Kaluza-Klein Quantum Fields on the Lattice

James Hackett

Abstract

The ultimate aim of this project is to develop a parallel algorithm that creates a picture of the interaction between a five dimensional field and a purely four dimensional field of similar character. The motivation behind the simulation comes from Kaluza-Klein theory, which was originally used to encompass the four dimensional theories of general relativity and electromagnetism in a concise five dimensional theory. My thesis is a precursory venture into the world of the quantum lattice. Hence, it's secondary aim is to explore the most basic quantum lattice objects, in order to ensure the accuracy of the final simulations and to come to better understanding of the underlying physics. The simulations begin with a one dimensional free quantum field, and move on to higher dimensions and interacting fields, before the Kaluza-Klein based model. All simulations involve measuring the two-point correlation function on a lattice, and conclusions are derived from it's alteration as the nature of the field is changed. The main conclusion is therefore an interpretation of the differences between the correlation function for two interacting fields of the same dimension and the correlation function for the Kaluza-Klein interaction, giving us an example of how the theory should affect experiment.

Contents

1 Introduction **2**
- 1.1 Introduction to Concepts . . . 2
  - 1.1.1 Kaluza-Klein Theory . . . 2
  - 1.1.2 Path Integrals in Quantum Mechanics . . . 4
- 1.2 Scalar QFT . . . 5
  - 1.2.1 The Action associated with Free Fields . . . 5
  - 1.2.2 Discretising the Action for Numerical Computation . . . 6
- 1.3 Using the Action to calculate the Amplitude . . . 7
- 1.4 Distributing the Field across the Lattice . . . 8
  - 1.4.1 Importance Sampling . . . 8
  - 1.4.2 Markov Chains . . . 9

2 Code Design **11**
- 2.1 Basic Serial Code . . . 11
  - 2.1.1 main.c . . . 12
  - 2.1.2 get_args.c . . . 12
  - 2.1.3 fielddist.c . . . 12
  - 2.1.4 update.c . . . 14
  - 2.1.5 showdist.c . . . 14
- 2.2 Adaptation for Various Dimensions . . . 15
  - 2.2.1 s.c . . . 15
  - 2.2.2 correlate.c . . . 17
- 2.3 Adaptation for Interacting Fields . . . 18
- 2.4 Parallel Adaptation . . . 19
- 2.5 Kaluza-Klein Code . . . 20
- 2.6 Software Testing . . . 21

3 Profiling and Parallel Performance **23**
- 3.1 Serial Performance . . . 23
- 3.2 Parallel Performance . . . 25

4 Results **27**
- 4.1 Free Scalar Field Theory . . . 27
- 4.2 Interacting Fields . . . 29
- 4.3 Kaluza-Klein Interaction . . . 32

5 Conclusions **34**

A Derivations of Formulae **36**
- A.1 Free Field Equation . . . 36
- A.2 One Dimensional Correlation Function . . . 36

B Normal Gaussian Distribution **38**

Chapter 1

Introduction

1.1 Introduction to Concepts

1.1.1 Kaluza-Klein Theory

Kaluza-Klein Theory merges the theory of electromagnetism and the theory of gravity into one theory by introducing a fifth dimension to our world. In General Relativity, gravity is described over four dimensional space-time, as is electromagnetism in Maxwell's theory. The equations from which all motion is derived for both fundamental forces can be described in tensor form in both cases, respectively:

$$G_{\alpha\beta} = 8\pi G\phi^2 T_{\alpha\beta} \quad and \quad \partial_a F_{\alpha\beta} = J^\beta \,, \tag{1.1}$$

where $G_{\alpha\beta}$ is the Einstein Tensor, $T^{EM}_{\alpha\beta}$ is the electromagnetic energy-momentum tensor, $F_{\alpha\beta}$ is the field strength (Faraday) tensor and J^β is the four dimensional current.

These equations describe the way the fields change over space-time by relating the energy of the field to the density of its matter (Electromagnetic matter is represented by charge, and gravitational by mass.) and this requires the highest order derivatives of motion. To find how matter accelerates through a field, we apply integration to the field equations. In general relativity, this involves a double integral to find the metric tensor, but note that this metric replaces the gravitational force in Newton's theory of gravity. In other words, general relativity explains acceleration due to gravity as a consequence of the curvature of space, rather than as a force in the intuitive sense. The metric tensor describes the local curvature of space by stressing certain co-ordinates differently to others. The co-ordinates describe position, and so, giving one co-ordinate more weight than another will pull space in the direction along which the weighted co-ordinate is measured. This has the same effect as an attractive force, causing acceleration. In electromagnetism, the electromagnetic potential effects how a charged body moves. In the most basic physics explanation, force is the rate at which work (energy) is carried out in space (sort of like energy density), or alternatively, energy is the the amount of force expended over a particular space. Potential energy represents the amount of force required to change position, and so the force due to a field is given by the gradient of the potential. Force is that which causes acceleration of matter, and since Maxwell's equations involve second derivatives of electric and magnetic potential, their integral provides the nature of acceleration due to the field.

Neither of these theories contain any mention of the other, so Kaluza's idea was that if there is a link between them, they must both be derived from some higher theory. Simply adding the two tensors equations together is not a solution, since motion is described in a different way in both theories. we have just seen that acceleration is found via a double integral of Einstein's equations, whereas it involves the first integral of Maxwell's equations. Certainly if we simply wish to discover the net force resulting from both, we could then add the Maxwell equations to the equations of motion derived from the metric (in turn derived from the Einstein equations), and could subsequently find an formula describing the position of a body influenced by both forces. However, in this case the forces come from completely separate theories, despite the fact that they

interact with the same object. This is like describing two people playing tug-of-war, showing how the rope reacts to both people without explaining their common genetic origin. we are searching for a higher understanding of gravity and electromagnetism, not simply for a description of their interactive behavior. Kaluza's solution was then to create a structure which encompassed both theories separately, but which described a relationship between both, just as Maxwell managed with electricity and magnetism. In order to achieve this, since we have already noted the completeness of both four-dimensional theories, a five-dimensional structure was suggested [1]. In four dimensions, the gravitational potential (Einstein's metric) defines how space is curved, so it was suggested that distances in the fifth dimension can be similarly defined by the electromagnetic potential. The five dimensional metric is thus constructed of the original gravitational metric and the electromagnetic potential as follows:

$$(\hat{g}_{AB}) = \begin{pmatrix} g_{\alpha\beta} + \kappa^2\phi^2 A_\alpha A_\beta & \kappa\phi^2 A_\alpha \\ \kappa\phi^2 A_\beta & \phi^2 \end{pmatrix}$$

This is the new five-dimensional metric tensor, where A, B, the latin indices, include all five dimensions, and *alpha*, *beta*, the greek indices, only include rhe four dimensions of the initial field theory. The term, κ, is just a constant to scale the forces appropriately. ϕ is a scalar field (known as the dilaton) and A represents the electromagnetic potential.

The trouble is that we haven't observed such a metric in everyday life. we normally only observe the effects of three dimensions of space and one of time. Kaluza proposed that the fifth dimension was too small to be seen, in the way a hose pipe looks like a line when viewed from afar. When one moves closer, it becomes apparent that it is three-dimensional. If we imagine that the hose is actually two-dimensional when viewed close up, and that it's cylindrical shape is just a result of the second dimension being circular in shape, we are getting close to the original picture proposed concerning our fifth dimension. When we try to derive the field equations from the metric above, applying the cylinder condition to our calculations is the same as saying that motion in the fifth dimension vanishes, since the fifth dimension is so short. So all derivatives in that direction go to zero, and the resulting four dimensional field equations are [1]:

$$G_{\alpha\beta} = \frac{\kappa^2\phi^2}{2} T^E_{\alpha\beta} M - \frac{1}{\phi}[\nabla_\alpha(\partial_\beta\phi) - g_{\alpha\beta}\Box\phi]\,, \tag{1.2}$$

$$\nabla^\alpha F_{\alpha\beta} = -3\frac{\partial^\alpha\phi}{\phi}F_{\alpha\beta} \quad , \quad \Box\phi = \frac{\kappa^2\phi^3}{4}F_{\alpha\beta}F^{\alpha\beta}\,, \tag{1.3}$$

Holding the scalar field ϕ constant results in equations 1.1, and hence we have a theory which merges electromagnetism and gravity.

Although Kaluza-Klein theory originally described only the idea of representing the Maxwell and Einstein equations in one tensor, it is often liberally used as an umbrella term for any higher dimensional physical theory, including modern day string-theory. Applying the name to my project is therefore not strictly correct, but *is* useful for the purpose of understanding the nature of the fifth dimension in my simulations. Among the mass of attempts to explain our universe through higher dimensional structures is the idea of the brane. The brane is a four dimensional "membrane" that exists within a higher dimensional universe known as the bulk. The human residents of the brane do not see what happens in the greater bulk, but postulate its existence as a necessary structure to explain events in their visible four dimensional world. Thus, the bulk is similar to the fifth electromagnetic dimension proposed by Kaluza and Klein.

A picture has been developed in cosmology where only gravity propagates the fifth dimension. The idea is that the extra gravitational dimension exerts tension on the brane which alters its curvature, similar to the way one might exert tension on a two dimensional flat piece of paper. If one holds the paper on two opposite sides and moves one's hands closer together (as though they were experiencing an attractive force) the paper will have to bend. On the quantum level, the particles (quantum fields) which propagate gravity are gravitons. The picture created in this thesis is of two interacting quantum fields, where only one of them propagates the fifth dimension, and it is interesting to see the quantum effects of this coupling.

1.1.2 Path Integrals in Quantum Mechanics

The path integral in quantum mechanics is based on the undeterministic nature of particle propagation. In classical mechanics, it is always safe to assume that a body will tavell along the path which requires the least work. This path is defined by the energy of the body, any external source affecting it, and their relative positions (defined in space-time/phase space). A quantum field or particle is also dependent on such conditions, but the difference at the quantum level is that such parameters can only describe paths which have a certain probability of being followed.

Since nothing is completely determined, we cannot say with certainty that a particle will move from point A to point B. Instead it may move to point C. From this uncertainty, we can see that there are a number of paths along which a particle can move from A to B. For example, once the particle has arrived at point C, there is now a certain probability that it will move to point B from point C. Thus we have two possible paths, given three points. To find the probability that the particle will travel from A to B via C, we just use basic probability theory. ie. Multiply the probability that it goes from A to C by the probability that it moves from C to B.

Now it is pointless in such a space as that described above to ask ourselves what the probability of the particle ever existing in B is, given that it once existed at A. If we evaluate this probability over an infinite time period, the normalised probability will be 1 (100%). What we are interested in is the probability that the particle moves from one point to another within a particular length of time. The superposition principle tells us that the amplitude for the particle to reach point B within a certain time is the sum of the amplitudes of all the paths it could possibly take. Now realise that there are an infinite number of points in the region around A and B, and that adjacent points are infinitesimally close together. In such a situation we know to represent the sum as an integral. Thus the amplitude for a particle to propagate from A to B is given by integrating over all possible paths, giving us a path integral.

In order to be able to calculate this path integral, we have to be able to describe a general form for each path. Just as in classical mechanics, we should assume that the energy of the particle is important, since this describes the work done by the particle as it interacts and moves. A particle may not have much chance of reaching some points within a certain time, depending on its kinetic energy, which reduces the probability for that path. With this in mind, the most likely path will be the one which extremises the Lagrangian at each point along the path. This indicates that the action should be involved in describing the probability of any particular path.

Now, remembering that a particle propagates as a wave, according to the Schrödinger and Klein-Gordon equations, for example) the total amplitude for a particle to move from A to B in time T must be expressed as:

$$U(x_A, x_B; T) = \int Dx(t)\, e^{i(phase)}, \tag{1.4}$$

where $\int Dx(t)$ means sum over all paths.

As we said before, the phase of the most likely path will involve extremising the action, and the phase of any other path will differ in phase by some constant (as in any wave equation, phase difference is a constant), so a good guess at the solution is:

$$U(x_A, x_B; T) = \int Dx(t) e^{iS[x(t)]}, \tag{1.5}$$

where $S = \int \mathcal{L}\, dt$ is the action. This turns out to be the correct formula, since it gives the correct solution to known problems [4], p.277.

In order to use this formula, we would like to know what the action is for different paths. So how does the action depend on the path chosen by a particle? A good way to see this is to break each path up into tiny intervals, so that the path in each interval can be approximated as a straight line. ie $S = \int \mathcal{L}\, dt \to S = \sum_k l_k\, \epsilon$, where $\epsilon = \delta t$

Now that the path has been discretised, the total amplitude for each path can be given as the product of the amplitudes for each interval along the path, just as it was in the three point example, $U(A, B;\, T) = U(A, C;\, T') \times U(C, B;\, T'')$, where T' + T" = T. See, [3] for an simple explanation of the amplitude between A and B in a continuously infinite space. The amplitude $U(A, B; T)$ is

given by $\langle B | e^{iHT} | A \rangle$, so the propagator at each interval is given by $\langle q_{k+1} | e^{iH\epsilon} | q_k \rangle$, and the total amplitude is given by $\Pi_k \langle q_{k+1} | e^{iH\epsilon} | q_k \rangle$, multiplying probabilities as usual (The amplitude may be described as the square of the probability.). We can describe the Hamiltonian for each interval as $H(\frac{q_{k+1}+q_k}{2}, p_k)$ since it depends on position and momentum, and the positions can be estimated as the average over a straight line between q_k and q_{k+1} (remembering that the intervals are tiny).

Remembering from classical mechanics that the Lagrangian can be written as $\mathcal{L} = \sum_{i,j} p_i, q_j - H$, we can represent the propagator at any particular time integral as:

$$\langle q_{k+1} | e^{-i\epsilon H} | q_k \rangle = (\Pi_i \int \frac{dp_k^i}{2\pi}) e^{[i \sum_i p_k^i (q_{k+1}^i - q_k^i) - i\epsilon H(\frac{q_{k+1}+q_k}{2}, p_k)]} \tag{1.6}$$

and multiplying the amplitude for each interval together, we get

$$U(q_0, q_N; T) = (\Pi_i \int dq_k^i \int \frac{dp_k^i}{2\pi}) e^{[i \sum_k \left(\sum_i p_k^i (q_{k+1}^i - q_k^i) - i\epsilon H(\frac{q_{k+1}+q_k}{2}, p_k)\right)]} \tag{1.7}$$

For a derivation of the above formula, see [4], pp. 280, 281 in particular.

1.2 Scalar QFT

1.2.1 The Action associated with Free Fields

In the section on path integrals we gave a general form for the two point correlation function in a system of co-ordinates and momenta. In the process, we demonstrated that in order to calculate this amplitude (via the path integral method) in a specific case, one *has* to know exactly what the action looks like for that situation. The action is of course linked to the Lagrangian, which is constructed from the kinetic and potential energy of the system. My initial simulations involve a single field ϕ, which has has kinetic energy, $\frac{1}{2}\Box\phi$, and potential energy, $1/2\,\Box m^2\phi^2$, giving us the Lagrangian:

$$\mathcal{L} = \frac{1}{2}\left(\Box - M^2\right)\phi(x) \tag{1.8}$$

This Lagrangian in turn gives us the field equation:

$$\left(\Box + M^2\right)\phi(x) = 0 \tag{1.9}$$

See appendix A.1 This is simply a wave equation with mass, which is what we expect.

The Lagrangian also gives us our action:

$$S = -\frac{1}{2}\int d^4x\, \phi(x)\left(\Box + M^2\right)\phi(x) \tag{1.10}$$

Unfortunately the path integral of e^{iS} oscillates and does not converge, but *does* converge if we do the integral over imaginary time. This involves a wick rotation, which is equivalent to rotating ninety degrees into the imaginary plane. From this vantage the action becomes Euclidean, and although this is different from the action 1.10 above, the contour integral over infinity has the same value in both spaces. The wick rotation involves making the replacement $x_0 \rightarrow -ix_4$ and substituting a real field (where all dimensions are treated the same) for the original complex one. See Peskin and Schroeder section 9.3 pg 293 for a relevant example of this rotation. Here is the Euclidean action:

$$S_E[\phi] = \frac{1}{2}\int d^4x\, \phi(x)\left(-\Box_E + M^2\right)\phi(x) \tag{1.11}$$

where all values in the 4 dimensional Laplacian now have the same sign. ie:

$$\Box_E = \sum_{\mu=1}^{4} \partial_\mu \partial_\mu \tag{1.12}$$

Working in imaginary time also changes the form of the probability wave for particle propagation to $e^{-S_E[\phi]}$. The two-point correlation function (my holy grail) will be evaluated with this in mind in section 1.3.

1.2.2 Discretising the Action for Numerical Computation

Since the action 1.11 is continuous, it is impossible to use on a discrete information processor such as the computer we are using to produce simulations. The next step is to derive an equivalent action made up of discrete points. These points would be laid out on a lattice which looks so similar to the continuous surface we are examining, that there is no apparent difference. Of course, this means the points on the lattice must be so tight-knit that it looks like like a continuous surface. The test of whether the discrete action is equivalent to the continuous one is to imagine the lattice spacing getting smaller and smaller, so that as the spacing approaches zero, the discrete action *becomes* the continuous one.

To discretise the d'Alambertian ($\Box$), we use Taylor expansions, the usual method for discretizing differential equations.

$$\phi(n+\hat{\mu}) = \phi(n) + a\partial_\mu \phi|_n + \frac{a^2}{2}\partial_\mu^2 \phi|_n \tag{1.13}$$

$$\phi(n-\hat{\mu}) = \phi(n) - a\partial_\mu \phi|_n + \frac{a^2}{2}\partial_\mu^2 \phi|_n \tag{1.14}$$

The constant, a, is the lattice spacing, which is the same as $(n+\hat{\mu}) - n$ in the Taylor expansion. Adding 1.13 and 1.14 together gives the second derivative of ϕ (at a particular point, n, on the lattice) completely in terms of the values of the field at and directly surrounding that point. A simple addition and manipulation is all that is needed to give us:

$$a^2\partial_\mu \phi|_n = \phi(n+\hat{\mu}) + \phi(n-\hat{\mu}) - 2\phi(n) \tag{1.15}$$

Remembering the definition of $\Box_E$ as the sum of partial derivatives over all dimensions (1.12), we can extend this definition for a d dimensional Laplacian. Written as an operator between ϕ_n and $\phi_{n'}$, and taking the a^2 term out, it looks like this:

$$\Box_{n,n'} = \sum_\mu \left(\delta_{n+\hat{\mu}} + \delta_{n-\hat{\mu}} - 2\delta_{n,n'}\right) \tag{1.16}$$

where μ represents any dimensional direction and δ is the Dirac delta function.

We are almost ready to decide on the form of our discrete action, but there remains one problem. Because we are using a lattice to approximate a continuous surface, the choice of lattice spacing is arbitrary, so long as it is small enough to represent the surface accurately. we want the results to be independent of the lattice spacing, which means that we have to deal with dimensionless quantities only. In other words, if the quantities are dimensionless, they don't depend on a length scale such as lattice spacing. What we are looking for is some sort of discrete sum in place of a continuous integral, and the sum must be made up purely of dimensionless parts. A possible solution would therefore be:

$$\frac{1}{2}\int d^4x\, \phi(x)\left(-\Box_E + M^2\right)\phi(x) \rightarrow \frac{1}{2}\sum_{n\,n'} \hat{\phi}_n \left(-\frac{1}{\mu}\Box_{n\,n'} + \mu\delta_{n\,n'}\right)\hat{\phi}_{n'} \tag{1.17}$$

Here, μ is some dimensionless co-efficient that we anticipate may enter our equations. The arrangement of co-efficients is slightly arbitrary, so we have chosen one which involves the using the same co-efficient on the kinetic ($\Box$) element as with the potential (δ) element of the equation. From 1.18 below we can see that the constant multiplying the potential term is a^2m^2 times the kinetic constant, so we can try $\mu = am$. This works perfectly because we know that mass has dimensions of inverse length, meaning that $\mu = am$ is a dimensionless quantity.

We start discretising by changing the continuous d'Alambertian for the lattice one (1.16) in formula 1.11. Note that since we took the $1/a^2$ term out of this operator, $\Box$, it must appear in the integral. Taking $1/a^2$ outside the entire integral, rather than just the d'Alembertian, means we must cancel it out in the potential part by multiplying the mass by a^2.

$$S_{lattice} = \frac{1}{2a^2}\int d^4x\, \phi_n\left(-\Box_{n,n'} + a^2m^2\delta_{n\,n'}\right)\phi_{n'} \tag{1.18}$$

$$= \frac{a^d}{2a^2} \sum_{n\,n'} \phi_n(-\Box_{n\,n'} + a^2 m^2 \delta_{n\,n'})\phi_{n'} \rightarrow \frac{1}{2} \sum_{n\,n'} \hat{\phi}_n \left(-\frac{1}{\mu}\Box_{n\,n'} + \mu\delta_{n\,n'}\right) \hat{\phi}_{n'} \qquad (1.19)$$

The a^d term is the result of $\int d^d x \rightarrow a^d \sum$ (d is the number of dimensions of the field). As we know, lengths ($d = 1$) are measured in meters (Standard International), areas ($d = 2$) are measured in meters squared (as a result of summing over lines), volumes ($d = 3$) in meters cubed (as a result of summing over 2 dimensional objects). The unit of length on the lattice is not the meter but a, and dimensionality scales equivalently in units of a. we have equated the discrete action developed from the continuous integral with the action we actually want (1.17), and now, rearranging factors of a and m, substituting $mu = am$ into the left hand side of 1.19 and canceling common terms on both sides gives:

$$\begin{aligned} a^{d-1} m\phi_n \left(-\frac{1}{\mu}\Box_{n\,n'} + \mu\delta_{n\,n'}\right) \phi' \quad &\rightarrow \quad \hat{\phi}_n \left(-\frac{1}{\mu}\Box_{n\,n'} + \mu\delta_{n\,n'}\right) \phi_{n'} \\ \Rightarrow a^{d-2}\mu\phi^2 \quad &\rightarrow \quad \hat{\phi} \\ \phi(x) \rightarrow a^{1-\frac{d}{2}} \mu^{-1} \hat{\phi}(na) & \qquad (1.20) \end{aligned}$$

Hence the dimensions of the field cancel out the dimensions of space and energy, leaving the action dimensionless as required. When calculating the correlation function later, we will need to invert 1.20 in order to get back the true function.

1.3 Using the Action to calculate the Amplitude

We saw at the start of 1.2.1 that we need the action in order to calculate the two-point correlation function, but we didn't state what the correlation function actually looks like for general fields. Peskin and Schroeder derive it in a section entitled “Correlation Functions”[4]. The function below looks slightly different because we are working in Euclidean space, as mentioned at the end of section 1.2.1.

$$< \phi(x_1)\phi(x_2) >= \frac{\int D\phi\,(\phi(x_1)\phi(x_2))e^{-S_E[\phi]}}{\int D\phi\, e^{-S_E[\phi]}} \qquad (1.21)$$

Peskin and Schroeder mentions, “For higher correlation functions, just insert additional factors of ϕ on both sides.”[4], but we are dealing with just two points here. Remember from the introduction to path integrals that the probability of getting from one point to another is a product of probabilities.

It is interesting, and useful, to examine the form of 1.21 to understand what it is doing. It is simply a statistical formula for calculating the weighted average (which is also the most probable) value of the product of two field values at x_1 and x_2. The weight associated with each product is the value $e^{-S_E[\phi]}$. The integrals are simply continuous sums, so the formula reads as the sum of the weighted products divided by the number of products (ie. the sum over all paths, see next paragraph), which is a weighted average. In basic statistics, a formula which gives a weight to each value in an ensemble describes a probability distribution. If we simply distribute all the points of our field according to this field distribution, then the product $\phi(x_1)\phi(x_2)$ is automatically weighted, and our correlation function may then be calculated by this simple formula!

Notice that although we are dealing with quantum field theory, we have developed statistical mechanics to describe it. Peskin and Schroeder describe this analogy in section 9.3. Remember the definition given under formula 1.4 of $\int Dx$ as, “the sum over all paths”. The denominator in formula 1.21 has the term e^{-S} added to this integral, which therefore defines the denominator as, “the sum over all paths distributed exponentially according to the action”. This is a loose definition, but we are developing a picture of what is going on. The number of paths between points on the field is, as indicated in the previous paragraph, the same as the number of of products $\phi(x_1)\phi(x_2)$, since there is just one path for each product (each product involves only two points, which allows for only one commutative combination, or path). Now using the analogy with statistical mechanics, remember the description of the partition function, Z, as “the sum over all states”. The states in the partition function of statistical mechanics are distributed according to $e^{-\beta E}$, where E is the

energy, and β is the inverse temperature, but notice that our paths are also states in an ensemble of possible amplitudes. So what we now have is that $<\phi(x_2)\phi(x_1)>= Z^{-1}\int D\phi\,\phi(x_1)\phi(x_2)e^{-S}$. In other words, Z is a Greens Function for our correlation function.

To represent an infinite continuum of paths, we create many random field distributions on a dense lattice and measure on each field the product $\phi(x_1)\phi(x_2)$ at a number of random points. we don't want to measure every possible product on each field, simply because it takes too much time. If each field is sufficiently different (though each must have the same distribution, e^{-S}), we can average over a much more diverse ensemble. This means we will get a good average quicker if we concentrate on measuring a fair ensemble of points on a good ensemble of fields. This is related to the Central Limit Theorum, which is also used in Markov Chain Monte-Carlo (MCMC) simulations. In fact, since MCMC involves applying random changes to a matrix (which could be a lattice) so that it evolves to a fixed solution, it is used in many problems of statistical mechanics. Since we have reduced our quantum problem to a statistical one, an MCMC method is a natural choice for us. we have chosen the Metropolis Algorithm in particular to give us a field distributed according to $e^{-S_E[\phi]}$, from which we can calculate the correlation function by randomly choosing n points x_1 and x_2 a distance na apart (where a is the lattice spacing) and applying the following formula:

$$<\phi(x_2)\phi(x_1)>_{lattice}= \frac{1}{n}\sum_{n}\phi(x_1)\phi(x_2) \tag{1.22}$$

Note that n replaces Z as the sum of products. But n is obviously the sum of products as we have changed from an infinite integral to a finite sum of n products, chosen randomly on the lattice.

1.4 Distributing the Field across the Lattice

The Metropolis Algorithm is based on two main concepts, Markov Chains and Importance Sampling.

1.4.1 Importance Sampling

Importance Sampling is a way of solving integrals by filling the volume of integration with a large number of points. The idea is that the average value of these points will represent the center of the volume (which is statistically valid). Each point is generated by taking an educated guess, and then testing to see whether or not this guess falls inside the volume. The guess is "educated" by generating points within a similar volume. The similar volume would be defined by a simpler random distribution, so that points can be easily distributed throughout that volume. The only other necessity is that it cannot be smaller than the volume we are trying to solve, since we don't want any areas of the solution to remain unfilled by points. Obviously that would lead to a false average. The closer the approximate volume is to the actual one, the less guesses will be rejected.

Consider how one would generate points randomly inside the approximating volume. For simplicity, we will deal in two dimensions, so that the volume is actually an area. Let $y = g(x)$ represent the curve under which the approximating area of integration lies. Also, let $u(x)$ be a random number, uniformly distributed between 0 and 1. Then $u(x)g(x)$ will give us a random point, $(x, u(x)g(x))$ which lies under the curve, and therefore inside the approximating area. Now, in order for this point to lie inside the actual area of integration, which lies under the curve $f(x)$, we say that $u(x)g(x) \leq f(x)$. Another way to say this is:

$$u(x) \leq \frac{f(x)}{g(x)} \tag{1.23}$$

I referenced the Central Limit Theorum at the end of section 1.3, and this is the moment to really see how Importance Sampling is tailor-made for solving our Correlation Function in it's statistical form. In order to produce a good average result, the algorithm just described must be run a number of times, so that we obtain an average of averages. Each time, we fill the area under the curve with points, find the average value, add it to the sum of averages, and finally divide that sum by the number of times the algorithm is run. This is almost exactly what we said we would do at the end

of section 1.3. Importance Sampling is incomplete without this cycle. In our case, we want to find the average product of points, not the average point, but this just means we calculate $\phi(x_1)\phi(x_2)$ each time, instead of just $\phi(x)$. Once we have generated enough points, we can use Importance Sampling to give us the average value of any function of the points, which could be represented mathematically as:

$$\frac{\int f(x)\, dx}{\int dx} \tag{1.24}$$

Notice the similarity with formula 1.21. In order for 1.24 to be equivalent to 1.21 in the same way that 1.22 is, it must be measured over a lattice of points distributed according to $e^{-S_E(\phi)}$. This is where Markov Chains are relevant.

1.4.2 Markov Chains

A Markov Chain consists of chain of states, where the state of each link in the chain is found by multiplying the state of the previous state by some operator. To be specific, each link has a certain probability of being in each possible state, and this probability is found by multiplying the states of a previous link by a probability matrix. This is still a little abstract, so we will use an example that is particularly relevant.

Let each link in the chain represent a specific day I spend on my thesis, so that the first link is today, the next represents the day afterwards, and so on. Now each link in the chain has two possibilities. Either the project is finished and handed up, or it is not. Now if the project is finished one day, it will be finished the next day too. Let's assume for simplicity's sake that if the project is not finished today, there is still a 10% chance it will be finished tomorrow, and this rule holds every day. Today, the project is not finished, so todays state may be represented as $(1, 0)^T$, where the first row represents an unfinished state, and the second, finished. The probability I am finished tomorrow is then 10%, and the probability that I am still not finished is 90%, giving the next day, or link, the following state: $(0.9, 0.1)^T$. Now if the 10% chance of finishing works out, this means I will be finished the next day, so means I automatically have a 10% chance of finishing on day 3, based on row 2 of day 2. However, it is more likely that the 90% chance of not being finished on day 2 is what actually happens. But there is a 10% of that 90% possibility that I will be finished on day 3, and this adds 9% to the 10% chance I already have of finishing on day 3. This means that row 2 of day 3 has the value, 0.19. This sounds much simpler if I just say: To find row 2 of tomorrow's state, multiply today's row 1 by 0.1, today's row 2 by 1, and add. And to find row 1 of tomorrow's state, multiply row 1 by 0.9, and row 2 by 0 (since if I am finished today there is a 0% chance of finishing tomorrow, and if I am not finished, there is a 90% chance I will remain unfinished tomorrow). This is described in matrix notation as:

$$\begin{pmatrix} 0.9 & 0 \\ 0.1 & 1 \end{pmatrix} = \begin{pmatrix} \phi_1 \\ \phi_2 \end{pmatrix},$$

where ϕ_1 represents an unfinished state and ϕ_2 represents a finished state.

With the chain we have described above, there is always a tiny possibility that I will not be finished, although this gets smaller and smaller with time. However, there are many Markov Chains that reach some fixed equilibrium state, and these are known as "ergodic". In other words, the probable states of each link in the chain remains the same after a certain number of links are passed, no matter where we start from. These are the chains used in the Metropolis Algorithm. Chains which reach a fixed state can be constructed using something called "detailed balance", which we will not explain here, for fear of getting too sidetracked. Not all chains which have fixed points have to obey detailed balance, but the reverse statement is true: All chains obeying detailed balance have fixed points. we know by this stage that we want our lattice to evolve to a fixed distribution, $e^{-S_E(\phi)}$, and so we want an algorithm to generate this state. The Metropolis algorithm obeys detailed balance, and deals with our problem nicely, as we will see from it's description below. In our case, the state of the system is represented by an n-dimensional lattice, rather than the vector in my simple example above, and the probabilities become field values on the lattice. The original state of the field can be

given any random values, since the fixed state is reached regardless of where we start from. Now we find each consecutive state as follows:

Metropolis Algorithm:

1. Propose a small change to the field, $\phi \rightarrow \phi'$ (The Metropolis Algorithm stipulates that the change must be area-preserving). This change can be reversed by applying the conjugate of the change we made (The algorithm also demands that the proposal must be reversible).
2. Accept the probability based on the probability:

$$\mathcal{P}_{acceptance} = min\left[1, \frac{\pi(\phi')}{\pi(\phi)}\right], \tag{1.25}$$

where $\pi(\phi)$ is the state of the field.

Otherwise, keep the old field.

3. Repeat until the field achieves a fixed state.

The term $\pi(\phi')/\pi(\phi)$ in step 2 of the Metropolis Algorithm is actually just Importance Sampling, since to accept the new field with this probability, we compare it with a uniform random variate, $u(x)$. If $u(x)$ falls below $\pi(\phi')/\pi(\phi)$, the field is accepted, otherwise not. This is just equation 1.23. Our lattice represents a volume of n dimensions, and we are simply filling it with points in the way mentioned in subsection 1.4.1 until, because of the detailed balance introduced by equation 1.25, it reaches a fixed state. Bringing all the maths together we have:

$$\frac{\pi(\phi')}{\pi(\phi)} = \frac{f(x)}{g(x)} = \frac{e^{-S(\phi')}}{e^{-S(\phi)}} = e^{S(\phi)-S(\phi')} \tag{1.26}$$

Substituting this into 1.25 generates the distribution we want, and since our algorithm is an importance sampling one, we can calculate equation 1.22.

In conclusion, we run the Metropolis Algorithm above, using 1.26 as the probability of accepting an update. When after a number of iterations, this achieves the desired distribution, we measure 1.22 over a number of points. To get a better result, we run the Metropolis Algorithm again to give a different field with the same fixed distribution, and measure 1.22 again in the same way. This is done again and again to achieve a large ensemble of fields, to produce a good average by the Central Limit Theorum.

Chapter 2

Code Design

Common sense tells us to design our code in a patient, methodical way, not jumping in at the deep end, but rather testing our abilities first on shallower challenges. With this in mind, we begin writing code for a non-interacting one dimensional field with only potential energy, and add the kinetic term afterward. We repeat this method for two, three four and five dimensions, then write parallel free field code and simultaneously, serial code for interacting fields. Finally we bring these latter two together to form the Kaluza-Klein simulation, having tested my code before moving on at every stage. Since the parallel code is actually serial code running in parallel on a number of processors, it makes sense to follow the same order in describing the code as we do in writing it.

2.1 Basic Serial Code

The skeleton of my code is comprised of the few simple steps of the Metropolis Algorithm (1.25), with extra steps for representing the data.

1. Initialise variables, such as lattice spacing, an amount by which to alter the field at each metropolis update step (1.25), the sizes for the various loops involved in later calculations and the number of points in the field (command line argument). Also calculate any other common variables which depend on any of the above.

2. Allocate space to hold points, and fill this space with random values. In our case, the points represent the value of the field at some position on a lattice. The positions of the points are unrelated to the value of the field at this stage however, since they hold randomly generated values. Hence this first step is nothing more than a simple “for” loop to give random values to some memory allocated using the malloc command.

3. Distribute the points throughout some volume. In our case this is the volume described by the distribution, e^{-S}. This step has a higher complexity. It involves a loop which runs step 2 of the Metropolis Algorithm (1.25) as many times as necessary before the required field is produced.

4. Analyse the field to generate a graph of it’s distribution. This step tells us if step 2 has been successful.

5. The most complex step, using the bulk of processor time, this involves calculating equation 1.22. We measure $\phi(x_1)\phi(x_2)$ for a number of distances $r = x_2 - x_1$, picking a number of positions x_1 at random, and choosing random directions to look for x_2, a distance r away. We find the average of these points for each distance r. We then update the field until it is significantly different, although the distribution obviously remains the same. Then we perform the same measurements as before. We do this a good number of times, and compute the average result from each time we measured the field.

6. We write the data calculated in step four to a file, with each line containing the displacement r in units of lattice spacing a (this gives us integer values even if the actual distance is double precision), and the average value $\phi(x_1)\phi(x_2)$ at this separation.

2.1.1 main.c

My main file contains separate functions to perform each of the above steps, and proceeds as follows: *get_args*() (step 1 command line arguments. The rest of step 1 is handled directly within the main function.), *unidist*() (step 2), *fielddist*() (step 3), *showdist*() (step 4), *correlate*() (steps 5 and 6).

2.1.2 get_args.c

This function simply exists to make the main file look tidier by reading the command line arguments and checking for input errors, such as a negative or zero number of points, or a wrong number of arguments. It also contains a small function, *printerr*(), which prints information on which arguments to input, if an error is caught. This tiny function exists just to keep get_args.c tidy!

2.1.3 fielddist.c

Since we have created a seperate file (update.c) to actually run the operation given by equation 1.25, you might expect this piece of code to consist of a single "for" loop that runs *update*() until the required field distribution is produced. However, we have included an optimisation here which makes fielddist.c much longer. In fact, the bulk of this code involves the optimisation step, although the step is actually only performed a small number of times throughout the loop (If it were run a lot, it would add to the complexity of the loop and hence slow the program down significantly.). The optimisation involves picking the best amount by which to change the field. I explain it below.

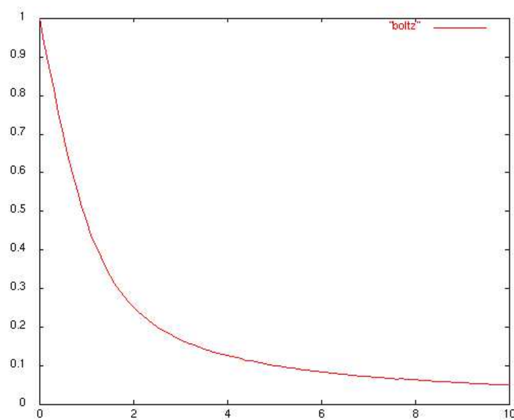

Figure 2.1: Probability of Acceptance vs. Step Size.

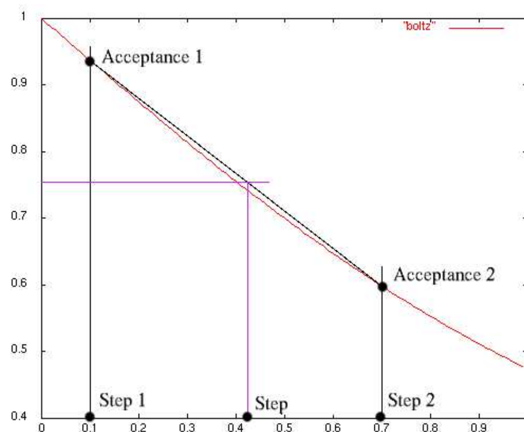

Figure 2.2: Search Algorithm

Figure 2.1 was created by running a simple version of my program using different step sizes. By step size, we mean the amount by which we alter the field in step 1 of the Metropolis Algorithm (subsection 1.4.2). See the next subsection (2.1.4) to see exactly what the change looks like in my code. For each step size, we record the number of times the metropolis step (equation 1.25) was accepted, and this value is represented along the vertical axis. So we see that the larger the step size, the lower the probability that it will be accepted. Now if the step size is zero, the field is not altered at all, and hence the field is always accepted (with probability 1, since $e^{S(\phi)-S(\phi')} = e^{(0-0)} = 1$ and $min[1, 1] = 1$ in 1.25.) Increasing the step size increases the absolute value of $S(\phi) - S(\phi')$, which decreases the probability of acceptance for negative changes to the action. As we can see from the graph, the probability trails off to zero as the step size increases to infinity. Actually, this trend is similar to the Boltzmann tail in statistical mechanics, though the graph of acceptance for my program decreases more rapidly from it's peak at 1.

We can see from this that if the step size is too small, the field will not change significantly each iteration of the metropolis algorithm, although it will almost definitely change every time. On the other hand, if the step size is too large, the field will not change at all most iterations, although

once in a while it will change significantly. Both of these cases would require a huge number of iterations to converge to a solution, which would result in a very slow program. It turns out that the acceptance rate that achieves the fastest convergence is approximately 75% of the time.

With this in mind, we write a search algorithm that very quickly finds the step size required for an acceptance very close to 75%. Figure 2.1 was generated quickly, but with some inconvenience, as we needed to creat a new program to produce it, and we would need to alter this program significantly if we were to find the 75% step size for a different simulation. Also, finding the 75% point this way involves checking every possible step size, whereas the following search involves trying out just a few. In figure 2.2 we magnify the area of the graph (the curved line) which shows 75% acceptance on the vertical axis. We will use this graph to visualise the search algorithm. First we pick a “left” step size, step 1, and a “right” step size, step 2, that we believe result in acceptance rates, acceptance 1 and acceptance 2, lower and greater than 75%, respectively. We run the metropolis algorithm for a number of iterations large enough to produce an accurate average, using the “left” step size. We then produce an average acceptance in the same way using the “right” step size, and apply the following code to produce a new guess at the step size:

1. Find the slope, m.

$$\mathbf{m = (\ acceptance2\ -\ acceptance1\ )/(\ step2\ -\ step1\ );}$$

2. Use this slope to guess where 0.75 is, by changing the unknown quantity in the above equation from m to step2, and letting $acceptance2 = 0.75$.

$$\mathbf{step = step1 + (\ 0.75\ -\ acceptance1\ )/m;} \tag{2.1}$$

Notice that while my program was running the metropolis algorithm in order to discover the acceptance rate, it was also updating the field, so no time is being wasted. Calculating the acceptance as we go along does use up a small portion of time, but once we have found a good step size, there will be no need to keep measuring the acceptance rate. This means that at a later stage, $correlate()$, where most of the time is used up, will not be slowed down, and will use the most efficient step size. Of course, my $update()$ function, which is used in $correlate()$ returns a value to say whether or not the change was accepted, but this is a minor operation, and the compiler may possibly notice that it is redundant when correlate is working away.

Using the new step, $fielddist()$ calculates a new acceptance rate, all the while updating the field. It decides whether the new step was a “left” guess, step1, or a “right” guess, step2, in the following code:

```
if( acceptance > 0.75 ){
  acceptance1 = acceptance;
  step1 = step;
},
```

and the same for $acceptance2$, if $acceptance \leq 0.75$. We are using the knowledge that the acceptance rate decreases with step size in figure 2.1. This code is run every so often throughout the loop, by running the optimisation step once in an outer loop, and the update step a large number of times within each iteration of the outer loop. This is more efficient than using a conditional. We hard code the value 10 for number of times the acceptance is measured before the initial field distribution is achieved. This works perfectly, the acceptance converging to about 75% within two or three tenths of the time it takes $fielddist()$ to run.

There is one problem with this method. It is unstable if the initial two guesses are to the right of the optimal step size, as the code (2.1) which finds the next step involves adding a positive value to an old guess. In theory this can be solved easily by checking in the code to see if this is the case, and using a similar formula which finds the next guess to the left, rather than to the right. However, we have implemented a quick fix solution by printing an error message which requests lower step sizes, and ending the program. We do this because if we pick two guesses in the far right of figure 2.1, the slope is so shallow that a value of 0.75 is guessed as resulting from a high negative step size. This

means that finding guesses to the left, when the initial guesses do not border the root at $y = 0.75$, also leads to an instability in the algorithm.

Lastly, *fielddist*() returns the average step size so that *correlate*() knows the best value to change the field by, without having to redo the work done here.

2.1.4 update.c

This runs a single metropolis step. The change made to the field involves altering one randomly chosen lattice point by a random amount. Changing one point at a time rather than the whole field means that the size of the change varies much less, which makes the acceptance rate more constant. Also, there is only one degree of freedom in changing one point, whereas there are as many degrees as lattice points if we change the entire field. In other words, the chances of finding a better configuration quickly is much grater if we make the change point by point. The maximum size of this random amount is controlled in fielddist.c, where we have referred to it above as the step size, and in the following code, step.

```
phi_ new = phi_ old + step*(drand48() - 0.5);
```

In this way the change can be either positive or negative, since drand48() is a uniform random variate (generated from a linear congruential sequence) between 0 and 1. The action of each is calculated by another function, *s*() (see section 2.2.1), and equation 1.25 is calculated as follows:

```
if( new < old || exp( old - new ) > drand48() ){
  (*phi)[n] = phi_new;
},
```

where new was the action calculated from *phi_new*, and old from *phi_old*. The conditional, $new < old$ stops exp(old-new) from going above 1 since the exponential of a positive number is always greater than zero, thus fulfilling all requirements of equation 1.25.

2.1.5 showdist.c

The more important method in this file is outlined in appendix B, so we will not go into it here, but we do need to show it's conversion to computer language. In applying it in computer code, we need to know the range that the field values fall into, so we run through the loop to find the highest absolute value of the field, and pick a range slightly higher than this value. As we run through this range, we also sum the field values in order to find their average. We use this average to calculate the standard deviation of the field. we multiply the range by 2 to find the length that my bins cover, and divide by the number of bins to find the width of each bin. Having set the value of each bin to zero, we run the following code:

```
for( i = 0; i < M; i++ ){
   bin = (int)((phi[i]+range)*n_bins/spread);
   bins[bin] = bins[bin] + 1;
},
```

where M is the number of field points, *bin* is the index in my array of *bins*[ ], *phi*[ ] is the array of field points, n_bins is the number of bins, and *spread* is the total length covered by the bins.

The logic behind the first line inside the "for" loop needs to be unravelled. Since *bin* must be a positive quantity, and $-range < phi[i] < +range$, we needed to add *range* to *phi*[*i*] to make it positive, giving us $0 < phi[i] + range < spread$. This quantity is normalised by dividing by spread, resulting in $0 < (phi[i]+range)/spread < 1$, and so, to find which bin holds the field value, *phi*[*i*], we multiply by the total number of bins, giving us $0 < (phi[i] + range) * n_bins/spread < n_bins$. Note that in my code, we round everything down by converting to type *int*, and so we do not run into memory problems by trying to access *bins*[*n_bins*], which does not exist. That *spread* and *range* are obviously related indicates that *spread* may be an unnecessary variable, but it is used several

times throughout showdist.c, and so it saves making the same calculation more than once. In fact, another optimisation we can make is to declare the variable, $bin_width = spread/n_bins$, which saves more processor time. We could also replace $n_bins/spread$ by dividing by bin_width in the above code.

In creating the file containing the distribution, we simply use the inverse function of the formula described in the earlier piece of code:

((double)bin)*bin_width - range + bin_width/2

, where we have added the term $bin_width/2$ in order to represent the average field value held by the bin. Leaving this term out leads to a distribution shifted slightly to the left. Against this, we plot the following values on the vertical axis.

(double)bins[bin]/(bin_width*M);

These are the normalised fractions of the field which fall into the bin labeled by *bin*, and we have explained this normalisation in appendix B, where you can also see the plotted distributions.

One other result that this file, showdist.c, produces when executed is an analytical plot of the distribution. Sometimes there are not enough points to produce a good plot, and so it is good to see if the points generally follow a Gaussian distribution. we simply use the variance we calculated earlier (in the process of evaluating the standard deviation) to produce a graph given by the formula $\frac{1}{\sqrt{2\pi}}e^{x^2/(2\sigma^2)}$, where $sigma^2$ is the variance. Then we check to see if the points are generally distributed around this curve.

2.2 Adaptation for Various Dimensions

I have left the description of s.c and correlate.c to this section, although they are necessary parts of any basic serial code. In changing the code to solve different correlation functions, these were the only two functions to change significantly.

2.2.1 s.c

From section 1.2.2, equation in particular, we see that the action varies with dimension for a free field, so that s.c, which calculates the discrete lattice action, must change if we run simulations over various dimensions. Of course, since the action for interacting fields involve a sum of the actions for free fields, the dimensionality also changes the action for interacting theory. Keeping dimensions the same, the action is still different when moving from free field simulations to interacting ones, so we see that my function, s.c, has been changed many times throughout the course of my work.

Substituting 1.16 into 1.17 gives us

$$S_{lattice} = \frac{1}{2}\left(\mu\phi_n^2 + \frac{1}{\mu}\sum_{n\,n\pm1}(2\hat{\phi}_n^2 - \hat{\phi}_n\hat{\phi}_{n+1} - \hat{\phi}_n\hat{\phi}_{n-1})\right) \tag{2.2}$$

In an attempt to simplify matters, we can change the above equation to:

$$S_{lattice} = \frac{\mu}{2}\phi_n^2 + \frac{1}{2\mu}\sum_{n\,n\pm1}\left((\phi_n - \phi_{n+1})^2 + (\phi_n - \phi_{n+1})^2\right), \tag{2.3}$$

where μ is the dimensionless quantity am, since a has dimensions of length, and m inverse length. Since we simply change one field point at a time, the difference between the old action and the new in the update step cancels out the sum in the above equation everywhere except at the updated point. The one dimensional s.c function simply consists of:

phi_minus = (n == 0) ? phi[M-1] : phi[n-1];
phi_plus = (n == M-1) ? phi[0] : phi[n+1];

```
return ((phi_n-phi_plus)*(phi_n-phi_plus) +
(phi_n-phi_minus)*(phi_n-phi_minus))/(2*mu) + (mu/2)*phi_n*phi_n;
```

Increasing dimensions simply means that we must find *phi_minus* and *phi_plus* in a second dimension and add another value computed from 2.3 to the version of s.c above. This means that we must store two dimensional field values. We chose to continue using a one dimensional array to store values, which meant that to represent two dimensions, we had to give each value in this array two co-ordinates. we decided that to represent an $N \times N$ lattice, we would store the first N values in the array as $(x, 0)$, where x runs from 0 to $N-1$, and the second N values as $(x, 1)$, and so on up to $(x, N-1)$, so that we have two degrees of freedom (x, y),both running from 0 to $N-1$. We don't actually store a set of co-ordinates in a struct. Rather we use the logic that in order to increase the y co-ordinate by 1, we must add N to the current position in my one dimensional array, and to increase x we add 1, as in the one-dimensional code above.

Note that we am using periodic boundary conditions in the above code, since points at the end of the array neighbour points at the start. In two dimensions though, we want points at the end of a *row* to neighbour points at the start of the row in the x direction, and we want the points at the end of a column to neighbour points at the start of the column in the y direction. This was implemented as:

```
phi_minus = (n%L == 0 ) ? phi[n+L-1] : phi[n-1];
phi_plus = ((n+1)%L == 0) ? phi[n-L+1] : phi[n+1];
```

for neighbours in the x direction, and

```
phi_minus = (n < L) ? phi[n+M-L] : phi[n-L];
phi_plus = (n >= M-L) ? phi[n-M+L] : phi[n+L];
```

for neighbours in the y direction. To find an x coordinate we get the remainder of n/L and to find the y coordinate we get the factor of n/L, where n is the index of the array and L is the number of points in each dimension. For example, in a 10×10 lattice, the x coordinates run from 0 to 9 in increments of 1, whereas the array index runs from 0 to 99 (in increments of 1). The y coordinate also runs from 0 to 9, but is incremented in steps of 10 on this lattice, so that in translating from array index to y-coordinate, we divide by 10, and discard the remainder. Picking an array index at random, 68, we get $(x, y) = (8, 6)$, since the factor represents y and the remainder represents x ($68/10 = 6$ remainder 8). This gives us the required coordinate's where $(1, y) = y1$ (eg. $(1, 5) = 51$) and so on, giving $(x, y) = yx$ (eg. $(8, 6) = 68$). we have described the system for a decimal 10×10 lattice, but the same logic applies for any $N \times N$ matrix, using base N aritmatic. So we see that on the left boundary $n\%L = 0$, and on the right boundary $(n+1)\%L = 0$, since the next array index is the first index of a new row. We simply insert *phi_minus* and *phi_plus* into the same formula (the one returned by the 1D code) for each dimension and add. Extending this logic to higher dimensions is now simple. For example, here are some fragments of the four dimensional code. The x dimension is the same as in two dimensions, the y dimension looks like this:

```
phi_minus = (n%L2 < L) ? phi[n+L2-L] : phi[n-L];
phi_plus = (n%L2 >= L2-L) ? phi[n-L2+L] : phi[n+L];,
```

the z dimension is as follows:

```
phi_minus = (n%L3 < L2) ? phi[n+L3-L2] : phi[n-L2];
phi_plus = (n%L3 >= L3-L2) ? phi[n-L3+L2] : phi[n+L2];
```

and the fourth (t) dimension in 4D is similar to the second (y) dimension in 2D:

```
phi_minus = (n < L3) ? phi[n+M-L3] : phi[n-L3];
phi_plus = (n >= M-L3) ? phi[n-M+L3] : phi[n+L3];
```

2.2.2 correlate.c

Making measurements in one dimension was straightforward, but extra dimensions lead to more degrees of freedom, which meant that measurements could be taken in many directions. This complicated the way we chose random points with fixed separation, and also meant we needed a coordinate system to find the relative position of the two points. However, the solution to the latter problem has already been introduced in the previous section, s.c.

The basic steps involved in correlate.c was described in section 2.1. The first part consists of a loop to firstly run *update*() until the field changes significantly (we simply use trial and error and a little intuition to decide how many iterations the loop runs for.), and secondly to measure the correlation function across the field. The second part involves recording my average measurements in a file. In the one dimensional version of the code, the measurements are done as follows:

```
amplitude = 0;
for( n = 0; n < M/2; n++ ){
   amplitude += phi[n]*phi[(n+displacement)%M];
}
amp[displacement-1] = amp[displacement-1] + amplitude;
```

The array, *amp*[ ], contains the amplitude at various separations. The index of this array grows proportionally to separation of points on the field. Note that in consistency with s.c, periodic boundary conditions are imposed via the $\%M$ in the code. The above code assumes that the lattice is small (There is no point in creating a large 1D lattice since the correlation function falls off quickly, and the lattice spacing does not need to be incredibly small.) but if the lattice were bigger, we would pick a number of random points, n, and use the same encoded formula at each of these points. There is no need to show this code for one-dimension, as the two dimensional code uses such a method also, although with added complications:

```
for( displacement = 1; displacement ¡ d_max; displacement++ ){

   amplitude = 0;
   for( j = 0; j < measurements2; j++ ){
      n = drand48()*M;
      dx = drand48()*(displacement+1);
      dy = displacement-dx;
      if( drand48() < 0.5 ) dy = -dy;
      m = (n/L)*L + (n + dx)%L;
      m = (m + M + dy*L)%M;

      amplitude += phi[n]*phi[m];
   }
   amp[displacement-1] = amp[displacement-1] + amplitude;
}
```

As we can see, we have divided the displacement randomly into two directions, x and y, with two simple steps. Then, to find the point in each direction, we use the coordinate system described in s.c. To move a distance dx, without moving in the y direction, we divide by the length of the x dimension, L, to find where we are in y-coordinates (Deviding a type int number by another int automatically rounds down in c.) and multiply back by L to get to the first position in the row containing the current y-coordinate. The current random point is positioned at $(n/L) * L + n\%L$, since $n\%L$ gives us the x-co-ordinate, and n/L gives us the y-coordinate, and each y-coordinate has L associated x-co-ordinates, which means it is incremented in steps of L. Imposing periodic boundary conditions on the current row of L x positions gives the final equation above. we have allowed for the possibility for y to be negative in order to cover all directions, so to avoid problems in using the % operator, we have added $M = L \times L$, the size of the matrix, to the sum: $m = (m + dy * L)\%M$.

Picking random directions in four dimensions was less simple. In the two dimensional case, the dx and dy directions are related by, $dx + dy = d$, where d is the relative displacement of points

n and m. The code which we first implemented in four dimensions involved breaking d into two directions, dx and dy, and further breaking dx into two directions dx and dz, and dy into dy and dz. This quickly resulted in four random directions, but left a problem that dz was dependant on dx and dt was dependant on dy. For example, if dx was 0, dz was automatically 0, which meant that we wasn't sampling properly. We tried switching dz with dt at random, and various similar random switches, but this just meant that the dependencies were more random. In the end, we had to resort to the following loop, which thankfully turned out to make less difference to the performance of my code than we expected.

```
dx = 0; dy = 0; dz = 0; dt = 0;
for( m = 0; m < displacement; m++ ){
   cnst = drand48();
   if( cnst < 0.25 ) dx++;
   else if( cnst < 0.5 ) dy++;
   else if( cnst < 0.75 ) dz++;
   else dt++;
}
```

I then chose randomly to sample in positive or negative directions in each dimension. To move in each direction, we used usual modular arithmetic.

```
m = (n/L)*L + ( n + L + dx )%L;
m = (m/L2)*L2 + ( m + L2 + dy*L )%L2;
m = (m/L3)*L3 + ( m + L3 + dz*L2 )%L3;
m = ( m + M + dt*L3 )%M;
```

2.3 Adaptation for Interacting Fields

The first type of interaction we simulated was ϕ^4 interaction, the simplest possible case. This is based on the Lagrangian,

$$\mathcal{L} = \frac{1}{2}(\partial_\mu \phi)^2 - \frac{1}{2}m^2\phi^2 - \frac{\lambda}{4!}\phi^4, \tag{2.4}$$

which is simply the Lagrangian of the free field theory plus an interaction term containing ϕ^4. A wick rotation turns the difference into a sum, and the change to the code is just as simple as the change to the Lagrangian. This is an example of the field interacting with itself, and no other fields are included, which means that the only function to change is $s()$, with

```
((phi_n-phi_plus)*(phi_n-phi_plus) + (phi_n-phi_minus)*(phi_n-phi_minus))/(2*mu)
+ (mu/2)*phi_n*phi_n + lambda*phi_n*phi_n*phi_n*phi_n;
```

being the new encoded action for the one dimensional field. As always, higher dimensions just involve summing over the kinetic terms in each dimension.

The second type of interaction was the interaction of two fields of the same type. It just involved adding two free fields together, plus a coupling term which was $\lambda\phi^2\psi^2$. This was as simple as it sounds, and simply involved creating two arrays of the same size, and updating the same points at the same time. We updated both fields by the same amount each time, otherwise the acceptance rate would have been too random. The change to s.c need not be shown here, as it just incorporated the same logic as before. The function, showdist.c, was also altered in order to give a two dimensional distribution. This was achieved by binning the two fields separately, and multiplying the bins for each co-ordinate (x,y) = (n^{th}bin of ϕ,n^{th} bin of ψ) in the range, ((1,1),(1,2)...(#bins,#bins-1),(#bins,#bins)) to get the distribution over an area, rather than a length.

2.4 Parallel Adaptation

My main desire in creating parallel code was to keep interprocess communication at a minimum. This was easier while updating the field distribution than actually measuring the field, since the measurements require instant knowledge of lattice values that may exist on another computer. The lattice must be spread over all the processors in order to distribute the work evenly, since all work involves the lattice. The first change we made to the code was therefore to create a new function to decide how to divide up the field. We named this function, $devideAndConquer()$, after the method it initiates. Since my n dimensional lattice is stored in a one dimensional array, we just gave each process a section of length M/np, where M is the number of lattice points, and np stands for number of processors. To distribute the remainder, we simply added 1 to the length of each processor ranked lower than the value of the remainder (the remainder of M/np is automatically less than np, so this is safe).

Since the action at any point depends not just on the value at that position, but also on it's "next-door" neighbours, the points along the border between any two processors will depend on values found in the other processor's section. In order to reduce communication between processors, it would be nice to store with each processor the neighbouring values held by the other, without attempting to update these values (They still fall within the other processor's domain). A problem arises though when the neighbouring processor (B) updates the values at the border, since the processor (A) reading these values may be using redundant values. The solution is for processor B to avoid the points that A is using, and so the processors must have an agreement as to which points are currently being updated. The obvious and best agreement is to update even points only until the updated values are communicated, and then to update the odd points. This is best because the immediate neighbours of an odd point will all be even, and vice-versa.

Picture a chess board, with even points on white squares and odd on black. Since the updates made to the field are small, it is possible to update only odd or only even points for some time before communicating across the relatively slow network. When communication does occur, it is a quick, once off transfer of L^{d-1} values, where L is the length of the field in each dimension (assuming equal dimensions) and d is the number of dimensions. If we remember that a chess board has eight squares along each side, mentally placing two chess boards together gives us an image of the border between processors. The chess board represents a processor containing sixty four lattice points. The number of dimensions in a chess board is 2, which gives us $8^{2-1} = 8$ squares along a border. Stacking seven more chess boards atop each of the two already proximate boards causes 8^2 squares to touch, where there are 8^3 squares in the collective body (and equivalently 8^3 lattice points in each processor represented by eight chess-boards). However, since either only even or only odd points have been updated, we only need to transfer information of approximately $L^{d-1}/2$ in length, both ways. Each processor will actually transfer $2L^{d-1}$ values, since it will have a processor to it's left *and* to it's right (assuming periodic boundary conditions).

Unfortunately, much of the work involved in my program happens within correlate.c (how much depends on how large an ensemble needs to be gathered), which only updates the field between measurements. Since the measurements must be made across the entire field, we are left with a few options.

Options

1. Each time we want to measure, store the entire field on one processor.
2. Every time we need a value from another processor, retrieve it via the network connecting the processors.
3. Only measure the correlation within each processor's section.
4. Do (3) for small displacements, and do (2) for larger displacements.

The first option almost negates the advantage (within correlate.c) of updating even and odd points only, since the communication of $L^{(d-1)/2}$ values does not need to happen very often before each measurement is carried out. In comparison, each measurement carried out according to (1) involves transferring $L^d * sizeof(double)$ to one processor, and while it is carrying out operations, the other processor nodes might as well be idle, since they will have to wait in order to communicate later.

One way of possibly getting around this problem is to have one processor carry out measurements to the field while the other(s) update it, transferring data every time measurements are finished. We didn't attempt to implement this method in my own code, as it seems unlikely to work well. The amount of time it takes to compute the correlation function is not large, and using a wide range of ensembles works better than making many calculations within each ensemble, so increasing the number of measurements for each iteration would not be very advantageous. The amount of data transferred after each field change would still be large. In addition to speed losses, there is no memory advantage, since the field must be small enough to fit on one processor.

The second option also involves a lot of communication, and the communication is more random, although we have written code which keeps the transfer of data orderly, preventing deadlocks. Every processor must gather information from all processors every single time the field is measured. This information tells each processor node whether a value required by another node is in its section. Each processor starts by sending any data required to processor's of lower rank, and once finished, sends data to the right. This prevents two processor's from sending data to the other at the same time. Unfortunately, this means that information is transferred more often than one would imagine. The amount of data transferred this way is $np * sizeof(int) * measurements + numberofvaluessent$. The first term in this formula will generally be much lower than the size data transferred in (1), and if the maximum length of displacement is small compared to the size of a processor's domain, the second term should not be high either. This is an especially useful fact in higher dimensions, where we still keep the displacement to less than the length of a dimension, to prevent measuring over a full periodic dimension. In other words, if we measured far enough in one direction, we would measure the same point against itself (like looking at the back of one's head by looking straight across the universe) which would be undesirable.

However, if it is unlikely enough that we will need to measure across two processors, there should be no need to consider inter-communication at this stage at all. Each processor could simply measure it's part of the lattice $measurements/np$ times, where $measurements$ is the total number of measurements to be made on the lattice. This should be the quickest method, but the least stable. In fact, my attempt to implement this code failed to achieve the correct results for short displacements. I am not sure if this is due to a code fault or the instability of the method, since we did not have enough time to investigate further.

2.5 Kaluza-Klein Code

The final, and most complicated, piece of code incorporated aspects of all the previous sections. In fact, it had an advantage over the free field parallel code in that we were only measuring over a L^{d-1} lattice, since we are only interested in our four dimensional world (encompassed by a five dimensional world). Just as in the interaction code, we needed to create two fields, but one field exists only in four dimensions (on the brane), and the other in all five. Unlike the interaction program where we placed both fields within one action function, it is easier to simply add the action from our four dimensional code for one field to the action from our five dimensional code for the other, and include the term $\lambda\phi\psi$ in my update function. When choosing random points within the lattice, we had to check to see if the points also existed on the brane. This meant setting up separate (though connected) coordinate systems for the field in five dimensions and the one in four. To convert from five dimensions to four, we use:

q = (r/L4)*L3 + r%L + r%L2 + r%L3 + L3;

, where % is the mod operator, r is the five dimensional coordinate, and q is the four dimensional coordinate. To convert from four dimensions to five we use:

r = (q/L3)*L4 + q%L + q%L2 + q%L3 - L3;

Of course there is a loss of information when converting from five to four, but this information loss is not apparent in four dimensions. Everything else followed from this logic. We are able to figure out the length of the section containing the brane for each processor, and are able to know when a

random point exists on the brane. Because the action is stronger for the interacting field, whenever our algorithm picks a point on the brane, we force it to pick a second random point from there, thus sampling the more important, and slower evolving area more often. When sampling points outside the brane, we only use the five dimensional action, $s()$, applied to the five dimensional field. If a random point falls on the brane, however, we use the following action:

```
old = s( phi_old, n, (*phi), L, L2, L3, L4, M, mu, nu, r )
    + s2( psi_old, m, (*psi), L, L2, L3, L4, mu, nu, r )
    + g*phi_old*psi_old;
```

This is a part of our $update2()$ function, which we use outside the brane. In the above code we calculate the *old* action, and the *new* action is calculated the same way, simply substituting *phi_new* for *phi_old* and *psi_new* for *psi_old*. The second part of the action in the above code, $s2()$ is identical to the four dimensional free field code, and the first part of the action $s()$ is the same as the five dimensional free field code.

The same options for parallel measurements were available for Kaluza-Klein code as were for the Free Field algorithm. The difference here is that option (1) was less of a setback, since it took less time to transfer the four dimensional field across than it did the borders during the update process. The measurements no longer involve as much of the communication.

2.6 Software Testing

The two main methods we employ to test our code involves looking at the internal computation and at the external results. The internal testing simply involves printing out values as they are computed or passed as arguments, and double and triple checking the logical progression of our algorithms. Externally, we test our results against known graphs (see graph 4.1 and appendix B for example), but also looked for a continuation of trends seen in simpler simulations (see graphs 4.2 and 4.7). Another, obvious way of testing code is of course to run with various parameters, and this gave us the opportunity to find points in the code where we needed to catch errors (see the end of section 2.1.3 for example).

Other checks we use involve looking at cross sections of the solutions outputted by the code.

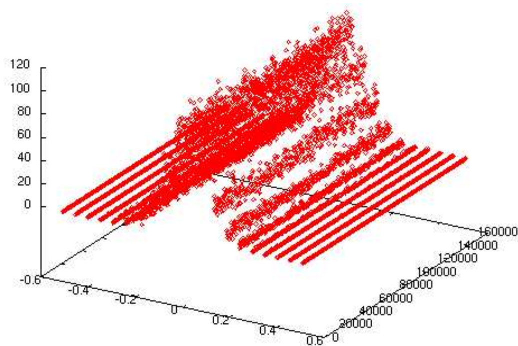

Figure 2.3: 2D cross sectional distributions on a 4D lattice

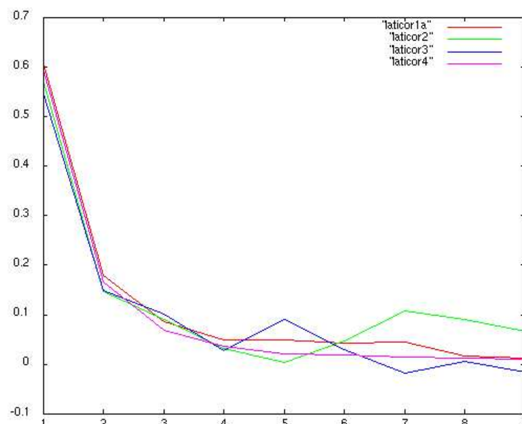

Figure 2.4: Correlation function measured in four directions

Figure 2.3 was produced by binning the distribution for every L^2 section of an L^4 lattice (L is the number of points in any direction). The entire L^4 lattice had been binned to check it's distribution, which worked out to be correct, but *this* check ensured that the distribution was locally correct at every point also. The other figure (2.4) is again measured on a four dimensional lattice. Instead of sampling the two-point correlation function over all space on the lattice, we measure it

for displacements in only one of the four directions, x, y, z and t each time. The four lines represent the four orthogonal correlation functions. In this way we tested the code against itself, since the plots must be equal (The Euclidean metric used in this space is equal to the identity matrix.), and the result above confirms this.

Chapter 3

Profiling and Parallel Performance

3.1 Serial Performance

Profiling: One Dimensional Free Field The compiler optimisation shows limited influence on the code efficiency for the one dimensional free field. The only optimisation possible is achieved by a -O1 flag and any other tricks the compiler attempts only hinder the process.

-O flag	total	% s	% update	% correlate
0	22m 6.447s	56.02	26.47	16.50
1	18m54.340s	44.76	32.24	21.79
2	20m36.022s	43.17	42.07	11.32
3	20m34.629s	43.28	42.28	11.11
4	20m34.883s	43.54	41.79	11.12
5	21m46.839s	43.29	41.65	11.33

Profiling: Two Dimensional Free Field Almost the same speed gains are achieved for two dimensions, with the peak optimisation (again -O1) somewhere between 85% and 90% of the original speed, as before. If anything, the peak optimisation is less effective here, but further optimisation, while not producing greater efficiency than the first optimisation, produces better results than that portrayed in the table above. This indicates that the complexity of the program is very relavent when choosing which flag to use when compiling.

-O flag	total	% s	% update	% correlate
0	51m50.769s	60.48	36.51	2.45
1	45m44.877s	56.36	41.01	2.45
2	47m44.765s	57.24	40.58	2.04
3	46m31.669s	62.29	35.42	2.15
4	47m34.328s	61.95	35.85	2.06
5	48m18.144s	62.68	35.15	2.05

Profiling: Three Dimensional Free Field The obvious difference here is that the peak optimisation is not achieved until the fourth row of the table below. However, the peak effect appears to decrease slightly (in terms of percentage speed up) as dimensions increase. Here the optimal time is roughly 90% of the base time.

-O flag	total	% s	% update	% correlate
0	59m21.322s	71.54	24.61	3.63
1	56m37.454s	65.31	28.08	6.37
2	54m27.367s	65.42	30.92	3.23
3	53m0.191s	65.13	31.23	3.19
4	56m24.985s	65.74	30.82	3.00
5	62m17.545s	72.37	23.93	3.49

Profiling: Four Dimensional Free Field The times taken in four dimensions are colossally different, but this is because we increased the number of iterations of the metropolis algorithm tenfold. As dimensionality increases, it takes longer for the field to settle to a fixed point, but it is unlikely that we needed to increase the time by tenfold, since we kept the number of iterations equal for one, two and three dimensions. Obviously the programmes *do* take longer as the code becomes more complex, but those time increases are exemplified by the earlier three tables, rather than the one below. The first and second optimisations are very close here, but it is the third that again provides the lowest period. Any optimisation at all immediately decreased the time by a large amount, and all optimisations are close enough that their differences could result from random noise on the computer itself. The percentage speed up here belies the previous trend, since it is roughly 82%, the highest so far when we might have expected the lowest. It seems that compiler optimisation is a little unpredictable, as the some of the methods it uses may interfere with the other methods, but may not, depending on the particular piece of code.

-O flag	total	% s	% update	% correlate
0	679m10.106s	82.73	13.55	3.35
1	557m39.435s	74.69	20.49	4.19
2	558m58.012s	74.57	20.85	3.98
3	555m30.802s	76.52	18.60	4.00
4	592m3.469s	75.67	19.04	4.59
5	591m51.823s	75.72	19.00	4.58
my opts.				
1	609m6.543s	81.26	14.93	3.40
2	589m50.753s	81.24	14.34	3.45

In all of the above cases, s.c is the most time consuming function, and also the one which optimisation effects the most, given that it dominates the other processors less when -O flags are used in compilation. With this in mind, we attempt to make our own changes to s.c. There is no point in trying to change the other functions, especially correlate.c, or those not worth showing in the above tables. The update.c file left little room for change anyway, but even were we to focuss on it, the performance difference would be minor. The changes we make to s.c involves removing dead code and common subexpressions.

The dead code was two arguments required by update.c and it's daughter, s.c, that should have been reduced to one argument. The arguments were the coefficients of the kinetic and potential terms of the action, but as it turned out (1.17), the same coefficient, μ, could be used with both terms, as a devisor and multiplier respectively. The code could have been further speeded up had we hard-coded μ into the function s.c, but hard-coding any such values into the main function instead made the program easier to use.

There were a number of common subexpressions, and it is easier to show them by looking at a section of the optimised s.c.

```
phi_minus = phi_n - ( (n%L3 < L2) ? phi[n+L3-L2] : phi[n-L2] );
phi_plus = phi_n - ( (n%L3 >= L3-L2) ? phi[n-L3+L2] : phi[n+L2] );

S = S + phi_plus*phi_plus + phi_minus*phi_minus;
```

We compare this to the code in section 2.2.1. The term **phi_n - phi_minus** and it's counterpart are now calculated once only. **phi_n** is placed outside the conditional in order to make branch prediction (automatically done at the hardware level) more effective, since this means that each branch is easier to calculate. The other change we make is to take the $\frac{1}{2\mu}$ term out of the kinetic sum, and divide afterwards. In the table above, "my opts" 1 represents some of these optimisations, while 2 includes all of the above. Notice that these optimisations almost reach the same speed up as the compiler optimisations.

3.2 Parallel Performance

In this section we simply demonstrate the performance of the four dimensional free field code, since the method of parallelization is similar for all simulations. We would expect the same speed up for the interacting fields, which we also parallelised in four dimensions. That code took a very long time to run in serial. Also, we did not create a serial version of the Kaluza-Klein code, since the bones of the parallel code had been previously designed in the parallelisation of simpler code.

Parallel Profiling (1): Four Dimensional Free Field The profile below uses method 1 in section 2.4, where only one processor carries out the work of measuring the field. Compared to the serial version in the previous section, the correlate.c function takes up a huge amount of time. This is obviously due to the overheads in parallelising the code. In particular, the method of finding even and odd points is quite intensive, and I would guess that this is the main reason for the extended use of correlate.c, especially since it is used every time update.c is called. The -O2 flag provides the greatest optimisation, with roughly 75% of the basic time. That this is the best optimisation yet is unsurprising, as it is also the most complicated algorithm, with more room for change.

-O flag	total	% s	% update	% correlate
0	340m58.792s	67.44	8.61	22.07
1	263m53.084s	71.31	10.66	16.43
2	259m32.603s	71.30	8.78	18.04
3	266m53.882s	70.08	10.97	16.94
4	263m55.643s	69.84	10.66	17.45
5	263m20.292s	69.86	10.76	17.37

Parallel Profiling (2): Four Dimensions This is the profile for method 2 of section 2.4. The difference between the two methods is carried out in correlate.c, but although this function appears to take up less time here, this method actually turns out to be slower! I am not sure why this is, but the two programs call different MPI functions, so perhaps that makes the difference. However, the profiler we are using here, gprof, does not show any time given to interprocess communication for either programs. Perhaps this is a weakness of the profiler. We had assumed that there would be less communication in this method, and were more worried that the large number of conditionals would slow it down. It seems that there is something else going on behind the scene here, unless the computers themselves were simply a lot slower when we ran method 2. The -O2 flag is again the most efficient compiler, but does not manage quite as good a speed up as it does in the previous method. However, the difference is not large, which is unsurprising from two such similar processes.

-O flag	total	% s	% update	% correlate
0	352m15.261s	68.31	8.93	20.57
1	282m20.777s	68.86	12.80	16.13
2	278m20.831s	68.14	9.32	20.27
3	299m29.089s	58.85	14.98	23.45
4	312m50.876s	64.39	10.59	22.37
5	330m03.036s	64.10	12.88	20.10

I mentioned in section 2.4 that the advantage of method 2 would be greater as the lattice size grows. The lattice size we used to test this code was $20 \times 20 \times 20 \times 20$ (20 points in each direction), which is good enough to get a correlation function, but the processors we used allow for a lattice size of somewhere around 75^4 in four dimensions in serial, and this is very slightly higher in parallel. In fact, using method 1 this size must be greatly reduced, as one processor must be able to store the entire field on it's own, as well as a section of the same field. Using a huge lattice would probably slow method 1 down a lot. The amount of data transferred across the network would rise since the entire field is communicated. Unfortunately we did not have time to test the two methods on a larger scale.

Since method 1 was the faster of the two, we tested it against the serial code. There was no need to run the code long enough to get an accurate result here, and running for a short time meant that we did not have to queue for a long time in order for the supercomputer to free up twelve

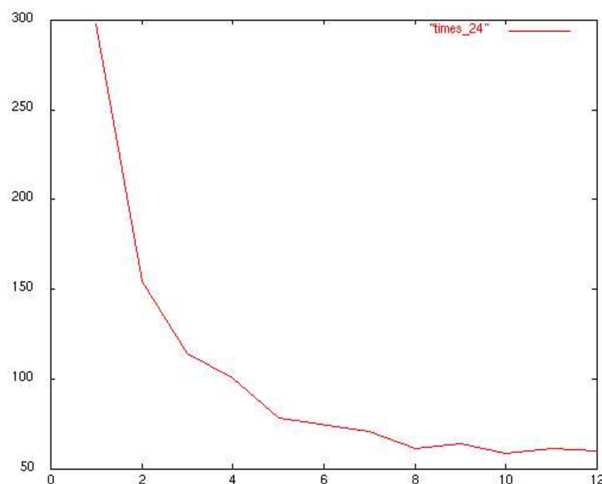

Figure 3.1: Speed up of time(s) vs processors for 4D simulator

processors. As we can see, the time is inversely proportional to the number of processors until we get above about four, at which stage it starts to flatten out. This is because each processor only needs to run for 1/(no. of processors) times the total number of iterations required to achieve a good field distribution. The profile graphs show that the overheads are quite large in correlate.c however (comparing with the serial version), and this explains why the speed up flattens out fairly soon.

In order to optimise the code for better speed up, we would need to develop a faster way of finding random even and odd points, as this appears to be the largest overhead (communications don't appear to take particularly long, at least for the lattice sizes we chose). A good alternative might be to store even and odd points separately in two parallel arrays. That way, if we are sampling even points, we simply sample from the even array, and vice-versa. Then, to access neighbouring points, one simple modulus calculation would decide whether a point is even or odd. In fact, in s.c, the neighbouring points are always even if the current point is odd, and odd if the latter is even, which requires no calculation. Since the arrays are parallel, the index of a neighbour is easy to find. The only possible loss here is in the cache, as it might not register close neighbours in the array, given that neighbours are now stored in a separate array. However, this would only have been an advantage for neighbours in the first dimension, on the same section of the array. I think that this would definitely be an improvement, and would hopefully provide better speed up beyond four processors.

Chapter 4

Results

4.1 Free Scalar Field Theory

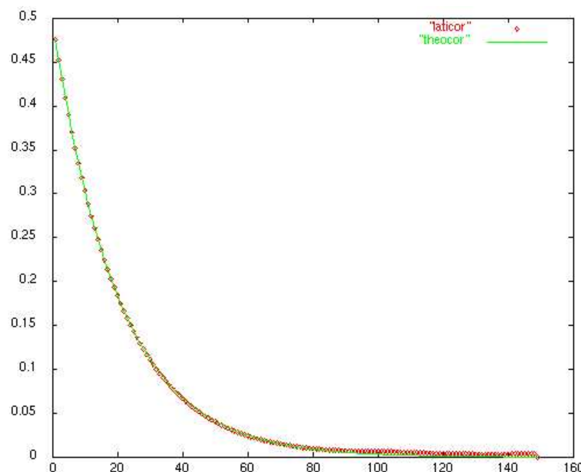

Figure 4.1: Correlation function for a 1D free field: $S = -\frac{1}{2}\int \phi(\partial_x + m)\phi$

This was an important result in that it paved the way for more advanced calculations. It represents the simplest possible correlation function, the one dimensional free field propagator. The red dots are the computational results, while the green line is the theoretical prediction, $\frac{1}{2}e^{-mr}$ (appendix A.2), where r is the distance separating the two field points. we should point out that the mass in this formula has necessarily been changed to $mu = ma$. The mass has a value of 1 and the lattice spacing is 0.05, giving $mu = 0.05$, and the number of points on the lattice is 300. The horizontal axis represents the number of lattice points separating field points.

The discrepancy as values fall towards zero disappears when the program is run using a wider sample space of fields, or by changing the field more between iterations. The graph tells us that there is a much lower probability of propagation between two points as the distance between those points widens. In fact, the amplitude (square of the probability) decreases exponentially.

To investigate the physics further, we want to know how the function (4.1 changes by varying the mass of the field. In figures 4.2 and 4.3, we show various correlation functions, where mass decreases from 1 to 9 with depth. Otherwise, each function has exactly the same specifications as in fig. 4.1. Notice that higher masses result in *increased* fall off with displacement. This makes sense, as an increase in mass means that the action becomes stronger (The field values are squared in 1.17; therefore the action is always positive.), and the principle of least action tells us that this reduces

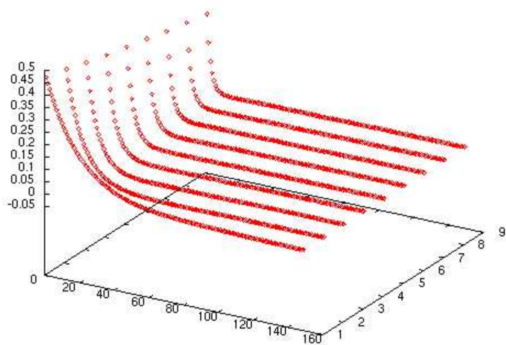

Figure 4.2: Correlation functions with various masses from 1 to 10 in steps of 1: $S = -\frac{1}{2}\int \phi(\partial_x^2 + m)\phi$

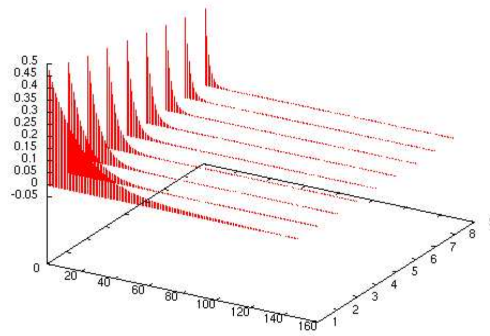

Figure 4.3: Another, perhaps clearer view of the figure 4.2:
$S = -\frac{1}{2}\int \phi(\partial_x^2 + m)\phi$

the amplitude between points on the lattice. Another way to see this is that the higher the value of our action, S, is the smaller the propagation term, e^{-S}, is. We can also see the effect that varying the mass has on the field distribution. Figure 4.4 shows this effect on a two dimensional free scalar field.

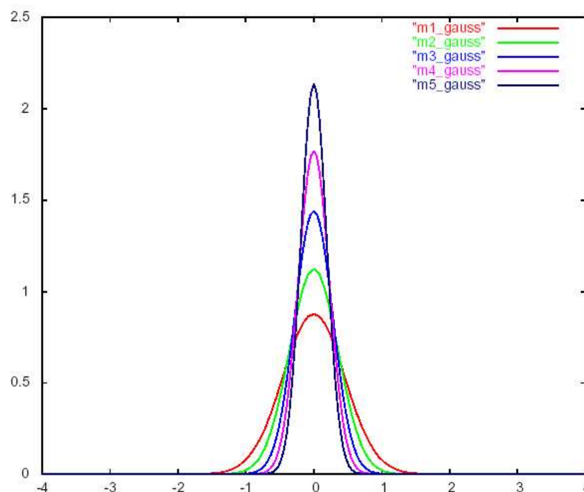

Figure 4.4: Changes to a 2D scalar field as mass increases from 1 to 5: $S = -\frac{1}{2}\int \phi(\partial_x^2 + \partial_y^2 + m^2)\phi$

The distribution with the lowest peak is for a mass of 1, the next highest has mass 2, and so on up to 5. We can see that the peak of each plot appears to rise proportionally to the mass. It seems that the more massive the field, the less easily the field fluctuates, since the field strength changes less easily. This is apparent from the fact that more points cluster around the center of distribution, and we have already seen that movement over the field is more restricted with higher mass, from figures 4.2 and 4.3.

In two dimensions the two-point correlation function showed the same trend with an increase in mass, indicating that our simulations still worked as we added dimensions. The lattice spacing and number of points is still the same in the graph below, where we have measured the correlation function for masses 1 to 5. Notice that for an equivalent mass in higher dimensions the graphs show steeper descent. This gives me a guideline for what my correlation function should look like in yet

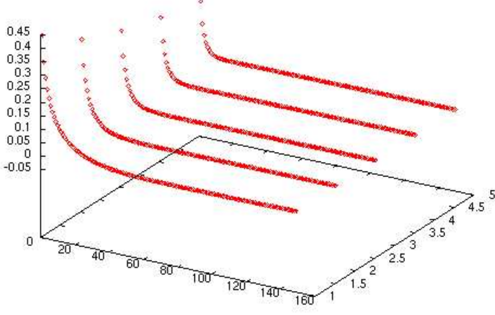

Figure 4.5: 2D free field correlation functions with various masses from 1 to 5: $S = -\frac{1}{2}\int \phi(\partial_x^2 + \partial_y^2 + m^2)\phi$

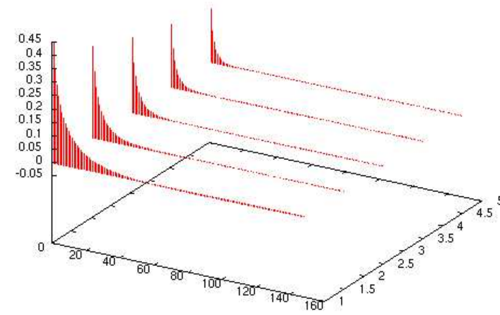

Figure 4.6: Another, perhaps clearer view of the figure 4.5: $S = -\frac{1}{2}\int \phi(\partial_x^2 + \partial_y^2 + m^2)\phi$

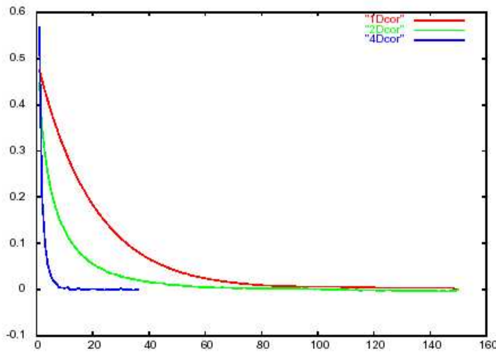

Figure 4.7: Correlation Functions for 1D, 2D and 4D (m = 1, $\mu = 0.05$): $S = -\frac{1}{2}\int \phi(m^2 + \sum_\mu \partial_\mu^2)\phi$

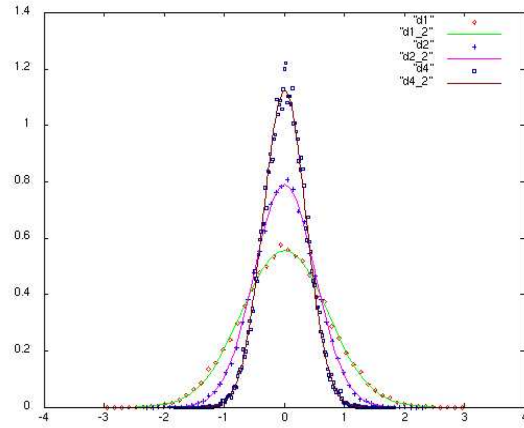

Figure 4.8: Field Distributions for 1D, 2D and 4D (m = 1, $\mu = 1$): $S = -\frac{1}{2}\int \phi(m^2 + \sum_\mu \partial_\mu^2)\phi$

higher dimensions. Figure 4.7 shows the continued trend as a succesful test for my simulation.

As expected, the four dimensional correlation falls off even faster. The next two graphs, figures 4.9 and 4.10 show a clearer picture of the individual four and two dimensional correlation functions, respectively.

I have plotted the four dimensional function beside it's theoretical prediction. The theoretical plot is given by the equation, $\frac{1}{r}e^{-mr}$, where r is measured along the lower axis, and m is the mass, in this case, 1. The four dimensional action was the same as the one dimensional action summed over four dimensions, $S = \frac{1}{2}\sum \phi_n(-\frac{1}{\mu}\Box_{n\,n'} + \mu)\phi_{n'}$, where $\mu = mass \times lattice\ spacing$. The correlation on the lattice was given by $\frac{1}{a^2.M}\sum_M \phi_n\phi_m$, where a is the lattice spacing and M is the number of measurements taken. The discrepancy between the theoretical prediction and the lattice result is distortion from the lattice. This effect is amplified as we add dimensions. The *lattice* correlation function can be found in, "Gauge theories: An Introduction" [5].

4.2 Interacting Fields

The first case of interaction we investigate is that of ϕ^4 theory, in which the scalar field interacts with itself. The results are as follows, with coupling constants 0, 0.005, 0.05 and 1. The functions

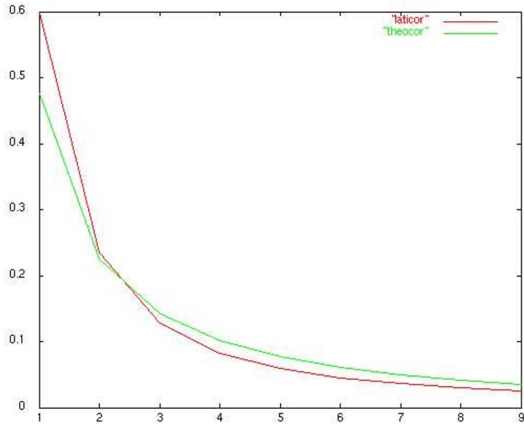

Figure 4.9: Correlation Function for 4D: $S = -\frac{1}{2}\int \phi(\Box + m)\phi$

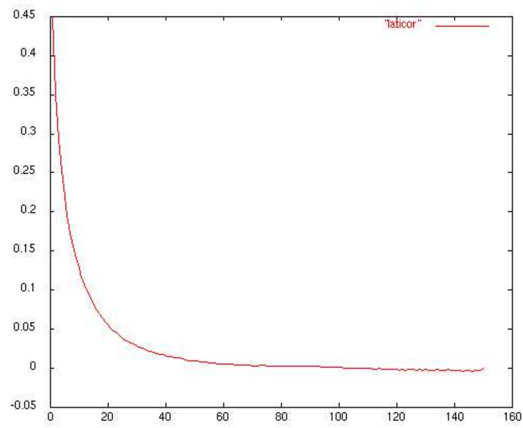

Figure 4.10: Correlation Function for 2D: $S = -\frac{1}{2}\int \phi(\partial_x^2 + \partial_y^2 + m^2)\phi$

of least amplitude are the ones with the highest coupling constant. Note that the true coupling constant is 4! times the couplings cited above, since in our code, it was inefficient to divide by 12 every time we calculated the action.

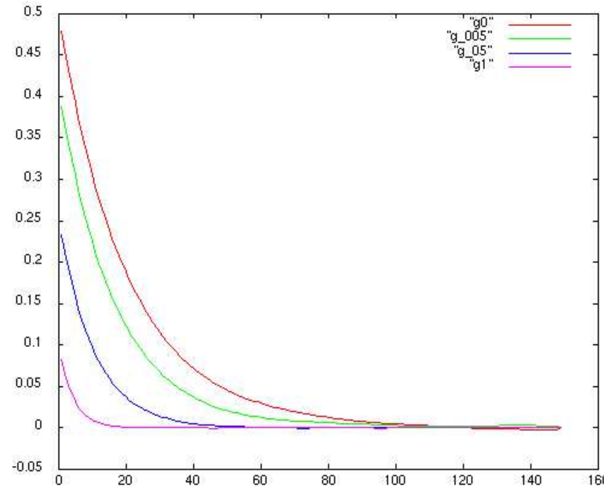

Figure 4.11: Changes in 1D ϕ^4 field with increased couplings: $S = -\frac{1}{2}\int \phi(\partial_x + m)\phi + \lambda\phi^4$

Next we investigate the effects of an interaction between two fields. We give these fields the same mass and first set the coupling between them to zero. This reduces the amplitude by one half of 4.1, but looking at how the action has changed, $S(\phi) \rightarrow S(\phi) + S(\psi)$, this is hardly surprising. $S(\phi)$ and $S(\psi)$ have equal strength, and so the action has doubled in strength for either field.

As we can see from figures 4.12 and 4.13, introducing a coupling between the two fields results in immediately decreased amplitude even for small displacements across the field. In terms of the simplest picture of the maths involved, this is obvious. The coupling adds an extra term to the action, $\lambda\phi^2\psi^2$, and we have seen so far that this decreases amplitude. However, the physics drawn from the picture is slightly different from the difference in adding dimensionality. The slope itself does not actually decrease noticibly with increased coupling. Instead the amplitude is decreased by a fairly constant amount at every displacement, and this, rather than increased fall off, is the reason that the lower plots reach near zero more rapidly. This constant decrease in amplitude is

synonymous with interference patterns that we see in wave amplitudes. The coupling of course causes interference, as both fields are now interacting directly.

Notice in figure 4.12 that the decrease going from 0 coupling to 0.01 is less than the difference caused between 0.04 and 0.05. Figure 4.13 goes from 0 to 0.5 in steps of 0.1, ten times the increment in 4.12. It seems that the coupling can only have so much effect on the correlation function. Since we have already suggested that we are seeing an interference pattern here, we can explain this easily. The interference between two fields has a minimum and maximum phase. The interference can be completely destructive, or completely constructive. The fields only have so much energy to lend to one another, and increasing the coupling only increases how much the energy of each field influences the other. Thus the amount of energy in the system limits how much the coupling can effect the interaction. This explains why we see the correlation function approaching a limit with increased coupling in graphs 4.12 and 4.13.

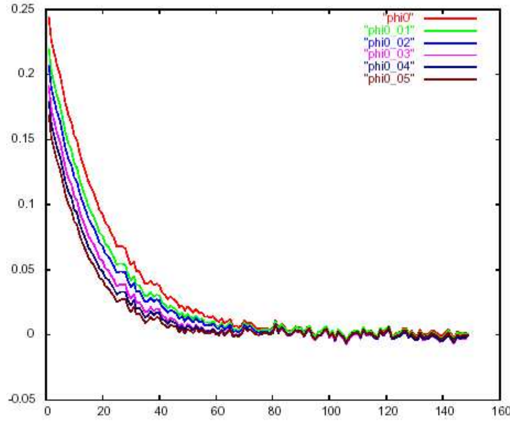

Figure 4.12: Increase in fall off with introduction of coupling: $S = -\frac{1}{2}\int(\phi(\partial_x^2 + m^2)\phi + \psi(\partial_x^2 + m^2)\psi + g\phi^2\psi^2)$

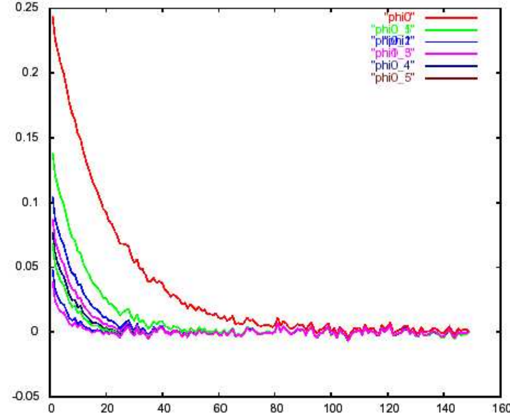

Figure 4.13: Continued trend with coupling increased tenfold: $S = -\frac{1}{2}\int(\phi(\partial_x^2 + m^2)\phi + \psi(\partial_x^2 + m^2)\psi + g\phi^2\psi^2)$

Perhaps it is a good idea to go back to the field distribution and observe the changes there. Since there are now two fields rather than one, the distribution becomes two dimensional. When we include only the potential term, with mass of 1, we get the same distribution as a standard two dimensional Gaussian distribution (Figure 4.14). Increasing the masses gives the same trend as we got for a free field. We can see this by comparing figure 4.15 to figure 4.4, but we must note that figure 4.15 does not involve a kinetic term, whereas figure 4.4 does. Both figures 4.14 and 4.15 have an action given by $S = -\frac{1}{2}\int m^2(\phi^2 + \psi^2) + g\phi^2 psi^2$, ie. it contains only the self-energy of the fields. Adding in a kinetic term gives a much lower standard deviation for the distribution, which we can see in figure 4.16. In this figure, $S = -\frac{1}{2}\int(\phi(\partial_x + m^2)\phi + \psi(\partial_x + m^2)\psi)$, and the higher peak is the one with the kinetic term added.

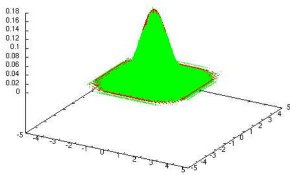

Figure 4.14: normalised two dimensional distribution

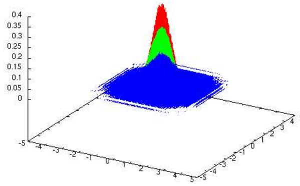

Figure 4.15: variations with mass

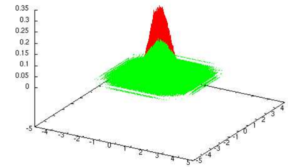

Figure 4.16: change in distribution with addition of kinetic term

Figure 4.17 shows how the potential field ($S = -\frac{1}{2}\int m^2(\phi^2 + \psi^2)$) changes with increased cou-

pling. The lowest peaked distribution has a coupling constant of zero, the next 0.05, the next 0.5, and the highest 1. Interestingly, the change from 0.05 to 0.5 is much more effective than the change between 0.5 and 1, even though the difference in coupling is roughly the same. This is a marked difference from the way that mass alters the distribution. The same can be said of the field which includes the kinetic term ($S = -\frac{1}{2}\int(\phi(\partial_x + m^2)\phi + \psi(\partial_x + m^2)\psi)$) in figure 4.18. The coupling for the lowest peaked plot is 0, then 1 for the middle peak, and 2 for the highest. Also, the kinetic term seems to damp the effect of the coupling further, since increasing the coupling has less effect in the second graph.

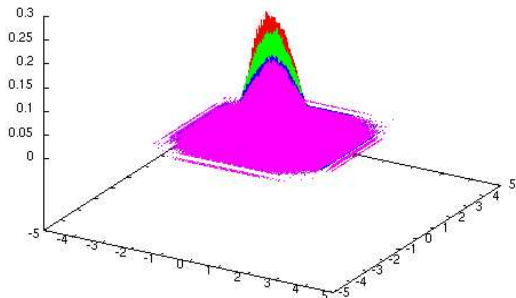

Figure 4.17: Change in distribution with introduction of coupling to purely potential fields: $S = -\frac{1}{2}\int(m^2\phi^2 + m^2\psi^2 + g\phi^2\psi^2)$

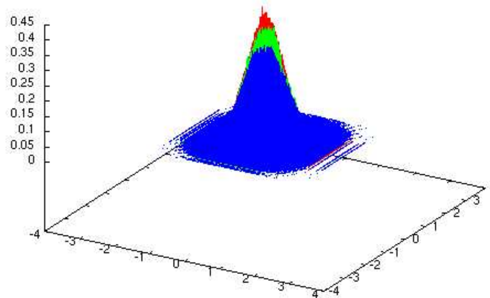

Figure 4.18: Changes to interacting fields with changes to coupling constant: $S = -\frac{1}{2}\int(\phi(\partial_x^2 + m^2)\phi + \psi(\partial_x^2 + m^2)\psi + g\phi^2\psi^2)$

4.3 Kaluza-Klein Interaction

The interaction between the field confined to the brane and the one confined to the bulk had dissapointingly little effect on the correlation function of the four dimensional field. This indicates that the five dimensional field is very weak (also true of gravity). The two-point correlation function was only measured on the field restricted to the brane, which means that there is a lot of room for further analysis here.

Figure 4.19 shows an *increase* in the amplitude as the coupling constant increases! This is the opposite of what happened for two fields of equal dimension, but a look at the last two graphs shows that this is probably just experimental error, as the lines are so close together. Figure 4.21 shows that while a coupling of 0.5 lies above the correlation function for 0 coupling, a coupling of 1 lies below. Figure 4.22 shows a range of correlation functions for various couplings from 0 to 10.

Two-point Correlation functions measured across ϕ

$$\mathbf{S_{brane}} = -\frac{\mathbf{1}}{\mathbf{2}}\int \phi(\Box + \mathbf{m^2})\phi + \mathbf{S_{bulk}} + \mathbf{g}\phi^{\mathbf{2}}\psi^{\mathbf{2}}$$

$$\mathbf{S_{bulk}} = -\frac{\mathbf{1}}{\mathbf{2}}\int \psi(\Box + \partial_{\mathbf{5}} + \mathbf{m^2})\psi$$

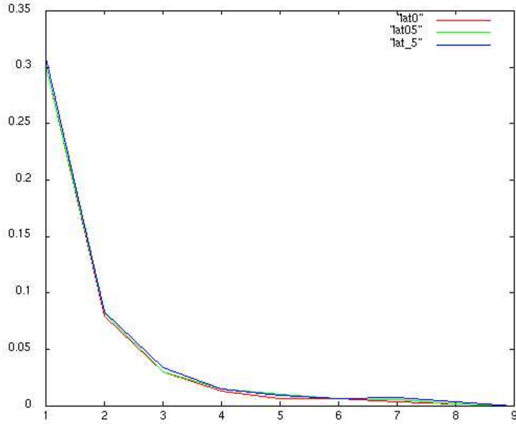

Figure 4.19: Change in correlation function on the brane with couplings g = 0, 0.05 and 0.5

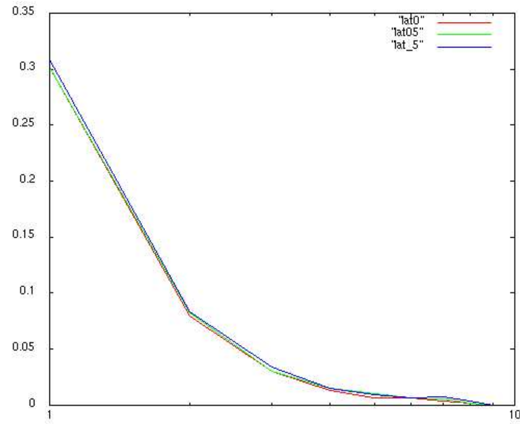

Figure 4.20: A slightly better view of figure 4.19, with the x axis scaled to $\log_{10}$ for clarity

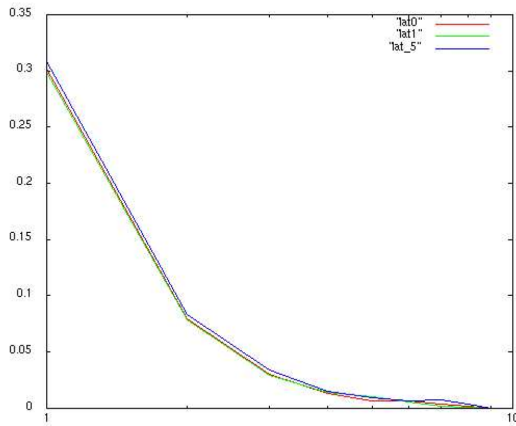

Figure 4.21: Change in correlation function on the brane with couplings g = 0, g = 0.5 and g = 1 (in log scale)

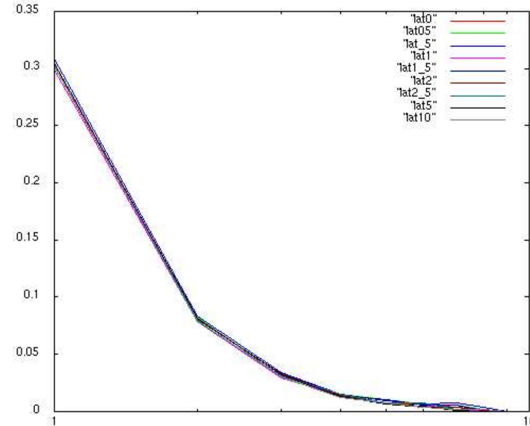

Figure 4.22: Change in correlation function on the brane for a number of couplings between 0 and 10 inclusive. (in log scale)

Chapter 5

Conclusions

The aim of this thesis was to probe the effect of an extra dimension in the correlation functions of a 4D universe. In the final graphs we show that our particular brane-bulk structure results in a very weak interaction between the two fields, but what would a different structure tell us? For example, the fifth dimension could have been made very short in length, or very long. Also, the two fields involved in the interaction can be altered. This thesis only involves similar physics to that involved in brane cosmology, as the field propagating the bulk in that picture is a graviton. The graviton of brane cosmology is a massless spin 2 boson, while both fields in my own simulation are massive scalar(spin 0) bosons. However, the bulk *is* used to explain why gravity is such a weak force, and from what we have seen here, the field on the bulk is indeed very weak, in that it does not effect the field on the brane very much at all. The brane can be said to be strongly curved, resulting in the extra fifth dimension being very short [2] (Remember the analogy between the extra dimension and a hose pipe in section 1.1.1). With this in mind, further work could be done to reduce the fifth dimension in my simulations. This would not present much difficulty, though the co-ordinate system we devised in section 2.5 would have to be revised.

Given that those experiments must be left to further study, an overview of the progression of physics throughout the project can help us to reach a more conclusive interpretation of the final results. The most helpful way to look at what is going on behind the maths and the plots might be to use the fact that the energy of a quantum mechanical system obeys the equation of the harmonic oscillator (For an example, see Peskin and Schroeder[4], section 2.3, pg 19 onwards). To see the flow of energy across the field, we can view the energy at each point on our lattice as being attached to the energy of it's neighbours by a spring, which is a good example of the harmonic oscillator.

Let's start with the basic correlation function. Every correlation function generated by my programs showed exponential fall off with increased displacement. In one dimension (which generated figure 4.1), each point would have a spring to it's left and one to it's right. Now, to cause a signal to travel along the springs, we would alter the energy at one point, which would effect the springs to either side of it, and these would effect the neighbouring points, which in turn would effect their own neighbouring points, and the signal would bounce back and forth on springs. Thus we have propagation across the field, with the springs passing on energy, and the further away from the origin of the signal, the more likely it is that the signal will die out. When we include the fact that every point has it's own kinetic energy, the interfering signals will complicate the likelihood of propagation further.

Now let's examine figure 4.2. We see that with increased mass, the signal travels a smaller distance. We should remind ourselves that we are dealing with random energies here. The energies at each point could happen to be aligned in such a way that the signal would pass a very large distance (in the way a slinky can propagate a stairway for example), but the *probability* of this is very low. Now, the mass effects the field in the same way that the weight of a ball held by two springs effects the tension of the springs. The heavier the ball, the less slack the springs are. Thus, the springs respond less easily to the signal with increased mass, and so the signal will die out more quickly on average. The analogy with spring tension can also be seen in the distribution plot, figure

4.4. We can clearly see that the energies of each point are being pulled more tightly together as mass increases, as will happen if they are attached by springs of higher tension.

The trend of figure 4.7, showing increased fall off with higher dimensions, can be described in the same way. Extending our idea of a line of points attached by springs, we can envisage the two dimensional scenario as a square mesh of springs (though in my simulations we have imposed periodic boundary conditions). At each point there is now a spring attached to the left, the right, above and below. Any movement of the energy at a particular point is therefore constrained by the movement along four instead of two springs, and so propagation is made more difficult, hence the resulting trend.

The next advancement was to simulate interacting fields. In this situation, the strength of the interaction was determined by the term $g\phi^2\psi^2$. The coupling constant, g, acts in the same way as mass did in our previous examples. Mass is still present in this scenario, but now we have another set of points attached by springs to the existing set. The tension on these springs is given by the coupling constant. The higher the coupling constant, the more rigid the spring is, and the more rigid the spring is the more easily one field moves the other. Thus, in one dimension, since there are only two springs joining any point to points on the same field, the coupling on the spring attaching the same point to the other field has a larger affect than it would in four dimensions. In this case, the energy at a given point is affected by eight springs on it's own field, and still by just one from the other field. If the coupling constant is high, much of the energy of one field will propagate into the other, rather than across itself, and so we see a reduction in the strength of the correlation function for higher couplings (figures 4.12 and 4.13).

Another study could be made on the effect of a more massive field on a less massive one, and vice-versa, but again, there is only so much one can cover in a project this size.

This analogy with springs brings us to the final conclusion. In the brane-bulk structure, we wanted to see how propagation along the four dimensional field is affected by the five dimensional field. Imagining the brane as a four dimensional mesh of points and springs and the bulk as a five dimensional mesh is difficult to picture. It is easier to reduce the problem to a one dimensional string of springs and points along the brane, and a square mesh of points in the bulk. Now imagine that each point along one line of the square mesh is attached by a single spring to an equivalent point along the line representing the brane. Kinetic energy occurs in the form of movement along the springs, which effects the points. Now, the springs oscillate in just one dimension along the brane, but movement along the equivalent line in the bulk will escape into two dimensions. Since every point in the square mesh has it's own random energy, these points will exert extrinsic tension on the line of points attached to the brane. This reduces slack in this line, and so the line along the bulk will have less strength to move than the line along the brane. If the brane had no kinetic energy of it's own, this should mean that it's correlation function would be affected greatly. This is an aspect that one could easily test using the programs we have built. However, we investigated the case where the brane *does* have its own kinetic energy. Since the energy passes along one dimension only (in our mini version), oscillations on the brane are much stronger than on the bulk, where the energy given by the coupling springs will leak out onto the mesh. The same effect should occur in four dimensions, and this explains why the brane is not affected by the bulk (though if we had measured the correlation function on the bulk, it should have been greatly affected by the brane!).

If the mesh was smaller, so that there was less random noise from the points on the bulk, we might have seen a more powerful interaction. This is obviously a matter for further study. On the computing end of things, there was an optimisation to the parallel method suggested at the end of chapter 3 that could also be investigated further. Another idea that occurred to me was the possibility of encoding a partly distributed, partly shared memory process. The advantage in this is that in my program there is only communication across the borders of each processors section of the lattice. If the borders could be stored and updated on shared memory, while the rest of a section was on the processors memory, the speed of accessing the border points could be increased. It is unlikely that two processors would want the same piece of memory at the same time if we seed the random number generator to different values on different processors. I am unaware if this sort of system has ever been implemented however, or of the problems facing it's implementation.

Appendix A

Derivations of Formulae

A.1 Free Field Equation

$$\mathcal{L} = \frac{1}{2}\dot{\phi}^2 - \frac{1}{2}(\nabla\phi)^2 - \frac{1}{2}m^2\phi^2$$

Euler-Lagrange equation (See [4] pp 15-16):

$$\begin{aligned} \partial_\mu\left(\frac{\partial\mathcal{L}}{\partial(\partial_\mu\phi)}\right) - \frac{\partial\mathcal{L}}{\partial\phi} &= 0 \\ \Rightarrow \partial_\mu\left(\frac{1}{2}2\dot{\phi} - \frac{1}{2}2\nabla\phi\right) - \left(-\frac{1}{2}2m^2\phi\right) &= 0 \\ \Rightarrow \ddot{\phi} - \nabla^2\phi + m^2\phi &= 0 \end{aligned}$$

$$\left(\Box + M^2\right)\phi(x) = 0 \tag{A.1}$$

A.2 One Dimensional Correlation Function

$$< \phi(x)\phi(y) >= \int_{-\infty}^{\infty}\frac{dk}{(2\pi)}\frac{e^{ik(x-y)}}{k^2+m^2}$$

Let:

$$f(z) = \frac{e^{iz(x-y)}}{z^2+m^2} \qquad g(z) = \frac{1}{z^2+m^2}$$

$g(z) \to 0$ as $|z| \to \infty$
$e^{iz(x-y)}$ is singular for imaginary $z \to -\infty$ but not for *imaginary* $z > 0$.
$|e^{iz(x-y)}| \leq 1$ for imaginary $z < 0$.

$$\frac{1}{m^2+k^2} = \frac{1}{z+im}.\frac{1}{z-im}$$

There are simple poles at $z = \pm im$ but only $z = im$ lies inside contour.

$$\begin{aligned} \Rightarrow residue &= \{(z-im)\left[\frac{e^{iz(x-y)}}{z+im}.\frac{1}{z-im}\right]\}_{z=im} \\ &= \frac{e^{i.im(x-y)}}{im+im} = \frac{e^{-m(x-y)}}{2im} \end{aligned}$$

$$
\begin{aligned}
\oint_c f(z)\,dz &= 2\pi i \sum (residues) \\
&= \int_{-\infty}^{\infty} g(k) e^{ik(x-y)}\,dk \\
&= 2\pi i \frac{e^{im(x-y)}}{2im} = \frac{\pi}{m} e^{-m(x-y)}
\end{aligned}
$$

$\Rightarrow$

$$
< \phi(x)\phi(y) >= \int_{-\infty}^{\infty} \frac{dk}{2\pi} \frac{e^{ik(x-y)}}{k^2+m^2} = \frac{1}{2m} e^{-m(x-y)} \tag{A.2}
$$

Appendix B

Normal Gaussian Distribution

In order to compare field distributions, a probability distribution graph is constructed for each field. To create such a graph, we pick a number of uniform intervals that together span most of the area of the graph. These intervals are called bins, because when a value in the sample falls inside an interval, it is as though it is *stored* in that bin. The graph is then produced based on how many values fell into each bin. Thus, probability distribution of my sample field depend on the number of bins and the width of each bin. The consequence of this is that in order to compare different distributions in an analysis, all graphs must be standardised as a normal distribution. A normal distribution is one where the area under the graph is equal to one.

A distribution based on an action involving derivatives is difficult to predict, so we first test our program on a simple field distribution. The free field distribution will always be Gaussian, so we take the simplest Gaussian, $\int e^{x^2/2}\,dx$, and compare it to a field distributed according to the action, $S = \phi^2/2$. This action is the one we use for our free field, except that now $d\phi = 0$ and $m = 1$.

Figure B.1 plots the distributions attained for 10^4, 10^5 and 10^6 lattice field points against the Gaussian, which is barely visible. Note that as more points are involved, each bin holds a higher number. Figure B.2 shows that the figure for each bin is also dependent on the number of bins. Using 10^5 points, we have plotted the bins as boxes. The lower graph uses 200 bins while the higher uses 20. By dividing the number of bins by 10, the height of the bins increases tenfold. Note that the 200 dots which represent 10^6 lattice points matches almost exactly with the center of each of the 20 bins. This makes perfect sense, as a wider interval can hold more values, and a larger number of lattice points results in a higher density of values in each interval.

Figure B.3 shows the graphs for 10^4, 10^5 and 10^6 lattice points overlapping with the standard Gaussian. Each are represented by 200 bin points. Intuitively from B.2 ew know to divide by the **total number of lattice points** and by the **bin width**, in order to achieve normalization. Mathematically, we work it out as follows:

Let $x =$ the signed magnitude of the field. Then for a normalised graph we have $\int_{-\infty}^{+\infty} P(x)\,dx = 1$ but our graph is made up of bins. Let w be the width of a bin. Let A_i be the bin value (the number of lattice points which fall into that bin). Then the probability of a point falling into a bin is $P_i = A_i/($**total number of lattice points**).

$$\lim_{w \to dx} \sum_{i=0}^{nbins} P_i(w) = \int_{-\infty}^{+\infty} P(x)\,dx \tag{B.1}$$

Now w represents a range of points whereas dx represents just one. To find the average probability in an range of w, we simply divide by w, **bin width**. This has the effect of reducing w to dx for the average value of x in a particular bin.

Noting that more values fall into a wider bin, and that the error in calculating my bin point is then averaged over a greater number of values, it is better to decrease the number of bins for sparser lattices. However, more bins leads to a more continuous graph, so we prefer to increase their number for denser lattices. Figures B.4,B.5,B.6 and B.7 use 40, 50, 60 and 200 bins respectively in order to achieve this balance. Note that they are much neater than figure B.3.

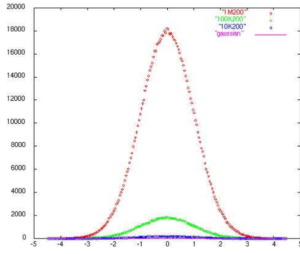

Figure B.1: unnormalised for various lattice sizes

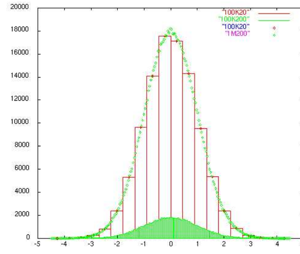

Figure B.2: unnormalised for various bins and latticess

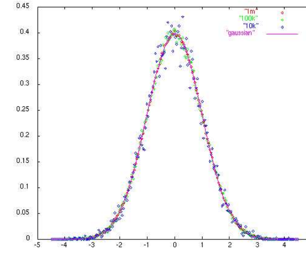

Figure B.3: normalised for various lattice sizes

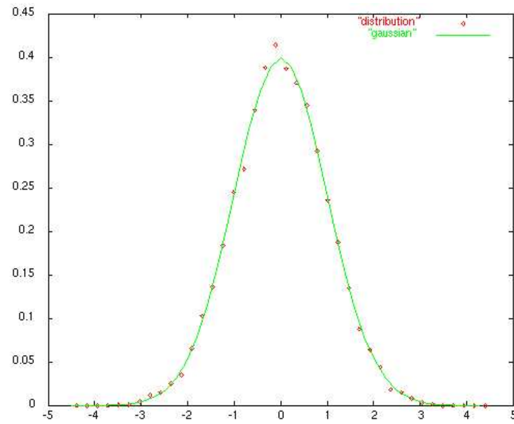

Figure B.4: normalised for 10^4 lattice points

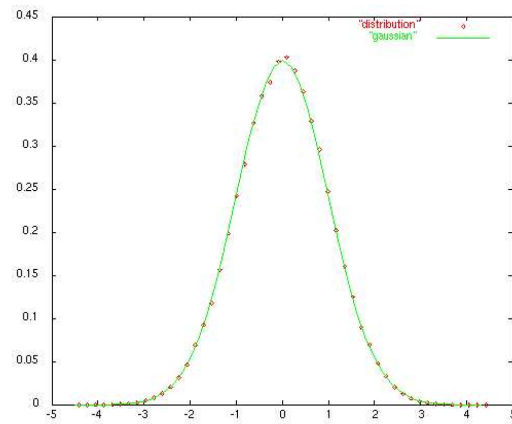

Figure B.5: normalised for 10^5 lattice points

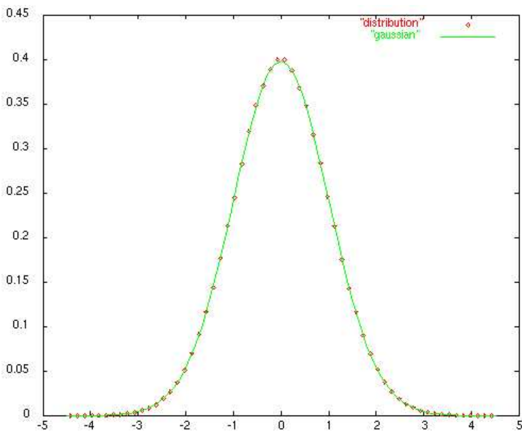

Figure B.6: normalised for 10^6 lattice points

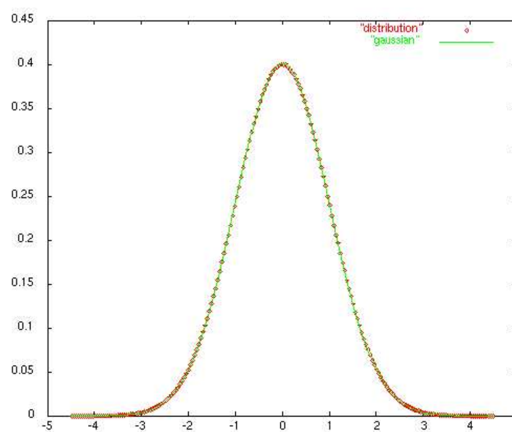

Figure B.7: normalised for 3×10^7 points

Legend: Figures B.1 and B.3: pink - standard normal, blue-10^4, green-10^5, red-10^6 points
Figure B.2: green boxes-200 bins 10^5 points, red boxes and red dots-20 bins 10^5 points, green dots-200 bins 10^6 points
Figures B.4 to B.7: green line - standard normal, red dots - lattice

Bibliography

[1] J.M Overduin and P.S. Wesson, Kaluza Klein Gravity

[2] E.Papantonopoulos, Brane Cosmology

[3] Jean Zinn-Justin, Quantum Field Theory and critical phenomena, section 2.1, p. 20

[4] Peskin and Schroeder, An Introduction to Quantum Field Theory, chapter 9

[5] Heinz J. Rothe, Lattice Gauge Theories - An Introduction