Collision Induced Decay of Metastable Baby Skyrmions

by

Daniel A. Dwyer

Submitted to the Department of Physics
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Physics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2000
[June 2000]

Author ....
.....
Department of Physics
May 5, 2000

Certified by..
.....
Krishna Rajagopal
Assistant Professor
Thesis Supervisor

Accepted by ....
.....
David E. Pritchard
Thesis Coordinator, Department of Physics

Collision Induced Decay of Metastable Baby Skyrmions

by

Daniel A. Dwyer

Submitted to the Department of Physics
on May 5, 2000, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Physics

Abstract

Many extensions of the standard model predict heavy metastable particles which may be modeled as solitons (skyrmions of the Higgs field), relating their particle number to a winding number. Previous work has shown that the electroweak interactions admit processes in which these solitons decay, violating standard model baryon number. We motivate the hypothesis that baryon–number–violating decay is a *generic* outcome of collisions between these heavy particles. We do so by exploring a 2+1 dimensional theory which also possesses metastable skyrmions. We use relaxation techniques to determine the size, shape and energy of static solitons in their ground state. These solitons could decay by quantum mechanical tunneling. Classically, they are metastable: only a finite excitation energy is required to induce their decay. We attempt to induce soliton decay in a classical simulation by colliding pairs of solitons. We analyze the collision of solitons with varying inherent stabilities and varying incident velocities and orientations. Our results suggest that winding-number violating decay is a generic outcome of collisions. All that is required is sufficient (not necessarily very large) incident velocity; no fine-tuning of initial conditions is required.

Thesis Supervisor: Krishna Rajagopal
Title: Assistant Professor

Acknowledgments

First and foremost I would like to thank my thesis advisor, Prof. Krishna Rajagopal. He gave me the chance to work on relevant research in theoretical physics when most professors would have turned away an undergraduate like me. When he found me wandering about the Center for Theoretical Physics here at MIT, he did not think twice about inviting me into his office and describing his research to me. While I was working on the research he not only made sure that I knew what I was doing, he made sure I understood *why* I was doing it. Even when we had been plagued by numerical instabilities for over five months, he continued to help me with new ideas and innovative approaches to each obstacle. Besides working with me to produce a thesis, at every chance he advised me on all aspects of a career in physics. I cannot thank him enough.

I would like to thank my friend Jesse Kirchner as well. If he had not been there I do not know how I would have gotten through my physics courses. A lot of late nights, pizza, and the occasional coffee ice cream and in the end we did alright.

I thank all the people at Student House for being there when I needed a break from my thesis; especially Madan for teaching me how to throw a forehand. I would also like to thank Prof. Maria Zuber. She hired me as a freshman with no experience and if it were not for her support my time at MIT would not have gone as well as it did.

I must not forget my family. Even though they found my research to be completely esoteric, they cheered me on and were behind me the whole way. I owe Mark Sylvester more than I can say. Aside from teaching me physics, he was a good friend and was always there to help me out, even when we were hundreds of meters underground and knee-deep in mud. Mark McCoy, my high school physics teacher in St. Charles, MO, deserves thanks for giving me my first look at physics (and also for tolerating my rambunctious friends and I.) I will never forget Knut Tarnowski; he showed me how much there is to learn and how little of it I understand.

Lastly, I want to thank my cousin Matt Woodward for telling me what physics

was and starting me on this journey which I have only just begun.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Many extensions to the standard model which involve strong dynamics at the electroweak scale include new heavy particles which have been modeled as solitons. The simplest model within which such particles can be analyzed is the standard electroweak theory with the Higgs boson mass m_H taken to infinity and with a Skyrme term [1] added to the Higgs sector. With these modifications, the Higgs sector supports a classically stable soliton whose mass is of order the weak scale, typically a few TeV.[2]

To understand how solitons arise, note that in the absence of the weak gauge interactions, the Higgs sector of the standard model is a four-component scalar field theory in which a global $O(4)$ symmetry is spontaneously broken to $O(3)$, with vacuum manifold S^3. In the $m_H \to \infty$ limit, the dynamics is that of an $O(4)$ nonlinear sigma model. Field configurations are maps from three dimensional space onto S^3, and the solitons (skyrmions) are configurations which carry the associated winding number. The winding number is topological and soliton number is conserved.

Gauging the weak interactions changes the picture qualitatively because the winding number of the Higgs field is not invariant under large gauge transformations. This means that a soliton can either be described as a skyrmion of the Higgs field with gauge field $A_\mu = 0$ or, equivalently, as a topologically trivial Higgs field configuration

with a suitably chosen nonvanishing A_μ. The latter description makes manifest the fact that there are sequences of gauge and Higgs field configurations, beginning with a soliton and ending with a vacuum configuration, such that all configurations in the sequence have finite energy. This means that the soliton is only metastable: it is separated from the vacuum only by a finite energy barrier and can decay quantum mechanically by tunneling.[3, 4, 5, 6] Or, the soliton can be kicked over the barrier if it is supplied with energy. The process in which an electroweak soliton is hit with a classical gauge field pulse (a coherent state of W-bosons) and caused to decay has been analyzed numerically.[7] It is even possible to find a limiting case of the theory in which the quantum mechanical cross-section for a process in which a soliton is struck by a single W-boson and induced to decay can be calculated analytically.[7] In any process in which a soliton is destroyed, one net baryon and one net lepton from each standard model generation is anomalously produced.[7]

Electroweak solitons have also been studied in the electroweak theory with finite Higgs mass, in which the Higgs sector is a linear sigma model.[8] If a Skyrme term is added to the theory, metastable solitons exist if m_H is sufficiently large. In the linear sigma model, the Higgs field can vanish at a point in space with only finite cost in energy. The Higgs winding number is therefore not topological even in the absence of gauge interactions. This means that in a world with gauge interactions and a finite Higgs mass, there are two ways for solitons to decay: either via nontrivial gauge field dynamics, as sketched in the previous paragraph, or via the Higgs field itself simply unwinding.[9]

The metastable electroweak soliton is an intriguing object to study. And yet, it is not found in the standard electroweak theory where the Higgs sector is a linear sigma model with no higher derivative terms. The Higgs sector of the standard model is best thought of as an effective field theory describing the low energy (weak scale) dynamics of the light degrees of freedom in some higher energy theory. The simplest examples of higher energy theories which feature particles which can be described as electroweak solitons in the low energy theory are technicolor theories, in which the technibaryons play this role. Regardless of whether the underlying

theory is specifically a technicolor model, it will introduce all higher derivative terms allowed by symmetries, including the Skyrme term, into the Lagrangian of the low energy effective theory. If the Higgs boson is discovered to be light (say, with mass $m_H \lesssim v = 250$ GeV), the correct low energy effective field theory will almost certainly not support solitons, regardless of the physics of the higher derivative terms. If the Higgs boson is discovered to be heavy, there will be some class of appropriate high energy theories whose low energy effective field theories, although more complicated than that obtained simply by adding a Skyrme term to the standard model, feature metastable electroweak solitons. Discovery of the corresponding TeV scale particles would confirm that nature chooses such a theory.

Processes in which two metastable electroweak solitons collide have to date not been studied. Our purpose in this paper is to use the analysis of a two dimensional toy model which shares some (but not all) of the features outlined above to motivate the hypothesis that the generic outcome of such collisions may be the destruction of one or both solitons. This suggests (but certainly does not demonstrate) that baryon number violation is the generic outcome of collisions between two of the TeV scale particles which can be modeled as solitons.

As a sideline, we note that our numerical methods work equally well for describing soliton–soliton and soliton–antisoliton collisions. Our focus is on soliton decay in soliton–soliton collisions; we note, however, that the numerical simulation of soliton–antisoliton annihilation in the Skyrme model is well-known as a difficult numerical problem, plagued with instabilities.[1] We are able to follow soliton–antisoliton annihilation without difficulty (with energy conserved at the part in 10^4 level). This suggests that our numerical methods — in particular the use of the linear sigma model — may be of broad utility when generalized to 3+1 dimensions.

[1]See Ref. [10] for classical simulations of skyrmion–skyrmion scattering in the 3+1 dimensional Skyrme model which report instabilities in the simulation of skyrmion–antiskyrmion annihilation; see Ref. [11] for a discussion of the origin of the instabilities and Refs.[11, 12] for efforts to overcome them.

1.2 Metastable Baby Skyrmions

Let us now introduce the 2+1 dimensional model whose metastable solitons we analyze. The Lagrangian density, which describes the dynamics of a three component scalar field $\vec{\phi} = (\phi^1, \phi^2, \phi^3)$, is

$$\mathcal{L} = F\left[\frac{1}{2}\partial_\alpha\vec{\phi}\cdot\partial^\alpha\vec{\phi} \;-\; \frac{\kappa^2}{4}(\partial_\alpha\vec{\phi}\times\partial_\beta\vec{\phi})\cdot(\partial^\alpha\vec{\phi}\times\partial^\beta\vec{\phi}) \right.$$
$$\left. -\; \mu^2(v-\vec{n}\cdot\vec{\phi}) - \lambda\left(\vec{\phi}\cdot\vec{\phi}-v^2\right)^2\right]. \qquad (1.1)$$

Here, $\vec{n}$ is a unit vector which we choose to be $(0,0,1)$.

To understand the features of this Lagrangian, it is worth beginning by setting $\mu^2 = 0$ and taking the limit $\lambda \to \infty$. When $\mu^2 = 0$, the theory has an $O(3)$ symmetry. For $\lambda \to \infty$, one removes the fourth term from (1.1) and instead imposes the constraint that $\vec{\phi}\cdot\vec{\phi} = v^2$ at all points in space and time. Because the field $\vec{\phi}$ is constrained to take values on a two-sphere of radius v, field configurations with fixed boundary conditions at infinity can be classified by their winding number

$$Q = \frac{1}{8\pi v^3}\int \epsilon_{ab}\vec{\phi}\cdot(\partial_a\vec{\phi}\times\partial_b\vec{\phi})d^2x = \frac{1}{4\pi v^3}\int \vec{\phi}\cdot(\partial_x\vec{\phi}\times\partial_y\vec{\phi})\,dx\,dy\;, \qquad (1.2)$$

which is integer-valued and topological: configurations with different winding number cannot be continuously deformed into one another. This suggests the possibility of soliton solutions to the classical equations of motion. Solitons in 2+1–dimensional $O(3)$ sigma models were first discussed in Ref. [13], and their quantum field theoretic properties were analyzed in Refs. [14, 15]. Such solitons are often called baby skyrmions [15] because of their similarity to 3+1–dimensional skyrmions. Although our motivation is the analogy to 3+1–dimensional electroweak solitons, we note that baby skyrmions themselves do arise in certain 2+1–dimensional electron systems which exhibit the quantum hall effect [16], although the Lagrangian used in their description differs from that in Eq. (1.1).

The four-derivative term in the Lagrangian (1.1) is the analogue of the Skyrme

term. It stabilizes putative solitons against shrinking to arbitrarily small size. If we were working in three spatial dimensions, the two-derivative term would stabilize putative solitons against growing to arbitrarily large size. In two spatial dimensions, however, the two-derivative term cannot play this role because its contribution to the energy of a configuration is scale invariant. We must therefore introduce a zero-derivative term in order to stabilize solitons against growing without bound. Such a term must explicitly break the $O(3)$ symmetry, and therefore has no analogue in 3+1–dimensional electroweak physics, in which no explicit $O(4)$ symmetry breaking terms are allowed. The particular form of the μ^2 term in (1.1) therefore has no electroweak motivation; it is analogous to a pion mass term in the 3+1 dimensional Skyrme model, but this is not relevant to us. This model (with μ^2 nonzero and $\lambda \to \infty$) was considered in Ref. [17], and its solitons have been analyzed in detail in Refs. [18, 19]. Similar models, differing only in the choice of the explicit symmetry breaking term in the Lagrangian, have also been analyzed.[20]

The soliton mass and size in the theory with Lagrangian (1.1) with $\lambda = \infty$ are given by[19]

$$M_{\rm sol} = 19.47F\left[a_1\sqrt{\frac{\kappa\mu}{0.316}} + a_2\right] , \qquad R_{\rm sol} \sim (3-4)\kappa\sqrt{\frac{0.316}{\kappa\mu}} , \qquad (1.3)$$

with a_1 and a_2 dimensionless constants (independent of $\kappa\mu$) satisfying $a_1 + a_2 = 1$. The parametric dependence of these results can be understood by noting that the energy of a configuration of size R receives contributions of order FR^0, $F\kappa^2R^{-2}$ and $F\mu^2R^2$ from the first three terms in the Lagrangian (1.1) and that, as described above, a soliton is stabilized by the balance between the four-derivative κ^2 term and the zero-derivative μ^2 term.

If we stopped here, with λ infinite, our solitons would be absolutely stable, rather than metastable. Soliton–soliton collisions have been simulated in this theory, but of course the solitons never decay.[19] Once λ is finite, the fields are allowed to deviate from $\vec{\phi}\cdot\vec{\phi} = v^2$, and the soliton configuration with $\vec{\phi}\cdot\vec{\phi} = v^2$ found previously in the $\lambda \to \infty$ theory may unwind and decay. Indeed, we will see that soliton solutions do

not exist for λ less than some λ_c. If $\lambda > \lambda_c$, metastable solitons exist: these solitons are classically stable if left unperturbed, but can be induced to decay if supplied with sufficient energy. Our goal is to determine whether the means by which the energy is delivered is important or whether soliton decay is the result of generic soliton–soliton collisions, without finely tuned initial conditions.

For our purposes, λ is the most important parameter in the theory because by choosing its value, we control the energy required to make the soliton decay and indeed control whether solitons exist in the first place. We are not interested in the dependence on the other parameters, and indeed most of them can be scaled away. We first set $v = 1$ by rescaling ϕ. Next, the constant F has units of energy and we henceforth measure energy in units such that $F = 1$. Next, κ has units of length and we henceforth measure length in units such that $\kappa = 1$. Note that this means that $\hbar \neq 1$ in our units, but this will not concern us as we only discuss the classical physics of this model. We have set the speed of light $c = 1$ throughout. The parameters μ^{-1} and $\lambda^{-\frac{1}{2}}$ are also length scales in the Lagrangian, and the theory is therefore fully specified by the two dimensionless parameters $\lambda\kappa^2 = \lambda$ and $\mu\kappa = \mu$. Although results do depend on μ, we are not very interested in this dependence, and we choose to follow Ref. [19] and set $\mu^2 = 0.1$ through most of the paper. The only time an alternate value of μ is used is outlined at the end of Section 2. Once we have chosen units with $F = \kappa = 1$ and have chosen to set $\mu^2 = 0.1$, then $M_{\rm sol} = 19.47$ and $R_{\rm sol} \sim (3 - 4)$ in the theory with $\lambda = \infty$.

In Section 2, we find metastable soliton configurations for finite values of λ with $\lambda > \lambda_c \sim 7.6$. We shall see that for all values of λ for which solitons exist, the soliton mass and size change little from their values at $\lambda \to \infty$. Although we do not fully explore their dependence on μ, we use the equations in (1.3) to make a rough estimate of λ_c for an alternate value of μ. In Section 3, we present our results on soliton–soliton collisions. We find that soliton decay occurs for incident velocities greater than some critical value v_c. We explore how this critical velocity depends on λ and on the initial impact parameter and relative orientation of the two solitons. We find that v_c is less than or of order half the speed of light regardless of the relative orientation as long

as $\lambda \lesssim 2\lambda_c$ and b is less than or of order the soliton size. Thus, inducing soliton decay does not require specially chosen initial conditions; it is a generic outcome of soliton–soliton collisions. We make concluding remarks in Section 4.

It perhaps goes without saying that our model is at best a crude toy model for the electroweak physics which motivates our analysis. First, we work in 2+1 dimensions. Second, in order for the theory to have soliton solutions we are forced to include a zero-derivative explicit symmetry breaking term not present in the electroweak theory. Third, we do not introduce a gauge field. Hence, our solitons can only decay via unwinding the scalar field; in the electroweak theory, gauge field dynamics introduces a second decay mechanism which has no analogue in our theory. Related to this, our solitons are absolutely stable for $\lambda = \infty$, whereas electroweak solitons are metastable even for $m_H = \infty$. This is perhaps the biggest qualitative difference between our model and electroweak physics. Fourth, one may worry that even if an analysis along the lines of ours were done in the 3+1 dimensional electroweak theory itself, the momenta required would make it impossible to analyze soliton decay within the effective theory. This concern may be evaded for solitons which are almost unstable: in this circumstance, for example, W-soliton collisions can result in soliton destruction even if the W-boson momentum is small enough that the calculation is controlled.[7] Soliton–soliton scattering in our model is far from being a complete analogue of the scattering of TeV scale particles which can be modeled as metastable electroweak solitons; we nevertheless hope that our central result, namely that metastable baby skyrmions in 2+1 dimensions are destroyed in collisions with generic initial conditions, motivates future work on baryon number violating scattering in this sector.

Chapter 2

Finding Static Solitons

Before we can study soliton–soliton collisions, we must find the metastable soliton configurations for different values of λ. We do this by looking for configurations which minimize the static Hamiltonian H_{static} at a given λ. The static Hamiltonian is given by

$$H_{\text{static}} = -\int d^2x \, \mathcal{L}_{\text{static}} \ , \tag{2.1}$$

where $\mathcal{L}_{\text{static}}$ is the Lagrangian density of (1.1) with all terms containing time derivatives set to zero.

We discretize H_{static} on a square lattice of 125×125 points, with the spatial separation between points given by $\Delta x = 0.2$ (in our units in which $\kappa = 1$). We discretize the two derivative term in the standard fashion, writing it as a sum over terms like

$$\left[\frac{\phi^i(x,y) - \phi^i(x-\Delta x, y)}{\Delta x}\right]^2 \ . \tag{2.2}$$

The Skyrme term is trickier to handle, because it involves terms like

$$\partial_x\phi^1 \, \partial_y\phi^1 \, \partial_x\phi^2 \, \partial_y\phi^2 \ . \tag{2.3}$$

We discretize this contribution to the Hamiltonian as a sum over terms like

$$\left(\frac{\phi^1(x+\Delta x, y) - \phi^1(x-\Delta x, y)}{2\Delta x}\right)\left(\frac{\phi^1(x, y+\Delta x) - \phi^1(x, y-\Delta x)}{2\Delta x}\right)$$

$$\times \left(\frac{\phi^2(x+\Delta x, y) - \phi^2(x-\Delta x, y)}{2\Delta x} \right) \left(\frac{\phi^2(x, y+\Delta x) - \phi^2(x, y-\Delta x)}{2\Delta x} \right) \quad (2.4)$$

In this way, we ensure that within each term in the sum over lattice sites, all spatial derivatives are centered at the same point in space. Discretizing the Hamiltonian in this fashion ensures that discretization errors are of order $(\Delta x)^2$ and is outlined fully in Appendix A.

In order to find a soliton, we begin with a guess (which we describe momentarily) for the configuration $\vec{\phi}(x, y)$ and perform a numerical minimization of the static Hamiltonian using the conjugate gradient method of Ref. [21]. This method is thoroughly described in Section B.4 and the computer code can be found in Section C.6(It is important to use a method such as this one, which minimizes a function of N variables using computer memory of order N rather than of order N^2 since we have an $N = 3 \times 125 \times 125$ dimensional configuration space.) In order to minimize the energy, the conjugate gradient routine needs expressions for the gradient of the energy at any point in our N dimensional configuration space, with respect to each direction in this configuration space. We obtain these expressions by varying the discretized H_{static} with respect to the ϕ^i at each lattice site. (These expressions will of course also appear as the terms with no time derivatives in the dynamical equations of motion of Section 3.)

For $\lambda \to \infty$, soliton solutions can be written in the form[17, 18]

$$\vec{\phi}(r, \theta) = \begin{pmatrix} \sin f(r) \cos\theta \\ \sin f(r) \sin\theta \\ \cos f(r) \end{pmatrix} \quad (2.5)$$

where $f(r)$ satisfies the following conditions:

$$f(0) = \pi, \quad (2.6)$$

$$\lim_{r\to\infty} f(r) = 0 \, . \quad (2.7)$$

(We define polar coordinates such that the soliton is centered at $r = 0$, $\theta = 0$ is

the positive y-axis, and θ increases in a clockwise direction.) Note that because of the μ^2 term in the Lagrangian which breaks the $O(3)$ symmetry, $\vec{\phi}$ must point in the ϕ^3 direction at large r. The $O(2)$ symmetry associated with rotations in the (ϕ^1, ϕ^2) plane is not broken in the Lagrangian; in the solution, these rotations are mapped onto rotation in the (x, y) plane about the soliton center. This configuration is thus a two-dimensional analogue of what in three dimensions is called a hedgehog configuration.

In our search for solitons at finite λ, we therefore begin by choosing a reasonably large λ, namely $\lambda = 15$, and making an initial guess of the form (2.5) with $f(r) = \pi \exp(-r/2)$. We then run the conjugate gradient relaxation algorithm repeatedly, until the change in the energy between successive relaxation steps is smaller than one part in 10^{10}.[1] The soliton configuration we find is a hedgehog configuration, as at $\lambda \to \infty$. However, when λ is finite, $\vec{\phi} \cdot \vec{\phi} \neq 1$. The soliton we find can be written in the form

$$\vec{\phi}(r, \theta) = \sigma(r) \begin{pmatrix} \sin f(r) \cos \theta \\ \sin f(r) \sin \theta \\ \cos f(r) \end{pmatrix} \tag{2.8}$$

with $f(r)$ satisfying the same boundary conditions as above.[2] We depict the soliton configuration in Fig. 2-1.

After obtaining a baby skyrmion at $\lambda = 15$, we used the resulting configuration as the initial condition for relaxation at $\lambda = 14$, and so found the soliton configuration at this λ. We repeated this process step-by-step in λ, finding solitons for values of λ down to $\lambda = 8$. At $\lambda = 7$, energy minimization led to a configuration with zero energy,

[1] As a check, we then used this configuration as an initial condition for the full time-dependent dynamical equations of motion described in the next section. The total kinetic energy during the time evolution was never more than one part in 10^7 of the soliton energy. This confirms that the relaxation algorithm has indeed converged to a static solution to the full equations of motion.

[2] Note that we could have rewritten the static Hamiltonian in terms of $\sigma(r)$ and $f(r)$, discretized that Hamiltonian in r, and then used a conjugate gradient algorithm to find these two functions of r. This would have been less computationally intensive than finding $\phi^i(x, y)$ as we did. However, the expressions we obtain by varying our static Hamiltonian relative to the fields ϕ^i at each lattice site, and indeed the results we obtain for ϕ^i at each lattice site in a soliton configuration, are precisely what we need in the next section when we analyze soliton–soliton collisions, which are of course not circularly symmetric and so cannot be written in terms of $\sigma(r)$ and $f(r)$.

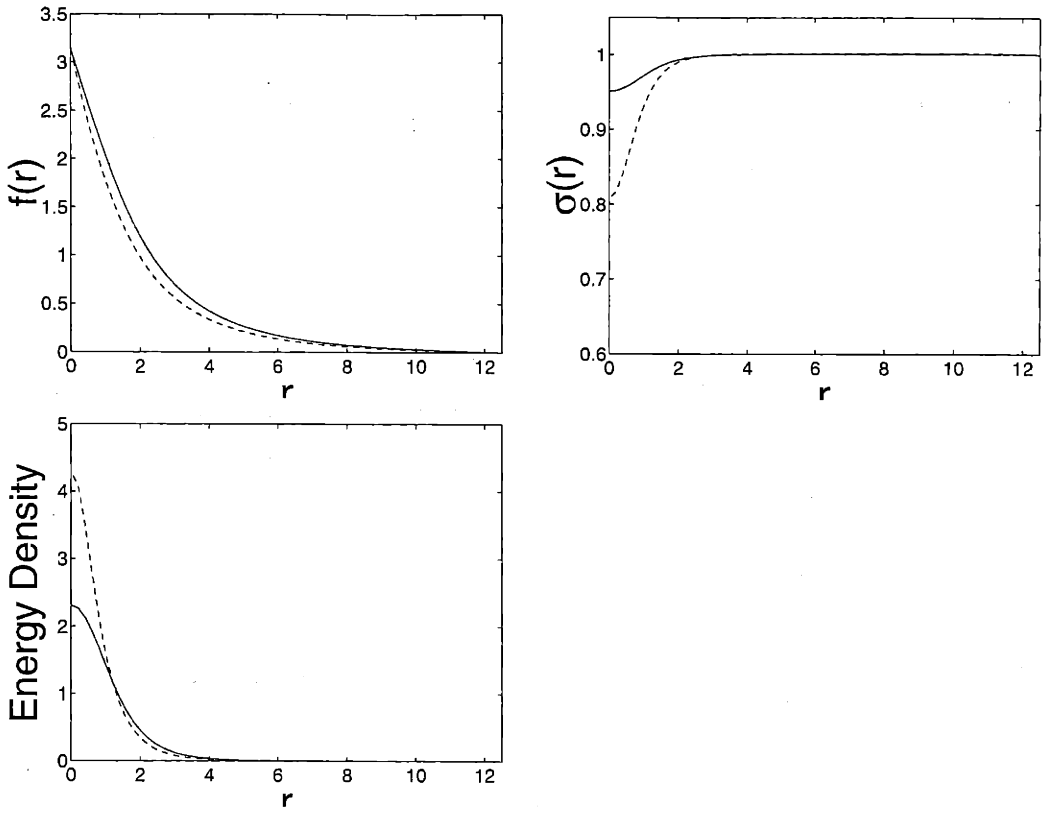

Figure 2-1: $f(r)$, $\sigma(r)$ and the energy density for the solitons with $\lambda = 15$ (solid curves) and $\lambda = 7.7$ (dashed curves).

instead of to a soliton. We then used the $\lambda = 8$ soliton as an initial configuration for relaxation at $\lambda = 7.9$, and so on down to $\lambda = 7.6$ where again no soliton was found. We therefore know that a stable soliton exists at $\lambda = 7.7$. It is a logical possibility that there is a stable soliton at $\lambda = 7.6$ even though our relaxation algorithm did not find one. We think this is unlikely, because the soliton configurations which we have found at $\lambda = 7.7$ and $\lambda = 7.8$ are very similar, and we therefore believe that the $\lambda = 7.7$ soliton is a very good starting configuration from which to find the $\lambda = 7.6$ soliton if it existed. We therefore conclude that classically stable solitons exist only for $\lambda > \lambda_c$, with $7.6 < \lambda_c < 7.7$.

In Table I, we give the energies of the solitons which we have found for various values of λ. In Fig. 2-1, we depict the field configuration and energy density for the

λ	Energy
15	19.1792
14	19.1503
13	19.1161
12	19.0751
11	19.0250
10	18.9618
9	18.8791
8	18.7619
7.9	18.7473
7.8	18.7309
7.7	18.7131
7.6	no soliton

Table 2.1: Energy of Static Solitons at various Lambdas.

solitons we have obtained for $\lambda = 15$ and $\lambda = 7.7$. We note that even though $\lambda = 7.7$ is only just above λ_c, the soliton configuration does not look very different from that at much larger values of λ, and the soliton energy is also little changed. Note that the deviation from $\sigma(r) = 1$ is only at most 20% for a soliton with $\lambda = 7.7$ which is on the edge of instability. The central energy density does increase by almost a factor of two as λ is reduced from 15 to 7.7. Note, however, that the total energy is almost unchanged, and actually decreases slightly. The soliton radius decreases as λ is reduced towards λ_c, but does not decrease dramatically. The definition of $R_{\rm sol}$ is of course somewhat arbitrary; if we take it to be the radius inside which 90% of the total energy of the soliton is found, we find $R_{\rm sol} = 3.31$ for $\lambda = 15$ and $R_{\rm sol} = 2.83$ for $\lambda = 7.7$.

Although the energy density and $\sqrt{\vec{\phi} \cdot \vec{\phi}} = \sigma$ are circularly symmetric, the fields ϕ^1 and ϕ^2 in a soliton configuration are not circularly symmetric. If we only observed a single static soliton, this would be of no consequence: in a hedgehog configuration, the different possible choices for ϕ^1 and ϕ^2 are related simply by rotations in space. However, when we describe a configuration of two well-separated solitons in the next Section, the relative angle α between their orientations does matter. That is, specifying such a configuration requires giving the relative position and velocity of the

centers of the two solitons and the angle α. The first soliton in such a configuration can be mapped onto the second by a translation followed by a rotation by an angle α about the soliton center.

It is also possible to roughly estimate the dependence of λ_c on μ. We argue that a soliton is unstable when the energy density approaches the energy necessary for $|\vec{\phi}|$ to pass through zero. According to the last term in the Lagrangian density (Eq. (1.1)), the energy density when $|\vec{\phi}| = 0$ is

$$\sim F\lambda v^4. \tag{2.9}$$

We then make the rough approximation that the energy density must be of this order in an area of order the soliton size. We write this area as ηR_{sol}^2 where R_{sol} is given by Eq. (1.3) and η is an unknown parameter. This results in a total energy of

$$E_{decay} = F\lambda v^4 \eta \kappa^2 \left(\frac{0.316}{\kappa\mu}\right) \tag{2.10}$$

necessary for decay. The total energy available is just given by M_{sol} of Eq. (1.3), where the total energy is about 19.18 for our $\lambda \neq \infty$ soliton;

$$M_{sol} = 19.18F\left[a_1\sqrt{\frac{\kappa\mu}{0.316}} + a_2\right]. \tag{2.11}$$

We therefore expect the soliton to become unstable when $E_{decay} = M_{sol}$, or

$$F\lambda_c v^4 \eta \kappa^2 \left(\frac{0.316}{\kappa\mu}\right) = 19.18F\left[a_1\sqrt{\frac{\kappa\mu}{0.316}} + a_2\right]. \tag{2.12}$$

If we were then to do multiple relaxations and determine the dependence of λ_c on μ we would then expect it to have the form

$$\lambda_c = b_1\mu^{\frac{3}{2}} + b_2\mu. \tag{2.13}$$

Without doing such a thorough study we can obtain λ_c for another value of μ using the relaxation algorithm and compare it to the estimate of Eq. (2.12). In our

case we chose a new $\mu_n = \frac{\mu}{2}$, i.e. $\mu_n^2 = \frac{0.1}{4}$. As stated in Section [1], we have set $F = v = \kappa = 1$, so it is only necessary to determine values for η, a_1 and a_2 to solve for the new λ_c in Eq. (2.12). A value for η is given from the fact that we found $\lambda_c \simeq 7.6$ when $\mu^2 = 0.1$. Using these values in Eq. (2.12), we find $\eta = 2.52$. We performed a relaxation with this new μ_n and found that the total energy of the static soliton was 16.30. Since we now know the mass of the soliton for two values of μ we have the set of equations

$$\begin{aligned} 19.18\,[a_1 + a_2] &= 19.18, \\ 19.18\left[a_1\sqrt{\frac{1}{2}} + a_2\right] &= 16.30. \end{aligned} \tag{2.14}$$

This system gives the values $a_1 = 0.51$ and $a_2 = 0.49$. It is then trivial to solve for the new λ_c at μ_n. Eq. (2.12) gives $\lambda_c = 3.2$. We can now compare this to a value obtained from the numerical relaxation algorithm. In the same manner that we found $\lambda_c \simeq 7.6$ for $\mu^2 = 0.1$, using the relaxation method we found that $\lambda_c \simeq 4.0$ for μ_n. The estimate and relaxation agree roughly, and the discrepancy can be attributed to the fact that the energy density is not distributed evenly over the area of the soliton. In fact, we expect the λ_c obtained by relaxation to be higher than our estimate, since the energy becomes more localized near the center for smaller values of λ.

Chapter 3

Colliding Solitons

With solitons in hand, we are ready to study what happens when they collide. For this purpose, we need discretized equations of motion and a numerical algorithm to evolve an initial configuration, now specified by ϕ^i and $\dot{\phi}^i$ at each lattice site, forward in time. We begin by writing a discretized Lagrangian which is a function of ϕ^i and $\dot{\phi}^i$ at each of the lattice sites, at a single time t. We discretize the time-independent terms as described in the previous Section. There are no spatial derivatives of $\dot{\phi}^i$ in the Lagrangian, so discretizing terms involving $\dot{\phi}^i$ is trivial. We then use the Euler-Lagrange procedure on this Lagrangian written in terms of $3 \times 125 \times 125$ ϕ's and $3 \times 125 \times 125$ $\dot{\phi}$'s, and obtain equations of motion which specify the $3 \times 125 \times 125$ $\ddot{\phi}$'s. These equations of motion take the form of three coupled linear equations for $\ddot{\phi}^1$, $\ddot{\phi}^2$ and $\ddot{\phi}^3$ at a given lattice site, which are easily solved. This entire derivation is outlined in Appendix A. We now have an expression for $\ddot{\phi}^i(t, x, y)$ written in terms of the values of ϕ^i and $\dot{\phi}^i$ at lattice sites within two spatial links of the site of interest, all at the same time t.[1] We are now ready to take a step forward in time.

We evolve the system forward in time using the Runge-Kutta-Feldberg algorithm and the adaptive algorithm of Ref. [21] for choosing the size of the time step Δt. That is, we first use the fifth-order Runge-Kutta-Feldberg algorithm to obtain ϕ^i and $\dot{\phi}^i$ at

[1]Note that because of the way we discretize spatial derivatives in the Lagrangian, expressions in the equations of motion with mixed time-space derivatives such as $\partial_t \partial_x \phi^i$ end up discretized as $[\dot{\phi}^i(x + \Delta x, y) - \dot{\phi}^i(x - \Delta x, y)]/2\Delta x$.

time $t + \Delta t$. This fifth-order method is special because a rearrangement of the fifth-order function evaluation terms results in a fourth-order Runge-Kutta expression.[2] We then have two different estimates (fourth order and fifth order) for ϕ^i at $t + \Delta t$ at each lattice site, and can evaluate the discrepancy between the two estimates for each of the $3 \times 125 \times 125$ ϕ's and $\dot{\phi}$'s. If the largest discrepancy is larger than a specified tolerance, we reject the step and begin anew with a smaller Δt. We use the largest discrepancy to estimate how much Δt should be reduced. If all discrepancies are smaller than the specified tolerance, we accept the result of the fifth-order calculation for ϕ^i and $\dot{\phi}^i$ at time $t + \Delta t$. After a successful step forward in time, we use the largest discrepancy (which must have been less than the tolerance since the step forward was accepted) to estimate by how much we can safely increase Δt when we take our next step forward in time. In the simulations of collisions which we describe below, the tolerance is approximately $0.01 \lesssim \Delta t \lesssim 0.05$. Note that we do *not* use conservation of energy as our criterion for acceptance or rejection of a step forward in time. This makes it fair to use a check of the conservation of energy as an independent measure of the accuracy of our evolution algorithm. We do this at various points below. A full description of this algorithm can be found in Section B.5, and the related C code can be found in Section C.7.

We choose fixed boundary conditions, with $\vec{\phi}$ fixed to its vacuum value $(0, 0, \sigma_{\rm vac})$ at the boundaries of our 125×125 grid, where $\sigma_{\rm vac}$ solves $(\sigma_{\rm vac}^2 - 1)\sigma_{\rm vac} = \frac{\mu^2}{4\lambda}$ and is $\sigma_{\rm vac} \simeq 1 + \frac{\mu^2}{8\lambda}$ for large λ. Since the solitons have radii of order $R_{\rm sol} \simeq 3$, we choose initial conditions with two solitons whose centers are a distance 10 apart. We initialize $\vec{\phi}$ by adding these two soliton configurations. (That is, we take $\vec{\phi}_{\rm vacuum} + (\vec{\phi}_{\rm first\ soliton} - \vec{\phi}_{\rm vacuum}) + (\vec{\phi}_{\rm second\ soliton} - \vec{\phi}_{\rm vacuum})$.) The resulting configuration is not precisely a minimum of the static Hamiltonian, but the two solitons are far enough apart that this is not a big concern. To obtain a soliton moving with an initial speed

[2] This hidden fourth-order expression is referred to as an *embedded* Runge-Kutta formula due to the fact that it can be obtained with no additional function evaluations.

v in the positive x direction, we simply initialize

$$\dot{\phi}^i(x,y) = -v[\phi^i(x,y) - \phi^i(x-\Delta x, y)]/\Delta x \tag{3.1}$$

at time zero. For simplicity, we are using a Galilean boost. This is appropriate for $v \ll 1$. When we use this prescription with a velocity at which relativistic corrections are becoming important, the initial condition we have specified is not the correct Lorentz-boosted, Lorentz-contracted soliton. In this circumstance, as the system is evolved forward in time, the soliton radiates some energy and quickly settles down to become a (correct) relativistic soliton moving with a velocity somewhat less than v. For example, when we set $v = 0.8$ in our Galilean boost prescription for the initial condition, we in fact end up with a soliton moving at a speed of 0.61.

We begin by analyzing collisions between two solitons in the theory with $\lambda = 10$. We choose initial conditions in which both solitons are moving (towards each other) with velocity $v = 0.25$, with zero impact parameter. We choose an initial relative orientation angle $\alpha = 0$, meaning that one soliton is obtained from the other by translation without rotation. Previous work shows that two static solitons with this relative orientation repel each other.[19] This is consistent with what we find: for low velocities, as for example for $v = 0.25$, the two solitons bounce off each other and return whence they came. We now increase v to 0.5. This time, the outcome, depicted in Fig. 3-1, is that the solitons are destroyed in the collision. The final state is a cloud of debris, namely small amplitude oscillations of the $\vec{\phi}$ field spreading outwards from the scene of the collision. In Fig. 3-2, we show the kinetic energy, potential energy and total energy for the collision shown in Fig. 3-1. (By "potential energy" we mean the contribution to the energy from all those terms in the Hamiltonian with no time derivatives. Most of this energy is due to spatial gradients of the fields.) First, we see that the total energy is conserved, in fact to better than two parts in 10^5. The kinetic energy is not zero initially, because the solitons are moving. As the solitons approach each other, the kinetic energy decreases. This confirms that the interaction is repulsive: the solitons slow down and deform as they approach. As the solitons

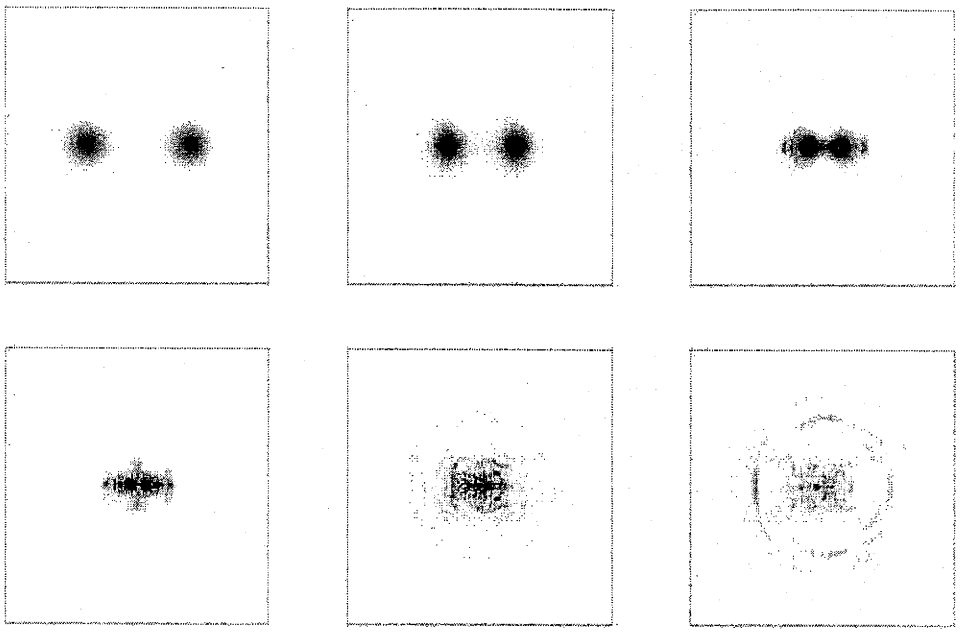

Figure 3-1: Sequence of snapshots of the energy density during a collision between two solitons which results in the destruction of both. The grey scale indicates energy density. In this simulation, $\lambda = 10$, the initial velocity of each soliton is $v = 0.5$, the impact parameter is $b = 0$, and the solitons have a relative orientation angle $\alpha = 0$ in the initial configuration. The images are at times $t = 0, 4, 8, 12, 16, 20$. In this and in all subsequent figures showing soliton–soliton collisions, each panel shows a 25×25 square (in our units in which $\kappa = 1$) and the initial separation between solitons is 10. The lattice spacing is $\Delta x = 0.2$.

approach each other more closely, at some point their deformation becomes sufficient that they are no longer stable, and they fall apart. The resulting outgoing waves have approximately equal kinetic and potential energy, as expected for traveling waves. It is quite clear from Fig. 3-2, if it was not already clear from Fig. 3-1, that the solitons have been destroyed.

As a stringent check of the accuracy of our time evolution algorithm, we take the final configuration from our simulation, reverse the sign of $\dot{\phi}^i$, and evolve it for the same period of time as we did initially. The second panel of Fig. 3-2 shows the behavior of the energies during this "backwards-in-time" evolution. It is clear that the debris reconstitutes itself into two solitons! The sequence of snapshots of the

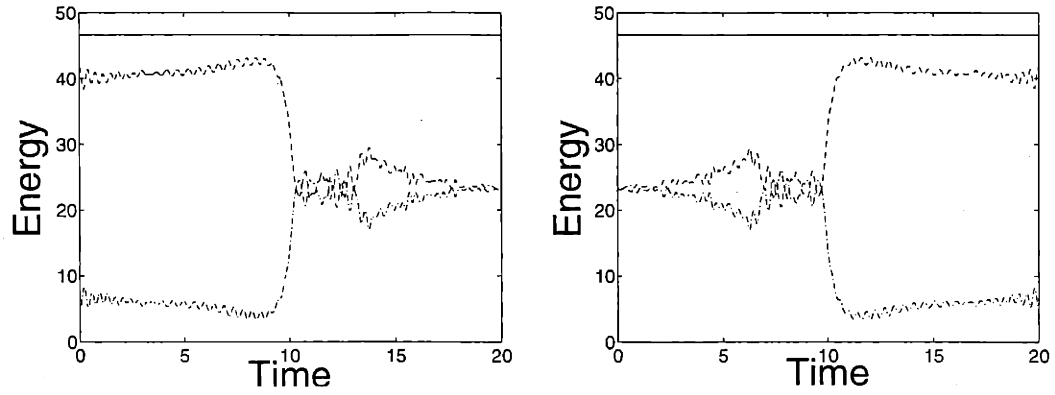

Figure 3-2: Left panel: Kinetic, potential and total energies during the soliton–soliton collision shown in Fig. 3-1 with $\lambda = 10$ and $v = 0.5$. The topmost curve (constant to better than two parts in 10^5) is the total energy. Of the other two curves, the one that begins low is the kinetic energy, the one that begins high is the potential energy, including spatial gradient energy. Right panel: Same, during the time–reversed evolution. We reverse the sign of all $\dot{\phi}^i$ in the final configuration of Fig. 3-1, and then watch the evolution algorithm recreate the initial configuration of Fig. 3-1.

energy density looks almost exactly like those in Fig. 3-1, but in the opposite order in time. The discrepancies between the energy density in the initial configuration and that in the configuration obtained after soliton collision and destruction followed by time-reversed evolution and soliton recreation differ by at most 1/40 of the energy density at the center of the soliton. The total energy is conserved to better than one part in 10^4.

As a further check of the stability of our algorithm, we have also simulated soliton–antisoliton annihilation. We obtain an antisoliton configuration from a soliton configuration by making the transformation $\phi^2 \to -\phi^2$, equivalent to taking $\theta \to -\theta$ in (2.5) or (2.8). This turns a hedgehog configuration into an anti-hedgehog configuration, and hence yields an antisoliton. We find that analyzing soliton–antisoliton collisions using our evolution algorithm is no more difficult than analyzing soliton–soliton collisions. We were able to follow the annihilation process with energy conserved to better than one part in 10^4. Now, with confidence in the accuracy and stability of our evolution algorithm, we proceed to analyze the outcome of soliton–soliton collisions with

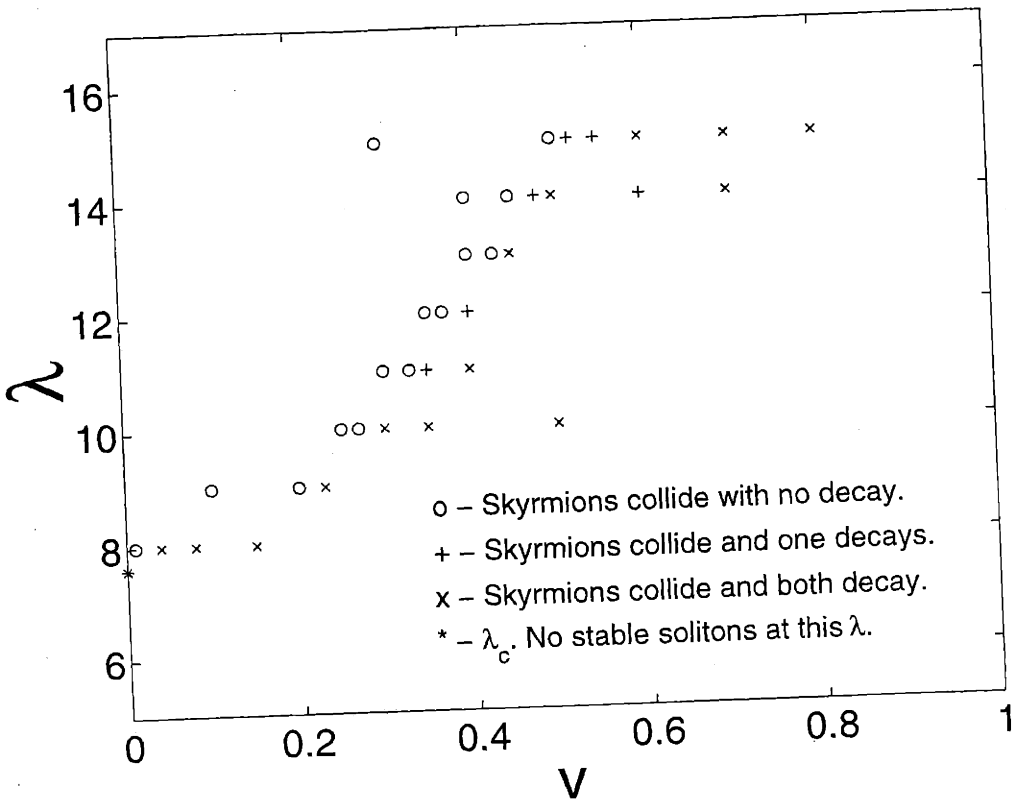

Figure 3-3: Outcome of soliton–soliton collisions with different initial velocities and different values of the parameter λ. All collisions have impact parameter $b = 0$ and relative orientation angle $\alpha = 0$. Note that v is the velocity parameter in (3.1). If v is large enough that relativistic effects are significant, the actual velocity of the soliton is somewhat less than v. For example, $v = 0.8$ yields a soliton with velocity 0.61.

a variety of initial conditions.

We first explore how the outcome of a collision depends on λ and v, keeping the impact parameter $b = 0$ and the relative orientation angle $\alpha = 0$ as above. The results of many simulations are summarized in Fig. 3-3. We discover that for any λ, there is a critical velocity v_c below which the solitons rebound without decaying, and above which one or both (usually both) solitons are destroyed. This critical velocity goes to zero as $\lambda \to \lambda_c$. As λ is increased, v_c increases, reaching about half the speed of light for λ about twice λ_c

We now return to $\lambda = 10$, $v = 0.5$, still keeping $\alpha = 0$ and ask how the outcome of

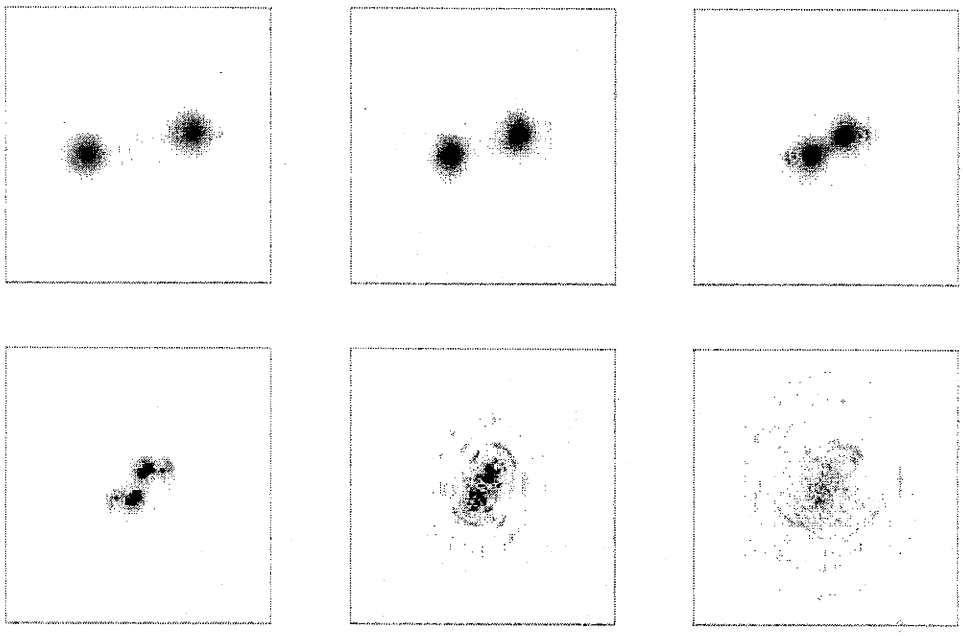

Figure 3-4: Snapshots of energy density during a collision between two solitons with impact parameter $b = 2.0$ in the theory with $\lambda = 10$. The relative orientation angle is $\alpha = 0$. The initial velocity $v = 0.5$ is large enough that the solitons are destroyed. The time between images is 4.0.

a collision depends on the impact parameter b. For $b = 2.0$, both solitons decayed into traveling waves, as we found for $b = 0$ above. We show the outcome of this collision in Fig. 3-4. Note that $b = 2.0$ is a substantial impact parameter, comparable to the soliton radius $R_{sol} \simeq 3$. We find that the solitons still decay if $b = 3.2$. An impact parameter $b = 4.0$, however, yields a collision which is sufficiently peripheral that the solitons emerge intact, deflected from their initial directions of motion by about 45°. We can describe our results by saying that the critical velocity v_c above which soliton decay is the outcome of the collision increases with increasing impact parameter. For $b = 0$, Fig. 3-3 shows that $0.27 < v_c < 0.3$. We now see that $v_c = 0.5$ for a nonzero impact parameter in the range $3.2 < b < 4.0$. We have also done several more simulations with $b = 2.0$ and various initial velocities, and find that for $b = 2.0$, the critical velocity is $0.3 < v_c < 0.4$. We conclude that soliton decay

does not require collisions with small or finely-tuned impact parameters. Although increasing b from zero increases the critical velocity v_c required to destroy the solitons somewhat, it remains easy to destroy solitons as long as the impact parameter is less than or comparable to the soliton radius.

All the collisions we have described to this point have had the same relative orientation. For $\alpha = 0$, low velocity collisions yield a rebound, in which each soliton reverses direction, while higher velocity collisions lead to soliton destruction. We now consider a collision (with $\lambda = 10$, $v = 0.25$ and $b = 0$) between two solitons with a relative orientation angle $\alpha = 180°$. That is, the second soliton in the initial configuration is obtainable from the first by a translation and a 180° rotation. The interaction between static solitons with this orientation is known to be attractive.[19] We show the outcome of a low velocity collision in Fig. 3-5. The work of Ref. [18] reveals that in the $\lambda \to \infty$ theory, there is a stable, ring-shaped, soliton with winding number 2. It appears that the final state of the collision in Fig. 3-5 will be a soliton of this form, although it will differ in its details from that of Ref. [18] since λ is finite. What we observe in Fig. 3-5 is that the incident solitons at first scatter by 90°, but then do not escape to infinity. They fall back upon one another, and rescatter by 90°. There are small outgoing ripples at late time, but they have too little energy density to be visible in Fig. 3-5. We expect that were we to run the simulation for a long time, in a big enough box that outgoing ripples never return, we would see repeated 90° scatterings, with the solitons escaping less and less far away each time, all the while radiating small outgoing ripples, and eventually settling down to become the static, ring-shaped configuration.

As we increase the incident velocity, we find that for $v > v_c$ with $0.43 < v_c < 0.48$, the outcome of the collision is soliton destruction rather than 90° degree scattering followed by the formation of a bound state. We show an example of collision induced decay in a collision with relative orientation $\alpha = 180°$ in Fig. 3-6. Note that the critical velocity above which soliton destruction is the outcome is somewhat larger than, but still comparable to, that we found previously for $\alpha = 0$. We have not mapped out v_c vs. λ for the $\alpha = 180°$ orientation as we did in Fig. 3-3, but we expect

that the figure would be qualitatively similar. One new feature, though, would be that at large λ there would be two different outcomes possible for collisions with $v < v_c$: bound state formation (for low enough v) and 90° scattering followed by the escape of the two intact solitons to infinity (for larger v which is still less than v_c). At $\lambda = 10$, we do not find any velocities for which 90° scattering followed by escape occurs. It must occur at larger λ, since it certainly occurs at large enough velocities for $\lambda \to \infty$, when $v_c \to 1$.

The collision shown in Fig. 3-6 is an example of a simulation in which the initial velocity ($v = 0.5$ in this case) is only just above the critical velocity ($0.43 < v_c < 0.48$ in this case). In this circumstance, what we generically observe is that the solitons scatter, separate a little, but are sufficiently distorted as a result of the scattering that after separating a little they fall apart. We observe this phenomenon also at $\alpha = 0$, except in this case the solitons scatter by bouncing back in the direction whence they came, then separate a little, and then fall apart. At velocities which are somewhat larger than v_c, as for example in the collision shown in Fig. 3-1, we find that soliton destruction occurs more promptly, during the initial collision.

We now consider collisions between solitons with a relative orientation angle $\alpha = 90°$, still with $\lambda = 10$ and $b = 0$. For this relative orientation, there is no force between static solitons.[19] We find the same possible outcomes as we did for $\alpha = 180°$. As a function of increasing velocity, the outcome of a collision is either capture to form the ring-shaped bound state, or soliton destruction. (Again, scattering by an angle of 90° followed by the escape of two intact solitons would be a possibility at larger λ.) The critical velocity above which soliton decay occurs is $0.25 < v_c < 0.3$.

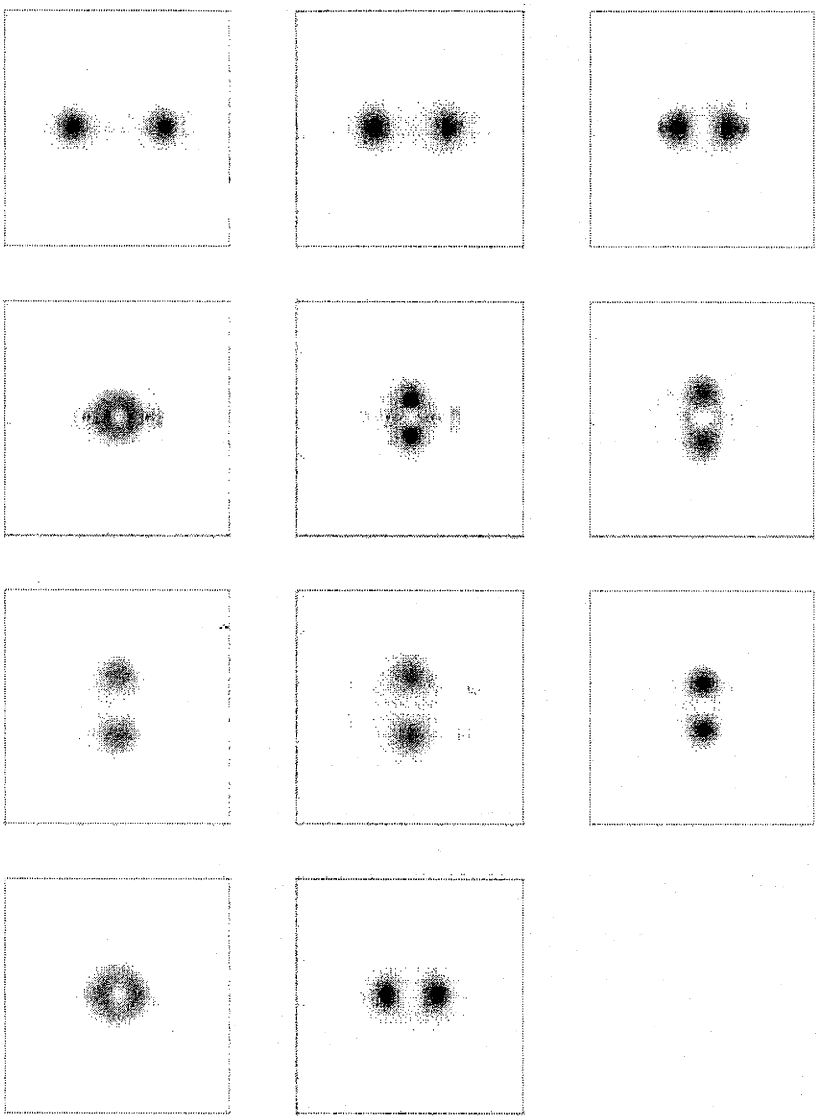

Figure 3-5: Snapshots of energy density during a collision between solitons with relative orientation $\alpha = 180°$, impact parameter $b = 0$, and initial velocity $v = 0.25$ in the theory with $\lambda = 10$. The solitons are not destroyed and (eventually) form a classically stable bound state. The time interval between images varies: the images are at times $t = 0, 4, 8, 12, 16, 20, 24, 28, 34, 42, 50$.

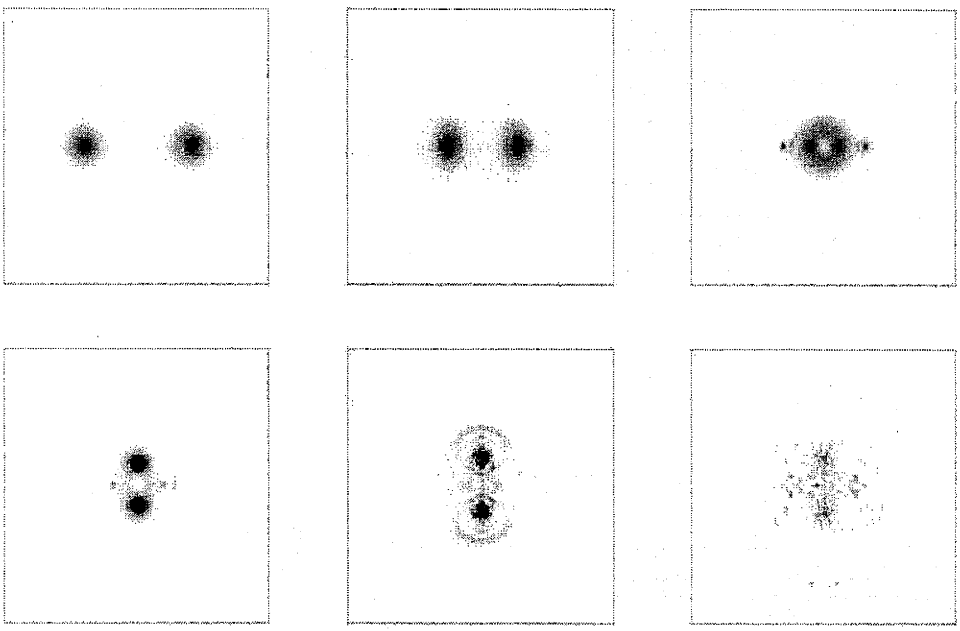

Figure 3-6: Snapshots of energy density during a collision between two solitons with relative orientation $\alpha = 180°$ in the theory with $\lambda = 10$. The impact parameter is $b = 0$. The initial velocity is large enough ($v = 0.5$) that the two solitons decay. The time between each image is 4.0.

Chapter 4

Concluding Remarks

We have analyzed soliton–soliton collisions in a 2+1-dimensional theory with metastable baby skyrmion solutions. We find classically stable soliton solutions for values of the parameter λ which are larger than $\lambda_c \sim 7.6$. These solitons are prevented from decaying by a finite energy barrier and so can decay if supplied with sufficient energy, for example in a collision with a second soliton. We have mapped out the space of initial conditions under which the outcome of a soliton–soliton collision is the destruction of one or both solitons. We find that soliton decay results whenever two solitons collide with an incident velocity greater than some v_c. This critical velocity depends on the parameters in the problem. It goes to zero as $\lambda \to 0$ and the solitons cease to be classically stable. It goes to the speed of light as $\lambda \to \infty$ and the barrier to decay becomes infinite. However, v_c does not rise particularly rapidly with λ: with other parameters chosen as in Fig. 3-3, v_c is only half the speed of light for $\lambda \sim 2\lambda_c$. Thus, soliton destruction does *not* require that the theory have a value of λ lying in some narrow range just above λ_c. The impact parameter b need not be finely tuned either. Not surprisingly, v_c is lowest for collisions with $b = 0$. However, v_c increases by less than a factor of two for b of order the soliton radius. v_c also depends on the relative orientation angle α between the two solitons in the initial state. Here too, the dependence is weak. In the example we explored in detail, we found that as α changes from 0° to 180°, v_c varies between $0.25 < v_c < 0.3$ and $0.43 < v_c < 0.48$. Thus, although v_c does depend on λ and on the parameters other than the velocity needed to

fully specify a choice of initial conditions, the variation of v_c is not dramatic. Soliton decay is not restricted to specially chosen velocities, impact parameters, orientations, or values of λ. Soliton decay is a generic outcome of soliton–soliton collisions.

Our findings motivate future investigation of collisions between metastable solitons in the $3+1$-dimensional electroweak theory. Previous work on two-particle collisions involving these electroweak solitons has focussed on collisions between a W boson and a soliton [7]. In such collisions, the probability for soliton decay falls exponentially as the (rough) analogue of λ is increased above the (rough) analogue of λ_c. This was traced to two facts: First, causing one of these solitons to decay requires delivering sufficient energy to one particular mode of oscillation of the soliton. Second, a generic incident W-boson couples very weakly to the mode which must be energized if decay is to be induced. We find no analogue of this difficulty in our analysis of soliton–soliton collisions in $2+1$ dimensions. If there is a particular mode which must be excited, then soliton–soliton collisions seem to generically deliver energy to this mode. And, we certainly see no evidence of soliton decay being restricted to theories with $|\lambda - \lambda_c| \ll \lambda_c$. This suggests that collisions between two TeV scale particles which can be modeled as electroweak solitons (rather than between one W-boson and one such particle) may be an arena in which two-particle collisions generically lead to baryon number violation. As we stressed in the Introduction, however, the metastable baby skyrmions we analyze differ in several important qualitative respects from metastable electroweak solitons. Furthermore, our analysis has been purely classical whereas the analysis of W-soliton collisions in Ref. [7] is quantum mechanical. Although our results motivate an analysis of collisions between electroweak solitons, they should not be taken to provide even qualitative guidance as to the outcome of such a study.

Appendix A

Derivation of the Equations of Motion

A.1 Discretization Methods

In order to perform simulations of continuous systems, it is essential to develop a corresponding discrete model. This discrete model no longer acts on continuous fields in space, but on a discrete array of values referred to as a lattice. In our case we began with a model intended to describe the evolution of three scalar fields in a 2-dimensional space. Therefore, we construct a 2-dimensional grid of points and evolve the field values at only these locations. The hope is that as long as the fields do not vary highly between neighboring points, the dynamics of this discrete system is a good approximation to the continuous system.

The next step is to determine how this new discrete system will evolve in time. In a classical continuous system, the value of a field $\phi(x, y)$ and its current derivatives in space and time at (x, y) completely determine it's value at an infinitesimal time δt later[1].

$$\phi(t + \delta t, x, y) = f(t, x, y, \phi(t, x, y), \partial_\alpha \phi(t, x, y)). \tag{A.1}$$

[1]This also requires that there is no *action at a distance*, or that the values of the fields and its derivatives away from the point (x, y) have no bearing on the dynamics.

In moving to the discrete system we give up the knowledge of the local spatial derivatives, since we do not know the field values in the infinitesimal neighborhood of the point (x, y). As a substitute, we assemble approximations to the spatial derivatives at (x, y) from the values of ϕ at (x, y) and at the neighboring lattice sites. This process is referred to as *finite-differencing*. For example a spatial derivative of the field ϕ at a location (x, y) could be approximated as

$$\frac{\partial\phi(x,y)}{\partial x} \rightarrow \frac{\phi(x,y) - \phi(x-\Delta x, y)}{\Delta x}. \tag{A.2}$$

Of course, the choice of approximation is not unique. One could just as well choose to approximate the derivative in Eq. A.2 as

$$\frac{\partial\phi(x,y)}{\partial x} \rightarrow \frac{\phi(x+\Delta x, y) - \phi(x-\Delta x, y)}{2\Delta x}. \tag{A.3}$$

The choice of discretization is determined by the form which minimizes the error obtained by approximating derivatives with finite differences. The first choice gives an estimate of $\partial_x\phi$ to order $(\Delta x)^2$ at the location $x = x + \frac{\Delta x}{2}$. The fact that this is not located at $x = x$ is not a problem when discretizing terms like $(\partial_x\phi)^2$, since in the equations of motion this will be of the form $\partial_x^2\phi$. You can approximate this second derivative as the difference of the first derivatives located at $x = x - \frac{\Delta x}{2}$ and $x = x + \frac{\Delta x}{2}$. This difference will be accurate to $O((\Delta x)^2)$ at the point $x = x$. When your Lagrangian density contains terms of the form $\partial_x\phi\partial_y\phi$, you will not have this benefit. You should therefore use the symmetric form of Eq. A.3. The error of this estimate is $O((2\Delta x)^2)$ for the derivative at $x = x$. For terms of this form this is better than Eq. A.2, whose error is $O(\Delta x)$ at $x = x$.

The next discretization problem enters the picture due to the fact that we do not want to be restricted to infinitesimal steps in time as we evolve our system. In general, we would like to take a finite step Δt forward in time. The larger step we take, the faster we can evolve the system to achieve our desired results. On the other hand, a larger step in time introduces more discrepancies between the discrete and continuous models. It then becomes a balance of determining the largest timestep

whose resulting dynamics deviate negligibly from those of the continuous model.

It is common to use the same discretization method for time derivatives as used for the spatial derivatives. To do this one would store the value of the field at a previous time, and use the above discretizations, only now in time, to approximate time derivatives. If a discretization of a time derivative is chosen such that it involves $\phi(t + \Delta t)$ at some point in space, you would then solve your equations of motion for this *future* value of ϕ. In our case, we encountered instabilities when using this method for performing the time evolution, and therefore did not discretize the time derivatives as such.

An alternate method to deal with the time derivatives is to store a value of $\dot{\phi}$ at each lattice point as opposed to $\phi(t - \Delta t)$. One then solves the equations of motion for $\ddot{\phi}$ and uses this information to take steps forward in time. This is the method we used to perform our simulations. Furthermore, for dealing with mixed time-space derivatives, we discretized them as

$$\frac{\partial^2 \phi(x,y)}{\partial x \partial t} \rightarrow \frac{\dot{\phi}(x + \Delta x, y) - \dot{\phi}(x - \Delta x, y)}{2\Delta x}. \tag{A.4}$$

After understanding the methods of finite-differencing, there is still an ambiguity as to when to apply them. The easier method is to apply the Euler-Lagrange equations for continuous systems to the Lagrangian density, thereby obtaining continuous equations of motion. One would then use these finite-differencing techniques to implement these equations on a discrete lattice. The problem with this is that to evaluate the energy of your system using the Lagrangian or Hamiltonian, you would have to discretize these in a different manner than the equations of motion. This can lead to energy conservation problems when performing time-evolution simulations. In the case of our skyrmions, this discrepancy actually resulted in numerical instabilities when attempting to simulate violent dynamics.

Therefore, we had to discretize the model before solving for the equations of motion. To do this we discretized the Lagrangian density and summed the resulting terms on a lattice of 125×125 points. Since our model involves 3 fields, we had 3 ϕ^i

and 3 $\dot{\phi}^i$ at each lattice site, resulting in a system with a total of $2N = 6 \times 125 \times 125$ variables. This in turn gave us a true Lagrangian for our discretized system of N variables. From this we obtain a set of $N = 3 \times 125 \times 125$ equations of motion by applying the Euler-Lagrange equation to each ϕ^i. To do the time evolution it is necessary to solve these equations for the three $\ddot{\phi}^i$ at each lattice site. Luckily, this set of equations is easy to solve explicitly, since only the three $\ddot{\phi}^i$ that are at the same point in space are coupled. We will now go through the above process explicitly.

A.2 Discretized Lagrangian

Let us look again at our Lagrangian density,

$$\mathcal{L} = F\left[\frac{1}{2}\partial_\alpha\vec{\phi}\cdot\partial^\alpha\vec{\phi} \;-\; \frac{\kappa^2}{4}(\partial_\alpha\vec{\phi}\times\partial_\beta\vec{\phi})\cdot(\partial^\alpha\vec{\phi}\times\partial^\beta\vec{\phi}) \right.$$
$$\left. -\; \mu^2(v-\vec{n}\cdot\vec{\phi}) - \lambda\left(\vec{\phi}\cdot\vec{\phi}-v^2\right)^2\right]. \qquad \text{(A.5)}$$

Note that as described in Section 1.2 we choose $\vec{n}$ to be $(0,0,1)$ and $F = \kappa = v^2 = 1$. Before we could discretize it, it was necessary to expand the first two terms. So considering a single point (x,y), the first term in the Lagrangian density would be

$$\frac{1}{2}\left[(\dot{\phi}^i(x,y))^2 - (\partial_x\phi^i(x,y))^2 - (\partial_y\phi^i(x,y))^2\right], \qquad \text{(A.6)}$$

where i is summed over the three fields $(1,2,3)$. We discretize the spatial terms in the asymmetric fashion of Eq. A.2, and the time derivative term we leave as $\dot{\phi}^i(x,y)\cdot\dot{\phi}^i(x,y)$.

The Skyrme term is not as easy. In summation notation it can be written as

$$(\partial_\alpha\vec{\phi}\times\partial_\beta\vec{\phi})\cdot(\partial^\alpha\vec{\phi}\times\partial^\beta\vec{\phi}) = \epsilon_{lmn}\epsilon_{lpq}\partial_\alpha\phi^m\partial_\beta\phi^n\partial^\alpha\phi^p\partial^\beta\phi^q. \qquad \text{(A.7)}$$

By applying the identity $\epsilon_{lmn}\epsilon_{lpq} = \delta_{mp}\delta_{nq} - \delta_{mq}\delta_{np}$, this can be expanded further to

obtain

$$\begin{aligned}\epsilon_{lmn}\epsilon_{lpq}\partial_\alpha\phi^m\partial_\beta\phi^n\partial^\alpha\phi^p\partial^\beta\phi^q &= (\delta_{mp}\delta_{nq}-\delta_{mq}\delta_{np})(\partial_\alpha\phi^m\partial_\beta\phi^n\partial^\alpha\phi^p\partial^\beta\phi^q)\\ &= \partial_\alpha\phi^m\partial^\alpha\phi^m\partial_\beta\phi^n\partial^\beta\phi^n\\ &\quad -\partial_\alpha\phi^m\partial^\alpha\phi^n\partial_\beta\phi^n\partial^\beta\phi^m. \end{aligned} \tag{A.8}$$

Of course, all of these terms are evaluated at the point (x, y).

The Skyrme term is not as easy to discretize due to its four-derivative structure (Eq. A.8). To ensure that discretization errors are of order $2(\Delta x)^2$, it is necessary that each derivative is centered about the same point in space. Therefore symmetric derivatives of the form of Eq. A.3 are used for this term. The remaining two terms of the Lagrangian density are just explicit functions of the ϕ^i, and thereby pose no difficulty. To obtain the full Lagrangian L for the lattice we just sum this discretized Lagrangian $\mathcal{L}(x, y)$ over the entire 125×125 lattice.

A.3 Euler-Lagrange Equations

Given each coordinate $\phi^i(x_n, y_m)$ and its respective conjugate momentum $\dot{\phi}^i(x_n, y_m)$, we obtain an equation of motion by applying the Euler-Lagrange equation

$$\frac{\partial L}{\partial\phi^i(x_n,y_m)} - \frac{d}{dt}\frac{\partial L}{\partial\dot{\phi}^i(x_n,y_m)} = 0. \tag{A.9}$$

To simplify the problem, let us divide the Lagrangian into two parts; L_{nm} which is dependent on $\phi^i(x_n, y_m)$ and $\dot{\phi}^i(x_n, y_m)$ at a given point (x_n, y_m) , and L_{in} which is independent of these terms;

$$L = L_{nm}(\phi^i(x_n, y_m), \dot{\phi}^i(x_n, y_m)) + L_{in}. \tag{A.10}$$

This will simplify matters since L_{in} will make no contributions to the equations of

motion at the site (x_n, y_m) due to the fact that

$$\begin{aligned} \frac{\partial L_{in}}{\partial \phi^i(x_n, y_m)} &= 0 \\ \frac{\partial L_{in}}{\partial \dot{\phi}^i(x_n, y_m)} &= 0. \end{aligned} \tag{A.11}$$

It now just becomes a problem of picking out every term in the full Lagrangian L which involves $\phi^i(x_n, y_m)$ and $\dot{\phi}^i(x_n, y_m)$. We can step through the terms of L, analyzing them by which term of the Lagrangian Density $\mathcal{L}$ they arose from. The first of these groups is terms of the form $\partial_\alpha \vec{\phi} \cdot \partial^\alpha \vec{\phi}$. We discretized these terms in the asymmetric fashion. Let us begin by looking at these finite-differences pictorially. In Figure A-1 one can see the four asymmetric finite-differences that contain $\phi^i(x_n, y_m)$;

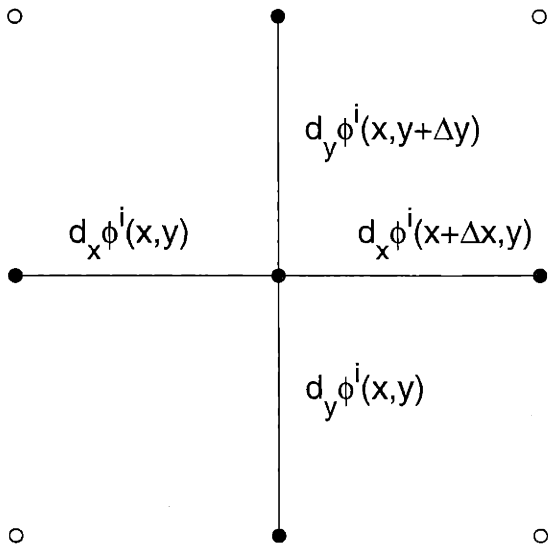

Figure A-1: Figure showing the asymmetric derivatives within the Lagrangian containing $\phi^i(x, y)$ (central dot). Circles are lattice sites in our 2-D space and solid dots connected by lines show the finite-differences which contain the term of interest.

$\partial_x \phi^i(x_n, y_m)$, $\partial_y \phi^i(x_n, y_m)$, $\partial_x \phi^i(x_{n+1}, y_m)$, and $\partial_y \phi^i(x_n, y_{m+1})$. The only place asymmetric finite-differences occur in our full Lagrangian L is in the terms coming from

the first term in our Lagrangian density, $\partial_\alpha \vec{\phi} \cdot \partial^\alpha \vec{\phi}$. Therefore, the terms

$$\begin{aligned} &-(\partial_x \phi^i(x_n, y_m))^2 \\ &-(\partial_y \phi^i(x_n, y_m))^2 \\ &-(\partial_x \phi^i(x_{n+1}, y_m))^2 \\ &-(\partial_y \phi^i(x_n, y_{m+1}))^2 \end{aligned} \tag{A.12}$$

will contain $\phi^i(x_n, y_m)$, and therefore will occur in L_{nm}. Furthermore, we must not forget $\dot{\phi}^i(x_n, y_m)$ will occur in this category of terms as well.

$$(\dot{\phi}^i(x_n, y_m))^2 \tag{A.13}$$

will also occur in L_{nm} due to the first term of the Lagrangian density.

The Skyrme term, on the other hand, is not as easy to see. We know that since we discretized this term using symmetric derivatives, only these will occur in the Skyrme term. Figure A-2 summarizes the symmetric terms containing $\phi^i(x_n, y_m)$. As in the previous case, one must not forget that $\dot{\phi}^i(x_n, y_m)$ will also appear in the Skyrme term.

Although we still have only five elements we must look for within the Skyrme term,

$$\begin{aligned} &\partial_x \phi^i(x_{n-1}, y_m) \\ &\partial_y \phi^i(x_n, y_{m-1}) \\ &\partial_x \phi^i(x_{n+1}, y_m) \\ &\partial_y \phi^i(x_n, y_{m+1}) \\ &\dot{\phi}^i(x_n, y_m) \end{aligned} \tag{A.14}$$

each will appear multiple times in the various permutations of Eq. A.8. Any of $\partial_\alpha \phi^m$, $\partial_\alpha \phi^n$, $\partial_\beta \phi^m$ or $\partial_\beta \phi^n$ of Eq. A.8 could take on the value of any of the five terms in (A.14). This leads to quite a number of terms. In fact, after simplification and the

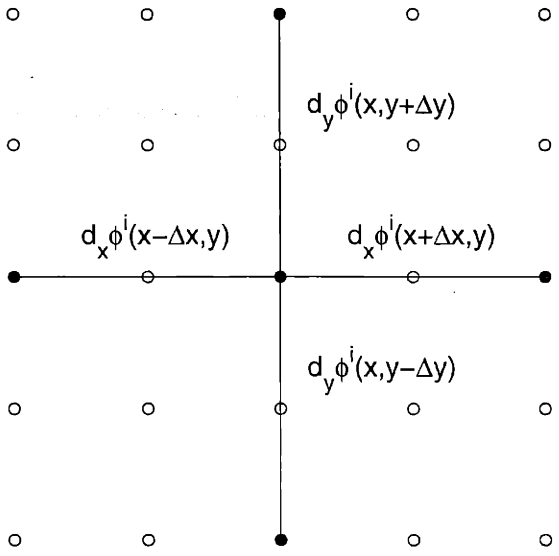

Figure A-2: Figure showing the symmetric derivatives within the Lagrangian containing $\phi^i(x,y)$ (central dot). Circles are lattice sites in our 2-D space and solid dots connected by lines show the finite-differences which contain the term of interest.

grouping of terms, the Skyrme term adds the following 20 terms to L_{nm},

$$-(\dot{\boldsymbol{\phi}}^{\mathbf{i}}(\mathbf{x_n},\mathbf{y_m}))^{\mathbf{2}}(\partial_x\phi^j(x_n,y_m))^2$$

$$-(\boldsymbol{\partial_x\phi}^{\mathbf{i}}(\mathbf{x_{n-1}},\mathbf{y_m}))^{\mathbf{2}}(\dot{\phi}^j(x_{n-1},y_m))^2$$

$$-(\boldsymbol{\partial_x\phi}^{\mathbf{i}}(\mathbf{x_{n+1}},\mathbf{y_m}))^{\mathbf{2}}(\dot{\phi}^j(x_{n+1},y_m))^2$$

$$\dot{\boldsymbol{\phi}}^{\mathbf{i}}(\mathbf{x_n},\mathbf{y_m})\partial_x\phi^i(x_n,y_m)\dot{\phi}^j(x_n,y_m)\partial_x\phi^j(x_n,y_m)$$

$$\dot{\phi}^i(x_{n-1},y_m)\boldsymbol{\partial_x\phi}^{\mathbf{i}}(\mathbf{x_{n-1}},\mathbf{y_m})\dot{\phi}^j(x_{n-1},y_m)\partial_x\phi^j(x_{n-1},y_m)$$

$$\dot{\phi}^i(x_{n+1},y_m)\boldsymbol{\partial_x\phi}^{\mathbf{i}}(\mathbf{x_{n+1}},\mathbf{y_m})\dot{\phi}^j(x_{n+1},y_m)\partial_x\phi^j(x_{n+1},y_m)$$

$$-(\dot{\boldsymbol{\phi}}^{\mathbf{i}}(\mathbf{x_n},\mathbf{y_m}))^{\mathbf{2}}(\partial_y\phi^j(x_n,y_m))^2$$

$$-(\boldsymbol{\partial_y\phi}^{\mathbf{i}}(\mathbf{x_n},\mathbf{y_{m-1}}))^{\mathbf{2}}(\dot{\phi}^j(x_n,y_{m-1}))^2$$

$$-(\boldsymbol{\partial_y\phi}^{\mathbf{i}}(\mathbf{x_n},\mathbf{y_{m+1}}))^{\mathbf{2}}(\dot{\phi}^j(x_n,y_{m+1}))^2$$

$$\dot{\boldsymbol{\phi}}^{\mathbf{i}}(\mathbf{x_n},\mathbf{y_m})\partial_y\phi^i(x_n,y_m)\dot{\phi}^j(x_n,y_m)\partial_y\phi^j(x_n,y_m)$$

$$\dot{\phi}^i(x_n,y_{m-1})\boldsymbol{\partial_y\phi}^{\mathbf{i}}(\mathbf{x_n},\mathbf{y_{m-1}})\dot{\phi}^j(x_n,y_{m-1})\partial_y\phi^j(x_n,y_{m-1})$$

$$\dot{\phi}^i(x_n,y_{m+1})\boldsymbol{\partial_y\phi}^{\mathbf{i}}(\mathbf{x_n},\mathbf{y_{m+1}})\dot{\phi}^j(x_n,y_{m+1})\partial_y\phi^j(x_n,y_{m+1})$$

$$
\begin{aligned}
&(\boldsymbol{\partial_x \phi^i(x_{n-1}, y_m)})^2 (\partial_y \phi^j(x_{n-1}, y_m))^2 \\
&(\boldsymbol{\partial_x \phi^i(x_{n+1}, y_m)})^2 (\partial_y \phi^j(x_{n+1}, y_m))^2 \\
&(\boldsymbol{\partial_y \phi^i(x_n, y_{m-1})})^2 (\partial_x \phi^j(x_n, y_{m-1}))^2 \\
&(\boldsymbol{\partial_y \phi^i(x_n, y_{m+1})})^2 (\partial_x \phi^j(x_n, y_{m+1}))^2 \\
&-\boldsymbol{\partial_x \phi^i(x_{n-1}, y_m)} \partial_y \phi^i(x_{n-1}, y_m) \partial_x \phi^j(x_{n-1}, y_m) \partial_y \phi^j(x_{n-1}, y_m) \\
&-\boldsymbol{\partial_x \phi^i(x_{n+1}, y_m)} \partial_y \phi^i(x_{n+1}, y_m) \partial_x \phi^j(x_{n+1}, y_m) \partial_y \phi^j(x_{n+1}, y_m) \\
&-\partial_x \phi^i(x_n, y_{m-1}) \boldsymbol{\partial_y \phi^i(x_n, y_{m-1})} \partial_x \phi^j(x_n, y_{m-1}) \partial_y \phi^j(x_n, y_{m-1}) \\
&-\partial_x \phi^i(x_n, y_{m+1}) \boldsymbol{\partial_y \phi^i(x_n, y_{m+1})} \partial_x \phi^j(x_n, y_{m+1}) \partial_y \phi^j(x_n, y_{m+1}), \qquad \text{(A.15)}
\end{aligned}
$$

where the terms in bold contain either $\phi^i(x_n, y_m)$ or $\dot{\phi}^i(x_n, y_m)$ and $j \in \{1, 2, 3 | \neq i\}$

This leaves only the last two parts of the Lagrangian, the μ^2 and λ terms. The former will contribute the term

$$-\mu^2(1 - \phi^3(x_n, y_m)) \qquad \text{(A.16)}$$

to L_{nm}, but only for $i = 3$. The latter will contribute the term

$$-\lambda((\phi^1(x_n, y_m))^2 + (\phi^2(x_n, y_m))^2 + (\phi^3(x_n, y_m))^2 - 1)^2. \qquad \text{(A.17)}$$

Therefore, each Euler-Lagrange equation now correlates 26 (ϕ^1,ϕ^2 equations) or 27 (ϕ^3 equations) terms from the full Lagrangian. After allowing the index j to take all its possible values, the number of terms increases to greater than 50.

A.4 Solving the System

Now, all that is required is to solve the entire system of $N = 3 \times 125 \times 125$ equations of 50+ terms each for the N $\ddot{\phi}^i$. Luckily, we found that only $\ddot{\phi}^i$ at the same lattice site occurred in a single Euler-Lagrange equation. The equations at a single lattice site (x_n, y_m) simplified to the form

$$
\begin{aligned}
0 &= a_1 + b_1\ddot{\phi}^1(x_n, y_m) + c_1\ddot{\phi}^2(x_n, y_m) + d_1\ddot{\phi}^3(x_n, y_m) \\
0 &= a_2 + b_2\ddot{\phi}^2(x_n, y_m) + c_2\ddot{\phi}^3(x_n, y_m) + d_2\ddot{\phi}^1(x_n, y_m) \\
0 &= a_3 + b_3\ddot{\phi}^3(x_n, y_m) + c_3\ddot{\phi}^1(x_n, y_m) + d_3\ddot{\phi}^2(x_n, y_m). \qquad \text{(A.18)}
\end{aligned}
$$

One further simplification comes from the fact that $c_1 = d_2$, $c_2 = d_3$, and $c_3 = d_1$. Even more helpful was the fact that this system of equations is linear can be explicitly solved for $\ddot{\phi}^i$. The solution is

$$
\begin{aligned}
\ddot{\phi}^1(x, y) &= -\frac{a_3c_2c_1 - a_3b_2c_3 + a_1b_2b_3 - c_1a_2b_3 - c_2^2a_1 + c_2a_2c_3}{-c_2^2b_1 + 2c_2c_3c_1 - c_1^2b_3 + b_2b_1b_3 - b_2c_3^2} \\
\ddot{\phi}^2(x, y) &= -\frac{c_2c_3a_1 + c_2b_1a_3 + c_1b_3a_1 - a_2b_1b_3 + a_2c_3^2 - c_1a_3c_3}{-c_2^2b_1 + 2c_2c_3c_1 - c_1^2b_3 + b_2b_1b_3 - b_2c_3^2} \\
\ddot{\phi}^3(x, y) &= -\frac{b_1a_3b_2 - b_1c_2a_2 - a_1c_3b_2 - a_3c_1^2 + c_1c_3a_2 + c_2c_1a_1}{-c_2^2b_1 + 2c_2c_3c_1 - c_1^2b_3 + b_2b_1b_3 - b_2c_3^2}. \qquad \text{(A.19)}
\end{aligned}
$$

Now that we can generate $\ddot{\phi}^i$ for all the lattice sites, we can apply any of the traditional numerical methods for evolving our system of ϕ^is and $\dot{\phi}^i$s. We describe our choice in Section B.5.

Appendix B

The Numerical Model

B.1 Data Structure

For the simulations in this paper all the code was written in C. For compatibility with the functions used from Ref. [21], the lattice was a simple linear array of double-precision floating point numbers. This array had $N = 3 \times 125 \times 125$ elements for the relaxation stage and $2N$ elements for the time-evolution. Although all the data was stored in a linear array, the elements have a natural multi-dimensionality to them. One could identify each element of the lattice array $\phi()$ with 4 integer indices (i, j, k, g): x-lattice-position, y-lattice-position, field, and time derivative. The first two take a value from zero to $n-1$, where n is equal to the number of sites along one edge of the lattice. These two integers just index the lattice site to which an element belongs. The third index takes a value of 0, 1 or 2 and allows you to select one of the three fields. The last index has two values $(0, 1)$ and chooses between ϕ and $\dot{\phi}$. This last index is always 0 during the relaxation stage, which involves finding the static soliton configuration of ϕ^i.

So given the fact that we are using a linear array to store the numbers, one must determine the location for each object given that they naturally have four indices. For this reason we have two functions grid_stor and grid_val which handle this for us.

```
void grid_stor(double grid[], int i, int j, int k, int g,
```

```
        double newval)
double grid_val(double grid[], int i, int j, int k, int g)
```

The first function takes a pointer to a grid and the value `newval` and inserts it into the grid at the correct place using the indices `i,j,k,g`. The second function takes a grid and all the indices, and returns the value at the specified location. By using these two functions, we no longer need worry about the exact order in which the data is stored in the linear array.

Two more functions complement the above. The functions

```
void put_datfile(double v[])
int get_datfile(double v[])
```

allow us to save and recall the grid data structure to and from the filesystem. The first takes in a pointer to a current grid and the prints the data in binary format to a file. The file is given a standard name generated from the current date and time. The current naming system is not sufficient given the fact that two files output in the same minute will be given identical names. This results in the first file being overwritten by the second. The second function takes in a pointer to an empty grid and fills it with the data in a file named `initcond.bd`. This was sufficient for my purposes, but it would be better to design this function to take in a arbitrary filename as an argument.

B.2 Altering the Fields

Given the data structure and methods for accessing it, we need functions that alter the fields in a specific way. The first of these is init_cond

```
void init_cond( double vstart[] )
```

This function takes in a pointer to a grid and assigns values to each location based on an algorithm of your choice. The two initial conditions we have used are a vacuum and an approximated $\lambda = \infty$ Baby Skyrmion located at the center of the lattice.

The next function of use in transforming the field is translate:

```
int trans(double v[], double vout[], int xdis)
```

This function takes in a pointer to a current grid, a pointer to an empty grid, and an integer `xdis`. It then reads out the current grid and inserts the value into the new grid displaced by `xdis` lattice sites in the x direction. This results in some lattice sites in the new grid having no assigned values. Therefore, this function fills these sites with the values taken from the border or the old grid, extending these edge values until it fills the new grid.

The next function is rotate,

```
int rotate(double v[], double vout[])
```

which, similar to translate, takes in pointers to a current grid and an empty grid. It then assigns values to the new grid by rotating the old square grid by 90^o clockwise about the center of the lattice.

Another useful function for acting on a single lattice is antisol,

```
int antisol(double v[], double vout[])
```

The effect of this function is that given a Baby Skyrmion field configuration, it will change it to an anti-soliton. The method by which it does this is by copying the exact fields from the old grid to the new grid with one minor change. The signs on all the values of ϕ^2 on the lattice are reversed, i.e. $\phi^2 \rightarrow -\phi^2$.

The next function is boost,

```
int boost(double v[], double vel)
```

which takes in a grid and assigns new values to the time derivatives by the equation

$$\dot{\phi}^i(x,y) = -v[\phi^i(x,y) - \phi^i(x-\Delta x, y)]/\Delta x. \tag{B.1}$$

This is a simple Galilean transformation.

The final function for altering the grid is one of the most useful. So far using the above functions, we can create a single soliton configuration and alter it in several fashions. To do soliton collisions though, we need more than one soliton with different locations, velocities, etc. We achieve this with addgrid,

```
int addgrid(double v1[], double v2[], double vout[])
```

This function merges two grids into a third grid defined by $\vec{\phi}_{\text{vacuum}} + (\vec{\phi}_{\text{first soliton}} - \vec{\phi}_{\text{vacuum}}) + (\vec{\phi}_{\text{second soliton}} - \vec{\phi}_{\text{vacuum}})$.

B.3 Visualization functions

This next category of functions do not alter the fields in any fashion, but allow us to read out certain numerical qualities about the fields. In essence, they process the data within the grid and output it in a form that we can understand. The simplest example of this is the snapshot function

```
void snapshot( double v[] )
```

which takes in a pointer to a grid. It then prints all the data within the fields to the standard output (`stdout`). It prints it in a matrix format suitable for loading into Matlab. The resolution at which it outputs the fields can be set with the global `snap_res`. By default is set to 1, which outputs every lattice site. Note that when this function acts on a single 125×125 grid and outputs the data in every lattice site to 6 decimal places, it produces about 1MB of output.

The next function produces more compact data about the fields.

```
void get_energy( double v[] )
```

This function takes in a pointer to a grid and computes the potential and kinetic energy as well as the winding number of the fields. It then stores these values in the global array `ener_dat` in the above order.

Closely related to this idea is the function `ener_surf`

```
void get_enersurf( double v[], double ev[])
```

This takes in a pointer to the current grid, and a pointer to an empty grid. It then computes the potential, kinetic and total energies of each lattice site, storing these values in the empty grid in the place of the three fields. This new grid is then passed on to a modified snapshot function

```
void esnapshot( double v[] )
```

which only prints out these fields.

B.4 Relaxation Method

In order to find the correct shape and energy of the Baby Skyrmions, we begin with an estimated field configuration and relax it into a minimum of the potential energy. We do this using an algorithm specifically designed to minimize a function of multiple variables. In our case, we would like to minimize the potential energy, which is a function of $N = 3 \times 125 \times 125$ variables (i.e. the three fields values at each lattice site). There are many algorithms designed for this purpose, some of which are outlined in Ref. [21]. The basic principle behind most of them is to choose a particular direction in your N-dimensional phase space and then do a line minimization along that direction. After the minimum along that direction is found, a new direction is chosen and the process is repeated. The questions then comes down to how to choose the direction along which to minimize at each step. The most obvious is called the *steepest descent method.* This requires the ability to compute the local gradient at any point of the multidimensional function you wish to minimize, and then to use this as the direction for the line minimization. Luckily, this gradient information is given to us by the terms in our equations of motion which contain no time derivatives. The problem here is that the *steepest descent method* is not very efficient. By always choosing to minimize in the direction of the local gradient, you tend to undo the minimization you have done along other directions in phase space.

This is where the idea of conjugate directions comes into play. After minimizing along the direction $\vec{a}$ in phase space, we choose the new direction of minimization $\vec{b}$ such that the local gradient remains perpendicular to $\vec{a}$ as we minimize. This guarantees that we need not redo our minimization along $\vec{a}$. $\vec{a}$ and $\vec{b}$ are referred to as *conjugate directions.* If this process is repeated, it should only require on the order of N separate line minimizations to reach the local minimum of our function. The application of this method is fully outlined in Ref. [21] in the section called

Conjugate Gradient Methods in Multidimensions. The function which performs this multidimensional minimization is

```
void dfrprmn(double p[], int n, double ftol, int *iter,
        double *fret,
        double (*func)(double []),
                    void (*dfunc)(double [], double []))
```

This is an exact copy of the function `frprmn` taken from Ref. [21] except that all the variables are changed from single-precision to double-precision floating point numbers. The function takes in the initial guess `p`, an array of `n` variables, and minimizes the function `func`. The function `dfunc` takes in a pointer to the current location in phase space and fills an empty array with the gradient at that location. The minimization continues until the change of `func` is less than `ftol` between successive line minimizations. The total number of iterations is stored in the variable `iter`, and the final minimum of the function is stored in `fret`. In our case, `p` is our approximated field configuration of a Baby Skyrmion, `func` is the potential energy function, and `dfunc` are the equations of motion with the time derivative terms removed.

Another reason this algorithm was chosen over the other possible candidates was that the number of variables it stores is on the order of N as opposed to other algorithms which store N^2. Given the fact that our data sets are already quite large, the other methods would require too much space, forcing us to use inefficient work-arounds. The conjugate gradient method on the other hand has the drawback that one must be able to compute the gradient at all points in phase space. This was not an extra burden in our case since we already had the gradient information provided by the equations of motion.

B.5 Time Evolution Algorithm

The idea behind all time evolution algorithms is to take some configuration $\vec{y}_n$ in phase space and advance it some time Δt to a new configuration $\vec{y}_{n+1}$. The simplest way of doing this is *Euler's method* where we advance a function using the equation

$$\vec{y}_{n+1} = \vec{y}_n + \Delta t \vec{y'}_n \tag{B.2}$$

This is generally not used since it is not very accurate. A slight modification of this result in Runge-Kutta methods, where the information gained from multiple smaller steps are used to fit a Taylor expansion of the function. In general this provides for a much more accurate time evolution. The most commonly used example of this is the *fourth-order Runge-Kutta formula*,

$$\begin{aligned} k_1 &= \Delta t y'(t_n, y_n) \\ k_2 &= \Delta t y'(t_n + \frac{\Delta t}{2}, y_n + \frac{k_1}{2}) \\ k_3 &= \Delta t y'(t_n + \frac{\Delta t}{2}, y_n + \frac{k_2}{2}) \\ k_4 &= \Delta t y'(t_n + \Delta t, y_n + k_3) \\ y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O((\Delta t)^5). \end{aligned} \tag{B.3}$$

here presented in one dimension. This method disregards terms higher than fourth order in the timestep Δt. Therefore, as long as your function is well-behaved (i.e. has small higher-order terms in its Taylor expansion) one can achieve a desired accuracy by choosing a sufficiently small timestep.[1]

The next optimization one can apply to the problem of time evolution then comes directly from this issue of sufficiently small timestep. The requirements on the size of the timestep of an average time evolution problem can vary from step to step. At certain times the evolution may go smoothly and a larger timestep is good enough, but you cannot increase it since it it not small enough for a few key moments during the simulation. You could keep it small, but you then have a large loss in time as it uses an extremely small timestep when it is not necessary. This is where adaptive timestep algorithms enter the picture. The basic concept for these methods is to change the timestep as needed, evolving the simple periods quickly, while focusing computation time on the trickier parts. This is implemented by estimating the error

[1] Of course this has a lower limit based on a computer's truncation error.

after a step is taken. If this estimate is higher than some allowed value, then the step is rewound and repeated with a smaller timestep. If the estimate is smaller than the allowed value, then the step is accepted and a new larger timestep is chosen based on the ratio of the error estimate to the allowed error.

The problem then boils down to estimating the error of the results of a step forward in time. In the Adaptive Timestep method we used [21], an *embedded Runge-Kutta formula* was used to estimate the error. It begins with a fifth-order Runge-Kutta equation of the form

$$y_{n+1} = y_n + c_1 k_1 + c_2 k_2 + c_3 k_3 + c_4 k_4 + c_5 k_5 + c_6 k_6 + O((\Delta t)^6). \tag{B.4}$$

where the individual components are intermediate steps of the form

$$\begin{aligned} k_1 &= \Delta t y'(t_n, y_n) \\ k_2 &= \Delta t y'(t_n + a_2 \Delta t, y_n + b_{21} k_1) \\ &\ldots \\ k_6 &= \Delta t y'(t_n + a_6 \Delta t, y_n + b_{61} k_1 + \ldots + b_{65} k_5) \end{aligned} \tag{B.5}$$

By changing the coefficients of the terms above, one can arrive at an embedded fourth-order Runge-Kutta equation

$$y^*_{n+1} = y_n + c^*_1 k_1 + c^*_2 k_2 + c^*_3 k_3 + c^*_4 k_4 + c^*_5 k_5 + c^*_6 k_6 + O((\Delta t)^5). \tag{B.6}$$

This is referred to as an embedded equation since it requires no additional function evaluations to obtain it from the fifth-order expression. The values of the constants a_i, b_{ij}, c_i and c^*_i which we used were those found by Cash and Karp as presented in Ref. [21]. The error estimate Δ is then

$$\Delta \equiv y_{n+1} - y^*_{n+1} = \sum_{i=1}^{6} (c_i - c^*_i) k_i. \tag{B.7}$$

In applying this to our multidimensional problem, if any single term in our array

i	a_i	b_{ij}					c_i	c_i^*
1							$\frac{37}{378}$	$\frac{2825}{27648}$
2	$\frac{1}{5}$	$\frac{1}{5}$					0	0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$				$\frac{250}{621}$	$\frac{18575}{48384}$
4	$\frac{3}{5}$	$\frac{3}{10}$	$\frac{-9}{10}$	$\frac{6}{5}$			$\frac{125}{594}$	$\frac{13525}{55296}$
5	1	$\frac{-11}{54}$	$\frac{5}{2}$	$\frac{-70}{27}$	$\frac{35}{27}$		0	$\frac{277}{14336}$
6	$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$	$\frac{1}{4}$
$j=$		1	2	3	4	5		

Table B.1: Cash-Karp Parameters for Embedded Runga-Kutta Method

of values has an error above the tolerance, then the step is not accepted for any of the terms. The function which does this timestepping is

```
void odeintd(double ystart[], int nvar, double x1, double x2,
        double eps, double h1, double hmin, int *nok, int *nbad,
        void (*derivs)(double, double [], double []),
        void (*rkqsd)(double [], double [], int, double *,
                double, double, double [], double *, double *,
                void (*)(double, double [], double [])))
```

It is the same as the function `odeint` of Ref. [21], except that all the variables have been changed from single-precision to double-precision floating point numbers. This function takes in an initial field configuration `vstart` of `nvar` variables and advances it from time `x1` to `x2`. It uses `h1` as a guess for the first timestep, `eps` as the error tolerance and `hmin` as the minimum allowed value for the timestep. It store the number of good steps in the variable `nok` and the number of bad in `nbad`. You are required to pass in two functions; `derivs` which fills and empty array with the first derivative of each element in a configuration you pass in, and `rkqsd` which does

the individual time steps. The function `rkqsd` is a copy of the function `rkqs` of Ref. [21], except that all the variables have been changed from single-precision to double-precision floating point numbers.

The fact that this function deals with the system as a linear array is the reason we chose to store the grid as such. Using the grid reading and writing functions I outlined in the section on data structures, we are able to transparently deal with it as a linear array at one time and as a multidimensional array at other times.

The function `deriv` is specific to each time evolution problem. It is written based on the equations of motion. The user passes in a pointer to the array of $(\phi^i, \dot{\phi}^i)$ and a pointer to an empty array. The function then fills the new array with the corresponding $(\dot{\phi}^i, \ddot{\phi}^i)$. The $\dot{\phi}^i$ are easy since they are just copied from the old array to their new position. The $\ddot{\phi}^i$ are computed using the equations of motion. This is where the entirety of the physics is contained.

As described in Section 3, this time evolution algorithm is stable. Even while tracking violent soliton–soliton collisions and decay, energy was conserved to greater than two parts in 10^5. The energy was conserved to better than one part in 10^4 when the remnants of two decayed solitons were time-reversed to recreate two solitons. In most simulations a tolerance value of `eps` $= 0.001$ was used, which resulted in timesteps around $dt = 0.02$.

Appendix C

C Code

C.1 Constants, Macros and Global Variables

Here is a list of the constants, macros and global variables that are used by the functions which will follow.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#define NR_END 1
#define FREE_ARG char*
#define r_steps 0
#define start_t 0
#define end_t 20.0
#define num_x 125
#define DX 0.2
#define DY 0.2
#define DT 0.001
#define mu2 0.1
#define kap 1.0
#define lam 15
#define kcoeff 1.0
#define s_res 1
#define e_res 1
#define snap_res 10
#define x_cen 62
#define y_cen 62
#define bound_12 0.0
#define bound_3 (sqrt(1.0 + (0.1 / (4 * 15))))
#define M_PI 3.14159265358979323846  /* pi */
#define DEBUG_VAR 0
#define EPS_ERR 0.001
```

```
#define MAXSTP 25000
#define TINY 1.0e-15
#define SAFETY 0.9
#define PGROW -0.2
#define PSHRNK -0.25
#define ERRCON 1.89e-4
static double maxarg1,maxarg2;
#define FMAX(a,b) (maxarg1=(a),maxarg2=(b),(maxarg1) > (maxarg2) ?\
        (maxarg1) : (maxarg2))
static double minarg1,minarg2;
#define FMIN(a,b) (minarg1=(a),minarg2=(b),(minarg1) < (minarg2) ?\
        (minarg1) : (minarg2))
#define SIGN(a,b) ((b) >= 0.0 ? fabs(a) : -fabs(a))

int kmax,kount;
double *xp,**yp,dxsav;
int sct = 0;
int ect = 0;
int esct = 0;
double ener_dat[3];
double lcoeff = 1.0;
double **step_dat;
double **ep;
```

C.2 Numerical Recipes Functions

The following are helper functions used by the Conjugate Gradient and Adaptive Timestep algorithms taken from Ref. [21].

```
void nrerror(char error_text[])
/* Numerical Recipes standard error handler */
{
        fprintf(stderr,"Numerical Recipes run-time error...\n");
        fprintf(stderr,"%s\n",error_text);
        fprintf(stderr,"...now exiting to system...\n");
        exit(1);
}

double *dvector(long nl, long nh)
/* allocate a double vector with subscript range v[nl..nh] */
{
        double *v;

        v=(double *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(double)));
        if (!v) nrerror("allocation failure in dvector()");
        return v-nl+NR_END;
}

double **dmatrix(long nrl, long nrh, long ncl, long nch)
```

```
/* allocate a double matrix with subscript range m[nrl..nrh][ncl..nch] */
{
        long i, nrow=nrh-nrl+1,ncol=nch-ncl+1;
        double **m;

        /* allocate pointers to rows */
        m=(double **) malloc((size_t)((nrow+NR_END)*sizeof(double*)));
        if (!m) nrerror("allocation failure 1 in matrix()");
        m += NR_END;
        m -= nrl;

        /* allocate rows and set pointers to them */
        m[nrl]=(double *) malloc((size_t)((nrow*ncol+NR_END)*sizeof(double)));
        if (!m[nrl]) nrerror("allocation failure 2 in matrix()");
        m[nrl] += NR_END;
        m[nrl] -= ncl;

        for(i=nrl+1;i<=nrh;i++) m[i]=m[i-1]+ncol;

        /* return pointer to array of pointers to rows */
        return m;
}

void free_dvector(double *v, long nl, long nh)
/* free a double vector allocated with dvector() */
{
        free((FREE_ARG) (v+nl-NR_END));
}

void free_dmatrix(double **m, long nrl, long nrh, long ncl, long nch)
/* free a double matrix allocated by dmatrix() */
{
        free((FREE_ARG) (m[nrl]+ncl-NR_END));
        free((FREE_ARG) (m+nrl-NR_END));
}
```

C.3 Data Structure Functions

```
double grid_val(double grid[], int i, int j, int k, int g)
{
  return(*(grid+num_x*6*i+6*j+2*k+g+1));
}

void grid_stor(double grid[], int i, int j, int k, int g, double newval)
{
  *(grid+num_x*6*i+6*j+2*k+g+1) = newval;
}

void put_datfile(double v[])
{
```

```
  int j;
  FILE *f;
  char strr[40];
  char news[40];
  time_t now = time(NULL);

  strcpy(strr, ctime(&now));
  news[0] = 'd';
  news[1] = 'a';
  news[2] = 't';
  news[3] = strr[4];
  news[4] = strr[5];
  news[5] = strr[6];
  news[6] = strr[9];
  news[7] = strr[11];
  news[8] = strr[12];
  news[9] = strr[14];
  news[10] = strr[15];
  news[11] = '.';
  news[12] = 'b';
  news[13] = 'd';
  news[14] = '\0';
  f=fopen(news,"w");
  if(!f){
    printf("Error making file %s\n",news);
    fflush(stdout);
    exit(1);
  }
  j=fwrite(v,sizeof(double),(num_x*num_x*3*2),f);
  fclose(f);
}

int get_datfile(double v[])
{
  int j;
  FILE *f;

  f=fopen("initcond.bd", "r");
  if(!f){
    printf("Error opening file initcond.bd\n");
    fflush(stdout);
    exit(1);
  }
  j=fread(v,sizeof(double),(num_x*num_x*3*2),f);
  fclose(f);
  return(0);
}
```

C.4 Field Altering Functions

```
void init_cond( double vstart[])
{
  int i, j, k;
  double sx, sy, r2, r, f;

  for(i = 0; i<num_x; i++){
    for(k = 0; k<2; k++){
      grid_stor(vstart,0,i,k,0, bound_12);
      grid_stor(vstart,num_x-1,i,k,0,bound_12);
      grid_stor(vstart,i,0,k,0,bound_12);
      grid_stor(vstart,i,num_x-1,k,0,bound_12);
    }
    grid_stor(vstart,0,i,2,0,bound_3);
    grid_stor(vstart,num_x-1,i,2,0,bound_3);
    grid_stor(vstart,i,0,2,0,bound_3);
    grid_stor(vstart,i,num_x-1,2,0,bound_3);
  }
  for(i = 0; i<num_x; i++){
    for(j = 0; j<num_x; j++){
      for(k = 0; k<3; k++){
        grid_stor(vstart,i,j,k,1,0.0);
      }
    }
  }
  for(i = 1; i < num_x - 1; i++){
    for(j = 1; j < num_x - 1; j++){
      sx = DX * (i - x_cen);
      sy = DY * (j - y_cen);
      r2 = (sx*sx + sy*sy);
      r = sqrt(r2);
      f = M_PI * exp(-0.5*r);
      if( r == 0.0 ){
        grid_stor(vstart,i,j,0,0,sin(f));
        grid_stor(vstart,i,j,1,0,sin(f));
        grid_stor(vstart,i,j,2,0,cos(f));
      }else{
        grid_stor(vstart,i,j,0,0,sin(f)*sx/r);
        grid_stor(vstart,i,j,1,0,sin(f)*sy/r);
        grid_stor(vstart,i,j,2,0,cos(f));
      }
    }
  }
}

int trans(double v[], double vout[], int xdis)
{
  int i,j,k,p;

  for(i=0;i<num_x;i++){
    for(k=0;k<3;k++){
```

```
      for(p=0;p<2;p++){
        grid_stor(vout,i,0,k,p,grid_val(v,i,0,k,p));
        grid_stor(vout,0,i,k,p,grid_val(v,0,i,k,p));
        grid_stor(vout,i,num_x-1,k,p,grid_val(v,i,num_x-1,k,p));
        grid_stor(vout,num_x-1,i,k,p,grid_val(v,num_x-1,1,k,p));
      }
    }
  }
  /*printf("before loop\n");*/
  if(xdis<0){
    for(j=1;j<num_x-1;j++){
      for(k=0;k<3;k++){
        for(p=0;p<2;p++){
          for(i=1;i<num_x-1+xdis;i++){
            grid_stor(vout,i,j,k,p,grid_val(v,i-xdis,j,k,p));
          }
          for(i=num_x-1+xdis;i<num_x-1;i++){
            grid_stor(vout,i,j,k,p,grid_val(v,num_x-1,j,k,p));
          }
        }
      }
    }
  }else{
    for(j=1;j<num_x-1;j++){
      for(k=0;k<3;k++){
        for(p=0;p<2;p++){
          for(i=1;i<xdis+1;i++){
            grid_stor(vout,i,j,k,p,grid_val(v,1,j,k,p));
          }
          for(i=xdis+1;i<num_x-1;i++){
            grid_stor(vout,i,j,k,p,grid_val(v,i-xdis,j,k,p));
          }
        }
      }
    }
  }
  /*printf("after loop\n");*/
  return(0);
}

int rotate(double v[], double vout[])
{
  int i,j,k,p;

  for(i=0;i<num_x;i++){
    for(j=0;j<num_x;j++){
      for(k=0;k<3;k++){
        for(p=0;p<2;p++){
          grid_stor(vout,j,num_x-1-i,k,p,grid_val(v,i,j,k,p));
        }
      }
    }
  }
  return(0);
```

```
}

int antisol(double v[], double vout[])
{
  int i,j,k,p;

  for(i=0;i<num_x;i++){
    for(j=0;j<num_x;j++){
      for(k=0;k<3;k++){
        for(p=0;p<2;p++){
          grid_stor(vout,i,j,k,p,grid_val(v,i,j,k,p));
        }
      }
    }
  }
  for(i=0;i<num_x;i++){
    for(j=0;j<num_x;j++){
      grid_stor(vout,i,j,1,0,-grid_val(v,i,j,1,0));
    }
  }
  return(0);
}

int boost(double v[], double vel)
{
  int i,j,k;

  printf("before loop\n");
  for(i=1;i<num_x-1;i++){
    for(j=1;j<num_x-1;j++){
      for(k=0;k<3;k++){
        /*if(i==60 && j==60)
          {printf("phidoti = %f\n", grid_val(v,i,j,k,1));
          }*/
        grid_stor(v,i,j,k,1,vel
                  *(grid_val(v,i-1,j,k,0)-grid_val(v,i,j,k,0))/DX);
        /*if(i==60 && j==60)
          {printf("phidotf = %f\n", vel
                  *(grid_val(v,i-1,j,k,0)-grid_val(v,i,j,k,0))/DX);
                  }*/
      }
    }
  }
  printf("before loop\n");
  return(0);
}

int addgrid(double v1[], double v2[], double vout[])
{
  int i,j,k,p;

  for(i=0;i<num_x;i++){
    for(k=0;k<3;k++){
      for(p=0;p<2;p++){
```

```
        grid_stor(vout,i,0,k,p,grid_val(v1,i,0,k,p));
        grid_stor(vout,0,i,k,p,grid_val(v1,0,i,k,p));
        grid_stor(vout,i,num_x-1,k,p,grid_val(v1,i,num_x-1,k,p));
        grid_stor(vout,num_x-1,i,k,p,grid_val(v1,num_x-1,1,k,p));
      }
    }
  }
  /*printf("before loop\n");*/
  for(i=1;i<num_x-1;i++){
    for(j=1;j<num_x-1;j++){
      for(k=0;k<2;k++){
        for(p=0;p<2;p++){
          grid_stor(vout,i,j,k,p,(grid_val(v1,i,j,k,p)+grid_val(v2,i,j,k,p)));
        }
      }
      grid_stor(vout,i,j,2,0,(grid_val(v1,i,j,2,0)+grid_val(v2,i,j,2,0)
                              -bound_3));
      grid_stor(vout,i,j,2,1,(grid_val(v1,i,j,2,1)+grid_val(v2,i,j,2,1)));
    }
  }
  /*printf("after loop\n");*/
  return(0);
}
```

C.5 Visualization Functions

```
void snapshot( double v[] )
{
  int i, j, k;

  for( k = 0; k < 3; k++){
    printf("p%d%d=[\n", (k+1),  sct);
    for(j = 0; j < num_x; j+=s_res){
      for(i = 0; i < num_x; i+=s_res){
        printf("%f ", grid_val(v,i,j,k,0));
      }
      printf("\n");
    }
    printf("];\n\n\n");
  }
  for( k = 0; k < 3; k++){
    printf("dp%d%d=[", (k+1),  sct);
    for(j = 0; j < num_x; j+=s_res){
      for(i = 0; i < num_x; i+=s_res){
        printf("%f ", grid_val(v,i,j,k,1));
      }
      printf("\n");
    }
    printf("];\n\n\n");
  }
```

```
  sct++;
}

void get_energy( double v[] )
{
  int i, j, k, m, n;
  double phi2;
  double d_t[3], d_x[3], d_y[3], dd_xx[3], dd_yy[3];
  double dd_tx[3], dd_ty[3], dd_xy[3];
  double ener = 0;
  double kener = 0;
  double windnum = 0;
  double wd_x[3], wd_y[3], phi2x3y2, phi2x1y2, phi2x2y3, phi2x2y1;
  double d_xa[3], d_ya[3];

  for(i = 1; i < num_x-1; i++){
    for(j = 1; j < num_x-1; j++){
      phi2x3y2 = grid_val(v,i+1,j,0,0)*grid_val(v,i+1,j,0,0)
        + grid_val(v,i+1,j,1,0)*grid_val(v,i+1,j,1,0)
        + grid_val(v,i+1,j,2,0)*grid_val(v,i+1,j,2,0);
      phi2x1y2 = grid_val(v,i-1,j,0,0)*grid_val(v,i-1,j,0,0)
        + grid_val(v,i-1,j,1,0)*grid_val(v,i-1,j,1,0)
        + grid_val(v,i-1,j,2,0)*grid_val(v,i-1,j,2,0);
      phi2x2y3 = grid_val(v,i,j+1,0,0)*grid_val(v,i,j+1,0,0)
        + grid_val(v,i,j+1,1,0)*grid_val(v,i,j+1,1,0)
        + grid_val(v,i,j+1,2,0)*grid_val(v,i,j+1,2,0);
      phi2x2y1 = grid_val(v,i,j-1,0,0)*grid_val(v,i,j-1,0,0)
        + grid_val(v,i,j-1,1,0)*grid_val(v,i,j-1,1,0)
        + grid_val(v,i,j-1,2,0)*grid_val(v,i,j-1,2,0);
      for(k = 0; k < 3; k++){
        d_xa[k] = (grid_val(v,i,j,k,0) - grid_val(v,i-1,j,k,0)) / DX;
        d_ya[k] = (grid_val(v,i,j,k,0) - grid_val(v,i,j-1,k,0)) / DY;
        d_t[k] = grid_val(v,i,j,k,1);
        d_x[k] = (grid_val(v,i+1,j,k,0) - grid_val(v,i-1,j,k,0)) / (2 * DX);
        d_y[k] = (grid_val(v,i,j+1,k,0) - grid_val(v,i,j-1,k,0)) / (2 * DY);
        dd_xx[k] = (grid_val(v,i+1,j,k,0) + grid_val(v,i-1,j,k,0)
                    - 2*grid_val(v,i,j,k,0)) / (DX * DX);
        dd_yy[k] = (grid_val(v,i,j+1,k,0) + grid_val(v,i,j-1,k,0)
                    - 2*grid_val(v,i,j,k,0)) / (DY * DY);
        dd_tx[k] = (grid_val(v,i+1,j,k,1) - grid_val(v,i-1,j,k,1)) / (2*DX);
        dd_ty[k] = (grid_val(v,i,j+1,k,1) - grid_val(v,i,j-1,k,1)) / (2*DY);
        dd_xy[k] = (grid_val(v,i+1,j+1,k,0) - grid_val(v,i-1,j+1,k,0)
                    - grid_val(v,i+1,j-1,k,0) + grid_val(v,i-1,j-1,k,0))
          / (4*DX*DY);
        wd_x[k] = (grid_val(v,i+1,j,k,0)/phi2x3y2
                   - grid_val(v,i-1,j,k,0)/phi2x1y2) / (2 * DX);
        wd_y[k] = (grid_val(v,i,j+1,k,0)/phi2x2y3
                   - grid_val(v,i,j-1,k,0)/phi2x2y1) / (2 * DY);
      }
      phi2 = grid_val(v,i,j,0,0)*grid_val(v,i,j,0,0)
        + grid_val(v,i,j,1,0)*grid_val(v,i,j,1,0)
        + grid_val(v,i,j,2,0)*grid_val(v,i,j,2,0);
      m = 1;
      n = 2;
```

```
      for(k = 0; k < 3; k++){
        ener += 0.5*(d_xa[k]*d_xa[k] + d_ya[k]*d_ya[k])
          + kap*kap*kcoeff*0.5
          * (d_x[k]*d_x[k]*d_y[m]*d_y[m]
             + d_x[m]*d_x[m]*d_y[k]*d_y[k]
             - 2*d_x[k]*d_y[k]*d_x[m]*d_y[m]);
        kener += 0.5*d_t[k]*d_t[k]
          + kap*kap*kcoeff*0.5
          * (d_t[k]*d_t[k]*d_x[m]*d_x[m]
             + d_t[m]*d_t[m]*d_x[k]*d_x[k]
             - 2*d_t[k]*d_x[k]*d_t[m]*d_x[m]
             + d_t[k]*d_t[k]*d_y[m]*d_y[m]
             + d_t[m]*d_t[m]*d_y[k]*d_y[k]
             - 2*d_t[k]*d_y[k]*d_t[m]*d_y[m]);
        windnum += grid_val(v,i,j,k,0)*((wd_x[m]*wd_y[n])-(wd_y[m]*wd_x[n]));
        m = n;
        n = k;
      }
      ener += mu2*(1-grid_val(v,i,j,2,0)) + lam*lcoeff*(phi2 - 1)*(phi2 - 1);
    }
  }
  for(i=1;i<num_x-1;i++){
    for(k=0;k<3;k++){
      ener += 0.5*(1/(DX*DX))
        *((grid_val(v,i,num_x-1,k,0)-grid_val(v,i,num_x-2,k,0))
          *(grid_val(v,i,num_x-1,k,0)-grid_val(v,i,num_x-2,k,0)));
      ener += 0.5*(1/(DX*DX))
        *((grid_val(v,num_x-1,i,k,0)-grid_val(v,num_x-2,i,k,0))
          *(grid_val(v,num_x-1,i,k,0)-grid_val(v,num_x-2,i,k,0)));
    }
  }
  ener_dat[0] = ener*DX*DY;
  ener_dat[1] = kener*DX*DY;
  ener_dat[2] = windnum*DX*DY/(4*M_PI);
}

void get_enersurf( double v[], double ev[])
{
  int i, j, k, m, n, p;
  double phi2;
  double d_t[3], d_x[3], d_y[3], dd_xx[3], dd_yy[3];
  double dd_tx[3], dd_ty[3], dd_xy[3];
  double d_xa[3], d_ya[3];

  for(i = 0; i < num_x; i++){
    for(j = 0; j < num_x; j++){
      for(k = 0; k < 3; k++){
        for(p = 0; p < 2; p++){
          grid_stor(ev,i,j,k,p,0);
        }
      }
    }
  }
  for(i = 1; i < num_x-1; i++){
```

```
    for(j = 1; j < num_x-1; j++){
      for(k = 0; k < 3; k++){
        d_xa[k] = (grid_val(v,i,j,k,0) - grid_val(v,i-1,j,k,0)) / DX;
        d_ya[k] = (grid_val(v,i,j,k,0) - grid_val(v,i,j-1,k,0)) / DY;
        d_t[k] = grid_val(v,i,j,k,1);
        d_x[k] = (grid_val(v,i+1,j,k,0) - grid_val(v,i-1,j,k,0)) / (2 * DX);
        d_y[k] = (grid_val(v,i,j+1,k,0) - grid_val(v,i,j-1,k,0)) / (2 * DY);
        dd_xx[k] = (grid_val(v,i+1,j,k,0) + grid_val(v,i-1,j,k,0)
                    - 2*grid_val(v,i,j,k,0)) / (DX * DX);
        dd_yy[k] = (grid_val(v,i,j+1,k,0) + grid_val(v,i,j-1,k,0)
                    - 2*grid_val(v,i,j,k,0)) / (DY * DY);
        dd_tx[k] = (grid_val(v,i+1,j,k,1) - grid_val(v,i-1,j,k,1)) / (2*DX);
        dd_ty[k] = (grid_val(v,i,j+1,k,1) - grid_val(v,i,j-1,k,1)) / (2*DY);
        dd_xy[k] = (grid_val(v,i+1,j+1,k,0) - grid_val(v,i-1,j+1,k,0)
                    - grid_val(v,i+1,j-1,k,0) + grid_val(v,i-1,j-1,k,0))
          / (4*DX*DY);
      }
      phi2 = grid_val(v,i,j,0,0)*grid_val(v,i,j,0,0)
        + grid_val(v,i,j,1,0)*grid_val(v,i,j,1,0)
        + grid_val(v,i,j,2,0)*grid_val(v,i,j,2,0);
      m = 1;
      n = 2;
      for(k = 0; k < 3; k++){
        grid_stor(ev,i,j,0,0, grid_val(ev,i,j,0,0) + 0.5*(d_xa[k]*d_xa[k]
                                                          + d_ya[k]*d_ya[k])
                  + kap*kap*kcoeff*0.5
                  * (d_x[k]*d_x[k]*d_y[m]*d_y[m]
                     + d_x[m]*d_x[m]*d_y[k]*d_y[k]
                     - 2*d_x[k]*d_y[k]*d_x[m]*d_y[m]));
        grid_stor(ev,i,j,1,0, grid_val(ev,i,j,1,0) + 0.5*d_t[k]*d_t[k]
          + kap*kap*kcoeff*0.5
          * (d_t[k]*d_t[k]*d_x[m]*d_x[m]
             + d_t[m]*d_t[m]*d_x[k]*d_x[k]
             - 2*d_t[k]*d_x[k]*d_t[m]*d_x[m]
             + d_t[k]*d_t[k]*d_y[m]*d_y[m]
             + d_t[m]*d_t[m]*d_y[k]*d_y[k]
             - 2*d_t[k]*d_y[k]*d_t[m]*d_y[m]));
        m = n;
        n = k;
      }
      grid_stor(ev,i,j,0,0, grid_val(ev,i,j,0,0)
                + mu2*(1-grid_val(v,i,j,2,0))
                + lam*lcoeff*(phi2 - 1)*(phi2 - 1));
      grid_stor(ev,i,j,0,0, grid_val(ev,i,j,0,0)*DX*DY);
      grid_stor(ev,i,j,1,0, grid_val(ev,i,j,1,0)*DX*DY);
      grid_stor(ev,i,j,2,0, grid_val(ev,i,j,0,0)+grid_val(ev,i,j,1,0));
    }
  }
}

void esnapshot( double v[] )
{
  int i, j, k;
```

```
  for( k = 0; k < 3; k++){
    printf("ep%d%d=[\n", (k+1),  esct);
    for(j = 0; j < num_x; j+=s_res){
      for(i = 0; i < num_x; i+=s_res){
        printf("%f ", grid_val(v,i,j,k,0));
      }
      printf("\n");
    }
    printf("];\n\n\n");
  }
  esct++;
}
```

The following functions access the specific global variables generated by the routines taken from Ref. [21].

```
void print_stepdat(int outvar, int nok)
{
  int i, j;

  printf("\nsd=[\n");
  for(i=1;i<=nok;i++){
    for(j=1;j<=outvar;j++){
      printf("%f ", step_dat[j][i]);
    }
    printf("\n");
  }
  printf("];\n\n");
}

void print_times()
{
  int i;

  printf("\nxp = [\n");
  for(i=1;i<=kount;i++){
    printf("%f\n", xp[i]);
  }
  printf("];\n\n");
}

void print_field(int nvar)
{
  int i, j;
  double *fld;

  fld = dvector(1,nvar);
  for(i=1;i<=kount;i++){
    for(j=1;j<=nvar;j++){
      fld[j]=yp[j][i];
    }
    snapshot(fld);
```

```
  }
  free_dvector(fld, 1, nvar);
}

void print_enersurf(int nvar)
{
  int i, j;
  double *fld;

  fld = dvector(1,nvar);
  for(i=1;i<=kount;i++){
    for(j=1;j<=nvar;j++){
      fld[j]=ep[j][i];
    }
    esnapshot(fld);
  }
  free_dvector(fld, 1, nvar);
}
```

C.6 Relaxation Functions

```
void mnbrak(double *ax, double *bx, double *cx, double *fa,
        double *fb, double *fc,         double (*func)(double))
{
        double ulim,u,r,q,fu,dum;

        *fa=(*func)(*ax);
        *fb=(*func)(*bx);
        if (*fb > *fa) {
                SHFT(dum,*ax,*bx,dum)
                SHFT(dum,*fb,*fa,dum)
        }
        *cx=(*bx)+GOLD*(*bx-*ax);
        *fc=(*func)(*cx);
        while (*fb > *fc) {
                r=(*bx-*ax)*(*fb-*fc);
                q=(*bx-*cx)*(*fb-*fa);
                u=(*bx)-((*bx-*cx)*q-(*bx-*ax)*r)/
                        (2.0*SIGN(FMAX(fabs(q-r),TINY),q-r));
                ulim=(*bx)+GLIMIT*(*cx-*bx);
                if ((*bx-u)*(u-*cx) > 0.0) {
                        fu=(*func)(u);
                        if (fu < *fc) {
                                *ax=(*bx);
                                *bx=u;
                                *fa=(*fb);
                                *fb=fu;
                                return;
                        } else if (fu > *fb) {
                                *cx=u;
```

```
                                *fc=fu;
                                return;
                        }
                        u=(*cx)+GOLD*(*cx-*bx);
                        fu=(*func)(u);
                } else if ((*cx-u)*(u-ulim) > 0.0) {
                        fu=(*func)(u);
                        if (fu < *fc) {
                                SHFT(*bx,*cx,u,*cx+GOLD*(*cx-*bx))
                                SHFT(*fb,*fc,fu,(*func)(u))
                        }
                } else if ((u-ulim)*(ulim-*cx) >= 0.0) {
                        u=ulim;
                        fu=(*func)(u);
                } else {
                        u=(*cx)+GOLD*(*cx-*bx);
                        fu=(*func)(u);
                }
                SHFT(*ax,*bx,*cx,u)
                SHFT(*fa,*fb,*fc,fu)
        }
}

double dbrent(double ax, double bx, double cx, double (*f)(double),
        double (*df)(double), double tol, double *xmin)
{
        int iter,ok1,ok2;
        double a,b,d,d1,d2,du,dv,dw,dx,e=0.0;
        double fu,fv,fw,fx,olde,tol1,tol2,u,u1,u2,v,w,x,xm;

        a=(ax < cx ? ax : cx);
        b=(ax > cx ? ax : cx);
        x=w=v=bx;
        fw=fv=fx=(*f)(x);
        dw=dv=dx=(*df)(x);
        for (iter=1;iter<=ITMAX;iter++) {
                xm=0.5*(a+b);
                tol1=tol*fabs(x)+ZEPS;
                tol2=2.0*tol1;
                if (fabs(x-xm) <= (tol2-0.5*(b-a))) {
                        *xmin=x;
                        return fx;
                }
                if (fabs(e) > tol1) {
                        d1=2.0*(b-a);
                        d2=d1;
                        if (dw != dx) d1=(w-x)*dx/(dx-dw);
                        if (dv != dx) d2=(v-x)*dx/(dx-dv);
                        u1=x+d1;
                        u2=x+d2;
                        ok1 = (a-u1)*(u1-b) > 0.0 && dx*d1 <= 0.0;
                        ok2 = (a-u2)*(u2-b) > 0.0 && dx*d2 <= 0.0;
                        olde=e;
                        e=d;
```

```
                        if (ok1 || ok2) {
                                if (ok1 && ok2)
                                        d=(fabs(d1) < fabs(d2) ? d1 : d2);
                                else if (ok1)
                                        d=d1;
                                else
                                        d=d2;
                                if (fabs(d) <= fabs(0.5*olde)) {
                                        u=x+d;
                                        if (u-a < tol2 || b-u < tol2)
                                                d=SIGN(tol1,xm-x);
                                } else {
                                        d=0.5*(e=(dx >= 0.0 ? a-x : b-x));
                                }
                        } else {
                                d=0.5*(e=(dx >= 0.0 ? a-x : b-x));
                        }
                } else {
                        d=0.5*(e=(dx >= 0.0 ? a-x : b-x));
                }
                if (fabs(d) >= tol1) {
                        u=x+d;
                        fu=(*f)(u);
                } else {
                        u=x+SIGN(tol1,d);
                        fu=(*f)(u);
                        if (fu > fx) {
                                *xmin=x;
                                return fx;
                        }
                }
                du=(*df)(u);
                if (fu <= fx) {
                        if (u >= x) a=x; else b=x;
                        MOV3(v,fv,dv, w,fw,dw)
                        MOV3(w,fw,dw, x,fx,dx)
                        MOV3(x,fx,dx, u,fu,du)
                } else {
                        if (u < x) a=u; else b=u;
                        if (fu <= fw || w == x) {
                                MOV3(v,fv,dv, w,fw,dw)
                                MOV3(w,fw,dw, u,fu,du)
                        } else if (fu < fv || v == x || v == w) {
                                MOV3(v,fv,dv, u,fu,du)
                        }
                }
        }
        nrerror("Too many iterations in routine dbrent");
        return 0.0;
}

double df1dim(double x)
{
        int j;
```

```
        double df1=0.0;
        double *xt,*df;

        xt=dvector(1,ncom);
        df=dvector(1,ncom);
        for (j=1;j<=ncom;j++) xt[j]=pcom[j]+x*xicom[j];
        (*nrdfun)(xt,df);
        for (j=1;j<=ncom;j++) df1 += df[j]*xicom[j];
        free_dvector(df,1,ncom);
        free_dvector(xt,1,ncom);
        return df1;
}

double f1dim(double x)
{
        int j;
        double f,*xt;

        xt=dvector(1,ncom);
        for (j=1;j<=ncom;j++) xt[j]=pcom[j]+x*xicom[j];
        f=(*nrfunc)(xt);
        free_dvector(xt,1,ncom);
        return f;
}

void dlinmin(double p[], double xi[], int n, double *fret,
        double (*func)(double []),
        void (*dfunc)(double [], double []))
{
        double dbrent(double ax, double bx, double cx,
                double (*f)(double), double (*df)(double),
                double tol, double *xmin);
        double f1dim(double x);
        double df1dim(double x);
        void mnbrak(double *ax, double *bx, double *cx, double *fa, double *fb,
                double *fc, double (*func)(double));
        int j;
        double xx,xmin,fx,fb,fa,bx,ax;

        ncom=n;
        pcom=dvector(1,n);
        xicom=dvector(1,n);
        nrfunc=func;
        nrdfun=dfunc;
        for (j=1;j<=n;j++) {
                pcom[j]=p[j];
                xicom[j]=xi[j];
        }
        ax=0.0;
        xx=1.0;
        mnbrak(&ax,&xx,&bx,&fa,&fx,&fb,f1dim);
        *fret=dbrent(ax,xx,bx,f1dim,df1dim,TOL,&xmin);
        for (j=1;j<=n;j++) {
                xi[j] *= xmin;
```

```
                p[j] += xi[j];
        }
        free_dvector(xicom,1,n);
        free_dvector(pcom,1,n);
}

void dfrprmn(double p[], int n, double ftol, int *iter, double *fret,
        double (*func)(double []), void (*dfunc)(double [], double []))
{
        void dlinmin(double p[], double xi[], int n, double *fret,
                double (*func)(double []),
                void (*dfunc)(double [], double []));
        int j,its;
        double gg,gam,fp,dgg;
        double *g,*h,*xi;

        g=dvector(1,n);
        h=dvector(1,n);
        xi=dvector(1,n);
        fp=(*func)(p);
        (*dfunc)(p,xi);
        for (j=1;j<=n;j++) {
                g[j] = -xi[j];
                xi[j]=h[j]=g[j];
        }
        printf("rxvar%d=[", rxnum);
        for (its=1;its<=ITMAX;its++) {
                *iter=its;
                printf("%d %2.10f %f\n",its,fp,ener_dat[2]);
                dlinmin(p,xi,n,fret,func, dfunc);
                if (2.0*fabs(*fret-fp) <= ftol*(fabs(*fret)+fabs(fp)+EPS)) {
                        FREEALL
                        return;
                }
                fp= *fret;
                (*dfunc)(p,xi);
                dgg=gg=0.0;
                for (j=1;j<=n;j++) {
                        gg += g[j]*g[j];
                        dgg += (xi[j]+g[j])*xi[j];
                }
                if (gg == 0.0) {
                        FREEALL
                        return;
                }
                gam=dgg/gg;
                for (j=1;j<=n;j++) {
                        g[j] = -xi[j];
                        xi[j]=h[j]=g[j]+gam*h[j];
                }
        }
        /*nrerror("Too many iterations in frprmn");*/
        fprintf(stderr, "Not yet converged to within ENER_TOL\n");
}
```

C.7 Time Evolution Functions

```
void rkckd(double y[], double dydx[], int n, double x, double h, double yout[],
        double yerr[], void (*derivs)(double, double [], double []))
{
        int i;
        static double a2=0.2,a3=0.3,a4=0.6,a5=1.0,a6=0.875,b21=0.2,
                b31=3.0/40.0,b32=9.0/40.0,b41=0.3,b42 = -0.9,b43=1.2,
                b51 = -11.0/54.0, b52=2.5,b53 = -70.0/27.0,b54=35.0/27.0,
                b61=1631.0/55296.0,b62=175.0/512.0,b63=575.0/13824.0,
                b64=44275.0/110592.0,b65=253.0/4096.0,c1=37.0/378.0,
                c3=250.0/621.0,c4=125.0/594.0,c6=512.0/1771.0,
                dc5 = -277.00/14336.0;
        double dc1=c1-2825.0/27648.0,dc3=c3-18575.0/48384.0,
                dc4=c4-13525.0/55296.0,dc6=c6-0.25;
        double *ak2,*ak3,*ak4,*ak5,*ak6,*ytemp;

        ak2=dvector(1,n);
        ak3=dvector(1,n);
        ak4=dvector(1,n);
        ak5=dvector(1,n);
        ak6=dvector(1,n);
        ytemp=dvector(1,n);
        for (i=1;i<=n;i++)
                ytemp[i]=y[i]+b21*h*dydx[i];
        (*derivs)(x+a2*h,ytemp,ak2);
        for (i=1;i<=n;i++)
                ytemp[i]=y[i]+h*(b31*dydx[i]+b32*ak2[i]);
        (*derivs)(x+a3*h,ytemp,ak3);
        for (i=1;i<=n;i++)
                ytemp[i]=y[i]+h*(b41*dydx[i]+b42*ak2[i]+b43*ak3[i]);
        (*derivs)(x+a4*h,ytemp,ak4);
        for (i=1;i<=n;i++)
                ytemp[i]=y[i]+h*(b51*dydx[i]+b52*ak2[i]+b53*ak3[i]+b54*ak4[i]);
        (*derivs)(x+a5*h,ytemp,ak5);
        for (i=1;i<=n;i++)
                ytemp[i]=y[i]+h*(b61*dydx[i]+b62*ak2[i]+b63*ak3[i]+b64*ak4[i]
                                +b65*ak5[i]);
        (*derivs)(x+a6*h,ytemp,ak6);
        for (i=1;i<=n;i++)
                yout[i]=y[i]+h*(c1*dydx[i]+c3*ak3[i]+c4*ak4[i]+c6*ak6[i]);
        for (i=1;i<=n;i++)
                yerr[i]=h*(dc1*dydx[i]+dc3*ak3[i]+dc4*ak4[i]+dc5*ak5[i]
                        +dc6*ak6[i]);
        free_dvector(ytemp,1,n);
        free_dvector(ak6,1,n);
        free_dvector(ak5,1,n);
        free_dvector(ak4,1,n);
        free_dvector(ak3,1,n);
        free_dvector(ak2,1,n);
}
```

```
void rkqsd(double y[], double dydx[], int n, double *x, double htry, double eps,
        double yscal[], double *hdid, double *hnext,
        void (*derivs)(double, double [], double []))
{
        void rkckd(double y[], double dydx[], int n, double x, double h,
                double yout[], double yerr[],
                void (*derivs)(double, double [], double []));
        int i;
        double errmax,h,htemp,xnew,*yerr,*ytemp;

        yerr=dvector(1,n);
        ytemp=dvector(1,n);
        h=htry;
        for (;;) {
                rkckd(y,dydx,n,*x,h,ytemp,yerr,derivs);
                /*snapshot(yscal);
                  snapshot(yerr);
                  for (i=1;i<=n;i++) ytemp[i] = fabs(yerr[i]/yscal[i]);
                  snapshot(y);
                  snapshot(ytemp);
                  exit(1);*/
                errmax=0.0;
                /*fprintf(stderr, "errmax1 = %f, eps = %f\n", errmax, eps);*/
                for (i=1;i<=n;i++) errmax=FMAX(errmax,fabs(yerr[i]/yscal[i]));
                /*fprintf(stderr, "errmax1 = %f, eps = %f\n", errmax, eps);*/
                errmax /= eps;
                /*fprintf(stderr, "errmax2 = %f\n", errmax);*/
                if (errmax <= 1.0) break;
                htemp=SAFETY*h*pow(errmax,PSHRNK);
                h=(h >= 0.0 ? FMAX(htemp,0.1*h) : FMIN(htemp,0.1*h));
                xnew=(*x)+h;
                if (xnew == *x) nrerror("stepsize underflow in rkqsd");
        }
        if (errmax > ERRCON) *hnext=SAFETY*h*pow(errmax,PGROW);
        else *hnext=5.0*h;
        *x += (*hdid=h);
        for (i=1;i<=n;i++) y[i]=ytemp[i];
        free_dvector(ytemp,1,n);
        free_dvector(yerr,1,n);
}

void odeintd(double ystart[], int nvar, double x1, double x2,
        double eps, double h1,
        double hmin, int *nok, int *nbad,
        void (*derivs)(double, double [], double []),
        void (*rkqsd)(double [], double [], int, double *, double, double,
                double [], double *, double *,
                void (*)(double, double [], double [])))
{
        int nstp,i;
        double xsav,x,hnext,hdid,h;
        double *yscal,*y,*dydx;
        double *ey;
```

```
yscal=dvector(1,nvar);
y=dvector(1,nvar);
dydx=dvector(1,nvar);
ey=dvector(1,nvar);
x=x1;
h=SIGN(h1,x2-x1);
*nok = (*nbad) = kount = 0;
for (i=1;i<=nvar;i++) y[i]=ystart[i];
if (kmax > 0) xsav=x-dxsav*2.0;
for (nstp=1;nstp<=MAXSTP;nstp++) {
        (*derivs)(x,y,dydx);
        for (i=1;i<=nvar;i++)
                yscal[i]=fabs(y[i])+fabs(dydx[i]*h)+TINY;
        if (kmax > 0 && kount < kmax-1 && fabs(x-xsav) > fabs(dxsav)) {
                xp[++kount]=x;
                get_enersurf(y, ey);
                for (i=1;i<=nvar;i++){
                  yp[i][kount]=y[i];
                  ep[i][kount]=ey[i];
                }
                xsav=x;
        }
        if ((x+h-x2)*(x+h-x1) > 0.0) h=x2-x;
        get_energy(y);
        printf("%ctime = %f, h%d = %1.10f, energy = %f, wind = %f,
           step = %d\n", '%', x, (*nok), h, (ener_dat[0]+ener_dat[1]),
           ener_dat[2], nstp);
        (*rkqsd)(y,dydx,nvar,&x,h,eps,yscal,&hdid,&hnext,derivs);
        if (hdid == h) {
          ++(*nok);
          step_dat[1][(*nok)]=h;
          step_dat[2][(*nok)]=x;
          get_energy(y);
          step_dat[3][(*nok)]=(ener_dat[0]+ener_dat[1]);
          step_dat[4][(*nok)]=ener_dat[0];
          step_dat[5][(*nok)]=ener_dat[1];
          step_dat[6][(*nok)]=ener_dat[2];
        }else {++(*nbad);}
        if ((x-x2)*(x2-x1) >= 0.0) {
                for (i=1;i<=nvar;i++) ystart[i]=y[i];
                if (kmax) {
                        xp[++kount]=x;
                        get_enersurf(y, ey);
                        for (i=1;i<=nvar;i++){
                          yp[i][kount]=y[i];
                          ep[i][kount]=ey[i];
                        }
                }
                free_dvector(dydx,1,nvar);
                free_dvector(y,1,nvar);
                free_dvector(yscal,1,nvar);
                free_dvector(ey,1,nvar);
                return;
```

```
            }
            if (fabs(hnext) <= hmin) nrerror("Step size too small in odeint");
            h=hnext;
        }
        nrerror("Too many steps in routine odeint");
}
```

C.8 Deriv Function

This function contains the equations of motion for our model. If you would like to use the Conjugate Gradient or Adaptive Timestep algorithms with a different model, you must alter this function.

```
void derivs(double x, double v[], double dv[])
{
  int i, j, k, p, m, n;
  double phi2, denom;
  double ca[3], cb[3], cc[3];
  double d_t[3], d_x[3], d_y[3], dd_xx[3], dd_yy[3];
  double dd_tx[3], dd_ty[3], dd_xy[3];
  double gridd_x[num_x][num_x][3];
  double gridd_y[num_x][num_x][3];

  for(i = 0; i < num_x; i++){
    for(k = 0; k < 3; k++){
      for(p = 0; p < 2; p++){
        grid_stor(dv,0,i,k,p,0);
        grid_stor(dv,num_x-1,i,k,p,0);
        grid_stor(dv,i,0,k,p,0);
        grid_stor(dv,i,num_x-1,k,p,0);
      }
    }
  }
  for(i = 0; i < num_x; i++){
    for(j = 0; j < num_x; j++){
      for(k = 0; k < 3; k++){
        grid_stor(dv,i,j,k,0,grid_val(v,i,j,k,1));
      }
    }
  }
  for(i = 0; i < num_x; i++){
    for(k = 0; k < 2; k++){
      gridd_x[0][i][k]=(grid_val(v,1,i,k,0) - bound_12)
          / (2 * DX);
      gridd_x[num_x-1][i][k]=(bound_12 - grid_val(v,num_x-2,i,k,0))
          / (2 * DX);
      gridd_x[i][0][k]=0.0;
      gridd_x[i][num_x-1][k]=0.0;
      gridd_y[0][i][k]=0.0;
```

```
    gridd_y[num_x-1][i][k]=0.0;
    gridd_y[i][0][k]=(grid_val(v,i,1,k,0) - bound_12)
        / (2 * DY);
    gridd_y[i][num_x-1][k]=(bound_12 - grid_val(v,i,num_x-2,k,0))
        / (2 * DY);
  }
  gridd_x[0][i][2]=(grid_val(v,1,i,2,0) - bound_3)
    / (2 * DX);
  gridd_x[num_x-1][i][2]=(bound_3 - grid_val(v,num_x-2,i,2,0))
    / (2 * DX);
  gridd_x[i][0][2]=0.0;
  gridd_x[i][num_x-1][2]=0.0;
  gridd_y[0][i][2]=0.0;
  gridd_y[num_x-1][i][2]=0.0;
  gridd_y[i][0][2]=(grid_val(v,i,1,2,0) - bound_3)
    / (2 * DY);
  gridd_y[i][num_x-1][2]=(bound_3 - grid_val(v,i,num_x-2,2,0))
    / (2 * DY);
}
for(i = 1; i < num_x-1; i++){
  for(j = 1; j < num_x-1; j++){
    for(k = 0; k < 3; k++){
      gridd_x[i][j][k]=(grid_val(v,i+1,j,k,0) - grid_val(v,i-1,j,k,0))
        / (2 * DX);
      gridd_y[i][j][k]=(grid_val(v,i,j+1,k,0) - grid_val(v,i,j-1,k,0))
        / (2 * DY);
    }
  }
}
for(i = 1; i < num_x-1; i++){
  for(j = 1; j < num_x-1; j++){
    for(k = 0; k < 3; k++){
      d_t[k] = grid_val(v,i,j,k,1);
      d_x[k] = (grid_val(v,i+1,j,k,0) - grid_val(v,i-1,j,k,0)) / (2 * DX);
      d_y[k] = (grid_val(v,i,j+1,k,0) - grid_val(v,i,j-1,k,0)) / (2 * DY);
      dd_xx[k] = (grid_val(v,i+1,j,k,0) + grid_val(v,i-1,j,k,0)
                  - 2*grid_val(v,i,j,k,0)) / (DX * DX);
      dd_yy[k] = (grid_val(v,i,j+1,k,0) + grid_val(v,i,j-1,k,0)
                  - 2*grid_val(v,i,j,k,0)) / (DY * DY);
      dd_tx[k] = (grid_val(v,i+1,j,k,1) - grid_val(v,i-1,j,k,1)) / (2*DX);
      dd_ty[k] = (grid_val(v,i,j+1,k,1) - grid_val(v,i,j-1,k,1)) / (2*DY);
      dd_xy[k] = (grid_val(v,i+1,j+1,k,0) - grid_val(v,i-1,j+1,k,0)
                  - grid_val(v,i+1,j-1,k,0) + grid_val(v,i-1,j-1,k,0))
        / (4*DX*DY);
    }
if(DEBUG_VAR==1){printf("deriv call 2\n"); fflush( stdout );}
phi2 = grid_val(v,i,j,0,0)*grid_val(v,i,j,0,0)
  + grid_val(v,i,j,1,0)*grid_val(v,i,j,1,0)
  + grid_val(v,i,j,2,0)*grid_val(v,i,j,2,0);
m = 1;
n = 2;
for(k = 0; k < 3; k++){
  ca[k] = dd_xx[k] + dd_yy[k]
    /*0.0 - gridd_x[i-1][j][k] / (2*DX) + gridd_x[i+1][j][k] / (2*DX)
```

```
  - gridd_y[i][j-1][k] / (2*DY) + gridd_y[i][j+1][k] / (2*DY)*/
- 4 * lam * lcoeff * (phi2 - 1) * grid_val(v,i,j,k,0)
-kap*kap*kcoeff
*(0.5)*(0.0
        -((gridd_x[i-1][j][k]/DX)
          *(grid_val(v,i-1,j,m,1)*grid_val(v,i-1,j,m,1)
            +grid_val(v,i-1,j,n,1)*grid_val(v,i-1,j,n,1)))
        +((gridd_x[i+1][j][k]/DX)
          *(grid_val(v,i+1,j,m,1)*grid_val(v,i+1,j,m,1)
            +grid_val(v,i+1,j,n,1)*grid_val(v,i+1,j,n,1)))
        +(1.0/DX)*(grid_val(v,i-1,j,k,1)*((grid_val(v,i-1,j,m,1)
                                           *gridd_x[i-1][j][m])
                                          +(grid_val(v,i-1,j,n,1)
                                           *gridd_x[i-1][j][n])))
        -(1.0/DX)*(grid_val(v,i+1,j,k,1)*((grid_val(v,i+1,j,m,1)
                                           *gridd_x[i+1][j][m])
                                          +(grid_val(v,i+1,j,n,1)
                                           *gridd_x[i+1][j][n])))
        -((gridd_y[i][j-1][k]/DY)
          *(grid_val(v,i,j-1,m,1)*grid_val(v,i,j-1,m,1)
            +grid_val(v,i,j-1,n,1)*grid_val(v,i,j-1,n,1)))
        +((gridd_y[i][j+1][k]/DY)
          *(grid_val(v,i,j+1,m,1)*grid_val(v,i,j+1,m,1)
            +grid_val(v,i,j+1,n,1)*grid_val(v,i,j+1,n,1)))
        +(1.0/DY)*(grid_val(v,i,j-1,k,1)*((grid_val(v,i,j-1,m,1)
                                           *gridd_y[i][j-1][m])
                                          +(grid_val(v,i,j-1,n,1)
                                           *gridd_y[i][j-1][n])))
        -(1.0/DY)*(grid_val(v,i,j+1,k,1)*((grid_val(v,i,j+1,m,1)
                                           *gridd_y[i][j+1][m])
                                          +(grid_val(v,i,j+1,n,1)
                                           *gridd_y[i][j+1][n])))
        +((gridd_x[i-1][j][k]/DX)*((gridd_y[i-1][j][m]
                                    *gridd_y[i-1][j][m])
                                   +(gridd_y[i-1][j][n]
                                    *gridd_y[i-1][j][n])))
        -((gridd_x[i+1][j][k]/DX)*((gridd_y[i+1][j][m]
                                    *gridd_y[i+1][j][m])
                                   +(gridd_y[i+1][j][n]
                                    *gridd_y[i+1][j][n])))
        +((gridd_y[i][j-1][k]/DY)*((gridd_x[i][j-1][m]
                                    *gridd_x[i][j-1][m])
                                   +(gridd_x[i][j-1][n]
                                    *gridd_x[i][j-1][n])))
        -((gridd_y[i][j+1][k]/DY)*((gridd_x[i][j+1][m]
                                    *gridd_x[i][j+1][m])
                                   +(gridd_x[i][j+1][n]
                                    *gridd_x[i][j+1][n])))
        -(1.0/DX)*(gridd_y[i-1][j][k]*((gridd_x[i-1][j][m]
                                        *gridd_y[i-1][j][m])
                                       +(gridd_x[i-1][j][n]
                                        *gridd_y[i-1][j][n])))
        +(1.0/DX)*(gridd_y[i+1][j][k]*((gridd_x[i+1][j][m]
                                        *gridd_y[i+1][j][m])
```

```
                                                +(gridd_x[i+1][j][n]
                                                  *gridd_y[i+1][j][n])))
                -(1.0/DY)*(gridd_x[i][j-1][k]*((gridd_x[i][j-1][m]
                                                 *gridd_y[i][j-1][m])
                                                +(gridd_x[i][j-1][n]
                                                  *gridd_y[i][j-1][n])))
                +(1.0/DY)*(gridd_x[i][j+1][k]*((gridd_x[i][j+1][m]
                                                 *gridd_y[i][j+1][m])
                                                +(gridd_x[i][j+1][n]
                                                  *gridd_y[i][j+1][n]))))
        +kap*kap*kcoeff
        *(0.0
          -d_t[k]*2*(d_x[m]*dd_tx[m]+d_x[n]*dd_tx[n])
          +dd_tx[k]*(d_t[m]*d_x[m]+d_t[n]*d_x[n])
          +d_x[k]*(d_t[m]*dd_tx[m]+d_t[n]*dd_tx[n])
          -d_t[k]*2*(d_y[m]*dd_ty[m]+d_y[n]*dd_ty[n])
          +dd_ty[k]*(d_t[m]*d_y[m]+d_t[n]*d_y[n])
          +d_y[k]*(d_t[m]*dd_ty[m]+d_t[n]*dd_ty[n]));
      cb[k] = 0.0 - 1.0 - kap*kap*kcoeff*((d_x[m]*d_x[m])
                                          + (d_x[n]*d_x[n])
                                          + (d_y[m]*d_y[m])
                                          + (d_y[n]*d_y[n]));
      cc[k] = kap*kap*kcoeff*(d_x[k]*d_x[m] + d_y[k]*d_y[m]);
      m = n;
      n = k;
    }
    ca[2] += mu2;
    /*if(i==15 && j==15){
    printf("\nca[0]=%f\nca[1]=%f\nca[2]=%f\n", ca[0], ca[1], ca[2]);
    printf("\ncb[0]=%f\ncb[1]=%f\ncb[2]=%f\n", cb[0], cb[1], cb[2]);
    printf("\ncc[0]=%f\ncc[1]=%f\nc[2]=%f\n", cc[0], cc[1], cc[2]);
    fflush( stdout );
    exit(1);
    }*/
    if(DEBUG_VAR==1){printf("deriv call 3\n"); fflush( stdout );}
    denom = 1.0/(-cc[1]*cc[1]*cb[0]+2*cc[1]*cc[2]*cc[0]-cc[0]*cc[0]*cb[2]
                 +cb[1]*cb[0]*cb[2]-cb[1]*cc[2]*cc[2]);
    grid_stor(dv,i,j,0,1,-(ca[2]*cc[1]*cc[0]-ca[2]*cb[1]*cc[2]
                       +ca[0]*cb[1]*cb[2]-cc[0]*ca[1]*cb[2]
                       -cc[1]*cc[1]*ca[0]+cc[1]*ca[1]*cc[2])*denom);
    grid_stor(dv,i,j,1,1,(-cc[1]*cc[2]*ca[0]+cc[1]*cb[0]*ca[2]
                      +cc[0]*cb[2]*ca[0]-ca[1]*cb[0]*cb[2]
                      +ca[1]*cc[2]*cc[2]-cc[0]*ca[2]*cc[2])*denom);
    grid_stor(dv,i,j,2,1,-(cb[0]*ca[2]*cb[1]-cb[0]*cc[1]*ca[1]
                       -ca[0]*cc[2]*cb[1]-ca[2]*cc[0]*cc[0]
                       +cc[0]*cc[2]*ca[1]+cc[1]*cc[0]*ca[0])*denom);
     }
    }
  }
```

C.9 Example Main Program

Here is an example of these functions used in conjunction to perform a time evolution on a pre-existing grid.

```
int main()
{
  double *vstart;
  char ener_flag[5];
  int nok, nbad;
  int outvar = 6;

  kmax = 6;
  dxsav = (end_t - start_t) / (kmax-1);
  vstart = dvector(1,(num_x*num_x*3*2));
  xp = dvector(1,kmax);
  yp = dmatrix(1, (num_x*num_x*3*2), 1, kmax);
  ep = dmatrix(1, (num_x*num_x*3*2), 1, kmax);
  step_dat = dmatrix(1, outvar, 1, MAXSTP);
  get_datfile(vstart);
  /*init_cond(vstart);*/
  ener_flag[0] = 'e';
  ener_flag[1] = 'n';
  ener_flag[2] = 'e';
  ener_flag[3] = 'r';
  ener_flag[4] = '\0';
  lcoeff = 1.0;
  odeintd( vstart, (num_x*num_x*3*2), start_t, end_t, EPS_ERR, DT,
          0, &nok, &nbad, derivs, rkqsd);
  printf("\nnok=%d;\nnbad=%d;\n", nok, nbad);
  print_stepdat(outvar, nok);
  print_times();
  print_field((num_x*num_x*3*2));
  print_enersurf((num_x*num_x*3*2));
  put_datfile(vstart);
  free_dvector(vstart, 1, (num_x*num_x*3*2));
  free_dvector(xp,1,kmax);
  free_dmatrix(yp, 1, (num_x*num_x*3*2), 1, kmax);
  free_dmatrix(ep, 1, (num_x*num_x*3*2), 1, kmax);
  free_dmatrix(step_dat, 1, outvar, 1, MAXSTP);
  return 0;
}
```

Bibliography

[1] T. H. R. Skyrme, Proc. R. Soc. London **A260**, 127 (1961).

[2] J. M. Gipson and H. C. Tze, Nucl. Phys. **B183**, 524 (1981); J. M. Gipson, *ibid.* **B231**, 365 (1984).

[3] E. D'Hoker and E. Farhi, Phys. Lett. **134B**, 86 (1984); Nucl. Phys. **B241**, 109 (1984).

[4] J. Ambjorn and V. A. Rubakov, Nucl. Phys. **B256**, 434 (1985); V. A. Rubakov, *ibid* **B256**, 509 (1985).

[5] V. A. Rubakov, B. E. Stern and P. G. Tinyakov, Phys. Lett. **B160**, 292 (1985).

[6] G. Eilam, D. Klabucar and A. Stern, Phys. Rev. Lett. **56**, 1331 (1986); G. Eilam and A. Stern, Nucl. Phys. **B294**, 775 (1987).

[7] E. Farhi, J. Goldstone, A. Lue and K. Rajagopal, Phys. Rev. D **54**, 5336 (1996) [hep-ph/9511219].

[8] J. Baacke, G. Eilam and H. Lange, Phys. Lett. **B199**, 234 (1987); L. Carson, proceedings of Beyond the Standard Model II, Norman, OK, 224 (1990).

[9] A. Lue and M. Trodden, Phys. Rev. **D58**, 057901 (1998) [hep-ph/9802281].

[10] J. J. Verbaarschot, T. S. Walhout, J. Wambach and H. W. Wyld, Nucl. Phys. **A461**, 603 (1987); A. E. Allder, S. E. Koonin, R. Seki and H. M. Sommermann, Phys. Rev. Lett. **59**, 2836 (1987); H. M. Sommermann, R. Seki, S. Larson and S. E. Koonin, Phys. Rev. D **45** 4303 (1992).

[11] W. Y. Crutchfield and J. B. Bell, J. Comp. Phys. **110**, 234 (1994).

[12] R. D. Amado, M. A. Halasz and P. Protopapas, Phys. Rev. **D61**, 074022 (2000) [hep-ph/9909426].

[13] A. A. Belavin and A. M. Polyakov, JETP Lett. **22**, 245 (1975).

[14] F. Wilczek and A. Zee, Phys. Rev. Lett. **51**, 2250 (1983).

[15] R. Mackenzie and F. Wilczek, Int. J. Mod. Phys. **A3**, 2827 (1988).

[16] S. L. Sondhi, A. Karlhede, S. A. Kivelson and E. H. Rezayi, Phys. Rev. B **47**, 16419 (1993).

[17] B. M. Piette, W. J. Zakrzewski, H. J. Mueller-Kirsten and D. H. Tchrakian, Phys. Lett. **B320**, 294 (1994).

[18] B. M. Piette, B. J. Schroers and W. J. Zakrzewski, Z. Phys. **C65**, 165 (1995) [hep-th/9406160].

[19] B. M. Piette, B. J. Schroers and W. J. Zakrzewski, Nucl. Phys. **B439**, 205 (1995) [hep-ph/9410256].

[20] R. A. Leese, M. Peyrard and W. J. Zakrzewski, Nonlinearity **3**, 773 (1990); M. Peyrard, B. Piette and W. J. Zakrzewski, Nonlinearity **5**, 563 (1992); A. Kudryavtsev, B. M. Piette and W. J. Zakrzewski, hep-th/9709187; T. Weidig, hep-th/9811238; P. Eslami, W. J. Zakrzewski and M. Sarbishaei, hep-th/0001153.

[21] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery, **Numerical Recipes in C**, Cambridge University Press, 1997.