin/tcsh -f

set taskName = Simulation
set baseDirectory = `pwd`
set resultsDirectory = ${baseDirectory}/results
set commandPathname = ${baseDirectory}/${taskName}.cmd

# Create the results directory and make it world-writeable so that
# Condor doesn't complain.
mkdir --parents --mode=3770 ${resultsDirectory}
chgrp condor ${resultsDirectory}

# Emit the arguments needed for the condor script.  These are for all
# tasks in the job.
printf "## Global job properties\n\n" > ${commandPathname}
printf "universe =     vanilla\n" >> ${commandPathname}
printf "notification = never\n" >> ${commandPathname}
printf "getenv =       true\n" >> ${commandPathname}
printf "initialdir =   ${baseDirectory}\n" >> ${commandPathname}
printf "priority =     15\n" >> ${commandPathname}
printf "executable =   ${taskName}\n" >> ${commandPathname}
printf "image_size =   4036\n" >> ${commandPathname}
printf "requirements = (Arch==`../print-quoted-${HOSTTYPE} x86_64`)
    && (OpSys==`../print-quoted-${HOSTTYPE} LINUX`)" >> ${commandPathname}

# Loop through the arguments to be passed for each run.
set lambda = 1
set size = 36
set runs = 4096

set temp = -100
set minTemp = -101
while ($temp >= $minTemp)
    # Emit the arguments for this task.
    printf "\n## Task properties\n\n" >> ${commandPathname}
#    printf "log = ${resultsDirectory}/${temp}-${lambda}log\n" >> ${commandPathname}
    printf "output = ${resultsDirectory}/${size}-${lambda}-${temp}out\n" >> ${commandPathname}
    printf "error = ${resultsDirectory}/${size}-${lambda}-${temp}err\n" >> ${commandPathname}
    printf "arguments = ${temp} ${lambda} ${size} ${size} ${size} ${size} ${runs} ${runs}\n"
        >> ${commandPathname}
    printf "queue\n" \

    set temp = `expr $temp - 1`
end

# Submit the job.
condor_submit ${commandPathname}

# Clean up.
rm ${commandPathname}
```

Programs in condor can run in any of several 'universes', of which the 'vanilla universe' and 'standard universe' are the most commonly used. In addition there is a 'java universe' for Java programs, a 'globus universe' for jobs using Grid resources on widely distributed machines, as well as 'MPI' (Message Passing Interface) and 'scheduler' universes. More details can be found in the condor documentation. The vanilla universe is the simplest to use. It makes no assumptions about the program – anything that will run by itself on a computer in the cluster will run in the vanilla universe.

The standard universe includes several helpful features, but requires programs to be compiled with the `condor_compile` command, which links in the necessary condor libraries. It is simple to use `condor_compile` by adding it to the CC (or equivalent) field of the Makefile. For example, the sample Makefile shown below in Code Snippet

B.2 would read `CC = condor_compile g++` instead of `CC = g++`. The most important advantage to using the standard universe is that condor can add its own checkpoints to the program, which allow a running job to be safely stored if it is descheduled to allow others' programs to run. If the vanilla universe is used, programs will be aborted when they are descheduled and restarted when scheduled again, eradicating any computations that may already have been performed.

This can be a significant burden, especially if programs are being swapped repeatedly, continually restarted before they have a chance to finish. We strongly recommend using `condor_compile` and the standard universe if possible. Unfortunately, in some cases this is not possible. In particular, for `condor_compile` to work, the compiler used (e.g., gcc) must be the same version that was used to compile condor itself.

We have noticed two other problems in the cluster as of April 2006 and condor version 6.6.10. First, computers in the cluster occasionally report absurdly high loads, even though `top` reveals no programs using significant resources. The problem is that the machines behave as though they were actually operating under such loads, becoming unresponsive to all users. On a regular cluster machine, this merely delays the execution of the job assigned to it by condor. On the gateway machine that manages the cluster, this problem prevents all users from logging in and submitting new jobs or managing those already running. The only remedy for this situation is to reboot the machines claiming high loads. In addition, we have recently observed that some programs that segfault after running on the cluster for a long time (tens of hours) do not report errors and are not descheduled. Rather, they just seem to keep running indefinitely. We have not yet had the opportunity to explore this issue in detail or speculate about its potential causes and solutions.

A.3 Parallel Processing

Computer clusters are motivated by a desire to perform parallel processing, in which large computational tasks are split between a number of different processors. Processors in clusters often work in close cooperation with each other on their tasks and thus need to be able to communicate quickly and efficiently. Typically they are therefore connected by fast local area networks.

Despite using the cluster, we did not perform any 'massively' parallel computing in our work. Instead our simulations were 'trivially' (or 'embarrassingly') parallel: we simply split up our tasks (such as sampling a phase space) into a number of *completely independent* computations, each of which was then individually tackled by a single processor in the cluster. We can imagine performing a lattice simulation using various kinds of nontrivially parallel algorithms. For instance, if we were using a local algorithm such as the Metropolis algorithm, the lattice could be broken up into a number of different pieces (domains), each of which would then be sent to a different processor for simulation. The boundaries between the different domains present difficulties, however, since any changes to sites on one side of a boundary would need to be communicated to the machine(s) sim-

ulating the neighboring domain(s), making use of the cluster's internal communications capabilities. Alternately, we could fix the values of all the boundary sites, and periodically shift around the domains in order to regain ergodicity (this, of course, also requires significant intra-cluster communication). This is an example of nontrivial parallelization through domain decomposition, which (along with parallelization through functional decomposition) is discussed in more detail in Newman and Barkema [40, Chap. 14].

Although domain decomposition makes it possible for computations on large lattices to be shared across several processors, it is clearly a good deal more complicated than the trivially parallel approach we took. The choice of whether and how to parallelize simulations depends on the details of the computations, which determine both the possibility of parallelization and the potential benefits (or lack thereof). Trivial parallelization was well-suited to our tasks and allowed us to exploit the processing power of the cluster easily and efficiently.

Appendix B: Efficient Programming

A week of hard work can sometimes save you an hour of thought. – G. V. Wilson.

In this and the following appendix we will briefly delve into some of the nuts and bolts programming issues that came up during our work and that we believe may be helpful to others planning to do similar work. We begin with a discussion of some potentially useful tools that can streamline programming and make both the design and execution of code more efficient. We will then briefly survey some of today's most popular programming languages and discuss the reasons we chose to write our code in C++. Some excerpts of that code will be presented in the next appendix, but before turning to it we will give a brief introduction to data structures and some tricks for improving efficiency that we encountered over the course of our work.

Those with significant programming experience will most likely be familiar with all of the issues addressed in this appendix, but may wish to skim through it nonetheless.

B.1 Potentially Useful Tools

In a recent article [68], computer scientist G. V. Wilson charged that many scientists exhibit extreme "computational illiteracy" that can disastrously obstruct their research. While there are undoubtedly many scientists with impeccably organized, systematic and efficient programming practices, both our experiences and those reported by Wilson suggest that a non-negligible fraction of researchers would benefit from an introduction to some efficient programming practices.

Wilson has set up an online lecture series[B.3] to promote programming literacy among scientists and engineers. In this section we will present our own views on some of the potentially useful tools recommended by Wilson and his collaborators, as well as one critical tool they have overlooked. We will not attempt to reproduce Wilson's lectures, which include discussions of both more basic issues (e.g. shells and coding style) and more advanced topics (e.g. security and unit test suites) than we will address.

[B.3] Available at http://www.third-bit.com/swc2/index.html (last accessed 10 May 2006)

B.1.1 Code Profiling

Even professional programmers are notoriously bad at figuring out how to improve the efficiency of their programs. All generally seem to misjudge where their programs spend the most time and, consequently, where the greatest gains in efficiency can be made. There are innumerable stories about skilled programmers who spend days or weeks fine-tuning code that doesn't actually affect the overall running times of their programs. We ourselves contributed to the literature before being introduced to code profiling by an acquaintance.

Code profiling refers to techniques for measuring the amount of time that a program spends in each of its methods or subroutines. It lets the programmer know what portions of their programs are meaningfully contributing to runtime and are worth making more efficient. We consider code profiling an essential tool for any programmer truly interested in producing fast code.

The standard code profiling tool on Linux platforms is *gprof*, which calculates and displays call graph profile data for C/C++, Pascal and Fortran programs. There are doubtless analogous programs for other operating systems and other programming languages such as Java. Some excerpts from a gprof analysis of our ϕ^4 theory simulation program in two dimensions are shown below in Code Snippet B.1 (explanatory notes have been cut and some lengthy method names have been truncated for formatting). Programs to be analyzed by gprof need to be compiled with the "-pg" option – just add -pg to the CFLAGS (or equivalent) field in your Makefile (see Section B.1.4). The -pg option will link in the necessary libraries for profiling and cause the program to produce a (binary) call graph profile file (gmon.out by default) when it runs. When gprof is then run, it reads the given executable and uses the call graph profile to determine the amount of time spent in each method.

As shown in Code Snippet B.1, gprof gives rather detailed information about the absolute and relative amount of time spent in each method (total and per call), the number of times the method is called, which other methods any particular method calls, and so on. The information is displayed both in a concise flat profile that lists methods in order of how much total time was spent in them, and in a more detailed call graph that traces the paths through which various methods are called and call others. gprof can also produce an annotated source file, a copy of the program's source code labeled with the number of times each line was executed. See the gprof manual pages (man gprof) for more information and more options.

The particular analysis shown in Code Snippet B.1 alerted us to the fact that our elaborate (and often ingenious) attempts to make our hash table more efficient were essentially worthless (from a practical point of view) since hash table methods already took up only around 3% of total runtime. Code Snippet B.1 is actually an analysis of our two-dimensional ϕ^4 simulation program, the source code of which can be found below in Code Snippets C.7 through C.9.[B.4]

[B.4] Since the growCluster methods seem to run for an anomalously long time, we should note that they are recursive methods and that gprof reported the total time of the whole recursion.

Code Snippet B.1: Excerpts from sample code profile produced by gprof

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 55.50     34.58    34.58                             main
 35.06     56.43    21.85 167772160     0.13     0.13  Lattice::metropolis(unsigned int)
  2.62     58.06     1.63    16394    99.47   153.88  Lattice::growClusterPos(unsigned int)
  2.10     59.37     1.31 34694250     0.04     0.04  HashTable::find(unsigned int)
  1.90     60.56     1.19    16374    72.40   126.88  Lattice::growClusterNeg(unsigned int)
  1.43     61.45     0.89    32768    27.17    27.17  Lattice::flipCluster()
  0.76     61.92     0.48  9498749     0.05     0.05  HashTable::insert(unsigned int)
  0.40     62.17     0.25    16384    15.26    15.26  Lattice::calcTotalEnergy()
  0.11     62.24     0.07    16384     4.27     4.27  Lattice::calcAveragePhi()
  0.10     62.31     0.07                             HashTable::~HashTable()
  0.04     62.33     0.03                             Lattice::printLattice()
  0.00     62.33     0.00    32768     0.00   167.61  Lattice::wolff(unsigned int)
  0.00     62.33     0.00       11     0.00     0.00  std::vector<siteNeighbors*, std::allocator...
  0.00     62.33     0.00       11     0.00     0.00  std::vector<double, std::allocator<double> >...
  0.00     62.33     0.00        1     0.00     0.00  calcAutocor(unsigned int, double*, double*)
  0.00     62.33     0.00        1     0.00     0.00  Lattice::calcSpecificHeat(double, double)
  0.00     62.33     0.00        1     0.00     0.00  Lattice::calcSusceptibility(double, double)
  0.00     62.33     0.00        1     0.00     0.00  Lattice::Lattice(double, double, unsigned int,...
  0.00     62.33     0.00        1     0.00     0.00  Lattice::~Lattice()
  0.00     62.33     0.00        1     0.00     0.00  HashTable::HashTable(unsigned int)
  0.00     62.33     0.00        1     0.00     0.00  std::vector<node*, std::allocator<node*> >::...
  0.00     62.33     0.00        1     0.00     0.00  node** std::fill_n<node**, unsigned int, node*>...

[...]

                     Call graph

granularity: each sample hit covers 2 byte(s) for 0.02% of 62.33 seconds

index % time    self  children    called     name
[1]     99.9   34.58   27.66                 main [1]
               21.85    0.00 167772160/167772160     Lattice::metropolis(unsigned int) [2]
                0.00    5.49   32768/32768       Lattice::wolff(unsigned int) [3]
                0.25    0.00   16384/16384       Lattice::calcTotalEnergy() [9]
                0.07    0.00   16384/16384       Lattice::calcAveragePhi() [10]
                0.00    0.00       1/1           Lattice::Lattice(double, double, unsigned int,...
                0.00    0.00       1/1           Lattice::calcSpecificHeat(double, double) [22]
                0.00    0.00       1/1           Lattice::calcSusceptibility(double, double) [23]
                0.00    0.00       1/1           Lattice::~Lattice() [25]
                0.00    0.00       1/1           calcAutocor(unsigned int, double*, double*) [21]
-----------------------------------------------
               21.85    0.00 167772160/167772160     main [1]
[2]     35.1   21.85    0.00 167772160         Lattice::metropolis(unsigned int) [2]
-----------------------------------------------
                0.00    5.49   32768/32768       main [1]
[3]      8.8    0.00    5.49   32768         Lattice::wolff(unsigned int) [3]
                1.63    0.89   16394/16394       Lattice::growClusterPos(unsigned int) [4]
                1.19    0.89   16374/16374       Lattice::growClusterNeg(unsigned int) [5]
                0.89    0.00   32768/32768       Lattice::flipCluster() [7]
                0.00    0.00   32768/9498749     HashTable::insert(unsigned int) [8]
-----------------------------------------------
                               4732235             Lattice::growClusterPos(unsigned int) [4]
                1.63    0.89   16394/16394       Lattice::wolff(unsigned int) [3]
[4]      4.0    1.63    0.89   16394+4732235 Lattice::growClusterPos(unsigned int) [4]
                0.66    0.00 17347795/34694250     HashTable::find(unsigned int) [6]
                0.24    0.00 4732235/9498749     HashTable::insert(unsigned int) [8]
                               4732235             Lattice::growClusterPos(unsigned int) [4]
-----------------------------------------------
                               4733746             Lattice::growClusterNeg(unsigned int) [5]
                1.19    0.89   16374/16374       Lattice::wolff(unsigned int) [3]
[5]      3.3    1.19    0.89   16374+4733746 Lattice::growClusterNeg(unsigned int) [5]
                0.66    0.00 17346455/34694250     HashTable::find(unsigned int) [6]
                0.24    0.00 4733746/9498749     HashTable::insert(unsigned int) [8]
                               4733746             Lattice::growClusterNeg(unsigned int) [5]
-----------------------------------------------
```

```
                0.66    0.00 17346455/34694250     Lattice::growClusterNeg(unsigned int) [5]
                0.66    0.00 17347795/34694250     Lattice::growClusterPos(unsigned int) [4]
[6]      2.1    1.31    0.00 34694250         HashTable::find(unsigned int) [6]
-----------------------------------------------
                0.89    0.00   32768/32768       Lattice::wolff(unsigned int) [3]
[7]      1.4    0.89    0.00   32768         Lattice::flipCluster() [7]
-----------------------------------------------
                0.00    0.00   32768/9498749     Lattice::wolff(unsigned int) [3]
                0.24    0.00 4732235/9498749     Lattice::growClusterPos(unsigned int) [4]
                0.24    0.00 4733746/9498749     Lattice::growClusterNeg(unsigned int) [5]
[8]      0.8    0.48    0.00 9498749         HashTable::insert(unsigned int) [8]
-----------------------------------------------
                0.25    0.00   16384/16384       main [1]
[9]      0.4    0.25    0.00   16384         Lattice::calcTotalEnergy() [9]
-----------------------------------------------
                0.07    0.00   16384/16384       main [1]
[10]     0.1    0.07    0.00   16384         Lattice::calcAveragePhi() [10]
-----------------------------------------------
                                                <spontaneous>
[11]     0.1    0.07    0.00                 HashTable::~HashTable() [11]
-----------------------------------------------
                                                <spontaneous>
[12]     0.0    0.03    0.00                 Lattice::printLattice() [12]
-----------------------------------------------
                0.00    0.00      11/11          Lattice::Lattice(double, double, unsigned int,...
[19]     0.0    0.00    0.00      11         std::vector<siteNeighbors*, std::allocator...
-----------------------------------------------
                0.00    0.00      11/11          Lattice::Lattice(double, double, unsigned int...
[20]     0.0    0.00    0.00      11         std::vector<double, std::allocator<double> >...
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[21]     0.0    0.00    0.00       1         calcAutocor(unsigned int, double*, double*) [21]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[22]     0.0    0.00    0.00       1         Lattice::calcSpecificHeat(double, double) [22]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[23]     0.0    0.00    0.00       1         Lattice::calcSusceptibility(double, double) [23]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[24]     0.0    0.00    0.00       1         Lattice::Lattice(double, double, unsigned int...
                0.00    0.00      11/11          std::vector<double, std::allocator<double> >...
                0.00    0.00      11/11          std::vector<siteNeighbors*, std::allocator...
                0.00    0.00       1/1           HashTable::HashTable(unsigned int) [26]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[25]     0.0    0.00    0.00       1         Lattice::~Lattice() [25]
-----------------------------------------------
                0.00    0.00       1/1           Lattice::Lattice(double, double, unsigned int...
[26]     0.0    0.00    0.00       1         HashTable::HashTable(unsigned int) [26]
                0.00    0.00       1/1           node** std::fill_n<node**, unsigned int, node*>...
                0.00    0.00       1/1           std::vector<node*, std::allocator<node*> >::operator=(...
-----------------------------------------------
                0.00    0.00       1/1           HashTable::HashTable(unsigned int) [26]
[27]     0.0    0.00    0.00       1         std::vector<node*, std::allocator<node*> >::operator=(...
-----------------------------------------------
                0.00    0.00       1/1           HashTable::HashTable(unsigned int) [26]
[28]     0.0    0.00    0.00       1         node** std::fill_n<node**, unsigned int, node*>(node**,...
-----------------------------------------------

[...]
```

B.1.2 Debugging

Of course, before we worry about the speed at which the code runs, we have to make it compile and run (correctly) in the first place. Debugging is one of the central aspects of programming, and potentially one of the most time-consuming. Fortunately, by using debuggers, programs designed to help us troubleshoot code and track down errors, the task can be accomplished much more quickly and painlessly. Those already familiar with debuggers may feel that we are needlessly stressing a relatively trivial point. However, we have met advanced computer science students at Amherst College (and Wilson reports meeting professional scientists) who have not heard of gdb and attempt to 'debug' their code by inserting `print` commands on various lines (an approach which is both horribly inefficient and also potentially misleading).

The GNU Debugger **gdb** is the standard debugger used in Unix and Linux environments. Programs to be debugged with gdb need to be compiled with the -ggdb option (again, just add it to the CFLAGS (or equivalent) field of the Makefile – see Code Snippet B.2, below). To debug a program with gdb, run `gdb <program_name>`. This will start a special gdb shell in which the following commands will come in handy:

- `set args <args>` tells gdb what command-line arguments to pass to the program when it is run.
- `run` simply starts executing the program.
- `break <file:line>` sets a breakpoint at the specified line in the specified file (a method name can be used as well). When the breakpoint is reached, gdb will stop execution so that the state of the program can be investigated.
- `print <options>` prints out the requested information on values of variables, pointers, and so on.
- `n` prints out and executes the next line in the source code.
- `c` continues execution of the program until the next breakpoint is reached.
- `bt` prints out a backtrace of all the methods that are currently active, so that the thread of execution can be traced.
- Finally, of course, `help <command>` can be used to access much more information about many more commands.

It should be clear how using just these simple capabilities of gdb can simplify the investigation of errors and dramatically speed up the debugging process.

Another valuable debugging tool (especially for those programming in languages without automatic garbage collection) is **LeakTracer**,[B.5] which tracks memory allocation and deallocation to help the programmer identify and correct memory leaks. Memory leaks occur when memory resources are claimed by the program but not released when no longer needed. A program with a memory leak can needlessly fills up the available RAM and force its own data (and that of whatever other programs might also be running) to be swapped out to the hard disk. Since accessing data on a hard disk is far slower than reading from RAM, memory leaks can disastrously impact performance. Since

[B.5] http://www.andreasen.org/LeakTracer/ – last accessed 10 May 2006.

LeakTracer is a relatively small tool specific to C++ (there are analogous tools for other languages), we will not give any details of its operation.

B.1.3 Version Control Systems

Wilson has declared that use of a version control system is the first of the four characteristics that distinguish professional programmers from amateurs (the other three being automation of repetitive tasks, systematic testing and the use of debuggers). Version control systems are critical to software development projects that involve collaboration between multiple programmers, but they also make it easy to discard changes that didn't work out and revert to earlier versions of the code, which can be very helpful even for single-person projects.

Consider what could go wrong if several collaborators are all developing a program, as is generally the case in both scientific and commercial software engineering. Several people working on the project could all try to make changes at the same time, wiping out each other's work in the process. Under version control systems, each programmer receives a local working copy of the code from a central repository, and submits any changes back to that repository. If multiple programmers then edit the code, the version control system will do its best to merge all of the changes into the master copy in the repository, alerting the user to any incompatible conflicts it is unable to reconcile. Additionally, the version control system will keep information on all older versions of the code, making it easy to revert to an earlier state should problems arise with newer versions.

Concurrent Versions System (CVS)[B.6] is one of the oldest and most widely used version control systems around. Subversion,[B.7] our software of choice, is a newer development that aims to replace CVS and has already had much success doing so. Using Subversion is easy, especially for users who don't need to worry about administering the repository. They only need to know a handful of simple commands, namely

- `svn co` checks out a working copy of the master files in the repository for local use;
- `svn update` synchronizes the local working copy with the repository, merging in any changes that have been made to the master files, so long as they don't conflict with changes made to the local copy;
- `svn diff <current:old> <file>` displays any differences between the specified current and old versions of the file;
- `svn ci -m "<Log message>"` commits to the repository the changes made to the local working copy, unless there is a conflict with other changes already made to the master copy;
- `svn merge -r <current:old> <file>` reverts from the current version of the file to the specified old version, (keeping a copy of the current version in the repository);
- and finally, `svn help` gives more information about all of the above and more.

[B.6] http://www.nongnu.org/cvs/ – last accessed 10 May 2006.
[B.7] http://subversion.tigris.org/ – last accessed 10 May 2006.

The only additional commands administrators really need to know are `svnadmin create`, which creates a new repository, and `svn import`, which commits a new, unversioned file or tree into the repository.

Despite the advantages of version control, we did not make use of it until near the end of our work, in part because we were initially unfamiliar with the concept, in part because the collaboration and reversion problems were not especially relevant to our situation. Since all of our code was developed and maintained by a single individual, we had no worries of conflicts between collaborators. Additionally, the code base was relatively small and by making occasional backups and keeping a detailed Changelog we were generally in a position to revert relatively painlessly, if necessary. However, reverting through a version control system is still easier than doing so manually, a consideration that eventually encouraged us to set up subversion repositories for our code.

B.1.4 Automated Builds

Typing `gcc -c -Wall -ansi -I/pkg/chempak/include dat2csv.c` *once is bad enough.* – G. V. Wilson

As a rule, anything worth doing repeatedly on the computer is worth automating. Compiling and linking code to form working programs is a useful example: not only will this be done very often during development and production, but it can often be a complicated process in which certain steps depend on others being correctly performed in the proper order. Fortunately, a program called **Make**[B.8] that uses an instruction file known as a Makefile makes it is easy to fully and automatically compile and link (or simply 'build') programs with a single command.

We'll be frank with our recommendation: if you aren't using a Makefile, start using a Makefile. Although Make has grown into a monster of a program over the more than thirty years since it was first written, it is simple enough to set up a basic Makefile such as that in Code Snippet B.2 that will handle building relatively simple programs like ours. Comprehensive tutorials and introductions to some of Make's many advanced features can easily be found online.

Code Snippet B.2: Sample Makefile (for ϕ^4 simulations)

```
CC = g++
CFLAGS = -O3 -Wall
INCLUDE_FLAGS = -I/home/daschaich/gsl/include
LIBRARY_FLAGS = -L/home/daschaich/gsl/lib -lgsl -lgslcblas -lm

EXECUTABLE = Simulation
OBJECT_FILES = HashTable.o Lattice.o Simulation.o

main: HashTable.o Lattice.o Simulation.o
$(CC) $(CFLAGS) $(INCLUDE_FLAGS) $(LIBRARY_FLAGS) -o $(EXECUTABLE) $(OBJECT_FILES)

clean:
rm -f $(EXECUTABLE) *.o
```

[B.8] http://www.gnu.org/software/make/ – last accessed 10 May 2006

```
Simulation.o: Simulation.cpp Lattice.cpp Lattice.hh
$(CC) $(CFLAGS) $(INCLUDE_FLAGS) -c Simulation.cpp

Lattice.o: Lattice.cpp Lattice.hh
$(CC) $(CFLAGS) $(INCLUDE_FLAGS) -c Lattice.cpp

HashTable.o: HashTable.hh HashTable.cpp
$(CC) $(CFLAGS) $(INCLUDE_FLAGS) -c HashTable.cpp
```

We'll quickly point out some highlights of Code Snippet B.2. The first few lines are simply defining variables; the real action begins with the line `main: ...`.[B.9] The word or file to the left of the colon is known as the target, those to the right are dependencies, and any indented lines below are commands. Make can run on a particular target (for example, `make clean`); if no target is specified, it defaults to main. First Make checks to see that the all the dependencies exist and are up to date. That is, it verifies that no dependency's dependency was modified more recently that the dependency itself. Once all dependencies have been checked, Make executes (through the shell) all the commands listed below the target being run.[B.10] As mentioned above, it is very easy to change the compiler flags used (for instance, adding -pg or -ggdb) simply by editing the CFLAGS field.

The output produced by running `make clean && make` in a directory containing the above makefile would be

```
rm -f Simulation *.o
g++ -O3 -Wall -I/home/daschaich/gsl/include -c HashTable.cpp
g++ -O3 -Wall -I/home/daschaich/gsl/include -c Lattice.cpp
g++ -O3 -Wall -I/home/daschaich/gsl/include -c Simulation.cpp
g++ -O3 -Wall -I/home/daschaich/gsl/include -L/home/daschaich/gsl/lib \
    -lgsl -lgslcblas -lm -o Simulation HashTable.o Lattice.o Simulation.o
```

Try typing that ten times fast![B.11]

B.1.5 Integrated Development Environments

Finally, we will very quickly point out an area where we disagree with Wilson. He remarks both in his article and in his online lectures that serious programmers should use integrated development environments (IDEs), which are graphical programs that include a wide variety of bells and whistles such as built-in debuggers and 'class browsers' that

[B.9] It may be worth noting that our include and library flags are simply those needed to use the GNU Scientific Library in our programs.

[B.10] Note that while most programs are not concerned with whitespace, Make requires that the indentation before each command be a tab, not just four spaces.

[B.11] Line break manually added for formatting purposes.

summarize the classes, methods and variables in a particular program. For example, Eclipse[B.12] is a popular IDE.

Wilson's only apparent objection to leaner text editing programs such as Vi and Emacs is simply that they're "ancient". Presumably, he believes that IDEs and the various services they offer make it more efficient to develop and manage code. We (along with many others) feel that IDEs present no benefits that cannot be gained by simply using the tools introduced above – debuggers, Makefiles and version control. Indeed, the relative simplicity of these basic tools may well make them easier and more efficient to use than complex and often bloated IDEs.

B.2 Programming Languages

Object Oriented Programming is like teenage sex. Everyone talks about it, nobody's actually doing it, and the few who are doing it are doing it badly. (Original author unknown)

The choice of a programming language obviously plays a major role to the nuts and bolts of scientific programming. C++ is currently challenging Fortran to become the dominant programming language of high-energy physics simulations, and is the language we used in our work.

Some thought went into this choice. In addition to personal matters of prior knowledge, we also considered general questions of programming efficiency and the speed of the resulting programs. For instance, Fortran, designed long ago specifically for scientific programming, has the reputation of producing extremely fast-executing programs. On the other hand, Fortran suffers from antiquated conventions derived from the days of punch-card computing that can make it difficult and frustrating to use, especially for complex programs. This is true at least of the classic FORTRAN 77; there are more modern versions (Fortran 90, Fortran 95 and Fortran 2003) that may be more worthy of consideration now that decent open-source compilers[B.13] are becoming available.

Object-orientation programming (OOP) is a popular programming paradigm that many (such as Wilson [68]) claim makes it easier to design and write functioning programs in the first place. Many of the programming languages most widely used today are object-oriented, including C++, Python, Java and C#. These languages all have their relative strengths and weaknesses, of course. C++, although it gives the programmer a good deal of flexibility and has a reputation for producing fast-running programs, is often criticized for being excessively complex, while Python is commonly singled out as especially friendly to programmers. Java's main strength is its portability, which it achieves by running on a virtual machine, potentially at some cost to execution speed. Java also attempts to be more programmer-friendly by automatically performing such tasks as array bounds

[B.12] http://www.eclipse.org/ – last accessed 10 May 2006.

[B.13] Such as g95 (http://www.g95.org – last accessed 10 May 2006) and gfortrtan (http://gcc.gnu.org/fortran/ – last accessed 10 May 2006)

checking (to prevent attempts to access arrays beyond their bounds, which may cause segmentation faults) and 'garbage collection' (which reduces the need for the programmer to manually manage memory and can limit the severity of memory leaks). Though these features can make the programmer's task easier, they can also reduce flexibility and possibly impact performance. C# is a relatively recent development that attempts to simplify and streamline C++ by adopting many of the characteristics of Java.

Unlike Java and C#, C++ supports both procedural and object-oriented programming. This can be beneficial, since OOP adds some overhead that slows down execution. For the purposes of our work, where the programs were relatively short and straightforward, and the main concern was speed of execution as opposed to complexity of design, the advantages of object orientation were not clear. Indeed, earlier programs tended to involve more OOP features than later ones, which were gradually removed over time to increase the efficiency of the code.

Early in our work we experimented with C, C++, Java and Python, writing roughly identical random walk programs in each of them. Although we observed that the C/C++ versions ran the fastest and the Python versions the slowest (in agreement with the conventional wisdom), we did not perform systematic tests to seriously compare efficiency across languages. In coming months we intend to rewrite our code in Fortran 90/95 in order to explore that language and (informally) see how it performs relative to C++ and the others.

What will the scientific programming language of the next century look like? Nobody knows, but it will be called 'Fortran'. (Popular joke circa 1980)

B.3 Data Structures

Lattice simulations rely heavily on data structures used to represent the lattice and store information. Goodrich and Tamassia [22] is a suitable introductory textbook that explains many data structures at some length. In this section we will very briefly survey the types of data structures used for my simulations, discuss some of the results, and present an implementation of a basic hash table used extensively. The following discussion will likely not interest those with any knowledge of data structures.

The programs used in the course of my work required only simple data structures, which needed only three abilities: inserting data into the structure, searching for data in the structure, and removing data from the structure. Since Monte Carlo simulations involve a small number of functions running a large number of times, the speed of the data structures and their methods was of the utmost importance.

To allow a more concrete discussion, let's consider the specific data structures used in the self-avoiding random walk programs introduced in Section 3.2. These programs needed to keep track of all the sites already visited in the course of the walk, and thus used all three of the functions introduced above: when a new site was visited, it had to be inserted into the data structure; when a new site was considered, the data structure

had to be searched to see if it had already been visited; and once the walk was complete the structure had to be cleared and prepared for the next run.

Initially we simply used a d-dimensional array to keep track of all the sites visited. This is unquestionably the simplest option: make an array of all sites that could be visited (N in each direction) containing Boolean (true/false) values. Both insert and search take (very short) constant time: when a site is visited, change its value to true; to search, simply access the value of the site. However, clearing the array after the run is finished requires scanning through all the sites in the array, which takes time $\mathcal{O}(N^d)$, where N is the number of steps in the walk and d is the number of dimensions in which the walk takes place.

An alternate approach that increases the speed of the clearing operation is to use a linked list. In a linked list, values (in this case the identities of sites that have been visited) are wrapped in 'nodes' (or 'links') that also include pointers to the next nodes in the list. To insert, just wrap the value to be inserted into a node and set up the pointers to add it to the list. This takes constant ($\mathcal{O}(1)$) time, just as with the array. To clear the list (or remove an element from it), simply trace through it from node to node, deleting them as we go (and rearranging pointers if necessary). This clearly takes time $\mathcal{O}(N)$, much faster than the array implementation. However, to search we now have to scan through all the contents in the list, which takes time $\mathcal{O}(N)$, much worse than the array implementation.[B.14]

The array (or vector, which is essentially an array with certain extra features) and linked list are two of the most basic and commonly used approaches to data structures. The strengths of these two implementations are complementary: while vectors are good at random access, linked lists take up less space and thus require less time to initialize or reset, but more to search. Of course, what's desired is a data structure that can perform all operations as quickly as possible. In the case addressed above, it is easy to see that a combination of an array and linked list will allow for constant-time searches and insertions, and clearings that take only $\mathcal{O}(N)$. Simply insert visited sites into both an array and a linked list, search using the array, and use the linked list to determine which N sites in the array need to be reset at the end of the run. Although this works, it is inelegant and somewhat inefficient: multiple data structures are required for what is really a very simple task, and the implementation requires every value to be inserted into and removed from both structures, even though clearing both is now much more efficient.

A more elegant solution is to use a hash table, which is essentially an array of linked lists (see Goodrich and Tamassia [22, Chap. 8] for much more detail). When a site is to be added to the hash table, it is passed through a hash function that pseudorandomly assigns it to one of the lists (the purpose of the randomness is to keep all of the lists roughly the same length). When we search for a site, we can calculate the list into which it would have been sorted, and only that list (which has average length N/n, where N is

[B.14] Alternately, we could impose an overall order the list by some criterion, which would allow you to perform binary searches on insertions and searches, meaning that both would run in time $\mathcal{O}(\log(N))$ (where the logarithm is base-2). The choice of whether to use an ordered or an unordered list depends on the size of N and whether searches or insertions will be more common.

the number of elements in the table and n is the number of lists) needs to be scanned. It can be shown that *on average* searching a hash table takes constant time, as does insertion. Removing a single element also takes constant time on average, and clearing the whole hash table takes only time $\mathcal{O}(N)$. Thus the hash table has the same asymptotic behavior as the array/list combination discussed above, but is simpler and more elegant.

The benefits of using a hash table over a plain array or linked list can be dramatic. As mentioned above, we initially used an array to keep track of visited sites during self-avoiding random walks. When that was replaced with an early (and slightly less efficient) version of the hash table presented below in Code Snippets B.3 and B.4, the four-dimensional self-avoiding random walk program's running time decreased by roughly a factor of 10^4. In addition, it is a good idea to adjust the number of lists in the hash table so that the average list is only a few elements long. When we increased the number of lists in the hash table from 97 to one quarter of the size of the lattice in ϕ^4 simulations, the programs' running times decreased by around 20%.[B.15]

Because the hash table included in the C++ Standard Template Library (STL) is designed to handle more general requirements, its implementation is much more complex than we needed for our work. Accordingly, we wrote a simple hash table that handles only the three functions listed above. In fact, since this hash table uses lists of 'node' structs, any value can be removed from the hash table by simply **delete**-ing its wrapper node.[B.16] Thus only **insert** and **find** need to be implemented. **gprof** analyses shows that each of these two functions runs in an average time of 40-50 nanoseconds on a modern processor.

Code Snippet B.3: Header file for basic hash table

```
// -------------------------------------------------------------------
// Phi4/HashTable.hh
// Basic hash table with chaining
// David Schaich -- daschaich@gmail.com
// Created 23 October 2005
// Last modified 25 January 2006
// -------------------------------------------------------------------



// -------------------------------------------------------------------
// Avoid multiple inclusion in a single executable
#ifndef _HASHTABLE_HH
#define _HASHTABLE_HH
// -------------------------------------------------------------------



// -------------------------------------------------------------------
// Include directives
#include <vector>
// -------------------------------------------------------------------
```

[B.15] 97 was chosen because having a prime number of lists in the hash table increases the randomness of the hash function.

[B.16] Assuming that values are only removed from the hash table when it is cleared following, for example, the creation and inversion of a Wolff cluster. If only one node is to be removed while keeping all others in the table, the list structure of pointers from node to node would need to be updated to avoid segmentation faults. This would be accomplished most easily through the creation of a **remove** function, which would run in constant time on average.

```
// ------------------------------------------------------------------
// Structs instead of classes - reduce OOP overhead
// Struct of a node in a slot in the hashtable
struct node {
    unsigned int value;
    node* next;
};
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Class declaration for HashTable
class HashTable {
    public:
        // Constructor, destructor
        HashTable(unsigned int tableNumber);
        HashTable();
        ~HashTable();

        // Member functions
        void insert(unsigned int site);
        bool find(unsigned int site);

        // Member data
        unsigned int size;
        unsigned int tableNumber;
        unsigned int mod;
        std::vector<node*> table;       // Vector of lists of nodes
};
// ------------------------------------------------------------------



// ------------------------------------------------------------------
#endif // _HASHTABLE_HH
// ------------------------------------------------------------------
```

Code Snippet B.4: Implementation of basic hash table

```
// ------------------------------------------------------------------
// Phi4/HashTable.cpp
// Hash table designed for lists of nodes
// David Schaich -- daschaich@gmail.com
// Created 23 October 2005
// Last modified 25 January 2006
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Include directives
#include "HashTable.hh"
#include <vector>
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Constructors and destructor
HashTable::HashTable(unsigned int numberOfTables) {
    size = 0;
    tableNumber = numberOfTables;
    std::vector<node*>* temp = new std::vector<node*>(tableNumber, NULL);
```

```
    table = *temp;

    mod = tableNumber - 1;
}

HashTable::HashTable() {
    HashTable(4093);
}

HashTable::~HashTable() {}
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Member functions - insertion, searching
void HashTable::insert(unsigned int site) {
    size++;
    unsigned int index = (17 * site - 97) & mod;

    node* toAdd = new node;
    toAdd->value = site;

    toAdd->next = table[index];
    table[index] = toAdd;
}
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// This returns whether or not site is in cluster
bool HashTable::find(unsigned int site) {
    unsigned int index = (17 * site - 97) & mod;
    node* temp = table[index];

    while (temp != NULL) {
        if (site == temp->value)
            return true;
        temp = temp->next;
    }
    return false;
}
// ------------------------------------------------------------------
```

B.4 Efficiency Tricks

Over the course of our work, we stumbled across a number of small tricks to decrease the running time of our programs, many of which had surprisingly large effects. In this section we'll briefly list some of them. Even if the specific circumstances in which we applied these tricks are not present in any particular program, we hope that this discussion suggests some generally applicable ideas and considerations for producing faster-running programs.

The most obvious trick is clearly applicable to a wide variety of situations: don't calculate anything more often than needed. The Ising model presents a simple example of this principle. To perform the Metropolis algorithm, we need to calculate the exponential of a quantity proportional to the change in energy resulting from flipping a single spin. Calculating this value each time it's needed would be simple but very inefficient, since exponentials are typically calculated through a brief power series, a relatively compli-

cated operation. The trick to this situation is to recognize that the change in energy can take one of only five possible values. If we simply calculate each of the five corresponding exponentials once and store the results in an array, we can reuse them instead of recalculating them, saving considerable time. Similarly, the exponential factor $e^{-2\beta J}$ used by the Wolff cluster algorithm is simply constant and so only needs to be calculated once. We do exactly this in our Ising model simulations, presented below in Code Snippets C.4 through C.6.

Note that even a slight increase in the efficiency of a method can have significant effects. From the code profile presented above in Code Snippet B.1, we see that the Metropolis method was called well over one hundred million times in the course of a very short (60 second) simulation. If a single microsecond were added to the running time of this method, the runtime of the whole program would nearly quadruple.

A similar procedure can be performed for boundary conditions, and the calculation of neighboring sites more generally. Recall that back in Section 3.3 we spent some time discussing the relative merits of helical and periodic boundary conditions. We saw that implementing periodic boundary conditions in the intuitive way requires performing many more multiplications than necessary for helical boundary conditions, which may make them slower since multiplication is a relatively slow operation (though not as slow as calculating exponentials). Another relatively slow operation is the mod operation %, which essentially needs to perform divisions in order to calculate remainders. Newman and Barkema [40, Chap. 13], who recommend helical boundary conditions for the reasons just discussed, also encourage the use of lattices of size $L \propto 2^n$, since in this case the mod function can be replaced by a much simpler and quicker bitwise AND (&) operation.[B.17]

However, there's a smarter, more efficient way to go about all of this: calculate all the neighboring sites and boundary conditions once (and only once) at the beginning of the simulation and store them in a suitable data structure. All subsequent calculations of neighbors and boundary conditions can then be performed simply by looking up the necessary information in the data structure, avoiding repeated multiplications and modulos. The data structure will take up some extra memory, but we typically have plenty of that and are much more interested in speed than in memory usage.[B.18] Note that trading space for speed is a common possibility in data structures and algorithms. This is what we do in our ϕ^4 model simulations, which can be found below in Code Snippets C.7 through C.9.

Speaking of data structures, one final observation we made is that the number of lists used in our hash table has a significant impact on the running time of our programs. If too few lists are used, the chains in them become longer and thus require more time to search through. If too many are created, some of them may just be unused, wasting space (and time, as we scan through the hash table to clear it). We eventually arbitrarily set the number of lists in the hash table equal to one quarter the number of sites in

[B.17] We actually replaced many mod operations with if–else conditions and observed a significant increase in the speed of our program as a result.

[B.18] Though perhaps we should not be quite so glib here, since if RAM is completely filled data has to be swapped to and from the hard disk, which generally takes a *very* long time.

the lattice, most likely larger than necessary.[B.19] It might be an interesting exercise to explore the relation between the number of lists in the hash table and the speed of the program. We did not perform such an analysis.

[B.19] Additionally, hash tables of this size do not have a prime number of lists. Containing a prime number of lists can increase the efficiency of a hash table, as discussed in Goodrich and Tamassia [22].

Appendix C: Sample Code

This chapter includes full C++ code for several of the programs written for our work. Only full working programs are included, but in the interests of space, not all of our programs are included, only representative samples. The organization of code within this appendix parallels that in the thesis – after random walk simulations comes Ising model code, and finally programs for calculating the critical phase transition line and soliton masses in ϕ^4 theory. In Section C.5 we include some excerpts from the Mathematica code used to analyze the data produced by our C++ simulations.

C.1 Random Walks

The following short random walk programs were among the first we created. We made little use of object-oriented programming. The programs take no command-line parameters and print results directly to a file. Later we deemed it simpler to print results to standard output and then pipe them to a file from the shell. For simplicity and economy of space we present only two-dimensional programs. Analogous random walk programs were created in three and four dimensions, and the results of those simulations presented above in Section 3.2.

Code Snippet C.1: Basic random walk

```
// RandomWalk2D/Basic2D.cpp
// Basic random walk in two dimensions
// Creates file of positions over time
// David Schaich
// Created 15 April 2005
// Last modified 17 February 2006

using namespace std;

#include <iostream>         // For cout, fopen, fprintf, etc.
#include <gsl/gsl_rng.h>    // Random number generators

int main(int argc, char* argv[]) {
    FILE* output;
    output = fopen("Basic2D.csv", "w");

    int sampleSize = 100000;
    int minLength = 1;
    int maxLength = 150;

    int x = 0;
    int y = 0;
```

```
    int stepLength = 1;
    int direction = 4;      // To be computed randomly

    double dist = 0;
    double distTotal = 0;
    double distAve = 0;

    // Mersenne Twister generator
    gsl_rng* generator = gsl_rng_alloc (gsl_rng_mt19937);

    for (int totalSteps = minLength; totalSteps <= maxLength; totalSteps+=1) {
        distTotal = 0;

        for (int test = 0; test < sampleSize; test++) {
            x = y = 0;
            for (int step = 0; step < totalSteps; step++) {
                direction = gsl_rng_get(generator) % 4;
                if       (direction == 0)    x += stepLength;
                else if  (direction == 1)    y += stepLength;
                else if  (direction == 2)    x -= stepLength;
                else if  (direction == 3)    y -= stepLength;
            }

            // Compute distance squared for each test
            dist = x*x + y*y;
            distTotal += dist;
        }

        // Compute average distance and print out
        distAve = distTotal / sampleSize;
        cout << totalSteps << " " << distAve << endl;

        fprintf(output, "%i,\t%lf\n", totalSteps, distAve);
    }
    gsl_rng_free(generator);
    fclose(output);
    return 0;
}
```

Code Snippet C.2: Nonreversal random walk

```
// RandomWalk2D/Nonreversal2D.cpp
// Nonreversal random walk in two dimensions
// Creates file of positions over time
// David Schaich
// Created 24 April 2005
// Last modified 17 February 2006

using namespace std;

#include <iostream>         // For cout, fopen, fprintf, etc.
#include <gsl/gsl_rng.h>    // Random number generators

int main(int argc, char* argv[]) {
    FILE* output;
    output = fopen("Nonreversal2D.csv", "w");

    int sampleSize = 100000;
    int minLength = 1;
    int maxLength = 150;

    int x = 0;
    int y = 0;
    int stepLength = 1;
    int dir = 4;                // (Random) Direction of potential step
    int prevDir = 4;            // Direction of previous step
```

```
    double dist = 0;
    double distTotal = 0;
    double distAve = 0;

    // Mersenne Twister generator
    gsl_rng* generator = gsl_rng_alloc(gsl_rng_mt19937);

    for (int totalSteps = minLength; totalSteps <= maxLength; totalSteps+=1) {
        distTotal = 0;

        for (int test = 0; test < sampleSize; test++) {
            x = y = 0;
            prevDir = 4;
            for (int step = 0; step < totalSteps; step++) {
                dir = gsl_rng_get(generator) % 4;

                if       (dir == 0 && prevDir != 2)    x += stepLength;
                else if  (dir == 1 && prevDir != 3)    y += stepLength;
                else if  (dir == 2 && prevDir != 0)    x -= stepLength;
                else if  (dir == 3 && prevDir != 1)    y -= stepLength;
                else step--;        // Didn't step, so doesn't count

                prevDir = dir;      // Works whether or not step taken
            }
            // Compute distance squared for each test
            dist = x*x + y*y;
            distTotal += dist;
        }

        // Compute average distance and print out
        distAve = distTotal / sampleSize;
        cout << totalSteps << " " << distAve << endl;

        fprintf(output, "%i,\t%lf\n", totalSteps, distAve);
    }
    gsl_rng_free(generator);
    fclose(output);
    return 0;
}
```

The self-avoiding random walk uses a hash table similar to that presented in Section B.3, above. The main difference is that the hash table used here is more object oriented (with the corresponding increase in overhead) and designed to contain triplet points (so that it could be used for both two- and three-dimensional random walks) instead of plain integers.

Code Snippet C.3: Self-avoiding random walk

```
// RandomWalk2D/Selfavoiding2D.cpp
// Self-avoiding random walk in two dimensions
// Creates file of positions over time
// David Schaich -- daschaich@gmail.com
// Created 24 April 2005
// Last modified 17 February 2006

using namespace std;

#include <cstdlib>
#include <iostream>         // For cout, fopen, fprintf, etc.
#include <gsl/gsl_rng.h>    // Random number generators
#include "../HashTable/HashTable.cpp"

int main(int argc, char* argv[]) {
    FILE* output;
```

```
output = fopen("Selfavoiding2D.csv", "w");

int sampleSize = 100000;
int minLength = 1;
int maxLength = 150;

int x = 0;
int y = 0;
int stepLength = 1;
int direction = 4;      // To be computed randomly
int aborted = 0;

double dist = 0;
double distTotal = 0;
double distAve = 0;

// Mersenne Twister generator
gsl_rng* generator = gsl_rng_alloc(gsl_rng_mt19937);

// Keeps track of path that has been followed so far
// and data about where the walker can go from the current position
int potSteps[8];
int openPaths = 0;
bool counts = true;
HashTable* temp = new HashTable();
HashTable path = *temp;

for (int totalSteps = minLength; totalSteps <= maxLength; totalSteps++) {
    distTotal = 0;

    for (int test = 0; test < sampleSize; test++) {
        x = y = 0;
        path.clear();
        path.insert(0, 0, 0);

        for (int step = 0; step < totalSteps; step++) {
            // Figure out possible steps
            openPaths = 0;
            if (!path.find(x + 1, y, 0) ) {
                potSteps[2 * openPaths] = x + 1;
                potSteps[2 * openPaths + 1] = y;
                openPaths++;
            }
            if (!path.find(x, y + 1, 0) ) {
                potSteps[2 * openPaths] = x;
                potSteps[2 * openPaths + 1] = y + 1;
                openPaths++;
            }
            if (!path.find(x - 1, y, 0) ) {
                potSteps[2 * openPaths] = x - 1;
                potSteps[2 * openPaths + 1] = y;
                openPaths++;
            }
            if (!path.find(x, y - 1, 0) ) {
                potSteps[2 * openPaths] = x;
                potSteps[2 * openPaths + 1] = y - 1;
                openPaths++;
            }

            // If nowhere to go, abort test and start a new one
            if (openPaths == 0) {
                // Check to make sure there are no infinite loops
                aborted++;
                if (aborted > 10000) {
                    cout << "Aborting test " << test << " of ";
                    cout << totalSteps << " steps" << endl;
                    aborted = 0;
                }
                counts = false;     // This iteration didn't count
                break;              // Out of step loop into test loop
            }
```

```
                direction = gsl_rng_get(generator) % openPaths;
                x = potSteps[2 * direction];
                y = potSteps[2 * direction + 1];

                path.insert(x, y, 0);
            }

            if (counts) {
                // Compute distance squared for each test
                dist = x*x + y*y;
                distTotal += dist;
            }
            else {
                test--;
                counts = true;
            }
        }

        // Compute average distance and print out
        distAve = distTotal / sampleSize;
        cout << totalSteps << " " << distAve << endl;

        fprintf(output, "%i,\t%lf\n", totalSteps, distAve);
    }
    gsl_rng_free(generator);
    fclose(output);
    return 0;
}
```

C.2 Ising Model

In this section we include much of the code used in our Ising model simulations – a lattice class and header file, along with a basic simulation file. Although both the Metropolis and Wolff cluster algorithms are implemented in Code Snippet C.5, the simulation program in Code Snippet C.6 uses only the Metropolis algorithm; a nearly identical program used only the Wolff cluster algorithm. We did this in order to compare the behavior of each algorithm on its own. Later, as shown below in Section C.3, we combined the two algorithms in our ϕ^4 theory simulations in order to maximize efficiency. This lattice uses the helical boundary conditions described above in Section 3.3.

The program assumes a two-dimensional rectangular lattice and takes five integer input parameters from the command line: the two dimensions of the lattice, the number of iterations for equilibration, the number of iterations for statistics, and the temperature at which to run the simulation.

Code Snippet C.4: Header file for Ising model lattice

```
// ------------------------------------------------------------------
// Ising2D/Lattice.hh
// Lattice of spins for Ising Model simulation
// Header file contains data and method declarations
// David Schaich -- daschaich@gmail.com
// Created 1 July 2005
// Last modified 19 March 2006
// ------------------------------------------------------------------



// ------------------------------------------------------------------
```

```
// Frontmatter and include directives
// Avoid multiple inclusion
#ifndef _LATTICE_HH
#define _LATTICE_HH

#include "HashTable/HashTable.hh"   // Hash table for searching cluster
#include <vector>                   // Lattice is vector of vectors
#include <gsl/gsl_rng.h>            // Random number generators
#include <gsl/gsl_sf_exp.h>         // Exponential functions
// ----------------------------------------------------------------



// ----------------------------------------------------------------
// Class definition
class Lattice {
    public:
        // ------------------------------------------------------
        // Data!
        // Member data
        std::vector<int> lattice;       // Values must be 1, -1
        unsigned int xDim;              // x dimension of lattice
        unsigned int yDim;              // y dimension of lattice
        unsigned int latticeSize;       // Number of sites in lattice

        float temp;             // kT in energy units (k = 1)
        float beta;             // 1/kT in energy units (k = 1)
        double randomU;         // Random number in range [0,1)
        double probability;     // Probability of flipping or adding
                                // to cluster.
                                // SHOULD NOT CHANGE FOR WOLFF

        double exponentials[2];
        double totalEnergy;

        // Neighboring lattice sites
        unsigned int nextX;
        unsigned int prevY;
        unsigned int prevX;
        unsigned int nextY;

        HashTable* cluster;
        gsl_rng* generator;
        // ------------------------------------------------------



        // ------------------------------------------------------
        // Methods!
        // Constructors, destructor
        Lattice(unsigned int x, unsigned int y, unsigned int RNSeed);
        Lattice();
        ~Lattice();

        // Lattice and cluster print methods
        void printLattice();
        void printCluster();

        // Setup periodic boundary conditions
        void getHalfNeighbors(unsigned int site);
        void getNeighbors(unsigned int site);

        // Calculation methods
        int calcHalfEnergy(unsigned int site);
        int calcEnergy(unsigned int site);
        double calcTotalEnergy();
        double calcMagnetization();
        double calcSpecificHeat(double average, double squared);
        double calcSusceptibility(double average, double squared);

        // Simulation methods - metropolis and wolff algorithms
        bool metropolis(unsigned int site); // Returns whether or not flipped
```

```
        void growCluster(unsigned int site, int spin);
        void flipCluster();
        void flipComplement();
        unsigned int wolff(unsigned int site);
        // -------------------------------------------------------
};
// ---------------------------------------------------------------



// ---------------------------------------------------------------
#endif  // _LATTICE_HH
// ---------------------------------------------------------------
```

Code Snippet C.5: Implementation of Ising model lattice

```
// ---------------------------------------------------------------
// Ising2D/Lattice.cpp
// Lattice of spins for Ising Model simulations
// Contains implementations of standard methods
// David Schaich -- daschaich@gmail.com
// Created 4 July 2005
// Last modified 20 March 2006
// ---------------------------------------------------------------



// ---------------------------------------------------------------
// Frontmatter and include directives
#include "Lattice.hh"                       // Method and variable declarations
// ---------------------------------------------------------------



// ---------------------------------------------------------------
// Constructors and destructor
Lattice::Lattice(unsigned int x, unsigned int y, unsigned int RNSeed) {

    cluster = new HashTable();
    generator = gsl_rng_alloc(gsl_rng_mt19937);     // Mersenne Twister
    gsl_rng_set (generator, RNSeed);

    temp = (float)RNSeed / 100;
    beta = 1 / temp;
    xDim = x;
    yDim = y;
    latticeSize = x * y;

    // Random initial state
    for (unsigned int i = 0; i < latticeSize; i++) {
        if (gsl_rng_uniform(generator) < 0.5)
            lattice.push_back(-1);
        else
            lattice.push_back(1);
    }

    // Calculate all exponentials once and for all
    exponentials[0] = gsl_sf_exp(-beta * 4);
    exponentials[1] = gsl_sf_exp(-beta * 8);

    // Initialize common data
    randomU = gsl_rng_uniform(generator);
    probability = 1 - gsl_sf_exp(-2 * beta);

    totalEnergy = calcTotalEnergy();
}
```

```
Lattice::Lattice()  {
    Lattice(32, 32, 227);
}

Lattice::~Lattice() {}
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Lattice and cluster print methods
void Lattice::printLattice() {
    for (unsigned int i = 0; i < latticeSize; i++) {
        if (i % xDim == 0)
            printf("\n");

        if (lattice[i] == -1)
            printf("o");
        else if (lattice[i] == 1)
            printf("x");
        else
            printf("ERROR");
    }
    printf("\n");
    fflush(stdout);
}

void Lattice::printCluster() {
    for (unsigned int i = 0; i < latticeSize; i++) {
        if (i % xDim == 0)
            printf("\n");

        if (cluster->find(i))
            printf("x");
        else
            printf(" ");
    }
    printf("\n");
    fflush(stdout);
}
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Set up helical boundary conditions
// Try to avoid the modulo % operation at all costs
void Lattice::getHalfNeighbors(unsigned int site) {
    if (site < latticeSize - xDim) {
        nextX = site + 1;
        nextY = site + xDim;
    }
    else if (site < latticeSize - 1) {
        nextX = site + 1;
        nextY = site + xDim - latticeSize;
    }
    else {                                      // site = latticeSize - 1
        nextX = 0;
        nextY = xDim;
    }
}

void Lattice::getNeighbors(unsigned int site) {
    getHalfNeighbors(site);

    if (site > xDim) {
        prevX = site - 1;
        prevY = site - xDim;
    }
    else if (site > 0) {
        prevX = site - 1;
```

```
        prevY = site + latticeSize - xDim;
    }
    else {                                      // site = 0
        prevX = latticeSize - 1;
        prevY = latticeSize - xDim;
    }
}
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Calculation methods
int Lattice::calcHalfEnergy(unsigned int site) {
    getHalfNeighbors(site);

    return -lattice[site] * (lattice[nextX] + lattice[nextY]);
}

int Lattice::calcEnergy(unsigned int site) {
    getNeighbors(site);

    return -lattice[site] *
        (lattice[nextX] + lattice[nextY] + lattice[prevX] + lattice[prevY]);
}

double Lattice::calcTotalEnergy() {
    totalEnergy = 0;
    for (unsigned int i = 0; i < latticeSize; i++)
        totalEnergy += calcHalfEnergy(i);
    return totalEnergy / latticeSize;
}

// Note: does not return absolute value
double Lattice::calcMagnetization() {
    int magnet = 0;
    for (unsigned int i = 0; i < latticeSize; i++)
        magnet += lattice[i];

    return (double) magnet / latticeSize;
}

// Uses energy per spin
double Lattice::calcSpecificHeat(double aveEnergy, double squaredEnergy) {
    double diff = squaredEnergy - (aveEnergy * aveEnergy);
    return beta * beta * diff * latticeSize;
}

// Uses magnetization per spin
double Lattice::calcSusceptibility(double aveMagnet, double squaredMagnet) {
    double diff = squaredMagnet - (aveMagnet * aveMagnet);
    return beta * diff * latticeSize;
}
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Metropolis method
bool Lattice::metropolis(unsigned int site) {
    // Only need to consider change in energy at site up for flipping
    // This is actually half the change...
    int difference = -calcEnergy(site);

    if (difference <= 0.0) {    // Lower energy (more negative), so flip
        lattice[site] *= -1;
        return true;
    }
    else {                      // Probabilistic acceptance
        randomU = gsl_rng_uniform(generator);
        probability = exponentials[difference / 2 - 1];
```

```
        if (randomU < probability) {
            lattice[site] *= -1;
            return true;
        }
        else
            return false;
    }
}
// -------------------------------------------------------------------



// -------------------------------------------------------------------
// Wolff methods for growing cluster and so on
void Lattice::growCluster(unsigned int site, int spin) {
    getNeighbors(site);
    // If this is not done, they will be overwritten...
    unsigned int curNextX = nextX;
    unsigned int curNextY = nextY;
    unsigned int curPrevX = prevX;
    unsigned int curPrevY = prevY;

    if (lattice[curPrevX] == spin && !cluster->find(curPrevX)) {
        randomU = gsl_rng_uniform(generator);
        if (randomU < probability) {
            cluster->insert(curPrevX);
            growCluster(curPrevX, spin);
        }
    }

    if (lattice[curNextY] == spin && !cluster->find(curNextY)) {
        randomU = gsl_rng_uniform(generator);
        if (randomU < probability) {
            cluster->insert(curNextY);
            growCluster(curNextY, spin);
        }
    }

    if (lattice[curNextX] == spin && !cluster->find(curNextX)) {
        randomU = gsl_rng_uniform(generator);
        if (randomU < probability) {
            cluster->insert(curNextX);
            growCluster(curNextX, spin);
        }
    }

    if (lattice[curPrevY] == spin && !cluster->find(curPrevY)) {
        randomU = gsl_rng_uniform(generator);
        if (randomU < probability) {
            cluster->insert(curPrevY);
            growCluster(curPrevY, spin);
        }
    }
}

void Lattice::flipCluster() {
    Node* temp = cluster->table[0]->head;
    for (unsigned int i = 0; i < 97; i++)
        if (cluster->table[i]->size != 0) {
            temp = cluster->table[i]->head;
            for (unsigned int j = 0; j < cluster->table[i]->size; j++) {
                lattice[temp->site] *= -1;
                temp = temp->next;
            }
        }
}

void Lattice::flipComplement() {
    for (unsigned int i = 0; i < latticeSize; i++)
        if (!cluster->find(i))       // Flip if not in cluster
            lattice[i] *= -1;
}
```

```
// Returns size of cluster
unsigned int Lattice::wolff(unsigned int site) {
    cluster->insert(site);
    growCluster(site, lattice[site]);

    if (cluster->size >= latticeSize / 2)
        flipComplement();
    else
        flipCluster();

    unsigned int toReturn = cluster->size;
    cluster->clear();
    return toReturn;
}
// -------------------------------------------------------------
```

Code Snippet C.6: Ising model simulation code

```
// -------------------------------------------------------------
// Ising2D/Metropolis.cpp
// Runs Monte Carlo simulation using Metropolis algorithm
// Writes energy, magnetization, specific heat and susceptibility
// David Schaich -- daschaich@gmail.com
// Created 13 July 2005
// Last modified 19 March 2006
// -------------------------------------------------------------



// -------------------------------------------------------------
// Include directives
#include <math.h>               // For floor and sqrt
#include <stdio.h>              // For printf
#include <stdlib.h>             // For string conversion atoi
#include <gsl/gsl_sf_log.h>     // For natural log
#include "Lattice.hh"           // Lattice on which simulation is run
// -------------------------------------------------------------



// -------------------------------------------------------------
// Main method runs simulation given command line parameters
int main(unsigned int argc, char** const argv) {
    if (argc != 6) {
        fprintf(stderr, "Usage: %s xDim yDim init sampleSize temp\n", argv[0]);
        fflush(stderr);
        exit(1);
    }

    unsigned int xDim = atoi(argv[1]);          // Lattice x-dimension
    unsigned int yDim = atoi(argv[2]);          // Lattice y-dimension
    unsigned int init = atoi(argv[3]);          // Equilibration sweeps
    unsigned int sampleSize = atoi(argv[4]);    // Iterations for statistics
    unsigned int RNSeed = atoi(argv[5]);        // 100x temperature (kT)

    float temp = (float)RNSeed / 100;

    unsigned int randomSite;
    unsigned int latticeSize = xDim * yDim;

    double aveEnergy = 0.0;
    double aveMagnet = 0.0;
    double aveMagnetAbs = 0.0;
    double squaredEnergy = 0.0;
    double squaredMagnet = 0.0;
    double specificHeat = 0.0;
```

```
double susceptibility = 0.0;

double energyData[sampleSize];
double magnetData[sampleSize];
double autocorrelation[sampleSize];     // Holds function
double scaleFactor = 0.0;
double autocorTime = 0.0;               // Holds time
double energyStDev = 0.0;
double magnetStDev = 0.0;

Lattice* theLattice = new Lattice(xDim, yDim, RNSeed);

// Initialize/equilibrate lattice
for (unsigned int i = 0; i < init * latticeSize; i++) {
    randomSite = (int) floor(latticeSize *
            gsl_rng_uniform(theLattice->generator));
    theLattice->metropolis(randomSite);
}

// Take data every 5 sweeps (somewhat arbitrary declaration
// based on checking out autocorrelation times)
unsigned int counter = 0;
for (unsigned int i = 0; i < sampleSize * latticeSize * 5; i++) {
    randomSite = (int) floor(latticeSize *
            gsl_rng_uniform(theLattice->generator));
    theLattice->metropolis(randomSite);

    // Every 5 sweeps, record data
    if (i % (latticeSize * 5) == 0) {
        energyData[counter] = theLattice->calcTotalEnergy();
        aveEnergy += energyData[counter];

        magnetData[counter] = theLattice->calcMagnetization();
        aveMagnet += magnetData[counter];
        magnetData[counter] = fabs(magnetData[counter]);
        aveMagnetAbs += magnetData[counter];

        squaredEnergy += energyData[counter] * energyData[counter];
        squaredMagnet += magnetData[counter] * magnetData[counter];

        counter++;
    }
}

// Take averages
aveEnergy /= sampleSize;
aveMagnet /= sampleSize;
aveMagnetAbs /= sampleSize;
squaredEnergy /= sampleSize;
squaredMagnet /= sampleSize;

// Add bootstrapping here...
specificHeat = theLattice->calcSpecificHeat(aveEnergy, squaredEnergy);
susceptibility =
        theLattice->calcSusceptibility(aveMagnetAbs, squaredMagnet);

delete theLattice;

// Now its time for some autocorrelation and standard deviation madness
// Use magnetization for autocorrelation time calculation
// Should be roughly the same for all variables

// Generate Chi[0] for scaling purposes
for (unsigned int i = 0; i < sampleSize; i++)
    scaleFactor += (magnetData[i] * magnetData[i]);
scaleFactor /= sampleSize;
scaleFactor -= aveMagnetAbs * aveMagnetAbs;

autocorrelation[0] = 1.0;

// Calculate autocorrelation function, chi[t], t > 0
for (unsigned int t = 1; t < sampleSize; t++) {
```

```
        autocorrelation[t] = 0.0;
        for (unsigned int i = 0; i < sampleSize - t; i++)
            autocorrelation[t] += (magnetData[i] * magnetData[i + t]);
        autocorrelation[t] /= (sampleSize - t);
        autocorrelation[t] -= aveMagnetAbs * aveMagnetAbs;
        autocorrelation[t] /= scaleFactor;      // Scale by Chi[0]
    }

    // Generate autocorrelation times from autocorrelation function
    double tempd;
    unsigned int i = 1;

    while (autocorrelation[i] > 0
            && autocorrelation[i] < autocorrelation[i - 1]
            && i < sampleSize) {

        tempd = -gsl_sf_log (autocorrelation[i]);
        tempd = i / tempd;
        autocorTime += tempd;

        i++;
    }

    // Use standard formula to generate standard deviations
    // from autocorrelation time, average, sampleSize, etc
    if (i == 1) {            // Assume autocorrelation time zero
        autocorTime = 0.0;
        energyStDev = 0.0;
        magnetStDev = 0.0;
    }
    else {
        autocorTime /= (i - 1);
        energyStDev = 2 * autocorTime / sampleSize;
        energyStDev *= squaredEnergy - aveEnergy * aveEnergy;
        energyStDev = sqrt(energyStDev);

        magnetStDev = 2 * autocorTime / sampleSize;
        magnetStDev *= squaredMagnet - aveMagnetAbs * aveMagnetAbs;
        magnetStDev = sqrt(magnetStDev);
    }

    printf("%f\t%lf\t", temp, autocorTime);
    printf("%lf\t%lf\t", aveEnergy, energyStDev);
    printf("%lf\t%lf\t", aveMagnetAbs, magnetStDev);
    printf("%lf\t%lf\t", specificHeat, susceptibility);
    printf("%lf\t%lf\n", aveMagnet, scaleFactor);

    return 0;
}
// ----------------------------------------------------------------------
```

C.3 ϕ^4 Theory

In this section we include much of the code used in our two-dimensional ϕ^4 simulations – a lattice class and header file, along with a simulation file including the main method and a couple of helper methods that calculate bimodality and autocorrelation times. We do not include the hash table used, which is presented above in Section B.3. As in the Ising model code above, this simulation uses the helical boundary conditions described above in Section 3.3.

Although this code is based off of the Ising model code presented above in Section C.2, it has enough differences and was used so extensively over the course of the thesis

that we present it here. The code for four-dimensional simulations is largely identical to that presented below, with only the obviously necessary changes made.

The program assumes a two-dimensional rectangular lattice and takes six input parameters from the command line: the values of μ_0^2 and λ, the two dimensions of the lattice, the number of iterations for equilibration and the number of iterations for statistics. This input is all integer so that the program to be called repeatedly from loops in shell scripts; μ_0^2 and λ are scaled by 10^4 and 10^2 (respectively) in the main method in order to achieve the desired precision.

Code Snippet C.7: Header file for ϕ^4 theory lattice

```
// ---------------------------------------------------------------
// Phi4/Lattice.hh
// Lattice of spins for phi^4 simulations using mu action
// Header file contains data and method declarations
// David Schaich -- daschaich@gmail.com
// Created 12 October 2005
// Last modified 9 February 2006
// ---------------------------------------------------------------



// ---------------------------------------------------------------
// Frontmatter and include directives
// Avoid multiple inclusion
#ifndef _LATTICE_HH
#define _LATTICE_HH

#include "HashTable.hh"             // Hash table for searching cluster
#include <vector>                   // Lattice is vector of vectors
#include <gsl/gsl_rng.h>            // Random number generators
#include <gsl/gsl_sf_exp.h>         // Exponential functions
// ---------------------------------------------------------------



// ---------------------------------------------------------------
// A simple struct to hold a site's neighbors
struct siteNeighbors {
    unsigned int prevX;
    unsigned int nextX;
    unsigned int prevY;
    unsigned int nextY;
};
// ---------------------------------------------------------------



// ---------------------------------------------------------------
// Class definition
class Lattice {
    public:
        // -------------------------------------------------------
        // Data!
        // Member data
        std::vector<double> lattice;    // Continuous values
        unsigned int xDim;              // x dimension of lattice
        unsigned int yDim;              // y dimension of lattice
        unsigned int latticeSize;       // Number of sites in lattice

        double muSquared;       // Mass of particles
        double lambda;          // Coupling strength

        // Neighboring lattice sites
        std::vector<siteNeighbors*> neighbors;
```

```
        HashTable* cluster;
        gsl_rng* generator;
        // ------------------------------------------------------------



        // ------------------------------------------------------------
        // Methods!
        // Constructors, destructor
        Lattice(double m, double l, unsigned int x, unsigned int y);
        Lattice();
        ~Lattice();

        // Lattice and cluster print methods
        // Replace with png drawing?
        void printLattice();
        void printSigns();
        void printCluster();

        // Set up periodic boundary conditions
        void getNeighbors(unsigned int site, siteNeighbors* toInit);

        // Calculation methods
        double calcTotalEnergy();
        double calcAveragePhi();

        // Simulation methods - metropolis and wolff algorithms
        void metropolis(unsigned int site);

        bool clusterCheck(unsigned int site, unsigned int toAdd);

        // Inelegant but faster
        void growClusterPos(unsigned int site);
        void growClusterNeg(unsigned int site);
        void flipCluster();
        unsigned int wolff(unsigned int site);      // Returns cluster size
        // ------------------------------------------------------------
};
// ------------------------------------------------------------------



// ------------------------------------------------------------------
#endif  // _LATTICE_HH
// ------------------------------------------------------------------
```

Code Snippet C.8: Implementation of ϕ^4 theory lattice

```
// ------------------------------------------------------------------
// Phi4/Lattice.cpp
// Lattice of spins for phi^4 simulations
// Contains implementations of standard methods
// David Schaich -- daschaich@gmail.com
// Created 12 October 2005
// Last modified 15 February 2006
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Frontmatter and include directives
#include "Lattice.hh"               // Method and variable declarations
// ------------------------------------------------------------------



// ------------------------------------------------------------------
```

```
// Constructors and destructor
Lattice::Lattice(double m, double l, unsigned int x, unsigned int y) {
    generator = gsl_rng_alloc(gsl_rng_mt19937);     // Mersenne Twister
    gsl_rng_set (generator, (unsigned int)(100 * m * l));

    muSquared = 2 + (m / 2);        // Do this to simplify calcs
    lambda = l / 4;                 // Do this to simplify calcs

    xDim = x;
    yDim = y;
    latticeSize = xDim * yDim;
    cluster = new HashTable(latticeSize / 4);

    // Random initial state in range [-1.5, 1.5)
    for (unsigned int i = 0; i < latticeSize; i++)
        lattice.push_back(3 * gsl_rng_uniform(generator) - 1.5);

    // Set up neighbors... calculate once for all sites
    for (unsigned int i = 0; i < latticeSize; i++) {
        siteNeighbors* temp = new siteNeighbors;
        getNeighbors(i, temp);
        neighbors.push_back(temp);
    }
}

Lattice::Lattice()  {
    Lattice(-1.25, 1, 32, 32);
}

Lattice::~Lattice() {}
// ---------------------------------------------------------------------



// ---------------------------------------------------------------------
// Lattice and cluster print methods
void Lattice::printLattice() {
    for (unsigned int i = 0; i < latticeSize; i++) {
        if (i % xDim == 0)
            printf("\n");
        printf("%lf\t", lattice[i]);

    }
    printf("\n");
    fflush(stdout);
}

void Lattice::printSigns() {
    for (unsigned int i = 0; i < latticeSize; i++) {
        if (i % xDim == 0)
            printf("\n");
        if (lattice[i] >= 0)
            printf(" ");
        else
            printf("x ");
    }
    printf("\n");
    fflush(stdout);
}

void Lattice::printCluster() {
    for (unsigned int i = 0; i < latticeSize; i++) {
        if (i % xDim == 0)
            printf("\n");

        if (cluster->find(i))
            printf("x ");
        else
            printf(" ");
    }
    printf("\n");
    fflush(stdout);
```

```
}
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Set up helical boundary conditions - only do once
// Try to avoid the modulo % operation
void Lattice::getNeighbors(unsigned int site, siteNeighbors* toInit) {
    if (site < latticeSize - xDim) {
        toInit->nextX = site + 1;
        toInit->nextY = site + xDim;
    }
    else if (site < latticeSize - 1) {
        toInit->nextX = site + 1;
        toInit->nextY = site + xDim - latticeSize;
    }
    else {                                      // site = latticeSize - 1
        toInit->nextX = 0;
        toInit->nextY = xDim - 1;
    }

    if (site >= xDim) {
        toInit->prevX = site - 1;
        toInit->prevY = site - xDim;
    }
    else if (site > 0) {
        toInit->prevX = site - 1;
        toInit->prevY = site + latticeSize - xDim;
    }
    else {                                      // site = 0
        toInit->prevX = latticeSize - 1;
        toInit->prevY = latticeSize - xDim;
    }
}
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Calculation methods
// Calculate total energy by looping through lattice
double Lattice::calcTotalEnergy() {
    double totalEnergy = 0;
    double currentPhi;
    for (unsigned int i = 0; i < latticeSize; i++) {
        currentPhi = lattice[i];
        totalEnergy -= currentPhi * (lattice[neighbors[i]->nextX]
                                  +  lattice[neighbors[i]->nextY]);

        currentPhi *= currentPhi;
        // Recall muSquared redefined
        totalEnergy += muSquared * currentPhi;

        currentPhi *= currentPhi;
        // Recall lambda redefined
        totalEnergy += lambda * currentPhi;
    }
    return totalEnergy / latticeSize;
}

// Note: does not return absolute value
double Lattice::calcAveragePhi() {
    double currentPhi = 0;
    for (unsigned int i = 0; i < latticeSize; i++)
        currentPhi += lattice[i];

    return currentPhi / latticeSize;
}
// ------------------------------------------------------------------
```

```
// ------------------------------------------------------------------
// Metropolis method
void Lattice::metropolis(unsigned int site) {
    double currentPhi = lattice[site];

    // Generate new value
    double newValue = currentPhi + (3 * gsl_rng_uniform(generator)
                                    - 1.5);
    double temp = newValue;

    // Calculate energy difference
    double difference = (currentPhi - newValue)
                      * (lattice[neighbors[site]->nextX]
                      +  lattice[neighbors[site]->nextY]
                      +  lattice[neighbors[site]->prevX]
                      +  lattice[neighbors[site]->prevY]);

    newValue *= newValue;
    currentPhi  *= currentPhi;
    // Recall muSquared redefined
    difference += muSquared * (newValue - currentPhi);

    newValue *= newValue;
    currentPhi *= currentPhi;
    // Recall lambda redefined
    difference += lambda * (newValue - currentPhi);

    // Flip if difference <= 0, otherwise probabilistic acceptance
    if (difference <= 0)
        lattice[site] = temp;
    else if (gsl_rng_uniform(generator) < gsl_sf_exp(-difference))
        lattice[site] = temp;
}
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Wolff methods for growing cluster and so on
// A convenience method that keeps me from having to write these few
// lines over and over again - adds to cluster probabilistically
bool Lattice::clusterCheck(unsigned int site, unsigned int toAdd) {
    if (cluster->find(toAdd))
        return false;       // Already in cluster

    // Have already checked that they have the same sign
    double probability = 1 - gsl_sf_exp(-2 * lattice[site] * lattice[toAdd]);

    if (gsl_rng_uniform(generator) < probability) {
        cluster->insert(toAdd);
        return true;
    }

    return false;
}

// Grows cluster from specified site - recursive
void Lattice::growClusterPos(unsigned int site) {
    unsigned int toCheck = neighbors[site]->prevX;
    if (lattice[toCheck] > 0 && clusterCheck(site, toCheck))
        growClusterPos(toCheck);

    toCheck = neighbors[site]->nextX;
    if (lattice[toCheck] > 0 && clusterCheck(site, toCheck))
        growClusterPos(toCheck);

    toCheck = neighbors[site]->prevY;
    if (lattice[toCheck] > 0 && clusterCheck(site, toCheck))
        growClusterPos(toCheck);

    toCheck = neighbors[site]->nextY;
```

```
    if (lattice[toCheck] > 0 && clusterCheck(site, toCheck))
        growClusterPos(toCheck);
}

void Lattice::growClusterNeg(unsigned int site) {
    unsigned int toCheck = neighbors[site]->prevX;
    if (lattice[toCheck] <= 0 && clusterCheck(site, toCheck))
        growClusterNeg(toCheck);

    toCheck = neighbors[site]->nextX;
    if (lattice[toCheck] <= 0 && clusterCheck(site, toCheck))
        growClusterNeg(toCheck);

    toCheck = neighbors[site]->prevY;
    if (lattice[toCheck] <= 0 && clusterCheck(site, toCheck))
        growClusterNeg(toCheck);

    toCheck = neighbors[site]->nextY;
    if (lattice[toCheck] <= 0 && clusterCheck(site, toCheck))
        growClusterNeg(toCheck);
}

// Since this method trawls through the whole cluster (which has reached
// the end of its usefulness), let's be efficient and clear it here
void Lattice::flipCluster() {
    node* temp1;
    node* temp2;

    for (unsigned int i = 0; i < cluster->tableNumber; i++) {
        temp1 = cluster->table[i];
        temp2 = cluster->table[i];
        while (temp1 != NULL) {
            lattice[temp1->value] *= -1;
            temp2 = temp2->next;
            delete temp1;
            temp1 = temp2;
        }
        cluster->table[i] = NULL;
    }
    cluster->size = 0;
}

unsigned int Lattice::wolff(unsigned int site) {
    cluster->insert(site);

    // Inelegant, but faster than having just one method
    if (lattice[site] > 0)
        growClusterPos(site);
    else
        growClusterNeg(site);

    unsigned int toReturn = cluster->size;
    flipCluster();

    return toReturn;
}
// -----------------------------------------------------------------
```

Code Snippet C.9: ϕ^4 theory simulation code

```
// -----------------------------------------------------------------
// Phi4/Simulation.cpp
// Runs Monte Carlo simulation for fixed beta using mix of Metropolis
// and Wolff algorithms
// Prints energy, average phi, specific heat, susceptibility,
// autocorrelation times, errors, etc
// David Schaich -- daschaich@gmail.com
```

```
// Created 23 October 2005
// Last modified 1 March 2006
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Include directives
#include <math.h>           // For floor and sqrt
#include <stdio.h>          // For printf and fprintf
#include <gsl/gsl_sf_log.h> // For natural log
#include <gsl/gsl_math.h>   // For power
#include "Lattice.hh"       // Method and data declarations
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Declare and zero data!
unsigned int randomSite = 0;

double aveEnergy        = 0;
double avePhi           = 0;
double avePhiAbs        = 0;
double squaredEnergy    = 0;
double squaredPhi       = 0;
double quartPhi         = 0;
double cumulant         = 0;
double specificHeat     = 0;
double susceptibility   = 0;

// Bimodality binning stuff
double maxPhi                   = 0;
static const unsigned int bins  = 21;
unsigned int counts[bins];
double midCounts                = 0;            // Make these double now
double maxCounts                = 0;            // to avoid casts later
double bimodality               = 0;

double scaleFactor  = 0;
double autocorTime  = 0;
double energyStDev  = 0;
double phiStDev     = 0;
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Calculate autocorrelation time and necessary points of
// autocorrelation function
unsigned int calcAutocor(unsigned int sampleSize, double* phiDataAbs,
        double* autocorrelation) {

    // Generate Chi[0] for scaling purposes
    for (unsigned int i = 0; i < sampleSize; i++)
        scaleFactor += (phiDataAbs[i] * phiDataAbs[i]);
    scaleFactor /= sampleSize;
    scaleFactor -= avePhiAbs * avePhiAbs;

    autocorrelation[0] = 1;

    // Only calculate points of autocorrelation function that
    // roughly conform to exponential approximation
    unsigned int t = 1;

    while (t < sampleSize) {
        for (unsigned int i = 0; i < sampleSize - t; i++)
            autocorrelation[t] += (phiDataAbs[i] * phiDataAbs[i + t]);

        autocorrelation[t] /= (sampleSize - t);
        autocorrelation[t] -= avePhiAbs * avePhiAbs;
        if (autocorrelation[t] < 0)     // Not exponential!
```

```
            break;

        autocorrelation[t] /= scaleFactor;      // Scale by Chi[0]
        if (autocorrelation[t] >= autocorrelation[t - 1])   // Not exponential!
            break;

        double temp = -gsl_sf_log (autocorrelation[t]);
        temp = t / temp;
        autocorTime += temp;

        t++;
    }

    return t;
}
// ---------------------------------------------------------------



// ---------------------------------------------------------------
// Bin values of phi and calculate bimodality
// Values of phi range from -maxPhi to +maxPhi
// Bin i of n will contain values greater than maxPhi(2*i/n - 1)
// and less than maxPhi*(2*(i+1)/n - 1)  (where i = 0...n - 1)
void calcBimodality(unsigned int sampleSize, double* phiData) {
    double lowerBound;
    double upperBound;
    for (unsigned int i = 0; i < bins; i++)
        counts[i] = 0;

    for (unsigned int i = 0; i < sampleSize; i++)
        for (unsigned int j = 0; j < bins; j++) {
            lowerBound = maxPhi * (2 * (double)j / (double)bins - 1);
            upperBound = maxPhi * (2 * ((double)j + 1) / (double)bins - 1);
            if (phiData[i] >= lowerBound && phiData[i] <= upperBound) {
                counts[j]++;
                break;
            }
        }

    midCounts = counts[(bins - 1) / 2];
    for (unsigned int i = 0; i < bins; i++)
        if (counts[i] > maxCounts)
            maxCounts = counts[i];

    bimodality = 1 - (midCounts / maxCounts);
}
// ---------------------------------------------------------------



// ---------------------------------------------------------------
// Main method runs simulation using command line parameters
int main(unsigned int argc, char** const argv) {
    if (argc != 7) {
        fprintf(stderr, "Usage: %s muSquared lambda xDim yDim ", argv[0]);
        fprintf(stderr, "init sampleSize\n");
        fflush(stderr);
        exit(1);
    }

    // Data that depends on command-line parameters

    double muSquared = atof(argv[1]) / 10000;   // Particle mass (squared...)
    double lambda = atof(argv[2]) / 100;        // Coupling strength
    unsigned int xDim = atoi(argv[3]);          // Lattice x-dimension
    unsigned int yDim = atoi(argv[4]);          // Lattice y-dimension
    unsigned int init = atoi(argv[5]);          // Iterations for equilibration
    unsigned int sampleSize = atoi(argv[6]);    // Iterations for statistics

    unsigned int latticeSize = xDim * yDim;
```

```
double energyData[sampleSize];
double phiData[sampleSize];
double phiDataAbs[sampleSize];
double autocorrelation[sampleSize];     // Autocorrelation function
                                        // Much bigger than necessary

Lattice* theLattice = new Lattice(muSquared, lambda, xDim, yDim);

// Initialize/equilibrate lattice
// Flip cluster and take data after every _gap_ Metropolis sweeps
unsigned int gap = 5;
for (unsigned int i = 0; i < init; i++) {
    for (unsigned int j = 0; j < latticeSize * gap; j++) {
        randomSite = (unsigned int) floor(latticeSize *
                gsl_rng_uniform(theLattice->generator));
        theLattice->metropolis(randomSite);
    }

    randomSite = (unsigned int) floor(latticeSize *
            gsl_rng_uniform(theLattice->generator));
    theLattice->wolff(randomSite);
}

for (unsigned int i = 0; i < sampleSize; i++) {
    for (unsigned int j = 0; j < latticeSize * gap; j++) {
        randomSite = (int) floor(latticeSize *
                gsl_rng_uniform(theLattice->generator));
        theLattice->metropolis(randomSite);
    }

    randomSite = (unsigned int) floor(latticeSize *
            gsl_rng_uniform(theLattice->generator));
    theLattice->wolff(randomSite);

    energyData[i] = theLattice->calcTotalEnergy();
    aveEnergy += energyData[i];

    phiData[i] = theLattice->calcAveragePhi();
    phiDataAbs[i] = fabs(phiData[i]);
    avePhi += phiData[i];
    avePhiAbs += phiDataAbs[i];

    // Maximum (absolute) value of phi
    if (phiDataAbs[i] > maxPhi)
        maxPhi = phiDataAbs[i];

    squaredEnergy += energyData[i] * energyData[i];
    squaredPhi += phiData[i] * phiData[i];
    quartPhi += gsl_pow_4(phiData[i]);
}

delete theLattice;

// Take averages
aveEnergy       /= sampleSize;
avePhi          /= sampleSize;
avePhiAbs       /= sampleSize;
squaredEnergy   /= sampleSize;
squaredPhi      /= sampleSize;
quartPhi        /= sampleSize;

// Add bootstrapping here...
specificHeat = squaredEnergy - (aveEnergy * aveEnergy);
specificHeat *= latticeSize;
susceptibility = squaredPhi - (avePhiAbs * avePhiAbs);
susceptibility *= latticeSize;

// Now it's time for some autocorrelation and standard deviation madness
// Use average phi for autocorrelation time calculation
// Should be roughly the same for all variables
unsigned int t = calcAutocor(sampleSize, phiDataAbs, autocorrelation);
```

```
    // Use standard formula to generate standard deviations
    // from autocorrelation time, average, sampleSize, etc
    if (t == 1) {            // Autocorrelation time zero
        autocorTime = 0;
        energyStDev = 0;
        phiStDev    = 0;
    }
    else {
        autocorTime /= (t - 1);
        energyStDev = 2 * autocorTime / sampleSize;
        energyStDev *= squaredEnergy - (aveEnergy * aveEnergy);
        energyStDev = sqrt(energyStDev);

        phiStDev = 2 * autocorTime / sampleSize;
        phiStDev *= squaredPhi - (avePhiAbs * avePhiAbs);
        phiStDev = sqrt(phiStDev);
    }

    // Calculate binder cumulant
    cumulant = 1 - quartPhi / (3 * squaredPhi * squaredPhi);

    // Calculate bimodality
    calcBimodality(sampleSize, phiData);

    printf("%lf,%lf,%lf,", muSquared, lambda, autocorTime);
    printf("%lf,%lf,", aveEnergy, energyStDev);
    printf("%lf,%lf,", avePhiAbs, phiStDev);
    printf("%lf,%lf,", specificHeat, susceptibility);
    printf("%lf,%lf,", cumulant, bimodality);
    printf("%lf,%lf\n", avePhi, scaleFactor);

    return 0;
}
// ------------------------------------------------------------------
```

C.4 Solitons

In this section we present much of the code used in our simulations of ϕ^4 solitons on the lattice – a lattice class and header file, along with a simulation file including the main method and a couple of helper methods used to calculate the bimodality and autocorrelation time. We do not include the hash table used, which is presented above in Section B.3.

Although this code is based off of the basic ϕ^4 theory code presented above in Section C.3, it has enough notable features to merit inclusion here. In particular, we no longer use the helical boundary conditions of the above simulations. Instead, for reasons discussed in Section 7.6, we use one lattice with standard periodic boundary conditions and another with antiperiodic boundary conditions.

In addition, this code illustrates one of the main benefits of object-oriented programming. Since our two lattices are each objects of a lattice class, they automatically have their own individual data fields, listed in the lattice header file, Code Snippet C.10. In order to exploit this feature, we moved much of the data formerly loose in the main method of Code Snippet C.9 directly into the lattice objects themselves themselves.

The program assumes a two-dimensional rectangular lattice and takes six input parameters from the command line: the values of μ_0^2 and λ, the two dimensions of

the lattice, the number of iterations for equilibration and the number of iterations for statistics. This input is all integer so that the program to be called repeatedly from loops in shell scripts; μ_0^2 and λ are scaled by 10 and 10^2 (respectively) in the main method in order to achieve the desired precision.

Code Snippet C.10: Header file for ϕ^4 soliton lattice

```
// ------------------------------------------------------------------
// Soliton/Lattice.hh
// Lattice of spins for phi^4 simulations using mu action
// Header file contains data and method declarations
// David Schaich -- daschaich@gmail.com
// Created 23 January 2006
// Last modified 9 February 2006
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Frontmatter and include directives
// Avoid multiple inclusion
#ifndef _LATTICE_HH
#define _LATTICE_HH

#include "HashTable.hh"             // Hash table for searching cluster
#include <vector>                   // Lattice is vector of vectors
#include <gsl/gsl_rng.h>            // Random number generators
#include <gsl/gsl_sf_exp.h>         // Exponential functions
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// A simple struct to hold a site's neighbors
struct siteNeighbors {
    unsigned int prevX;
    unsigned int nextX;
    unsigned int prevY;
    unsigned int nextY;

    // Boundary conditions
    int isLeft;             // +/- 1 For periodic/antiperiodic
    int isRight;            // +/- 1 For periodic/antiperiodic
};
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Class definition
class Lattice {
    public:
        // ----------------------------------------------------------
        // Data!
        // Member data
        std::vector<double> lattice;    // Continuous values
        unsigned int xDim;              // x dimension of lattice
        unsigned int yDim;              // y dimension of lattice
        unsigned int latticeSize;       // Number of sites in lattice

        double muSquared;       // Mass of particles
        double lambda;          // Coupling strength

        bool antiperiodic;      // Antiperiodic boundary conditions?

        // Neighboring lattice sites
        std::vector<siteNeighbors*> neighbors;

        HashTable* cluster;
```

```
        gsl_rng* generator;

        // Data storage to simplify boundary conditions
        double aveEnergy;                   // <E>
        double energyStDev;
        double avePhi;                      // <phi>
        double avePhiAbs;                   // <|phi|>
        double phiStDev;

        double squaredEnergy;               // <E^2>
        double squaredPhi;                  // <phi^2>
        double quartPhi;                    // <phi^4>

        double cumulant;
        double specificHeat;
        double susceptibility;

        double maxPhi;
        double bimodality;

        double scaleFactor;
        double autocorTime;

        // Need to use vectors to let length of run be command-line input
        std::vector<double> energyData;
        std::vector<double> phiData;
        std::vector<double> phiDataAbs;
        std::vector<double> autocor;
        // ------------------------------------------------------



        // ------------------------------------------------------
        // Methods!
        // Constructors, destructor
        Lattice(double m, double l, unsigned int x, unsigned int y,
                unsigned int t, bool p);
        Lattice();
        ~Lattice();

        // Lattice and cluster print methods, mainly for debugging
        // Could replace with png drawing to generate pretty pictures
        void printLattice();
        void printSigns();
        void printCluster();
        void printData();           // Convenience method for simulation

        // Set up periodic boundary conditions
        void getNeighbors(unsigned int site, siteNeighbors* toInit);

        // Calculation methods
        double calcTotalEnergy();
        double calcAveragePhi();
        void calcSpecificHeat();
        void calcSusceptibility();

        // Simulation methods - metropolis and wolff algorithms
        void metropolis(unsigned int site);

        bool clusterCheck(double site, double value, unsigned int toAdd);

        // Inelegant but faster
        void growClusterPos(unsigned int site);
        void growClusterNeg(unsigned int site);
        void flipCluster();
        unsigned int wolff(unsigned int site);      // Returns cluster size
        // ------------------------------------------------------
};
// ------------------------------------------------------------------
```

```
// ---------------------------------------------------------------
#endif  // _LATTICE_HH
// ---------------------------------------------------------------
```

Code Snippet C.11: Implementation of ϕ^4 soliton lattice

```
// ---------------------------------------------------------------
// Soliton/Lattice.cpp
// Lattice of spins for phi^4 simulations
// Contains implementations of standard methods
// David Schaich -- daschaich@gmail.com
// Created 23 January 2006
// Last modified 1 March 2006
// ---------------------------------------------------------------



// ---------------------------------------------------------------
// Frontmatter and include directives
#include "Lattice.hh"               // Method and variable declarations
// ---------------------------------------------------------------



// ---------------------------------------------------------------
// Constructors and destructor
Lattice::Lattice(double m, double l, unsigned int x, unsigned int y,
        unsigned int sampleSize, bool anti) {

    generator = gsl_rng_alloc(gsl_rng_mt19937);     // Mersenne Twister
    gsl_rng_set (generator, (unsigned int)(100 * m * l));

    muSquared = 2 + (m / 2);        // Do this to simplify calcs
    lambda = l / 4;                 // Do this to simplify calcs

    antiperiodic = anti;
    xDim = x;
    yDim = y;
    latticeSize = xDim * yDim;
    cluster = new HashTable(latticeSize / 4);

    // Random initial state in range [-1.5, 1.5)
    for (unsigned int i = 0; i < latticeSize; i++)
        lattice.push_back(3 * gsl_rng_uniform(generator) - 1.5);

    // Set up neighbors... calculate once for all sites
    for (unsigned int i = 0; i < latticeSize; i++) {
        siteNeighbors* temp = new siteNeighbors;
        getNeighbors(i, temp);
        neighbors.push_back(temp);
    }

    // Initialize data to zero
    aveEnergy       = 0;
    energyStDev     = 0;
    avePhi          = 0;
    avePhiAbs       = 0;
    phiStDev        = 0;

    squaredEnergy   = 0;
    squaredPhi      = 0;
    quartPhi        = 0;

    cumulant        = 0;
    specificHeat    = 0;
    susceptibility  = 0;
```

```
    maxPhi          = 0;
    bimodality      = 0;

    scaleFactor     = 0;
    autocorTime     = 0;

    for (unsigned int i = 0; i < sampleSize; i++) {
        energyData.push_back(0);
        phiData.push_back(0);
        phiDataAbs.push_back(0);
        autocor.push_back(0);
    }
}

Lattice::Lattice()  {
    Lattice(-1.25, 1, 32, 32, 16384, false);
}

Lattice::~Lattice() {}
// ---------------------------------------------------------------------



// ---------------------------------------------------------------------
// Lattice and cluster print methods, mainly for debugging
// Could replace with png drawing to generate pretty pictures
void Lattice::printLattice() {
    for (unsigned int i = 0; i < latticeSize; i++) {
        if (i % xDim == 0)
            printf("\n");
        printf("%lf\t", lattice[i]);

    }
    printf("\n");
    fflush(stdout);
}

void Lattice::printSigns() {
    for (unsigned int i = 0; i < latticeSize; i++) {
        if (i % xDim == 0)
            printf("\n");
        if (lattice[i] >= 0)
            printf(" ");
        else
            printf("x ");
    }
    printf("\n");
    fflush(stdout);
}

void Lattice::printCluster() {
    for (unsigned int i = 0; i < latticeSize; i++) {
        if (i % xDim == 0)
            printf("\n");

        if (cluster->find(i))
            printf("x ");
        else
            printf(" ");
    }
    printf("\n");
    fflush(stdout);
}

// Convenience method
void Lattice::printData() {
    printf("%lf,",      autocorTime);
    printf("%lf,%lf,", aveEnergy, energyStDev);
    printf("%lf,%lf,", avePhiAbs, phiStDev);
    printf("%lf,%lf,", specificHeat, susceptibility);
    printf("%lf,%lf,", cumulant, bimodality);
    printf("%lf,%lf,", avePhi, scaleFactor);
```

```
}
// -------------------------------------------------------------



// -------------------------------------------------------------
// Set up periodic boundary conditions - only do once
void Lattice::getNeighbors(unsigned int site, siteNeighbors* toInit) {
    if ((site + 1) % xDim == 0) {
        toInit->nextX = site + 1 - xDim;
        if (antiperiodic)
            toInit->isRight = -1;           // Antiperiodic
        else
            toInit->isRight = 1;
    }
    else {
        toInit->nextX = site + 1;
        toInit->isRight = 1;
    }

    if (site >= latticeSize - xDim)
        toInit->nextY = site + xDim - latticeSize;
    else
        toInit->nextY = site + xDim;

    if (site % xDim == 0) {
        toInit->prevX = site + xDim - 1;
        if (antiperiodic)
            toInit->isLeft = -1;            // Antiperiodic
        else
            toInit->isLeft = 1;
    }
    else {
        toInit->prevX = site - 1;
        toInit->isLeft = 1;
    }

    if (site < xDim)
        toInit->prevY = site + latticeSize - xDim;
    else
        toInit->prevY = site - xDim;
}
// -------------------------------------------------------------



// -------------------------------------------------------------
// Calculation methods
// Calculate total energy by looping through lattice
double Lattice::calcTotalEnergy() {
    double totalEnergy = 0;
    double currentPhi;
    for (unsigned int i = 0; i < latticeSize; i++) {
        currentPhi = lattice[i];
        totalEnergy -= currentPhi
                    * (lattice[neighbors[i]->nextX] * neighbors[i]->isRight
                    +  lattice[neighbors[i]->nextY]);

        currentPhi *= currentPhi;
        // Recall muSquared redefined
        totalEnergy += muSquared * currentPhi;

        currentPhi *= currentPhi;
        // Recall lambda redefined
        totalEnergy += lambda * currentPhi;
    }
    return totalEnergy / latticeSize;
}

// Note: does not return absolute value
double Lattice::calcAveragePhi() {
    double currentPhi = 0;
```

```
    for (unsigned int i = 0; i < latticeSize; i++)
        currentPhi += lattice[i];

    return currentPhi / latticeSize;
}

// Simple specific heat and susceptibility calculations
void Lattice::calcSpecificHeat() {
    specificHeat = squaredEnergy - (aveEnergy * aveEnergy);
    specificHeat *= latticeSize;
}

void Lattice::calcSusceptibility() {
    susceptibility = squaredPhi - (avePhiAbs * avePhiAbs);
    susceptibility *= latticeSize;
}
// ---------------------------------------------------------------



// ---------------------------------------------------------------
// Metropolis method
void Lattice::metropolis(unsigned int site) {
    double currentPhi = lattice[site];

    // Generate new value
    double newValue = currentPhi + (3 * gsl_rng_uniform(generator) - 1.5);
    double temp = newValue;

    // Calculate energy difference
    double difference;
    difference = (currentPhi - newValue)
            * (lattice[neighbors[site]->nextX] * neighbors[site]->isRight
            +  lattice[neighbors[site]->nextY]
            +  lattice[neighbors[site]->prevX] * neighbors[site]->isLeft
            +  lattice[neighbors[site]->prevY]);

    newValue *= newValue;
    currentPhi  *= currentPhi;
    // Recall muSquared redefined
    difference += muSquared * (newValue - currentPhi);

    newValue *= newValue;
    currentPhi *= currentPhi;
    // Recall lambda redefined
    difference += lambda * (newValue - currentPhi);

    // Flip if difference <= 0, otherwise probabilistic acceptance
    if (difference <= 0)
        lattice[site] = temp;
    else if (difference > 500       // Prevent underflow
                || gsl_rng_uniform(generator) < gsl_sf_exp(-difference))
        lattice[site] = temp;
}
// ---------------------------------------------------------------



// ---------------------------------------------------------------
// Wolff methods for growing cluster and so on
// A convenience method that keeps me from having to write these few
// lines over and over again - adds to cluster probabilistically
// Value can be +/-lattice[toAdd] due to antiperiodic
// boundary conditions... need to pass as parameter
bool Lattice::clusterCheck(double site, double value, unsigned int toAdd) {
    if (cluster->find(toAdd))
        return false;

    if (site * value < 0)
        printf("%lf\n", site * value);
    if (site * value > 250 ||   // Prevent underflow
            gsl_rng_uniform(generator) < 1 - gsl_sf_exp(-2 * site * value)) {
```

```
        cluster->insert(toAdd);
        return true;
    }

    return false;
}

// Grows cluster from specified site - recursive
// Antiperiodic boundary conditions make a little ugly
void Lattice::growClusterPos(unsigned int site) {
    unsigned int toCheck;
    double current = lattice[site];
    double value;

    // Previous X
    toCheck = neighbors[site]->prevX;
    value = lattice[toCheck];
    if (neighbors[site]->isLeft == -1 && value <= 0) {
        value *= -1;
        if (clusterCheck(current, value, toCheck))
            growClusterNeg(toCheck);
    }
    else if (neighbors[site]->isLeft == 1 && value > 0)
        if (clusterCheck(current, value, toCheck))
            growClusterPos(toCheck);

    // Next X
    toCheck = neighbors[site]->nextX;
    value = lattice[toCheck];
    if (neighbors[site]->isRight == -1 && value <= 0) {
        value *= -1;
        if (clusterCheck(current, value, toCheck))
            growClusterNeg(toCheck);
    }
    else if (neighbors[site]->isRight == 1 && value > 0)
        if (clusterCheck(current, value, toCheck))
            growClusterPos(toCheck);

    // Previous Y
    toCheck = neighbors[site]->prevY;
    if (lattice[toCheck] > 0
            && clusterCheck(current, lattice[toCheck], toCheck))
        growClusterPos(toCheck);

    // Next Y
    toCheck = neighbors[site]->nextY;
    if (lattice[toCheck] > 0
            && clusterCheck(current, lattice[toCheck], toCheck))
        growClusterPos(toCheck);
}

void Lattice::growClusterNeg(unsigned int site) {
    unsigned int toCheck;
    double current = lattice[site];
    double value;

    // Previous X
    toCheck = neighbors[site]->prevX;
    value = lattice[toCheck];
    if (neighbors[site]->isLeft == -1 && value > 0) {
        value *= -1;
        if (clusterCheck(current, value, toCheck))
            growClusterPos(toCheck);
    }
    else if (neighbors[site]->isLeft == 1 && value <= 0)
        if (clusterCheck(current, value, toCheck))
            growClusterNeg(toCheck);

    // Next X
    toCheck = neighbors[site]->nextX;
    value = lattice[toCheck];
    if (neighbors[site]->isRight == -1 && value > 0) {
```

```
        value *= -1;
        if (clusterCheck(current, value, toCheck))
            growClusterPos(toCheck);
    }
    else if (neighbors[site]->isRight == 1 && value <= 0)
        if (clusterCheck(current, value, toCheck))
            growClusterNeg(toCheck);

    // Previous Y
    toCheck = neighbors[site]->prevY;
    if (lattice[toCheck] <= 0
            && clusterCheck(current, lattice[toCheck], toCheck))
        growClusterNeg(toCheck);

    // Next Y
    toCheck = neighbors[site]->nextY;
    if (lattice[toCheck] <= 0
            && clusterCheck(current, lattice[toCheck], toCheck))
        growClusterNeg(toCheck);
}

// Since this method trawls through the whole cluster (which has reached
// the end of its usefulness), let's be efficient and clear it here
void Lattice::flipCluster() {
    node* temp1;
    node* temp2;

    for (unsigned int i = 0; i < cluster->tableNumber; i++) {
        temp1 = cluster->table[i];
        temp2 = cluster->table[i];
        while (temp1 != NULL) {
            lattice[temp1->value] *= -1;
            temp2 = temp2->next;
            delete temp1;
            temp1 = temp2;
        }
        cluster->table[i] = NULL;
    }
    cluster->size = 0;
}

unsigned int Lattice::wolff(unsigned int site) {
    cluster->insert(site);

    // Inelegant, but simpler than having just one method
    if (lattice[site] > 0)
        growClusterPos(site);
    else
        growClusterNeg(site);

    unsigned int toReturn = cluster->size;
    flipCluster();

    return toReturn;
}
// -----------------------------------------------------------------
```

Code Snippet C.12: ϕ^4 soliton simulation code

```
// -----------------------------------------------------------------
// Soliton/Simulation.cpp
// Simulation to calculate mass of soliton on lattice
// David Schaich -- daschaich@gmail.com
// Created 23 January 2005
// Last modified 1 March 2006
// -----------------------------------------------------------------
```

```
// ------------------------------------------------------------------
// Include directives
#include <math.h>           // For floor and sqrt
#include <stdio.h>          // For printf and fprintf
#include <gsl/gsl_sf_log.h> // For natural log
#include <gsl/gsl_math.h>   // For power
#include "Lattice.hh"       // Method and data declarations
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Most data moved into lattice class.
unsigned int randomSite         = 0;
static const unsigned int bins  = 21;
unsigned int counts[bins];
double midCounts                = 0;            // Make these double now
double maxCounts                = 0;            // to avoid casts later
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Convenience method to set central time slice (vertical) to zero
// This confines the kink to the center of the lattice if called
void zeroCenter(Lattice* toCenter) {
    for (unsigned int i = toCenter->xDim / 2; i < toCenter->latticeSize;
            i += toCenter->xDim)

        toCenter->lattice[i] = 0;
}
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Start lattice in ground state to avoid metastable states
void coldStart(Lattice* toStart) {
    for (unsigned int i = 0; i < toStart->latticeSize; i++) {
        toStart->lattice[i] = (double)(i % toStart->xDim) -
            (double)(toStart->xDim / 2);
        toStart->lattice[i] /= toStart->xDim;
    }
}
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Calculate autocorrelation time and necessary points of
// autocorrelation function
// Use average phi (abs) for autocorrelation time calculation
// Should be roughly the same for all variables
unsigned int calcAutocor(Lattice* toCalc, unsigned int sampleSize) {
    // Generate Chi[0] for scaling purposes
    for (unsigned int i = 0; i < sampleSize; i++)
        toCalc->scaleFactor += (toCalc->phiDataAbs[i] * toCalc->phiDataAbs[i]);
    toCalc->scaleFactor /= sampleSize;
    toCalc->scaleFactor -= toCalc->avePhiAbs * toCalc->avePhiAbs;

    toCalc->autocor[0] = 1;

    // Only calculate points of autocorrelation function that
    // roughly conform to exponential approximation
    unsigned int t = 1;

    while (t < sampleSize) {
        for (unsigned int i = 0; i < sampleSize - t; i++)
            toCalc->autocor[t] +=
                    (toCalc->phiDataAbs[i] * toCalc->phiDataAbs[i + t]);
```

```
        toCalc->autocor[t] /= (sampleSize - t);
        toCalc->autocor[t] -= toCalc->avePhiAbs * toCalc->avePhiAbs;
        if (toCalc->autocor[t] < 0)         // Not exponential!
            break;

        toCalc->autocor[t] /= toCalc->scaleFactor;  // Scale by Chi[0]
        if (toCalc->autocor[t] >= toCalc->autocor[t - 1])   // Not exponential!
            break;

        double temp = -gsl_sf_log (toCalc->autocor[t]);
        temp = t / temp;
        toCalc->autocorTime += temp;

        t++;
    }

    return t;
}
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Bin values of phi and calculate bimodality
// Values of phi range from -maxPhi to +maxPhi
// Bin i of n will contain values greater than maxPhi(2*i/n - 1)
// and less than maxPhi*(2*(i+1)/n - 1)  (where i = 0...n - 1)
void calcBimodality(Lattice* toCalc, unsigned int sampleSize) {
    double lowerBound;
    double upperBound;
    for (unsigned int i = 0; i < bins; i++)     // Zero histogram
        counts[i] = 0;

    for (unsigned int i = 0; i < sampleSize; i++)
        for (unsigned int j = 0; j < bins; j++) {
            lowerBound = toCalc->maxPhi * (2 * (double)j / (double)bins - 1);
            upperBound = toCalc->maxPhi
                       * (2 * ((double)j + 1) / (double)bins - 1);

            if (toCalc->phiData[i] >= lowerBound
                    && toCalc->phiData[i] <= upperBound) {

                counts[j]++;
                break;
            }
        }

    midCounts = counts[(bins - 1) / 2];
    maxCounts = counts[0];
    for (unsigned int i = 1; i < bins; i++)
        if (counts[i] > maxCounts)
            maxCounts = counts[i];

    toCalc->bimodality = 1 - (midCounts / maxCounts);
}
// ------------------------------------------------------------------



// ------------------------------------------------------------------
// Main method runs simulation using command line parameters
int main(unsigned int argc, char** const argv) {
    if (argc != 7) {
        fprintf(stderr, "Usage: %s muSquared lambda xDim yDim ", argv[0]);
        fprintf(stderr, "init sampleSize\n");
        fflush(stderr);
        exit(1);
    }

    // Data that depends on command-line parameters
    double muSquared = atof(argv[1]) / 10;      // muSquared
```

```
    double lambda = atof(argv[2]) / 100;        // Selfcoupling strength
    unsigned int xDim = atoi(argv[3]);          // Lattice x-dimension
    unsigned int yDim = atoi(argv[4]);          // Lattice y-dimension
    unsigned int init = atoi(argv[5]);          // Iterations for equilibration
    unsigned int sampleSize = atoi(argv[6]);    // Iterations for statistics

    unsigned int latticeSize = xDim * yDim;

    // Periodic and antiperiodic boundary conditions
    // Expanded lattice class contains most data
    Lattice* peri = new Lattice(muSquared, lambda, xDim, yDim, sampleSize,
                    false);
    Lattice* anti = new Lattice(muSquared, lambda, xDim, yDim, sampleSize,
                    true);
//    coldStart(anti);        // Uncomment to initialize in ordered state
    zeroCenter(anti);       // Uncomment to constrain kink

    // Initialize/equilibrate lattice
    // Flip cluster and take data after every _gap_ Metropolis sweeps
    unsigned int gap = 5;
    for (unsigned int i = 0; i < init; i++) {
        for (unsigned int j = 0; j < latticeSize * gap; j++) {
            randomSite = (unsigned int) floor(latticeSize *
                    gsl_rng_uniform(peri->generator));
            peri->metropolis(randomSite);
            anti->metropolis(randomSite);
            zeroCenter(anti);       // Uncomment to constrain kink
        }

        randomSite = (unsigned int) floor(latticeSize *
                gsl_rng_uniform(peri->generator));
        peri->wolff(randomSite);
        anti->wolff(randomSite);
        zeroCenter(anti);       // Uncomment to constrain kink
    }

    for (unsigned int i = 0; i < sampleSize; i++) {
        for (unsigned int j = 0; j < latticeSize * gap; j++) {
            randomSite = (int) floor(latticeSize *
                    gsl_rng_uniform(peri->generator));
            peri->metropolis(randomSite);
            anti->metropolis(randomSite);
            zeroCenter(anti);       // Uncomment to constrain kink
        }

        randomSite = (unsigned int) floor(latticeSize *
                gsl_rng_uniform(peri->generator));
        peri->wolff(randomSite);
        anti->wolff(randomSite);
        zeroCenter(anti);       // Uncomment to constrain kink

        // Raw data and averages
        peri->energyData[i] = peri->calcTotalEnergy();
        anti->energyData[i] = anti->calcTotalEnergy();
        peri->aveEnergy += peri->energyData[i];
        anti->aveEnergy += anti->energyData[i];

        peri->phiData[i] = peri->calcAveragePhi();
        anti->phiData[i] = anti->calcAveragePhi();
        peri->phiDataAbs[i] = fabs(peri->phiData[i]);
        anti->phiDataAbs[i] = fabs(anti->phiData[i]);
        peri->avePhi += peri->phiData[i];
        anti->avePhi += anti->phiData[i];
        peri->avePhiAbs += peri->phiDataAbs[i];
        anti->avePhiAbs += anti->phiDataAbs[i];

        // Maximum (absolute) value of phi
        if (peri->phiDataAbs[i] > peri->maxPhi)
            peri->maxPhi = peri->phiDataAbs[i];
        if (anti->phiDataAbs[i] > anti->maxPhi)
            anti->maxPhi = anti->phiDataAbs[i];
```

```
    // Squared and quartic averages
    peri->squaredEnergy += peri->energyData[i] * peri->energyData[i];
    anti->squaredEnergy += anti->energyData[i] * anti->energyData[i];
    peri->squaredPhi += peri->phiData[i] * peri->phiData[i];
    anti->squaredPhi += anti->phiData[i] * anti->phiData[i];
    peri->quartPhi += gsl_pow_4(peri->phiData[i]);
    anti->quartPhi += gsl_pow_4(anti->phiData[i]);
}

// Take averages
peri->aveEnergy     /= sampleSize;
anti->aveEnergy     /= sampleSize;
peri->avePhi        /= sampleSize;
anti->avePhi        /= sampleSize;
peri->avePhiAbs     /= sampleSize;
anti->avePhiAbs     /= sampleSize;
peri->squaredEnergy /= sampleSize;
anti->squaredEnergy /= sampleSize;
peri->squaredPhi    /= sampleSize;
anti->squaredPhi    /= sampleSize;
peri->quartPhi      /= sampleSize;
anti->quartPhi      /= sampleSize;

// Calculate specific heat and susceptibility
// Can put bootstrapping/jackknifing in methods
peri->calcSpecificHeat();
anti->calcSpecificHeat();
peri->calcSusceptibility();
anti->calcSusceptibility();

// Now for autocorrelation time and standard deviation calcs
unsigned int t = calcAutocor(peri, sampleSize);
unsigned int tanti = calcAutocor(anti, sampleSize);

// Use standard formula to generate standard deviations
// from autocorrelation time, average, sampleSize, etc
if (t == 1) {            // Autocorrelation time zero
    peri->autocorTime = 0;
    peri->energyStDev = 0;
    peri->phiStDev    = 0;
}
else {
    peri->autocorTime /= (t - 1);
    peri->energyStDev = 2 * peri->autocorTime / sampleSize;
    peri->energyStDev *= peri->squaredEnergy
                       - (peri->aveEnergy * peri->aveEnergy);
    peri->energyStDev = sqrt(peri->energyStDev);

    peri->phiStDev = 2 * peri->autocorTime / sampleSize;
    peri->phiStDev *= peri->squaredPhi
                    - (peri->avePhiAbs * peri->avePhiAbs);
    peri->phiStDev = sqrt(peri->phiStDev);
}

if (tanti == 1) {       // Autocorrelation time zero
    anti->autocorTime = 0;
    anti->energyStDev = 0;
    anti->phiStDev    = 0;
}
else {
    anti->autocorTime /= (tanti - 1);
    anti->energyStDev = 2 * anti->autocorTime / sampleSize;
    anti->energyStDev *= anti->squaredEnergy
                       - (anti->aveEnergy * anti->aveEnergy);
    anti->energyStDev = sqrt(anti->energyStDev);

    anti->phiStDev = 2 * anti->autocorTime / sampleSize;
    anti->phiStDev *= anti->squaredPhi
                    - (anti->avePhiAbs * anti->avePhiAbs);
    anti->phiStDev = sqrt(anti->phiStDev);
}
```

```
    // Calculate binder cumulant
    peri->cumulant = 1 - peri->quartPhi
                   / (3 * peri->squaredPhi * peri->squaredPhi);
    anti->cumulant = 1 - anti->quartPhi
                   / (3 * anti->squaredPhi * anti->squaredPhi);

    // Calculate bimodality
    calcBimodality(peri, sampleSize);
    calcBimodality(anti, sampleSize);

    // Quantities for calculation of soliton mass
    double bcDiff = anti->aveEnergy - peri->aveEnergy;
    bcDiff *= latticeSize;      // Total - not per lattice site
    // Divide by lattice length in time direction in preparation for integral
    double bcDiffLength = bcDiff / yDim;

    printf("%lf,%lf,", muSquared, lambda);
    printf("%lf,%lf,", bcDiff, bcDiffLength);
    peri->printData();
    anti->printData();
    printf("\n");

    return 0;
}
// ----------------------------------------------------------------
```

C.5 Mathematica Analysis Code

In this section we present a few simple Mathematica scripts used to extract estimates for critical points of the ϕ^4 phase transitions, the corresponding uncertainties, and the masses of ϕ^4 solitons. They are included in large part to illustrate and clarify our earlier discussions of the methods we used to analyze our data.

Since these code snippets were relatively independent parts of largely uninteresting programs, we have simply excerpted them to present here, with some editing for length and readibility. An unfortunate side effect is that our scripts often involve certain variables defined elsewhere in the Mathematica notebook. To compensate, we will briefly introduce each code snippet and summarize any variables it uses but are not defined it its scope.

Code Snippet C.13 locates the peaks of the susceptibilities for all values of λ, finds the corresponding half-maxima and uncertainty, uncovers the points satisfying the $B \geq .95$ and $B \leq .1$ bounds of the (smoothed) bimodality (discussed in Subsection 7.4.2) and finds the point between those bounds at which the bimodality is closest to $B = .5$. All the data it uses is located in the four-dimensional table `data[[k, j, i, m]]`, where `k` specifies the size of the lattice, `j` specifies the value of λ, `i` specifies the value of μ_0^2 and `m` specifies a field in the output of the simulation defined by the previous three indices (for example, susceptibility is field 9, bimodality is field 11, and μ_0^2 is field 1). Since Mathematica apparently does not have a convenient way to extract the number of elements in a table, we use the table `length[[i, j]]` to specify the number of μ_0^2 data points sampled at each lattice size `i` and value of λ `j`.

Code Snippet C.13 also uses the two-dimensional tables `suscept`, `susceptErr`, `bimod`, `bimodErr` and `testBimod`, which are used to store (respectively) the estimates of the critical point calculated using susceptibilities and bimodalities, the uncertainties in those values, and the bimodalities corresponding to the susceptibility peak for each value of the lattice size L and value of λ. The infinite-volume limit is later calculated (beyond the scope of these code snippets) using simple linear regressions. Finally, there are the simple helper tables `sizeList` and `lambdaList`, which simply map the array index to the corresponding values of L and λ.

Code Snippet C.13: Mathematica script to analyze the susceptibility and bimodality

```
For[k = 1, k <= 6, k++, (* Loop over lattice sizes *)
    For[j = 1, j <= 9, j++, (* Loop over lambda values *)

        susceptIndex = 1;
        bimodIndex = 1;
        bimodDiff = 2;
        bimodLower = 0;
        bimodUpper = 0;

        For[i = 1, i <= length[[k, j]], i++,
            (* Locate maximum of susceptibility *)
            If[data[[k, j, i, 9]] >= data[[k, j, susceptIndex, 9]], susceptIndex = i];

            (* Bimodality upper bound: last data point under .1 *)
            If[data[[k, j, i, 11]] < .1, bimodUpper = i];

            (* Bimodality lower bound: first data point above .95 *)
            If[bimodLower == 0 && data[[k, j, i, 11]] > .95, bimodLower = i];

            (* Bimodality estimate closest te .5 *)
            If[Abs[.5 - data[[k, j, i, 11]]] < bimodDiff,
                bimodDiff = Abs[.5 - data[[k, j, i ,11]]];
                bimodIndex = i;
            ];
        ];

        (* Make sure bimod bounds in proper order *)
        (* Set new lower bound below upper bound if necessary *)
        If[bimodLower < bimodUpper,
            bimodLower = 0;

            (* First data point above .95 below upper bound *)
            For[i = bimodUpper, i <= length[[k, j]], i++,
                If[bimodLower == 0 && data[[k, j, i, 11]] > .95, bimodLower = i];
            ];
        ];

        (* Confine bimod value to within bounds *)
        (* Only look for estimate (closest to .5) between bounds *)
        For[i = bimodLower, i <= bimodUpper, i++,
            If[Abs[.5 - data[[k, j, i, 11]]] < bimodDiff,
                bimodDiff = Abs[.5 - data[[k, j, i, 11]]];
                bimodIndex = i;
            ];
        ];

        (* Store values calculated so far *)
        suscept[[k, j]] = data[[k, j, susceptIndex, 1]];
        testBimod[[k, j]] = data[[k, j, susceptIndex, 11]];
        bimod[[k, j]] = data[[k, j, bimodIndex, 1]];

        (* Find susceptibility half-maxima *)
        susceptMax = data[[k, j, susceptIndex, 9]];
        susceptLower = 1;
        susceptUpper = 1;
```

```
        (* Find first data point above half-max *)
        For[i = susceptIndex, i >= 1, i--,
            If[data[[k, j, i, 9]] > .5 * susceptMax, susceptUpper=i];
          ];

        (* Find last data point below half-max *)
        For[i = susceptIndex, i <= length[[k, j]], i++,
            If[data[[k, j, i, 9]] > .5 * susceptMax, susceptLower = i];
        ];

        (* Upper bound is last point before first above half-max *)
        (* Lower bound is first point after last above half-max *)
        If[susceptUpper != 1, susceptUpper = susceptUpper - 1];
        If[susceptLower != length[[k,j]] && susceptLower != 1, susceptLower = susceptLower + 1];

        (* Do we have enough data for clear bounds? *)
        If[susceptLower == 1 || susceptLower == length[[k, j]], Print["Warning: lower half-max not
                found: ", sizeList[[k]], "-", lambdaList[[j]]]];
        If[susceptUpper == 1, Print["Warning: upper half-max not found: ",    sizeList[[k]], "-",
                lambdaList[[j]]]];
        If[bimodLower == 0,   Print["Warning: lower bimod bound not found: ", sizeList[[k]], "-",
                lambdaList[[j]]];
                bimodLower = 1];
        If[bimodUpper == 0,   Print["Warning: upper bimod bound not found: ", sizeList[[k]], "-",
                lambdaList[[j]]];
                bimodUpper = 1];

        (* Store uncertainties *)
        susceptErr[[k, j]] = (data[[k, j, susceptUpper, 1]] - data[[k, j, susceptLower, 1]]) / 2;
        bimodErr[[k,j]] = (data[[k, j, bimodUpper, 1]] - data[[k, j, bimodLower, 1]]) / 4;
    ];
];
```

Code Snippet C.14 locates the stationary point of the Binder cumulant, considering for simplicity only the largest three lattice sizes (in this case $L = 20$, 24 and 36, since this particular code originally analyzed data from four-dimensional simulations). It operates by first finding the value of μ_0^2 below which simulations on larger lattices always have larger Binder cumulants and the value above which simulations on larger lattices always have smaller cumulants (at least until the cumulants are all randomly fluctuating around zero). This calculation is simplified somewhat by considering the differences between the Binder cumulants for the different lattice sizes, defined below as `sepX, X` $= 1, 2, 3$. These two values correspond to the lower and upper bounds on the estimate of the critical point, which is then identified by selecting the point between the two bounds at which the separation between the cumulants is smallest. See Subsection 7.4.3 for background and additional details.

Code Snippet C.14 uses several of the tables introduced in Code Snippet C.13 and described above, namely `data`, `length`, `sizeList` and `lambdaList` (the Binder cumulant is the tenth field in the final index of `data`). In addition, it also uses the one-dimensional tables `binder` and `binderErr`, which store the estimates of the critical point and corresponding uncertainties (respectively) for each value of λ.

Code Snippet C.14: Mathematica script to analyze the Binder cumulant

```
upperLimit = Table[0, {9}];
lowerLimit = Table[0, {9}];
```

```
binder = Table[0, {9}];
binderErr = Table[0, {9}];

For[j = 1, j <= 9, j++,
    (* Only look at the points calculated for three largest lattice sizes*)
    sep1 = Table[0, {length[[6, j]]}];
    sep2 = Table[0, {length[[6, j]]}];
    sep3 = Table[0, {length[[6, j]]}];

    (* Sync 20^4 and 24^4 data with 36^4 data *)
    iter24 = 0;
    While[data[[5, j, 1 + iter24, 1]] != data[[6, j, 1, 1]], iter24++;];

    iter20 = 0;
    While[data[[4, j, 1 + iter20, 1]] != data[[6, j, 1, 1]], iter20++;];

    (* Calculate differences between Binder cumulants for different lattice sizes *)
    For[i = 1, i <= length[[6, j]], i++,
        sep1[[i]] = data[[6, j, i, 10]] - data[[5, j, i + iter24, 10]];
        sep2[[i]] = data[[6, j, i, 10]] - data[[4, j, i + iter20, 10]];
        sep3[[i]] = data[[5, j, i + iter24, 10]] - data[[4, j, i + iter20, 10]];
    ];

    (* Now go through and look at signs of differences *)
    (* "Neg" (upper) limit when all preceding differences negative *)
    (* "Pos" (lower) limit when all following differences positive *)
    Neg = 0;
    For[i = 1, i <= length[[6, j]], i++,
        (* Need to ignore random fluctuations at very low values: require average > 0.25 *)
        If[sep1[[i]] < 0 && sep2[[i]] < 0 && sep3[[i]] < 0 && Neg == i - 1
                || (data[[6, j, i, 10]] + data[[5, j, i + iter24, 10]] + data[[4, j, i + iter20, 10]])
                    / 3 < .25,

            Neg = i;
        ];
    ];
    upperLimit[[j]] = data[[6, j, Neg, 1]];

    Pos = length[[6, j]] + 1;
    For[i = length[[6, j]], i > 0, i--,
        If[sep1[[i]] > 0 && sep2[[i]] > 0 && sep3[[i]] > 0 && Pos == i + 1, Pos = i;];
    ];

    (* Do we have enough data for clear bounds? *)
    If[Pos == length[[6, j]] + 1,
        Print[j, ": Warning: Binder cumulant never fully positive.  Setting to max."]; Pos--;];
        lowerLimit[[j]] = data[[6, j, Pos, 1]
    ];

    (* Extract uncertainties from limits *)
    binderErr[[j]] = (upperLimit[[j]] - lowerLimit[[j]]) / 2;

    (* Extract fixed point (critical point) looking only at data between limits *)
    (* Define smallest sum of differences as fixed point *)
    min = 10;
    index = 0;
    For[i = Neg, i <= Pos, i++,
        temp = Abs[sep1[[i]] + sep2[[i]] + sep3[[i]]];
        If[temp < min, min = temp; index = i;];
        binder[[j]] = data[[6, j, index, 1]];
    ];
];
```

Finally, Code Snippet C.15 analyzes the mass of the ϕ^4 solitons by extracting the difference in action (per time slice) between simulations run with periodic and antiperiodic boundary conditions (field 4 in the fourth index of `data`) and propagating

the uncertainties in the actions. As described above in Section 7.6, we calculated these values for lattices of five increasing sizes, $L = 32$, 48, 64, 128 and 256. After extracting this data, we make a linear regression to the infinite-volume limit. The final section of code plugs the continuum limit regressions into the integral, Eqn 7.27, used to calculate the mass of the soliton. Since identical procedures are performed for $\mu_0^2 =$ -1, -2.2 and -4, we have cut the calculations for $\mu_0^2 =$ -2.2 and -4.

In addition to `data` introduced from above, Code Snippet C.15 uses `lambdaFit`, a equation for critical bare μ_0^2 in terms of λ obtained through a nonlinear regression involving terms up to and including order $\lambda^2 \log[\lambda]$ (see Section 7.6). It also uses the table `beta`, which maps the array index to the corresponding value of $1/\lambda$.

Code Snippet C.15: Mathematica script to analyze the soliton mass

```
(* Mu^2 = -1 *)
actionData = Table[0, {100}];
massData = Table[0, {100}];
regressData = Table[0, {100}];
regressHighData = Table[0, {100}];
regressLowData = Table[0, {100}];

For[i = 1, i <= 100, i++,

    (* Take continuum limit from different-sized lattices *)
    (* Simple error propagation -- recall dividing by T *)
    For[j = 1, j <= 5, j++,
        tempData[[j]] = data[[i,j,i,4]];
        tempWeights[[j]] = Abs[tempData[[j]] * Sqrt[(data[[i,j,i,7] / data[[i,j,i,6]])^2
                + (data[[i,j,i,18] / data[[i,j,i,17]])^2] / (2 * sizeData[[j]]);
    ];

    regress = Regress[tempData, {i, x}, x, Weights -> tempWeights];
    regressData[[i]] = regress[[1, 2, 1, 1, 1]] * data[[1, 5, i, 2]]; (* (Delta S / T) * Lambda *)

    (* High estimate from errors *)
    regressHighData[[i]] = (regress[[1, 2, 1, 1, 1]] + regress[[1, 2, 1, 1, 2]]) * data[[1, 5, i, 2]];

    (* Low estimate from errors *)
    regressLowData[[i]] = (regress[[1, 2, 1, 1, 1]] - regress[[1, 2, 1, 1, 2]]) * data[[1, 5, i, 2]];

    (* We're going to square this, so remove negative values by hand *)
    If[regress[[1, 2, 1, 1, 1]] > 0
        actionData[[i]] = {{1 / data[[1, 5, i, 2]], (regress[[1, 2, 1, 1, 1]] * data[[1, 5, i, 2]])^2},
                ErrorBar[2 * regress[[1, 2, 1, 1, 2]] * data[[1, 5, i, 2]]]},
        actionData[[i]] = {1 / data[[1, 5, i, 2]], 0}, ErrorBar[0]};
    ];

    (* ListInterpolation to get integrable functions from data *)
    Integrand = ListInterpolation[regressData, {beta}];
    IntegrandHigh = ListInterpolation[regressHighData, {beta}];
    IntegrandLow = ListInterpolation[regressLowData, {beta}];

    (* Find critical beta *)
    betaC = FindRoot[lambdaFit[[1, 2, 1, 2]] + lambdaFit[[1, 2, 2, 2]] * x
            + lambdaFit[[1, 2, 3, 2]] * x * Log[x]
            + lambdaFit[[1, 2, 4, 2]] * x^2 * Log[x] == -1, {x, .1}];
    betaC = 1 / betaC[[1, 2]]

    (* Calculate masses -- again set negatives to zero *)
    (* Uncertainties from high and low estimates (need absolute value) *)
    For[i = 1, i <= 100, i++,
        massData[[i]] = {{1 / data[[1, 5, i, 2]], (Integrate[Integrand[x], {x, betaC,
                1/data[[1, 5, i, 2]]}])^2 * (data[[1, 5, i, 2]])^2},
                ErrorBar[Abs[(Integrate[IntegrandHigh[x], {x, betaC, 1 / data[[1, 5, i, 2]]}]
                - Integrate[IntegrandLow[x], {x, betaC, 1 / data[[1, 5, i, 2]]}])] / 2]};
        If[Integrate[Integrand[x], {x, betaC, 1/data[[1, 5, i, 2]]}] < 0,
```

```
            massData[[i]] = {{1/data[[1, 5, i, 2]], 0}, ErrorBar[0]};
        ];
    ];
];

(* Mu^2 = -2.2 *)
[ ... ]

(* Mu^2 = -4 *)
[ ... ]
```

Appendix D: Mathematica Regressions

In this final appendix we present the output of the nonlinear regressions we used to determine the continuum limit of the two-dimensional critical coupling constant $[\lambda/\mu^2]_{crit}$ as a function of λ, as described above in Section 7.5. These regressions were originally performed using Mathematica 5.0 running on Linux. When we later happened to run the same program on a Windows machine running Mathematica 5.2, we were surprised to notice striking disagreements with our earlier results. In this section we present the output of regressions performed on both platforms, for each of the following three models:

$$\text{Linear}: \qquad \left[\lambda/\mu^2\right] = c_1 + c_2\lambda \tag{D.1}$$

$$\lambda\log[\lambda]: \qquad \left[\lambda/\mu^2\right] = c_1 + c_2\lambda + c_3\lambda\log[\lambda] \tag{D.2}$$

$$\lambda^2\log[\lambda]: \qquad \left[\lambda/\mu^2\right] = c_1 + c_2\lambda + c_3\lambda\log[\lambda] + c_4\lambda^2\log[\lambda] \tag{D.3}$$

Although Mathematica is capable of converting its 'notebook' documents into LaTeX, the output produced in this manner does not play nicely with the other children. Accordingly, we have converted the relevant information into images, which we include below. Figs. D.1 and D.2 show that linear regressions performed on both platforms agree perfectly, while Figs. D.3 through D.6 show serious disagreements in the results of nonlinear regressions between the two platforms. As we mentioned above in Section 7.5, we adopted the results from Mathematica 5.2 on Windows, since that is a newer version of the software operating in its natural habitat.

```
regress = Regress[crit, {1, x}, x, Weights → weights]
line = Plot[regress[[1, 2, 1, 1, 1]] + regress[[1, 2, 1, 2, 1]] x, {x, 0, 1.01}];
Show[pointsPlot, line];
```

```
{ParameterTable →
```

	Estimate	SE	TStat	PValue
1	10.305	0.0645799	159.569	1.00142×10^{-13},
x	-0.0550767	0.0825315	-0.667342	0.525934

```
RSquared → 0.0598153, AdjustedRSquared → -0.0744968, EstimatedVariance → 47.8105,
```

		DF	SumOfSq	MeanSq	FRatio	PValue
ANOVATable →	Model	1	21.2922	21.2922	0.445346	0.525934}
	Error	7	334.673	47.8105		
	Total	8	355.966			

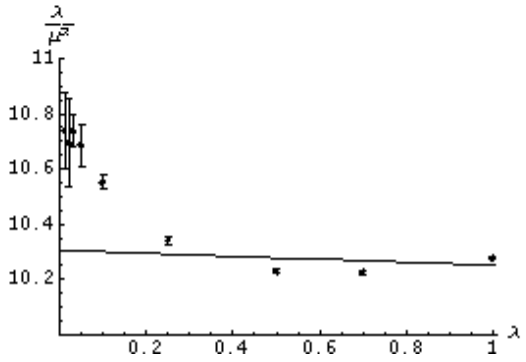

Figure D.1: Linear regression – Mathematica 5.0 – Linux

```
regress = Regress[crit, {1, x}, x, Weights → weights]
line = Plot[regress[[1, 2, 1, 1, 1]] + regress[[1, 2, 1, 2, 1]] x, {x, 0, 1.01}];
Show[pointsPlot, line];
```

```
{ParameterTable →
        Estimate        SE              TStat           PValue
  1     10.305          0.0645799       159.569         1.00142 × 10^-13,
  x     -0.0550767      0.0825315       -0.667342       0.525934

 RSquared → 0.0598153, AdjustedRSquared → -0.0744968,
 EstimatedVariance → 47.8105, ANOVATable →
            DF      SumOfSq     MeanSq      FRatio      PValue
  Model     1       21.2922     21.2922     0.445346    0.525934
  Error     7       334.673     47.8105
  Total     8       355.966
```

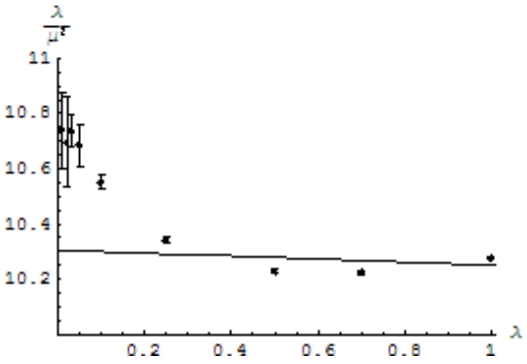

Figure D.2: Linear regression – Mathematica 5.2 – Windows

```
nonlin = NonlinearRegress[crit, c1 + c2 * lambda + c3 * lambda * Log[lambda], {lambda},
  {c1, c2, c3}, Weights → weights]
logLine = Plot[nonlin[[1, 2, 1, 2]] + nonlin[[1, 2, 2, 2]] * x + nonlin[[1, 2, 3, 2]] * x * Log[x],
   {x, 0, 1.01}];
Show[pointsPlot, logLine];
```

```
{BestFitParameters → {c1 → 10.692, c2 → -0.416293, c3 → 0.727248},

                                  Estimate      Asymptotic SE     CI
ParameterCITable →  c1            10.692        0.0461295         {10.5791, 10.8049}
                    c2            -0.416293     0.0454669         {-0.527547, -0.30504}
                    c3            0.727248      0.0787183         {0.534631, 0.919865}

EstimatedVariance → 2.7967,

                                  DF    SumOfSq            MeanSq
                    Model         3     1.02961×10^7       3.43203×10^6
ANOVATable →        Error         6     16.7802            2.7967
                    Uncorrected Total   9   1.02961×10^7
                    Corrected Total     8   355.966

                                  ( 1.        -0.985065   0.940933  )
AsymptoticCorrelationMatrix →     ( -0.985065  1.         -0.898477 ),
                                  ( 0.940933  -0.898477    1.       )

                                                  Curvature
FitCurvatureTable →  Max Intrinsic                0
                     Max Parameter-Effects        0
                     95. % Confidence Region      0.458491
```

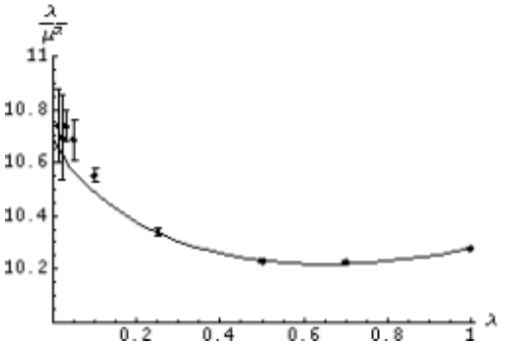

Figure D.3: $\lambda \log[\lambda]$ regression – Mathematica 5.0 – Linux

```
nonlin = NonlinearRegress[crit, c1 + c2 * lambda + c3 * lambda * Log[lambda],
  {lambda}, {c1, c2, c3}, Weights → weights]
logLine = Plot[nonlin[[1, 2, 1, 2]] + nonlin[[1, 2, 2, 2]] * x + nonlin[[1, 2, 3, 2]] * x * Log[x],
   {x, 0, 1.01}];
Show[pointsPlot, logLine];
```

```
{BestFitParameters → {c1 → 10.7744, c2 → -0.49687, c3 → 0.851321},

                          Estimate     Asymptotic SE     CI
ParameterCITable → c1     10.7744      0.0309908         {10.6986, 10.8502}
                   c2     -0.49687     0.0305456         {-0.571612, -0.422128}
                   c3     0.851321     0.0528846         {0.721917, 0.980725}

EstimatedVariance → 1.26227,

                                  DF     SumOfSq          MeanSq
                   Model          3      1.02961×10^7     3.43203×10^6
ANOVATable → Error                6      7.57362          1.26227      ,
                   Uncorrected Total     9      1.02961×10^7
                   Corrected Total       8      355.966

                                                  1.          -0.985065   0.940933
AsymptoticCorrelationMatrix → {{-0.985065, 1.}, {-0.985065,   1.          -0.898477},
                                                  0.940933    -0.898477   1.

                                                  Curvature
FitCurvatureTable → Max Intrinsic                 0
                    Max Parameter-Effects         0          }
                    95. % Confidence Region       0.458491
```

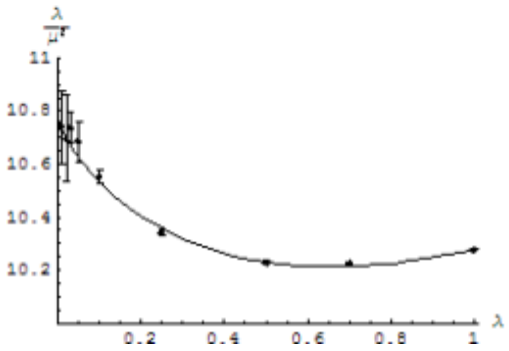

Figure D.4: $\lambda \log[\lambda]$ regression – Mathematica 5.2 – Windows

```
nonlin = NonlinearRegress[crit, c1 + c2 * lambda + c3* lambda * Log[lambda] + c4* lambda^2*Log[lambda],
  {lambda}, {c1, c2, c3, c4}, Weights → weights]
fourLine = Plot[nonlin[[1, 2, 1, 2]] + nonlin[[1, 2, 2, 2]] * x + nonlin[[1, 2, 3, 2]] * x*Log[x] +
    nonlin[[1, 2, 4, 2]] * x^2*Log[x], {x, 0, 1.01}];
Show[pointsPlot, fourLine];
```

```
{BestFitParameters → {c1 → 10.8616, c2 → -0.586309, c3 → 1.17505, c4 → -0.381804},

                          Estimate      Asymptotic SE     CI
                    c1    10.8616       0.0180857         (10.8151, 10.9081)
ParameterCITable →  c2    -0.586309     0.0182677         (-0.633267, -0.53935) ,
                    c3    1.17505       0.0551279         (1.03334, 1.31677)
                    c4    -0.381804     0.0595629         (-0.534915, -0.228693)

EstimatedVariance → 0.149821,

                                DF    SumOfSq            MeanSq
            Model               4     1.02961 × 10^7     2.57402 × 10^6
ANOVATable → Error              5     0.749107           0.149821      ,
            Uncorrected Total   9     1.02961 × 10^7
            Corrected Total     8     355.966

                                  ⎛    1.     -0.994766   0.945376   -0.807148 ⎞
AsymptoticCorrelationMatrix →     ⎜-0.994766      1.     -0.942529    0.8174   ⎟
                                  ⎜ 0.945376  -0.942529      1.      -0.943807 ⎟
                                  ⎝-0.807148    0.8174   -0.943807      1.    ⎠ ,

                                                  Curvature
FitCurvatureTable →  Max Intrinsic                0          }
                     Max Parameter-Effects        0
                     95. % Confidence Region      0.43886
```

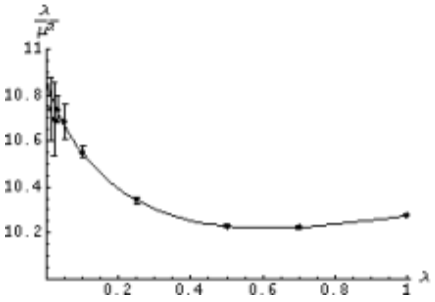

Figure D.5: $\lambda^2 \log[\lambda]$ regression – Mathematica 5.0 – Linux

```
nonlin = NonlinearRegress[crit, c1 + c2 * lambda + c3* lambda * Log[lambda] + c4* lambda^2* Log[lambda],
  {lambda}, {c1, c2, c3, c4}, Weights → weights]
fourLine = Plot[nonlin[[1, 2, 1, 2]] + nonlin[[1, 2, 2, 2]] * x + nonlin[[1, 2, 3, 2]] * x * Log[x] +
    nonlin[[1, 2, 4, 2]] * x^2 * Log[x], {x, 0, 1.01}];
Show[pointsPlot, fourLine];
```

```
{BestFitParameters → {c1 → 10.8735, c2 → -0.598206, c3 → 1.20442, c4 → -0.404223},

                          Estimate      Asymptotic SE     CI
                    c1    10.8735       0.0171473         {10.8294, 10.9175}
 ParameterCITable → c2    -0.598206     0.0173198         {-0.642728, -0.553684},
                    c3    1.20442       0.0522675         {1.07006, 1.33878}
                    c4    -0.404223     0.0564723         {-0.549389, -0.259056}

 EstimatedVariance → 0.134677,

                                  DF    SumOfSq           MeanSq
              Model               4     1.02961×10^7      2.57402×10^6
 ANOVATable → Error               5     0.673386          0.134677     ,
              Uncorrected Total   9     1.02961×10^7
              Corrected Total     8     355.966

 AsymptoticCorrelationMatrix → {{1.,        -0.994766, 0.945376,  -0.807148},
                                {-0.994766, 1.,        -0.942529, 0.8174},
                                {0.945376,  -0.942529, 1.,        -0.943807},
                                {-0.807148, 0.8174,    -0.943807, 1.}},

                                  Curvature
 FitCurvatureTable → Max Intrinsic  0
                     Max Parameter-Effects  0
                     95. % Confidence Region  0.43886
```

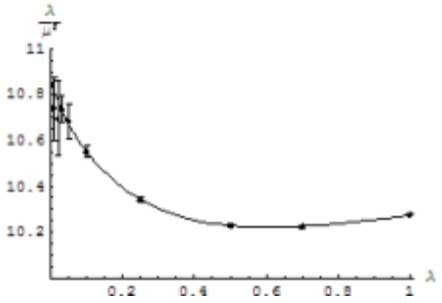

Figure D.6: $\lambda^2 \log[\lambda]$ regression – Mathematica 5.2 – Windows

Bibliography

[1] A. Ardekani and A. G. Williams. Lattice Study of the Kink Soliton and the Zero-mode Problem for ϕ^4 in Two Dimensions. *arXiv:hep-lat/9811002*, Nov. 1998.

[2] D. C. Baird. *Experimentation: An Introduction to Measurement Theory and Experiment Design*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1995.

[3] J. Balog, A. Duncan, R. Willey, F. Niedermayer, and P. Weisz. The $4d$ One Component Lattice ϕ^4 Model in the Broken Phase Revisited. *Nucl. Phys. B*, 714:256, May 2005.

[4] J. Balog, F. Niedermayer, and P. Weisz. Repairing Stevenson's Step in the $4d$ Ising Model. *arXiv:hep-lat/0601016*, Jan. 2006.

[5] C. J. Bednarzyk. Monte Carlo Simulations of ϕ^4 Theory. Honors Thesis: Amherst College, May 2001.

[6] B. A. Berg. *Markov Chain Monte Carlo Simulations and Their Statistical Analysis*. World Scientific, Singapore, 2004.

[7] K. Binder and D. W. Heermann. *Monte Carlo Simulation in Statistical Physics: An Introduction*. Springer, Berlin, fourth edition, 2005.

[8] A. I. Bochkarev and R. S. Willey. Electroweak Parameters in the $\bar{M}S$-Scheme. *arXiv:hep-ph/9607240*, July 1996.

[9] R. C. Brower and P. Tamayo. Embedded Dynamics for ϕ^4 Theory. *Phys. Rev. Lett.*, 62:1087, 1989.

[10] S. Caracciolo, A. J. Guttmann, I. Jensen, A. Pelissetto, A. N. Rogers, and A. D. Sokal. Correction-to-Scaling Exponents for Two-Dimensional Self-Avoiding Walks. *arXiv:cond-mat/0409355*, Sep. 2004.

[11] P. Cea, M. Consoli, and L. Cosmai. Large Logarithmic Rescaling of the Scalar Condensate: New Lattice Evidences. *arXiv:hep-lat/0407024*, July 2004.

[12] P. Cea, M. Consoli, and L. Cosmai. Large Logarithmic Rescaling of the Scalar Condensate: A Subtlety with Substantial Phenomenological Implications. *arXiv:hep-lat/0501013*, Jan. 2005.

[13] S.-J. Chang. Existence of a Second-Order Phase Transition in a Two-Dimensional ϕ^4 Field Theory. *Phys. Rev. D.*, 13(6):2778, 1976.

[14] B. A. Cipra. The Ising Model Is NP-Complete. *SIAM News*, 33(6), July 2000. http://www.siam.org/pdf/news/654.pdf (last accessed 10 May 2006).

[15] J. C. Ciria and A. Tarancón. Renormalization Group Study of the Soliton Mass in the (1 + 1)-Dimensional $\lambda\phi^4$ Lattice Model. *Phys. Rev. D.*, 49(2):1020, Jan. 1994.

[16] A. K. De, A. Harindranath, J. Maiti, and T. Sinha. Topological Charge in $1 + 1$ Dimensional Lattice ϕ^4 Theory. *Phys. Rev. D.*, 72(9):094504, Nov. 2005.

[17] I. Dukovski, J. Machta, and L. V. Chayes. Invaded Cluster Simulations of the XY Model in Two and Three Dimensions. *Phys. Rev. E.*, 65:026702, Jan. 2002.

[18] F. J. Dyson. The S Matrix in Quantum Electrodynamics. *Phys. Rev.*, 75(11):1736, June 1949.

[19] R. P. Feynman and A. R. Hibbs. *Quantum Mechanics and Path Integrals*. McGraw-Hill, New York, 1965.

[20] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi. *GNU Scientific Library: Reference Manual*. Network Theory Ltd., Bristol, second edition, 2003. http://www.gnu.org/software/gsl/ (last accessed 10 May 2006).

[21] T. W. Gamelin. *Complex Analysis*. Undergraduate Texts in Mathematics. Springer, New York, 2001.

[22] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, Hoboken, third edition, 2004. http://www.datastructures.net (last accessed 10 May 2006).

[23] B. Hatfield. *Quantum Field Theory of Point Particles and Strings*, volume 75 of *Frontiers in Physics*. Westview Press, Boulder, Colorado, 1998.

[24] E. Ising. Beitrag zur Theorie des Ferromagnetizmus. *Z. Phys.*, 31:253, 1925.

[25] A. V. Izyumov and K. V. Samokhin. Field Theory of Self-Avoiding Walks in Random Media. *arXiv:cond-mat/9909407*, Sep. 1999.

[26] J. D. Jackson. *Classical Electrodynamics*. John Wiley & Sons, Inc., New York, third edition, 1999.

[27] H. Kleinert and V. Schulte-Frohlinde. *Critical Properties of ϕ^4-Theories*. World Scientific, Singapore, 2001.

[28] P. Kraus and D. Griffiths. Renormalization of a Model Quantum Field Theory. *Am. J. Phys.*, 60(11):1013, Nov. 1992.

[29] L. D. Landau and E. M. Lifshitz. *The Classical Theory of Fields*, volume 2 of *Course of Theoretical Physics*. Butterworth-Heinemann, Oxford, fourth edition, 2000.

[30] G. F. Lawler. A Self-Avoiding Random Walk. *Duke Math. J.*, 47(3):655–693, Sep. 1980.

[31] G. P. Lepage. What is Renormalization? *arXiv:hep-ph/0506330*, June 1989.

[32] W. Loinaz and R. S. Willey. Monte Carlo Simulation Calculation of Critical Coupling Constant for Continuum ϕ^4 Theory. *Phys. Rev. D*, 58:076003, Sep. 1998.

[33] M. Lüscher and P. Weisz. Scaling Laws and Triviality Bounds in the Lattice ϕ^4 Theory: (I). One-Component Model in the Symmetric Phase. *Nuc. Phys. B*, 290:25–60, 1987.

[34] M. Lüscher and P. Weisz. Scaling Laws and Triviality Bounds in the Lattice ϕ^4 Theory: (II). One-Component Model in the Phase with Spontaneous Symmetry Breaking. *Nuc. Phys. B*, 295:65–92, 1988.

[35] J. Machta, Y. S. Choi, A. Lucke, T. Schweizer, and L. V. Chayes. Invaded Cluster Algorithm for Equilibrium Critical Points. *Phys. Rev. Lett.*, 75:2792–2795, Oct. 1995.

[36] J. Machta, Y. S. Choi, A. Lucke, T. Schweizer, and L. V. Chayes. Invaded Cluster Algorithm for Potts Models. *Phys. Rev. E.*, 54:1332–1345, Aug. 1996.

[37] M. Matsumoto and T. Nishimura. A 623-Dimensionally Equidistributed Uniform Pseudorandom Number Generator. *ACM Transactions on Modeling and Computer Simulation*, 8:3, Jan. 1998. http://www.math.sci.hiroshima-u.ac.jp/∼m-mat/MT/emt.html (last accessed 10 May 2006).

[38] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equations of State Calculations by Fast Computing Machines. *J. Chem. Phys.*, 21(6):1087–1092, June 1953.

[39] I. Montvay and G. Münster. *Quantum Fields on a Lattice*. Cambridge Monographs on Mathematical Physics. Cambridge University Press, Cambridge, 1994.

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

[41] L. Onsager. Crystal Statistics I. A Two-Dimensional Model with an Order-Disorder Transition. *Phys. Rev.*, 65:117, 1944.

[42] Z. Parsa. Topological Solitons in Physics. *Am. J. Phys.*, 47:56, Jan. 1978.

[43] M. E. Peskin and D. V. Schroeder. *An Introduction to Quantum Field Theory*. Addison-Wesley Publishing Co., Reading, MA, 1995.

[44] M. Di Pierro. From Monte Carlo Integration to Lattice Quantum Chromo Dynamics. *arXiv:hep-lat/0009001*, Oct. 2001.

[45] M. Di Pierro. An Algorithmic Approach to Quantum Field Theory. *arXiv:hep-lat/0509013*, Sep. 2005.

[46] R. B. Potts. Some Generalized Order-Disorder Transformations. *Proc. Cambridge Philos. Soc.*, 48:106, 1952.

[47] N. Prokof'ev and B. Svistunov. Worm Algorithms for Classical Statistical Models. *Phys. Rev. Lett.*, 87(16):160601, Oct. 2001.

[48] N. V. Prokof'ev, B. V. Svistunov, and I. S. Tupitsyn. "Worm" Algorithm in Quantum Monte Carlo Simulations. *Phys. Lett. A*, 238:253, Feb. 1998.

[49] R. Rajaraman. *Solitons and Instantons*. North Holland, Amsterdam, 1987.

[50] P. Ramond. *Field Theory: A Modern Primer*, volume 51 of *Frontiers in Physics*. The Benjamin/Cummings Publishing Co., Inc., Reading, MA, first edition, 1981.

[51] L. H. Ryder. *Quantum Field Theory*. Cambridge University Press, Cambridge, second edition, 1996.

[52] A. W. Sandvik. Ground State Projection of Quantum Spin Systems. *arXiv:cond-mat/0509558*, Sep. 2005. To appear in Phys. Rev. Lett.

[53] S. R. Shannon, T. C. Choy, and R. J. Fleming. An Improved Perturbation Approach to the 2D Edwards Polymer. *arXiv:cond-mat/9511010*, Nov. 1995.

[54] J. Smit. *Introduction to Quantum Fields on a Lattice*, volume 15 of *Cambridge Lecture Notes in Physics*. Cambridge University Press, Cambridge, 2002.

[55] A. D. Sokal. Monte Carlo Methods for the Self-Avoiding Walk. *arXiv:hep-lat/9405016*, May 1994.

[56] F. Spitzer. *Principles of Random Walk*, volume 34 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, second edition, 2001.

[57] P. M. Stevenson. Comparison of Conventional RG Theory with Lattice Data for the 4*d* Ising Model. *arXiv:hep-lat/0507038*, July 2005.

[58] X. Sun. Monte Carlo Studies of Three-Dimensional O(1) and O(4) ϕ^4 Theory Related to Bose-Einstein Condensation Phase Transition Temperatures. *Phys. Rev. E.*, 67(6):066702, June 2003.

[59] R. H. Swendsen and J.-S. Wang. Nonuniversal Critical Dynamics in Monte Carlo Simulations. *Phys. Rev. Lett.*, 58:86, 1987.

[60] J. R. Taylor. *An Introduction to Error Analysis*. A Series of Books in Physics. University Science Books, Mill Valley, CA, 1982.

[61] W. Thirring. *Classical Mathematical Physics: Dynamical Systems and Field Theories*. Springer, New York, third edition, 2003.

[62] J. S. Townsend. *A Modern Approach to Quantum Mechanics*. University Science Books, Sausalito, CA, 2000.

[63] M. Veltman. *Diagrammatica*. Cambridge Lecture Notes in Physics. Cambridge University Press, Cambdridge, 1994.

[64] A. Watson. *The Quantum Quark*. Cambridge University Press, Cambridge, 2004.

[65] T. Weidig. Quantum Mass Correction of Solitons in (1 + 1)D via Numerical Methods. *arXiv:hep-th/9912005*, Dec. 1999.

[66] S. Weinberg. High-Energy Behavior in Quantum Field Theory. *Phys. Rev.*, 118(3):838, May 1960.

[67] E. W. Weisstein. Random walk – 2 dimensional. *MathWorld – A Wolfram Web Resource*, 2006. http://mathworld.wolfram.com/RandomWalk2-Dimensional.html (last accessed 10 May 2006).

[68] G. V. Wilson. Where's the Real Bottleneck in Scientific Computing? *Am. Scientist*, 94(1):5, Jan. 2006. http://www.americanscientist.org/template/AssetDetail/assetid/48548 (last accessed 24 April 2006).

[69] U. Wolff. Collective Monte Carlo Updating for Spin Systems. *Phys. Rev. Lett.*, 62(4):361, Jan. 1989.

[70] A. Zee. *Quantum Field Theory in a Nutshell*. Princeton University Press, Princeton, 2003.