2024

Simulation Studies of Topological Defect Behavior in Rank-2 Anti-Symmetric Tensor Fields

Wyatt Carbonell

Simulation Studies of Topological Defect Behavior in Rank-2 Anti-Symmetric Tensor Fields

Presented by

Wyatt Carbonell

to the Department of Physics, Astronomy, and Geophysics

Supervised by

Prof. Michael Seifert

Connecticut College
New London, CT, 06320
May 1, 2024

Abstract

Lorentz-violating models are a class of theories in which a fundamental field breaks the underlying Lorentz symmetry of a system action by assuming a nonzero vacuum expectation value. In this work, we consider a model which achieves such a Lorentz violation through a rank-2, anti-symmetric tensor field. We present a derivation of the equations governing the time evolution of this tensor field, the specifications of a program capable of numerically simulating this evolution through the solving of differential equations, evidence for the success of our simulation in accurately representing the model, and evidence against the physical viability of such a field. In particular, we find that monopole topological defects would develop in abundance in the early universe and would maintain a relative proliferation. These monopoles have clear observational signatures and have yet to be found experimentally. We also determine that the fundamental field in question is only correlated over short scales and, therefore, appears randomized over long distances. The combination of these factors casts doubt on the viability of this model, but a more sophisticated simulation is necessary to draw definitive conclusions.

Acknowledgements

I would like to begin by thanking my advisor, Michael Seifert, for his invaluable contributions to this work and my education in physics. I would also like to thank my readers, Perry Susskind and Alex Gianninas, for their detailed comments and encouragement throughout the review process. Last but not least, my many thanks to the Yale Center for Research Computing, Andrew Sherman, and Thomas Langford for making this project possible through their advice and high-performance computing resources.

Wyatt Carbonell
Department of Physics, Astronomy, and Geophysics
Connecticut College
May 1, 2024

For my grandfather.

Contents

List of Figures

Introduction

1.1 Background

Lorentz symmetry has been a central component in the modern study of physics since the introduction of special relativity by Albert Einstein in 1905 [1]. However, there is much to learn from models which break Lorentz symmetry in some way. A number of theories have found great success in considering situations in which some symmetry is violated. Many such theoretical models involve the introduction of a field with a nonzero field value at the potential minimum, effectively breaking the symmetry of the underlying Lagrangian [2] [3] [4] [5]. The set of all field values for which the potential energy is minimized is known as the vacuum manifold. Perhaps the most well-known of these models is the Higgs field in particle physics, which arises from the introduction of a symmetry-breaking scalar field [6]. These models also underlie the modern study of phase transitions in condensed matter systems, most notably the Ginzburg-Landau model of superconductivity [7].

An important classification of solutions that may arise in any model with a spontaneously broken symmetry are topological defects, which arise when the vacuum manifold has certain nontrivial topological properties [8]. Topological defects are generally separated into three classifications depending on their effective number of spatial dimensions: two-dimensional 'domain walls', one-dimensional 'cosmic strings', and point-like 'monopoles'. These topological defects arise from a conceptually straightforward principle. Any field which breaks a symmetry will, in general, restore the broken symmetry at sufficiently high temperatures. This fact implies that a phase transition from a highly symmetric to a spontaneously broken system should occur at some point in the Universe's expansion. In the event that the minimum of the potential is not unique, causally separated parts of the universe can settle into different parts of the vacuum manifold and be unable to evolve into equivalent minima without substantial input energy. These field configurations give rise to topological defects [9]. The Kibble Mechanism describes how this process can occur naturally in the Universe [10] [11].

In particular, we consider topological defects in a rank-2, anti-symmetric tensor field capable of violating Lorentz symmetry by taking on a nonzero vacuum expectation value (VEV). This particular field has been shown by Seifert to only produce monopole topological defects [8]. Lau and Seifert have also described the behavior of these monopole topological defects when coupling to electromagnetic radiation, providing a potential mechanism for their observation through gravitational lensing effects [12]. The direct observation of a monopole topological defect could be relevant to our understanding of cosmic inflation, a phenomenon which would greatly suppress the number of monopoles one would expect to find in the Universe [13]. Monopole topological defects have also been proposed as one mechanism through which large-scale structure may have formed in the Universe [14]. Because of these potential implications, we aim in this work to predict how many monopole topological defects we should expect to find in the modern Universe.

Topological defects have been studied for some time through numerical simulation, including a notable paper published by Bennett and Rhie in 1990 [15]. A previous study attempted to establish the recombination rate of monopole topological defects in a Lorentz-violating, rank-2, anti-symmetric tensor field model through numerical simulation, and was ultimately unsuccessful [16]. We have reason to believe this failure was due to a systematic problem in models constrained by a potential later identified by Seifert [17]. We instead propose the use of a Lagrange multiplier to constrain the tensor field to a nonzero VEV, which has been shown by Seifert to avoid this systematic issue [17]. Lagrange multiplier models differ from potential models in that the system state is always strictly on the vacuum manifold, unlike that of a potential model which is capable of evolving some amount away from the vacuum manifold [18].

1.2 Overview

In Section 2, we present derivations and justifications for equations and algorithms necessary to construct and benchmark our simulation and track monopole topological defects. In Section 3, we present a technical overview of the simulation we developed, and describe the data which was collected for our study. In Section 4, we present data and analysis relevant to the stability and performance of our simulation and to the counting of monopole topological defects. In Section 5, we discuss the implications of the data produced by our simulation. In Section 6, we conclude that our limited simulation casts doubt on the viability of this model and suggest directions for follow-up studies. Appendix A.1 contains relevant mathematical background on derivatives with respect to vector quantities. Appendix A.2 contains a more general form of our model which considers both possible definitions for invariants. Appendix A.3 contains code files used to run an instance of our simulation. Appendix A.4 contains supplementary results to complement those highlighted in the main body of the text.

The metric signature $(-,+,+,+)$ and the units $h = c = 1$ are used implicitly throughout this work. We also assume the use of Einstein summation notation, i.e. any tensor index which is raised on one term and lowered on an adjacent term implies a sum over all possible values of the index. For example,

$$B^{\nu}B_{\nu} = \sum_{\nu} B^{\nu}B_{\nu}.$$

Theory

2.1 System Action and Hamiltonian For a Single Invariant

In this work, we study the time evolution of an anti-symmetric, rank-2 tensor field, denoted B^{ab}. In particular, we consider a model in which the invariant vacuum expectation value of the norm of B^{ab}, defined as

$$X = B^{ab}B_{ab}, \tag{2.1.1}$$

is nonzero. A second invariant can be written, but is not used for the majority of this work (See Appendix A.2). This class of models has been studied at length by Seifert [19], whose notation we will adopt. The remainder of this subsection is a brief summary of his work.

The tensor B^{ab} can be decomposed into 'electric' and 'magnetic' parts,

$$P^i = B^{0i} \tag{2.1.2}$$

$$Q^i = \frac{1}{2}\epsilon^{ijk}B^{jk}, \tag{2.1.3}$$

where ϵ^{ijk} is the volume element on a constant-t hyper-surface in spacetime. We will find it convenient to define

$$F^{abc} = 3\partial_{[a}B_{bc]}, \tag{2.1.4}$$

which allows us to write the kinetic component of the field Lagrangian as

$$-\frac{1}{12}F_{abc}F^{abc} = \frac{1}{2}\left[(\dot{\vec{Q}} - \vec{\nabla}\times\vec{P})^2 - (\vec{\nabla}\cdot\vec{Q})^2\right]. \tag{2.1.5}$$

It is also possible to write the vacuum expectation value X as

$$X = 2Q^2 - 2P^2. \tag{2.1.6}$$

We will take the system action to be

$$S = \int -\frac{1}{12}F^{abc}F_{abc} - \lambda(X - b)\mathrm{d}x^4, \tag{2.1.7}$$

where the vacuum expectation value X has been constrained to the nonzero constant b by the Lagrange multiplier λ. Writing out the Lagrangian explicitly in terms of the fields $\vec{P}$ and $\vec{Q}$ gives

$$\mathcal{L} = \frac{1}{2}\left[(\dot{\vec{Q}} - \vec{\nabla}\times\vec{P})^2 - (\vec{\nabla}\cdot\vec{Q})^2\right] - \lambda(2Q^2 - 2P^2 - b). \tag{2.1.8}$$

From this form of the Lagrangian, it is relatively easy to determine the conjugate momenta to the fields $\vec{P}$ and $\vec{Q}$ to be

$$\vec{\Pi}_Q = \frac{\partial\mathcal{L}}{\partial\dot{\vec{Q}}} = \dot{\vec{Q}} - (\vec{\nabla}\times\vec{P}) \tag{2.1.9}$$

$$\vec{\Pi}_P = \frac{\partial \mathcal{L}}{\partial \dot{\vec{P}}} = 0, \tag{2.1.10}$$

respectively[1]. We also define the conjugate momentum to the field λ to be

$$\varpi = \frac{\partial \mathcal{L}}{\partial \dot{\lambda}} = 0. \tag{2.1.11}$$

One can show, through Dirac-Bergmann analysis [20], that the augmented Hamiltonian of this action is

$$\begin{aligned}\mathcal{H}_A =& \frac{1}{2}\Pi_Q^2 + \vec{\Pi}_Q \cdot (\vec{\nabla} \times \vec{P}) + \frac{1}{2}(\vec{\nabla} \cdot \vec{Q})^2 \\ &+ \lambda(X - b) + \vec{u} \cdot \vec{\Pi}_p + u_\lambda \varpi.\end{aligned} \tag{2.1.12}$$

where we have included the extra Lagrange multipliers $\vec{u}$ and u_λ to enforce the primary constraints $\vec{\Pi}_P = 0$ and $\varpi = 0$, respectively. The secondary constraints for this system are

$$\vec{\Psi} = 4\lambda\vec{P} - \vec{\nabla} \times \vec{\Pi}_Q \tag{2.1.13}$$

and

$$\Psi = -(X - b). \tag{2.1.14}$$

The secondary constraints must also be preserved under time evolution, and enforcing this fact determines the values of the additional Lagrange multipliers to be

$$u_\lambda = -\frac{1}{P^2}\left[\vec{P} \cdot \vec{\nabla} \times (\lambda\vec{Q}) + \lambda\vec{Q} \cdot (\vec{\Pi}_Q + \vec{\nabla} \times \vec{P})\right] \tag{2.1.15}$$

$$\vec{u} = -\frac{1}{\lambda}\left[u_\lambda\vec{P} + \vec{\nabla} \times (\lambda\vec{Q})\right], \tag{2.1.16}$$

so long as $P^2 \neq 0$ and $\lambda \neq 0$. Seifert concludes his analysis at this point.

2.2 Time Evolution Equations For a Single Invariant

There are a total of fourteen fields which are capable of evolving in time, all of which are represented in the Hamiltonian as vector or scalar quantities. We take vector fields to each be composed of three scalar fields corresponding to the three dimensions of space. We note that u_λ and $\vec{u}$ are fully determined by other field quantities, so do not need explicit time evolution. In this section, we present the derivations of time evolution equations for the ten currently unconstrained fields. Together, these equations form a system of differential equations, which we ultimately aim to solve numerically.

It is possible to construct time evolution equations through the use of a Poisson bracket, defined as

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \{f, \mathcal{H}_A\} \equiv \sum_\alpha \frac{\partial f}{\partial q_\alpha}\frac{\partial \mathcal{H}_A}{\partial p_\alpha} - \frac{\partial \mathcal{H}_A}{\partial q_\alpha}\frac{\partial f}{\partial p_\alpha}, \tag{2.2.1}$$

where the sum is over all pairs of Hamiltonian coordinates q_α with their conjugate momenta p_α [21]. Using this, we find

$$\dot{\vec{Q}} = \frac{\mathrm{d}\vec{Q}}{\mathrm{d}t} = \frac{\partial \mathcal{H}_A}{\partial \vec{\Pi}_Q} \tag{2.2.2}$$

$$\dot{\vec{\Pi}}_Q = \frac{\mathrm{d}\vec{\Pi}_Q}{\mathrm{d}t} = -\frac{\partial \mathcal{H}_A}{\partial \vec{Q}} \tag{2.2.3}$$

[1]See Appendix A.1 for an explanation of derivatives with respect to vector fields.

$$\dot{\vec{P}} = \frac{\mathrm{d}\vec{P}}{\mathrm{d}t} = \frac{\partial \mathcal{H}_A}{\partial \vec{\Pi}_P} = \vec{u} \tag{2.2.4}$$

$$\dot{\vec{\Pi}}_P = \frac{\mathrm{d}\vec{\Pi}_P}{\mathrm{d}t} = -\frac{\partial \mathcal{H}_A}{\partial \vec{P}} \tag{2.2.5}$$

$$\dot{\lambda} = \frac{\mathrm{d}\lambda}{\mathrm{d}t} = \frac{\partial \mathcal{H}_A}{\partial \varpi} = u_\lambda \tag{2.2.6}$$

$$\dot{\varpi} = \frac{\mathrm{d}\varpi}{\mathrm{d}t} = -\frac{\partial \mathcal{H}_A}{\partial \lambda}. \tag{2.2.7}$$

A few of these derivatives are simple to take and are represented above. For the remainder, we will need to make use of a few vector derivative identities, whose proofs are available in Appendix A.1. For ease, these identities have been replicated below.

Vector Derivative Identities Suppose $\vec{A}$ and $\vec{B}$ are vector fields and f is a scalar field. Then,

$$\frac{\partial}{\partial \vec{A}}\left[\vec{B}\cdot(\vec{\nabla}\times\vec{A})\right] = \vec{\nabla}\times\vec{B} \tag{2.2.8}$$

$$\frac{\partial}{\partial \vec{A}}\left[(\vec{\nabla}\cdot\vec{A})^2\right] = -2\vec{\nabla}(\vec{\nabla}\cdot\vec{A}) \tag{2.2.9}$$

$$\frac{\partial}{\partial \vec{A}}\left[\vec{B}\cdot\left(\vec{\nabla}\times(f\vec{A})\right)\right] = f(\vec{\nabla}\times\vec{B}) - (\vec{\nabla}f\times\vec{B}) \tag{2.2.10}$$

$$\frac{\partial}{\partial f}\left[\vec{\nabla}\times(f\vec{A})\right] = \vec{\nabla}\times\vec{A}. \tag{2.2.11}$$

With these rules in mind, the remaining derivatives in Equations (2.2.2-7) are relatively straightforward. The direct product between two vector quantities (without an explicit dot product) should be interpreted as dyads for the remainder of this section, so as to imply contraction along the appropriate indices without explicit Einstein summation notation. In order, the remaining derivatives are

$$\frac{\partial \mathcal{H}_A}{\partial \vec{\Pi}_Q} = \vec{\Pi}_Q + (\vec{\nabla}\times\vec{P}) + \frac{\partial u_\lambda}{\partial \vec{\Pi}_Q}\varpi + \frac{\partial \vec{u}}{\partial \vec{\Pi}_Q}\cdot\vec{\Pi}_P \tag{2.2.12}$$

$$\frac{\partial \mathcal{H}_A}{\partial \vec{Q}} = -\vec{\nabla}(\vec{\nabla}\cdot\vec{Q}) + 4\lambda\vec{Q} + \frac{\partial u_\lambda}{\partial \vec{Q}}\varpi + \frac{\partial}{\partial \vec{Q}}(\vec{u}\cdot\vec{\Pi}_P) \tag{2.2.13}$$

$$\frac{\partial \mathcal{H}_A}{\partial \vec{P}} = \vec{\nabla}\times\vec{\Pi}_Q - 4\lambda\vec{P} + \frac{\partial u_\lambda}{\partial \vec{P}}\varpi + \frac{\partial \vec{u}}{\partial \vec{P}}\cdot\vec{\Pi}_P \tag{2.2.14}$$

$$\frac{\partial \mathcal{H}_A}{\partial \lambda} = 2Q^2 - 2P^2 - b + \frac{\partial u_\lambda}{\partial \lambda}\varpi + \frac{\partial \vec{u}}{\partial \lambda}\cdot\vec{\Pi}_P, \tag{2.2.15}$$

where the derivatives of u_λ are

$$\frac{\partial u_\lambda}{\partial \vec{\Pi}_Q} = -\frac{\lambda\vec{Q}}{P^2} \tag{2.2.16}$$

$$\frac{\partial u_\lambda}{\partial \vec{Q}} = -\frac{1}{P^2}\left[\lambda\vec{\nabla}\times\vec{P} - (\vec{\nabla}\lambda)\times\vec{P} + \lambda(\vec{\Pi}_Q + \vec{\nabla}\times\vec{P})\right] \tag{2.2.17}$$

$$\frac{\partial u_\lambda}{\partial \vec{P}} = \frac{2\vec{P}}{P^4}\left[\lambda\vec{Q}\cdot(\vec{\Pi}_Q + \vec{\nabla}\times\vec{P})\right] \tag{2.2.18}$$

$$\frac{\partial u_\lambda}{\partial \lambda} = -\frac{1}{P^2}\left[\vec{P}\cdot(\vec{\nabla}\times\vec{Q}) + \vec{Q}\cdot(\vec{\Pi}_Q + \vec{\nabla}\times\vec{P})\right] \tag{2.2.19}$$

and the derivatives of $\vec{u}$ are

$$\frac{\partial \vec{u}}{\partial \vec{\Pi}_Q} = -\frac{\vec{P}}{\lambda}\frac{\partial u_\lambda}{\partial \vec{\Pi}_Q} \tag{2.2.20}$$

$$\frac{\partial}{\partial\vec{Q}}(\vec{u}\cdot\vec{\Pi}_P) = -\frac{1}{\lambda}\left[\frac{\partial u_\lambda}{\partial\vec{Q}}(\vec{P}\cdot\vec{\Pi}_P) + \lambda\vec{\nabla}\times\vec{\Pi}_P - (\vec{\nabla}\lambda)\times\vec{\Pi}_P\right] \tag{2.2.21}$$

$$\frac{\partial\vec{u}}{\partial\vec{P}} = -\frac{1}{\lambda}\left[\frac{\partial u_\lambda}{\partial\vec{P}}\vec{P} + u_\lambda\overleftrightarrow{I}\right] \tag{2.2.22}$$

$$\frac{\partial\vec{u}}{\partial\lambda} = -\frac{1}{\lambda^2}\left[\lambda(\frac{\partial u_\lambda}{\partial\lambda}\vec{P} + \vec{\nabla}\times\vec{Q}) - u_\lambda\vec{P} - \vec{\nabla}\times(\lambda\vec{Q})\right], \tag{2.2.23}$$

where $\overleftrightarrow{I}$ is the unit dyadic.

A number of simplifications can be made to these equations in practice. First of all, while we needed to retain weakly constrained quantities for the purposes of differentiation, we can now safely impose $\vec{\Pi}_P = 0$ and $\varpi = 0$. We also note that $X = b$ by definition. Thus, the simplest form of the equations of motion are

$$\frac{\mathrm{d}\vec{Q}}{\mathrm{d}t} = \vec{\Pi}_Q + \vec{\nabla}\times\vec{P} \tag{2.2.24}$$

$$\frac{\mathrm{d}\vec{\Pi}_Q}{\mathrm{d}t} = \vec{\nabla}(\vec{\nabla}\cdot\vec{Q}) - 4\lambda\vec{Q} \tag{2.2.25}$$

$$\frac{\mathrm{d}\vec{P}}{\mathrm{d}t} = \vec{u} \tag{2.2.26}$$

$$\frac{\mathrm{d}\vec{\Pi}_P}{\mathrm{d}t} = 4\lambda\vec{P} - \vec{\nabla}\times\vec{\Pi}_Q \tag{2.2.27}$$

$$\frac{\mathrm{d}\lambda}{\mathrm{d}t} = u_\lambda \tag{2.2.28}$$

$$\frac{\mathrm{d}\varpi}{\mathrm{d}t} = b + 2P^2 - 2Q^2 = b - X = 0. \tag{2.2.29}$$

In addition, $\frac{\mathrm{d}\vec{\Pi}_P}{\mathrm{d}t} = 0$ because $\vec{\Pi}_P = 0$. This fact uniquely determines λ as

$$\lambda = \frac{1}{4P^2}(\vec{P}\cdot\vec{\nabla}\times\vec{\Pi}_Q), \tag{2.2.30}$$

so long as $P^2 \neq 0$.

2.3 Initial Conditions

Generating initial field values will be a necessary step in constructing a numerical simulation of the field evolution. While purely random field values could be used in principle, they are unlikely to preserve the constraints placed on the system in practice. In this section, we determine a procedural method to generate an initial field state.

In addition to the ten fields necessary to fully describe the system (the components of $\vec{P}$, $\vec{Q}$, and $\vec{\Pi}_Q$, and the scalar field λ), there are four equations which constrain the values which these fields can attain. In particular, we have

$$\begin{aligned} b &= 2Q^2 - 2P^2 \\ u_\lambda &= -\frac{1}{P^2}\left[\vec{P}\cdot\vec{\nabla}\times(\lambda\vec{Q}) + \lambda\vec{Q}\cdot(\vec{\Pi}_Q + \vec{\nabla}\times\vec{P})\right] \\ \vec{u} &= -\frac{1}{\lambda}\left[u_\lambda\vec{P} + \vec{\nabla}\times(\lambda\vec{Q})\right] \\ \lambda &= \frac{1}{4P^2}(\vec{P}\cdot\vec{\nabla}\times\vec{\Pi}_Q). \end{aligned}$$

For the purposes of this analysis, we will treat the constant b as having been determined, since it is a measure of the norm of the tensor B_{ab} in arbitrary units. The existence of these equations leaves only six degrees of freedom in the initial conditions of the system. Therefore, we generate six fields randomly and determine the others from the random fields. We will choose to determine the fields in the following way:

1. Generate $\vec{\Pi}_Q$ under the condition that the resulting field has a nonzero curl everywhere. Step 5 shows why we need this condition. We were unable to find a reliable algorithm capable of generating such a field, and have instead used a determined form for $\vec{\Pi}_Q$ with the hopes of randomizing it under equilibration, as detailed in Section 3.2.1.

2. Generate $\vec{Q}$ randomly.

3. Determine the direction of $\vec{P}$ by noticing that

$$\vec{P} = \frac{\vec{\nabla} \times \vec{\Pi}_Q}{4\lambda} \tag{2.3.1}$$

   implies that $\vec{P}$ and $\vec{\nabla} \times \vec{\Pi}_Q$ share a direction.

4. Determine the magnitude of $\vec{P}$ according to

$$|\vec{P}| = \sqrt{\vec{Q}^2 - \frac{b}{2}}. \tag{2.3.2}$$

   At this point, we see that it will be most convenient to take b to be a negative constant to guarantee that the magnitude of $\vec{P}$ is real-valued. It is worth noting that a negative value for b is analogous to a timelike vector field. Likewise, a positive b would be analogous to a spacelike vector field. In principle, b can take any nonzero value, but we choose to define $b \equiv -1$ in arbitrary units for the duration of this work. Now, $\vec{P}$ is fully determined.

5. Determine λ according to

$$\lambda = \frac{1}{4P^2}(\vec{P} \cdot \vec{\nabla} \times \vec{\Pi}_Q). \tag{2.3.3}$$

   Note that this definition of λ requires $P^2 \neq 0$, as discussed previously. However, we can now see that this restriction also imposes the requirement that $\vec{\Pi}_Q$ has a nonzero curl, such that $\lambda \neq 0$. This condition was already imposed for consistency in the determined form of $\vec{u}$, c.f. Equation 2.1.16.

This process results in the full specification of the initial conditions for the simulation.

2.4 Monopole Definition and Detection

In this work, we are ultimately interested in counting the number of monopole topological defects present in a given volume of space. To do so, it will be necessary to create an algorithm capable of determining whether a given point contains a monopole defect. Because our study is confined to a discrete computer simulation, we will first consider monopoles in a continuous context and then extend that result to the discrete case. In this section, we give a rigorous definition of a monopole and outline two algorithms for their detection.

Our simulation runs on a discrete grid of finite size, with periodic boundary conditions to remove any complications that may be caused by having a boundary in the arguments that follow. These periodic boundary conditions also recover the ability to take gradients, divergences, and curls of the fields near the boundaries of the grid. To simplify our wording, we let 'grid-space' refer to the set of points in physical space (or in our case, the discretized simulation grid), and we let 'field-space' refer to the set of values for the fields $\vec{P}$, $\vec{Q}$, $\vec{\Pi}_Q$, and λ at each point in grid-space.

2.4.1 Monopole Definition

Consider the map represented by the tensor $B_{ab} : \mathbb{R}^3 \to \mathbb{F}$, where $\mathbb{F}$ represents 6-dimensional field-space. This map formalizes the idea that the tensor field relates each point in grid-space to a 6-dimensional collection of field values. Because this tensor can be represented by a constrained combination of $\vec{P}$ and $\vec{Q}$, we will consider field-space to be $\mathbb{F} = \{\mathbb{R}^6 | 2\vec{Q}^2 - 2\vec{P}^2 = b\}$. This set is effectively 5-dimensional, since it has six free variables (the components of $\vec{P}$ and $\vec{Q}$) and one constraint. This set is also homeomorphic to $\mathbb{S}^2 \times \mathbb{R}^3$ [8], and there exists a deformation retraction from $\mathbb{F}$ to the sub-manifold homotopic to $\mathbb{S}^2$ on which $Q^2 = 0$. This sub-manifold then also constrains $P^2 = -b/2$ (recall that we have taken $b < 0$). Regarding the domain of the map, we consider the set of points on a sphere 'at infinity', which is homeomorphic to $\mathbb{S}^2$.

With the above in mind, we consider the inclusion map $\iota : \mathbb{S}^2 \to \mathbb{R}^3$, the tensor $B_{ab} : \mathbb{R}^3 \to \mathbb{F}$, and the deformation retraction $r : \mathbb{F} \to \mathbb{S}^2$ as a single map $B_{ab} : \mathbb{S}^2 \to \mathbb{S}^2$, i.e. from a 2-sphere onto itself (note the abuse of notation, effectively redefining B_{ab}). The degree of a map from an N-sphere onto itself is well defined, is constrained by definition to be an integer, and is equivalent to the winding number of the image in field-space [22]. At last, we can define a monopole topological defect as occurring somewhere within grid-space when the degree of this map is nonzero, i.e. the image in field-space (constricted to $\vec{P}$-space) has a nonzero winding number. Note that the degree is a generalization of the concept of a winding number to higher dimensions. By restricting the scope of our consideration to a subset of grid-space and otherwise arguing as above, we can make a statement about whether a monopole is contained at a given 'point' (i.e. within some small region of grid-space).

In certain cases, we will divide monopole topological defects into sub-types; depending on whether the degree of the map is positive or negative, we will call the topological defect a 'monopole' or an 'anti-monopole', respectively. In addition, we will include the magnitude of the degree as a qualifier, e.g. a monopole arising from a map with degree 2 would be called a 'degree 2 monopole'.

We are now prepared to outline an algorithm to determine the location of monopoles in grid-space. Two methods are presented, specifically a convex hull method (determined to be flawed in Section 5.5), and a triangulation method which resolves the drawbacks of the convex hull method.

2.4.2 Convex Hull Method

We begin with the flawed convex hull method. We claim that, for any set of eight adjacent points in grid-space forming a cube, a monopole is contained within the cube if the corresponding convex hull in field-space contains the origin.

Justification First, consider the map $B_{ab} : \mathbb{S}^2 \to \mathbb{S}^2$ as defined earlier in this section. This map is continuous and, for any given point $x \in \mathbb{R}^3$, associates every point on a sphere around x with a point on some 2-dimensional manifold in field-space. Note that only the values of $\vec{P}$ will determine the structure of this manifold, as $\vec{Q} = 0$ by construction of the map. The degree of this map is equivalent to the winding number of the resulting manifold. Suppose the origin is not enclosed by the manifold in field-space. Then, the winding number of the manifold is trivially 0, and there is no monopole located at x. Instead, suppose the manifold does include the origin. Then, the winding number must be nonzero, and there is a monopole located at x.

We simulate the behavior of this field on a discrete grid, and so it is necessary to approximate the continuous behavior of the map in a discrete way. Let the point x be the center of each cubic grid cell formed by eight adjacent grid points. Consider the sphere around x which passes through the corners of the cube in which x is contained. Each of those eight points map to a particular value of $\vec{P}$ in field-space. We form a convex hull from those field values to approximate the manifold, i.e. the simplest convex polygon enclosing every field value. We can now determine whether the origin is contained within the convex hull, and thus determine whether there is a monopole at the point x.

Issue The convex hull method has two easily-identifiable problems. First, we can only say whether the degree of the map is zero or nonzero, giving no way to distinguish between monopoles and anti-monopoles, nor to investigate the possibility of higher degrees of monopole. In addition, the definition of a convex hull does not account for the ordering of the vertices, which is to say that the implementation of the convex hull algorithm in SciPy [23] (as a wrapper for QHull [24]), will generate the same convex hull regardless of the relative positions of the vertices of the cube in grid-space. This loss of geometric information counts non-monopole objects composed of points which happen to enclose the origin, but which, when accounting for the ordering of the cube vertices, do not have a nonzero winding number. Thus, the convex hull method overestimates the number of monopoles in practice. See Section 4.3.2 for more information.

2.4.3 Triangulation Method

Motivated by the inherent issues present in the convex hull method, we instead consider a triangulation method which accounts for the geometric information provided by the relative placement of cube vertices in grid-space and directly calculates the degree of the map. We claim that the degree of the map can be found by considering a triangulation of each cubic grid cell in grid-space and the map of that triangulation into field-space, as motivated by Definition 1.1.3 in Lloyd's *Degree Theory* [25]. Let D be a bounded, open subset of $\mathbb{R}^n$, p a point in $\mathbb{R}^n$, $C(\bar{D})$ the linear space of continuous functions from $\bar{D}$ to $\mathbb{R}^n$, and $C^1(\bar{D})$ the subspace of $C(\bar{D})$ such that $f \in C^1(\bar{D})$ if $f \in C(\bar{D})$ has continuous first order partial derivatives in $\bar{D}$. Note that these definitions are slightly simplified relative to those used by Lloyd.

Definition 1.1.3 from Lloyd Suppose that $\phi \in C^1(\bar{D})$, $p \notin \phi(\partial D)$, and $p \notin$ crease ϕ. The meaning of crease ϕ here is the set of all points at which ϕ is not well approximated to be locally linear. Define the degree of ϕ at p relative to D to be $d(\phi, D, p)$, where

$$d(\phi, D, p) = \sum_{x \in \phi^{-1}(p)} \text{sign } J_\phi(x). \tag{2.4.1}$$

The sign of the Jacobian counts +1 or -1, depending on whether ϕ is orientation preserving or reversing near x, respectively.

Method Consider the eight points forming each cubic grid cell in grid-space. Divide the surface of each cube into oriented, right-handed triangles. Twelve triangles are required for this process. The triangulation used in this work is displayed graphically in Figure 2.1, and is oriented with respect to the exterior of the cube. Note that the strange ordering of vertices was chosen to simplify algorithms, and is not necessarily the easiest visual representation. Choose one triangle, and consider its mapping onto the surface of a sphere in field-space. Let p be the point at the center of this spherical triangle, so it is sufficiently far from any boundaries. By the definition of the degree given previously, we have that

$$\deg B_{ab} = \sum_{x \in B_{ab}^{-1}(p)} \text{sign } J_B(x), \tag{2.4.2}$$

where the sign of the Jacobian at x should be taken to determine whether the map is orientation preserving or reversing near x. It should be noted that in subdividing the cube into triangles, we have also subdivided the sphere in field-space into spherical triangles. Therefore, it is logical to consider the sign of the Jacobian to be an indicator of whether a given spherical triangle containing p is orientation preserving or reversing.

In practice, it is relatively easy to simultaneously determine the sign of the Jacobian for a given spherical triangle and whether that spherical triangle contains p. Consider a spherical triangle specified by the points $(\vec{v}_1, \vec{v}_2, \vec{v}_3)$, traced in that order. A quick geometrical argument shows that the quantity

$$h(\vec{v}_1, \vec{v}_2, \vec{v}_3) = \text{sign}[(\vec{v}_1 \times \vec{v}_2) \cdot \vec{v}_3] = \text{sign}[\det(\vec{v}_1, \vec{v}_2, \vec{v}_3)] \tag{2.4.3}$$

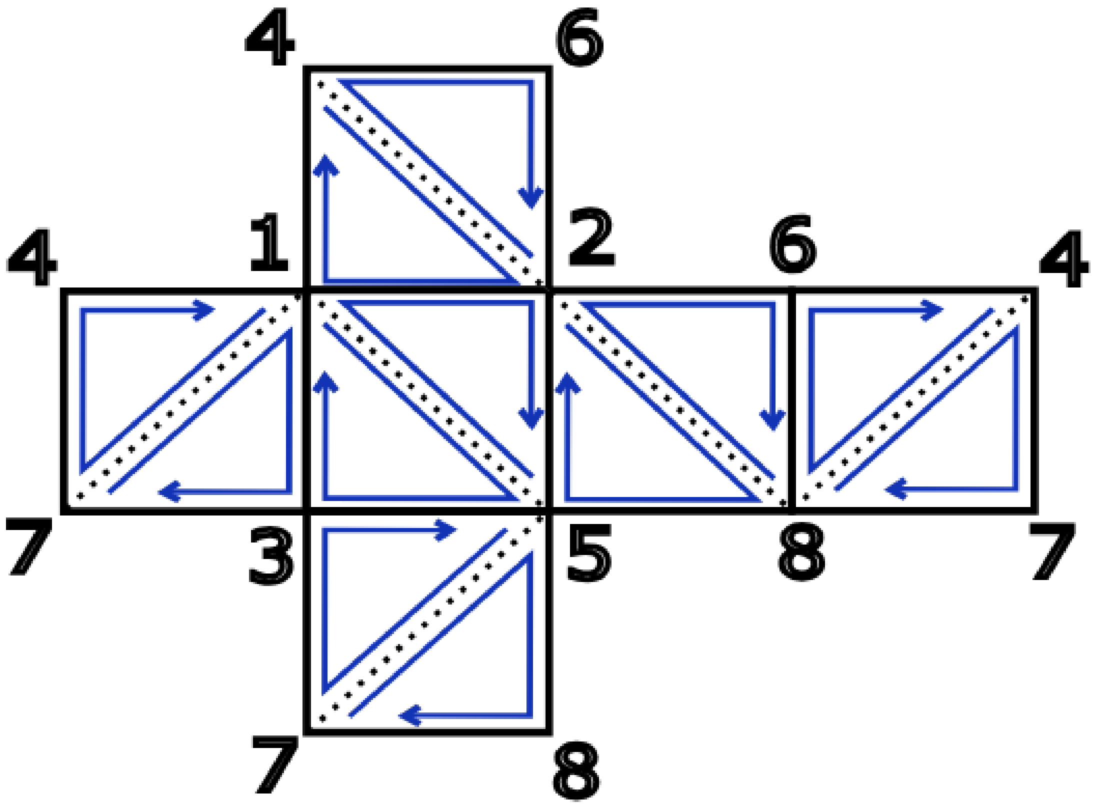

Figure 2.1: The triangulation of a cube (flattened into a net) used in this work, with orientations labelled.

is an indicator of orientation, yielding $h(\vec{v}_1, \vec{v}_2, \vec{v}_3) = 1$ in the case of a right-handed orientation and $h(\vec{v}_1, \vec{v}_2, \vec{v}_3) = -1$ in the case of a left-handed orientation (see Figure 2.2 and note the directions of $\vec{v}_3$ and $\vec{v}_1 \times \vec{v}_2$). Since we originally defined each spherical triangle to be right-handed, a value of $h = 1$ represents no reversal, and $h = -1$ represents a reversal. Therefore, in some sense h and sign J_B are equivalent. Furthermore, if p is within the spherical triangle then the three spherical triangles specified by $(\vec{v}_1, \vec{v}_2, \vec{p})$, $(\vec{v}_2, \vec{v}_3, \vec{p})$, and $(\vec{v}_3, \vec{v}_1, \vec{p})$ must subdivide the original, and thus have the same orientation and value for h. If any of these three spherical triangles have the opposite orientation, then p must not be in the original spherical triangle. Thus, if

$$h(\vec{v}_1, \vec{v}_2, \vec{v}_3) = h(\vec{v}_1, \vec{v}_2, \vec{p}) = h(\vec{v}_2, \vec{v}_3, \vec{p}) = h(\vec{v}_3, \vec{v}_1, \vec{p}),$$

then the spherical triangle formed by $(\vec{v}_1, \vec{v}_2, \vec{v}_3)$ contributes a value of $h(\vec{v}_1, \vec{v}_2, \vec{v}_3)$ to the sum, and otherwise contributes 0. The combination of these facts allows for the simple calculation of Equation (2.4.2).

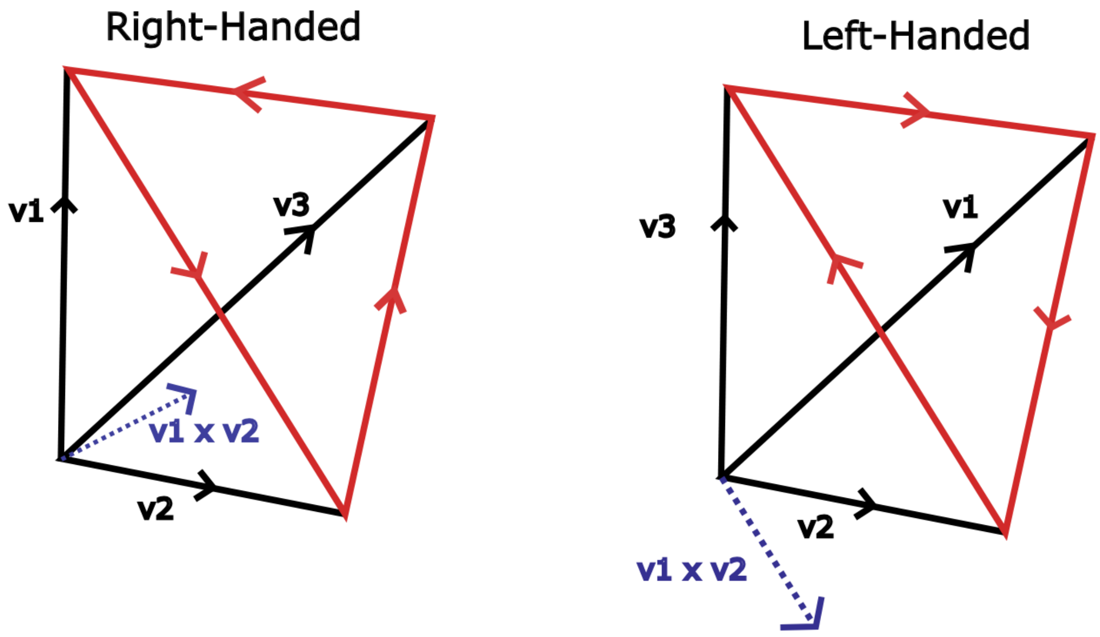

Figure 2.2: A left-handed and right-handed triangle demonstrating the relative directions of $\vec{v}_3$ and $\vec{v}_1 \times \vec{v}_2$.

2.4.4 Preservation of Topological Order

Through a short argument, we will show that the total degree of all monopoles, i.e. the sum of the degrees of all monopoles minus the degrees of all anti-monopoles, must be 0. We will refer to this principle as the conservation of topological order. This principle is only useful if the degree of a given monopole is known, so we will refer to the triangulation method described above in the argument. While the argument presented relies on a specific detection algorithm, all monopole detection methods should detect the same monopoles, so the principle should also be more general than the specific case considered.

Justification Consider a generic cube in grid-space formed by eight adjacent points. Since the simulation grid includes periodic boundary conditions, every cube has a neighbor adjacent to each of its faces. Consider the triangulation of each cube, which will have two triangles per face. These triangles will align with the triangles on adjacent faces (note that the triangulation used in this work was chosen in part to exhibit this property). Consider taking the degree of the map over all of grid-space. Then, all triangles which are counted in Equation (2.4.2) will come in pairs of triangles aligned on adjacent faces. Both triangles will be counted because their coordinates in field-space are identical, so if $\vec{p}$ can be said to be within one, then it is within the other as well. Since the triangles are oriented with respect to the exterior of their cube, each pair of triangles will have opposite orientations. Thus, the overall degree will exactly cancel[2].

2.5 Energy Damping

The current version of our simulation does not consider the expansion of the Universe as a simplifying assumption. Therefore, energy can only be removed from the system through an *ad hoc* method. However, removing energy from the system is a necessary condition for the Kibble Mechanism [10] [11]. In this section, we present an *ad hoc* method to remove energy from the system while preserving conserved quantities.

Recall that the single-invariant system has a Hamiltonian which is equal to

$$\mathcal{H}_A = \frac{1}{2}\Pi_Q^2 + \vec{\Pi}_Q \cdot (\vec{\nabla} \times \vec{P}) + \frac{1}{2}(\vec{\nabla} \cdot \vec{Q})^2 \tag{2.5.1}$$

after accounting for all weakly constrained quantities. Taking the time derivative of the energy directly, we see that

$$\frac{\mathrm{d}\mathcal{H}_A}{\mathrm{d}t} = \vec{\Pi}_Q \cdot \frac{\mathrm{d}\vec{\Pi}_Q}{\mathrm{d}t} + (\vec{\nabla} \times \vec{P}) \cdot \frac{\mathrm{d}\vec{\Pi}_Q}{\mathrm{d}t} + ... = \frac{\mathrm{d}\vec{\Pi}_Q}{\mathrm{d}t} \cdot (\vec{\Pi}_Q + \vec{\nabla} \times \vec{P}) + ..., \tag{2.5.2}$$

where the remaining terms are importantly not dependent on $\frac{\mathrm{d}\vec{\Pi}_Q}{\mathrm{d}t}$. By virtue of being a conserved quantity, the time derivative of the system energy is defined to be 0 in its current form. To successfully damp the system energy, it will be necessary to include an additional term such that the system energy is negative-definite. To this end, consider modifying $\frac{\mathrm{d}\vec{\Pi}_Q}{\mathrm{d}t}$ by inserting an additional term of the form $-\epsilon(\vec{\Pi}_Q + \vec{\nabla} \times \vec{P})$, where ϵ is a positive damping coefficient. Then, the time derivative of the Hamiltonian becomes

$$\frac{\mathrm{d}\mathcal{H}_A}{\mathrm{d}t} = \left(\frac{\mathrm{d}\vec{\Pi}_Q}{\mathrm{d}t} - \epsilon(\vec{\Pi}_Q + \vec{\nabla} \times \vec{P})\right) \cdot (\vec{\Pi}_Q + \vec{\nabla} \times \vec{P}) + ... = -\epsilon(\vec{\Pi}_Q + \vec{\nabla} \times \vec{P})^2, \tag{2.5.3}$$

where all other terms have reduced to 0 by the conservation of system energy without damping. It is now clear that our modified system energy has a negative-definite time derivative, as desired. Thus, damping is achieved by redefining the time derivative of $\vec{\Pi}_Q$ to be

$$\frac{\mathrm{d}\vec{\Pi}_Q}{\mathrm{d}t} = \vec{\nabla}(\vec{\nabla} \cdot \vec{Q}) - 4\lambda\vec{Q} - \epsilon(\vec{\Pi}_Q + \vec{\nabla} \times \vec{P}). \tag{2.5.4}$$

2.6 Spatial Cross-Correlations

In this section, we detail the calculation of the spatial cross correlation, a quantity ultimately used in our analysis as an indicator of randomness in the field values of our simulation.

[2]We believe this could be proven more formally through the Generalized Stokes' Theorem, but have yet to work out the details of such a proof.

Definition The correlation coefficient between two random variables X and Y is defined to be

$$\rho_{XY} \equiv \frac{\sigma_{XY}}{\sigma_X \sigma_Y}, \tag{2.6.1}$$

where σ_{XY} is the covariance between X and Y and σ_X and σ_Y the standard deviations in X and Y, respectively [26]. The correlation coefficient is preferred to the covariance because it is normalized to lie between -1 and 1, with ± 1 reflecting an exactly linear relationship between X and Y of positive or negative slope, respectively, and 0 representing a purely random (or at least, purely non-linear) relationship between X and Y.

Discrete Correlation Coefficient Within a discrete grid of field points with N total points, we can define the mean value of the random variable X to be

$$\overline{X} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{2.6.2}$$

so that the standard deviation in X is

$$\sigma_X = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{X})^2} \tag{2.6.3}$$

and the covariance between X and Y is

$$\sigma_{XY} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{X})(y_i - \overline{Y}). \tag{2.6.4}$$

These are common definitions for these quantities, c.f. [26]. Alternatively, the standard deviation in X can be expressed as

$$\sigma_X = \overline{X^2} - \overline{X}^2, \tag{2.6.5}$$

i.e. the difference between the mean squared value and the square of the mean value [26]. Then, the correlation coefficient is given by

$$\rho_{XY} = \frac{\sigma_{XY}}{\sigma_X \sigma_Y} = \frac{\sum_{i=1}^{N}(x_i - \overline{X})(y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{N}(x_i - \overline{X})^2}\sqrt{\sum_{i=1}^{N}(y_i - Y)^2}} = \frac{1}{N}\frac{\sum_{i=1}^{N}(x_i - \overline{X})(y_i - \overline{Y})}{(\overline{X^2} - \overline{X}^2)(\overline{Y^2} - \overline{Y}^2)}. \tag{2.6.6}$$

In particular, we are interested in the spatial cross correlation, which is a measure of the dependence of the field value at a given point to that of nearby points.

Definition Let i denote some value sampled from a scalar field across all of grid-space. It follows that the average value of i will be the same as the average of the field as a whole, call it $\overline{f}$. Let J be the set of all field points at a Manhattan distance r of i. The points in J are also sampled from the whole of field-space, so we take their average to be $\overline{f}$ as well. Then, we can calculate the spatial cross correlation between i and all points within a given distance r of i via the sum

$$\rho_i(r) = \frac{1}{\left(\overline{f^2} - \overline{f}^2\right)^2}\frac{1}{|J|}\sum_{j\in J}(f_i - \overline{f})(f_j - \overline{f}). \tag{2.6.7}$$

If we compute this value for all N points in the grid, we can find the average spatial cross correlation for a given radius via

$$\rho(r) = \frac{1}{N}\sum_{i=1}^{N}\rho_i(r). \tag{2.6.8}$$

The spatial cross correlation will serve as an indicator of overall grid coupling, with values close to 0 indicating a relatively random field and values closer to 1 indicating interactions between field points at some distance, for example creation and annihilation of monopole and anti-monopole pairs [26].

Methods

3.1 Data Collection

This simulation was run on the Grace cluster at Yale ('the cluster'), with resources provided to us by the Yale Center for Research Computing. In total, approximately 40,000 CPU hours were used across 330 instances of the simulation. Instances 1 through 50, 151 through 180, and 241 through 270 were allowed to equilibrate for 100 simulation iterations. After equilibration, instances 1 through 50 were replicated twice to create the basis for instances 51 through 150. Likewise, instances 151 through 180 were replicated to create instances 181 through 240, and instances 241 through 270 were replicated to create instances 271 through 330. Once the simulation states for all 330 iterations were prepared, damping was activated for as long as possible, varying from 200 to 250 additional simulation iterations. The reason for differing simulation durations is the increased computational overhead caused by larger grid sizes. The grid sizes and damping rates for each set of instances are summarized in Table 3.1. Note that we have sampled a range of grid sizes and damping rates to gain some understanding of how these parameters impact the results of our simulation, if at all.

Run Indices	Grid Size (x, y, z)	Damping Constant (ϵ)
1-50	20×20×20	10^{-4}
51-100	20×20×20	10^{-3}
101-150	20×20×20	10^{-2}
151-180	10×10×10	10^{-4}
181-210	10×10×10	10^{-3}
211-240	10×10×10	10^{-2}
241-270	15×15×15	10^{-4}
271-300	15×15×15	10^{-3}
301-330	15×15×15	10^{-2}

Table 3.1: Grid Sizes and Damping Constants for all 330 Simulation Instances

3.2 Simulation Overview

The results presented in this work, as well as much of the data used during development, were collected by the simulation we developed to track the behavior of all ten fields in the single-invariant system as governed by their time evolution equations (See Section 2.2). All code used to generate these results, as well as all information regarding packages and their versions used, is provided in Appendix A.3, online at the Connecticut College Digital Commons[3], and via GitHub[4].

The simulation was primarily composed of two Python program files, Simulation.py and TimeEvolutionEquations.py, which are the primary focus of this section. Broadly speaking, Simulation.py was

[3]digitalcommons.conncoll.edu/

[4]github.com/WyattCarbonell/UndergradThesis

responsible for handling initial simulation conditions and general setup procedures, whereas TimeEvolutionEquations.py implemented the mathematics of the field evolution in the format of the py-pde library [27] as a wrapper for the Scipy library [23].

3.2.1 Simulation.py

Simulation.py primarily implemented the simulation method, which set up one instance of the field simulation. The exact configuration of the simulation was determined by the supplied parameters, which control the simulation duration, grid dimensions, and value of the constant ϵ used for damping energy. An existing simulation could be continued from its last state by providing the location of a file containing the relevant field parameters. Finally, a parameter called 'run' was used as a kind of 'serial number' to distinguish between data files for each instance of the simulation.

These operators were all calculated through methods defined by the py-pde package, with the exception of the curl of a vector field. We defined the curl of a vector field to be

$$(\vec{\nabla} \times \vec{P})_x = \frac{\partial P_z}{\partial y} - \frac{\partial P_y}{\partial z} \tag{3.2.1}$$

$$(\vec{\nabla} \times \vec{P})_y = \frac{\partial P_x}{\partial z} - \frac{\partial P_z}{\partial x} \tag{3.2.2}$$

$$(\vec{\nabla} \times \vec{P})_z = \frac{\partial P_y}{\partial x} - \frac{\partial P_x}{\partial y}, \tag{3.2.3}$$

where each partial derivative has been taken to be a component of the gradient calculated by py-pde, c.f. [28].

If a file containing a field state to use as an initial state was provided, those values were loaded. Otherwise, field values were generated as described in Section 2.3, with the exception of $\vec{\Pi}_Q$. Because we were unable to determine a suitable algorithm to generate randomized fields with nonzero curl, a determined form for the components of $\vec{\Pi}_Q$ was used. In particular, we chose $\vec{\Pi}_Q$ to be

$$\vec{\Pi}_Q = \langle 0, \sin(2\pi x/25), -\cos(2\pi x/25)\rangle \,. \tag{3.2.4}$$

We note that the quantity $2\pi x/25$ was originally chosen to force periodicity on a $25 \times 25 \times 25$ grid, but we do not expect this choice to impact our results for other grid sizes. To compensate for this determined form, we included a period for equilibration, allowing the field to effectively randomize itself by evolving away from the determined form.

The initial field values for all ten fields were packaged together into an initial state, which was passed to py-pde alongside an instance of TimeEvolutionEquations (see Section 3.1.2) to handle the calculations for the simulation. Several additional parameters were specified in py-pde. We chose to use an adaptive step size to maintain stability in situations with large changes in field strength over a short period of time. We also used a Runge-Kutta scheme as implemented by py-pde [27] to solve the differential equations due to its observed stability (see Section 4.1). The alternative to a Runga-Kutta scheme would have been a symplectic integrator, which is a numerical method developed specifically for Hamiltonian systems from Hamilton's equations. We would have preferred to use a symplectic integrator for this reason, but SciPy does not offer integrated support for any symplectic methods as of the completion of this work [23]. In particular, symplectic methods are considered ideal when some quantities need to be conserved to within high precision. Since we could not use a symplectic method, we chose to instead monitor the conservation of our constraints.

3.2.2 TimeEvolutionEquations.py

TimeEvolutionEquations.py implemented the mathematics of the field evolution in the form of a TimeEvolutionEquations object. Most results and diagnostic data files were also produced by this file.

The initialization parameters for a TimeEvolutionEquations object were passed by Simulation.py and controlled the exact mechanics of the simulation. For readability, the methods implemented for an instance of TimeEvolutionEquations are detailed separately.

Two systems of time-keeping were used simultaneously throughout this file, which served different purposes. We use 'counts' to track the number of individual calculations of the field evolution which have occurred. This measure is useful for checking that our conserved quantities are stable throughout the calculation process. However, counts are a poor choice for long term monitoring of the simulation, since the introduction of adaptive step sizes causes the actual time between calculations to be inconsistent. To remedy this, we also monitor the current time, 't', which is a measure of the total number of iterations which py-pde has undergone. Iterations are defined internally by py-pde, but can be thought of as some unit of actual time passed (though we have not determined whether the conversion between iterations and seconds is definite). Typically, one count corresponds to $10^{-3} - 10^{-4}$ iterations, varying with the adaptive step size. Counts are integer-valued, and iterations can assume decimal values.

evolution_rate

This method handled the calculation of the field evolution equations derived in Section 2.2, and exported diagnostic data and results. It took an array of all ten fields as input, each of which is best thought of as a 3-dimensional array of field values for every point in grid-space. It returned the time derivative of each field as determined by Equations 2.2.24-29.

This method also handled the exportation of diagnostic and results data, including spatial cross-correlations and monopole counts, which are discussed in detail in Section 3.3.

calculate_hamiltonian

This method calculated the total Hamiltonian for a given simulation state, i.e. the Hamiltonian integrated over all of grid-space. Necessary quantities were calculated using divergence and gradient operators as defined by py-pde, and curl was calculated according to Equations 3.2.1-3.

find_and_export_monopoles_hull

This method counted the number of monopoles in the grid according to the convex hull method outlined in Section 2.4.1. We note that this method only exists for the sake of comparison, and should not be used as a reliable count of the monopoles in the grid, as explained further in Section 2.4.1.

Starting at the point (0, 0, 0) in grid-space, we iterated through every possible choice of cubic region defined by 8 points, each uniquely specified by the first corner. For example, the first cube was composed of the points $(0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1), (1, 1, 0), (1, 0, 1), (0, 1, 1)$, and $(1, 1, 1)$. For each cube, the series of 8 corresponding vectors $\vec{P}$ in field-space were passed to the convex hull algorithm as implemented by SciPy, alongside the origin in field-space. After the hull had been generated, we checked if any edges were visible from the origin using the built-in SciPy object 'hull.good'. We found a monopole if no edges were visible from the origin, indicating that the origin was strictly interior to the hull. This method is counter-intuitive, but relies on the fact that SciPy only considers an edge visible from *outside* the hull, so the existence of a visible edge implies the origin is outside the hull. The total number of monopoles was counted, and stored so it could be exported to a file later. For more information about the exported data, see Section 3.3.

find_and_export_monopoles_triangulation

This method counted the number of monopoles in the grid according to the triangulation method outlined in Section 2.4.3.

Once again starting at the point (0, 0, 0) in grid-space, we iterated through every possible choice of cubic region defined by eight points, each uniquely specified by the first corner. For each cube, the 12 oriented triangles representing the cube in field-space were each tested for orientation reversal and whether they contained $\vec{p}$ (see check_triangle below). We take $\vec{p}$ to be the center of the first triangle, whichever that happens to be. The values of the function h for each triangle were summed, and the degree of the map was thus determined. The total number of monopoles and anti-monopoles of degree up to 7 were counted and stored so their values could be exported to a file later. For more information about the exported data, see Section 3.3.

check_triangle

This method simply implements the orientation checking algorithm described in Section 2.4.3 for a given oriented triangle.

corr_in_grid

This method calculated the average spatial cross correlation over the entire grid for a given vector field, according to Equation 2.6.8. The x, y, and z spatial cross-correlations of the vector field were calculated separately. The spatial cross-correlation for multiple radii were calculated at the same time by passing each radius separately to correlation_at_p.

correlation_at_p

This method implemented the spatial cross correlation of a given point i within a Manhattan distance r as described by Equation 2.6.7.

3.3 Data Files

In this section, we outline the contents and formatting of each type of data file exported by the simulation. In all cases, files are labelled according to their run number.

Diagnostic Data

This file contains a variety of diagnostic data used to ensure that the simulation is behaving as expected, in comma separated values format. We track the integrated magnitudes of the $\vec{P}$, $\vec{Q}$, and λ fields, the Hamiltonian (representing system energy), and the invariants Ψ and the three components of $\vec{\Psi}$. Variances in $\vec{P}$ and $\vec{Q}$ are also included. Measurements of these quantities are taken every 10 simulation steps, and the iteration time of each measurement is included alongside the step number.

Field Data

This file contains all simulation state information necessary to restart the simulation from the last known point, including the last known step number, iteration time, and the values of all ten fields at all grid points. The file is stored in the numpy-zip file format to save on memory space [29], and is updated every 5000 simulation steps. The uncompressed file contains twelve arrays, corresponding in order to the values of all ten field quantities at every grid point (the three components of $\vec{\Pi}_Q$, $\vec{P}$, and $\vec{Q}$, and λ), the current count number, and the current iteration time.

Monopoles

The following two files both contain data related to the total number of monopoles found within the simulation grid. Both files are comma separated values.

Triangulated Monopoles This file contains the count numbers for monopoles and anti-monopoles of degrees one through six, separately, and the total number of monopoles and anti-monopoles of degree at least 7, as determined by the triangulated monopole method described in Section 2.4.2. Measurements are taken approximately every 0.25 iterations, and the exact time and count number for each measurement are included in the file.

Hull Monopoles This file is deprecated. If enabled, the file includes the number of monopoles determined by the convex hull monopole method described in Section 2.4.2. Measurements are taken at the same time as in the previous monopole method, and those iteration times and count numbers are included in this file as well.

Correlations

This file contains the spatial cross-correlations for either the components of $\vec{P}$, $\vec{Q}$, or $\vec{\Pi}_Q$, as appropriately labelled, for radii of 1, 2, 4, 6, and 8 grid cells. Measurements are taken approximately every 5 iterations, and the exact time and count number for each measurement are included in the file. The file format is comma separated values.

Monopole P Vecs

This file is deprecated. If enabled, the file contains sets of eight different $\vec{P}$ which were determined to form a monopole by the convex hull method. The file format is numpy-zip [29], and when decompressed contains an array of sets of eight vectors.

Results

In this section, we present data collected from 330 simulation instances. For brevity, relevant examples have been highlighted in each of the following sections. All remaining data has been relegated to Appendix A.4 to serve as an additional reference, and is generally in agreement with the results presented here.

4.1 Preservation of Conserved Quantities

As discussed in Section 2, the definition of our system implies that some quantities are conserved. In particular, we investigate the conservation of energy, primary and secondary constraints, and topological order.

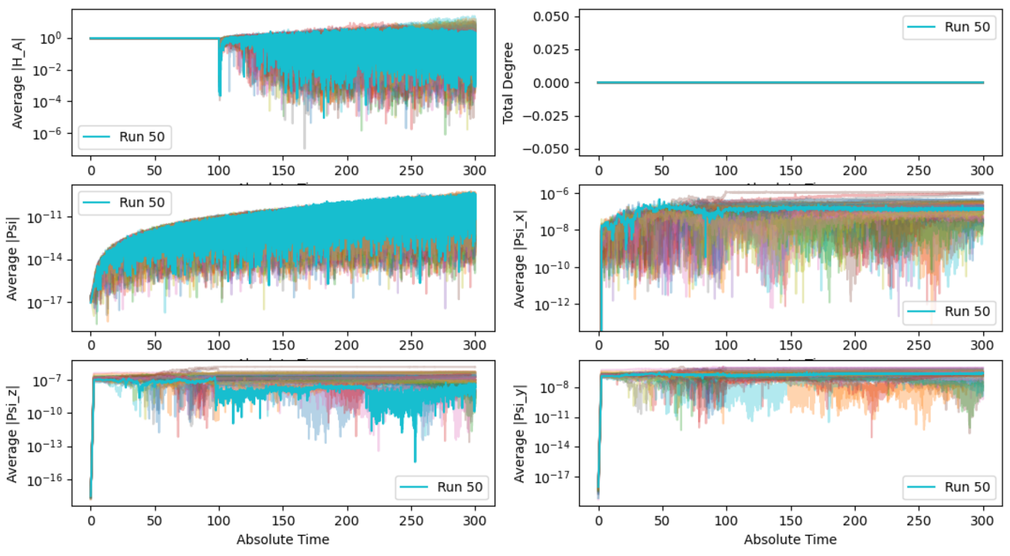

Figure 4.1: Preservation of Conserved Quantities in Instances 1-50

Figures 4.1-3 display the results of instances 1-50, 181-210, and 301-330, respectively. These instances have been highlighted in this section because they represent a variety of grid sizes and damping rates after equilibration; respectively, they correspond to grid sizes of 20x20x20, 10x10x10, and 15x15x15, and damping rates of 10^{-4}, 10^{-3}, and 10^{-2}. In all cases, one instance has been highlighted as an example of typical behavior.

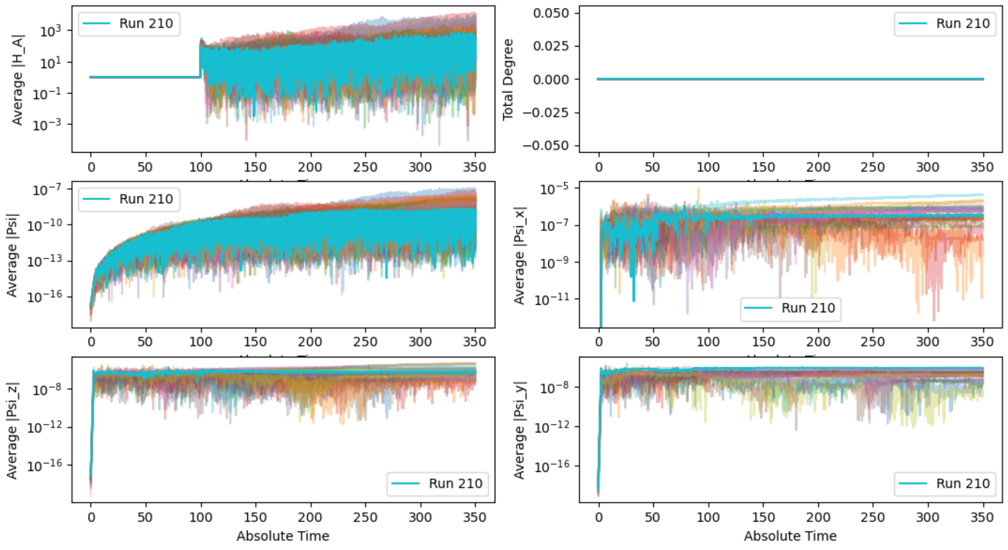

Figure 4.2: Preservation of Conserved Quantities in Instances 181-210

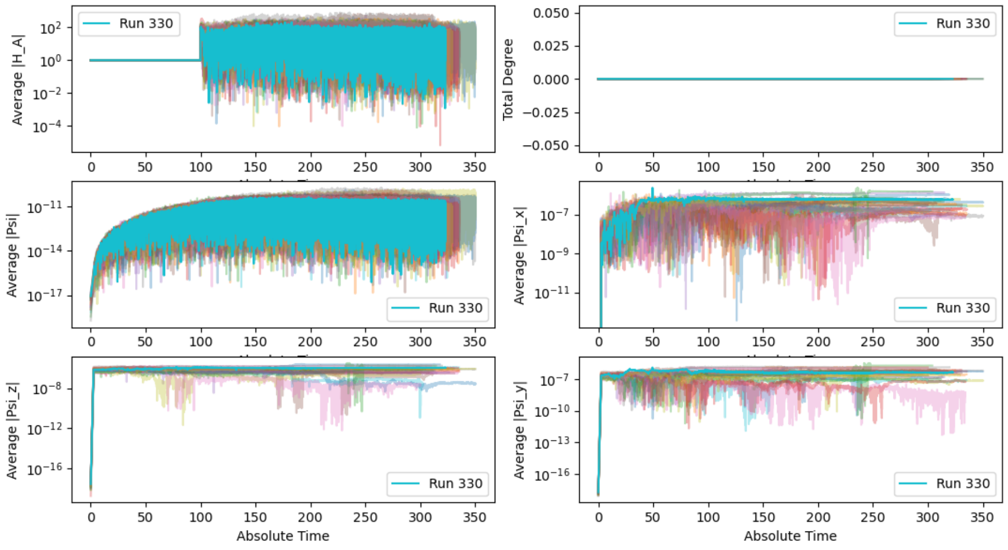

Figure 4.3: Preservation of Conserved Quantities in Instances 301-330

Note that the energy is stable up until 100 iteration seconds have passed in all cases – this instability is easily explained by energetic damping, which was activated at this point and necessarily violates the conservation of energy. Other notable features in these plots at the 100 iteration seconds point should be considered with this fact in mind. Also note that the system energy often appears to increase in the damping process. This effect is a direct result of the use of energy magnitudes (necessary for the logarithmic scale), and is in fact caused by the system settling into large negative energies. Furthermore, the conservation of the invariant X was not directly tracked, since the conservation of Ψ necessitates the conservation of X, by virtue of b being a constant (c.f. Equation 2.1.14).

4.2 Long-Term Field Behavior

From the perspectives of computational viability and physical realism for this model, one would hope that the field values are not divergent. In this section, we present the average magnitudes of all ten simulation fields.

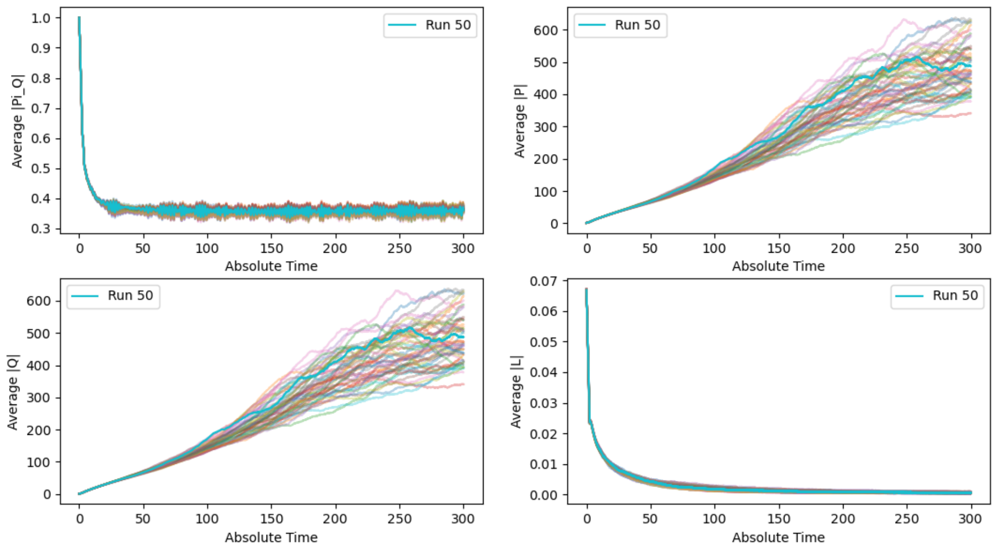

Figure 4.4: Convergence of Field Values in Instances 1-50

Figures 4.4-6 display the results of instances 1-50, 181-210, and 301-330, respectively. These instances have been highlighted in this section because they represent a variety of grid sizes and damping rates after equilibration; respectively, they correspond to grid sizes of 20x20x20, 10x10x10, and 15x15x15, and damping rates of 10^{-4}, 10^{-3}, and 10^{-2}. In all cases, one instance has been highlighted as an example of typical behavior.

Note that the behavior of each field, but in particular $\vec{\Pi}_Q$, may deviate after 100 iteration seconds have passed – this instability is easily explained by energetic damping, which was activated at this point. Also note that damping is generally intended to decrease $\vec{\Pi}_Q$ by introducing a negative term to the time derivative for each component of $\vec{\Pi}_Q$ (c.f. Equation 2.5.4); however, $\vec{\Pi}_Q$ often appears to increase during damping. This effect is a direct result of the use of magnitudes (for the logarithmic scale), and is in fact caused by negative values of $\vec{\Pi}_Q$.

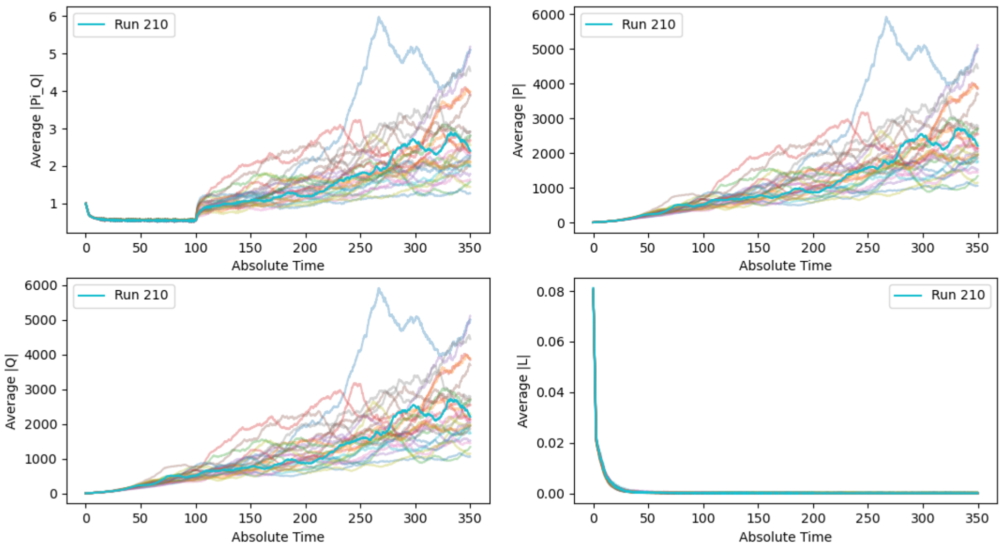

Figure 4.5: Convergence of Field Values in Instances 181-210

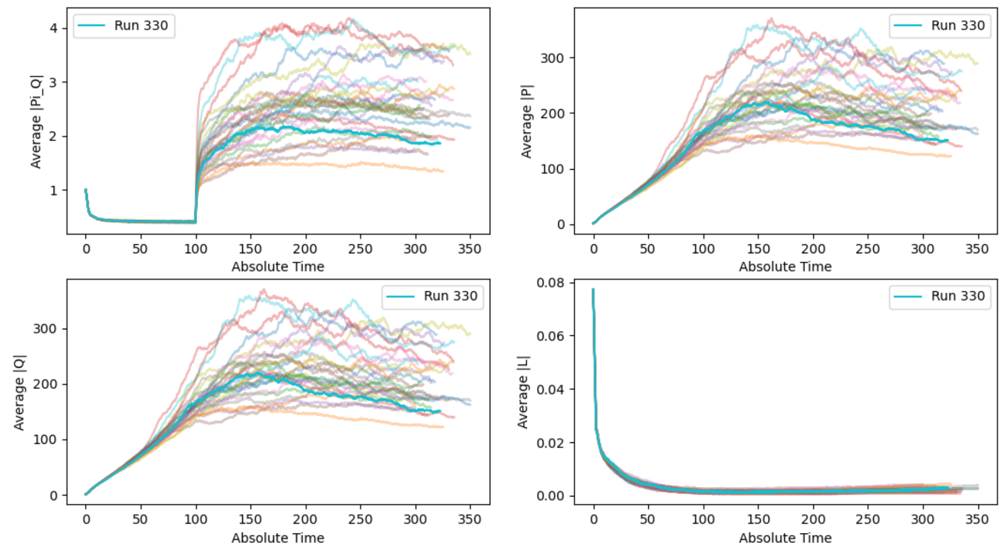

Figure 4.6: Convergence of Field Values in Instances 301-330

4.3 Monopole Behavior

4.3.1 Incidence Rate in Simulation

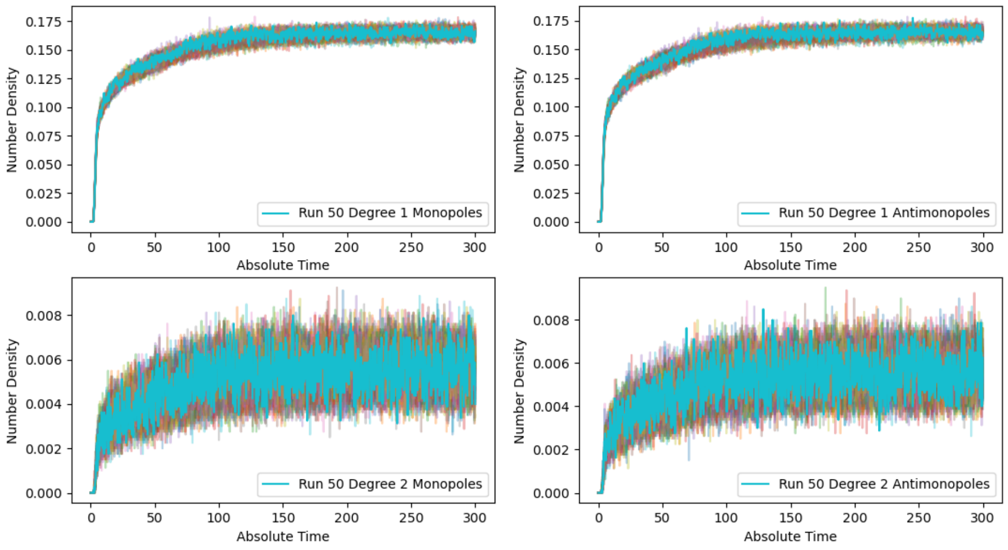

Figure 4.7: Monopole proportions in instances 1-50 for grid size 20x20x20 and damping 10^{-4}.

Using the triangulation method (see Section 2.4.3), we counted the number of monopoles and anti-monopoles of all possible degrees for various simulation parameters. Figures 4.7-15 display the proportion of grid cells containing monopoles and anti-monopoles of degrees 1 and 2.

Notice that the proportion of each type of monopole tends to approximately converge to some value, which may depend on the grid size but is seemingly independent of the damping parameter and whether monopoles or anti-monopoles are considered. Motivated by this observation, Figures 4.16-18 display the number of instances of each monopole count after equilibration (100 iteration seconds) for each grid size and degrees from 1 to 4. Notice that degree 4 monopoles are extremely unlikely, but are possible (as is hard to see in this plot). A total of 981 degree 4 monopoles were detected across all 330 instances of the simulation. The maximum degree of any monopole or anti-monopole measured was 4.

Using this data, we produced best estimates for the average number density of monopoles in our simulation grid after equilibration. We found that degree 1 monopoles account for $16.434 \pm 0.001\%$ of grid cells, degree 2 monopoles account for $0.5506 \pm 0.0003\%$, degree 3 monopoles account for $0.00500 \pm 3 \cdot 10^{-5}\%$, and degree 4 monopoles account for $7.3 \cdot 10^{-5} \pm 3 \cdot 10^{-6}\%$. In total, monopoles of degrees 1 through 4 accounted for $16.990 \pm 0.001\%$ of all grid cells.

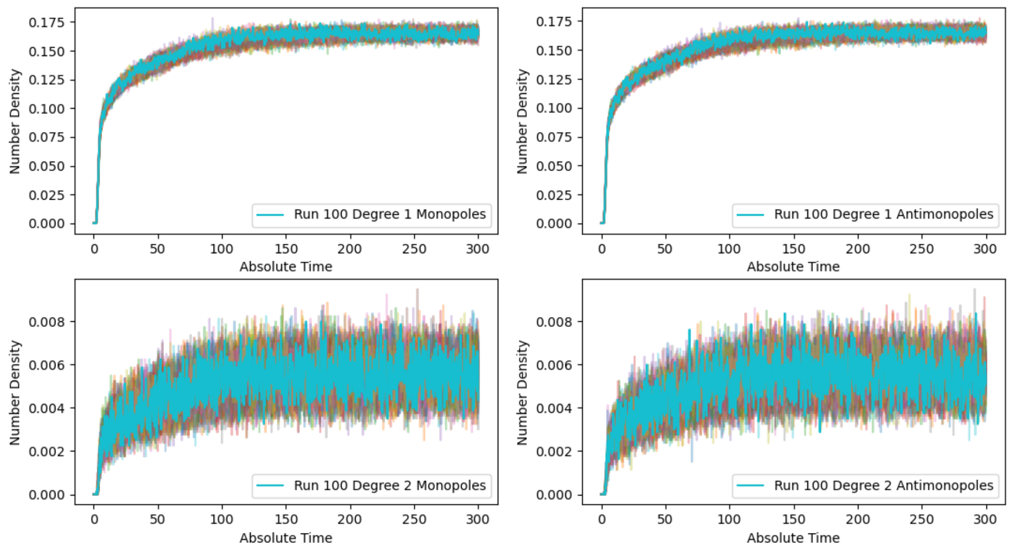

Figure 4.8: Monopole proportions in instances 51-100 for grid size 20x20x20 and damping 10^{-3}.

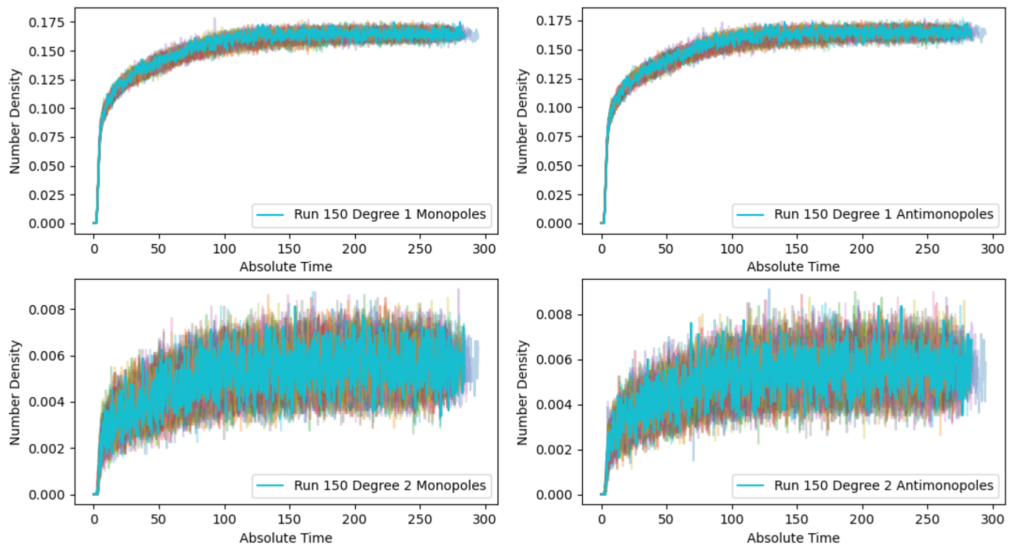

Figure 4.9: Monopole proportions in instances 101-150 for grid size 20x20x20 and damping 10^{-2}.

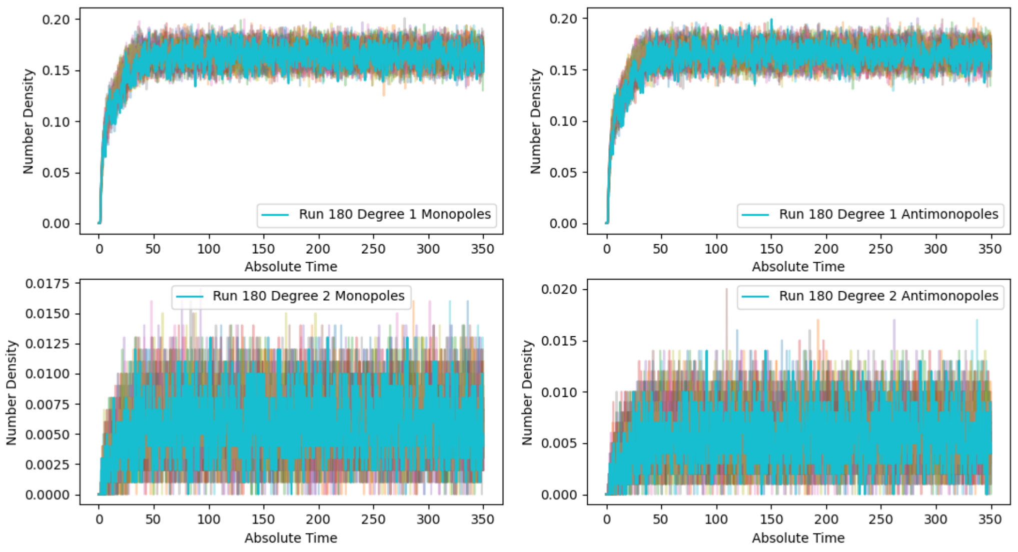

Figure 4.10: Monopole proportions in instances 151-180 for grid size 10x10x10 and damping 10^{-4}.

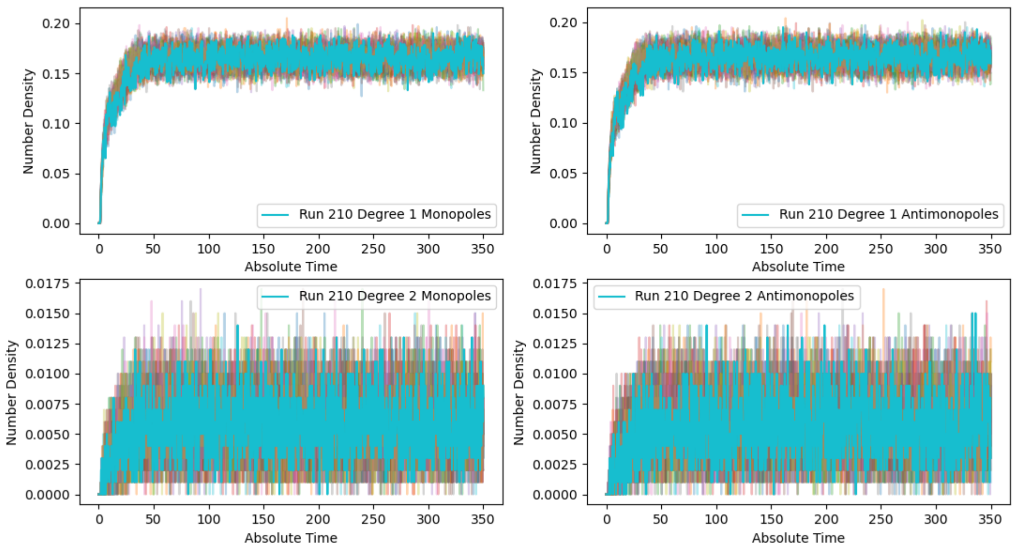

Figure 4.11: Monopole proportions in instances 181-210 for grid size 10x10x10 and damping 10^{-3}.

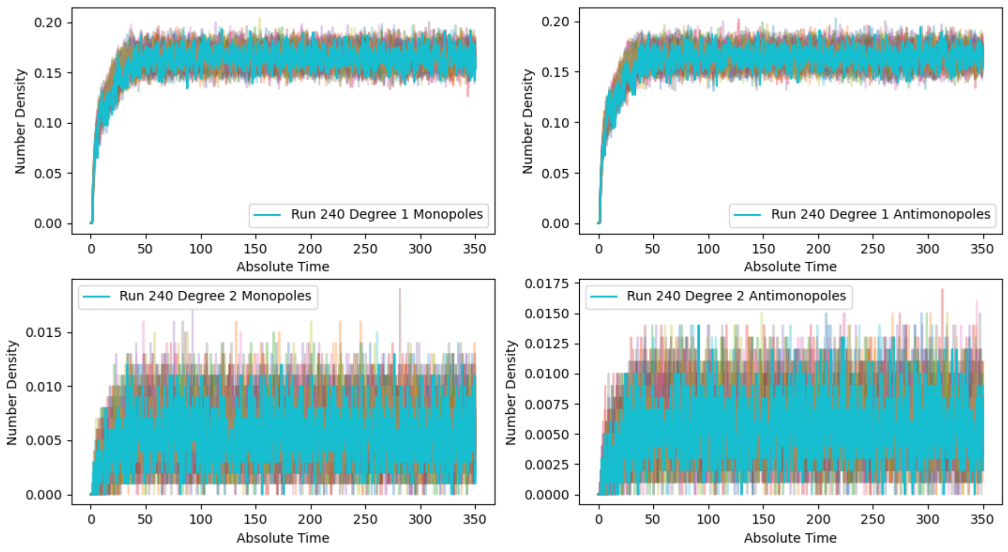

Figure 4.12: Monopole proportions in instances 211-240 for grid size 10x10x10 and damping 10^{-2}.

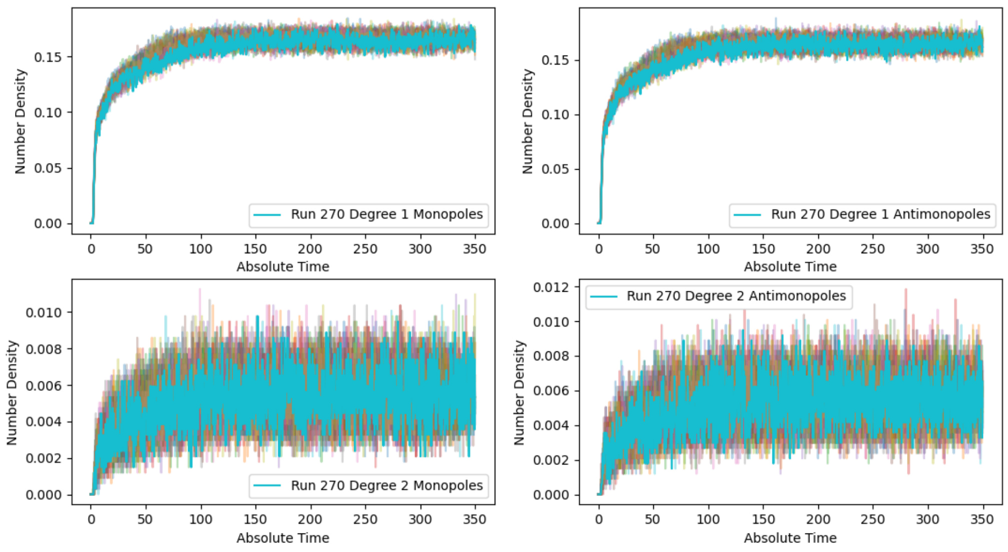

Figure 4.13: Monopole proportions in instances 241-270 for grid size 15x15x15 and damping 10^{-4}.

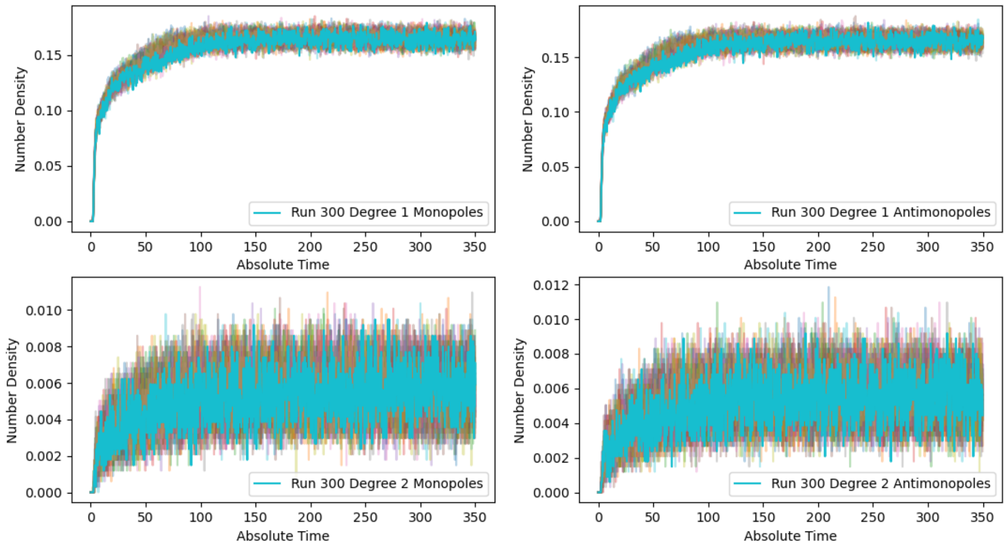

Figure 4.14: Monopole proportions in instances 271-300 for grid size 15x15x15 and damping 10^{-3}.

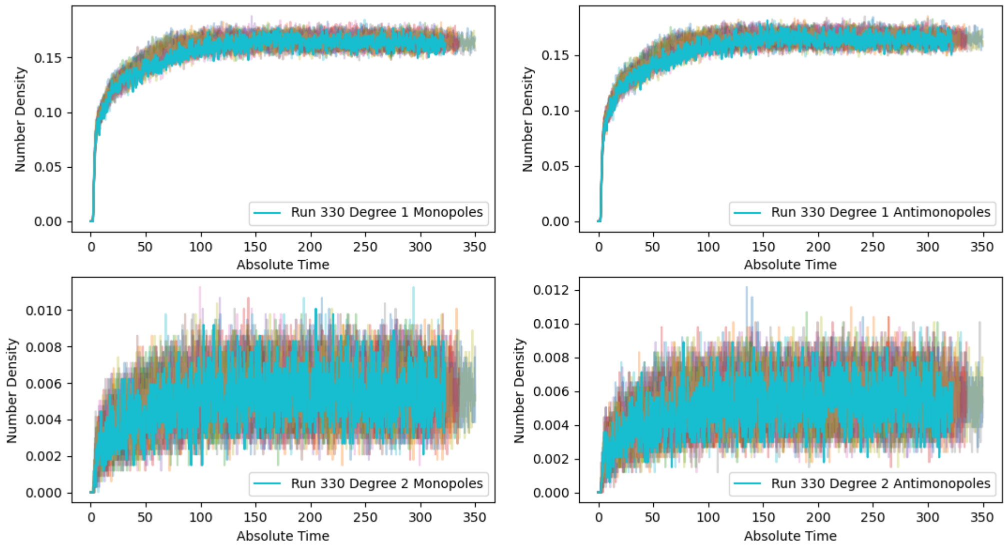

Figure 4.15: Monopole proportions in instances 301-330 for grid size 15x15x15 and damping 10^{-2}.

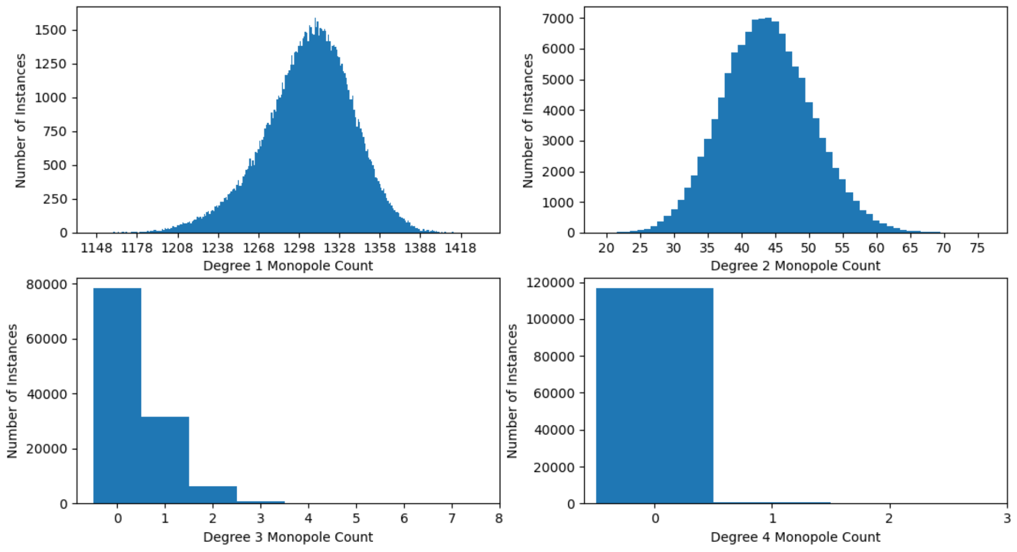

Figure 4.16: Monopole counts post-equilibration for instances 1-150 and grid size 20x20x20.

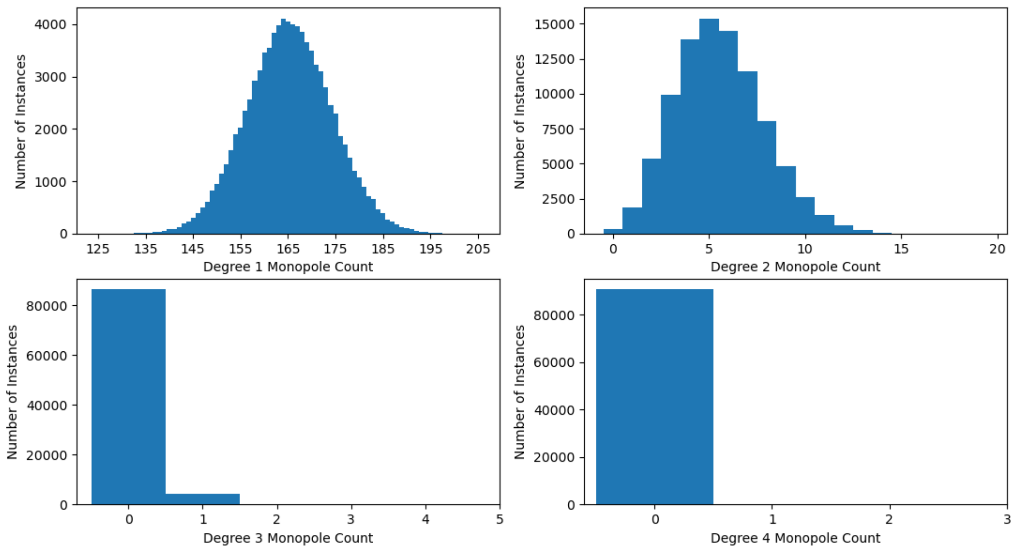

Figure 4.17: Monopole counts post-equilibration for instances 151-240 and grid size 10x10x10.

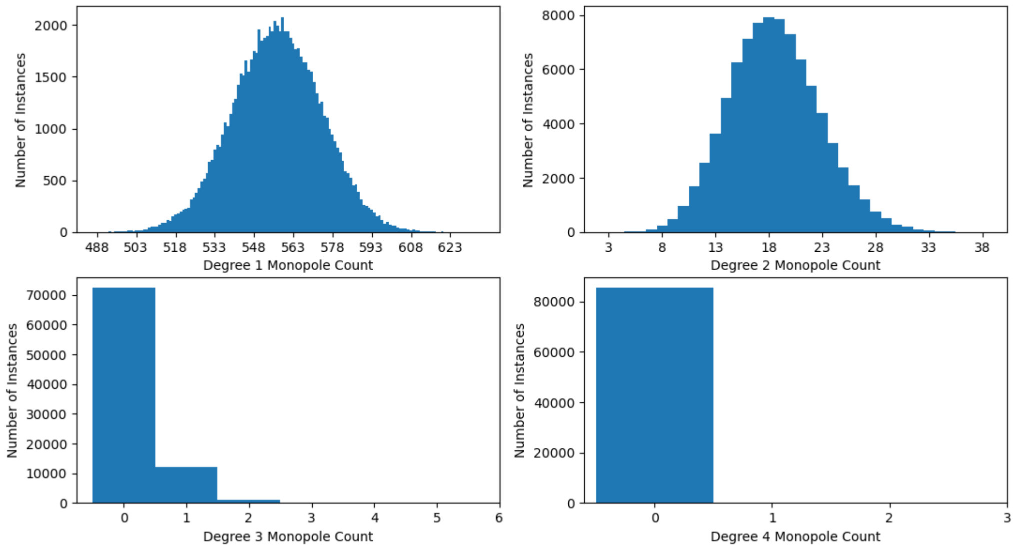

Figure 4.18: Monopole counts post-equilibration for instances 241-330 and grid size 15x15x15.

4.3.2 Differentiating Hull and Triangulation Monopole Detection Methods

In a private communication[5], Michael Seifert identified that there are objects which are found to be monopoles by the convex hull method, but which are, in fact, not monopoles. One such 'false monopole' is depicted in Figure 4.19. Note that the origin (red dot) is distinctly exterior to the manifold, but would be considered interior to a shape which does not account for the ordering of vertices.

4.3.3 Incidence Rate in Random Data

The faculty advisor for this work, Michael Seifert, identified an issue with the convex hull method for detecting monopoles, as briefly discussed in the previous section. In the process of identifying this issue, a large amount of randomized data was generated as test cases for monopoles which may have been misidentified. As an unintended consequence, we simultaneously arrived at an approximate incidence rate for monopoles in random data[5].

Figure 4.20 displays a histogram of the degrees of maps from 100,000 randomized sets of values of $\vec{P}$, as determined by the triangulation method (see Section 2.4.3). From this data, one can determine that approximately $34.2 \pm 0.2\%$ of randomized field values produce a monopole or anti-monopole (degree not equal to 0).

4.4 Spatial Cross-Correlations

We calculated the average spatial cross correlation across the entire grid for radii of 1 (nearest-neighbor), 2, 4, 6, and 8 grid cells for all 150 simulation instances of grid size 20x20x20 across all components of $\vec{P}$, $\vec{Q}$, and $\vec{\Pi}_Q$. In all cases, one instance has been highlighted as an example of typical behavior. We focus on instances 1-50 in this section, with instances 51-150 being similar in character and relegated to Appendix A.4.

[5]Private communication from Michael Seifert on Feb. 13, 2024.

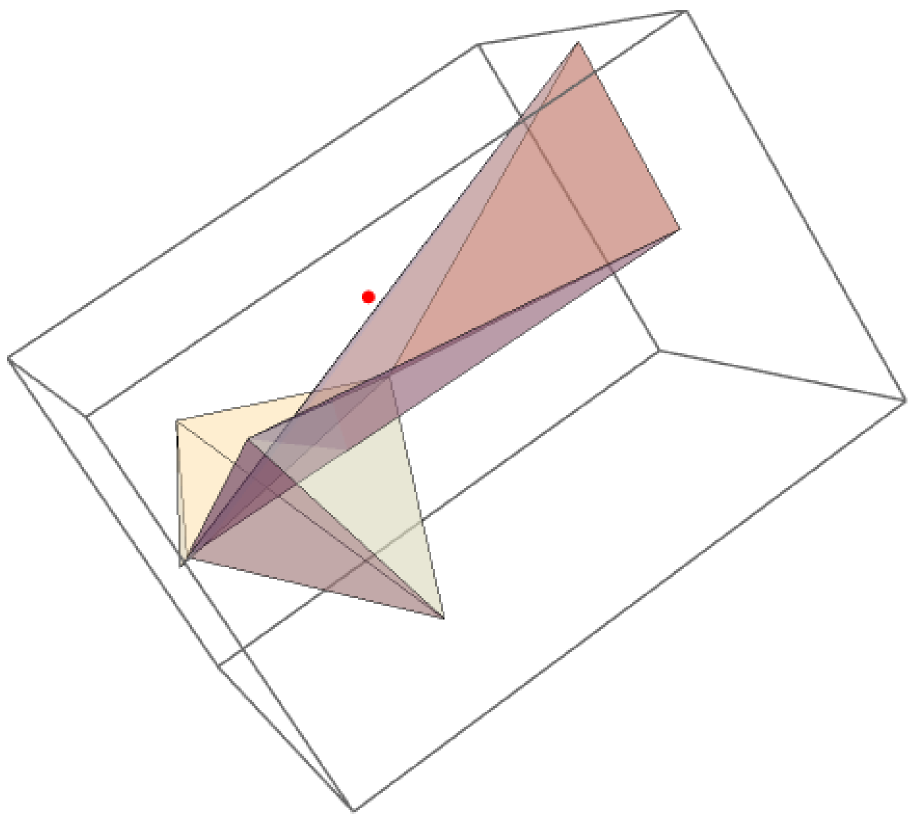

Figure 4.19: An example of a false monopole.

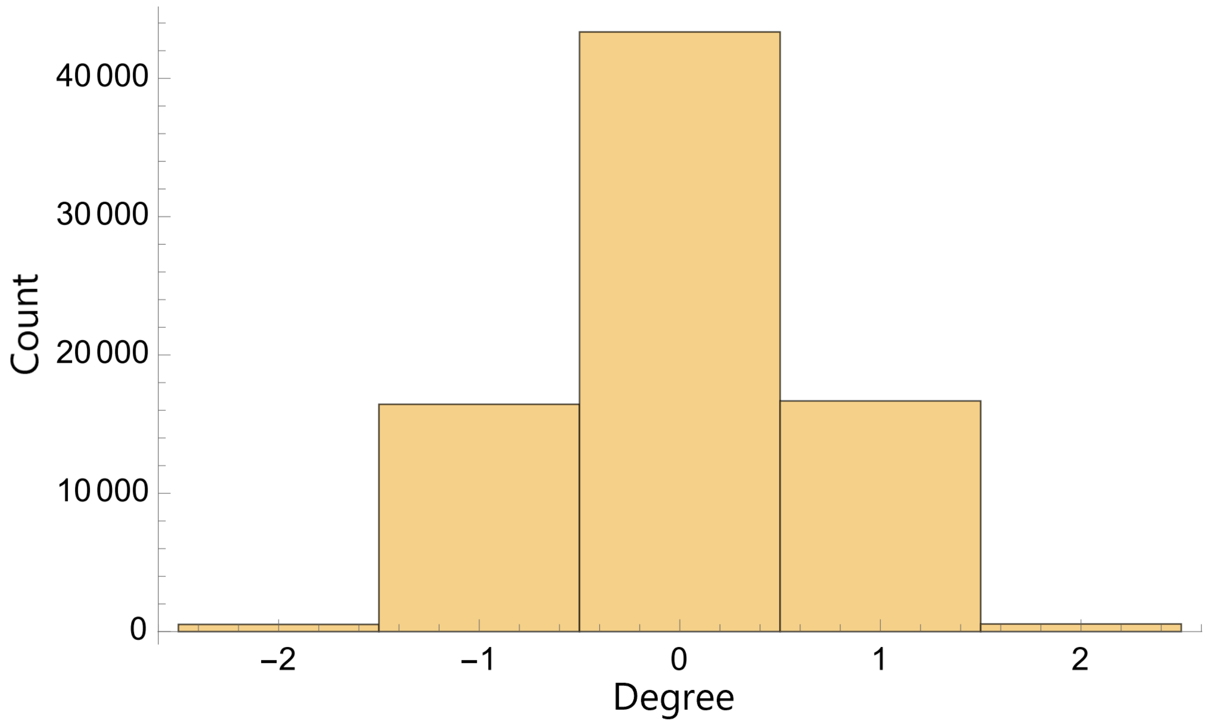

Figure 4.20: Degrees of 100,000 sets of 8 randomized $\vec{P}$ values.

4.4.1 Spatial Cross Correlation Variability in Time

Figures 4.21-29 show spatial cross-correlations across all studied radii and fields for instances 1-50 over time.

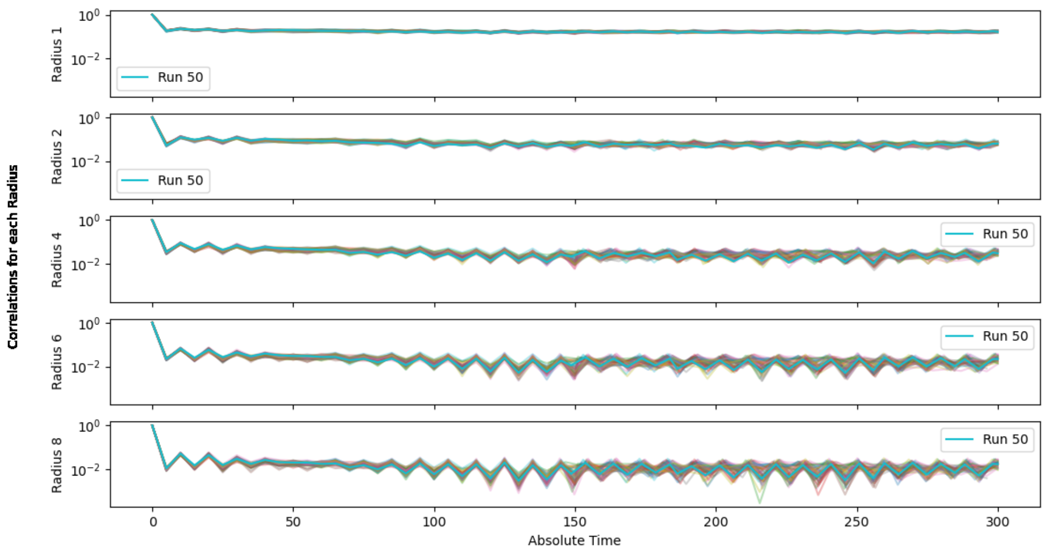

Figure 4.21: Spatial cross-correlations for $[\vec{\Pi}_Q]_x$ over time in instances 1-50.

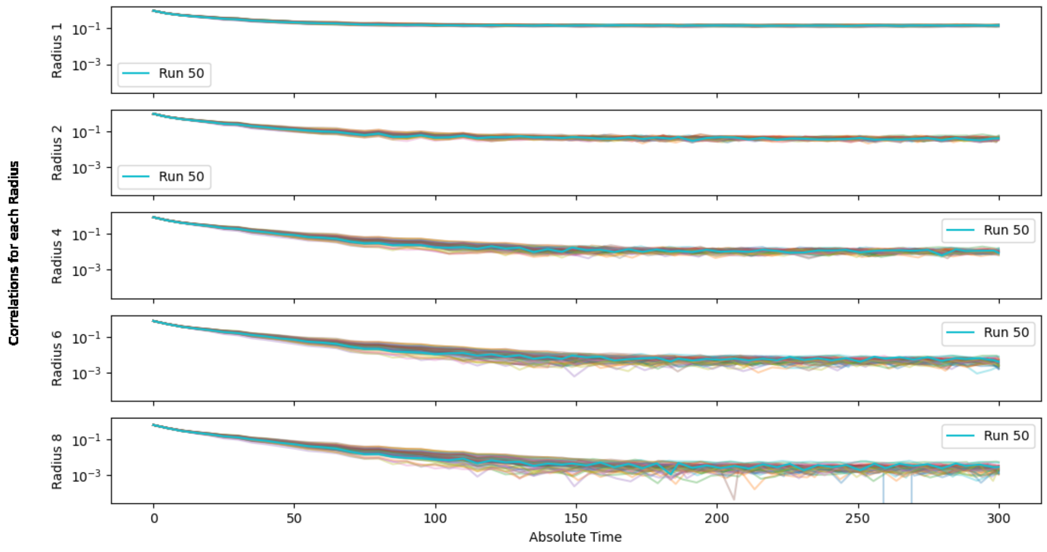

Figure 4.22: Spatial cross-correlations for $[\vec{\Pi}_Q]_y$ over time in instances 1-50.

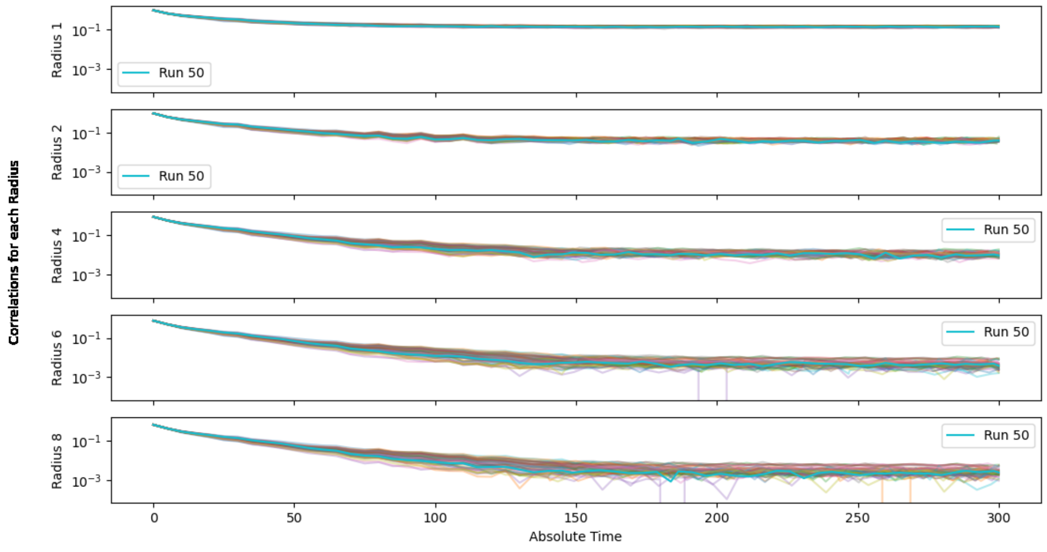

Figure 4.23: Spatial cross-correlations for $[\vec{\Pi}_Q]_z$ over time in instances 1-50.

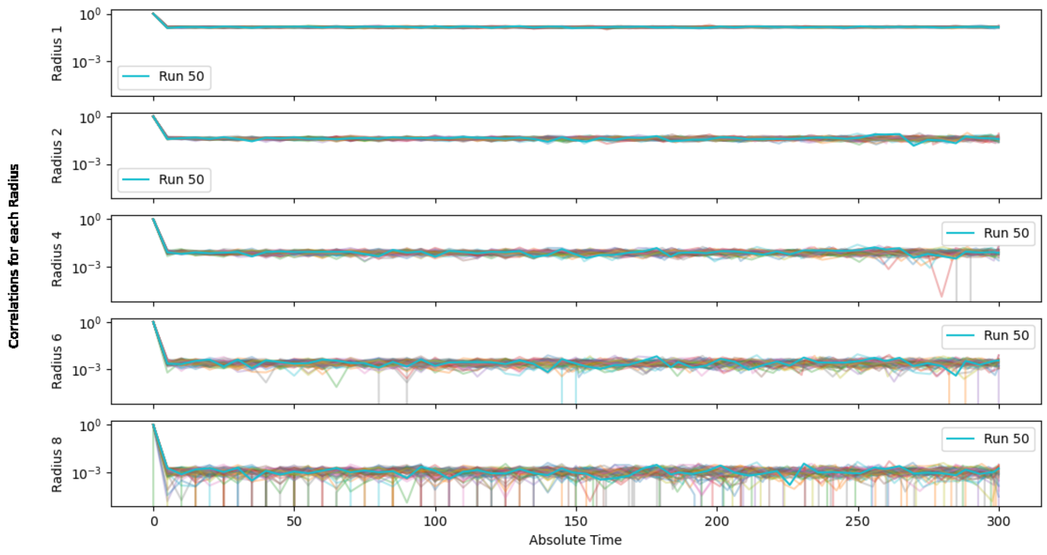

Figure 4.24: Spatial cross-correlations for $[\vec{P}]_x$ over time in instances 1-50.

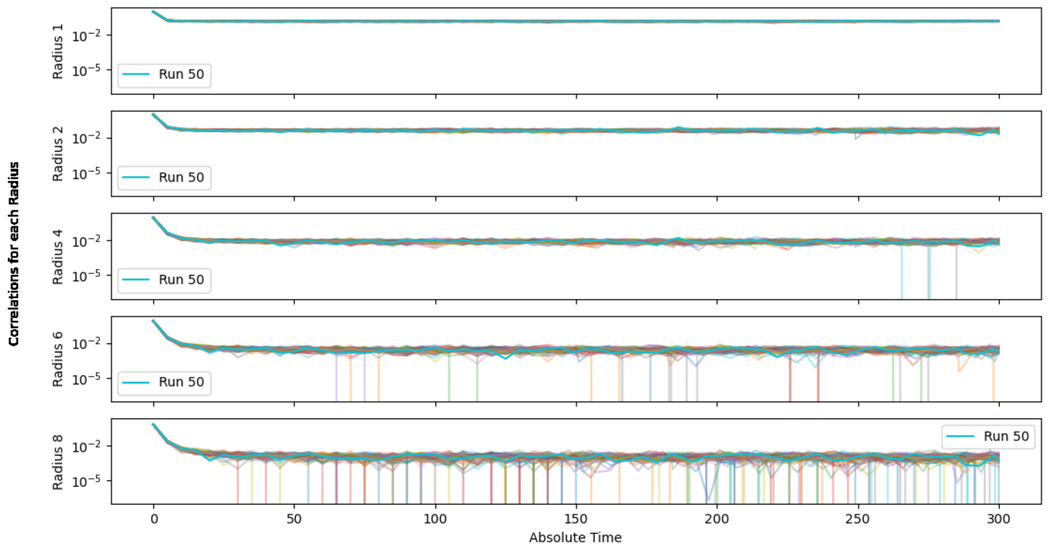

Figure 4.25: Spatial cross-correlations for $[\vec{P}]_y$ over time in instances 1-50.

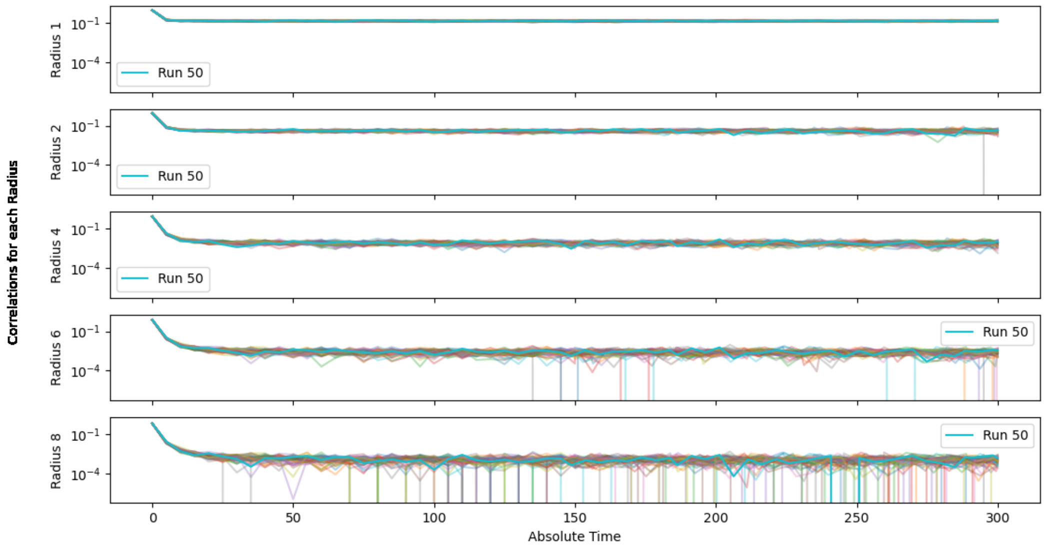

Figure 4.26: Spatial cross-correlations for $[\vec{P}]_z$ over time in instances 1-50.

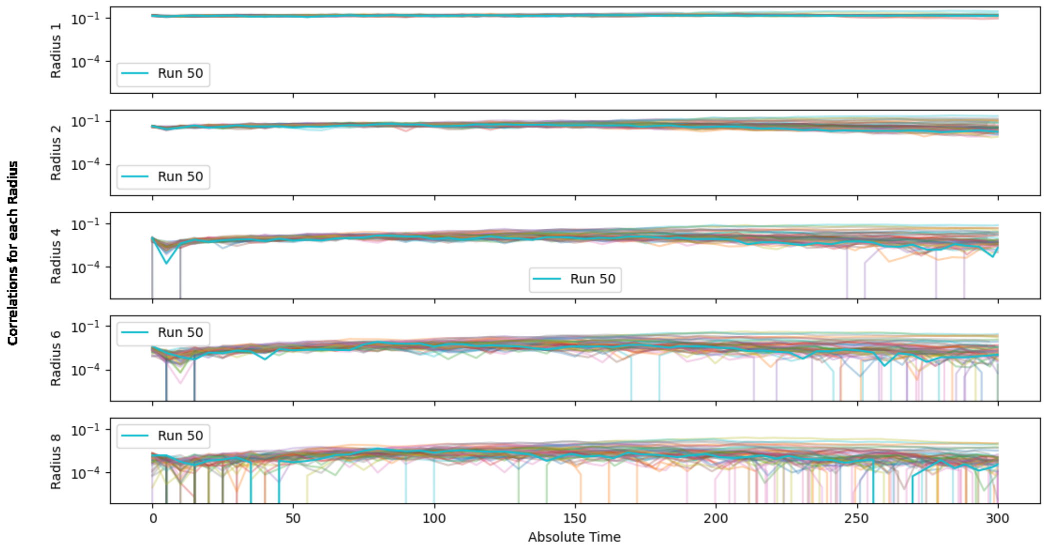

Figure 4.27: Spatial cross-correlations for $[\vec{Q}]_x$ over time in instances 1-50.

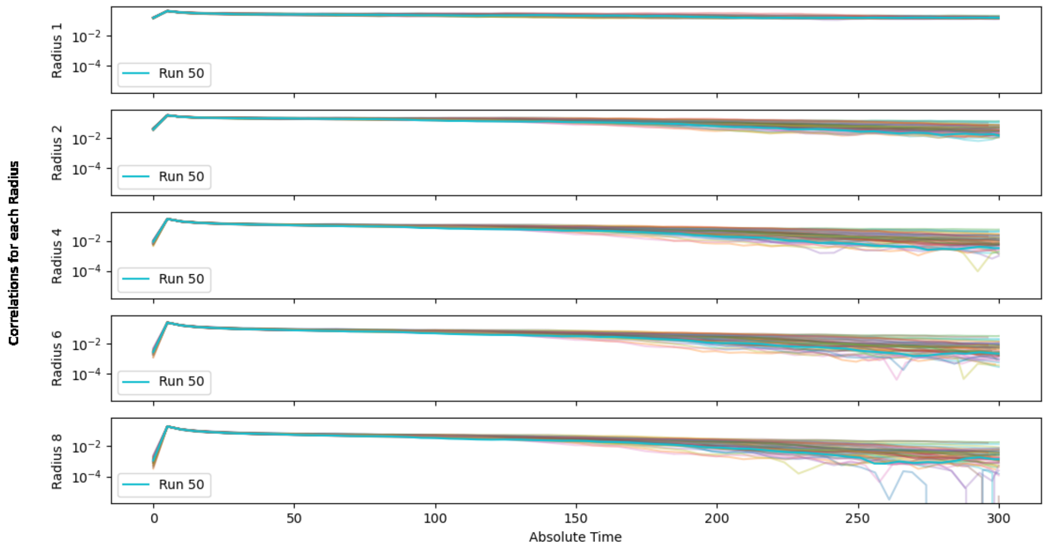

Figure 4.28: Spatial cross-correlations for $[\vec{Q}]_y$ over time in instances 1-50.

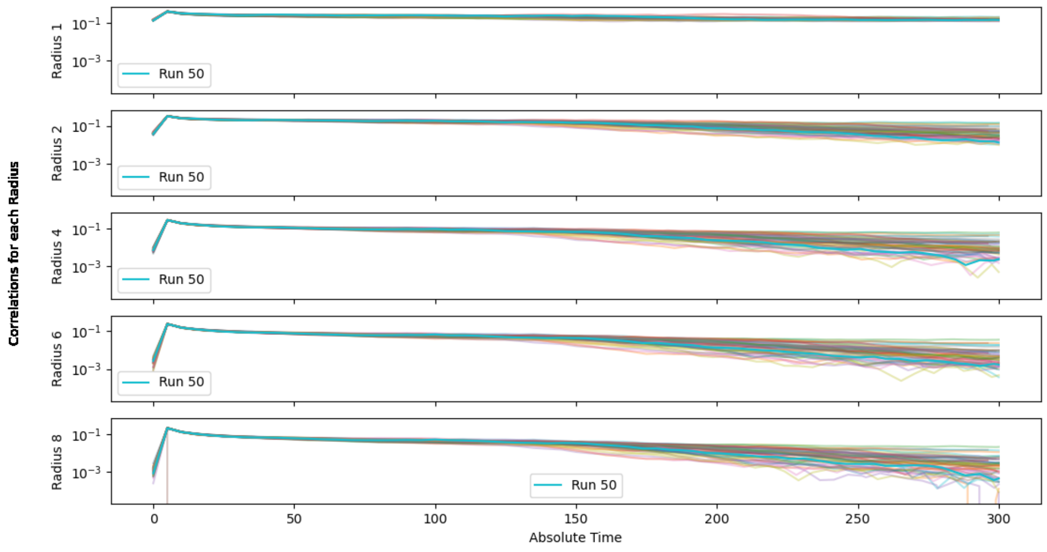

Figure 4.29: Spatial cross-correlations for $[\vec{Q}]_z$ over time in instances 1-50.

Discussion

5.1 Preservation of Conserved Quantities

As will be outlined in the following sections, the quantities which are meant to be conserved by our simulation are in fact being conserved to a satisfactory degree. This success suggests that our simulation has accurately reflected the mathematical constraints of the model, and that our simulation accurately reflects the behavior of the model.

5.1.1 System Energy

As is clear from Figures 4.1-3, average energy is conserved through the full equilibration time, and is damped after 100 iteration seconds, as intended. The conservation of the average energy directly implies the conservation of the total energy. Notably, the average energy decreases most rapidly shortly after damping is activated, and quickly settles into a negative minimum. It is unclear at this moment what the cause of this minimum value is, since in theory the damping term should always be negative definite, and the energy strictly decreasing. It is possible, however, that the damping term has evolved to become extraordinarily small, which could only be the case if $\vec{\Pi}_Q \approx -\vec{\nabla} \times \vec{P}$. It is also unclear at this moment why this minimum is so 'unstable', in the sense that the energy rapidly oscillates. We believe that this is not a physical oscillation, but rather a numerical instability. Without an alternative solving algorithm (ideally a symplectic integrator), we are unable to verify whether or not this is the case.

5.1.2 Primary and Secondary Constraints

In all 330 instances of the simulation, neither the average values of Ψ nor any component of $\vec{\Psi}$ exceeded a value of 10^{-5} (c.f. Figures 4.1-3). It would be preferable if these quantities remained precisely at 0, as they are defined in the system. However, a maximum deviation on the order of 10^{-5}, with a more typical deviation on the order of 10^{-7}, is reasonable.

5.1.3 Topological Order

Topological order is perfectly preserved through the full simulation duration in all cases (c.f. Figures 4.1-3), which supports the idea that our monopole detection algorithm is successful. While it is not represented in any of the provided figures, the convex hull method as implemented typically violates the conservation of topological order, and indeed was observed to deviate on the order of 10^3 at times. This fact was the primary motivation to develop the triangulation method as an alternative.

5.2 Long-Term Field Behavior

In this section, we find that the field values for our simulation appear to be bounded, though only under large enough damping. Bounded values confirm that our simulation is viable over long time scales, since it would be impossible to compute with numbers that grow endlessly. In addition, the limited precision of a computer could lead to cumulative rounding errors, eventually violating our conservation laws.

5.2.1 $\vec{P}$ and $\vec{Q}$

Figures 4.4-6 show that $\vec{P}$ and $\vec{Q}$ exhibit exactly parallel trends in their average magnitude, as one might have expected from the definition of the invariant X. It is also apparent from these figures, in combination with supplemental data from Section A.3, that the damping rate impacts the average magnitudes of $\vec{P}$ and $\vec{Q}$. To some extent this is desirable, since it is evident that these fields consistently increase approximately linearly for small damping coefficients, which may be problematic for the viability of simulating the field and for physical realism. Notably, these fields also appear to grow linearly in time when no damping is applied. A large enough value for the damping rate (a value of 10^{-2} seems sufficient) appears to alleviate this problem, with eventual bounds on the average magnitude.

5.2.2 $\vec{\Pi}_Q$

In all cases, the average magnitude of $\vec{\Pi}_Q$ appears to be decreasing. Without damping, this value appears to converge to some small, positive quantity. Once damping is activated, the average magnitude of this field appears to remain relatively bounded. Small damping rates seemingly impart little to no change in the magnitude of $\vec{\Pi}_Q$, while larger damping rates are seemingly bounded below by some negative value.

5.2.3 λ

In all cases, we observe that the average magnitude of λ quickly tends towards some small value. Our equations were derived under the assumption that $\lambda \neq 0$, so we would expect that λ does not tend to exactly 0. Indeed, further investigation shows that the magnitude of λ seems to become stable at a value on the order of 10^{-5}, which is distinctly nonzero. Our assumption that $\lambda \neq 0$ appears to hold, possibly by construction.

5.3 Monopole Behavior

The conservation of topological order also guarantees a somewhat desirable property, which we will refer to as the principle that monopoles and anti-monopoles are created and annihilated in pairs. To be more specific, if ever a degree 1 monopole forms in the grid, or the degree of a monopole increases by 1, then one of the following must have happened: the degree of a monopole decreased by 1, the degree of an anti-monopole increased by 1, or a degree 1 anti-monopole formed. The equivalent case for anti-monopoles must also hold. The symmetry of this argument suggests that monopoles and anti-monopoles should occur in approximately equal proportions. Figures 4.7-15 show that degree 1 and 2 monopoles and anti-monopoles are in fact approximately equally likely. Degree 3 monopoles are relatively unlikely, as evidenced by Figures 16-18, but we have confirmed that degree 3 anti-monopoles are also approximately equally likely as degree 3 monopoles.

Since we have established that monopoles and anti-monopoles are equally likely, it is reasonable to assume that the topological defects in the grid are half monopoles and half anti-monopoles, with some small error. This implies that of the $34.2 \pm 0.2\%$ of randomized grid cells which contain a topological defect (see Figure 4.20), about $17.1 \pm 0.1\%$ of them are monopoles of some kind. The symmetry of Figure 4.20 further supports this idea. We found that $16.990 \pm 0.001\%$ of grid cells in our simulation were monopoles, which is in near agreement with the result for randomized data, but not exact agreement. This measurement suggests that the field values in our simulation appear relatively – but not completely – random on large scales.

We also note that degree 1 and 2 monopoles and anti-monopoles converge to approximately the same number densities, regardless of damping rate or grid size. Our results are limited to some extent, since our finite grid size prevents us from making statements about monopole densities below some finite scale; however, it is reasonable to conclude from the available results that monopoles would not be sparse in the Universe, and in fact would be created readily. This fact casts some doubt on the viability of the model, since we are not aware of any detection of such a monopole or any other unexplained phenomenon in great enough abundance and with the observational properties (c.f. [12]) necessary to account for such an abundance.

5.4 Spatial Cross-Correlations

5.4.1 Spatial Cross Correlation Variability in Time

Figures 4.21-29 show the spatial cross-correlations of instances 1-50 over time for all 9 vector field components. In both the $\vec{P}$ and $\vec{\Pi}_Q$ fields, the correlations remain fairly constant after about 10 and 100 iteration seconds, respectively. After damping is activated, the $\vec{Q}$ field begins to become less correlated at each radius, for reasons which are not immediately obvious.

5.4.2 Spatial Cross-Correlations Over Radii

Motivated by the study of correlation lengths in Monte Carlo simulations and statistical mechanics, we attempted to determine the correlation lengths of each of the 9 vector field components. Let $\bar{C}(r)$ be the average correlation of a point to other points within a radius r. Many systems obey some variant of the relationship

$$\bar{C}(r) = A\mathrm{e}^{-r/\xi}(r/\xi)^{-(d-2+\eta)}, \tag{5.4.1}$$

where A is a constant of proportionality, ξ is the correlation length, d is the number of dimensions for the system, and η is a critical exponent [30]. In this context, we will not be treating η as a true critical exponent, since we are not working with a system near a critical point; however, this form is a useful ansatz for characterizing the behavior of our system.

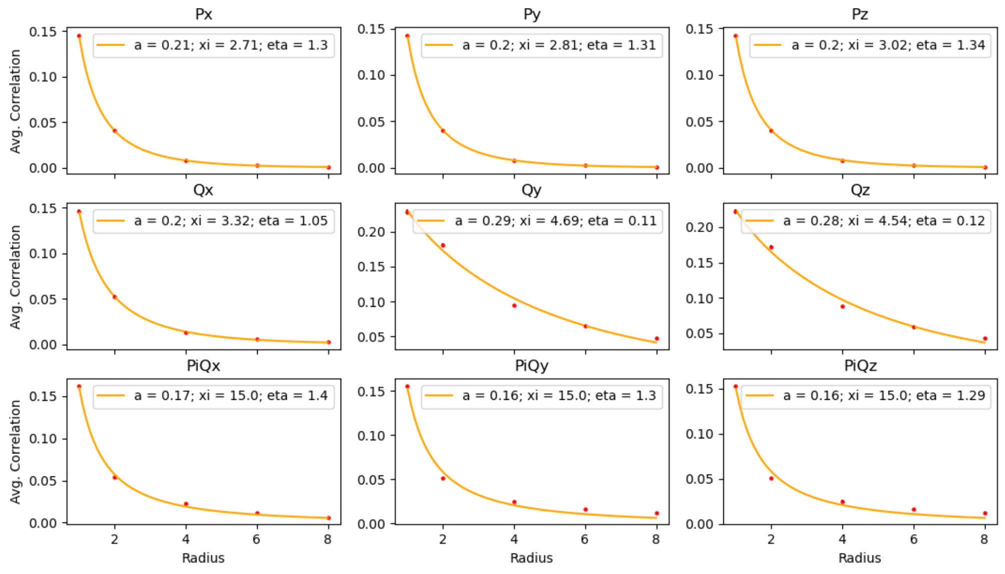

Figure 5.1: Spatial cross-correlations over distances for all 9 components of vectors fields at time 100 in instances 1-50.

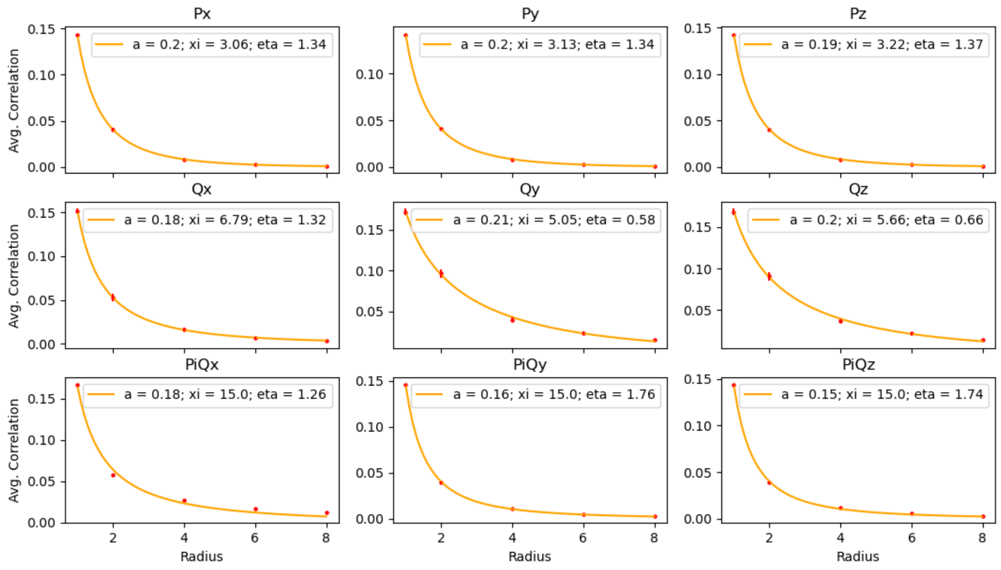

Figure 5.2: Spatial cross-correlations over distances for all 9 components of vectors fields at time 200 in instances 1-50.

Figures 5.1-6 show spatial cross-correlations as a function of radius in instances 1-50 and for specified times. The points presented are averages across all 50 instances, with error bars determined by the standard error and typically smaller than the points themselves. Due to the limitations of using an adaptive step size, the time for any individual instance may deviate slightly from the intended value, i.e. a given instance used in the calculation of the average may have been measured at time 99.98 instead of time 100.

A best fit to Equation 5.4.1 (shown in orange) was found for each field at 100 iteration seconds and 200 simulation seconds. Each plot also displays the best fit values for the parameters A, η, and ξ. The inconsistent determination of η for $\vec{Q}$ and $\vec{\Pi}_Q$ may be an indication that the assumption that this function models this system is incorrect, since it should have a similar value throughout. However, η is determined with some regularity for $\vec{P}$, which is also conveniently the field which determines the behavior of monopoles in our model. If the fit results are to be believed, the correlation length for $\vec{P}$ in this system is around 3 grid cells in a 20x20x20 grid. We see similar results for grids of size 10x10x10 and 15x15x15, as depicted in Figures 5.1-6. The correlation length is the typical distance over which field values are related to each other, and can be thought of as the typical radius for regions of organized behavior. For a sense of scale, approximately 300 such regions would fit into a 20x20x20 grid. This result is consistent with some short scale correlation, but a lack of organized behavior over large scales. We might have been able to guess this result from the result of Section 5.3.

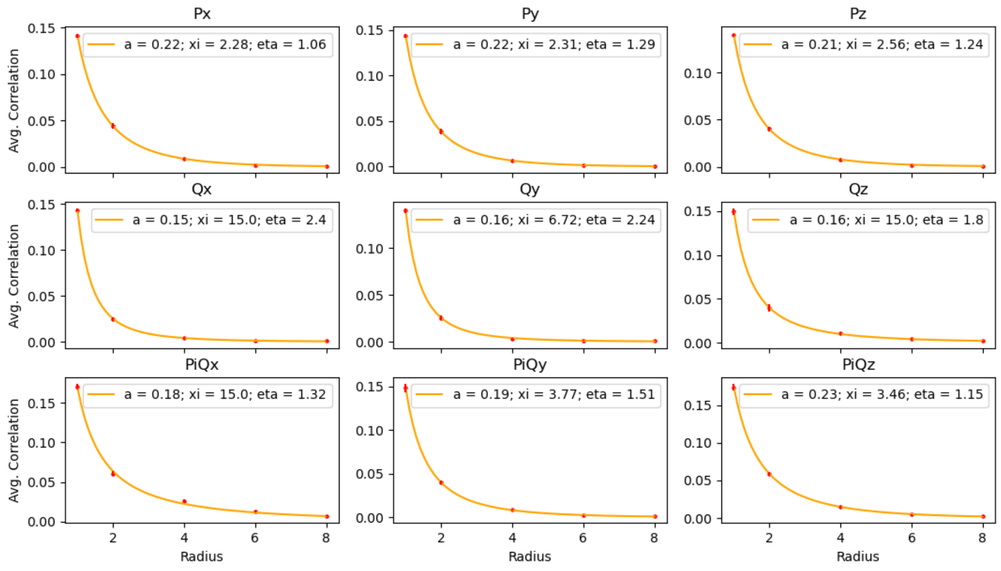

Figure 5.3: Spatial cross-correlations over distances for all 9 components of vectors fields at time 100 in instances 151-180.

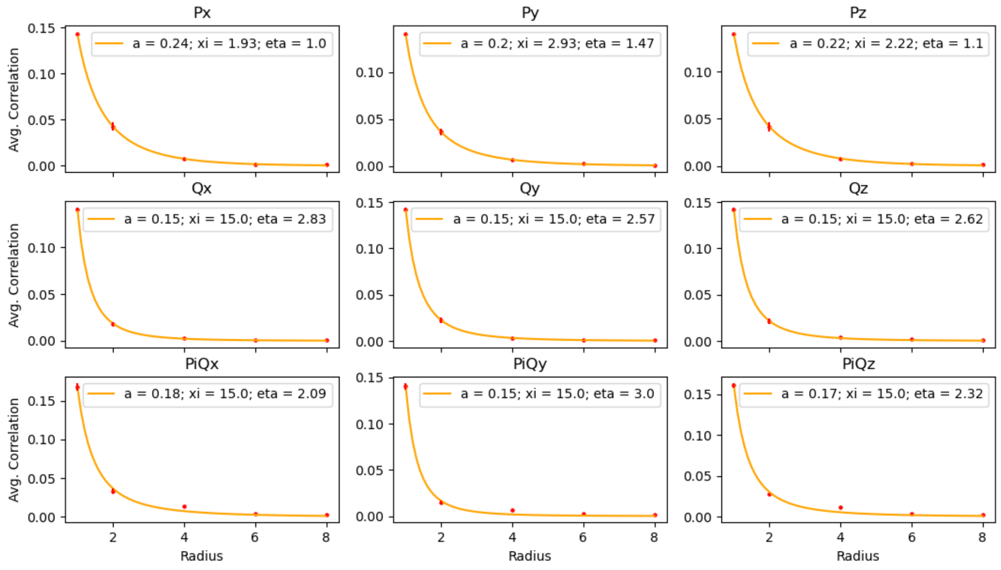

Figure 5.4: Spatial cross-correlations over distances for all 9 components of vectors fields at time 200 in instances 151-180.

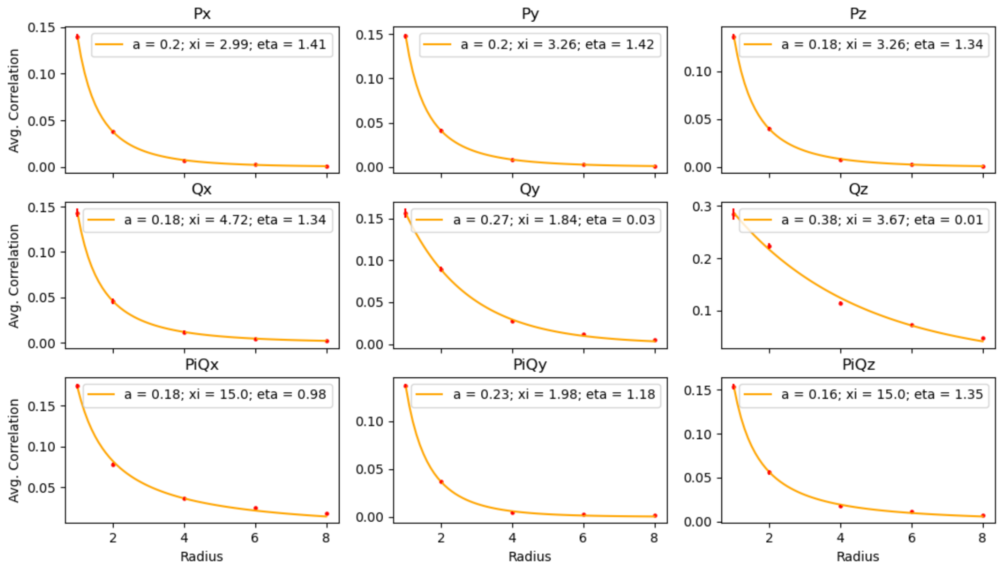

Figure 5.5: Spatial cross-correlations over distances for all 9 components of vectors fields at time 100 in instances 301-330.

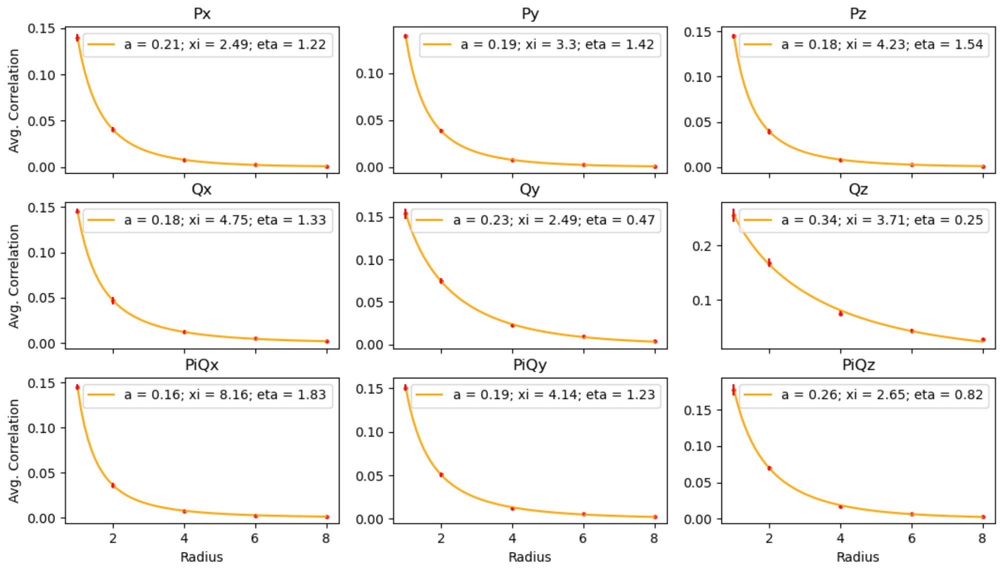

Figure 5.6: Spatial cross-correlations over distances for all 9 components of vectors fields at time 200 in instances 301-330.

5.5 Differentiating Hull and Triangulation Monopole Detection Methods

As was mentioned in Section 5.1.3, the triangulation method evidently conserves topological order, while the convex hull method was observed to violate this conservation regularly, at least as far as we could tell. It is possible that our inability to determine the degree of a monopole was the cause of the discrepancy, but as is evident from Figures 4.16-18, degree 1 monopoles are by far the most common, so this discrepancy should be relatively small – smaller than what we observed. We believe the following explanation to be more likely. Figure 4.19 gives an example of a false monopole which we believe to be the cause of this flaw. These false monopoles, which would contain the origin if the order of the vertices did not matter, are counted by the convex hull method. The triangulation method disregards false monopoles by accounting for the ordering of the vertices, and therefore only counts 'true' monopoles. Regardless of which explanation is the dominant effect, the triangulation method simultaneously resolves both potential issues by accounting for varying degrees of monopole and by disregarding false monopoles.

Conclusion

6.1 Overview

In this work, we presented the derivation of time evolution equations for a rank-2, anti-symmetric tensor field capable of violating Lorentz symmetry by assuming a nonzero vacuum expectation value. These equations were used to develop a numerical simulation capable of tracking the properties of the field, including the incidence rate of monopole topological defects also defined herein. Through 330 test instances of this simulation, we determined that the dynamics of our system were accurately represented, and considered our simulation to be successful. We also analyzed the correlation between field values at adjacent points and the incidence rates of monopoles, concluding that field values are primarily correlated over short scales and monopole defects would be readily produced by this field and be relatively abundant. The combination of these factors suggests that this field is not physically viable, though more detailed studies are necessary to draw a definitive conclusion. In addition, we found that the field values may be divergent when the energy of the system is not damped above an as of yet undetermined level, which may also limit the physical viability of this model.

6.2 Future Work

Our simulation is limited in a few key respects which should be addressed in a future study. For one, it would ideal for a future study to replicate our results using a symplectic integrator for numerical differential equation solving, once one is available in SciPy. In addition, our method of removing energy from the system is entirely *ad hoc*, and should be replaced with a physically justified damping method. The obvious solution would be to include the expansion of the Universe, which one would expect to damp the system energy in the same way that electromagnetic radiation is redshifted to longer wavelengths. The expansion of the Universe should be included in our simulation regardless of its impact on the system energy. We also have yet to consider the time evolution equations generated by the inclusion of the second invariant as presented in Appendix A.2. The inclusion of the second invariant represents a further constraint on the system, which would in turn change the structure of field-space in a way which could impact monopole counts. Finally, we did not consider the impact of changing the value of the constant b. By taking $b = -1$ throughout this work, we considered only a 'timelike' version of the model. The impact of choosing a positive ('spacelike') b and varying the magnitude of b should be investigated.

References

[1] Albert Einstein. “On the Electrodynamics of Moving Bodies”. In: *Annalen der Physik* (1905). URL: users.physics.ox.ac.uk/~rtaylor/teaching/specrel.pdf.

[2] V. Alan Kostelecký and Stuart Samuel. “Gravitational phenomenology in higher-dimensional theories and strings”. In: *Phys. Rev. D* 40.6 (Sept. 1989). DOI: 10.1103/PhysRevD.40.1886.

[3] V. Alan Kostelecký. “Gravity, Lorentz violation, and the standard model”. In: *Phys. Rev. D* 69.10 (May 2004). DOI: 10.1103/PhysRevD.69.105009.

[4] Robert Bluhm and V. Alan Kostelecký. “Spontaneous Lorentz violation, Nambu-Goldstone modes, and gravity”. In: *Phys. Rev. D* 71.6 (Mar. 2005). DOI: 10.1103/PhysRevD.71.065008.

[5] Bailey Altschul and Kostelecký. “Lorentz violations with an antisymmetric tensor”. In: *Phys. Rev. D* 81.6 (Mar. 2010). DOI: 10.1103/PhysRevD.81.065028.

[6] Peter Higgs. “Spontaneous Symmetry Breakdown without Massless Bosons”. In: *Physical Review* 145.4 (May 1966). DOI: 10.1103/PhysRev.145.1156.

[7] L. D. Landau. “On the Theory of Superconductivity”. In: *Collected Papers of L.D. Landau*. Pergamon, 1965. ISBN: 978-0-08-010586-4. URL: www.sciencedirect.com/science/article/pii/B9780080105864500353.

[8] Michael D. Seifert. “Monopole Solution in a Lorentz-Violating Field Theory”. In: *Phys. Rev. Lett.* 105.20 (Nov. 2010). DOI: 10.1103/PhysRevLett.105.201601.

[9] Michael D. Seifert. “Dynamical Lorentz symmetry breaking and topological defects”. In: *Phys. Rev. D* 82.12 (Dec. 2010). DOI: 10.1103/PhysRevD.82.125015.

[10] T. W. B. Kibble. “Topology of cosmic domains and strings”. In: *J. of Phys. A: Math. Gen.* (1976). DOI: 10.1088/0305-4470/9/8/029.

[11] T. W. B. Kibble. “Some implications of a cosmological phase transition”. In: *Physics Reports* 67.1 (1980). DOI: 10.1016/0370-1573(80)90091-5.

[12] Kamuela N. Lau and Michael D. Seifert. “Direct-coupling lensing by antisymmetric tensor monopoles”. In: *Phys. Rev. D* 95.2 (Jan. 2017). Publisher: American Physical Society. DOI: 10.1103/PhysRevD.95.025023.

[13] Alan H. Guth. “Inflationary universe: A possible solution to the horizon and flatness problems”. In: *Phys. Rev. D* 23.2 (Jan. 1981). Publisher: American Physical Society. DOI: 10.1103/PhysRevD.23.347.

[14] R. Durrer, M. Kunz, and A. Melchiorri. “Cosmic structure formation with topological defects”. In: *Physics Reports* 364.1 (2002). DOI: 10.1016/S0370-1573(02)00014-5.

[15] David P. Bennett and Sun Hong Rhie. “Cosmological evolution of global monopoles and the origin of large-scale structure”. In: *Phys. Rev. Lett.* 65.14 (Oct. 1990). DOI: 10.1103/PhysRevLett.65.1709.

[16] George Sarkar. “Simulating Interactions of Lorentz-Violating Monopoles”. In: *Physics, Astronomy and Geophysics Honors Papers* 8 (2017). URL: digitalcommons.conncoll.edu/physicshp/8.

[17] Michael D. Seifert. “Constraints and degrees of freedom in Lorentz-violating field theories”. In: *Phys. Rev. D* 99.4 (Feb. 2019). DOI: 10.1103/PhysRevD.99.045003.

[18] V.I. Arnold. *Mathematical Methods of Classical Mechanics.* Springer, 1989. Chap. 4. ISBN: 0 387 96890 3.

[19] Michael D. Seifert. “Singular Hamiltonians in models with spontaneous Lorentz symmetry breaking”. In: *Phys. Rev. D* (Sept. 2019). DOI: `10.1103/PhysRevD.100.065017`.

[20] Paul A. M. Dirac. *Lectures on Quantum Mechanics.* Belfer Graduate School of Science Monographs 2. New York: Belfer Graduate School of Science, Yeshiva University, 1964.

[21] Jerry B. Marion and Stephen T. Thornton. *Classical Dynamics of Particles and Systems.* Harcourt Brace & Company, 1995. ISBN: 0 03 097302 3.

[22] M. A. Armstrong. *Basic Topology.* Springer-Verlag New York, 1983, pp. 195–7. ISBN: 0 387 90839 0.

[23] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020). DOI: `10.1038/s41592-019-0686-2`.

[24] C. B. Barber, D. P. Dobkin, and H. T. Huhdanpaa. “The Quickhull algorithm for convex hulls”. In: *ACM Trans. on Mathematical Software* (Dec. 1996). URL: `http://www.qhull.org`.

[25] N. G. Lloyd. *Degree theory.* Cambridge University Press, 1978. ISBN: 0 521 21614 1.

[26] P.R. Bevington and D.K. Robinson. *Data Reduction and Error Analysis in the Physical Sciences.* McGraw Hill, 2003. ISBN: 0 07 247227 8.

[27] David Zwicker. “py-pde: A Python package for solving partial differential equations”. In: *Journal of Open Source Software* (2020). DOI: `10.21105/joss.02158`.

[28] James Stewart. *Multivariable Calculus, 8th Edition.* Cengage Learning, 2016. ISBN: 978 1 305 26664 3.

[29] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (2020). DOI: `10.1038/s41586-020-2649-2`.

[30] Gould and Tobochnik. *Statistical and Thermal Physics With Computer Applications.* Princeton University Press, 2010. ISBN: 978 0 691 13744 5.

[31] David Griffiths. *Introduction to Electrodynamics.* 3. ed. Prentice Hall, 1999. ISBN: 0-13-805326-X.

Appendices

A.1 Derivatives With Respect To Vectors and Identities

For the purposes of this work, we will define the derivative of a function f of a vector field $\vec{A}$ with respect to $\vec{A}$ to be approximately linear under integration, i.e.

$$\int f(\vec{A}+\delta\vec{A})\,\mathrm{d}^4x \approx \int f(\vec{A}) + \frac{\partial f(\vec{A})}{\partial\vec{A}}\delta\vec{A}\,\mathrm{d}^4x \tag{A.1.1}$$

for 'small' values of the variation $\delta\vec{A}$. We will omit the integration for the sake of clarity, as is typical when dealing with Hamiltonian systems. This definition is equivalent to a more familiar limit definition of the derivative,

$$\frac{\partial f(\vec{A})}{\partial\vec{A}} \equiv \lim_{\delta\vec{A}\to 0}\frac{f(\vec{A}+\delta\vec{A})-f(\vec{A})}{\delta\vec{A}} \tag{A.1.2}$$

By simple rearrangement and the application of a limit to handle division by the 'small' quantity $\delta\vec{A}$. It is trivial to see that the standard properties of the derivative all follow from this definition.

The following four identities, Equations (A.1.3-6), prove to be useful in deriving the time evolution equations of the Hamiltonian for the studied system. Let $\vec{A}$ and $\vec{B}$ be vector fields, and f a scalar field. Then,

$$\frac{\partial}{\partial\vec{A}}\left[\vec{B}\cdot(\vec{\nabla}\times\vec{A})\right] = \vec{\nabla}\times\vec{B} \tag{A.1.3}$$

$$\frac{\partial}{\partial\vec{A}}\left[(\vec{\nabla}\cdot\vec{A})^2\right] = -2\vec{\nabla}(\vec{\nabla}\cdot\vec{A}) \tag{A.1.4}$$

$$\frac{\partial}{\partial\vec{A}}\left[\vec{B}\cdot\left(\vec{\nabla}\times(f\vec{A})\right)\right] = f(\vec{\nabla}\times\vec{B}) - (\vec{\nabla}f\times\vec{B}) \tag{A.1.5}$$

$$\frac{\partial}{\partial f}\left[\vec{\nabla}\times(f\vec{A})\right] = \vec{\nabla}\times\vec{A}. \tag{A.1.6}$$

The proofs of these identities require a handful of additional vector identities, which can be found in [31] and have been replicated below. Let $\vec{A}$, $\vec{B}$, and $\vec{C}$ be vectors and f a scalar. Then,

$$\vec{A}\cdot(\vec{B}\times\vec{C}) = \vec{B}\cdot(\vec{C}\times\vec{A}) = \vec{C}\cdot(\vec{A}\times\vec{B}) \tag{A.1.7}$$

$$\vec{\nabla}\cdot(f\vec{A}) = f(\vec{\nabla}\cdot\vec{A}) + \vec{A}\cdot(\vec{\nabla}f) \tag{A.1.8}$$

$$\vec{\nabla}\cdot(\vec{A}\times\vec{B}) = \vec{B}\cdot(\vec{\nabla}\times\vec{A}) - \vec{A}\cdot(\vec{\nabla}\times\vec{B}) \tag{A.1.9}$$

$$\vec{\nabla}\times(f\vec{A}) = f(\vec{\nabla}\times\vec{A}) - \vec{A}\times(\vec{\nabla}f). \tag{A.1.10}$$

In addition, for the purposes of this work we must think of the Hamiltonian as being integrated over the entirety of space, so terms which contain only the divergence of some vector field will automatically vanish through integration by parts and the application of Stokes' Theorem around the boundary [28]. This detail will prove to be necessary to prove the identities in their presented form.

The proofs of Equations (A.1.3-6) are as follows, and are all similar in character. We will find the notation of a variation in a vector field, δ, preferable to standard derivative notation for many of the intermediate steps.

Proof. Let $\vec{A}$ and $\vec{B}$ be independent vector fields, and consider the expression

$$\frac{\partial}{\partial \vec{A}}\left[\vec{B}\cdot(\vec{\nabla}\times\vec{A})\right]$$

In the language of variations, we can instead write

$$\frac{\partial}{\partial \vec{A}}\left[\vec{B}\cdot(\vec{\nabla}\times\vec{A})\right] = \vec{B}\cdot(\vec{\nabla}\times\delta\vec{A})$$

This equality follows from Clairaut's theorem [28], which allows for the free exchange of differential operators, including divergence and curl. By making use of Equation (A.1.7),

$$\vec{B}\cdot(\vec{\nabla}\times\delta\vec{A}) = \vec{\nabla}\cdot(\vec{B}\times\delta\vec{A}) + (\vec{\nabla}\times\vec{B})\cdot\delta\vec{A} = (\vec{\nabla}\times\vec{B})\cdot\delta\vec{A}$$

where the final step is the result of a divergence vanishing from the boundary conditions. Thus,

$$\frac{\partial}{\partial \vec{A}}\left[\vec{B}\cdot(\vec{\nabla}\times\vec{A})\right] = \vec{\nabla}\times\vec{B} \qquad \square$$

Proof. Let $\vec{A}$ be a vector field, and consider the expression

$$\frac{\partial}{\partial \vec{A}}\left[(\vec{\nabla}\cdot\vec{A})^2\right]$$

In the language of variations, we can instead write

$$\frac{\partial}{\partial \vec{A}}\left[(\vec{\nabla}\cdot\vec{A})^2\right] = 2(\vec{\nabla}\cdot\vec{A})(\vec{\nabla}\cdot\delta\vec{A})$$

This equality follows from Clairaut's theorem [28], which allows for the free exchange of differential operators, including divergence and curl. By making use of Equation (A.1.8), noting that $\delta\vec{A}$ is a vector and $\vec{\nabla}\cdot\vec{A}$ is a scalar,

$$2(\vec{\nabla}\cdot\vec{A})(\vec{\nabla}\cdot\delta\vec{A}) = -2\delta\vec{A}\cdot\vec{\nabla}(\vec{\nabla}\cdot\vec{A}) + 2\vec{\nabla}\cdot((\vec{\nabla}\cdot\vec{A})\delta\vec{A}) = -2\delta\vec{A}\cdot\vec{\nabla}(\vec{\nabla}\cdot\vec{A})$$

where the final step is the result of a divergence vanishing from the boundary conditions. Thus,

$$\frac{\partial}{\partial \vec{A}}\left[(\vec{\nabla}\cdot\vec{A})^2\right] = -2\vec{\nabla}(\vec{\nabla}\cdot\vec{A}) \qquad \square$$

Proof. Let $\vec{A}$ and $\vec{B}$ be independent vector fields and λ a scalar field. Consider the expression

$$\frac{\partial}{\partial \vec{A}}\left[\vec{B}\cdot(\vec{\nabla}\times(\lambda\vec{A}))\right]$$

In the language of variations, we can instead write

$$\frac{\partial}{\partial \vec{A}}\left[\vec{B}\cdot(\vec{\nabla}\times(\lambda\vec{A}))\right] = \vec{B}\cdot(\vec{\nabla}\times(\lambda\delta\vec{A}))$$

This equality follows from Clairaut's theorem [28], which allows for the free exchange of differential operators, including divergence and curl. Note that the scalar λ acts as a constant. By making use of Equation (A.1.10),

$$\vec{B}\cdot(\vec{\nabla}\times(\lambda\delta\vec{A})) = \vec{B}\cdot\left[\lambda(\vec{\nabla}\times\delta\vec{A}) - \delta\vec{A}\times(\vec{\nabla}\lambda)\right]$$

By distributing the dot product between the two interior terms, we can see that the first term is of the form of an intermediate step in the proof of Equation (A.1.3), and the second term can be simplified through the use of Equation (A.1.7). Thus,

$$\vec{B} \cdot \left[\lambda(\vec{\nabla} \times \delta\vec{A}) - \delta\vec{A} \times (\vec{\nabla}\lambda)\right] = \left[\lambda(\vec{\nabla} \times \vec{B}) - \vec{\nabla}\lambda \times \vec{B}\right] \cdot \delta\vec{A}$$
$$\frac{\partial}{\partial\vec{A}}\left[\vec{B} \cdot (\vec{\nabla} \times (\lambda\vec{A}))\right] = \lambda(\vec{\nabla} \times \vec{B}) - \vec{\nabla}\lambda \times \vec{B} \qquad \square$$

Proof. Let $\vec{A}$ and $\vec{B}$ be independent vector fields and λ a scalar field. Consider the expression

$$\frac{\partial}{\partial\lambda}\left[\vec{B} \cdot (\vec{\nabla} \times (\lambda\vec{A}))\right]$$

By making use of Equation (A.1.10) and Equation (A.1.7),

$$\begin{aligned}\frac{\partial}{\partial\lambda}\left[\vec{B} \cdot (\vec{\nabla} \times (\lambda\vec{A}))\right] &= \frac{\partial}{\partial\lambda}\left[\vec{B} \cdot (\lambda(\vec{\nabla} \times \vec{A}) - \vec{A} \times (\vec{\nabla}\lambda))\right] \\ &= \frac{\partial}{\partial\lambda}\left[\lambda\vec{B} \cdot (\vec{\nabla} \times \vec{A}) - (\vec{\nabla}\lambda) \cdot (\vec{B} \times \vec{A})\right] \\ &= \vec{B} \cdot (\vec{\nabla} \times \vec{A})\end{aligned}$$

noting that

$$\frac{\partial(\vec{\nabla}\lambda)}{\partial\lambda} = \vec{\nabla}(\frac{\partial\lambda}{\partial\lambda}) = \vec{\nabla}(1) = 0$$

since the gradient is a differential operator and can be exchanged with the partial derivative by Clairaut's theorem [28]. Thus,

$$\frac{\partial}{\partial\lambda}\left[\vec{B} \cdot (\vec{\nabla} \times (\lambda\vec{A}))\right] = \vec{B} \cdot \vec{\nabla} \times \vec{A}$$
$$\frac{\partial}{\partial\lambda}\left[\vec{\nabla} \times (\lambda\vec{A})\right] = \vec{\nabla} \times \vec{A} \qquad \square$$

A.2 Extension to a Second Invariant

The model considered in this section will not be simulated in this work. However, this model is arguably more complete than the single-invariant version considered in the rest of this work, and so is derived in full to benefit any future studies on this topic.

A.2.1 System Action and Hamiltonian

As noted by Seifert in [19], a second invariant can be written for this system, which we will call

$$Y = \frac{1}{2}\epsilon^{abcd}B_{ab}B_{cd} = -4\vec{P} \cdot \vec{Q}. \tag{A.2.1}$$

In analogy to our treatment of the invariant X, we will force $Y = a$ for some constant $a \neq 0$. This modifies the system action to

$$S = \int -\frac{1}{12}F^{abc}F_{abc} - \lambda(X - b) - \mu(Y - a)\mathrm{d}x^4, \tag{A.2.2}$$

where $Y - a = 0$ has been enforced by a new Lagrange multiplier, μ. By once again following Dirac-Bergmann analysis [20], one can fairly easily find that the augmented Hamiltonian for this new system is

$$\mathcal{H}_A = \frac{1}{2}\vec{\Pi}_Q^2 + \vec{\Pi}_Q \cdot (\vec{\nabla} \times \vec{P}) + \frac{1}{2}(\vec{\nabla} \cdot \vec{Q})^2 \tag{A.2.3}$$

$$+\lambda(X-b)+\mu(Y-a)+\vec{u}\cdot\vec{\Pi}_P+u_\lambda\varpi+u_\mu\varphi,$$

where φ is the conjugate momentum to μ and u_μ is a new Lagrange multiplier in analogy to u_λ. We must also have a handful of secondary constraints related to the time derivatives of the weakly constrained quantities $\varpi = 0$, $\vec{\Pi}_P = 0$, and $\varphi = 0$, which in turn are determined by the Poisson brackets of the constrained quantities with the augmented Hamiltonian. In particular,

$$\Psi \equiv \frac{\partial\varpi}{\partial t} = \{\varpi, \mathcal{H}_A\} = -\frac{\partial\mathcal{H}_A}{\partial\lambda} = -(X-b) \tag{A.2.4}$$

$$\vec{\Psi} \equiv \frac{\partial\vec{\Pi}_P}{\partial t} = \{\vec{\Pi}_P, \mathcal{H}_A\} = -\frac{\partial\mathcal{H}_A}{\partial\vec{P}} = 4\lambda\vec{P}+4\mu\vec{Q}-\vec{\nabla}\times\vec{\Pi}_Q \tag{A.2.5}$$

$$\Phi \equiv \frac{\partial\varphi}{\partial t} = \{\varphi, \mathcal{H}_A\} = -\frac{\partial\mathcal{H}_A}{\partial\mu} = 4\vec{P}\cdot\vec{Q}+a. \tag{A.2.6}$$

Each secondary constraint has a time derivative given by

$$\frac{\partial\Psi}{\partial t} = \{\Psi, \mathcal{H}_A\} = 4\left[\vec{u}\cdot\vec{P}-\vec{Q}\cdot(\vec{\Pi}_Q+\vec{\nabla}\times\vec{P})\right] \tag{A.2.7}$$

$$\frac{\partial\vec{\Psi}}{\partial t} = \{\vec{\Psi}, \mathcal{H}_A\} = 4\left[\lambda\vec{u}+\mu(\vec{\Pi}_Q+\vec{\nabla}\times\vec{P})+u_\lambda\vec{P}+u_\mu\vec{Q}+\vec{\nabla}\times(\lambda\vec{Q})-\vec{\nabla}\times(\mu\vec{P})\right] \tag{A.2.8}$$

$$\frac{\partial\Phi}{\partial t} = \{\Phi, \mathcal{H}_A\} = 4\left[\vec{u}\cdot\vec{Q}+\vec{P}\cdot(\vec{\Pi}_P+\vec{\nabla}\times\vec{P})\right] \tag{A.2.9}$$

It is worth noting that the time derivative of Ψ is the same as that found by Seifert for the single-invariant case, though the equation given that work includes an erroneous factor of 4 [19]. Since each secondary constraint is weakly constrained to 0, we must have each of their time derivatives equal to 0 as well. Using this fact, one can show that

$$\vec{u} = \frac{1}{\lambda}\left[\vec{C}-u_\lambda\vec{P}-u_\mu\vec{Q}\right] \tag{A.2.10}$$

$$u_\mu = \frac{1}{Q^2P^2-(\vec{Q}\cdot\vec{P})^2}\left[P^2\vec{Q}\cdot\vec{C}-P^2\lambda A-(\vec{P}\cdot\vec{C})(\vec{P}\cdot\vec{Q})+\lambda B(\vec{P}\cdot\vec{Q})\right] \tag{A.2.11}$$

$$u_\lambda = \frac{1}{P^2}\left[\vec{P}\cdot\vec{C}-\lambda B\right]-\frac{\vec{Q}\cdot\vec{P}}{Q^2P^2-(\vec{Q}\cdot\vec{P})^2}\left[\vec{Q}\cdot\vec{C}-\lambda A-\frac{1}{P^2}\left[(\vec{P}\cdot\vec{C})(\vec{P}\cdot\vec{Q})+\lambda B(\vec{P}\cdot\vec{Q})\right]\right] \tag{A.2.12}$$

where we have defined

$$A \equiv -\vec{P}\cdot(\vec{\Pi}_Q+\vec{\nabla}\times\vec{P}) \tag{A.2.13}$$

$$B \equiv \vec{Q}\cdot(\vec{\Pi}_Q+\vec{\nabla}\times\vec{P}) \tag{A.2.14}$$

$$\vec{C} \equiv \vec{\nabla}\times(\mu\vec{P})-\vec{\nabla}\times(\lambda\vec{Q})-\mu(\vec{\Pi}_Q+\vec{\nabla}\times\vec{P}) \tag{A.2.15}$$

and we have imposed the additional conditions

$$\lambda \neq 0 \tag{A.2.16}$$

$$Q^2P^2-(\vec{Q}\cdot\vec{P})^2 \neq 0. \tag{A.2.17}$$

This final condition is equivalent to $\vec{Q}$ and $\vec{P}$ neither sharing a direction nor having a magnitude of 0.

A.2.2 Time Evolution Equations

As in the single-invariant model, the time evolution equation of each field quantity is found through the use of a Poisson bracket. For the two-invariant model, this process gives

$$\dot{\vec{Q}} = \frac{\mathrm{d}\vec{Q}}{\mathrm{d}t} = \frac{\partial\mathcal{H}_A}{\partial\vec{\Pi}_Q} \tag{A.2.18}$$

$$\dot{\vec{\Pi}}_Q = \frac{\mathrm{d}\vec{\Pi}_Q}{\mathrm{d}t} = -\frac{\partial \mathcal{H}_A}{\partial \vec{Q}} \tag{A.2.19}$$

$$\dot{\vec{P}} = \frac{\mathrm{d}\vec{P}}{\mathrm{d}t} = \frac{\partial \mathcal{H}_A}{\partial \vec{\Pi}_P} = \vec{u} \tag{A.2.20}$$

$$\dot{\vec{\Pi}}_P = \frac{\mathrm{d}\vec{\Pi}_P}{\mathrm{d}t} = -\frac{\partial \mathcal{H}_A}{\partial \vec{P}} \tag{A.2.21}$$

$$\dot{\lambda} = \frac{\mathrm{d}\lambda}{\mathrm{d}t} = \frac{\partial \mathcal{H}_A}{\partial \varpi} = u_\lambda \tag{A.2.22}$$

$$\dot{\varpi} = \frac{\mathrm{d}\varpi}{\mathrm{d}t} = -\frac{\partial \mathcal{H}_A}{\partial \lambda} \tag{A.2.23}$$

$$\dot{\mu} = \frac{\mathrm{d}\mu}{\mathrm{d}t} = \frac{\partial \mathcal{H}_A}{\partial \varphi} = u_\mu \tag{A.2.24}$$

$$\dot{\varphi} = \frac{\mathrm{d}\varphi}{\mathrm{d}t} = -\frac{\partial \mathcal{H}_A}{\partial \mu}. \tag{A.2.25}$$

Again, some of these derivatives are straightforward and have been listed explicitly. We take the direct product between two vector quantities (without an explicit dot product) to represent a dyad so as to imply the appropriate index contractions without explicit Einstein summation notation. The remaining derivatives are given by

$$\frac{\partial \mathcal{H}_A}{\partial \vec{\Pi}_Q} = \vec{\Pi}_Q + \vec{\nabla} \times \vec{P} + \frac{\partial \vec{u}}{\partial \vec{\Pi}_Q} \cdot \vec{\Pi}_P + \frac{\partial u_\lambda}{\partial \vec{\Pi}_Q}\varpi + \frac{\partial u_\mu}{\partial \vec{\Pi}_Q}\varphi \tag{A.2.26}$$

$$\frac{\partial \mathcal{H}_A}{\partial \vec{Q}} = -\vec{\nabla}(\vec{\nabla} \cdot \vec{Q}) + 4\lambda\vec{Q} - 4\mu\vec{P} + \frac{\partial}{\partial \vec{Q}}(\vec{u} \cdot \vec{\Pi}_P) + \frac{\partial u_\lambda}{\partial \vec{Q}}\varpi + \frac{\partial u_\mu}{\partial \vec{Q}}\varphi \tag{A.2.27}$$

$$\frac{\partial \mathcal{H}_A}{\partial \vec{P}} = \vec{\nabla} \times \vec{\Pi}_Q - 4\lambda\vec{P} - 4\mu\vec{Q} + \frac{\partial}{\partial \vec{P}}(\vec{u} \cdot \vec{\Pi}_P) + \frac{\partial u_\lambda}{\partial \vec{P}}\varpi + \frac{\partial u_\mu}{\partial \vec{P}}\varphi \tag{A.2.28}$$

$$\frac{\partial \mathcal{H}_A}{\partial \lambda} = 2Q^2 - 2P^2 - b + \frac{\partial \vec{u}}{\partial \lambda} \cdot \vec{\Pi}_P + \frac{\partial u_\lambda}{\partial \lambda}\varpi + \frac{\partial u_\mu}{\partial \lambda}\varphi \tag{A.2.29}$$

$$\frac{\partial \mathcal{H}_A}{\partial \mu} = -4\vec{P} \cdot \vec{Q} - a + \frac{\partial \vec{u}}{\partial \mu} \cdot \vec{\Pi}_P + \frac{\partial u_\lambda}{\partial \mu}\varpi + \frac{\partial u_\mu}{\partial \mu}\varphi, \tag{A.2.30}$$

where the derivatives of $\vec{u}$ are

$$\frac{\partial \vec{u}}{\partial \vec{\Pi}_Q} = \frac{1}{\lambda}\left[\mu \overleftrightarrow{I} - \frac{\partial u_\lambda}{\partial \vec{\Pi}_Q}\vec{P} - \frac{\partial u_\mu}{\partial \vec{\Pi}_Q}\vec{Q}\right] \tag{A.2.31}$$

$$\frac{\partial}{\partial \vec{Q}}(\vec{u} \cdot \vec{\Pi}_P) = \frac{1}{\lambda}\left[\vec{\nabla}\lambda \times \vec{\Pi}_P - \lambda\vec{\nabla} \times \vec{\Pi}_P - \frac{\partial u_\lambda}{\partial \vec{Q}}(\vec{P} \cdot \vec{\Pi}_P) - \frac{\partial u_\mu}{\partial \vec{Q}}(\vec{Q} \cdot \vec{\Pi}_P)\right] \tag{A.2.32}$$

$$\frac{\partial}{\partial \vec{P}}(\vec{u} \cdot \vec{\Pi}_P) = \frac{1}{\lambda}\left[\mu\vec{\nabla} \times \vec{\Pi}_P - \vec{\nabla}\mu \times \vec{\Pi}_P - u_\lambda - \frac{\partial u_\lambda}{\partial \vec{P}}(\vec{P} \cdot \vec{\Pi}_P) - \frac{\partial u_\mu}{\partial \vec{P}}(\vec{Q} \cdot \vec{\Pi}_P)\right] \tag{A.2.33}$$

$$\frac{\partial \vec{u}}{\partial \lambda} = \frac{1}{\lambda^2}\left[u_\lambda\vec{P} + u_\mu\vec{Q} - \vec{C} - \lambda\vec{\nabla} \times \vec{Q} - \frac{\partial u_\lambda}{\partial \lambda}\lambda\vec{P} - \frac{\partial u_\mu}{\partial \lambda}\lambda\vec{Q}\right] \tag{A.2.34}$$

$$\frac{\partial \vec{u}}{\partial \mu} = -\frac{1}{\lambda}\left[\vec{\Pi}_Q + \frac{\partial u_\lambda}{\partial \mu}\vec{P} + \frac{\partial u_\mu}{\partial \mu}\vec{Q}\right], \tag{A.2.35}$$

where $\overleftrightarrow{I}$ is the unit dyadic, the derivatives of u_μ are

$$\frac{\partial u_\mu}{\partial \vec{\Pi}_Q} = \frac{1}{Q^2P^2 - (\vec{Q} \cdot \vec{P})^2}\left[P^2(\mu\vec{Q} + \lambda\vec{P}) + (\vec{P} \cdot \vec{Q})(\lambda\vec{Q} - \mu\vec{P})\right] \tag{A.2.36}$$

$$\frac{\partial u_\mu}{\partial \vec{Q}} = \frac{1}{Q^2P^2 - (\vec{Q} \cdot \vec{P})^2}\Big[P^2(\vec{C} + \vec{\nabla}\lambda \times \vec{Q} - \lambda\vec{\nabla} \times \vec{Q}) + (\vec{P} \cdot \vec{Q})(\lambda\vec{\Pi}_Q + 2\lambda\vec{\nabla} \times \vec{P} - \vec{\nabla}\lambda \times \vec{P}) \tag{A.2.37}$$

$$+\lambda B\vec{P} - (\vec{P}\cdot\vec{C})\vec{P}\Big] + 2\frac{(\vec{Q}\cdot\vec{P})\vec{P} - P^2\vec{Q}}{\left(Q^2P^2 - (\vec{Q}\cdot\vec{P})^2\right)^2}\Big[P^2(\vec{Q}\cdot\vec{C}) - \lambda A P^2 - (\vec{P}\cdot\vec{C}+\lambda B)(\vec{P}\cdot\vec{Q})\Big]$$

$$\frac{\partial u_\mu}{\partial \vec{P}} = \frac{1}{Q^2P^2 - (\vec{Q}\cdot\vec{P})^2}\Big[P^2(\lambda\vec{\Pi}_Q + 2\lambda\vec{\nabla}\times\vec{P} + \mu\vec{\nabla}\times\vec{Q} - \vec{\nabla}\mu\times\vec{Q}) + 2(\vec{Q}\cdot\vec{C} - \lambda A)\vec{P} \tag{A.2.38}$$
$$+ (\vec{P}\cdot\vec{Q})(\vec{C} + \lambda\vec{\nabla}\times\vec{Q} + \mu\vec{\nabla}\times\vec{P} - \vec{\nabla}\mu\times\vec{P}) - (\vec{P}\cdot\vec{C}+\lambda B)\vec{Q}\Big]$$
$$+ 2\frac{(\vec{Q}\cdot\vec{P})\vec{Q} - Q^2\vec{P}}{\left(Q^2P^2 - (\vec{Q}\cdot\vec{P})^2\right)^2}\Big[P^2(\vec{Q}\cdot\vec{C} - \lambda A) + (\vec{P}\cdot\vec{Q})(\lambda B - \vec{P}\cdot\vec{C})\Big]$$

$$\frac{\partial u_\mu}{\partial \lambda} = \frac{1}{Q^2P^2 - (\vec{Q}\cdot\vec{P})^2}\Big[(\vec{P}\cdot\vec{Q})(B + \vec{P}\cdot\vec{\nabla}\times\vec{Q}) - P^2(A + \vec{Q}\cdot\vec{\nabla}\times\vec{Q})\Big] \tag{A.2.39}$$

$$\frac{\partial u_\mu}{\partial \mu} = \frac{1}{Q^2P^2 - (\vec{Q}\cdot\vec{P})^2}\Big[(\vec{P}\cdot\vec{Q})(\vec{P}\cdot\vec{\Pi}_Q) - P^2\vec{Q}\cdot\vec{\Pi}_Q\Big], \tag{A.2.40}$$

and the derivatives of u_λ are

$$\frac{\partial u_\lambda}{\partial \vec{\Pi}_Q} = \frac{1}{P^2}\left(\mu\vec{P} - \lambda\vec{Q}\right) - \frac{\vec{Q}\cdot\vec{P}}{Q^2P^2 - (\vec{Q}\cdot\vec{P})^2}\left[\mu\vec{Q} + \lambda\vec{P} + \frac{\vec{Q}\cdot\vec{P}}{P^2}\left(\lambda\vec{Q} - \mu\vec{P}\right)\right] \tag{A.2.41}$$

$$\frac{\partial u_\lambda}{\partial \vec{Q}} = \frac{1}{P^2}\left(\vec{\nabla}\lambda\times\vec{P} - \lambda\vec{\nabla}\times\vec{P} - \lambda(\vec{\Pi}_Q + \vec{\nabla}\times\vec{P})\right) \tag{A.2.42}$$
$$- \frac{Q^2P^2\vec{P} - 3(\vec{Q}\cdot\vec{P})^2\vec{P} - 2P^2(\vec{Q}\cdot\vec{P})\vec{Q}}{\left((Q^2P^2) - (\vec{Q}\cdot\vec{P})^2\right)^2}\left[\vec{Q}\cdot\vec{C} - \lambda A + \frac{\vec{P}\cdot\vec{Q}}{P^2}\left(\lambda B - \vec{P}\cdot\vec{C}\right)\right]$$
$$- \frac{\vec{Q}\cdot\vec{P}}{Q^2P^2 - (\vec{Q}\cdot\vec{P})^2}\Bigg[\vec{C} + \vec{\nabla}\lambda\times\vec{Q} - \lambda\vec{\nabla}\times\vec{Q}$$
$$+ \frac{1}{P^2}\left((\lambda B - \vec{P}\cdot\vec{C})\vec{P} + (\vec{P}\cdot\vec{Q})(\lambda\vec{\Pi}_Q + 2\lambda\vec{\nabla}\times\vec{P} - \vec{\nabla}\lambda\times\vec{P})\right)\Bigg]$$

$$\frac{\partial u_\lambda}{\partial \vec{P}} = \frac{1}{P^4}\Big[P^2(\mu\vec{\nabla}\times\vec{P} - \vec{\nabla}\mu\times\vec{P} + \vec{C} - \lambda\vec{\nabla}\times\vec{Q}) + 2\lambda B\vec{P} - 2(\vec{P}\cdot\vec{C})\vec{P}\Big] \tag{A.2.43}$$
$$- \frac{Q^2P^2\vec{Q} - 3(\vec{Q}\cdot\vec{P})^2\vec{Q} - 2Q^2(\vec{Q}\cdot\vec{P})\vec{P}}{\left((Q^2P^2) - (\vec{Q}\cdot\vec{P})^2\right)^2}\left[\vec{Q}\cdot\vec{C} - \lambda A - \frac{\vec{P}\cdot\vec{Q}}{P^2}\left(\lambda B - \vec{P}\cdot\vec{C}\right)\right]$$
$$\frac{\vec{Q}\cdot\vec{P}}{Q^2P^2 - (\vec{Q}\cdot\vec{P})^2}\Bigg[\mu\vec{\nabla}\times\vec{Q} - \vec{\nabla}\mu\times\vec{Q} + \lambda\vec{\Pi}_Q + 2\lambda\vec{\nabla}\times\vec{P}$$
$$+ \frac{1}{P^4}\Bigg(P^2\Big(\lambda B\vec{Q} + \lambda(\vec{P}\cdot\vec{Q})\vec{\nabla}\times\vec{Q} - (\vec{P}\cdot\vec{C})\vec{Q}$$
$$- (\vec{P}\cdot\vec{Q})(\vec{C} + \mu\vec{\nabla}\times\vec{P} - \vec{\nabla}\mu\times\vec{P})\Big) - 2\lambda B(\vec{P}\cdot\vec{Q})\vec{P}\Bigg)\Bigg]$$

$$\frac{\partial u_\lambda}{\partial \lambda} = \frac{\vec{Q}\cdot\vec{P}}{Q^2P^2 - (\vec{Q}\cdot\vec{P})^2}\left[A + \vec{Q}\cdot\vec{\nabla}\times\vec{Q} + \frac{\vec{Q}\cdot\vec{P}}{P^2}\left(B + \vec{P}\cdot\vec{\nabla}\times\vec{Q}\right)\right] - \frac{B}{P^2} \tag{A.2.44}$$

$$\frac{\partial u_\lambda}{\partial \mu} = \frac{\vec{Q}\cdot\vec{P}}{Q^2P^2 - (\vec{Q}\cdot\vec{P})^2}\left[\vec{Q}\cdot\vec{\Pi}_Q - \frac{\vec{P}\cdot\vec{Q}}{P^2}(\vec{P}\cdot\vec{\Pi}_Q)\right] - \frac{1}{P^2}\left(\vec{P}\cdot\vec{\Pi}_Q\right). \tag{A.2.45}$$

At this point, terms which are weakly constrained to 0 may be disregarded throughout.

A.3 Simulation Code Files

A.3.1 Dependencies

This simulation has been developed in Python, and requires a number of dependencies to function. The following dependency list includes all necessary packages and the exact versions used during development and data collection, in the format of the environment manager Anaconda.

```
channels:
 - defaults
dependencies:
 - alabaster=0.7.12=pyhd3eb1b0_0
 - anyio=3.5.0=py38haa95532_0
 - appdirs=1.4.4=pyhd3eb1b0_0
 - argh=0.26.2=py38_0
 - argon2-cffi=21.3.0=pyhd3eb1b0_0
 - argon2-cffi-bindings=21.2.0=py38h2bbff1b_0
 - astroid=2.14.2=py38haa95532_0
 - asttokens=2.0.5=pyhd3eb1b0_0
 - atomicwrites=1.4.0=py_0
 - attrs=22.1.0=py38haa95532_0
 - autopep8=1.5.6=pyhd3eb1b0_0
 - babel=2.11.0=py38haa95532_0
 - backcall=0.2.0=pyhd3eb1b0_0
 - bcrypt=3.2.0=py38h2bbff1b_1
 - beautifulsoup4=4.12.2=py38haa95532_0
 - black=23.3.0=py38haa95532_0
 - blas=1.0=mkl
 - bleach=4.1.0=pyhd3eb1b0_0
 - bottleneck=1.3.5=py38h080aedc_0
 - brotli=1.0.9=h2bbff1b_7
 - brotli-bin=1.0.9=h2bbff1b_7
 - brotlipy=0.7.0=py38h2bbff1b_1003
 - ca-certificates=2023.08.22=haa95532_0
 - certifi=2023.7.22=py38haa95532_0
 - cffi=1.15.1=py38h2bbff1b_3
 - chardet=4.0.0=py38haa95532_1003
 - charset-normalizer=2.0.4=pyhd3eb1b0_0
 - click=8.0.4=py38haa95532_0
 - cloudpickle=2.2.1=py38haa95532_0
 - colorama=0.4.6=py38haa95532_0
 - comm=0.1.2=py38haa95532_0
 - console_shortcut=0.1.1=4
 - contourpy=1.0.5=py38h59b6b97_0
 - cryptography=41.0.2=py38h31511bf_0
 - cycler=0.11.0=pyhd3eb1b0_0
 - debugpy=1.6.7=py38hd77b12b_0
 - decorator=5.1.1=pyhd3eb1b0_0
 - defusedxml=0.7.1=pyhd3eb1b0_0
 - diff-match-patch=20200713=pyhd3eb1b0_0
 - dill=0.3.7=py38haa95532_0
 - docutils=0.18.1=py38haa95532_3
 - entrypoints=0.4=py38haa95532_0
 - executing=0.8.3=pyhd3eb1b0_0
 - ffmpeg=4.2.2=he774522_0
 - flake8=3.9.0=pyhd3eb1b0_0
 - fonttools=4.25.0=pyhd3eb1b0_0
 - freetype=2.12.1=ha860e81_0
```

```
  - future=0.18.3=py38haa95532_0
  - giflib=5.2.1=h8cc25b3_3
  - glib=2.69.1=h5dc1a3c_2
  - icc_rt=2022.1.0=h6049295_2
  - icu=58.2=ha925a31_3
  - idna=3.4=py38haa95532_0
  - imagesize=1.4.1=py38haa95532_0
  - importlib_metadata=6.0.0=hd3eb1b0_0
  - importlib_resources=5.2.0=pyhd3eb1b0_1
  - intel-openmp=2023.1.0=h59b6b97_46319
  - intervaltree=3.1.0=pyhd3eb1b0_0
  - ipykernel=6.25.0=py38h9909e9c_0
  - ipython=8.12.2=py38haa95532_0
  - ipython_genutils=0.2.0=pyhd3eb1b0_1
  - isort=5.9.3=pyhd3eb1b0_0
  - jaraco.classes=3.2.1=pyhd3eb1b0_0
  - jedi=0.17.2=py38haa95532_1
  - jinja2=3.1.2=py38haa95532_0
  - jpeg=9e=h2bbff1b_1
  - json5=0.9.6=pyhd3eb1b0_0
  - jsonschema=4.17.3=py38haa95532_0
  - jupyter_client=8.1.0=py38haa95532_0
  - jupyter_core=5.3.0=py38haa95532_0
  - jupyter_server=1.23.4=py38haa95532_0
  - jupyterlab=3.3.2=pyhd3eb1b0_0
  - jupyterlab_pygments=0.1.2=py_0
  - jupyterlab_server=2.22.0=py38haa95532_0
  - keyring=23.13.1=py38haa95532_0
  - kiwisolver=1.4.4=py38hd77b12b_0
  - lazy-object-proxy=1.6.0=py38h2bbff1b_0
  - lerc=3.0=hd77b12b_0
  - libbrotlicommon=1.0.9=h2bbff1b_7
  - libbrotlidec=1.0.9=h2bbff1b_7
  - libbrotlienc=1.0.9=h2bbff1b_7
  - libclang=14.0.6=default_hb5a9fac_1
  - libclang13=14.0.6=default_h8e68704_1
  - libdeflate=1.17=h2bbff1b_0
  - libffi=3.4.4=hd77b12b_0
  - libiconv=1.16=h2bbff1b_2
  - libpng=1.6.39=h8cc25b3_0
  - libsodium=1.0.18=h62dcd97_0
  - libspatialindex=1.9.3=h6c2663c_0
  - libtiff=4.5.1=hd77b12b_0
  - libwebp=1.2.4=hbc33d0d_1
  - libwebp-base=1.2.4=h2bbff1b_1
  - libxml2=2.10.4=h0ad7f3c_1
  - libxslt=1.1.37=h2bbff1b_1
  - lxml=4.9.2=py38h2bbff1b_0
  - lz4-c=1.9.4=h2bbff1b_0
  - markupsafe=2.1.1=py38h2bbff1b_0
  - matplotlib=3.7.2=py38haa95532_0
  - matplotlib-base=3.7.2=py38h4ed8f06_0
  - matplotlib-inline=0.1.6=py38haa95532_0
  - mccabe=0.6.1=py38haa95532_2
  - mistune=0.8.4=py38he774522_1000
  - mkl=2023.1.0=h6b88ed4_46357
  - mkl-service=2.4.0=py38h2bbff1b_1
  - mkl_fft=1.3.6=py38hf11a4ad_1
  - mkl_random=1.2.2=py38hf11a4ad_1
  - more-itertools=8.12.0=pyhd3eb1b0_0
  - munkres=1.1.4=py_0
  - mypy_extensions=0.4.3=py38haa95532_1
  - nbclassic=0.5.5=py38haa95532_0
  - nbclient=0.5.13=py38haa95532_0
  - nbconvert=6.5.4=py38haa95532_0
  - nbformat=5.7.0=py38haa95532_0
  - nest-asyncio=1.5.6=py38haa95532_0
  - notebook-shim=0.2.2=py38haa95532_0
  - numexpr=2.8.4=py38h7b80656_1
  - numpy=1.24.3=py38h79a8e48_1
  - numpy-base=1.24.3=py38h8a87ada_1
  - numpydoc=1.5.0=py38haa95532_0
  - openssl=1.1.1w=h2bbff1b_0
  - packaging=23.1=py38haa95532_0
  - pandas=1.5.3=py38hf11a4ad_0
  - pandocfilters=1.5.0=pyhd3eb1b0_0
  - paramiko=2.8.1=pyhd3eb1b0_0
  - parso=0.7.0=py_0
  - pathspec=0.10.3=py38haa95532_0
  - pcre=8.45=hd77b12b_0
  - pexpect=4.8.0=pyhd3eb1b0_3
  - pickleshare=0.7.5=pyhd3eb1b0_1003
  - pillow=9.4.0=py38hd77b12b_0
  - pip=23.2.1=py38haa95532_0
  - pkgutil-resolve-name=1.3.10=py38haa95532_0
  - platformdirs=3.10.0=py38haa95532_0
  - pluggy=1.0.0=py38haa95532_1
  - ply=3.11=py38_0
  - pooch=1.4.0=pyhd3eb1b0_0
  - prometheus_client=0.14.1=py38haa95532_0
  - prompt-toolkit=3.0.36=py38haa95532_0
  - psutil=5.9.0=py38h2bbff1b_0
  - ptyprocess=0.7.0=pyhd3eb1b0_2
  - pure_eval=0.2.2=pyhd3eb1b0_0
  - pycodestyle=2.6.0=pyhd3eb1b0_0
  - pycparser=2.21=pyhd3eb1b0_0
  - pydocstyle=6.3.0=py38haa95532_0
  - pyflakes=2.2.0=pyhd3eb1b0_0
  - pygments=2.15.1=py38haa95532_1
  - pylint=2.16.2=py38haa95532_0
  - pyls-black=0.4.6=hd3eb1b0_0
  - pyls-spyder=0.3.2=pyhd3eb1b0_0
  - pynacl=1.5.0=py38h8cc25b3_0
  - pyopenssl=23.2.0=py38haa95532_0
  - pyparsing=3.0.9=py38haa95532_0
  - pyqt=5.9.2=py38hd77b12b_6
  - pyqt5-sip=12.11.0=py38hd77b12b_0
  - pyrsistent=0.18.0=py38h196d8e1_0
  - pysocks=1.7.1=py38haa95532_0
  - python=3.8.17=h6244533_0
  - python-dateutil=2.8.2=pyhd3eb1b0_0
  - python-fastjsonschema=2.16.2=py38haa95532_0
  - python-jsonrpc-server=0.4.0=py_0
  - python-language-server=0.36.2=pyhd3eb1b0_0
  - python-tzdata=2023.3=pyhd3eb1b0_0
  - pytoolconfig=1.2.5=py38haa95532_1
  - pytz=2022.7=py38haa95532_0
  - pywin32=305=py38h2bbff1b_0
  - pywin32-ctypes=0.2.0=py38_1000
  - pywinpty=2.0.10=py38h5da7b33_0
  - pyyaml=6.0=py38h2bbff1b_1
  - pyzmq=25.1.0=py38hd77b12b_0
  - qdarkstyle=2.8.1=py_0
  - qt=5.9.7=vc14h73c81de_0
  - qtawesome=1.2.2=py38haa95532_0
  - qtconsole=5.4.2=py38haa95532_0
  - qtpy=2.2.0=py38haa95532_0
  - requests=2.31.0=py38haa95532_0
  - rope=1.7.0=py38haa95532_0
  - rtree=1.0.1=py38h2eaa2aa_0
  - scipy=1.10.1=py38hdcfc7df_1
  - send2trash=1.8.0=pyhd3eb1b0_1
  - setuptools=68.0.0=py38haa95532_0
  - sip=4.19.13=py38hd77b12b_0
  - six=1.16.0=pyhd3eb1b0_1
  - sniffio=1.2.0=py38haa95532_1
  - snowballstemmer=2.2.0=pyhd3eb1b0_0
  - sortedcontainers=2.4.0=pyhd3eb1b0_0
  - soupsieve=2.4=py38haa95532_0
  - sphinx=5.0.2=py38haa95532_0
  - sphinxcontrib-applehelp=1.0.2=pyhd3eb1b0_0
  - sphinxcontrib-devhelp=1.0.2=pyhd3eb1b0_0
  - sphinxcontrib-htmlhelp=2.0.0=pyhd3eb1b0_0
  - sphinxcontrib-jsmath=1.0.1=pyhd3eb1b0_0
  - sphinxcontrib-qthelp=1.0.3=pyhd3eb1b0_0
  - sphinxcontrib-serializinghtml=1.1.5=pyhd3eb1b0_0
  - spyder=4.2.5=py38haa95532_0
  - spyder-kernels=1.10.2=py38haa95532_0
  - sqlite=3.41.2=h2bbff1b_0
  - stack_data=0.2.0=pyhd3eb1b0_0
  - tbb=2021.8.0=h59b6b97_0
  - terminado=0.17.1=py38haa95532_0
  - textdistance=4.2.1=pyhd3eb1b0_0
  - three-merge=0.1.1=pyhd3eb1b0_0
  - tinycss2=1.2.1=py38haa95532_0
  - tk=8.6.12=h2bbff1b_0
  - toml=0.10.2=pyhd3eb1b0_0
  - tomli=2.0.1=py38haa95532_0
  - tomlkit=0.11.1=py38haa95532_0
  - tornado=6.3.2=py38h2bbff1b_0
  - traitlets=5.7.1=py38haa95532_0
  - typing-extensions=4.7.1=py38haa95532_0
  - typing_extensions=4.7.1=py38haa95532_0
  - ujson=5.4.0=py38hd77b12b_0
  - urllib3=1.26.16=py38haa95532_0
  - vc=14.2=h21ff451_1
  - vs2015_runtime=14.27.29016=h5e58377_2
  - watchdog=1.0.2=py38haa95532_1
  - wcwidth=0.2.5=pyhd3eb1b0_0
  - webencodings=0.5.1=py38_1
  - websocket-client=0.58.0=py38haa95532_4
  - wheel=0.38.4=py38haa95532_0
  - win_inet_pton=1.1.0=py38haa95532_0
  - winpty=0.4.3=4
  - wrapt=1.14.1=py38h2bbff1b_0
  - xz=5.4.2=h8cc25b3_0
  - yaml=0.2.5=he774522_0
  - yapf=0.31.0=pyhd3eb1b0_0
  - zeromq=4.3.4=hd77b12b_0
  - zipp=3.11.0=py38haa95532_0
  - zlib=1.2.13=h8cc25b3_0
  - zstd=1.5.5=hd43e919_0
  - pip:
    - importlib-metadata==6.8.0
    - llvmlite==0.40.1
    - mpmath==1.3.0
    - numba==0.57.1
    - py-pde==0.32.2
    - sympy==1.12
    - tqdm==4.66.1
```

A.3.2 Simulation.py

```python
import pde
import numpy as np
from TimeEvolutionEqns import TimeEvolutionEquations


def simulation(total_time = 10, gridx = 25, gridy = 25, gridz = 25, eps = 0, run = 0, file
↪ = ''):

    #Define the constant b and the simulation grid
    b = -1
    grid = pde.CartesianGrid([[0, gridx-1], [0, gridy-1], [0, gridz-1]], [gridx, gridy,
    ↪ gridz], periodic = [True,True,True])  # generate grid

    #If no file supplied, generate new field data
    if file=='':

        #Define the gradient on this grid
        apply_gradient = grid.make_operator("gradient", bc = ["periodic", "periodic",
        ↪ "periodic"])

        #Generate Pi_Q
        PQ1 = pde.fields.ScalarField.from_expression(grid, "0")
        PQ2 = pde.fields.ScalarField.from_expression(grid, "sin(2*pi*x/25)")
        PQ3 = pde.fields.ScalarField.from_expression(grid, "-cos(2*pi*x/25)")

        #Generate Q
        Q1 = pde.fields.ScalarField.random_uniform(grid)
        Q2 = pde.fields.ScalarField.random_uniform(grid)
        Q3 = pde.fields.ScalarField.random_uniform(grid)


        #Curl of Pi_Q
        grad_PQ1 = apply_gradient(PQ1.data)
        grad_PQ2 = apply_gradient(PQ2.data)
        grad_PQ3 = apply_gradient(PQ3.data)
        assert grad_PQ1.shape == (3, gridx, gridy, gridz)
        assert grad_PQ2.shape == (3, gridx, gridy, gridz)
        assert grad_PQ3.shape == (3, gridx, gridy, gridz)

        curl_PQ = [grad_PQ3[1] - grad_PQ2[2], grad_PQ1[2] - grad_PQ3[0], grad_PQ2[0] -
        ↪  grad_PQ1[1]]

        #Find direction of P from direction of the curl of Pi_Q
        curl_PQ1 = np.reshape(curl_PQ[0].data, (-1, 1))
        curl_PQ2 = np.reshape(curl_PQ[1].data, (-1, 1))
        curl_PQ3 = np.reshape(curl_PQ[2].data, (-1, 1))
        curl_PQ_vecs = np.concatenate((curl_PQ1, curl_PQ2, curl_PQ3),axis = 1)
        curl_PQ_vecs = curl_PQ_vecs.reshape([gridx, gridy, gridz, 3])
        mag_curl_PQ = (np.sqrt((curl_PQ_vecs ** 2).sum(-1))[...,
        ↪  np.newaxis]).reshape([gridx, gridy, gridz, 1])
        dir_P = curl_PQ_vecs/mag_curl_PQ

        #Find magnitudes of P from Q
        Q_vecs1 = np.reshape(Q1.data, (-1, 1))
        Q_vecs2 = np.reshape(Q2.data, (-1, 1))
        Q_vecs3 = np.reshape(Q3.data, (-1, 1))
        Q_vecs = np.concatenate((Q_vecs1, Q_vecs2, Q_vecs3),axis = 1)
        Q_vecs = Q_vecs.reshape([gridx, gridy, gridz, 3])
        mag_P = np.sqrt(((Q_vecs ** 2)).sum(-1)[..., np.newaxis] - b/2)

        #Recreate P and split it into scalar fields
        P = mag_P*dir_P
        P = np.split(P, 3, axis=3)
        P1_data, P2_data, P3_data = P
        P1 = pde.ScalarField(grid, data=np.array(P1_data).reshape([gridx, gridy, gridz]))
        P2 = pde.ScalarField(grid, data=np.array(P2_data).reshape([gridx, gridy, gridz]))
        P3 = pde.ScalarField(grid, data=np.array(P3_data).reshape([gridx, gridy, gridz]))

        #Define lambda
        L = (1/(4*(P1*P1 + P2*P2 + P3*P3)))*(P1*curl_PQ[0] + P2*curl_PQ[1] +
        ↪  P3*curl_PQ[2])

        #Set the time to initial parameters
        count = 0
        time = 0

    #If a file is supplied, load in field data
    else:
        npzfile = np.load(file)

        Q1_data = npzfile['arr_0']
        Q2_data = npzfile['arr_1']
        Q3_data = npzfile['arr_2']

        Pi_Q1_data = npzfile['arr_3']
        Pi_Q2_data = npzfile['arr_4']
        Pi_Q3_data = npzfile['arr_5']

        P1_data = npzfile['arr_6']
        P2_data = npzfile['arr_7']
        P3_data = npzfile['arr_8']

        L_data = npzfile['arr_9']

        count_arr = npzfile['arr_10']
        time_arr = npzfile['arr_11']
```

```python

        #Convert loaded data to fields and scalars, as appropriate
        Q1 = pde.ScalarField(grid, data=np.array(Q1_data).reshape([gridx, gridy, gridz]))
        Q2 = pde.ScalarField(grid, data=np.array(Q2_data).reshape([gridx, gridy, gridz]))
        Q3 = pde.ScalarField(grid, data=np.array(Q3_data).reshape([gridx, gridy, gridz]))

        PQ1 = pde.ScalarField(grid, data=np.array(Pi_Q1_data).reshape([gridx, gridy,
        ↪   gridz]))
        PQ2 = pde.ScalarField(grid, data=np.array(Pi_Q2_data).reshape([gridx, gridy,
        ↪   gridz]))
        PQ3 = pde.ScalarField(grid, data=np.array(Pi_Q3_data).reshape([gridx, gridy,
        ↪   gridz]))

        P1 = pde.ScalarField(grid, data=np.array(P1_data).reshape([gridx, gridy, gridz]))
        P2 = pde.ScalarField(grid, data=np.array(P2_data).reshape([gridx, gridy, gridz]))
        P3 = pde.ScalarField(grid, data=np.array(P3_data).reshape([gridx, gridy, gridz]))

        L = pde.ScalarField(grid, data=np.array(L_data).reshape([gridx, gridy, gridz]))

        count = count_arr[0]
        time = time_arr[0]



    #Define the problem, and solve it
    eq = TimeEvolutionEquations(gridx, gridy, gridz, grid, run, eps, count, time)  #
    ↪   define the PDE

    field = pde.FieldCollection([Q1, Q2, Q3, PQ1, PQ2, PQ3, P1, P2, P3, L])

    solver = pde.ExplicitSolver(eq, scheme = 'runge-kutta', adaptive=True)
    controller = pde.Controller(solver, t_range=total_time)
    solution = controller.run(field, dt=0.001)

    solution.plot()

def main():
    simulation(total_time = 1, run=1, eps=0)

if(__name__=='__main__'):
    main()

```

A.3.3 TimeEvolutionEquations.py

```python
from pde import PDEBase, FieldCollection, CartesianGrid, VectorField, ScalarField
import numpy as np
import pandas as pd
from scipy.spatial import ConvexHull

class TimeEvolutionEquations(PDEBase):
    """ Time Evoltuon Equations for Rank-2, Anti-Symmetric Tensor field """

    def __init__(self, gridx, gridy, gridz, grid, run, eps, count = 0, time = 0):

        #initialize the PDEBase
        super().__init__()

        #store the boundary conditions
        self.bc = ["periodic", "periodic", "periodic"]

        #store the grid
        self.grid = grid
        self.gridx = gridx
        self.gridy = gridy
        self.gridz = gridz
        self.run = run

        self.count = count
        self.time = time

        self.monopole_P_vecs = []

        self.epsilon = eps
        self.b = -1

        #For tracking when to next take a measurement
        self.last_monopole_t = self.time
        self.last_corr_t = self.time

        self.corr_active = True

    def evolution_rate(self, state, t=0):
        """
        Define the actual equations of motion for the field

        Parameters
        ----------
        state : packaging of fields
            Contains the fields being solved. Order convention is Q1, Q2, Q3, Pi_Q1,
            ↪  Pi_Q2, Pi_Q3, P1, P2, P3, L (lambda)
        t : float, optional
            User should not set, and this convention is consistent with SciPy and PyPDE.
            ↪  Time step of the system. The default is 0.

        Returns
        -------
        None.

        """



        #Define the gradient and divergence on this grid
        apply_gradient = self.grid.make_operator("gradient", bc = self.bc)

        #Load in the current state
        Q1, Q2, Q3, Pi_Q1, Pi_Q2, Pi_Q3, P1, P2, P3, L = state


        #Find magnitudes of P from Q to satisfy X = 0 (To handle procedurally built up
        ↪  simulation errors)
        Q_vecs1 = np.reshape(Q1.data, (-1, 1))
        Q_vecs2 = np.reshape(Q2.data, (-1, 1))
        Q_vecs3 = np.reshape(Q3.data, (-1, 1))
        Q_vecs = np.concatenate((Q_vecs1, Q_vecs2, Q_vecs3),axis = 1)
        Q_vecs = Q_vecs.reshape([self.gridx, self.gridy, self.gridz, 3])
        mag_P = np.sqrt(((Q_vecs ** 2)).sum(-1)[..., np.newaxis] -
        ↪  self.b/2).reshape([self.gridx,self.gridy,self.gridz, 1])

        #Extract the current magnitudes of P
        P_vecs1 = np.reshape(P1.data, (-1, 1))
        P_vecs2 = np.reshape(P2.data, (-1, 1))
        P_vecs3 = np.reshape(P3.data, (-1, 1))
        P_vecs = np.concatenate((P_vecs1, P_vecs2, P_vecs3),axis = 1)
        P_vecs = P_vecs.reshape([self.gridx, self.gridy, self.gridz, 3])
        current_mag_P = np.sqrt(((P_vecs ** 2)).sum(-1)[...,
        ↪  np.newaxis]).reshape([self.gridx,self.gridy,self.gridz, 1])

        #Get the current direction of P
        dir_P = P_vecs/current_mag_P

        #For the new P vectors by multiplying new the magnitude by the direction
        P = dir_P*mag_P

        #Split P into P1, P2, and P3
        P = np.split(P, 3, axis=3)
        P1_data, P2_data, P3_data = P

        #Reset the scalar field objects with the adjusted P magntidues
        P1 = ScalarField(self.grid, data=np.array(P1_data).reshape([self.gridx,
        ↪  self.gridy, self.gridz]))
        P2 = ScalarField(self.grid, data=np.array(P2_data).reshape([self.gridx,
        ↪  self.gridy, self.gridz]))
        P3 = ScalarField(self.grid, data=np.array(P3_data).reshape([self.gridx,
        ↪  self.gridy, self.gridz]))



        #Define difficult vector quantities, like curls

        #Curl of P
        grad_P1 = apply_gradient(P1.data)
        grad_P2 = apply_gradient(P2.data)
        grad_P3 = apply_gradient(P3.data)
        assert grad_P1.shape == (3, self.gridx, self.gridy, self.gridz)
        assert grad_P2.shape == (3, self.gridx, self.gridy, self.gridz)
        assert grad_P3.shape == (3, self.gridx, self.gridy, self.gridz)

        curl_P = [grad_P3[1] - grad_P2[2], grad_P1[2] - grad_P3[0], grad_P2[0] -
        ↪  grad_P1[1]]

        #gradient of divergence of Q
        vf = VectorField.from_scalars([Q1, Q2, Q3])
        div_Q = vf.divergence(self.bc)
        grad_div_Q = apply_gradient(div_Q.data)
        assert grad_div_Q.shape == (3, self.gridx, self.gridy, self.gridz)

        #gradient of lambda
        grad_L = apply_gradient(L.data)
        assert grad_L.shape == (3, self.gridx, self.gridy, self.gridz)

        #Lambda * Q
        LQ = [L*Q1, L*Q2, L*Q3]
```

```python

        #curl of LQ
        grad_LQ1 = apply_gradient(L.data*Q1.data)
        grad_LQ2 = apply_gradient(L.data*Q2.data)
        grad_LQ3 = apply_gradient(L.data*Q3.data)
        assert grad_LQ1.shape == (3, self.gridx, self.gridy, self.gridz)
        assert grad_LQ2.shape == (3, self.gridx, self.gridy, self.gridz)
        assert grad_LQ3.shape == (3, self.gridx, self.gridy, self.gridz)

        curl_LQ = [grad_LQ3[1] - grad_LQ2[2], grad_LQ1[2] - grad_LQ3[0], grad_LQ2[0] -
        ↪  grad_LQ1[1]]

        #grad_L cross P
        GLCP = [grad_L[1]*P3 - grad_L[2]*P2, grad_L[2]*P1 - grad_L[0]*P3, grad_L[0]*P2 -
        ↪  grad_L[1]*P1]

        #u_L
        u_L = -(1/(P1*P1 + P2*P2 + P3*P3)) * ((P1*curl_LQ[0] + P2*curl_LQ[1] +
        ↪  P3*curl_LQ[2]) + ((LQ[0]*(Pi_Q1 + curl_P[0])) + (LQ[1]*(Pi_Q2 + curl_P[1])) +
        ↪  (LQ[2]*(Pi_Q3 + curl_P[2]))))

        #vec_u
        vec_u = [(-1/L)*(u_L*P1 + curl_LQ[0]), (-1/L)*(u_L*P2 + curl_LQ[1]),
        ↪  (-1/L)*(u_L*P3 + curl_LQ[2])]

        #Complicated scalar A
        A = (P1*Pi_Q1 + P2*Pi_Q2 + P3*Pi_Q3)/(L*(P1*P1 + P2*P2 + P3*P3))

        #Calculate time derivatives
        Q1_t = Pi_Q1 + curl_P[0]
        Q2_t = Pi_Q2 + curl_P[1]
        Q3_t = Pi_Q3 + curl_P[2]
        Pi_Q1_t = grad_div_Q[0] - 4*L*Q1 - self.epsilon*(Pi_Q1 + curl_P[0])
        Pi_Q2_t = grad_div_Q[1] - 4*L*Q2 - self.epsilon*(Pi_Q2 + curl_P[1])
        Pi_Q3_t = grad_div_Q[2] - 4*L*Q3 - self.epsilon*(Pi_Q3 + curl_P[2])
        P1_t = vec_u[0]
        P2_t = vec_u[1]
        P3_t = vec_u[2]
        L_t = u_L


        #Curl of Pi_Q for the calculation of Psi vec
        grad_PQ1 = apply_gradient(Pi_Q1.data)
        grad_PQ2 = apply_gradient(Pi_Q2.data)
        grad_PQ3 = apply_gradient(Pi_Q3.data)
        assert grad_PQ1.shape == (3, self.gridx, self.gridy, self.gridz)
        assert grad_PQ2.shape == (3, self.gridx, self.gridy, self.gridz)
        assert grad_PQ3.shape == (3, self.gridx, self.gridy, self.gridz)

        curl_PQ = [grad_PQ3[1] - grad_PQ2[2], grad_PQ1[2] - grad_PQ3[0], grad_PQ2[0] -
        ↪  grad_PQ1[1]]

        #Export diagnostic data to file
        if self.count % 1000 == 0:
            diagDict = {
                "Index": self.count,
                "Time": t + self.time,
                "Hamiltonian": self.calculate_hamiltonian(state),
                "magQ": np.sum(np.sqrt((Q1*Q1 + Q2*Q2 + Q3*Q3).data)),
                "magP": np.sum(np.sqrt((P1*P1 + P2*P2 + P3*P3).data)),
                                "magPiQ": np.sum(np.sqrt((Pi_Q1*Pi_Q1 + Pi_Q2*Pi_Q2 +
                                ↪  Pi_Q3*Pi_Q3).data)),
                "magP_var": 1/(self.gridx*self.gridy*self.gridz)*np.sum((P1*P1 + P2*P2 +
                ↪  P3*P3).data) -
                ↪  np.power(1/(self.gridx*self.gridy*self.gridz)*np.sum(np.sqrt((P1*P1 +
                ↪  P2*P2 + P3*P3).data)), 2),
                "magQ_var": 1/(self.gridx*self.gridy*self.gridz)*np.sum((Q1*Q1 + Q2*Q2 +
                ↪  Q3*Q3).data) -
                ↪  np.power(1/(self.gridx*self.gridy*self.gridz)*np.sum(np.sqrt((Q1*Q1 +
                ↪  Q2*Q2 + Q3*Q3).data)), 2),
                "magPiQ_var": 1/(self.gridx*self.gridy*self.gridz) * np.sum((Pi_Q1*Pi_Q1 +
                ↪  Pi_Q2*Pi_Q2 + Pi_Q3*Pi_Q3).data) -
                ↪  np.power(1/(self.gridx*self.gridy*self.gridz) *
                ↪  np.sum(np.sqrt((Pi_Q1*Pi_Q1 + Pi_Q2*Pi_Q2 + Pi_Q3*Pi_Q3).data)), 2),
                "Psi": np.sum(2*(Q1*Q1 + Q2*Q2 + Q3*Q3 - P1*P1 - P2*P2 - P3*P3).data -
                ↪  self.b),
                "L": np.sum(L.data),
                "Psivec1":np.sum(4*L.data*P1.data - curl_PQ[0]),
                "Psivec2":np.sum(4*L.data*P2.data - curl_PQ[1]),
                "Psivec3":np.sum(4*L.data*P3.data - curl_PQ[2])
            }

            dataframe = pd.DataFrame([diagDict])
            if (self.time + t) == 0:
                dataframe.to_csv("DiagnosticData_run" + str(self.run) + ".csv", header =
                ↪  True, index = False)
            else:
                dataframe.to_csv("DiagnosticData_run" + str(self.run) + ".csv", header =
                ↪  False, index = False, mode = 'a')

        #Export monopole counts every 0.25 iterations or so
        if (self.time + t) - self.last_monopole_t >= 0.25 or t==0:

            self.last_monopole_t = self.time + t

            #reshape the 3 P fields into an array of 3-dimensional points in grid-space
            field_points = np.array([P1.data.flatten(), P2.data.flatten(),
            ↪  P3.data.flatten()])
            pointsArray = np.array(list(zip(field_points[0], field_points[1],
            ↪  field_points[2]))).reshape(self.gridx, self.gridy, self.gridz, 3)
            pointsArray = pointsArray.copy() / np.sqrt((pointsArray.copy() **
            ↪  2).sum(-1))[..., np.newaxis]

            #self.find_and_export_monopoles_hull(pointsArray, t)
            self.find_and_export_monopoles_triangulation(pointsArray, t)

        #Export simulation state every 5000 steps
        if self.count % 5000 == 0:

            #Save current simulation state to file so it can be accessed again later
            count_arr = np.array([self.count])
            time_arr = np.array([t + self.time])
            np.savez('FieldData_run' + str(self.run), Q1.data, Q2.data, Q3.data,
            ↪  Pi_Q1.data, Pi_Q2.data, Pi_Q3.data, P1.data, P2.data, P3.data, L.data,
            ↪  count_arr, time_arr)
            np.savez('Monopole_P_vecs_run' + str(self.run), self.monopole_P_vecs)

        #Export correlations to file every 5 iterations
        if ((self.time + t)  - self.last_corr_t >= 5 or t==0) and self.corr_active:

            self.last_corr_t = self.time + t

            correlations = self.corr_in_grid([1, 2, 4, 6, 8], P1.data, P2.data, P3.data)

            #export correlations to file
            corrDict = {
                "Index": self.count,
                "Time": t + self.time,
                "Corr1x": correlations[0][0],
                "Corr1y": correlations[1][0],
                "Corr1z": correlations[2][0],
                "Corr2x": correlations[0][1],
                "Corr2y": correlations[1][1],
                "Corr2z": correlations[2][1],
                "Corr4x": correlations[0][2],
                "Corr4y": correlations[1][2],
                "Corr4z": correlations[2][2],
                "Corr6x": correlations[0][3],
                "Corr6y": correlations[1][3],
                "Corr6z": correlations[2][3],
                "Corr8x": correlations[0][4],
                "Corr8y": correlations[1][4],
                "Corr8z": correlations[2][4]
            }

            dataframe = pd.DataFrame([corrDict])
            if (self.time + t) ==0:
                dataframe.to_csv("CorrelationsOutP_run" + str(self.run) + ".csv", header =
                ↪  True, index = False)
            else:
                dataframe.to_csv("CorrelationsOutP_run" + str(self.run) + ".csv", header =
                ↪  False, index = False, mode = 'a')

            correlations = self.corr_in_grid([1, 2, 4, 6, 8], Q1.data, Q2.data, Q3.data)

            #export correlations to file
            corrDict = {
                "Index": self.count,
                "Time": t + self.time,
                "Corr1x": correlations[0][0],
                "Corr1y": correlations[1][0],
                "Corr1z": correlations[2][0],
                "Corr2x": correlations[0][1],
                "Corr2y": correlations[1][1],
                "Corr2z": correlations[2][1],
                "Corr4x": correlations[0][2],
                "Corr4y": correlations[1][2],
                "Corr4z": correlations[2][2],
                "Corr6x": correlations[0][3],
                "Corr6y": correlations[1][3],
                "Corr6z": correlations[2][3],
                "Corr8x": correlations[0][4],
                "Corr8y": correlations[1][4],
                "Corr8z": correlations[2][4]
            }

            dataframe = pd.DataFrame([corrDict])
            if (self.time + t)==0:
                dataframe.to_csv("CorrelationsOutQ_run" + str(self.run) + ".csv", header =
                ↪  True, index = False)
            else:
                dataframe.to_csv("CorrelationsOutQ_run" + str(self.run) + ".csv", header =
                ↪  False, index = False, mode = 'a')

            correlations = self.corr_in_grid([1, 2, 4, 6, 8], Pi_Q1.data, Pi_Q2.data,
            ↪  Pi_Q3.data)

            #export correlations to file
            corrDict = {
                "Index": self.count,
                "Time": t + self.time,
                "Corr1x": correlations[0][0],
                "Corr1y": correlations[1][0],
                "Corr1z": correlations[2][0],
                "Corr2x": correlations[0][1],
                "Corr2y": correlations[1][1],
                "Corr2z": correlations[2][1],
                "Corr4x": correlations[0][2],
                "Corr4y": correlations[1][2],
                "Corr4z": correlations[2][2],
                "Corr6x": correlations[0][3],
                "Corr6y": correlations[1][3],
                "Corr6z": correlations[2][3],
                "Corr8x": correlations[0][4],
                "Corr8y": correlations[1][4],
                "Corr8z": correlations[2][4]
            }

            dataframe = pd.DataFrame([corrDict])
            if (self.time + t) ==0:
                dataframe.to_csv("CorrelationsOutPiQ_run" + str(self.run) + ".csv", header
                ↪  = True, index = False)
            else:
                dataframe.to_csv("CorrelationsOutPiQ_run" + str(self.run) + ".csv", header
                ↪  = False, index = False, mode = 'a')

        self.count = self.count + 1

        #Return time derivatiives
        return FieldCollection([Q1_t, Q2_t, Q3_t, Pi_Q1_t, Pi_Q2_t, Pi_Q3_t, P1_t, P2_t,
        ↪  P3_t, L_t])

    def calculate_hamiltonian(self, state):

        #Define the gradient and divergence on this grid
        apply_gradient = self.grid.make_operator("gradient", bc = self.bc)

        #Load in the current state
        Q1, Q2, Q3, Pi_Q1, Pi_Q2, Pi_Q3, P1, P2, P3, L = state

        #Curl of P
        grad_P1 = apply_gradient(P1.data)
        grad_P2 = apply_gradient(P2.data)
        grad_P3 = apply_gradient(P3.data)
        assert grad_P1.shape == (3, self.gridx, self.gridy, self.gridz)
        assert grad_P2.shape == (3, self.gridx, self.gridy, self.gridz)
        assert grad_P3.shape == (3, self.gridx, self.gridy, self.gridz)

        curl_P = [grad_P3[1] - grad_P2[2], grad_P1[2] - grad_P3[0], grad_P2[0] -
        ↪  grad_P1[1]]

        #divergence of Q
        vf = VectorField.from_scalars([Q1, Q2, Q3])
        div_Q = vf.divergence(self.bc)

        #Calculate H_A
        H = (1/2)*(Pi_Q1*Pi_Q1 + Pi_Q2*Pi_Q2 + Pi_Q3*Pi_Q3) + (Pi_Q1*curl_P[0] +
        ↪  Pi_Q2*curl_P[1] + Pi_Q3*curl_P[2]) + (1/2)*(div_Q*div_Q)

        return np.sum(H.data)

    def find_and_export_monopoles_hull(self, pointsArray, t):
        #reset counter tracking number of detected monopoles
        monopole_counter = 0

        #for each cube of 8 adjacent points, check for a monopole
        for point in np.vstack(np.meshgrid(np.array(range(0,
        ↪  self.gridx)),np.array(range(0, self.gridy)),np.array(range(0,
        ↪  self.gridz)))).reshape(3, -1).T:

            #extract the coordinates of the point
            i,j,k = point

            #reset boolean tracking whether a monopole has been found
            monopole_exists = True

            #generate convex hull from 8 points, last point is the origin and is used to
            ↪  determine if the origin is inside the hull
            hullPoints = [pointsArray[i, j, k], pointsArray[(i + 1)%(self.gridx), j, k],
            ↪  pointsArray[i, (j + 1)%(self.gridy), k], pointsArray[i, j, (k +
            ↪  1)%(self.gridz)], pointsArray[(i + 1)%(self.gridx), (j + 1)%(self.gridy),
            ↪  k], pointsArray[(i + 1)%(self.gridx), j, (k + 1)%(self.gridz)],
            ↪  pointsArray[i, (j + 1)%(self.gridy), (k + 1)%(self.gridz)], pointsArray[(i
            ↪  + 1)%(self.gridx), (j + 1)%(self.gridy), (k + 1)%(self.gridz)],
            ↪  np.array([0,0,0])]
            hull = ConvexHull(hullPoints, qhull_options = 'QG-8')

            #if every edge can be seen from the origin, the origin is inside the hull
            for edge in hull.good:
                if not edge:
                    monopole_exists = False
                    break

            #if applicable, increment the monopole counter
            if monopole_exists:
                monopole_counter = monopole_counter+1
                self.monopole_P_vecs.append(hullPoints)

        #export monopole counting to file
        monopoleDictH = {
```

```python
            "Index": self.count,
            "Time": t + self.time,
            "Monopoles": monopole_counter
        }

        dataframeH = pd.DataFrame([monopoleDictH])
        if (self.time + t)==0:
            dataframeH.to_csv("TestHullMonopoleOut_run" + str(self.run) + ".csv", header = True, index = False)
        else:
            dataframeH.to_csv("TestHullMonopoleOut_run" + str(self.run) + ".csv", header = False, index = False, mode = 'a')

    def find_and_export_monopoles_triangulation(self, pointsArray, t):

        #reset counter tracking number of detected monopoles
        monopole_counter = [0,0,0,0,0,0]
        antimonopole_counter = [0,0,0,0,0,0]
        deg7 = 0
        deg0 = 0

        #for each cube of 8 adjacent points, check for a monopole
        for point in np.vstack(np.meshgrid(np.array(range(0, self.gridx)),np.array(range(0, self.gridy)),np.array(range(0, self.gridz)))).reshape(3, -1).T:

            #extract the coordinates of the point
            i,j,k = point

            #Get all 8 corners of the cube, with wrapping around the edges
            cubePoints = [pointsArray[i, j, k], pointsArray[(i + 1)%(self.gridx), j, k], pointsArray[i, (j + 1)%(self.gridy), k], pointsArray[i, j, (k + 1)%(self.gridz)], pointsArray[(i + 1)%(self.gridx), (j + 1)%(self.gridy), k], pointsArray[(i + 1)%(self.gridx), j, (k + 1)%(self.gridz)], pointsArray[i, (j + 1)%(self.gridy), (k + 1)%(self.gridz)], pointsArray[(i + 1)%(self.gridx), (j + 1)%(self.gridy), (k + 1)%(self.gridz)]]

            #Arrange into oriented triangles (all counter-clockwise relative to the exterior of the cube)
            test_triangles = [[cubePoints[3], cubePoints[7], cubePoints[5]], [cubePoints[3], cubePoints[6], cubePoints[7]], [cubePoints[6], cubePoints[2], cubePoints[4]], [cubePoints[7], cubePoints[6], cubePoints[4]], [cubePoints[2], cubePoints[0], cubePoints[4]], [cubePoints[4], cubePoints[0], cubePoints[1]], [cubePoints[0], cubePoints[2], cubePoints[6]], [cubePoints[3], cubePoints[0], cubePoints[6]], [cubePoints[1], cubePoints[7], cubePoints[4]], [cubePoints[5], cubePoints[7], cubePoints[1]], [cubePoints[3], cubePoints[5], cubePoints[1]], [cubePoints[3], cubePoints[1], cubePoints[0]]]

            #Choose a point within the first triangle, normalized to be on the unit sphere
            triangle = test_triangles[0]
            p = np.array([triangle[0][0] + triangle[1][0] + triangle[2][0], triangle[0][1] + triangle[1][1] + triangle[2][1], triangle[0][2] + triangle[1][2] + triangle[2][2]])
            p = p.copy() / np.sqrt((p.copy() ** 2).sum(-1))[..., np.newaxis]

            #reset the degree counter for this particular cube
            degree_counter = 0

            for triangle in test_triangles:
                degree_counter = degree_counter + self.check_triangle(triangle, p)

            if degree_counter <=6 and degree_counter >0:
                monopole_counter[degree_counter - 1] = monopole_counter[degree_counter -
                 ↪  1] + 1
            elif degree_counter >=-6 and degree_counter <0:
                antimonopole_counter[-1*degree_counter - 1] =
                 ↪  antimonopole_counter[-1*degree_counter - 1] + 1
            elif degree_counter>=7 or degree_counter<=-7:
                deg7 = deg7+1
            else:
                deg0 = deg0 + 1

        #export monopole counting to file
        monopoleDictT = {
            "Index": self.count,
            "Time": t + self.time,
            "Deg1Monopole": monopole_counter[0],
            "Deg1Antimonopole": antimonopole_counter[0],
            "Deg2Monopole": monopole_counter[1],
            "Deg2Antimonopole": antimonopole_counter[1],
            "Deg3Monopole": monopole_counter[2],
            "Deg3Antimonopole": antimonopole_counter[2],
            "Deg4Monopole": monopole_counter[3],
            "Deg4Antimonopole": antimonopole_counter[3],
            "Deg5Monopole": monopole_counter[4],
            "Deg5Antimonopole": antimonopole_counter[4],
            "Deg6Monopole": monopole_counter[5],
            "Deg6Antimonopole": antimonopole_counter[5],
            "Deg7Plus": deg7
        }

        dataframeT = pd.DataFrame([monopoleDictT])
        if (self.time + t)==0:
            dataframeT.to_csv("TriangleMonopoleOut_run" + str(self.run) + ".csv", header =
             ↪  True, index = False)
        else:
            dataframeT.to_csv("TriangleMonopoleOut_run" + str(self.run) + ".csv", header =
             ↪  False, index = False, mode = 'a')




    def check_triangle(self, triangle, p):
        """Checks a triangle for orientation preservation or reversal and inclusion of p
         ↪  (c.f. the function h in Section 2)"""
        v1, v2, v3 = triangle
        trior = np.sign(np.linalg.det(np.array([v1, v2, v3])))
        v12or = np.sign(np.linalg.det(np.array([v1, v2, p])))
        v23or = np.sign(np.linalg.det(np.array([v2, v3, p])))
        v31or = np.sign(np.linalg.det(np.array([v3, v1, p])))

        if((trior == v12or) and (v12or == v23or) and (v23or == v31or)):
            return int(trior)
        else:
            return 0

    def correlation_at_p(self, center_point, radius, F):
        """The correlation at a given point p for all supplied radii (see Section 2.6)"""

        corr_tot = 0
        corr_num = 0

        F_av = np.sum(F)/F.size

        F_var = np.sum(np.power(F - F_av, 2))/F.size

        center_vec = np.array([F[center_point[0], center_point[1], center_point[2]]])

        vec1 = center_vec - F_av

        #sum over all points within a certain Manhattan distance
        for i in range(-1*radius, radius + 1, 1):
            for j in range(-1*radius + abs(i), radius - abs(i) + 1, 1):
                for k in range(-1*radius + abs(i) + abs(j), radius - abs(i) - abs(j) + 1,
                ↪  1):
                    corr_num = corr_num + 1

                    vec2 = np.array([F[(center_point[0] + i)%self.gridx, (center_point[1]
                    ↪  + j)%self.gridy, (center_point[2] + k)%self.gridz]]) - F_av

                    corr_tot = corr_tot + np.dot(vec1, vec2)

        corr_tot = corr_tot / F_var

        return corr_tot / corr_num

    def corr_in_grid(self, radii, F1, F2, F3):
        """Sub-method of the above (see Section 2.6)"""

        xcorr_list = []
        ycorr_list = []
        zcorr_list = []

        for radius in radii:

            xcorrs = 0
            ycorrs = 0
            zcorrs = 0
            corr_num = 0

            for point in np.vstack(np.meshgrid(np.array(range(0,
            ↪  self.gridx)),np.array(range(0, self.gridy)),np.array(range(0,
            ↪  self.gridz)))).reshape(3, -1).T:
                xcorrs = xcorrs + self.correlation_at_p(point, radius, F1)
                ycorrs = ycorrs + self.correlation_at_p(point, radius, F2)
                zcorrs = zcorrs + self.correlation_at_p(point, radius, F3)
                corr_num = corr_num + 1

            avcorrx = xcorrs / corr_num
            avcorry = ycorrs / corr_num
            avcorrz = zcorrs / corr_num

            xcorr_list.append(avcorrx)
            ycorr_list.append(avcorry)
            zcorr_list.append(avcorrz)

            corr_list = [xcorr_list, ycorr_list, zcorr_list]

        return corr_list
```

A.3.4 RunOneSimulation.py

```python
import numpy as np
import pandas as pd
import argparse
from Simulation import simulation

#set up parser to interact with the cluster
parser = argparse.ArgumentParser(description='Run one simulation')
parser.add_argument("f")
parser.add_argument("total_time",
                    help='the number of additional iterations this simulation should
                    ↪  complete', type = int)
parser.add_argument("gridx",
                    help='the size of the x dimension of the simulation grid', type = int)
parser.add_argument("gridy",
                    help='the size of the y dimension of the simulation grid', type = int)
parser.add_argument("gridz",
                    help='the size of the z dimension of the simulation grid', type = int)
parser.add_argument("eps",
                    help='value of the constant eps (should typically be 0)', type =
                    ↪  float)
parser.add_argument("run",
                    help='which run number this is (for file output naming)', type = int)
args = parser.parse_args()
f, total_time, gridx, gridy, gridz, eps, run = [args.f, args.total_time, args.gridx,
↪  args.gridy, args.gridz, args.eps, args.run]

def runOne():
    f = args.f
    total_time = args.total_time
    if (f == 'none'):
        f = ''
    if (f == 'restart'):
        f = 'FieldData_run' + str(run) + '.npz'
        npzfile = np.load(f)
        curr_time = npzfile['arr_11'][0]
        total_time = total_time - curr_time
    #Pass everything to the simulation
    simulation(total_time, gridx, gridy, gridz, eps, run, f)

def main():
    runOne()

if(__name__ == "__main__"):
    main()
```

A.4 Additional Data for Reference

In this appendix, we supply the data collected from simulations which were not chosen to appear in Section 4. These should be viewed as a reference, and further evidence that the trends identified in Section 5 are justified by sufficient data to draw conclusions from.

A.4.1 Preservation of Conserved Quantities

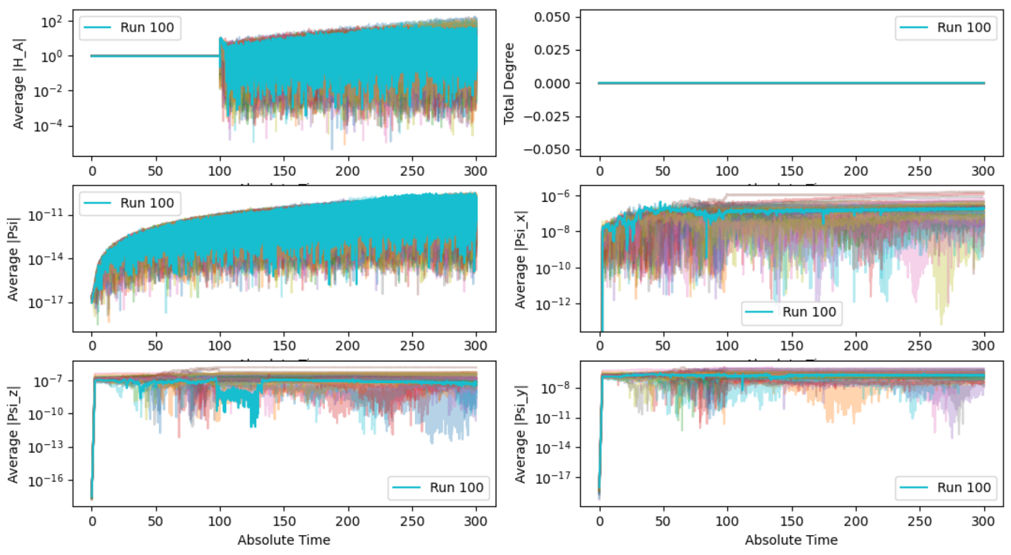

Figure A.1: Preservation of Conserved Quantities in Instances 51-100

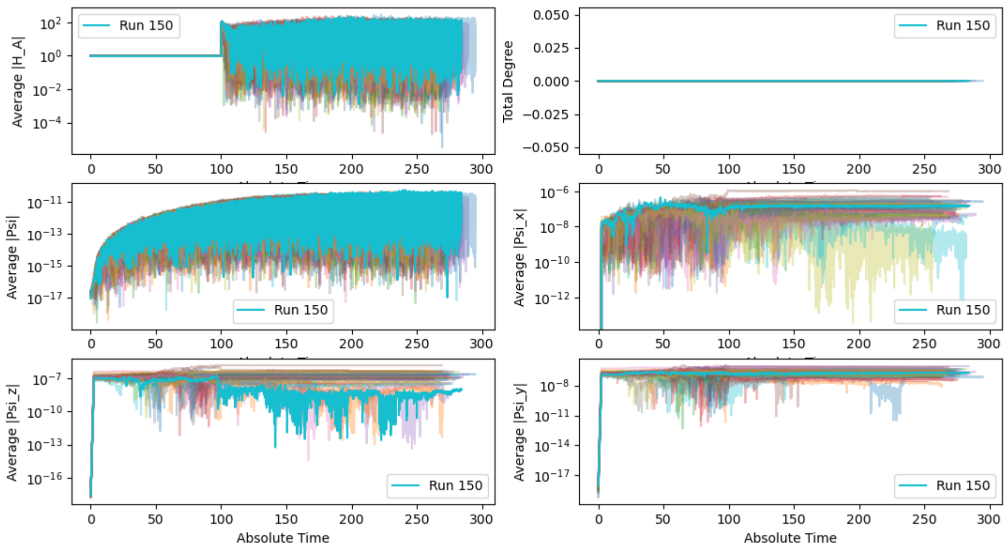

Figure A.2: Preservation of Conserved Quantities in Instances 101-150

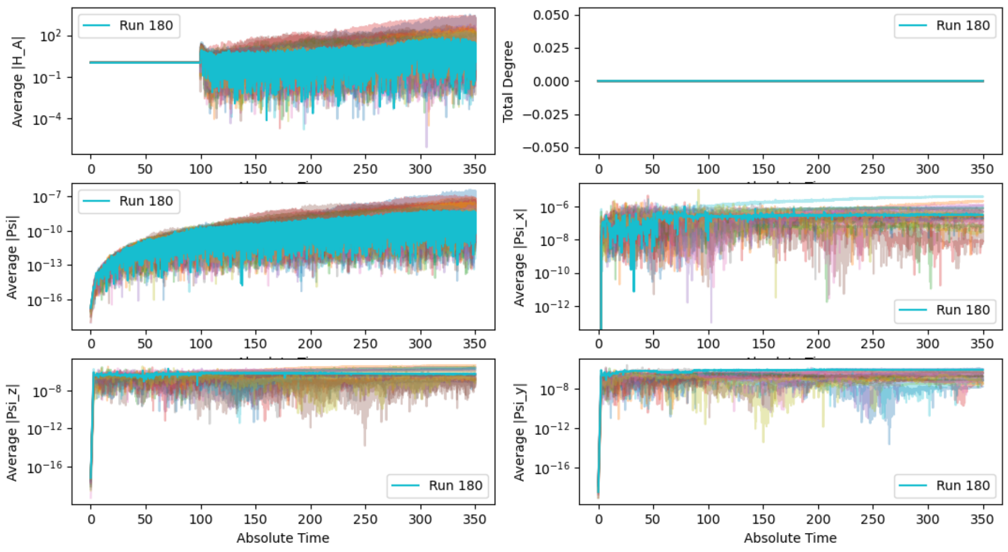

Figure A.3: Preservation of Conserved Quantities in Instances 151-180

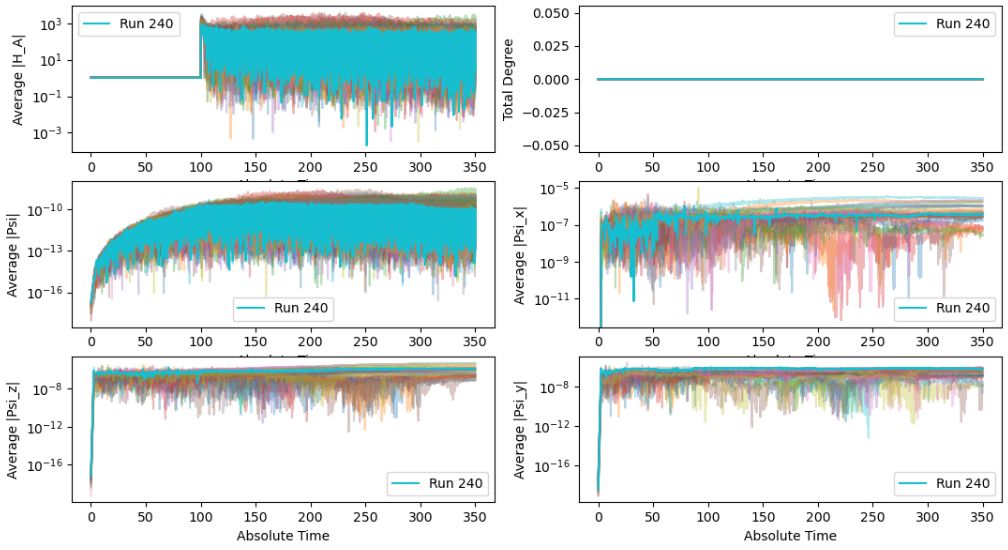

Figure A.4: Preservation of Conserved Quantities in Instances 211-240

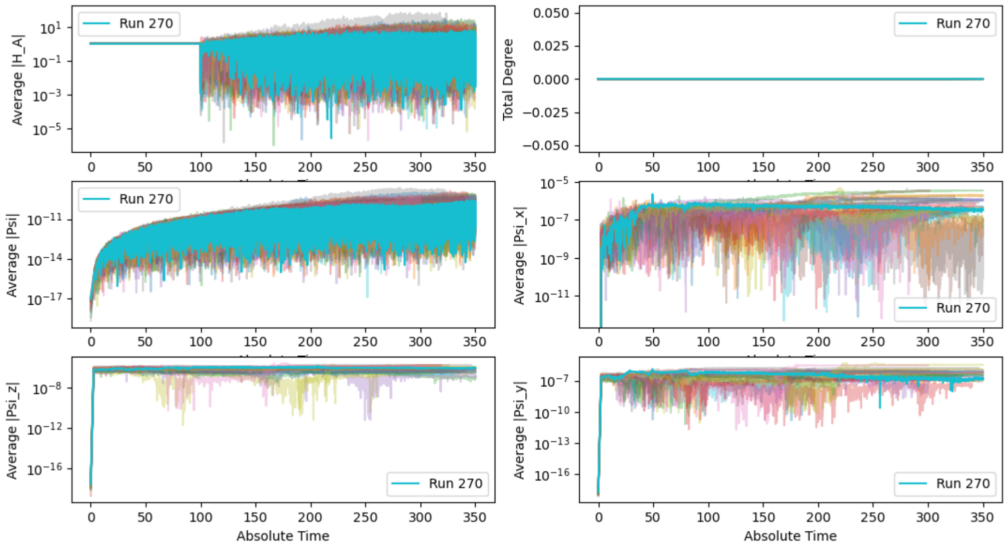

Figure A.5: Preservation of Conserved Quantities in Instances 241-270

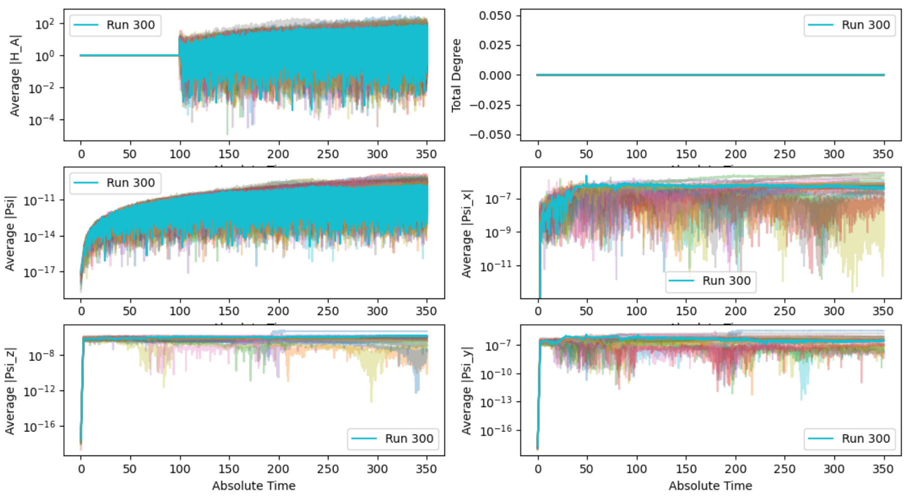

Figure A.6: Preservation of Conserved Quantities in Instances 271-300

A.4.2 Convergence of Field Values

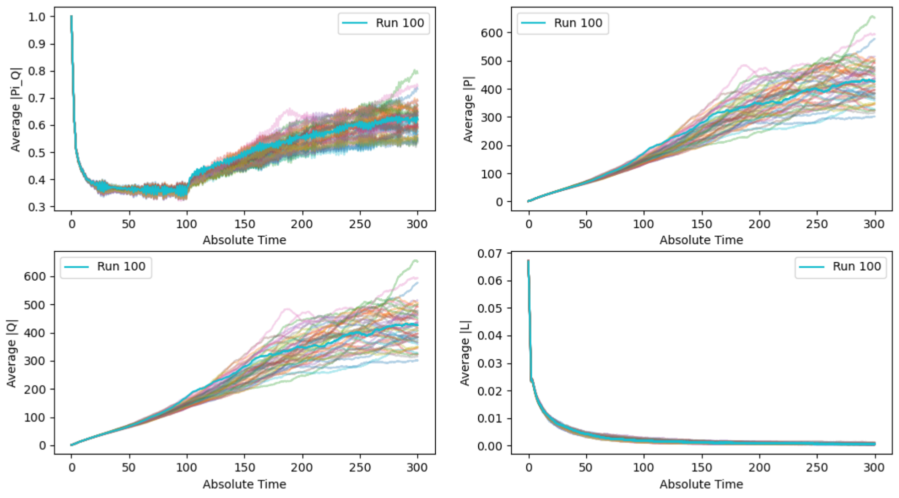

Figure A.7: Convergence of Field Values in Instances 51-100

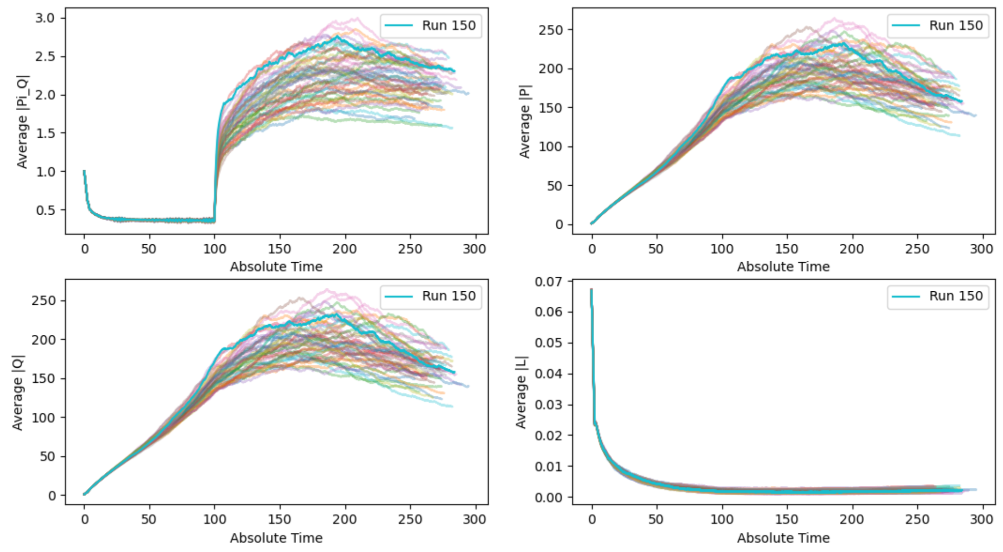

Figure A.8: Convergence of Field Values in Instances 101-150

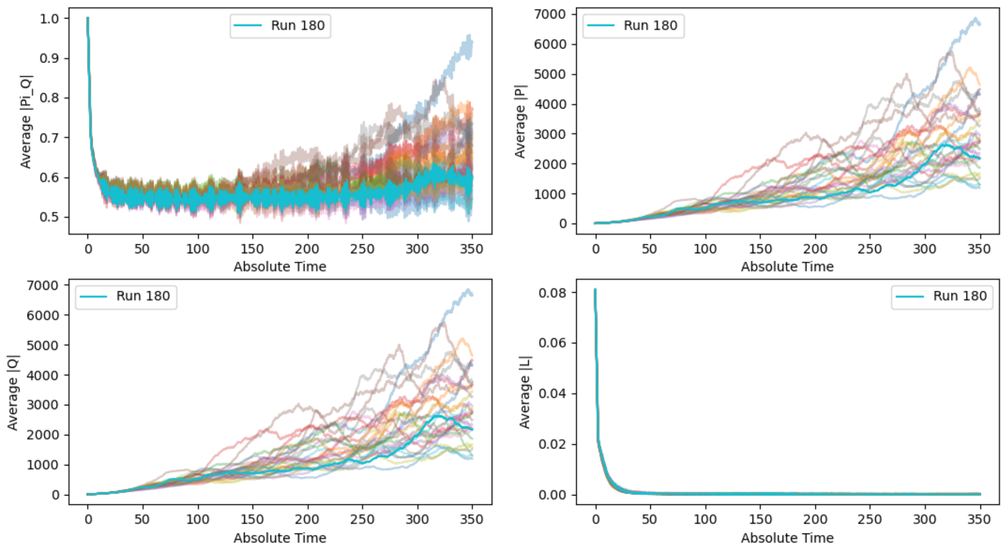

Figure A.9: Convergence of Field Values in Instances 151-180

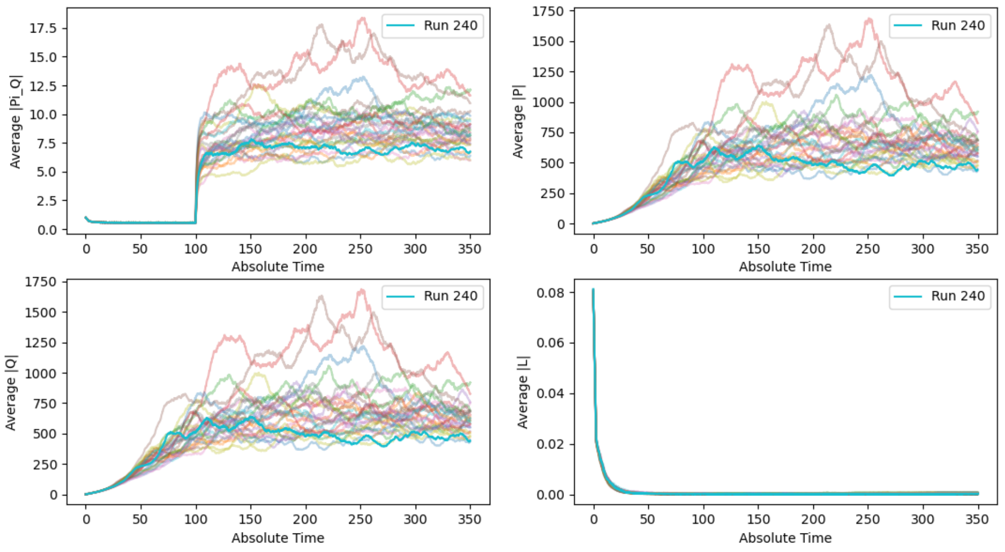

Figure A.10: Convergence of Field Values in Instances 211-240

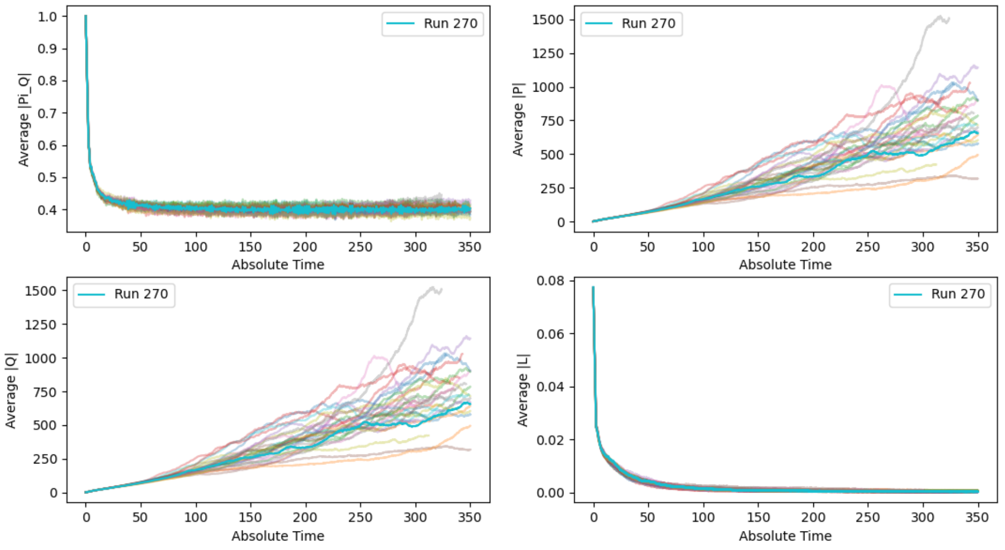

Figure A.11: Convergence of Field Values in Instances 241-270

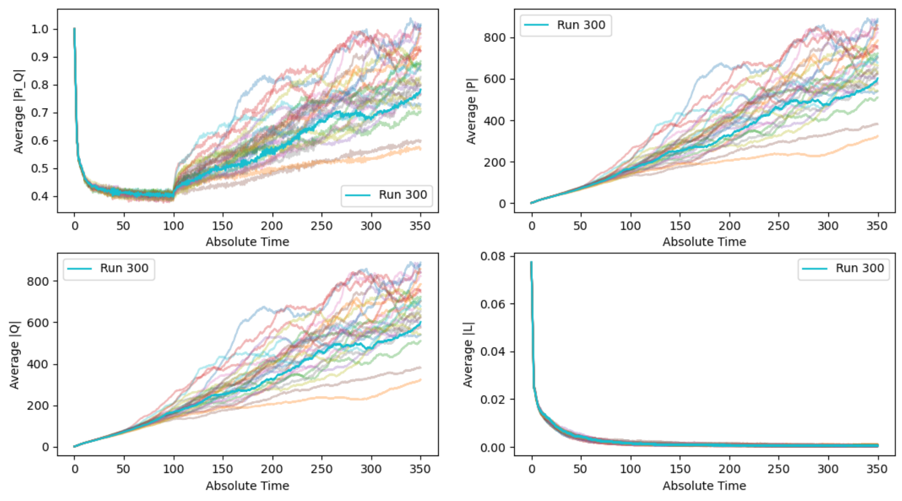

Figure A.12: Convergence of Field Values in Instances 271-300

A.4.3 Spatial Cross-Correlations

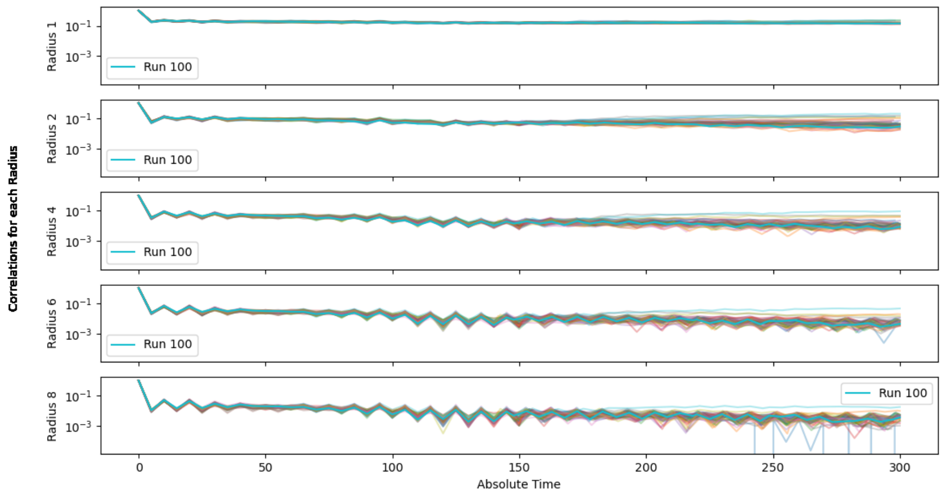

Figure A.13: Spatial cross-correlations for $[\vec{\Pi}_Q]_x$ over time in instances 51-100.

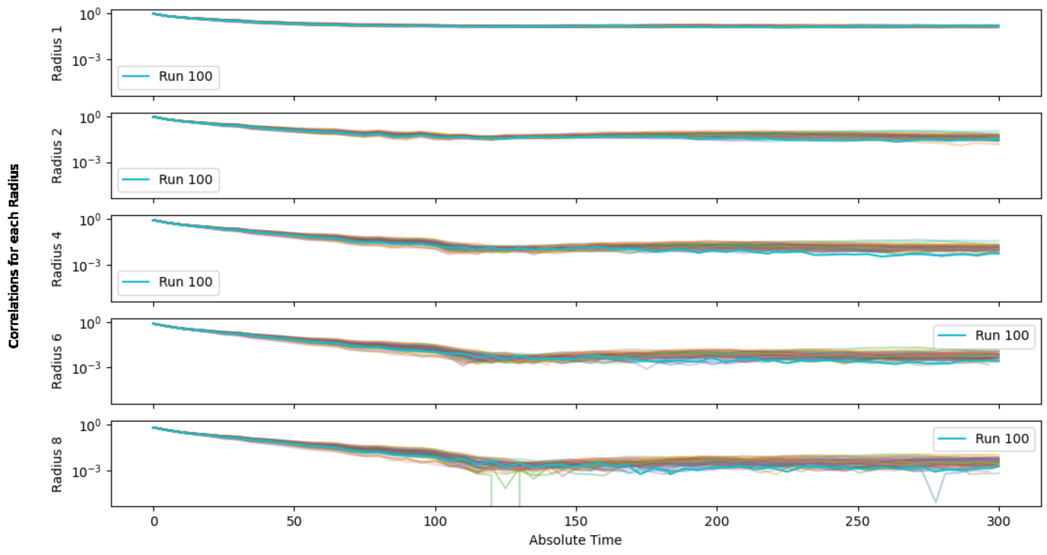

Figure A.14: Spatial cross-correlations for $[\vec{\Pi}_Q]_y$ over time in instances 51-100.

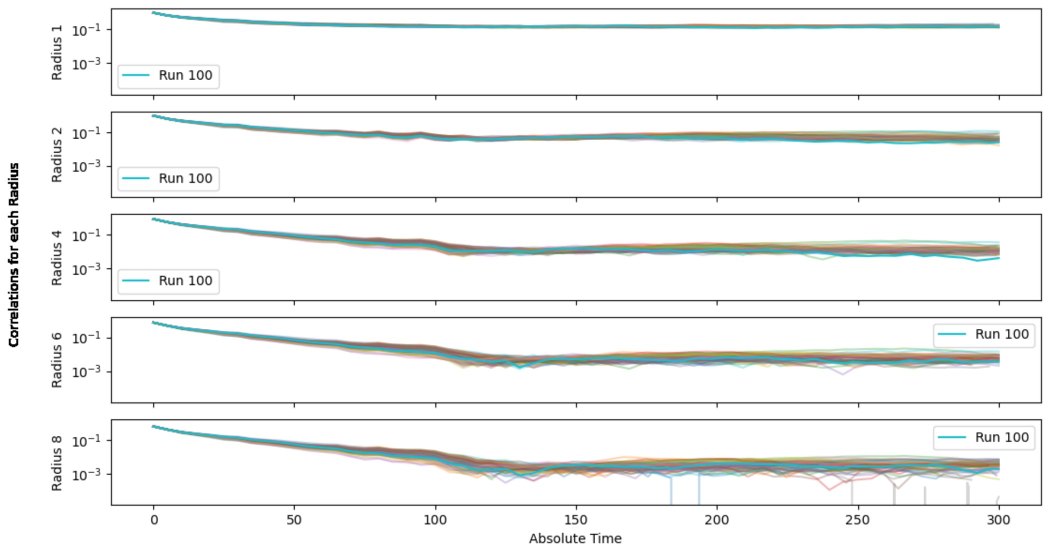

Figure A.15: Spatial cross-correlations for $[\vec{\Pi}_Q]_z$ over time in instances 51-100.

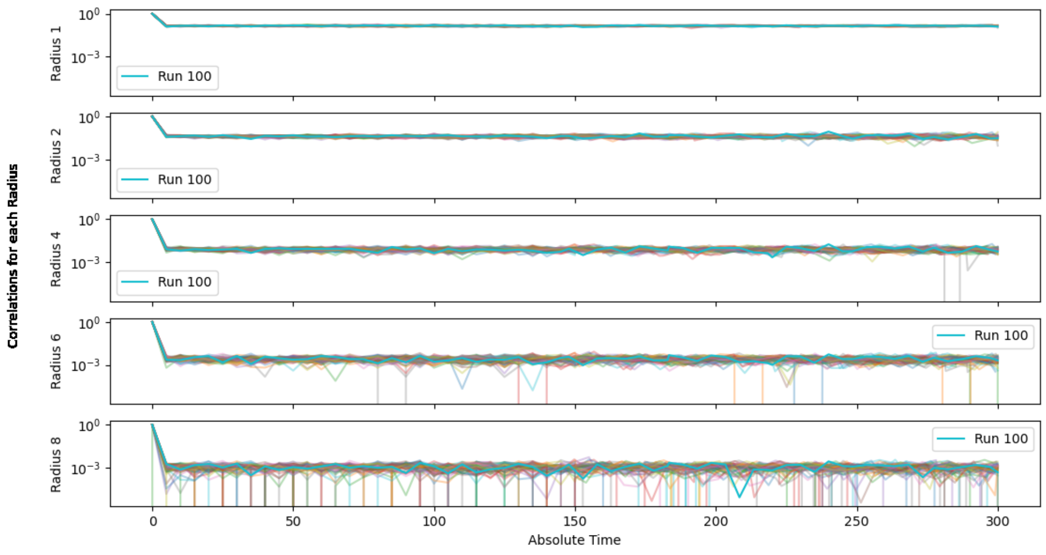

Figure A.16: Spatial cross-correlations for $[\vec{P}]_x$ over time in instances 51-100.

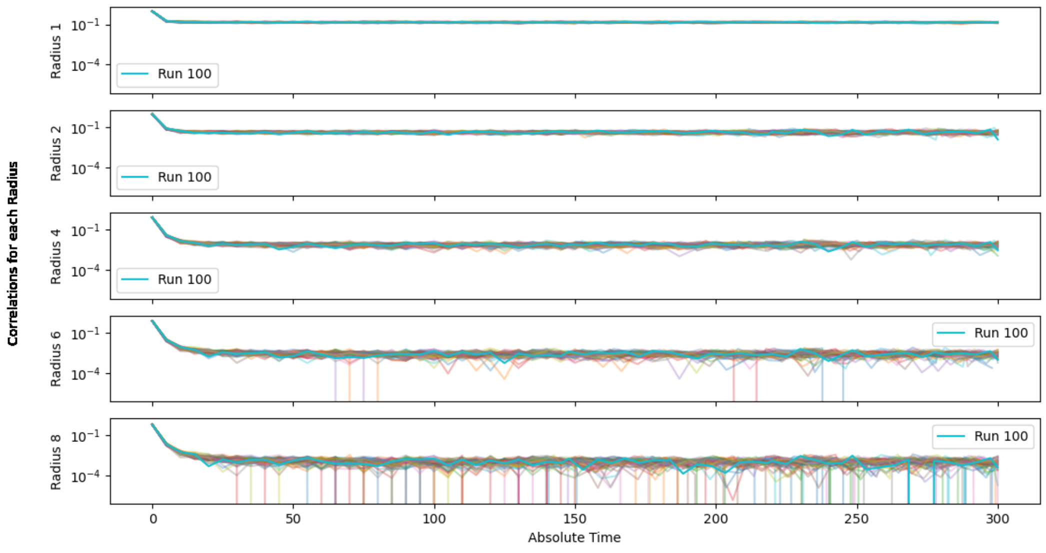

Figure A.17: Spatial cross-correlations for $[\vec{P}]_y$ over time in instances 51-100.

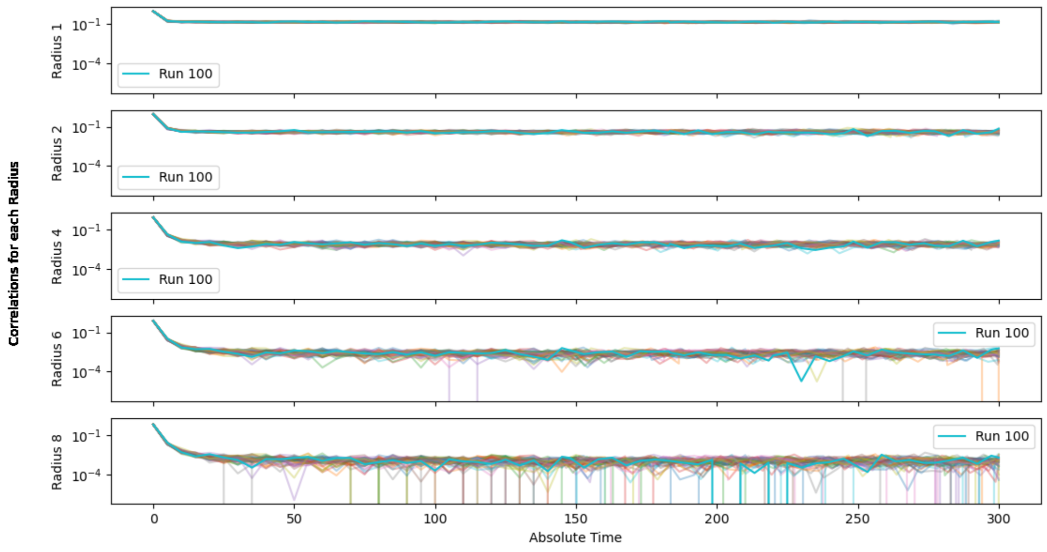

Figure A.18: Spatial cross-correlations for $[\vec{P}]_z$ over time in instances 51-100.

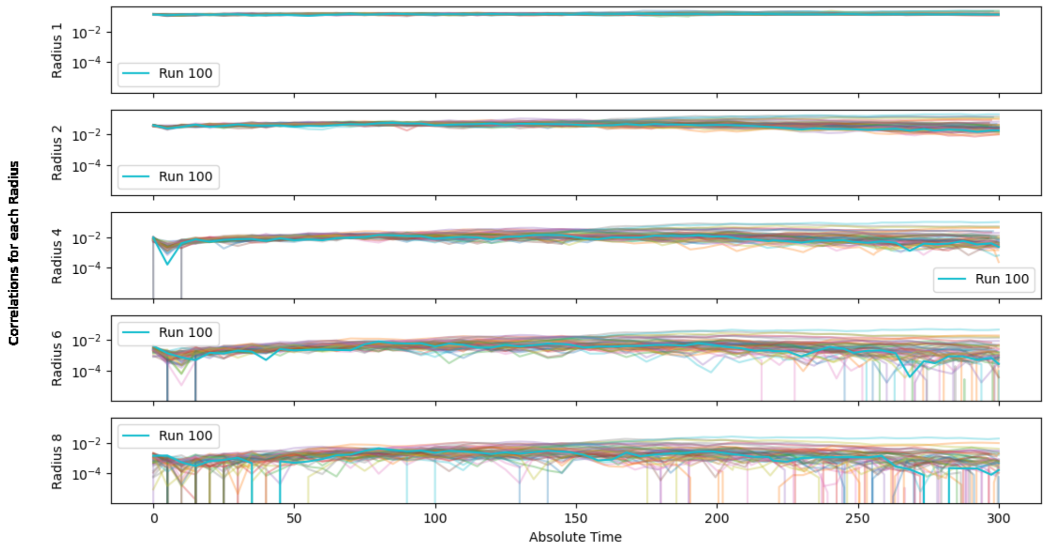

Figure A.19: Spatial cross-correlations for $[\vec{Q}]_x$ over time in instances 51-100.

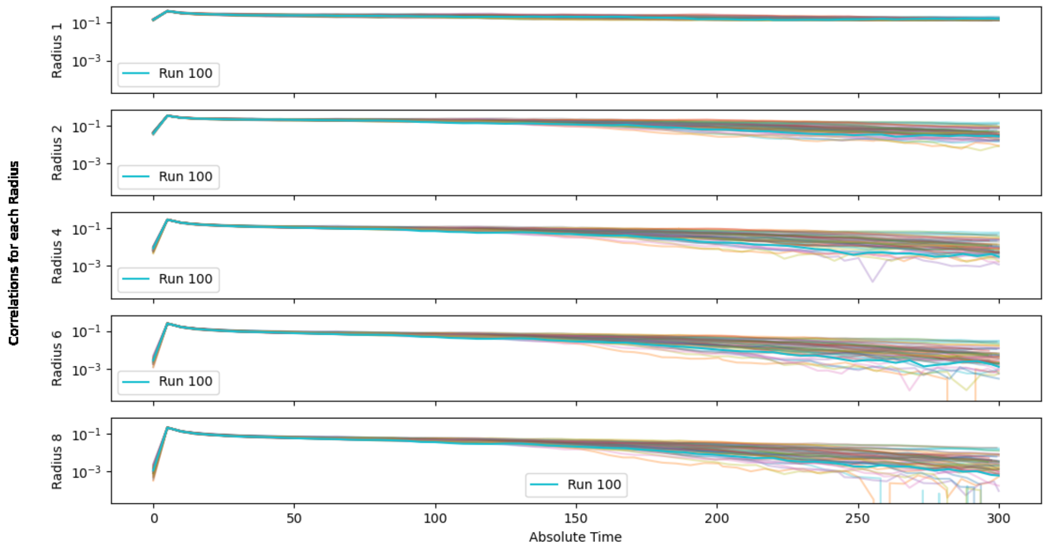

Figure A.20: Spatial cross-correlations for $[\vec{Q}]_y$ over time in instances 51-100.

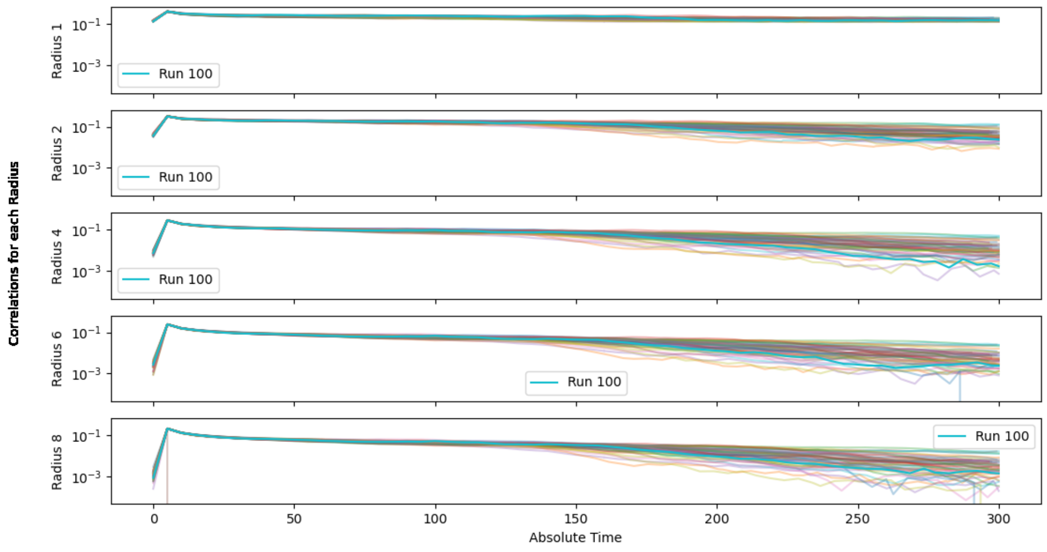

Figure A.21: Spatial cross-correlations for $[\vec{Q}]_z$ over time in instances 51-100.

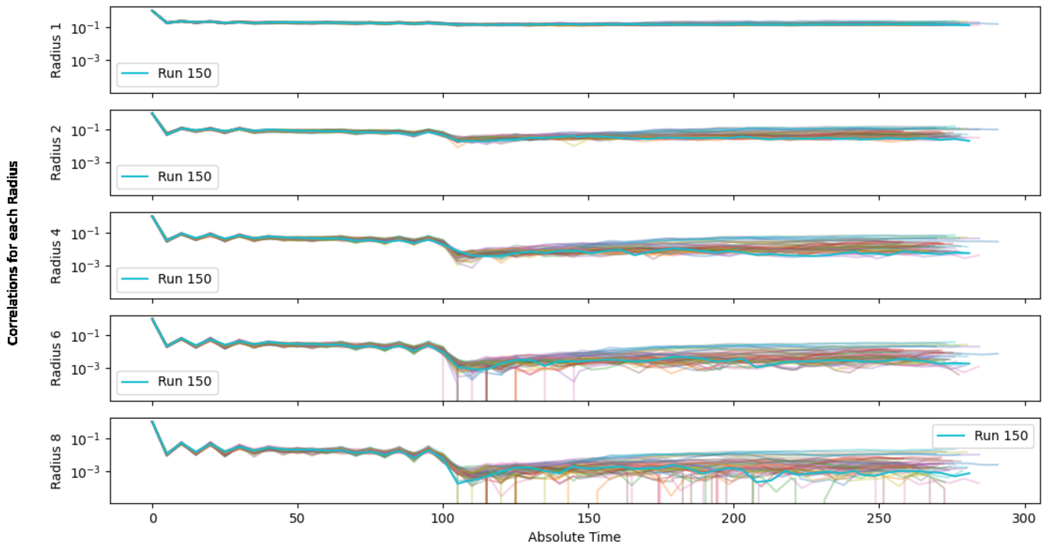

Figure A.22: Spatial cross-correlations for $[\vec{\Pi}_Q]_x$ over time in instances 101-150.

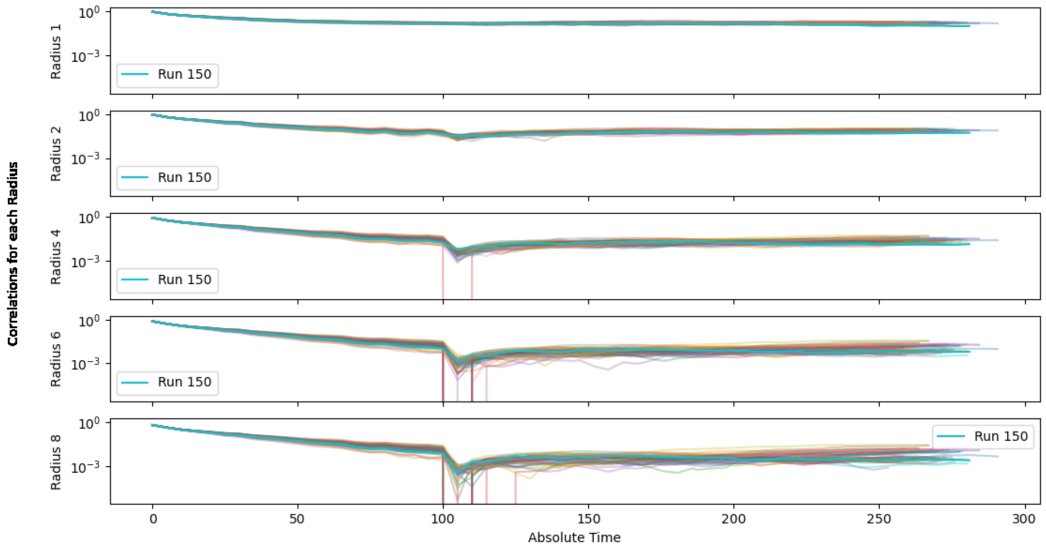

Figure A.23: Spatial cross-correlations for $[\vec{\Pi}_Q]_y$ over time in instances 101-150.

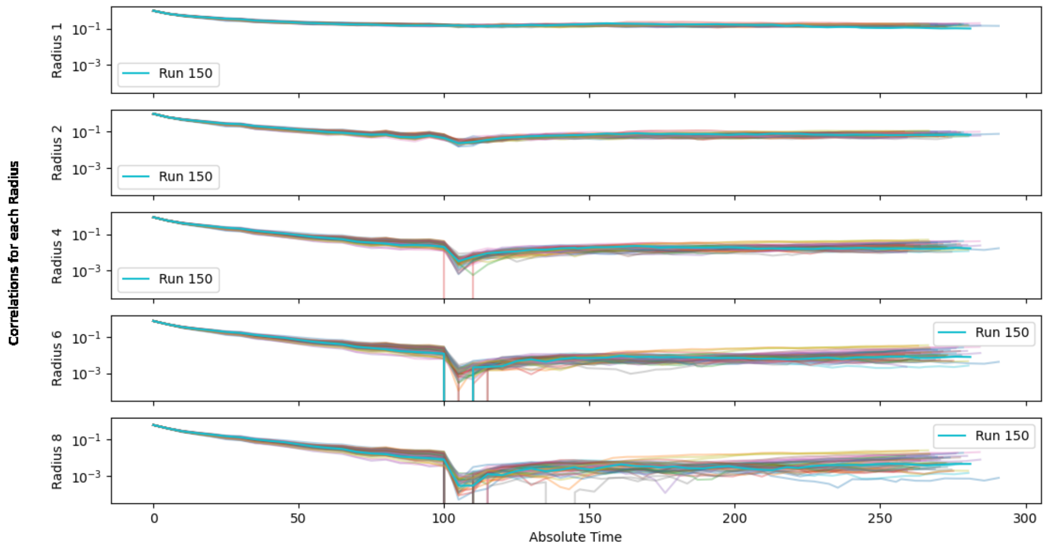

Figure A.24: Spatial cross-correlations for $[\vec{\Pi}_Q]_z$ over time in instances 101-150.

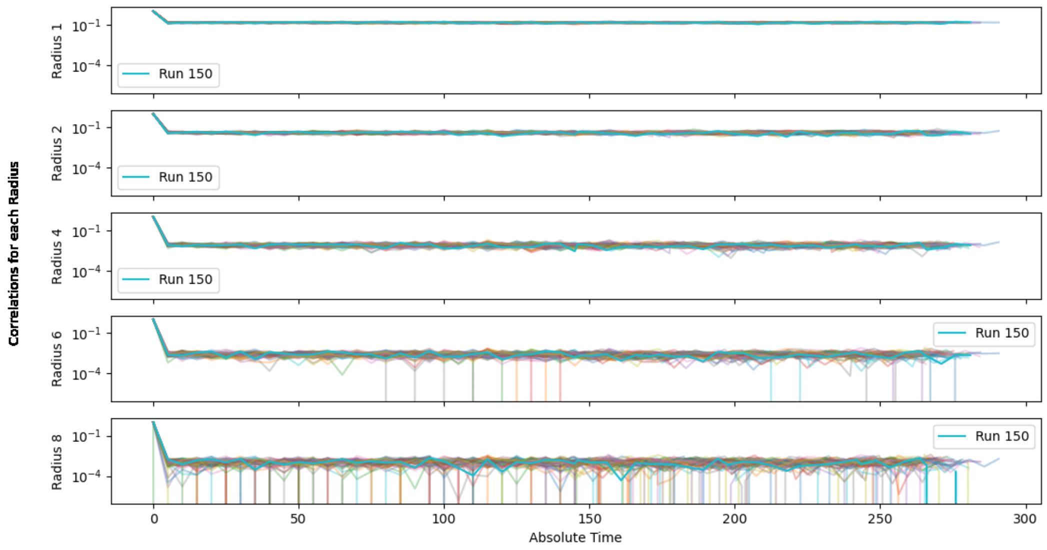

Figure A.25: Spatial cross-correlations for $[\vec{P}]_x$ over time in instances 101-150.

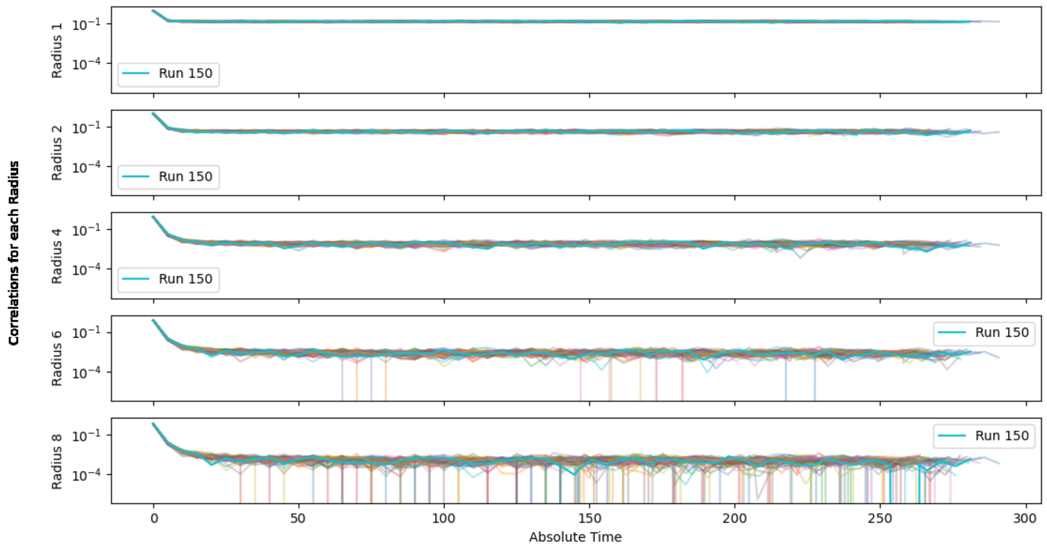

Figure A.26: Spatial cross-correlations for $[\vec{P}]_y$ over time in instances 101-150.

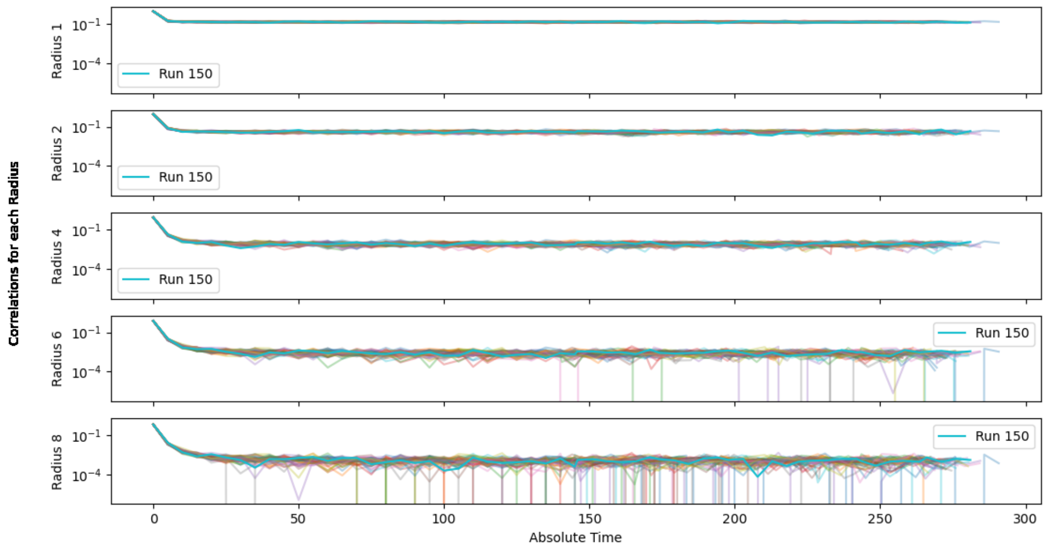

Figure A.27: Spatial cross-correlations for $[\vec{P}]_z$ over time in instances 101-150.

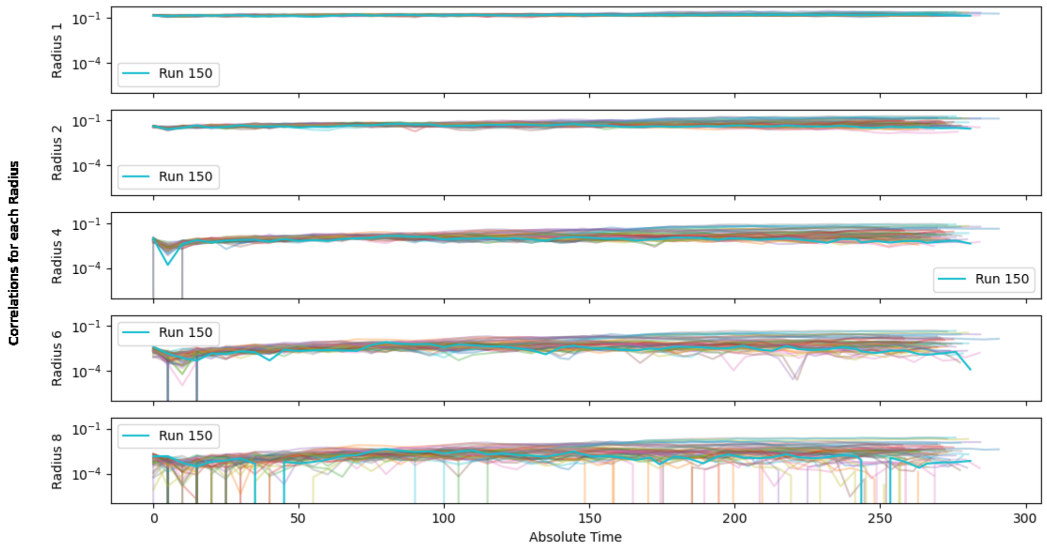

Figure A.28: Spatial cross-correlations for $[\vec{Q}]_x$ over time in instances 101-150.

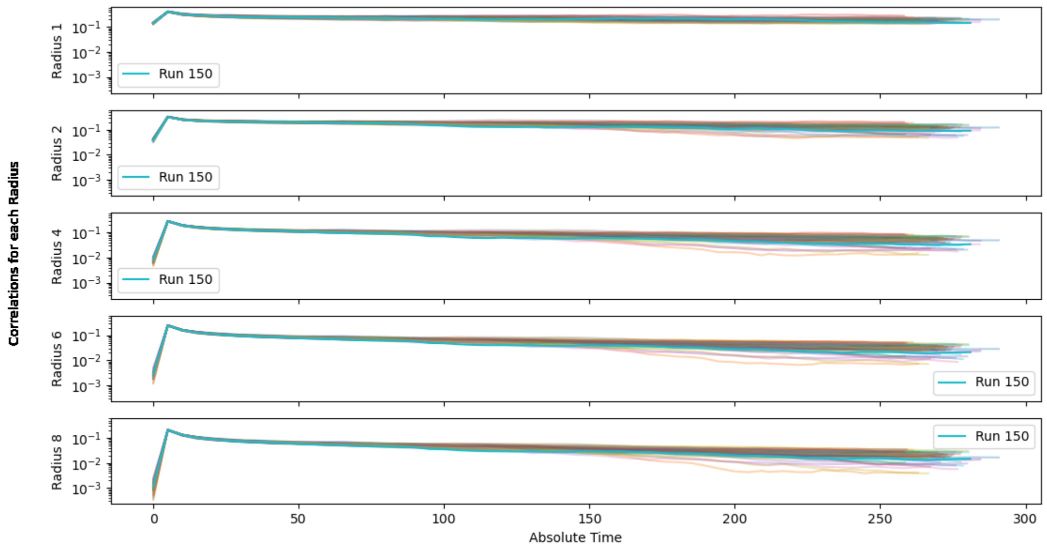

Figure A.29: Spatial cross-correlations for $[\vec{Q}]_y$ over time in instances 101-150.

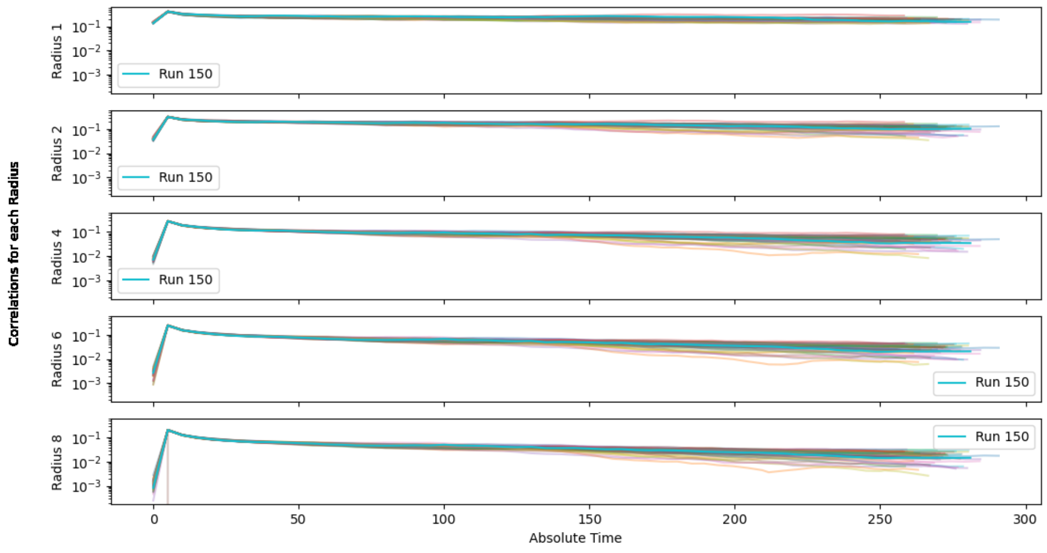

Figure A.30: Spatial cross-correlations for $[\vec{Q}]_z$ over time in instances 101-150.