Using numerics to explore the EPRL spinfoam amplitude

by

Courtney Charlotte Allen

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirements for the degree of

Master of Mathematics
in
Applied Mathematics

Waterloo, Ontario, Canada, 2020

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The use of numerical methods in loop quantum gravity is still relatively untapped. Numerical methods are, however, a useful tool, and offer access to regimes that are as of yet inaccessible by analytical methods. This thesis offers an example of this utility by presenting numerical methods for computing the vertex amplitude of a spinfoam model composed of cuboid intertwiners. The vertex amplitude has a known analytical solution in the semi-classical regime, but no known analytical solution in the purely quantum regime. These numerical methods, therefore, allow for an examination of the transition between the quantum and semi-classical regimes, and provide a foundation for future numerical methods in loop quantum gravity.

Acknowledgements

I would first like to thank Dr. Florian Girelli and Dr. Sebastian Steinhaus for supervising my work and offering their guidance and support. Their patience and understanding has been invaluable to me during my time at Waterloo. I would also like to thank Dr. Erik Schnetter for answering all of my many questions about optimizing code in Julia. And finally I would like to thank my mother, for always encouraging me and believing in me.

Contents

List of Figures

List of Tables

1 Introduction

Quantum mechanics and general relativity both offer useful descriptions of the universe, but they also have mathematical and conceptual physical differences. General relativity is formulated in terms of geometry, whereas quantum mechanics is formulated in terms of functional analysis. While quantum mechanics describes interactions on a fixed background spacetime, general relativity asserts that no such background spacetime exists, and that spacetime is instead dynamical itself. In particular, quantum mechanics necessitates an independent time variable, whereas, in general relativity, the notion of time is relative or relational, and the physical properties of the clock, such as its mass, might influence the spacetime geometry. These differences mean that there are two vastly different ways of describing the universe, depending on the regime being examined.

Many theories have been proposed to unify these two disparate paradigms, one of which is loop quantum gravity (LQG). LQG is based solely on the tenants of quantum mechanics and general relativity, and makes no further assumptions, such as the extra dimensions and supersymmetry required by string theory. It uses a straight-forward, nonperturbative, quantization of general relativity, imagining the spacetime field as loops similar to electromagnetic field lines [3]. The basic idea behind LQG will be discussed in section 2.1, and the more technical aspects of the formulation and discretization will be discussed in section 2.2.

While the analytical formulation of LQG is well understood, applying numerical techniques to LQG is a relatively new endeavour. The development and application of numerical methods to LQG is, however, of great importance. Numerics allow the exploration of regimes that analytical techniques alone cannot describe. For example, a numerical approach might be useful to tackle the difficult task of studying the nonperturbative dynamics of LQG and of implementing renormalization schemes.

Numerical methods can be used in tandem with analytical methods. In order to hone numerical tools, one can attempt to solve problems that have already been solved analytically. Prior knowledge of the solution places the focus on building an efficient code, and provides a guide line to ensure that the code does not get spurred by unwanted numerical artefacts. On the other hand, the numerical approach, while constrained in some regimes by analytical results, can still provide new, interesting, information on other

regimes that are difficult to probe at the analytical level. This thesis is concerned with one such case, computing the vertex amplitude of the ERPL spinfoam model with cuboid intertwiners, which is presented in section 2.3.

The semi-classical limit of the vertex amplitude for cuboid intertwiners is well known [4], determined using analytical methods, and is given in by equation (23). Computing the amplitude in the strictly quantum regime, however, is analytically complex, and the result is not known. By comparison, the computation is relatively straight-forward using numerical methods, as outlined in section 3. So, using the results of the numerical computation, presented in section 4, it is possible to analyze the transition from the quantum regime to the semi-classical regime, as is done in section 5. An overview of this work, and an outlook for the future is then given in section 6.

Numerical methods can therefore be used in tandem with analytical methods to explore new regimes of LQG that cannot be investigated with analytical methods alone. To that end, the numerical methods presented in this thesis demonstrate the utility that numerics offer, and lay the foundation for the development of future numerical methods in LQG. The results of this work, then, only offer a glimpse into the possible impact that numerics could have on the field of loop quantum gravity.

2 The Model

2.1 Loop Quantum Gravity, Intertwiners, and Spin-foams

Later sections will introduce the more technical aspects of LQG, but it is conceptually useful to briefly illustrate the LQG description of spacetime, beginning with three spatial dimensions and then introducing time. To that end, imagine a cube, A. This cube represents a region of space, a volume. It has six faces, each with a given area. Cube A can be connected to another cube, B, by gluing one of cube A's faces to a face of cube B. Connecting cubes describes larger and larger regions of space. Now, since volume and area are functions of the gravitational field, and the gravitational field is an operator, volume and area are both operators, with, as it turns out, discrete spectra [3], thus discretizing space.

Since area and volume have discrete spectra, we define quantum states of area and volume in terms of the associated quantum numbers. A cube can be associated to a node, and the face of a cube to a link. See figure 1 for reference. A set of glued cubes, then, gives rise to a graph where each node is labelled with a quantum number of volume, and each link is labelled with a quantum number of area. The quantum numbers of volume and area are referred to as intertwiners and spins, respectively [5]. The graph is then called a "spin-network", which labels the quantum states of space.

Of course, we need not divide space into cubes specifically, any type of 3D polyhedra will do, but using cubes relates directly to the cuboid intertwiner model, which is discussed in section 2.3, and on which the subsequent code is built. Different polyhedra have different numbers of faces, and therefore yield different intertwiners. Cubes yield six-valent intertwiners, octahedra would yield eight-valent intertwiners, and so on. Furthermore, the polyhedra also need not be all the same, cubes can connected to any combination of polyhedra. An octahedron can be attached to a tetrahedron, so long as their attached faces have the same area encoded in the associated spin. The idea of connecting these polyhedra is discussed in more detail in section 2.3.

The dynamics of the spin-network states Ψ are given by the Wheeler-DeWitt equation, which is a specialization of the Schrödinger equation to the gravitational case, and is given by

$$\hat{H}\Psi = 0, \tag{1}$$

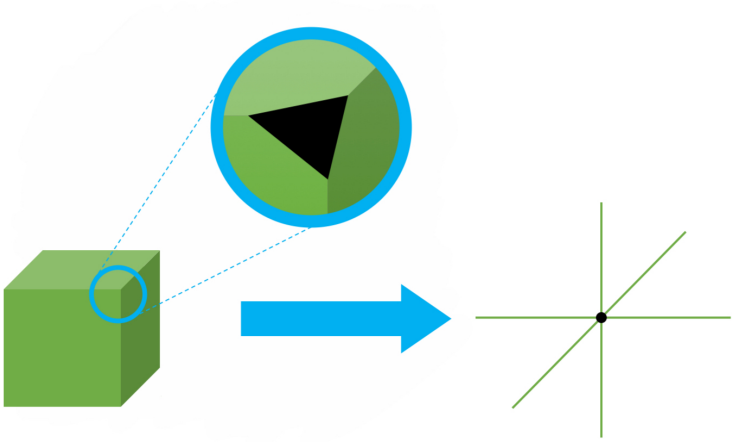

Figure 1: Visualizing a node with edges as a cube of space. Black represents the volume, or intertwiner, while green represents area, or spin. The blue circle is meant to show that the interior of the cube is black, and that it is only the faces that are green.

where $\hat{H}$ is the Hamiltonian, which we see is set to be zero, hence the name "Hamiltonian constraint". There is no time derivative since the time parameter used to formulate the general relativity framework is not physical. In the spin network basis, the Hamiltonian acts only on the nodes of the graph (intertwiners), so it follows that a loop without nodes would be a solution of equation (1). Historically, it is these simple loops that give loop quantum gravity its name [3].

To move from three dimensions to four, i.e. to describe how a quantum state of space can evolve to another quantum state of space, we extend the notion of a cube in three dimension to that of a hypercube in four dimensions. A hypercube can be represented in three dimensions by eight cuboids glued at their faces. This is analogous to unfolding a 3D cube into a 2D net. Each cube represents either an initial position, a space step, or a time step. See figure 2 for more information. So the cuboids show the possible paths of a three dimensional cube through time, or, in other words, they illustrate the dynamics of the cube. Extending this notion to that of spin networks, connecting networks forms a four dimensional picture of spacetime known as a spinfoam. The spinfoam gets its name from the fact that the connected spin networks form "bubbles" and look like a "foam", as illustrated by figure 3.

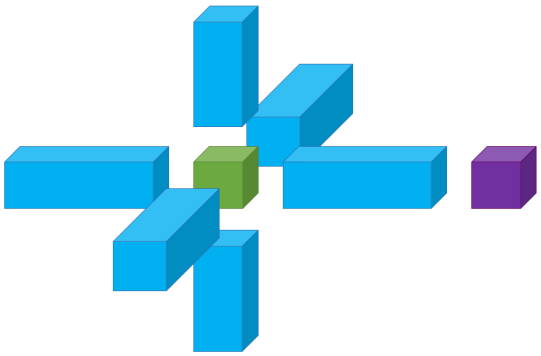

Figure 2: A three dimensional representation of a four dimensional hypercube. The green cube is the initial state, the blue cuboids represent possible space steps, and the purple cube represents a time step.

Section 2.2.3 formalizes the notion of a spinfoam, and section 2.3 formalizes the notion of a spinfoam defined by a hypercube. The spinfoam is one of the principle ways of describing the dynamics of LQG.

Loop quantum gravity, therefore, has a relatively simple foundation. It merely assumes that the fundamental principles of general relativity (background independence) and quantum mechanics (discretization of dynamical fields) are correct, and interpolates from there. The results are intertwiners and spins forming a spin network, which describe the quantum states of space, and spinfoams, which describe the evolution of these quantum states of space. The remainder of this section describes the relevant spinfoam model in greater detail and introduces the cuboid model that will be used to obtain numerical results.

2.2 Formulating LQG: BF theory, intertwiners, and spinfoams

2.2.1 BF Theory and the Equations of Motion

One method of describing the dynamics of LQG begins with a topological field theory known as BF theory, which derives its name from the convention of using a B field and the F curvature of a connection ω. The action of BF

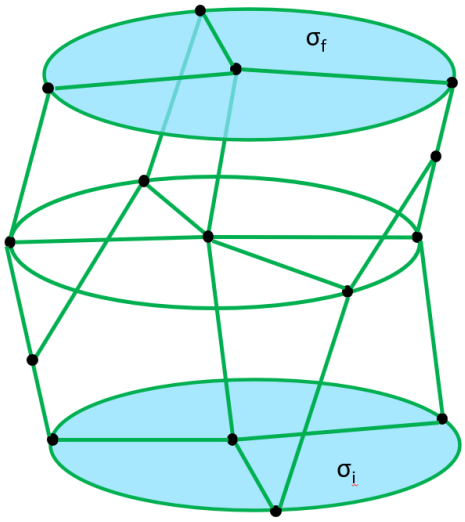

Figure 3: An illustration of the path from an initial spin-network, σ_i, to a final spin-network, σ_f given by linking spin-networks. This is called a spinfoam.

theory is

$$S = \int_M B_{IJ} \wedge F(\omega)^{IJ}, \tag{2}$$

where M is a 4 dimensional manifold, B^{IJ} is a 2-form valued in the Lie algebra SO(4), ω^{IJ} is the gauge connection, a 1-form valued in the Lie algebra SO((4), and $F(\omega)^{IJ}$ is the curvature of the connection ω^{IJ}.

In order to obtain the equations of motion, it is necessary that the action be stationary. Or, in other words, it is necessary that the variations of the action are zero

$$\delta_\omega S = 0\,, \quad \delta_B S = 0. \tag{3}$$

Performing the variations, we obtain the equations of motion

$$F(\omega) = 0\,, \quad d_\omega B = 0. \tag{4}$$

Counting the degrees of freedom shows that there are no local degrees of freedom [6], hence the theory is topological. The former equation states that the curvature of the connection vanishes, and therefore that the manifold is locally flat. This property is preserved when discretizing, as shown in section 2.2.3. Equations (2) and (4) describe the basic dynamics of BF theory.

2.2.2 The path integral

In the the path integral formalism, we first quantize BF theory. Recall that the partition function is the integral over all possible configurations, in this case of B and ω, weighted by the exponential e^{iS}, where S is the action. The partition function $Z(M)$ of the BF action is

$$Z(M) = \int \mathcal{D}\omega\, \mathcal{D}B\, \mathrm{e}^{iS[\omega,B]}, \tag{5}$$

where S is given by (2) and is seen as a function of the B field and the connection ω [7]. We can simplify the partition function formally by integrating out the B-field, which acts as Lagrange multiplier, enforcing the flatness of the connection given by (4) [6]. Heuristically, the result is an integral over the connections given by,

$$Z(M) = \int \mathcal{D}\omega\, \delta(F(\omega)). \tag{6}$$

The jump from equation (5) to equation (6) makes intuitive sense when considering the integral definition of the delta function,

$$\delta(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \mathrm{e}^{ikx}\, dk.$$

The partition function, and the subsequent integration, is, however, ill-defined. Since M is a continuous manifold, the measures $\mathcal{D}B$ and $\mathcal{D}\omega$ have no precise meaning as measures over field spaces. To make the integration over $\mathcal{D}B$ well-defined, and to regularize $\mathcal{D}\omega$, we can discretize the fields. In particular the discretization of the connection will be done in terms of holonomies [7], as outlined in the next section.

2.2.3 Triangulations and 2-complexes

One of the ways to formalize the above approach is to discretize the manifold, M of dimension $d \leq 4$, using a cellular decomposition such as a triangulation, Δ, which is a subdivison of M into d-cells. In the case of a triangulation, cells are called "simplices", and a 0-cell is a vertex, a 1-cell is a edge, a 2-cell is a triangle, a 3-cell is a tetrahedron, and, finally, a 4-cell is a 4-simplex, i.e. a 4d object made of five tetrahedra glued together. We note that higher dimensional simplices are made of lower dimensional ones. That is, a triangle

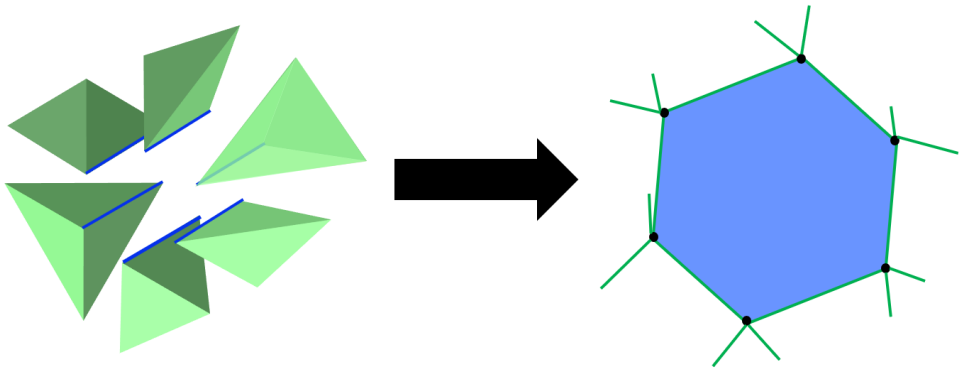

Figure 4: In the triangulation on the left, six tetrahedra are glued together by their faces and share a single edge, highlighted in blue. This edge is dual to the blue face of the 2-complex.

is made of edges, a tetrahedron is made of triangles, and so on. Instead of using a triangulation, we could use a cubulation, in which case the 2-cell is a square, the 3-cell is a cube and the 4-cell is a hypercube. In general, we can deal with cells of any arbitrary shape.

Of special importance is the dual 2-complex to the cellular decomposition. This dual 2-complex is specified by $\sigma = \{f, e, v\}$, where f is a face of the 2-complex, e is an edge, and v is a vertex. v is dual to a d-cell, e is dual to a $d-1$-cell, and f is dual to a $d-2$-cell. The above equations are all four dimensional, however, it is more straightforward to illustrate the triangulation and its dual 2-complex in the analogous three dimensional spacetime, so the following discussion will be in three dimensions.

In three dimensions, the triangulation of M subdivides the manifold into tetrahedra. Once again, this is a special feature of the triangulation, and other choices of cells yield different polyhedra. These tetrahedra are dual to the vertices of the 2-complex where each vertex is an intersection of four edges. Each edge of the 2-complex is dual to a face of the tetrahedron, $e^* = f_\Delta$, as in figure 1, and each face of the 2-complex is dual to an edge of the tetrahedron, $f^* = e_\Delta$. See figure 4 for more information.

When discussing the discretization of the theory, it is common to deal with both the dual 2-complex, and the triangulation itself. The discretized

B field is then given in terms of Lie algebra elements, B_{f^*}, assigned to the edges, $e_\Delta = f^*$, of the triangulation Δ [8],

$$B_{f^*} = \int_{f^* \in \Delta} B. \tag{7}$$

The information contained in the connection can be encoded in a parallel transport from one point to any other point along a path, which in the discrete case means parallel transport from one tetrahedron to another. When discretizing, ω is replaced by the associated holonomy along an edge. Since tetrahedra are connected by their faces, which are dual to the edges, e, of σ, to discretize the connection, a holonomy is assigned to each edge in σ. The notion of holonomy is encoded through the path-ordered exponentiation $\mathcal{P}\exp$. The discretized connection can then be given by the path ordered exponentiation along e,

$$g_e := \mathcal{P} \exp\left(-\int_{e\in\sigma} \omega\right). \tag{8}$$

where g_e is the discretized connection, a group element associated to the edges of the 2-complex [8]. We make a choice that all group elements are in SU(2), which is a choice frequently made in spinfoams, due to considerations inherent to LQG. Another possible choice is $SO(4)$, and a mapping between the two is established when implementing simplicity constraints, as will be discussed in section 2.3. The choice of the group SU(2) leads into the idea of SU(2) coherent intertwiners, which will be necessary when describing quantum polyhedra in section 2.3.

Now to discretize the curvature, $F(\omega)$. The edges, e, of σ that share a face, f, form a closed loop and can be denoted e_f. The discretized curvature, U_f, can then be written as the holonomy around the face, f, [8]

$$U_f := \prod_{e_f}^{\vec{}} g_e. \tag{9}$$

Rewriting the partition function (5) in terms of the discrete variables (7), (8), and (9), the path integral becomes

$$Z_\sigma = \int_{\mathrm{SU}(2)} \left(\prod_{e\in\sigma} dg_e\right) \int \left(\prod_{f\in\sigma} dB_{f^*}\; e^{iB_{f^*}U_f}\right),$$

where dg_e is the $SU(2)$ Haar measure. Integrating out the B field gives the discrete, and well-defined, version of equation (6) [7],

$$Z_\sigma = \int_{\mathrm{SU}(2)} \left(\prod_e dg_e \right) \left(\prod_f \delta \left(\vec{\prod_{e_f}} g_e \right) \right), \tag{10}$$

where $\delta(g)$ denotes the delta function on SU(2). It states that each holonomy, U_f, around the closed loop e_f must be equal to the identity, which corresponds to a locally flat geometry as in continuum BF theory. By definition, we note that $\delta(U_f)$ is invariant under gauge transformations, due to the cyclicity of the delta function.

Now it is possible to rewrite the above path integral using the intertwiners and spins introduced in section 2.1. Peter-Weyl's theorem implies in particular that any gauge invariant function can be written in terms of the trace of irreducible representations,

$$\delta(g) = \sum_j d_j \mathrm{Tr}\left[D^j(g) \right], \tag{11}$$

where $j \in \frac{\mathbb{N}}{2}$ labels the irreducible representations of SU(2), $d_j = 2j+1$ is the dimension of the representation vector space labelled by j denoted V_j, and D^j_{mn} is the Wigner-D matrix for the irreducible representation j. Returning to (10), a spin, $j_f \in \mathrm{SU}(2)$ is assigned to each face, f, of the 2-complex σ. This allows each delta function to be expanded in terms of (11). Performing this expansion and exploiting the identity

$$D^j(g_1 g_2) = D^j(g_1) D^j(g_2)$$

gives

$$Z_\sigma = \sum_{j_f} \int_{\mathrm{SU}(2)} \left(\prod_{e \in \sigma} dg_e \right) \left(\prod_{f \in \sigma} (2j_f + 1)\ \mathrm{Tr}\left[D^{j_f}(g_{e_1}) ... D^{j_f}(g_{e_N}) \right] \right).$$

Grouping the terms that belong to edge e results in the expression

$$Z_\sigma = \sum_{j_f} \left(\prod_{f \in \sigma} (2j_f + 1) \right) \left(\mathrm{Tr} \left[\int_\sigma \prod_{e \in \sigma} dg_e \bigotimes_{f \supset e} D^{j_f}(g_e) \right] \right), \tag{12}$$

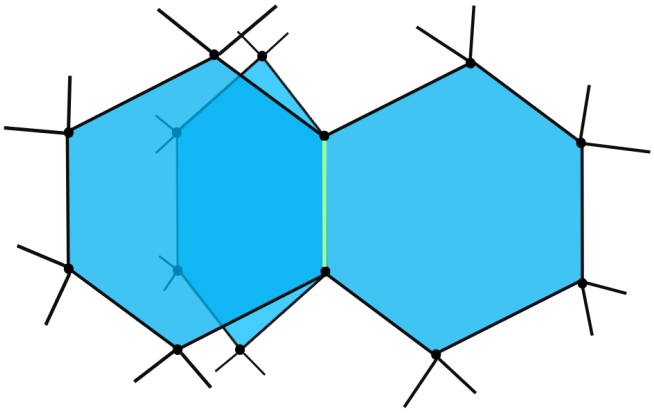

Figure 5: In three dimensions, the faces of the tetrahedron are attached to each other at their edges. The six edges are dual to the spins j_1, j_2, j_3, j_4, j_5, and j_6; and the four faces are dual to the intertwiners $\iota_{j_1 j_2 j_3}$, $\iota_{j_1 j_5 j_6}$, $\iota_{j_2 j_4 j_6}$, and $\iota_{j_3 j_4 j_5}$. In the dual 2-complex, four edges meet at a vertex, and each edge is associated to three faces of the tetrahedron. The is illustrated by the green edge.

where the term in square brackets is a projector onto the invariant subspace $\mathrm{Inv}(\bigotimes_f V_{j_f})$ where V_{j_f} is the representation vector space associated to face f. $\mathrm{Inv}(\bigotimes_f V_{j_f})$ is known as the Haar-projector, $\mathrm{P}_{\mathrm{Haar}}$ [8]. In a 3D triangulation, e is three valent, meaning that each edge is associated to three faces, as shown in figure 5. The Haar-projector is then given by

$$\mathrm{P}_{\mathrm{Haar}} = \int_\sigma dg\ D^{j_{f_1}}(g) \otimes D^{j_{f_2}}(g) \otimes D^{j_{f_3}}(g) = |\iota\rangle \langle\iota| \,, \tag{13}$$

where $|\iota\rangle \langle\iota|$ is a projector, and ι is an element of $\mathrm{Inv}(\bigotimes_f V_{j_f})$ and represents the intertwiners mentioned in section 2.1. In the case of a 3D triangulation, ι_e is a normalized unique vector in $\mathrm{Inv}(j_1 \otimes j_2 \otimes j_3)$. So ι_e is a three valent intertwiner and a projector onto $|\iota_e\rangle$, the half-edge of e.

Looking back at the triangulation, the tetrahedron has four faces and six edges. The dual 2-complex has four edges all connected at a central vertex,

v. From figure 5, the whole vertex amplitude can be given by

$$A_v = \iota_{j_1 j_2 j_3}\, \iota^{j_1}_{j_5 j_6}\, \iota^{j_2}_{j_4 j_6}\, \iota^{j_3 j_4 j_5}.$$

The spins are used to show the pattern of contraction of the entries in the associated representation vector spaces, sometimes called magnetic indices. In this way, intertwiners have an orientation, since spins go from one intertwiner to another, but this orientation has no physical meaning, it is only necessary for the contraction. Also, the spins must be obey the triangle inequalities, which will be revisited in section 2.3. In the dual 2-complex, the edges each correspond to a three valent intertwiner, as illustrated by figure 6. The trace over the Haar-projector can now be rewritten as a contraction over all of the edges e meeting at vertex v, and the partition function can be rewritten as a spinfoam [6]

$$Z_\sigma = \sum_{j_f} \prod_{f \subset \sigma} (2j_f + 1) \prod_{v \subset \sigma} A_v, \quad A_v = \{ \bigotimes_{e \in v} \iota_e \}. \tag{14}$$

The term $A_v = \{ \bigotimes_{e \in v} \iota_e \}$ is known as the vertex amplitude, and, in a three dimensional triangulation, is given by the 6-j symbol

$$\left\{ \begin{array}{c} j_1\ j_2\ j_3 \\ j_4\ j_5\ j_6 \end{array} \right\}.$$

The vertex amplitude is the contraction of all intertwiners sharing vertex v. In three dimensions this is trivial since the three-valent intertwiner is uniquely defined. The calculation of the vertex amplitude in four dimensions is the subject of this thesis.

This derivation has been done in three dimensions using a triangulation, but these two things were chosen for simplicity. In the remainder of the thesis, the spinfoam is four dimensional, not three, and the disrectization is a cubulation, not a triangulation. This means that the intertwiners are six valent, not three valent, and the spinfoam vertex is dual to a hypercube. This means that each edge of the dual 2-complex belongs to six faces, while eight edges meet at each vertex. So while the contraction pattern for the vertex amplitude is relatively easy to illustrate, as shown in figure 7, the actual 2-complex and its dual are more difficult to draw, since each edge of the 2-complex would be connected to six faces. Despite this simplification, the methods used in this section can be applied to more general discretization schemes and higher dimensions, or in other words, the actual procedure is analogous in all cases.

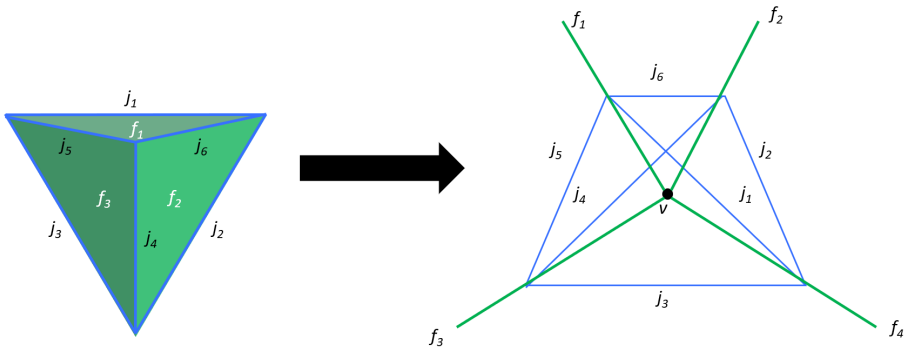

Figure 6: The above shows how the edges are contracted to find the vertex amplitude.

2.2.4 From BF-theory to LQG

We intend now to move from BF-theory, which is a topological theory with no-local degrees of freedom, to general relativity, which is a theory with local degrees of freedom. As previously mentioned, general relativity has two degrees of freedom per point in space time. To constrain the BF action (2) to the Palatini-Holst action of general relativity, we consider the constraint

$$C_{IJ} = B_{IJ} - \frac{1}{2}\epsilon_{IJKL}e^K \wedge e^L - \frac{1}{\gamma}e_I \wedge e_J = 0 \tag{15}$$

where e_I is the tetrad, a set of 1-forms from which the spacetime metric can be reconstructed, as

$$g_{\mu\nu} = \eta_{IJ}e^I_\mu e^J_\nu,$$

where η_{IJ} is the Minkowski metric, I and J are internal indices, and μ and ν are space-time indices. When plugging the constraint $B_{IJ} = \frac{1}{2}\epsilon_{IJKL}e^K \wedge e^L$ alone in the BF action, we recover the Palatini action,

$$S = \int_M \frac{1}{2}\epsilon_{IJKL}e^K \wedge e^L \wedge F^{IJ}, \tag{16}$$

which is equivalent to the Einstein-Hilbert action. The term $\frac{1}{\gamma}e^I \wedge e^J \wedge F^{IJ}$ is known as the Holst term. It does not affect the equations of motion,

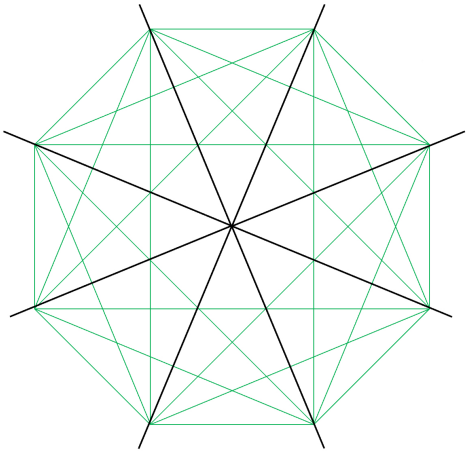

Figure 7: The above shows how the edges (green) are contracted to find the vertex amplitude in the case of six-valent intertwiners. The black lines mark opposite cubes inside the hypercuboid.

and is added to simplify the expression of the constraints to facilitate the quantization. The parameter γ controls the effect of the Holst term and is known as the Barbero-Immirzi parameter.

The Peblański action takes the BF action (2) as a starting point and implements the constraint (15) using the Lagrange multipliers [6],

$$S = \int_M B_{IJ} \wedge F(\omega)^{IJ} + \lambda_{IJ} C^{IJ}(B). \tag{17}$$

By varying with respect to λ one gets $C_{IJ}(B) = 0$ and this results on-shell to the same equations of motion as the ones coming from the Palatini-Holst action (i.e. the Palatini action with the Holst term). The Peblański action is the starting point of the discretization process to recover the spinfoam form of the partition function for four dimensional gravity.

2.3 Cuboid intertwiners

It is now necessary to introduce the cuboid model of intertwiners that will be the focus of the remainder of the thesis. The idea is to not consider all possible intertwiners, but to restrict ourselves to a special type that is

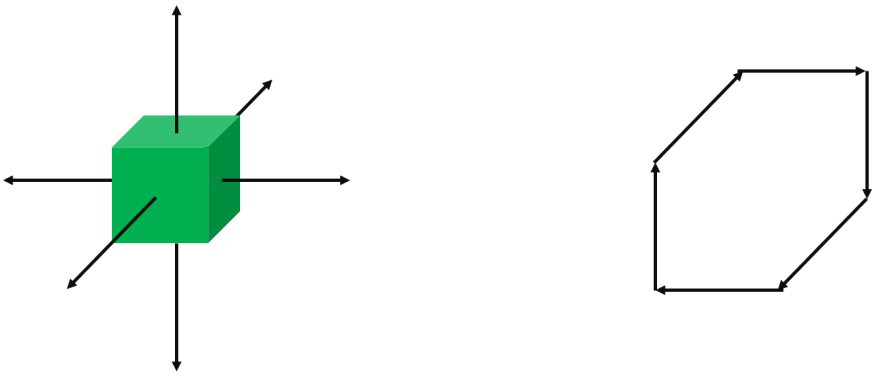

Figure 8: Illustration of the closure property of the normal vectors scaled by their associated areas in the case of a cube. The normal vectors to the cube on the left are arranged end to end on the right showing that their sum is zero.

sharply peaked on a cuboid geometry. The geometric interpretation of four-valent intertwiners as quantum tetrahedra was first introduced in [9] and generalized to quantum polyhedra with an arbitrary number of faces in [10]. The notion of a quantum polyhedron extends the classical definition of a convex polyhedron in three dimensional space to describe the intertwiners introduced in the previous section.

In three dimensional Euclidean space, the convex hull of a finite set of points is defined as a convex polyhedron [10]. Polyhedra with an immediate geometric interpretation are given by a set of $\mathcal{F}$ faces that have an area, $A_i \in \mathbb{R}$, a normal vector, $n_i \in S^2$, and satisfy the closure condition

$$C = \sum_{i=1}^{F} A_i n_i = 0. \tag{18}$$

This closure property is shown graphically in figure 8. The faces of two polyhedra can also be glued together when their area scaled normal vectors are equal, which leads into how intertwiners are connected by their spins.

When dealing with SU(2), the Livine-Speziale coherent intertwiners are designed to be peaked on classical configurations, and are given by the in-

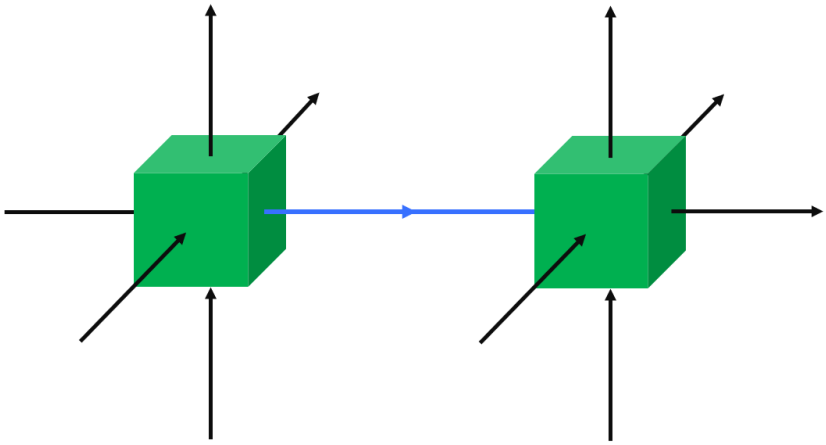

Figure 9: The blue arrow shows how the two intertwiners are contracted.

variant projection of a tensor product of states peaked on the direction of the spin

$$|\iota_{\{j_f\},\{\hat{n}_f\}}\rangle = \int_{\mathrm{SU}(2)} dg_e \bigotimes_{f\supset e} D^{j_f}(g_e)\,|j_f,\hat{n}_f\rangle\,, \tag{19}$$

where $\hat{n}$ is the direction of the spin and $\langle j,n|\vec{J}|j,n\rangle = j\hat{n}$ [10]. Since coherent intertwiners also obey a closure property,

$$\sum_i j_i n_i = 0,$$

they can be associated to classical polyhedra. The spin, j, is analogous to the area of a face of a polyhedron, and the direction of the spin, $\hat{n} \in S^2$, is analogous to the normal vector. Intertwiners are contracted along their spins similarly to how the faces of polyhedra can be glued together. See figure 9.

The spinfoam model used in this thesis is based on the Riemannian EPRL-FK model with Barbero-Immirizi parameter $\gamma < 1$. The ERPL-FK model imposes the Peblański constraints on spins and intertwiners. The full 4D model is defined for intertwiners in $\mathrm{SU}(2) \times \mathrm{SU}(2)$, but the model used in this thesis uses intertwiners in $\mathrm{SU}(2)$ for simplicity. Defining the boost map

Φ to map the SU(2)-intertwiners, ι_e, to the SU(2)$\times$SU(2)-intertwiners, by [2]

$$\Phi : \mathrm{Inv}_{\mathrm{SU}(2)} \bigotimes_f V_{j_f} \to \mathrm{Inv}_{Spin(4)} \bigotimes_f V_{j_f^+} \otimes V_{j_f^-},$$

means that the full vertex amplitude is given by [2]

$$A_v = \mathrm{tr}\left(\bigotimes_{e \supset v} \Phi(\iota_e) \right). \tag{20}$$

So, the partition function calculated in this thesis, relying solely on one copy of SU(2), is essentially only half of the partition function, but, for all intents and purposes, the partition function is analogous to the three dimensional partition function described in (14).

To constrain the spinfoam model to something that is relatively easy to compute, the coherent intertwiners in this thesis are peaked on cuboid geometry. Cuboids are hexahedrons which have internal angles of $\frac{\pi}{2}$, and as a consequence of this, opposing sides have the same area and their normals are opposite to each other. See figure 9 for more information. Extending this idea to intertwiners, cuboid intertwiners are six-valent. This means that in the 2-complex described in the previous section, each of the edges is attached to a face. Spins on opposite faces have the same value, and normal vectors are either mutually orthogonal to each other or anti-parallel. This means that, although the intertwiner is six-valent, it can be described by only three spins. So the state of a cuboid intertwiner, ι, is given by

$$|\iota_{j_1,j_2,j_3}\rangle = \int_{\mathrm{SU}(2)} dg \; g \triangleright \bigotimes_{i=1}^{3} |j_i, e_i\rangle \, |j_i, -e_i\rangle \,. \tag{21}$$

where the e_i are mutually orthogonal unit vectors in $\mathbb{R}^3$, the j_i are the three spins describing the cuboid, and g is the group element $g \in \mathrm{SU}(2)$ acting on the tensor product. Integrating over g ensures that the intertwiner is invariant, as discussed in section 2.2.3. In SU(2), this rotation is the same as having the Wigner matrix $D^j_{m'm}(\alpha, \beta, \gamma)$ act on the vectors $|j_i, \pm e_i\rangle$ and integrating over the angles α, β, and γ.

A four dimensional hypercube is bounded by eight cuboids, as shown in figures 2 and 10. This means that each vertex in the 2-complex is eight valent and that the vertex amplitude is given in terms of eight intertwiners and six

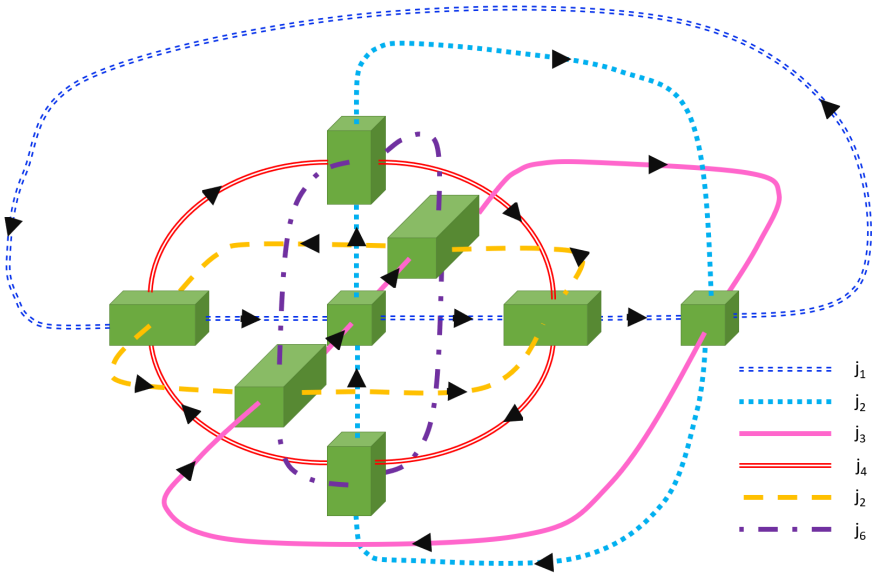

Figure 10: Illustrating how there are only six spins in a hypercuboidal lattice. The spins of opposing faces on a cuboid must match, and the spins of connecting faces must match. The arrows show the direction of the spins

spins, where the small number of spins comes from the fact that faces must have the same spin to be glued together, and the requirement that spins on oposite faces of the intertwiner must have the same spin. This is shown in figure 10.

2.4 The Semi-Classical Limit

The vertex amplitude of such a hypercuboid is only known in the asymptotic, or large-j, limit that is, it is only know in the case when all spins are very large. The vertex amplitude for cuboid intertwiners factorizes into two SU(2) vertex amplitudes, $\mathcal{A}_v = \mathcal{A}_v^+ \mathcal{A}_v^-$, and $\mathcal{A}_v^\pm$ is given by [4]

$$\mathcal{A}_v^\pm = \int_{\mathrm{SU}(2)^8} d^8 g_a \prod_{a\to b} \langle -n_{ab}| \, g_a^{-1} g_b \, |n_{ba}\rangle^{(1\pm\gamma)j_{ab}} \tag{22}$$

where the product ranges over links in the boundary graph oriented from cuboid a to cuboid b, as shown in figure 11. This contraction can be written

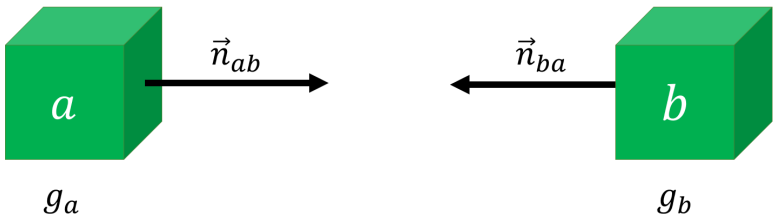

Figure 11: Two cuboid intertwiners, a and b, contracted along their faces by $\vec{n}_{ab}$ and $\vec{n}_{ba}$

algebraically as

$$\langle j_{ab}, -\vec{n}_{ab} \mid g_a^{-1} g_b \mid j_{ba}, \vec{n}_{ba} \rangle = \langle -\vec{n}_{ab} \mid g_a^{-1} g_b \mid \vec{n}_{ba} \rangle^{2j_{ab}} ,$$

where $j_{ab} = j_{ba}$ is the spin of the glued faces, and the left hand side of the equation is written in terms of the coherent states. The factor of $(1 \pm \gamma)$ in the power in equation (22) comes from the simplicity constraints imposed on the spins.

In [2], (22) is evaluated in the large j limit via a stationary phase approximation of the 21-dimensional integral over seven copies of SU(2). The formula is reported in [4] as

$$\mathcal{A}_v^{\pm} = \left(\frac{1 \pm \gamma}{2} \right)^{\frac{21}{2}} \mathcal{B}_v \tag{23}$$

with

$$\mathcal{B}_v(j_1, ..., j_6) = \frac{1}{\sqrt{-\det H}} + \text{c.c.} \tag{24}$$

where H is the Hessian matrix of the action S and is given by

$$\begin{aligned}\det H = (((&2(j_1^2(j_2+j_4)+j_2j_4(j_2+j_4)+j_1(j_2^2(1+I)j_2j_4+j_4^2)))\\ &(j_1^2(j_3+j_5)+j_3j_5(j_3+j_5)+j_1(j_3^2+(1+I)j_3j_5+j_5^2))\\ &(j_3j_4j_5+j_2(j_4j_5+j_3(j_4+j_5)))(j_2^2(j_3+j_6)+\\ &j_3j_6(j_3+j_6)+j_2(j_3^2+(1+I)j_3j_6+j_6^2))(j_4^2(j_5+j_6)\\ &+j_5j_6(j_5+j_6)+j_4(j_5^2+(1+I)j_5j_6+j_6^2))(j_3j_4j_6+j_1\\ &(j_4j_6+j_3(j_4+j_6)))j_2j_5j_6+j_1(j_5j_6+j_2(j_5+j_6))).\end{aligned} \tag{25}$$

Equation (23) is not, however, what is being calculated in this thesis. Due to the choice of convention for Euler angles discussed in 3.1.1, there is an additional factor of $\frac{1}{8\pi^2}$ in the measure for each of the seven group integrations, a factor of 2^7 due to the discrete symmetries of the critical and stationary points [2, 4] and a factor of $(2\pi)^{\frac{21}{2}}$ from the stationary phase approximation of the 21-dimensional integral, which results in a factor of

$$\left(\frac{2}{8\pi^2}\right)^7 (2\pi)^{\frac{21}{2}} = \frac{\sqrt{2}}{16\pi^{\frac{7}{2}}}$$

in the final function. So the code discussed in the next section is expected to conform to,

$$\mathcal{B}_v(j_1,...,j_6) = \frac{\sqrt{2}}{16\sqrt{-\pi^7 \det H}} + \text{c.c.} \tag{26}$$

in the large j limit.

The strategy to the large j vertex amplitude put forth by equation (22) is to first contract the intertwiners and then integrating via the stationary phase approximation. This was also the strategy used by Klöser in his masters thesis [11], where he attempted to compute the group integrations numerically. However this method proved to not be numerically convergent, even for small spins, because equation (22) is a 21 dimensional, highly oscillatory, integral. The opposite strategy, to first compute the integration on the individual intertwiners and then to contract them is used this thesis, and discussed in section 3.

3 Numerical Work

The code to compute the spinfoam vertex amplitude is adapted from code written by Sebastian Steinhaus to compute the amplitude in the case that all spins are ($j = \frac{1}{2}$). Steinhaus' code was written in Mathematica, but the code was translated to Julia to compute the vertex amplitudes for general configurations, in particular for $j > \frac{1}{2}$. Julia was chosen as the coding language for ease of use and efficiency. For instance, on a consumer laptop, in the ($j = \frac{1}{2}$) case without any optimizations, the contraction took 6 minutes and 47 seconds to compute in Mathematica, while it took 4 minutes and 34 seconds to compute in Julia.

The code to compute the spinfoam amplitude is split into two parts. The first part computes the eight intertwiners as arrays, and saves them to a text file, as discussed in section 3.3. The next part converts the text file to an array in Julia and then contracts the intertwiners, thus computing the spinfoam amplitude. The first part of the code is discussed in sections 3.1.1-3.1.3, and the second part is discussed in section 3.1.4. The code is split in this way to reduce the number of times an intertwiner needs to be computed, since computing intertwiners is computationally expensive, the ability to save them for later use reduces computation time. Creating intertwiners is so computationally expensive because, for each of the components, which grow in number as we increase the spins j, we need to integrate over a product of Wigner matrix elements. These integrals contain powers of sin and cos and are thus highly oscillatory, as will be discussed in subsection 3.1.3. Section 3.1 discusses the basic structure of the code. The optimizations made to this basic structure are then discussed in section 3.2, and the use of high-performance computing (HPC), on the Perimeter Institute's HPC, Symmetry, is discussed in section 3.4.

3.1 The Code

The code for the intertwiner has three main functions, one to create a Wigner matrix, called `G()`; one to create a tensor that represents the integrand of the intertwiner at a specific global rotation angle, called `Tensor()`; and one that integrates the tensor, thus performing the group averaging over SU(2), and outputs the desired intertwiner, called `Intertwiner()`.

3.1.1 The Wigner Matrix Function: `G()`

`G()` computes the Wigner matrix using the equation [12]

$$D^j_{m'm}(\alpha,\beta,\gamma) = e^{-im'\alpha} d^j_{m'm}(\beta) e^{-im\gamma}, \tag{27}$$

where α, β, and γ are the Euler angles and $d^j_{m'm}$ is the Wigner small d-matrix given by

$$\begin{aligned} d^j_{m'm} = {} & [(j+m')!(j-m')!(j+m)!(j-m)!]^{\frac{1}{2}} \\ & \sum_s \left[\frac{(-1)^{m'-m+s} \left(\cos\frac{\beta}{2}\right)^{2j+m-m'-2s} \left(\sin\frac{\beta}{2}\right)^{m'-m+2s}}{(j+m-s)!s!(m'-m+s)!(j-m'-s)!} \right], \end{aligned} \tag{28}$$

where the sum runs over all integers, s, such that the factorials are non-negative.

The function `G()` takes five arguments: `alpha`, `beta`, `gamma`, `J`, `MM`, and `DD`. The inputs `alpha`, `beta`, and `gamma` are the Euler angles, $\alpha \in [-\pi, \pi]$, $\beta \in [-\frac{\pi}{2}, \frac{\pi}{2}]$, and $\gamma \in [-\pi, \pi]$. The z-x-z convention for Euler angles, illustrated in figure 12, is chosen [12]. `J` is an integer that describes the spin j, the two are related by by $j = \frac{\mathtt{J}}{2}$. `MM` is a matrix that contains the magnetic indices, m and m', for each entry in the Wigner matrix, and is given by $\mathtt{MM}^j_{mm'} = (m, m')$. So, for instance, in the $j = \frac{1}{2}$ case,

$$\mathtt{MM}^{\frac{1}{2}} = \begin{bmatrix} \left(\frac{1}{2},\frac{1}{2}\right) & \left(\frac{1}{2},\frac{-1}{2}\right) \\ \left(\frac{-1}{2},\frac{1}{2}\right) & \left(\frac{-1}{2},\frac{-1}{2}\right) \end{bmatrix}.$$

Finally, `DD` is the term $[(j+m')!(j-m')!(j+m)!(j-m)!]$ in the Wigner small d-matrix. It is necessary to convert the inputs of the factorial terms to type `BigInt`, otherwise the output for large spins would be zero[1]. The optimized code for the `G()` function is given in C.1.

3.1.2 Tensor Function

This code is designed to model cuboid intertwiners, which are defined to have three in-going faces, and three out-going faces, as discussed in section 2.3. We can define an $x-y-z$ axis that points in the direction of the faces of the cuboid, as shown in figure 13. If we define the outgoing faces to

[1] The mazimum integer in `Int64` is 9 223 372 036 854 775 807, or approximately 9×10^{18}.

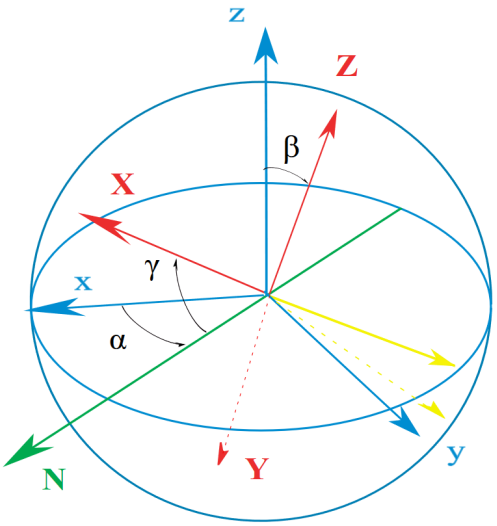

Figure 12: Illustration of z-x-z Euler angle. Beginning on the blue axis, α is a rotation about the z-axis, β is a rotation about the new N-axis (in green), and γ is a rotation about the new Z-axis (in red). Source: [1]

be kets and the ingoing faces to be bras, then in figure 14, intertwiner 1 has orientation $(|x\rangle, |y\rangle, |z\rangle, \langle -x|, \langle -y|, \langle -z|)$, and intertwiner 7 has orientation $(|x\rangle, \langle y|, |z\rangle, \langle -x|, |-y\rangle, \langle -z|)$.

Cuboid intertwiners can be modelled by six dimensional arrays, $I^{m_4',m_5',m_6'}_{m_1,m_2,m_3}$, where each magnetic index m_1, m_2, ... corresponds to a face of the cuboid. The first step in creating an intertwiner is creating this array. The function `Tensor()` creates the integrand of the intertwiner with a specific rotation angle. It takes the arguments `alpha`, `beta`, `gamma`, `v`, `J`, `MM`, and `DD`, and outputs a six dimensional array. `v` is a 3x1 array of strings that indicates the orientation of the intertwiner. The first element in the array gives the x-direction, the second gives the y-direction, and the third gives the z-direction, while the remaining three components refer to the opposite faces and are thus fixed to be antiparallel. The possible input strings are `"x"`, `"minusx"`, `"y"`, `"minusy"`, `"z"`, and `"minusz"`. So a possible input is

$$\texttt{v} = \begin{bmatrix} \texttt{"x"} \\ \texttt{"minusy"} \\ \texttt{"z"} \end{bmatrix}$$

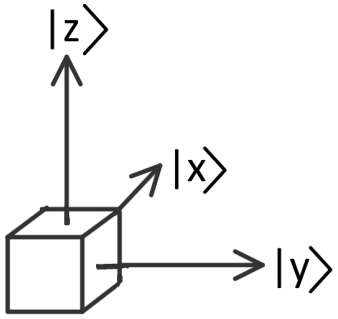

Figure 13: Intertwiner with arrows showing orientation

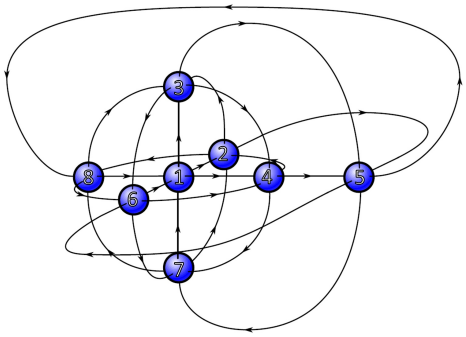

Figure 14: Model of spinfoam with cuboid intertwiners. Source: [2]

which corresponds to intertwiner 7 in Figure 14. This allows the code to be generalized to different coherent states, as different `v` inputs specify different states. The other inputs correspond to the same inputs for the Wigner matrix, where `alpha`, `beta`, and `gamma` are the Euler angles and give the rotation angle of the intertwiner.

`Tensor()` takes the input `v` and creates six vectors, v_i that describe the orientation of each of the cuboid's six faces. For example, in the $j = \frac{1}{2}$ case, intertwiner 1 has: $v_1 = [\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}]$, $v_2 = [\frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}}]$, $v_3 = [\frac{1}{\sqrt{2}}, i]$, $v_4 = [i, \frac{1}{\sqrt{2}}]$, $v_5 = [1, 0]$, $v_6 = [0, 1]$, which are the eigenvectors of the Pauli matrices. So `x` corresponds to v_1, `minusx` corresponds to v_2, `y` corresponds to v_3, and so on.

Define $D_i = D_i(\alpha, \beta, \gamma)$ as the Wigner matrices for each face of the cuboid. Each Wigner matrix is a function of the Euler angles α, β, and γ. The code then computes the following multiplication:

$$I = D_1 \left|v_1\right\rangle \otimes D_2 \left|v_2\right\rangle \otimes D_3 \left|v_3\right\rangle \otimes \left\langle v_4\right| D_1^\dagger \otimes \left\langle v_5\right| D_2^\dagger \otimes \left\langle v_6\right| D_3^\dagger \qquad (29)$$

where I is the resulting integrand with a specific rotation angle given by `alpha`, `beta`, and `gamma`. The optimized code for the `tensor()` function is given in appendix C.2.

3.1.3 Intertwiner function

The `Intertwiner()` function takes the `Tensor()` function, and uses it to create an intertwiner with a group averaging over SU(2). It does this by taking equation (29), which is a function of α, β, and γ through the Wigner matrices D_i, and integrating over α, β, and γ to remove the dependence on those variables. So

$$I_{ind} = \frac{1}{8\pi^2} \int_{-\pi}^{\pi} d\alpha \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} d\beta \ \sin(\beta) \int_{-\pi}^{\pi} d\gamma \ I(\alpha, \beta, \gamma) \tag{30}$$

where $\frac{\sin(\beta)}{8\pi^2}$ is the measure and $I(\alpha, \beta, \gamma)$ is given by Equation (29). Here we integrate over SU(2) in a specific parametrization, which is also called group averaging. Thus, I_{ind} is an SU(2) invariant tensor, i.e. it is invariant under the action of any SU(2) element defined as in (29). This integration is computed using the `cuhre()` function in Julia.

The `cuhre()` function is part of the `Cuba.jl` package, which is a "Julia wrapper around the C Cuba library, version 4.2, by Thomas Hahn." [13] `cuhre()` is a deterministic algorithm that employs a cubature rule to estimate the value of an integral. Of the algorithms offered in the `Cuba` package, most of which employ Monte Carlo integration, it is the most efficient at performing integrations that are highly oscilliatory. [14]

To create the intertwiners, `cuhre()` was run with 100 as the minimum number of evaluations, and 10^4 as the maximum number of evaluations. This gave a relative accuracy of 99.99860730065676% in the $j = \frac{1}{2}$ case. As previously discussed, computing all of these integrations is the most computationally expensive process in the code, one integration takes approximately 0.03 seconds in the $j = \frac{1}{2}$ case and 6 seconds for $j = 5$.

After the intertwiner is created, the entries need to be sorted to ensure that the intertwiner has the appropriate orientation, this is accomplished with the `srt()` function. Every intertwiner is created so that the entries in the array are arranged as if the intertwiner has the $(|x\rangle, |y\rangle, |z\rangle)$ orientation, but if the intertwiner has a different orientation, say $(|x\rangle, \langle y|, |z\rangle)$, then the entries must be rearranged. In this case the intertwiner would be created with the orientation $(|x\rangle, |-y\rangle, |z\rangle)$ and then the array entries would have their second and fifth indices swapped, so if the intertwiner is created as $I^{m_4',m_5',m_6'}_{m_1,m_2,m_3}$, it becomes $I^{m_4',m_2,m_6'}_{m_1,m_5',m_3}$.

The optimized code for the `Intertwiner()` and `srt()` functions is given in Appendix C.3.

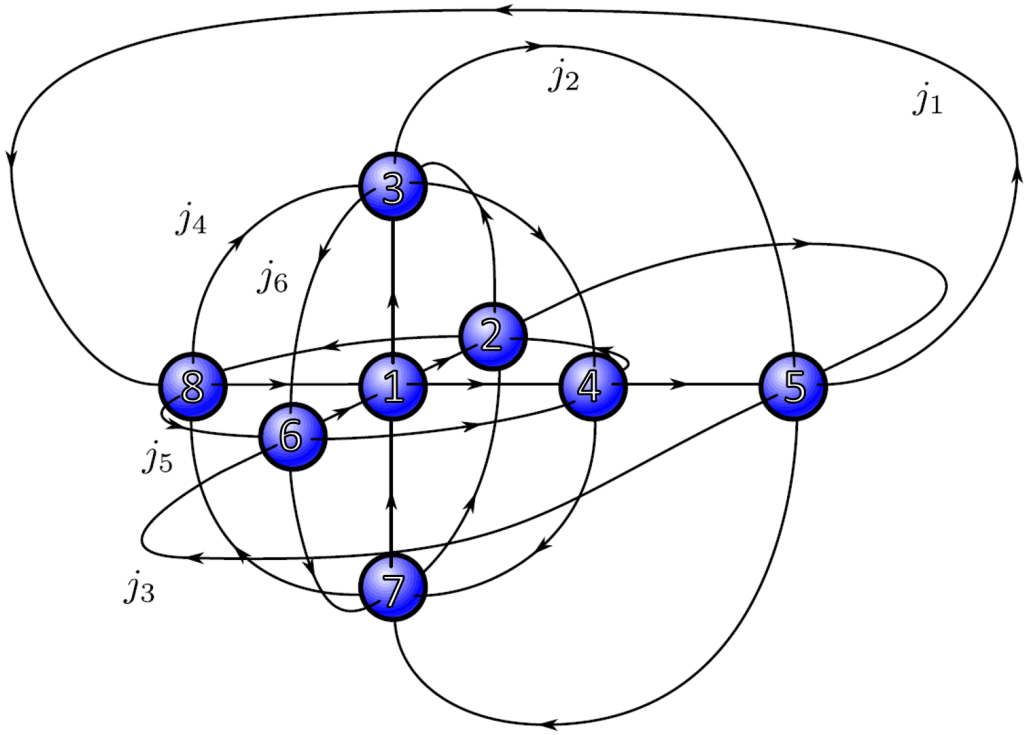

Figure 15: Model of spinfoam with cuboid intertwiners. Source: [2]

3.1.4 Performing the Contraction

To compute the spinfoam amplitude, eight intertwiners must be created, as shown in Figure 15. Once all of the intertwiners are created, they are contracted over all indices in a particular pattern, as described in section 2.2.3. This is where the orientation defined in section 3.1.3 become important, because it determines which edges are contracted together, and therefore which edges are bras and which are kets. For instance, if we label intertwiner 1 in Figure 15 as $A^{a_4,a_5,a_6}_{a_1,a_2,a_3}$ and intertwiner 2 as $B^{b_4,b_5,b_6}_{b_1,b_2,b_3}$, then contracting the two intertwiners requires setting $a_1 = b_4$, since the $|x\rangle$ component of intertwiner 1 connects to the $\langle -x|$ component of intertwiner 2, so the orientation defines the inner product. There are 24 independent indices in the spinfoam, so to compute the contraction naïvely, it is necessary to loop over each index. The code to perform the contraction naïvely is give in Appendix D.3.

3.2 Optimizing the code

Since, in the code, there are several instance of nested loops that are performing operations independently of each other, there are two primary ways to optimize the code. The first is to employ parallel computing so that operations can be performed simultaneously, and the second is to ensure that the time spent in each step of the loop is as short as possible. The former will be discussed in section 3.2.1, and the latter can be accomplished in part by declaring input variable types when defining function, but must otherwise be addressed on a case by case basis, as seen in sections 3.2.2-3.2.5.

3.2.1 Parallel Computing in Julia

Parallel computing is a method of optimizing a computation that involves many calculations that are independent of each other. For instance, when finding the sums of the rows of a matrix, the result of summing one row does not affect the sums of the others. If a person was performing this computation manually, they could assign one row to each of their friends, and end up with the same result much faster than if they had summed each row, one after another, as an individual. This is the manner in which parallel computing works. Instead of performing the calculations in order on one core, the task is divided among several cores. Julia offers three different levels of parallelism [15]: Julia Coroutines, which allows the user to suspend and resume computations by manually interfacing with the operating system; Multi-Threading, which allows parallel threads to execute tasks simultaneously; and Distributed Processing, which is provided by the `Distributed` module and is the subject of the remainder of this section.

The `Distributed` package is part of the main Julia library, and is the most straight forward method to employ parallel computing in Julia. Once the package is loaded, `addproc(n)` is called where `n` is the number of cores that are available. One of the cores is then assigned to process 1, which does not perform work unless it is the only available core, the remaining `n-1` cores are assigned as workers.

Now, the computation that is going to be parallelized is written as a single loop and the calculation to be performed is written as a function, `kernel(v)`, outside of the loop. In the example of summing the rows of a matrix, the rows would be the variable `v` and would be looped over, and the `kernel` function would be the operation of summing a row. Before entering the loop,

an empty `Nx1` array of `Futures`[2], is declared as `arr`, where `N` is the number of steps in the loop. Inside of the loop, a number from `2` to `n` is assigned to a variable, say `p`. The function `remotecall(kernel,p,v)` is then called and assigned to an entry `i` of `arr`. The function `remotecall(kernel,p,v)` assigns the calculation of `kernel(v)` to the worker `p`. Each worker begins their assigned tasks, and the loop is exited. An empty array `arrFinal` is declared to hold the results of the computation, and a new loop over the entries of `arr` is entered. Inside this loop, the function `fetch(arr[i])` is called and assigned to `arrFinal[i]`. Once a computation is complete, `fetch` retrieves its result and assigns that result to `arrFinal[i]`. Once all of the calculations are complete, the loop is exited and `arrFinal` holds the results of the computation.

3.2.2 Optimizing the Wigner Matrix

Since integrating the tensor requires the Wigner matrix to be computed many, many times, it is important that it be computed as quickly as possible. To achieve this, the processes that were taking the longest were found using the `@profile` command in the `Profile` package in Julia, which is run with the function to be profiled. As the function runs, @profile regularly makes note of what line of code a function is on. Using `Profile.print()` shows haw many times the `@profile` found the function on each line of code, the lines that have the highest number of times are the lines that the function is spending the longest time on.

The lines that were found to be taking the longest time were the lines computing the factorials in (28). Since this function is called multiple times, often repeating the same inputs, these lines were rewritten as functions preceded by `@memoize` from the `Memoize` package. `@memoize` remembers the inputs of a function and their results, so if that input is used again, it quickly returns the results so that time is not wasted recalculating results that have already been found. This increased the speed of calculating the intertwiner in the $j = \frac{1}{2}$ case by a factor of approximately 1.8.

[2]A `Future` is the place holder that Julia uses for a single computation of unknown status and time.

3.2.3 Optimizing the Tensor

In the original code for the `Tensor()` function, the vectors v_i in equation (29) were computed inside the loop that computed the tensor product. This meant that they were computed each time they were used, but these vectors are constant for any given `J` and `v`, and therefore don't need to be computed inside the `Tensor()` function at all. A new function, `PreTensor(j,v,MM,DD)`, was written to compute these vectors before the `Tensor()` function is called. It returns a 2D array, `A`, that contains the vectors as columns of the array. `Tensor()` was then rewritten to take `A` as an input. This reduces the number of calculations that need to occur when `Tensor()` is called, and increased the speed of calculating the intertwiner in the $j = \frac{1}{2}$ case by a factor of approximately 1.7.

3.2.4 Optimizing the Intertwiner

Cuboid intertwiners are a subspace of a larger vector space. They have a high degree of symmetry and contain many zero elements. To minimize the computation time, the integration was only done on those elements of the array that are non-zero. These elements are found by employing a rule for the intertwiner's magnetic indices. This rule defines the subspace in which the cuboid intertwiners live. For an intertwiner $I^{m_4',m_5',m_6'}_{m_1,m_2,m_3}$, if

$$m_1 + m_2 + m_3 - m_4' - m_5' - m_6' = 0 \tag{31}$$

then that entry in the array may be non-zero, otherwise the entry vanishes. This is analogous to the 3-valent intertwiners, whose entries are specified by the Clebsch-Gordon coeffients. This rule was employed by creating an array containing the result of the left hand side of equation (31) and then using the Julia command `findall()` to find the indices of all of the entries that equal zero, and then running the integration over those indices. This integration was parallelized using the `Distributed` module, which implements distributed memory computing to spilt a loop between multiple CPUs. Another method of finding the zeros is to express one magnetic index in terms of the other five, and then running a `for` loop over the other five indices, but this method is more difficult to parallelize. When the two methods are not parallelized, they both take approximately the same time to run in the $j = \frac{1}{2}$ case, taking approximately 39 seconds for the `findall()` method as

opposed to 41 seconds for the other method. For this reason, the `findall()` method was chosen.

To parallelize the code, the inside of the loop that computes the integration using `cuhre()` was rewritten as a function outside of `intertwiner()` and the loop was parallelized as detailed in 3.2.1. This resulted in a speed increase that was approximately equal to the number of workers, so two workers makes the code twice as fast, three works makes it three times as fast, etc.

3.2.5 Optimizing the Contraction

As with creating the intertwiners, computing the contraction is very computationally expensive. So, similarly to the function `Intertwiner()`, the code to contract the intertwiners, `contract()`, employs rules to find zero entries in the intertwiners, reducing the number of computations. It sets the value of seven of the magnetic indices in terms of the other 17 indices[3]. For instance, if intertwiner 2 is labelled as $B^{b_4,b_5,b_6}_{b_1,b_2,b_3}$, set $b_1 = -b_2 - b_3 + b_4 + b_5 + b_6$. These seven magnetic indices are then converted to code notation. In this way, seven `for` loops are omitted. This increases the speed of the code by a factor of 1000 in the $j = \frac{1}{2}$ case. Since there are $(2j+1)^{17}$ contractions (in the case that all spins are equal to j), this is saving exponentially more time as j increases. See the code for the `contract()` function in Appendix D.1 for more details.

Another method for reducing the computation time is to compute the contractions in steps. For instance, contract intertwiners 1 and 2, 3 and 4, 5 and 6, and 7 and 8 separately, and then contract the resulting four contracted intertwiners. The code for this method is given in appendix D.2. Contracting in steps requires more memory than both of the aforementioned methods, since the results of the first round of contractions need to be stored, and this method is not fully optimized. The resulting four tensors could be contracted in pairs, and then the two new tensors could be contracted to achieve a final result, which would use even more memory. The partially optimized method only decreases the computational time by a factor of 100 in the $j = \frac{1}{2}$ case, and, since optimizing further would use even more memory, the method given in Appendix D.1 was ultimately chosen.

[3]Note that after this condition has been implemented for seven intertwiners, the condition for the last remaining intertwiner is automatically satisfied.

3.3 `toText()` and `fromTxt()` Functions

Since computing intertwiners is very computationally expensive, it makes sense to have a method for saving the intertwiners for later use. Since the same intertwiner can be needed for multiple contractions / vertex amplitudes, it is more efficient to compute it once and then read it in from a file for all subsequent uses than it is to compute it every time. The functions `toTxt()` and `fromTxt()` are used to save the intertwiners to `.txt` files, and then convert the `.txt` files back into arrays in Julia. These functions use the base Julia input and output functions `open()`, `read()`, and `write()`.

The function `toTxt()` simply runs over all of the elements of the array and writes them to a `.txt` file. The function `fromTxt()` reads the `.txt` file in as a string. It then splits the string into a vector containing smaller strings, for instance, the string "1.0 + 2.0im" would become the strings "1.0", "+", and "2.0im". Since all of the numbers in the array that were originally written to the `.txt` file are complex numbers, each number now corresponds to three strings. The vector of strings is then reshaped into a `3xn` array, `n` being the length of the original vector divided by 3. The rows of strings are then concatenated and the resulting vector of strings is then converted to a vector of complex numbers. This vector is then rearranged into a six dimensional array which corresponds to an intertwiner. So using these functions, each intertwiner only needs to be computed once. These functions are given in appendix E.

3.4 Running the Code on an HPC (Symmetry)

Once the code was fully optimized for the $j = \frac{1}{2}$ case, the code was then run on the Perimeter Institutes's high performance computer, Symmetry. Symmetry uses the resourse manager Slurm and has two head nodes that are used to interact with the system and 76 compute nodes to run applications. Each of the compute nodes contains 40 Intel Xeon Gold (Skylake) cores and 200 GB of memory (RAM). Symmetry uses a high-performance InfiniBand network to connect the nodes to a file server that hosts a 233 TB GPFS file system and Julia 1.3.0 is pre-installed. [16]

Symmetry was accessed via ssh using PuTTY, and the code for computing the intertwiners and their contraction was run using a `.batch` script that was submitted to Slurm using the `sbatch` command. An example `.batch` script is included in Appendix F. The script runs a Julia code, `test.jl`, which

computes the intertwiners and their contraction in the $j = \frac{1}{2}$ case. `test.jl` also saves the intertwiners to `.txt` files and outputs information about the code's performance. The `.batch` script specifies that any output will be saved in `test.out`, error messages are saved in `test.err`, and all files are saved in the `/home/callen/Testing` directory. Saved files were accessed via ssh using WinSCP.

4 Results

The hypercuboid vertex amplitude was computed and studied in four different cases. To easily differentiate these cases, in this following sections the six spins associated with the hypercuboid are denoted $j = [j_1, j_2, j_3, j_4, j_5, j_6]$.

- In the first case, presented in section 4.1, all six spins were increased at the same time. So beginning from all spins $j = \frac{1}{2}$, the contraction is then computed for all spins $j = 1$, then all spins $j = \frac{3}{2}$, and so on. This case is denoted $j = [x, x, x, x, x, x]$, where x is the variable representing the spin, and corresponds to hypercubes of different sizes.

- In the second case, presented in section 4.2 three of the spins were held constant while the remaining three spins were increased at the same rate, denoted $j = [c, c, c, x, x, x]$ where c represents the spins that remain constant. This example can be understood as having one initial and final cube of fixed size connected by "stretched" hypercuboids of varying size.

- In the third case, presented in section 4.3, four spins were held constant while the remaining two were increased. This case is denoted $j = [x, c, c, x, c, c]$, and corresponds to a hypercuboid with two faces that vary in size. Note that these configurations in general do not correspond to hypercubes, i.e. cannot be understood as a polyhedron prescribed by four edge lengths.

- In the final case, presented in section 4.4, five spins were held constant while one was varied, denoted $j = [x, c, c, c, c, c]$. As for the previous case, these configurations in general do not correspond to hypercuboids.

This section presents the computed results of these cases graphically, for the straight numerical results of these cases, see appendix A.

These cases were chosen to get a sense of how the numerically computed amplitude behaves. The case where all of the spins are increased at the same rate is the case that best exemplifies the large-j limit (for sufficiently large j), and should therefore converge to the formula computed for the asymptotic vertex amplitude, equation (26). It is however, the most computationally expensive, since the number of entries in the arrays grows exponentially. The cases where fewer spins are varied are less representative of the idea of

the large-j limit, since some spins remain small. Conversely, these cases thus require fewer entries to be computed, and are therefore less computationally expensive. Selected computation times are given in appendix B.

Each of the above cases allows the equation for the semi-classical vertex amplitude, (26), to be written as a function of a single variable, $f(x)$, and to be plotted along with the computed values. All of the plots were created using Maple. The function $f(x)$ is approximately of the form,

$$f(x) \simeq \frac{a}{\sqrt{p(x)}}, \tag{32}$$

where a is some constant and $p(x)$ is a polynomial. The degree of the polynomial $p(x)$ depends on which case is being looked at. For instance, in the $j = [x, x, x, x, x, x]$ case, the polynomial is of degree 57, whereas in the $j = [c, c, c, x, x, x]$ case the polynomial is of degree 17.

For all of the data, the percent difference between the computed values and the semi-classical formula, (26), was found using the formula

$$\% \text{ difference} = \frac{|f(\texttt{X[i]}) - \texttt{Y[i]}|}{\texttt{Y[i]}} \tag{33}$$

where $f(x)$ is the semi-classical formula for the vertex amplitude, and `Y[i]` is the computed entry `i`, `X[i]` is the value of x for entry `i`.

For selected cases, a curve was fitted to the computed values of the vertex amplitude. This was done using the `NonlinearFit()` function in Maple's statistics package. The `NonlinearFit()` function fits a specified model function to the data by minimizing the least-squares error. The `NonlinearFit()` function requires at least four parameters as input. These parameters are `f`, a model function that the data will be fit to; `X`, a vector containing the data corresponding to the independent variable; `Y` a vector containing the data corresponding to the dependent variable; and `x`, the name of the independent variable in the model function. In addition the model function chosen for the fit was

$$g(x) = \frac{a}{bx^A + cx^B + dx^C}, \tag{34}$$

where a, b, c, and d are the parameters chosen by the fit and A, B, and C were chosen manually to create a curve that best reflects the data. An optional fifth parameter, `initialvalues`, was also used as an input in `NonlinearFit` to specify initial values for the parameters to be fit, a, b, c, and d. This model

was chosen since it closely resembles the form of the vertex amplitude, (32), but omits the square root, since the square root introduces complex numbers into the least squares regression which the `NonlinearFit()` function is not made to handle.

The results for the hypercuboid vertex amplitude are presented in the following sections and compared it to the asymptotic formula. Where possible intersection points between the asymptotic formula and curve fit are found, and compiled in a table in section 4.5. These results are discussed in section 4.

4.1 All spins the same ($j = [x, x, x, x, x, x]$)

The following plots are for the case that $j = [x, x, x, x, x, x]$. They contain the least number of data points because, as seen in appendix B, the computational times grow in this case grow the most rapidly as x increases. This is because there are $(2x+1)^6$ entries in each array, more than in any other case. For example, the intertwiners in the $j = [0.5, 0.5, 0.5, 0.5, 0.5, 0.5]$ case can be computed in approximately 0.06 seconds, and their contraction can be computed in 0.557978 seconds, meanwhile, it takes aproximately 220 seconds to compute intertwiners in the $j = [2, 2, 2, 2, 2, 2]$ case, and approximately 3.3 hours to contract them. In the curve fit, the parameters A, B, C in the model function (34) were chosen to be 17, 9, and 8.

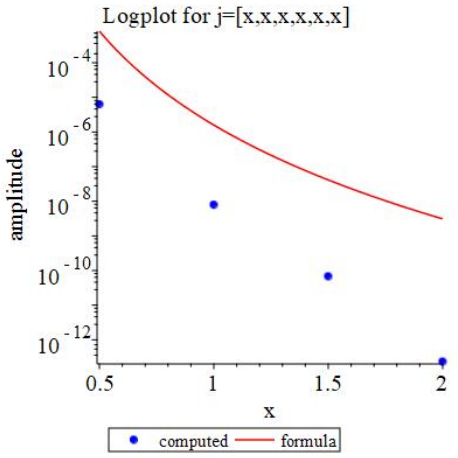

Figure 16: Log plot comparing the computed amplitudes to the formula for the amplitude in the semi-classical limit for $j = [x, x, x, x, x, x]$.

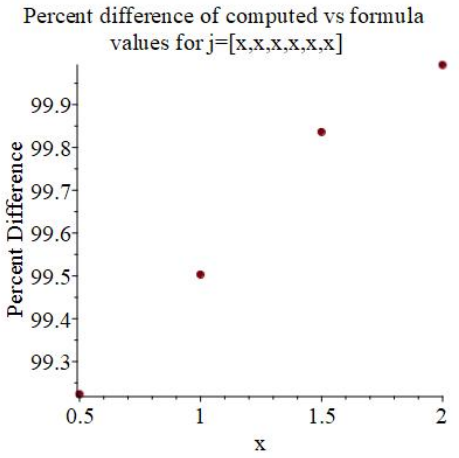

Figure 17: Plot showing the percent difference between the computed and semi-classical formula values for $j = [x, x, x, x, x, x]$.

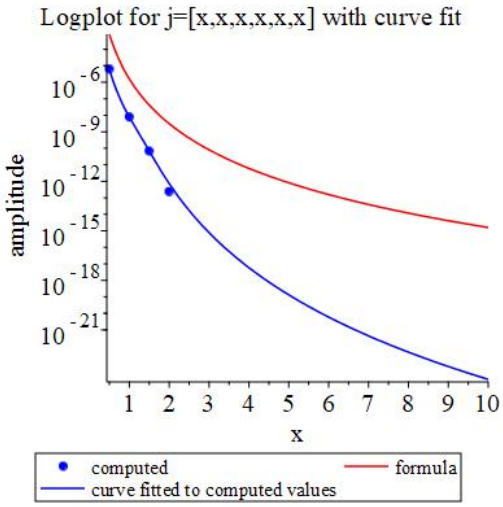

Figure 18: Log plot showing the curve fit for $j = [x, x, x, x, x, x]$. The parameters A, B, C in the model function (34) were chosen to be 17, 9, and 8.

4.2 Three spins varied ($j = [c, c, c, x, x, x]$)

In this section, three cases are studied, where $c = 0.5$, $c = 1$, and $c = 1.5$

4.2.1 $j = [0.5, 0.5, 0.5, x, x, x]$

The following plots are for the case that $j = [0.5, 0.5, 0.5, x, x, x]$. In the curve fit, the parameters A, B, and C in the model function (34) were chosen to be 8, 6, and 5.

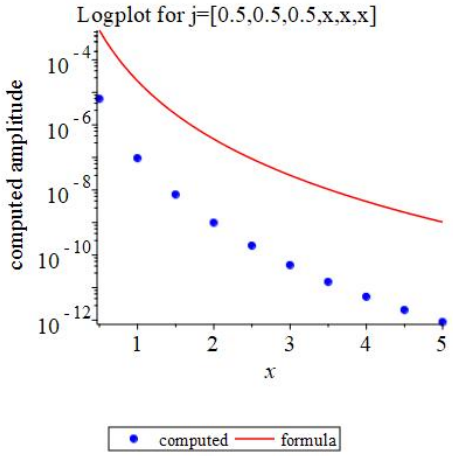

Figure 19: Log plot comparing the computed amplitudes to the formula for the amplitude in the semi-classical limit for $j = [0.5, 0.5, 0.5, x, x, x]$.

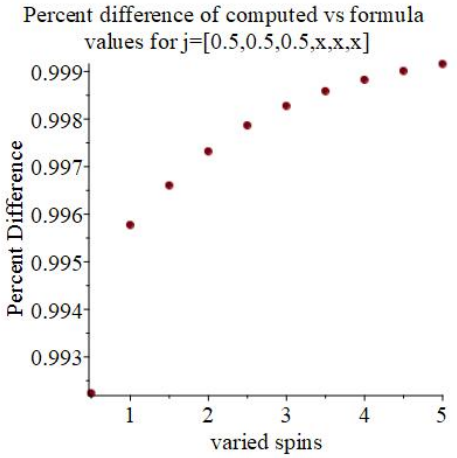

Figure 20: Plot showing the percent difference between the computed and semi-classical formula values for $j = [0.5, 0.5, 0.5, x, x, x]$.

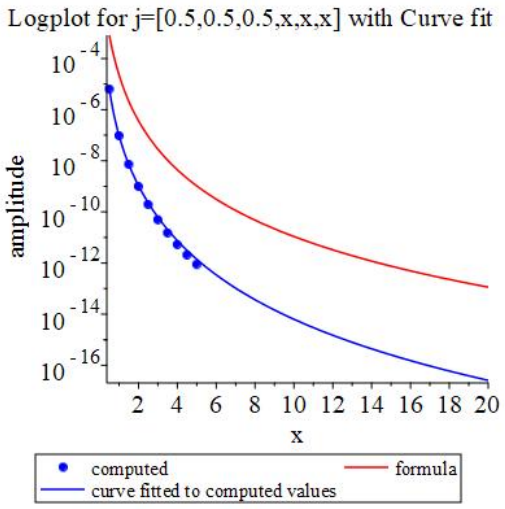

Figure 21: Log plot showing the curve fit for $j = [0.5, 0.5, 0.5, x, x, x]$. The parameters A, B, C in the model function (34) were chosen to be 8, 6, and 5.

4.2.2 $j = [1, 1, 1, x, x, x]$

The following plots are for the case that $j = [1, 1, 1, x, x, x]$. In the curve fit, the parameters A, B, and C in the model function (34) were chosen to be 6, 5, and 4, and figure 25 shows where the curve fit intersects with the formula for the semi-classical limit, this intersection point could not be found using Maple's `fsolve()` function. This could potentially be because the x-value of the intersection point is too large ($x \approx 10^7$).

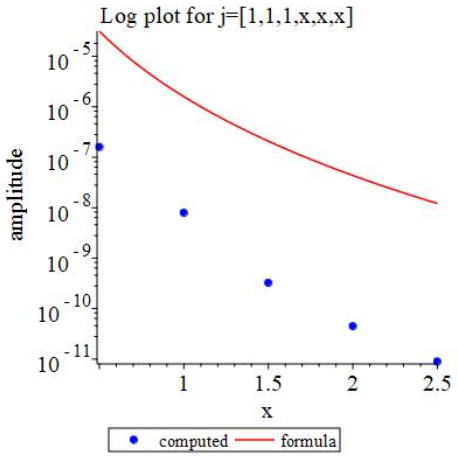

Figure 22: Log plot comparing the computed amplitudes to the formula for the amplitude in the semi-classical limit for $j = [1, 1, 1, x, x, x]$.

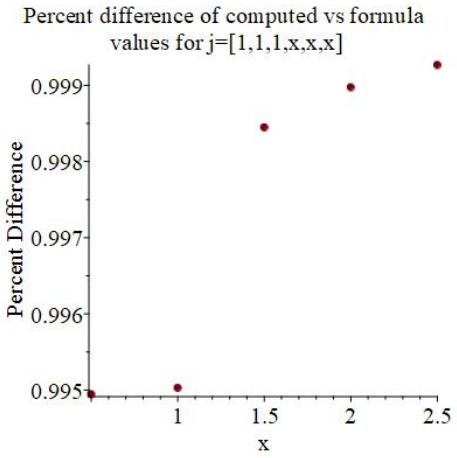

Figure 23: Plot showing the percent difference between the computed and semi-classical formula values for $j = [1, 1, 1, x, x, x]$.

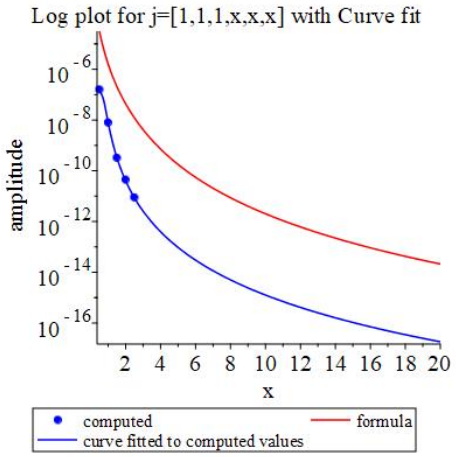

Figure 24: Log plot showing the curve fit for $j = [1, 1, 1, x, x, x]$. The parameters A, B, C in the model function (34) were chosen to be 6, 5, and 4.

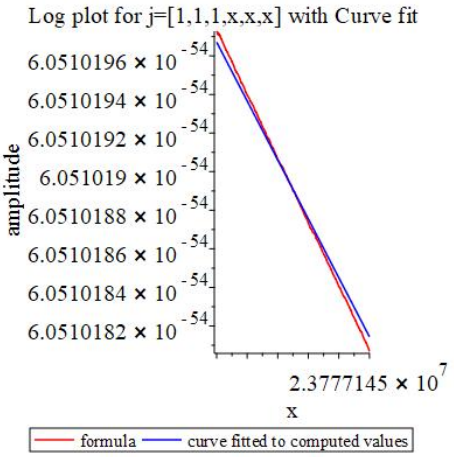

Figure 25: Log plot showing where the curve fit intersects with the formula for the semi-classical limit for $j = [1, 1, 1, x, x, x]$.

4.2.3 $j = [1.5, 1.5, 1.5, x, x, x]$

The following plots are for the case that $j = [1.5, 1.5, 1.5, x, x, x]$.

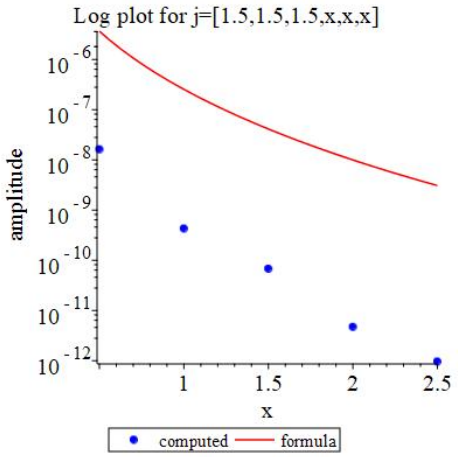

Figure 26: Log plot comparing the computed amplitudes to the formula for the amplitude in the semi-classical limit for $j = [1.5, 1.5, 1.5, x, x, x]$.

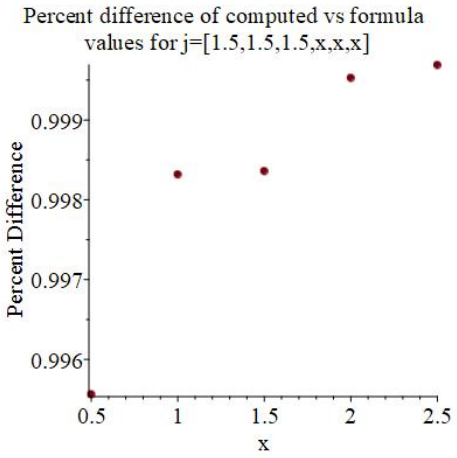

Figure 27: Plot showing the percent difference between the computed and semi-classical formula values for $j = [1.5, 1.5, 1.5, x, x, x]$.

4.3 Two spins varied

4.3.1 $j = [x, 0.5, 0.5, x, 0.5, 0.5]$

The following plots are for the case that $j = [x, 0.5, 0.5, x, 0.5, 0.5]$. In the curve fit, the parameters A, B, and C in the model function (34) were chosen to be 5, 4, and 2, and figure 25 shows where the curve fit intersects with the formula for the semi-classical limit. The intersection point for the curve fit and the semi-classical limit was found to be $x = 131.9609642$ using Maple's `fsolve()` function.

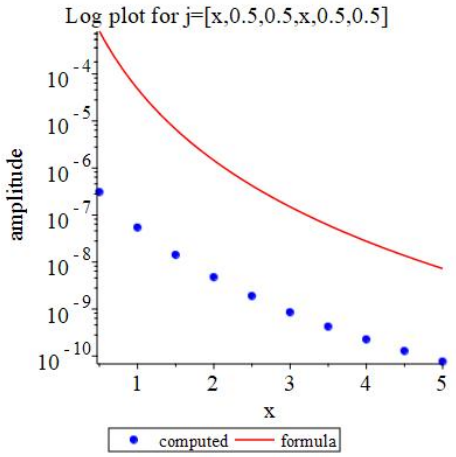

Figure 28: Log plot comparing the computed amplitudes to the formula for the amplitude in the semi-classical limit for $j = [x, 0.5, 0.5, x, 0.5, 0.5]$.

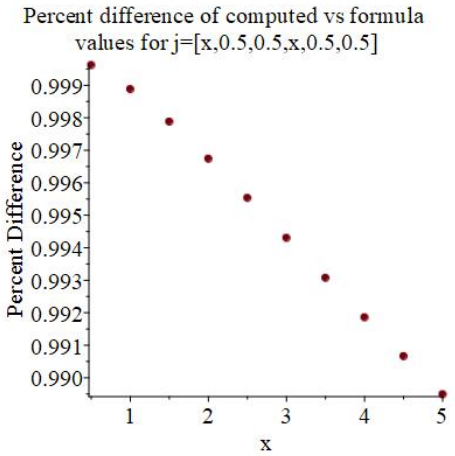

Figure 29: Plot showing the percent difference between the computed and semi-classical formula values for $j = [x, 0.5, 0.5, x, 0.5, 0.5]$.

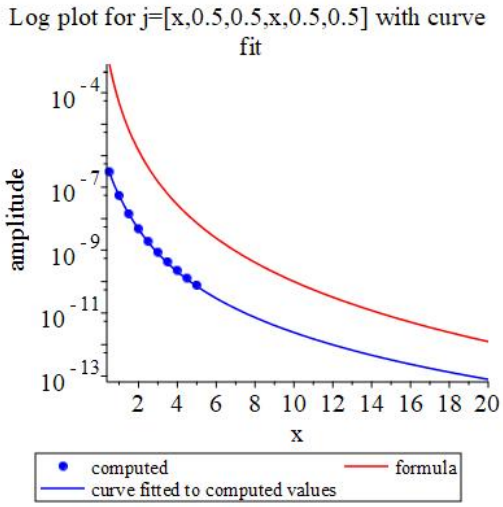

Figure 30: Log plot showing the curve fit for $j = [x, 0.5, 0.5, x, 0.5, 0.5]$. The parameters A, B, C in the model function (34) were chosen to be 5, 4, and 2.

4.3.2 $j = [x, 1, 1, x, 1, 1]$

The following plots are for the case that $j = [x, 1, 1, x, 1, 1]$. In the curve fit, the parameters A, B, and C in the model function (34) were chosen to be 5, 4, and 2, and figure 25 shows where the curve fit intersects with the formula for the semi-classical limit. The intersection point for the curve fit and the semi-classical limit was found to be $x = 1865.559744$ using Maple's `fsolve()` function.

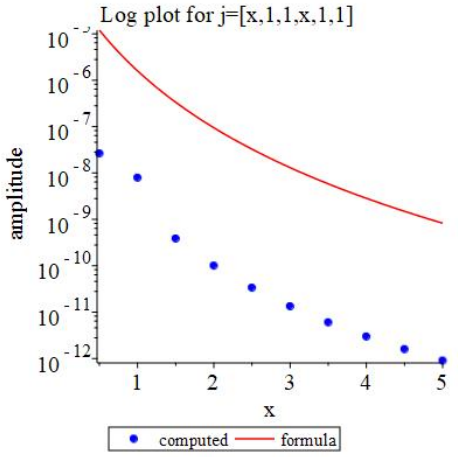

Figure 31: Log plot comparing the computed amplitudes to the formula for the amplitude in the semi-classical limit for $j = [x, 1, 1, x, 1, 1]$.

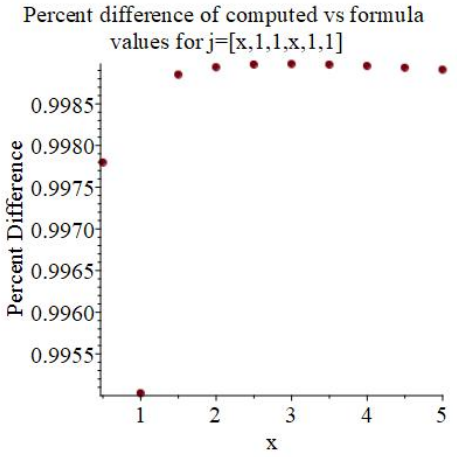

Figure 32: Plot showing the percent difference between the computed and semi-classical formula values for $j = [x, 1, 1, x, 1, 1]$.

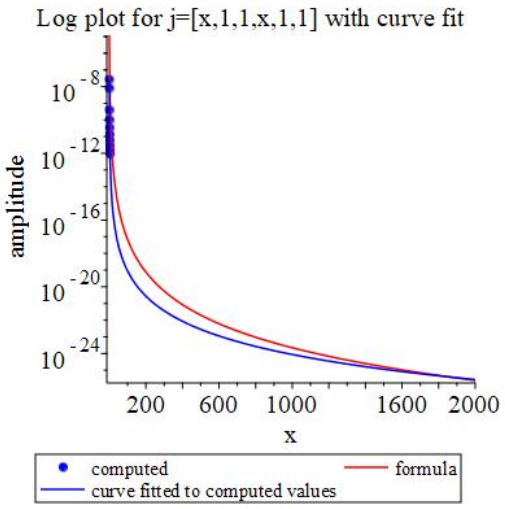

Figure 33: Log plot showing the curve fit for $j = [x, 1, 1, x, 1, 1]$. The parameters A, B, C in the model function (34) were chosen to be 5, 4, and 2.

4.4 One spin varied

In this section, the model function for the curve fit was changed to

$$g(x) = \frac{a}{|b|\, x^A + |c|\, x^B + |d|\, x^C}, \tag{35}$$

because, using the model function given by (34) on the data in this section, minimizing the least-squares error resulted in at least one of the parameters b, c, and d being negative, which fit the initial data well, but did not give an accurate idea of what the function will do long term, as shown in figure 34.

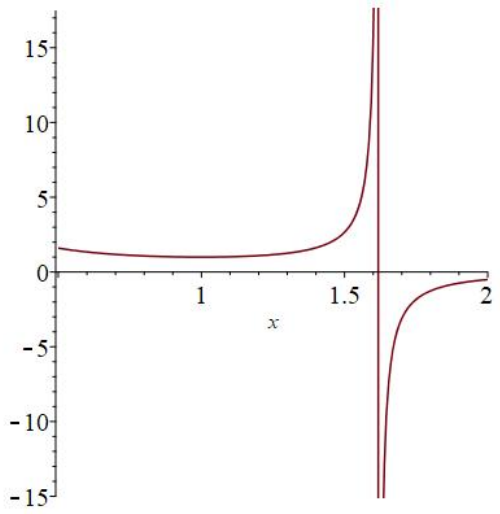

Figure 34: Plot showing what occurs when $a = 1$, $b = -1$, $c = 1$, and $d = 1$ in (34). When one of the parameters b, c, or d is negative, the resulting function does not look as expected.

4.4.1 $j = [x, 0.5, 0.5, 0.5, 0.5, 0.5]$

The following plots are for the case that $j = [x, 0.5, 0.5, 0.5, 0.5, 0.5]$. In the curve fit, the parameters A, B, and C in the model function (35) were chosen to be 6, 3, and 0.

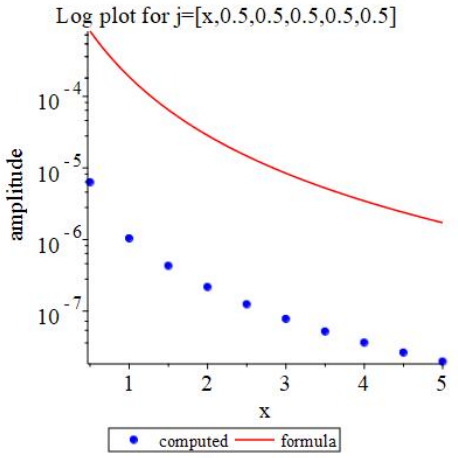

Figure 35: Log plot comparing the computed amplitudes to the formula for the amplitude in the semi-classical limit for $j = [x, 0.5, 0.5, 0.5, 0.5, 0.5]$.

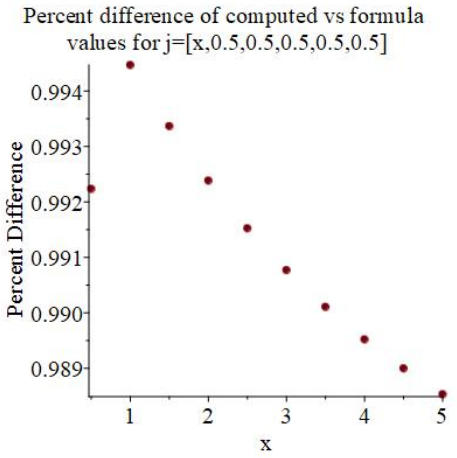

Figure 36: Plot showing the percent difference between the computed and semi-classical formula values for $j = [x, 0.5, 0.5, 0.5, 0.5, 0.5]$.

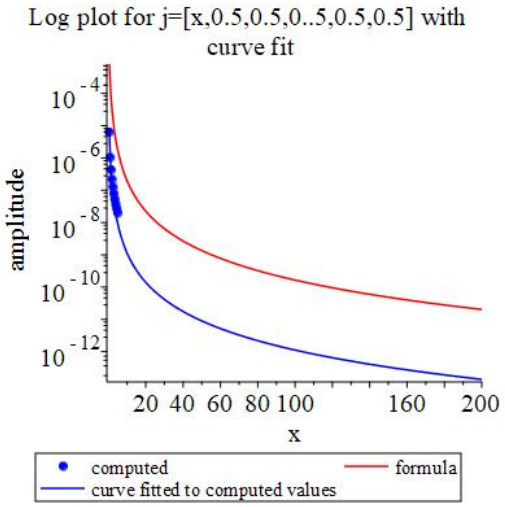

Figure 37: Log plot showing the curve fit for $j = [x, 0.5, 0.5, 0.5, 0.5, 0.5]$. The parameters A, B, C in the model function (35) were chosen to be 6, 3, and 0.

4.4.2 $j = [x, 1, 1, 1, 1, 1]$

The following plots are for the case that $j = [x, 1, 1, 1, 1, 1]$. In the curve fit, the parameters A, B, and C in the model function (35) were chosen to be 3, 2, and 1.7.

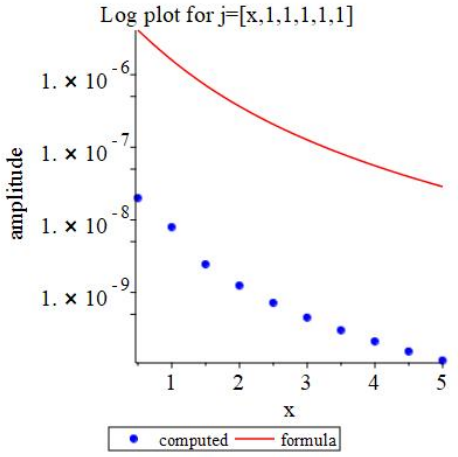

Figure 38: Log plot comparing the computed amplitudes to the formula for the amplitude in the semi-classical limit for $j = [x, 1, 1, 1, 1, 1]$.

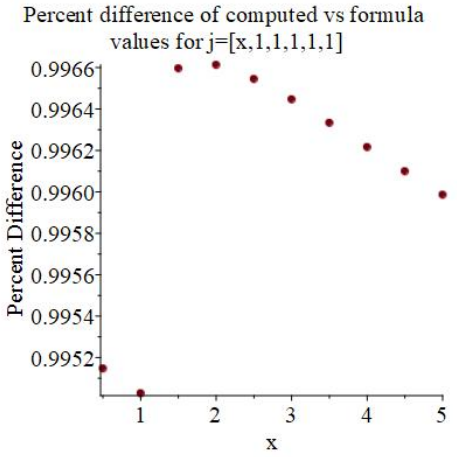

Figure 39: Plot showing the percent difference between the computed and semi-classical formula values for $j = [x, 1, 1, 1, 1, 1]$.

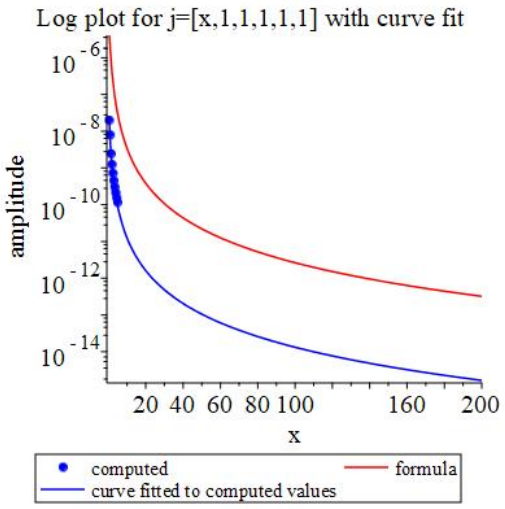

Figure 40: Log plot showing the curve fit for $j = [x, 1, 1, 1, 1, 1]$. The parameters A, B, C in the model function (35) were chosen to be 3, 2, and 1.7.

4.4.3 $j = [x, 1.5, 1.5, 1.5, 1.5, 1.5]$

The following plots are for the case that $j = [x, 1.5, 1.5, 1.5, 1.5, 1.5]$. In the curve fit, the parameters A, B, and C in the model function (35) were chosen to be 3, 2, and 1.7.

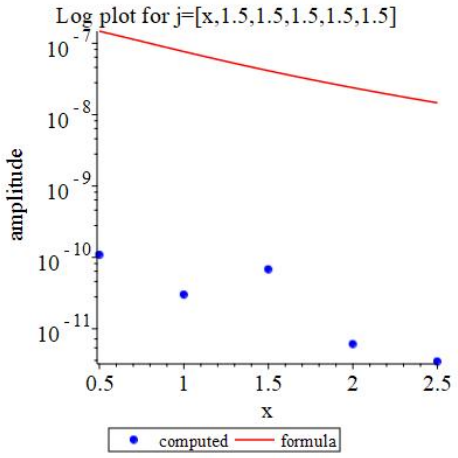

Figure 41: Log plot comparing the computed amplitudes to the formula for the amplitude in the semi-classical limit for $j = [x, 1.5, 1.5, 1.5, 1.5, 1.5]$.

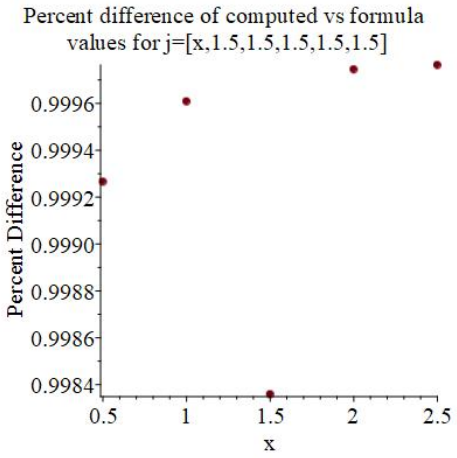

Figure 42: Plot showing the percent difference between the computed and semi-classical formula values for $j = [x, 1.5, 1.5, 1.5, 1.5, 1.5]$.

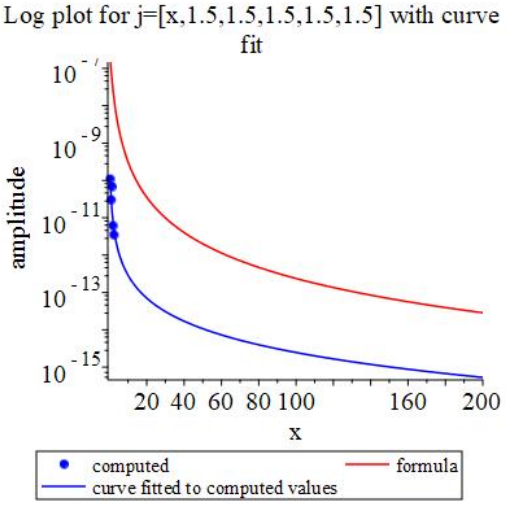

Figure 43: Log plot showing the curve fit for $j = [x, 1.5, 1.5, 1.5, 1.5, 1.5]$. The parameters A, B, C in the model function (35) were chosen to be 3, 2, and 1.7.

4.5 Intersection points

All of the intersection points found in the prior sections are compiled in the table below.

Spins $j =$	**intersection** $x \approx$
$[1, 1, 1, x, x, x]$	2.4×10^7
$[x, 0.5, 0.5, x, 0.5, 0.5]$	1.4×10^2
$[x, 1, 1, x, 1, 1]$	1.9×10^3

Table 1: Table of known intersection points

5 Discussion

From [17] and [18], it is expected that the semi-classical formula should become a good approximation to the vertex amplitude around $j \sim 10$. These results were obtained for a vertex amplitude dual to a 4-simplex, where five 4-valent intertwiners are contracted, which is a significantly simpler case compared to the one studied here. Due to time constraints as well as the scaling of computational cost as the spins are increased, it was not possible to achieve computational results for spins this large. Thus it is not surprising that we do not observe a convergence of the vertex amplitudes and its semi-classical approximation, since we did not reach the anticipated semi-classical regime. However, the curve fits for the cases where $j = [1, 1, 1, x, x, x]$, $j = [x, 0.5, 0.5, x, 0.5, 0.5]$, and $j = [x, 1, 1, x, 1, 1]$ do provide evidence that the semi-classical formula does begin to approach the computed amplitudes. The other cases, however, do not show this convergence.

Why the discrepancy? The first thing to note is that the plots presented in section 4 are log plots, and that while the percent difference between the computed and semi-classical formula values increases, the absolute difference is actually decreasing, since the values are very small. The curve fit itself is also a factor. Since the model function used for the curve fit is a rough guess at what the curve should look like, and necessarily omits the square root seen in the semi-classical formula, it could be that the computational results would quickly diverge from their predicted values.

This flaw with the curve fit could explain a few features seen in section 4. Such as why, in the $j = [x, 0.5, 0.5, 0.5, 0.5, 0.5]$ and $j = [x, 1, 1, 1, 1, 1]$ cases,

the percent error decreases, but no intersection point is observed. Or why, when looking at the intersection points in table 1, it does appear that the intersection is occurring later for higher values of c, which is the opposite of what is expected. The poor model function combined with the relatively few converging cases, means that an accurate idea of when intersection occurs cannot be ascertained. It only suggests that intersection does occur.

There is another possibility worth considering, namely that the transition to the asymptotic formula occurs for $j > 10$ in the hypercuboid model. While the construction is analogous to the triangulation case, the different combinatorics may be one explanation. Moreover, it has been shown that the hypercuboid models, as well as higher valent spin foam vertices, miss an implementation of the so-called volume simplicity constraint and thus allow for more general configurations even in the semi-classical limit [18], [19].

Taking the results at face value, however, is still useful since there is a clear difference in convergence rate between the different cases. These differences suggest that the transition to the semi-classical amplitude is not uniform, and may depend on the choice of spins. In the cases where two and three spins are varied and the rest remain constant, the constant terms remain Planckian, and therefore are expected to prevent the amplitude from becoming semi-classical as quickly. In fact, they appear to have the opposite effect. This suggests that the smaller spins in the amplitude have a larger effect on the path integral than anticipated, and that there is no straightforward transition to the semi-classical regime.

Moving away from the question of convergence for a moment, the results at least qualitatively confirm a feature of the cuboid intertwiner model: that the semi-classical amplitude of the cuboid intertwiner model does not show any oscillatory behaviour, as can be seen in equation (23). The reason for this is that the associated discrete gravity action of the hypercuboid vanishes. In the case of several semi-classical hypercuboids, this implies that they are glued in a flat way [2, 4]. The exact calculations of the vertex amplitudes also do not show any oscillatory behaviour, and thus confirm this feature far beyond the viability of the semi-classical regime. Hence, it is reasonable to assume that these quantum cuboids give rise to flat geometries in the deep quantum regime as well.

6 Conclusion

The objective of the work has been to design and implement algorithms to compute the spin foam vertex amplitude for cuboid intertwiners. Once the results were obtained, the goal was to investigate whether, and in which regime, the semi-classical formula for the vertex amplitude becomes a good approximation to the exactly computed vertex amplitude. Working in that direction, code was written to compute cuboid intertwiners of arbitrary spin, and to then contract those intertwiners, thus computing the vertex amplitude. A few preliminary cases were then examined, and while these cases do not prove convergence, they do show promising evidence that convergence may occur. This work also demonstrates differences between spinfoam models based on cubulations and spinfoam models based on triangulations, and shows that the smaller spin building blocks do have an effect on the vertex amplitude. Nevertheless, the exact calculations confirm a feature of the quantum cuboid vertex amplitude in the semi-classical regime, namely the absence of an oscillatory behaviour. This suggests that several of the quantum hypercuboids give rise to a flat space-time even in the deep quantum regime.

Therefore, there are several avenues for future work. Work can be done on obtaining results for higher spins and more cases to see how relevant the small spins are to the path integral. Since the code runs more efficiently for smaller spins, and therefore the small spin regime is readily accessible, using this code to examine the effect of smaller spins would be of particular interest. It would also be worthwhile to parallelize the `contract()` function, so that an examination of the higher spin cases can be done more efficiently. The hope would be to determine whether the semi-classical amplitude in the cuboid case is a good approximation around $j \sim 10$, similar to the triangulation case, or whether this formula is valid for spins larger than that. The code presented in this thesis can be adapted for other uses as well. The `intertwiner()` function can be generalized to create different six-valent intertwiners by adapting the coherent states, and the `contract()` function can be used to contract arbitrary six-valent intertwiners, e.g. corresponding to more general shapes or expressed in an orthonormal basis, which could help to better understand the properties of the vertex amplitude. Note that these intertwiners must be correctly configured in terms of their orientation and spins of the contracted components.

The purpose of this work, to prove the convergence of the semi-classical formula and the vertex amplitude, has not yet been met. However, this work has provided first evidence of convergence, and it has opened new avenues of inquiry. Over all, it is one step in proving convergence, and in employing numerical methods in loop quantum gravity, but more work must be done in this area.

References

[1] Juansempere, "Gimbaleuler2.svg." [Online; accessed 24-Jan-2020]. Available: `https://en.wikipedia.org/wiki/Euler_angles#/media/File:Gimbaleuler2.svg`. License: `https://creativecommons.org/licenses/by-sa/3.0/legalcode`.

[2] B. Bahr and S. Steinhaus, "Investigation of the spinfoam path integral with quantum cuboid intertwiners," *Phys. Rev. D*, vol. 93, p. 104029, May 2016.

[3] C. Rovelli, *Quantum Gravity*. Cambridge Monographs on Mathematical Physics, Cambridge University Press, 2004.

[4] B. Bahr and S. Steinhaus, "Hypercuboidal renormalization in spin foam quantum gravity," *Phys. Rev. D*, vol. 95, p. 126006, Jun 2017.

[5] C. Rovelli, "Zakopane lectures on loop gravity," *PoS*, vol. QGQGS2011, p. 003, 2011.

[6] E. Bianchi, "The construction of spin foam vertex amplitudes," *Symmetry, Integrability and Geometry: Methods and Applications*, Jan 2013.

[7] J. C. Baez, "An introduction to spin foam models of quantum gravity and bf theory," 1999. [Online; accessed 21-Apr-2020]. Available: `https://arxiv.org/abs/gr-qc/9905087`.

[8] A. Perez, "The spin-foam approach to quantum gravity," *Living Reviews in Relativity*, vol. 16, Feb 2013.

[9] E. R. Livine and S. Speziale, "New spinfoam vertex for quantum gravity," *Phys. Rev. D*, vol. 76, p. 084028, Oct 2007.

[10] E. Bianchi, P. Doná, and S. Speziale, "Polyhedra in loop quantum gravity," *Phys. Rev. D*, vol. 83, p. 044035, Feb 2011.

[11] S. Klöser, "Numerical investigations of the eprl spinfoam amplitude," Master's thesis, Universität Hamburg, 2017.

[12] J. J. Sakurai and J. Napolitano, *Modern Quantum Mechanics*. Cambridge University Press, 2 ed., 2017.

[13] Giordano, Mosè, "Cuba.jl," 2020. [Online; accessed 16-Feb-2020]. Available: `https://github.com/giordano/Cuba.jl`.

[14] T. Hahn, "Cuba — a library for multidimensional numerical integration," *Computer Physics Communications*, vol. 168, no. 2, pp. 78 – 95, 2005.

[15] "Julia manual: Parallel computing," 2020. [Online; accessed 06-Jul-2020]. Available: `https://docs.julialang.org/en/v1/manual/parallel-computing/`.

[16] PerimeterInstitute, "Symmetry documentation," 2020. [Online; accessed 06-Mar-2020]. Available: `https://perimeterinstitute.github.io/SymmetryDocs/l`.

[17] V. Bayle, F. Collet, and C. Rovelli, "Short-scale Emergence of Classical Geometry, in Euclidean Loop Quantum Gravity," 2016. [Online; accessed 04-Jul-2020]. Available: `https://arxiv.org/abs/1603.07931`.

[18] P. Donà, M. Fanizza, G. Sarno, and S. Speziale, "SU(2) graph invariants, Regge actions and polytopes," *Class. Quant. Grav.*, vol. 35, no. 4, p. 045011, 2018.

[19] B. Bahr and V. Belov, "Volume simplicity constraint in the Engle-Livine-Pereira-Rovelli spin foam model," *Phys. Rev. D*, vol. 97, no. 8, p. 086009, 2018.

A Numerical Results

The following are the numerical results for selected cases. Looking at equation (26), all of the $B_{v\,\mathrm{comp}}$s should be real. In actuality, however, they all have non-zero complex components. The complex components are the result of a build-up of error when summing over the entries in the arrays. Since the numerically computed values are not exact, the cancellation between the complex conjugates is not perfect. The build-up of error is always at least 2 orders of magnitude less than the desired computed value. If the errors were to become too large, they could be decreased by increasing the number of evaluations when using `cuhre()` to compute the intertwiners.

Spins j	**Result of the contraction** $B_{v\,\mathrm{comp}}$
$[0.5, 0.5, 0.5, 0.5, 0.5, 0.5]$	$6.316\,181 \times 10^{-6} + 1.125\,655 \times 10^{-22}\,\mathrm{i}$
$[1, 1, 1, 1, 1, 1]$	$7.900\,011 \times 10^{-9} + 3.158\,255 \times 10^{-24}\,\mathrm{i}$
$[1.5, 1.5, 1.5, 1.5, 1.5, 1.5]$	$6.782\,086 \times 10^{-11} - 8.310\,682 \times 10^{-19}\,\mathrm{i}$
$[2, 2, 2, 2, 2, 2]$	$2.350\,234 \times 10^{-13} + 8.465\,403 \times 10^{-16}\,\mathrm{i}$

Table 2: Results of the contraction when $j = [x, x, x, x, x, x]$

Spins j	**Result of the contraction** $B_{v\,\mathrm{comp}}$
$[0.5, 0.5, 0.5, 1, 1, 1]$	$9.428\,128 \times 10^{-8} + 1.841\,245 \times 10^{-10}\,\mathrm{i}$
$[0.5, 0.5, 0.5, 1.5, 1.5, 1.5]$	$7.192\,854 \times 10^{-9} + 1.430\,109 \times 10^{-11}\,\mathrm{i}$
$[0.5, 0.5, 0.5, 2, 2, 2]$	$9.815\,433 \times 10^{-10} - 1.765\,282 \times 10^{-12}\,\mathrm{i}$
$[0.5, 0.5, 0.5, 2.5, 2.5, 2.5]$	$1.931\,391 \times 10^{-10} + 3.047\,332 \times 10^{-13}\,\mathrm{i}$

Table 3: Results of the contraction when $j = [0.5, 0.5, 0.5, x, x, x]$

Spins j	**Result of the contraction** $B_{v\,\mathrm{comp}}$
$[1, 1, 1, 0.5, 0.5, 0.5]$	$1.594\,082 \times 10^{-7} + 1.934\,362 \times 10^{-16}\mathrm{i}$
$[1, 1, 1, 1.5, 1.5, 1.5]$	$3.231\,313 \times 10^{-10} - 9.703\,623 \times 10^{-17}\mathrm{i}$
$[1, 1, 1, 2, 2, 2]$	$4.467\,575 \times 10^{-11} + 1.811\,313 \times 10^{-17}\mathrm{i}$
$[1, 1, 1, 2.5, 2.5, 2.5]$	$8.912\,041 \times 10^{-12} + 1.656\,311 \times 10^{-17}\mathrm{i}$

Table 4: Results of the contraction when $j = [1, 1, 1, x, x, x]$

Spins j	**Result of the contraction** $B_{v\,\mathrm{comp}}$
$[1.5, 1.5, 1.5, 0.5, 0.5, 0.5]$	$1.627\,547 \times 10^{-8} - 1.256\,032 \times 10^{-15}\mathrm{i}$
$[1.5, 1.5, 1.5, 1, 1, 1]$	$4.291\,545 \times 10^{-10} - 7.518\,624 \times 10^{-17}\mathrm{i}$
$[1.5, 1.5, 1.5, 2, 2, 2]$	$4.718\,185 \times 10^{-12} - 2.182\,834 \times 10^{-18}\mathrm{i}$
$[1.5, 1.5, 1.5, 2.5, 2.5, 2.5]$	$9.613\,562 \times 10^{-13} - 2.727\,459 \times 10^{-18}\mathrm{i}$

Table 5: Results of the contraction when $j = [1.5, 1.5, 1.5, x, x, x]$

Spins j	**Result of the contraction** $B_{v\,\mathrm{comp}}$
$[1, 0.5, 0.5, 1, 0.5, 0.5]$	$3.076\,342 \times 10^{-7} + 5.612\,957 \times 10^{-10}\mathrm{i}$
$[1.5, 0.5, 0.5, 1.5, 0.5, 0.5]$	$5.418\,473 \times 10^{-8} + 1.178\,762 \times 10^{-10}\mathrm{i}$
$[2, 0.5, 0.5, 2, 0.5, 0.5]$	$1.416\,473 \times 10^{-8} + 3.348\,647 \times 10^{-11}\mathrm{i}$
$[2.5, 0.5, 0.5, 2.5, 0.5, 0.5]$	$4.750\,232 \times 10^{-9} + 1.178\,557 \times 10^{-11}\mathrm{i}$

Table 6: Results of the contraction when $j = [x, 0.5, 0.5, x, 0.5, 0.5]$

Spins j	**Result of the contraction** $B_{v\,\mathrm{comp}}$
$[0.5, 1, 1, 0.5, 1, 1]$	$2.638\,464 \times 10^{-8} + 9.378\,343 \times 10^{-10}\mathrm{i}$
$[1.5, 1, 1, 1.5, 1, 1]$	$3.849\,538 \times 10^{-10} - 2.007\,330 \times 10^{-12}\mathrm{i}$
$[2, 1, 1, 2, 1, 1]$	$1.002\,670 \times 10^{-10} + 5.062\,184 \times 10^{-13}\mathrm{i}$
$[2.5, 1, 1, 2.5, 1, 1]$	$3.356\,026 \times 10^{-11} + 1.625\,479 \times 10^{-13}\mathrm{i}$

Table 7: Results of the contraction when $j = [x, 1, 1, x, 1, 1]$

Spins j	**Result of the contraction** $B_{v\,\mathrm{comp}}$
$[1, 0.5, 0.5, 0.5, 0.5, 0.5]$	$1.036\,231 \times 10^{-6} + 1.526\,222 \times 10^{-9}$ i
$[1.5, 0.5, 0.5, 0.5, 0.5, 0.5]$	$4.292\,017 \times 10^{-7} + 7.064\,235 \times 10^{-10}$i
$[2, 0.5, 0.5, 0.5, 0.5, 0.5]$	$2.170\,337 \times 10^{-7} - 3.767\,604 \times 10^{-10}$i
$[2.5, 0.5, 0.5, 0.5, 0.5, 0.5]$	$1.244\,728 \times 10^{-7} + 2.224\,834 \times 10^{-10}$i

Table 8: Results of the contraction when $j = [x, 0.5, 0.5, 0.5, 0.5, 0.5]$

Spins j	**Result of the contraction** $B_{v\,\mathrm{comp}}$
$[0.5, 1, 1, 1, 1, 1]$	$1.987\,901 \times 10^{-8} - 2.697\,334 \times 10^{-15}$i
$[1.5, 1, 1, 1, 1, 1]$	$2.431\,433 \times 10^{-9} - 1.187\,490 \times 10^{-15}$i
$[2, 1, 1, 1, 1, 1]$	$1.238\,886 \times 10^{-9} + 3.581\,121 \times 10^{-16}$i
$[2.5, 1, 1, 1, 1, 1]$	$7.149\,931 \times 10^{-10} + 1.799\,593 \times 10^{-15}$i

Table 9: Results of the contraction when $j = [x, 1, 1, 1, 1, 1]$

Spins j	**Result of the contraction** $B_{v\,\mathrm{comp}}$
$[0.5, 1.5, 1.5, 1.5, 1.5, 1.5]$	$1.085\,763 \times 10^{-10} - 6.077\,692 \times 10^{-13}$i
$[1, 1.5, 1.5, 1.5, 1.5, 1.5]$	$3.000\,928 \times 10^{-11} - 1.768\,437 \times 10^{-13}$i
$[2, 1.5, 1.5, 1.5, 1.5, 1.5]$	$6.092\,347 \times 10^{-12} - 3.084\,264 \times 10^{-14}$i
$[2.5, 1.5, 1.5, 1.5, 1.5, 1.5]$	$3.470\,428 \times 10^{-12} - 1.627\,477 \times 10^{-14}$i

Table 10: Results of the contraction when $j = [x, 1.5, 1.5, 1.5, 1.5, 1.5]$

B Computational Times

The following are the computational times for creating intertwiners with selected spins and for selected contractions.

Spins $j =$	**Creation Time (s))** $t \approx$
$[0.5, 0.5, 0.5]$	0.050
$[1, 0.5, 0.5]$	0.38
$[1, 1, 1]$	4.9
$[1, 1.5, 1.5]$	10
$[1.5, 1.5, 1.5]$	71
$[2, 2, 2]$	340
$[2.5, 2.5, 2.5]$	1200

Table 11: Time to create one intertwiner with the given spins

Spins $j =$	**Contraction Time (s)** $t \approx$
$[0.5, 0.5, 0.5, 0.5, 0.5, 0.5]$	0.0035
$[1, 1, 1, 1, 1, 1]$	2.5
$[1.5, 1.5, 1.5, 1.5, 1.5, 1.5]$	290
$[2, 2, 2, 2, 2, 2]$	12000

Table 12: Time to compute the contraction when $j = [x, x, x, x, x, x]$

Spins $j =$	**Contraction Time (s)** $t \approx$
$[0.5, 0.5, 0.5, 1, 1, 1]$	0.90
$[0.5, 0.5, 0.5, 1.5, 1.5, 1.5]$	2.9
$[0.5, 0.5, 0.5, 2, 2, 2]$	20
$[0.5, 0.5, 0.5, 2.5, 2.5, 2.5]$	100

Table 13: Time to compute the contraction when $j = [0.5, 0.5, 0.5, x, x, x]$

Spins $j =$	**Contraction Time (s)** $t \approx$
$[1, 1, 1, 0.5, 0.5, 0.5]$	1.5
$[1, 1, 1, 1.5, 1.5, 1.5]$	72
$[1, 1, 1, 2, 2, 2]$	500
$[1, 1, 1, 2.5, 2.5, 2.5]$	2500

Table 14: Time to compute the contraction when $j = [1, 1, 1, x, x, x]$

Spins $j =$	**Contraction Time (s)** $t \approx$
$[1.5, 1.5, 1.5, 0.5, 0.5, 0.5]$	2.4
$[1.5, 1.5, 1.5, 1, 1, 1]$	64
$[1.5, 1.5, 1.5, 2, 2, 2]$	5500
$[1.5, 1.5, 1.5, 2.5, 2.5, 2.5]$	27000

Table 15: Time to compute the contraction when $j = [1.5, 1.5, 1.5, x, x, x]$

Spins $j =$	**Contraction Time (s)** $t \approx$
$[1, 0.5, 0.5, 1, 0.5, 0.5]$	0.19
$[1.5, 0.5, 0.5, 1.5, 0.5, 0.5]$	0.51
$[2, 0.5, 0.5, 2, 0.5, 0.5]$	1.2
$[2.5, 0.5, 0.5, 2.5, 0.5, 0.5]$	2.4

Table 16: Time to compute the contraction when $j = [x, 0.5, 0.5, x, 0.5, 0.5]$

Spins $j =$	**Contraction Time (s)** $t \approx$
$[1.5, 1, 1, 1.5, 1, 1]$	24
$[2, 1, 1, 2, 1, 1]$	68
$[2.5, 1, 1, 2.5, 1, 1]$	160
$[3, 1, 1, 3, 1, 1]$	330

Table 17: Time to compute the contraction when $j = [x, 1, 1, x, 1, 1]$

Spins $j =$	**Contraction Time (s)** $t \approx$
$[1.5, 0.5, 0.5, 0.5, 0.5, 0.5]$	0.027
$[2, 0.5, 0.5, 0.5, 0.5, 0.5]$	0.038
$[2.5, 0.5, 0.5, 0.5, 0.5, 0.5]$	0.055
$[3, 0.5, 0.5, 0.5, 0.5, 0.5]$	0.072

Table 18: Time to compute the contraction when $j = [x, 0.5, 0.5, 0.5, 0.5, 0.5]$

Spins $j =$	**Contraction Time (s)** $t \approx$
$[0.5, 1, 1, 1, 1, 1]$	1.3
$[1.5, 1, 1, 1, 1, 1]$	5.1
$[2, 1, 1, 1, 1, 1]$	8.8
$[2.5, 1, 1, 1, 1, 1]$	14

Table 19: Time to compute the contraction when $j = [x, 1, 1, 1, 1, 1]$

Spins $j =$	**Contraction Time (s)** $t \approx$
$[0.5, 1.5, 1.5, 1.5, 1.5, 1.5]$	45
$[1, 1.5, 1.5, 1.5, 1.5, 1.5]$	130
$[2, 1.5, 1.5, 1.5, 1.5, 1.5]$	520
$[2.5, 1.5, 1.5, 1.5, 1.5, 1.5]$	820

Table 20: Time to compute the contraction when $j = [x, 1.5, 1.5, 1.5, 1.5, 1.5]$

C Code for createIntertwiner

C.1 Code for G()

```
@memoize function fact(j::Int,s::Int,M1::Float64,M2::Float64)
    return Float64(factorial(BigInt((j/2)+M1-s))*
           factorial(BigInt(s))*factorial(BigInt(M2-M1+s))*
           factorial(BigInt((j/2)-M2-s)))
end

@memoize function factCosSin(j::Int,s::Int,M1::Float64,M2::
   Float64,beta::Float64,dfact::Float64)
    return ((-1.0)^(s)*(1.0*im)^(M1-M2))*(dfact)^(-1)*
    cos(beta/2)^(j+M1-M2-2*s)*sin(beta/2)^(M2-M1+2*s)
end

function G(alpha::Float64,beta::Float64,gamma::Float64,j::Int,M
   ::Array{Any,2},d::Array{Complex{Float64},2})

    # Initialize matrices

    length = j+1 # size of matrix

    D = zeros(Complex{Float64},length,length)

    # Create Large D matrix

    Dtemp = zeros(Complex{Float64},length,length)

    for I = 1:length, J= 1:length
        Dtemp[I,J] = cis(-M[I,J][2]*alpha-M[I,J][1]*gamma)
    end

    dee = zeros(Complex{Float64},length,length)
    dee[:,:] = d

    for I = 1:length, J = 1:length
```

```
        SUM = 0

        for s = 0:trunc(Int,j)
            if ((j/2)+M[I,J][1]-s)>=0 && (M[I,J][2]-M[I,J][1]+s)
               >=0 && ((j/2)-M[I,J][2]-s)>=0

                # find the factorial separately
                dfact = fact(j,s,M[I,J][1],M[I,J][2])

                # If the value of s does not cause the
                   factorials to be negative, add the
                # value to the sum

                SUM += factCosSin(j,s,M[I,J][1],M[I,J][2],beta,
                   dfact)

            end

        end

        # Multiply the sum by the square root

        dee[I,J] = dee[I,J]*SUM

        # Multiply the sum by the square root

    end

    # Combine the small d and large D matrices

    for I = 1:length, J = 1:length
        D[I,J] = Dtemp[I,J]*dee[I,J]
    end

    return D
end
```

C.2 Code for `Tensor()`

```
# function to make a tensor with given parameters takes global
   rotation angles, and
# an array of strings that indicate direction and an array of j

export PreTensor
function PreTensor(j::Vector{Int},v::Vector{String},MM::Array{
   Any,1},DD::Array{Any,1})

    # Create a array that contains the length of each index

    L = Array{Int,1}(undef,3)
    for i = 1:3
        L[i] = trunc(Int,(j[i])+1)
    end

    # Find the longest length in L

    long = L[1]
    for i = 2:3
        if L[i] > long
            long = L[i]
        end
    end

    # Initialize and array that will store the parameters and
       an array that will hold
    # the initial z vector for each rotation

    A = zeros(Complex{Float64},long,3)
    z = zeros(Complex{Float64},long,3)

    for i = 1:3
        z[1,i] = 1.0 + 0*im
    end

    # Rotate the z vectors to match the input v and store in A.
```

```
    for i = 1:3
        if v[i] == "x"
            A[1:L[i],i] = G(-pi/2,pi/2,pi/2,j[i],MM[i],DD[i]) *
               z[1:L[i],i]
        elseif v[i] == "minusx"
            A[1:L[i],i] = G(-pi/2,-pi/2,pi/2,j[i],MM[i],DD[i]) *
                z[1:L[i],i]
        elseif v[i] == "y"
            A[1:L[i],i] = G(0.0,-pi/2,0.0,j[i],MM[i],DD[i]) * z
               [1:L[i],i]
        elseif v[i] =="minusy"
            A[1:L[i],i] = G(0.0,pi/2,0.0,j[i],MM[i],DD[i]) * z
               [1:L[i],i]
        elseif v[i] == "z"
            A[1:L[i],i] = z[1:L[i],i]
        elseif v[i] == "minusz"
            A[1:L[i],i] = G(0.0,pi*1.0,pi*1.0,j[i],MM[i],DD[i])
               * z[1:L[i],i]
        else
            return error("InputError: Incorrect input, input
               must be x, y, z, minusx, minusy, or minusz")
        end
    end
    return A
end

export Tensor
function Tensor(alpha::Float64,beta::Float64,gamma::Float64,j::
   Vector{Int},
        MM::Array{Any,1},DD::Array{Any,1},A::Array{Complex{
           Float64},2})

        # Create a array that contains the length of each index

        L = Array{Int,1}(undef,3)
        for i = 1:3
            L[i] = trunc(Int,(j[i])+1)
```

```
        end

        # Create a table that will store the tensor values

        tbl = zeros(Complex{Float64},L[1],L[2],L[3],L[1],L[2],L
            [3]) # store the tensor as a 6D matrix

        # Calculate each of the wigner matrices

        G1 = G(alpha,beta,gamma,j[1],MM[1],DD[1])
        G1con = conj(transpose(G1))
        G2 = G(alpha,beta,gamma,j[2],MM[2],DD[2])
        G2con = conj(transpose(G2))
        G3 = G(alpha,beta,gamma,j[3],MM[3],DD[3])
        G3con = conj(transpose(G3))

        # pre-calculate the vectors

        v1 = (G1*A[1:L[1],1])
        v2 = (G2*A[1:L[2],2])
        v3 = (G3*A[1:L[3],3])
        v4 = ((A[1:L[1],1]')*G1con)
        v5 = (((A[1:L[2],2])')*G2con)
        v6 = ((A[1:L[3],3]')*G3con)

        for i = 1:L[1], J = 1:L[2], k = 1:L[3], l = 1:L[1], m =
             1:L[2], n = 1:L[3]

            # Create the tensor by taking the product of g and a
                given column of A

            tbl[i,J,k,l,m,n] = v1[i]*v2[J]*v3[k]*v4[l]*v5[m]*v6[
               n]

        end
        return tbl
end
```

C.3 Code for `Intertwiner()` and `srt()`

```
# Kernel for parallelization
function kernel(L1::Int,Inds1::Array{CartesianIndex{6},1},jay::
   Array{Int,1},MM::Array{Any,1},DD::Array{Any,1},A::Array{
   Complex{Float64},2})

    i = Inds1[L1][1]
    j = Inds1[L1][2]
    k = Inds1[L1][3]
    l = Inds1[L1][4]
    m = Inds1[L1][5]
    n = Inds1[L1][6]

    # integrate the integrand function and store in tbl

    result,err = cuhre((x, f) -> (f[1],f[2]) = reim((1/(8*(pi
       ^2)))*sin(pi*x[2])*4*(pi^3)*
       Tensor((2*pi*x[1]),(pi*x[2]),(2*pi*x[3]),jay,MM,DD,A)[i
          ,j,k,l,m,n]),3,2,minevals = 1e2,maxevals=1e4)
    return result[1]+result[2]*im

end

# Intertwiner() takes vectors as input and skips zeros and is
   parallelized
function Intertwiner(V::Array{String,1},Jay::Array{Int,1},MM::
   Array{Any,1},DD::Array{Any,1})
    # Create array of lengths of each index

    lngth = Array{Int,1}(undef,3)

    for i = 1:3
        lngth[i] = trunc(Int,(Jay[i])+1)
    end

    tbl1 = zeros(lngth[1],lngth[2],lngth[3],lngth[1],lngth[2],
       lngth[3])
```

```
# store parameters as global variables so they can be
   called by the integrand function

v = V
jay = Jay

A = PreTensor(jay,v,MM,DD)

for i = 1:lngth[1], j = 1:lngth[2], k = 1:lngth[3], l = 1:
   lngth[1], m = 1:lngth[2]
    for n = 1:lngth[3]

        tbl1[i,j,k,l,m,n] = -(MM[1][i][1]+MM[2][j][1]+MM[3][
           k][1])+MM[1][l][1]+MM[2][m][1]+MM[3][n][1]

    end
end

Inds1 = findall(x->x==0,tbl1)

tbl = zeros(Complex{Float64},lngth[1],lngth[2],lngth[3],
   lngth[1],lngth[2],lngth[3]) # Store the intertwiner in a
    6D matrix

ftbl = Array{Future}(undef, size(tbl))

for L1 = 1:length(Inds1)
       i = Inds1[L1][1]
       j = Inds1[L1][2]
       k = Inds1[L1][3]
       l = Inds1[L1][4]
       m = Inds1[L1][5]
       n = Inds1[L1][6]
       p = workers()[mod1(L1, nworkers())]
       ftbl[i,j,k,l,m,n] = remotecall(kernel, p, L1,Inds1,
          jay,MM,DD,A) # remember to use @everywhere before
           include
```

```
    end

    for L1 = 1:length(Inds1)
        i = Inds1[L1][1]
        j = Inds1[L1][2]
        k = Inds1[L1][3]
        l = Inds1[L1][4]
        m = Inds1[L1][5]
        n = Inds1[L1][6]
        tbl[i,j,k,l,m,n] = fetch(ftbl[i,j,k,l,m,n])
    end

    return tbl

end

function srt(arr::Array{Complex{Float64},6},i1::Int,i2::Int,i3
    ::Int)

    dim = size(arr)

    newArr = zeros(Complex{Float64}, dim[1],dim[2],dim[3],dim
        [4],dim[5],dim[6])

    for i = 1:dim[1], j = 1:dim[2], k = 1:dim[3], l = 1:dim[4],
         m = 1:dim[5], n = 1:dim[6]

        if i1==1 && i2==1 && i3==1
            newArr[i,j,k,l,m,n] = arr[i,j,k,l,m,n]
        elseif i1==2 && i2==1 && i3==1
            newArr[i,j,k,l,m,n] = arr[l,j,k,i,m,n]
        elseif i1==2 && i2==2 && i3==1
            newArr[i,j,k,l,m,n] = arr[l,m,k,i,j,n]
        elseif i1==2 && i2==2 && i3==2
            newArr[i,j,k,l,m,n] = arr[l,m,n,i,j,k]
        elseif i1==1 && i2==2 && i3==2
            newArr[i,j,k,l,m,n] = arr[i,j,k,l,m,n]
        elseif i1==1 && i2==1 && i3==2
```

```
            newArr[i,j,k,l,m,n] = arr[i,j,n,l,m,k]
        elseif i1==2 && i2==1 && i3==2
            newArr[i,j,k,l,m,n] = arr[l,j,n,i,m,k]
        elseif i1==1 && i2==2 && i3==1
            newArr[i,j,k,l,m,n] = arr[i,m,k,l,j,n]
        else
            return print("error")
        end

    end

    return newArr
end
```

D Code for Contraction

D.1 Code for contracting using the rule for finding zeros

```
function Contraction(Int1::Array{Complex{Float64}},Int2::Array{
   Complex{Float64}},Int3::Array{Complex{Float64}},Int4::Array
   {Complex{Float64}},Int5::Array{Complex{Float64}},Int6::
   Array{Complex{Float64}},Int7::Array{Complex{Float64}},int8
   ::Array{Complex{Float64}})

    SUM = 0 # Initialize the result

    # Find the dimensions of each of the intertwiners

    dim1 = size(Int1)
    dim2 = size(Int2)
    dim3 = size(Int3)
    dim4 = size(Int4)
    dim5 = size(Int5)
    dim6 = size(Int6)
    dim7 = size(Int7)

    Mmg1 = magIndices(dim1[2])
    Mmg2 = magIndices(dim1[3])
    Mmg3 = magIndices(dim1[1])
    Mmg4 = magIndices(dim3[2])
    Mmg5 = magIndices(dim2[2])
    Mmg6 = magIndices(dim3[1])

    # Loop over the given indices (all indices except a3, b2,
       and c1)

    for c4 = 1:dim7[4],c2 = 1:dim5[3],c1 = 1:dim5[2],b6 = 1:
       dim4[6],b5 = 1:dim4[4],
        b3 = 1:dim3[5],b2 = 1:dim3[1],b1 = 1:dim3[3],a10 = 1:
```

```
   dim2[6],a9 = 1:dim2[5],
a8 = 1:dim2[3],a7 = 1:dim2[2],a5 = 1:dim1[6],a4 = 1:
   dim1[5],a3 = 1:dim1[4],
a2 = 1:dim1[3],a1 = 1:dim1[2] # last index should be
   first called and vice versa

Con1 = -Mmg1[a1] - Mmg2[a2] + Mmg3[a3] + Mmg1[a4] +
   Mmg2[a5]

Jay = ((dim1[1]-1)/2) # Value of j at index
   corresponging to A3

# If the value of A3 is allowed, convert to index
   notation and find B2
if abs(Con1) <= Jay && abs(Con1-floor(Con1)) == Jay-
   floor(Jay)

   y0 = 1+Jay

   con1 = Int(-Con1+y0)

   # a6 = a7 - a8 + con1 - a9 + a10

   A6 = Mmg5[a7] - Mmg6[a8] + Mmg3[con1] - Mmg5[a9] +
      Mmg6[a10]

   Jay = ((dim2[1]-1)/2)

   # If the value of A6 is allowed, convert to index
      notation and find Con2
   if abs(A6) <= Jay && abs(A6-floor(A6)) == Jay-floor(
      Jay)

      y0 = 1+Jay

      a6 = Int(-A6+y0)

      # con2 = a8 - b1 - b2 + b3 + a2
```

```
Con2 = Mmg6[a8] - Mmg2[b1] - Mmg6[b2] + Mmg4[b3]
    + Mmg2[a2]

Jay = ((dim3[2]-1)/2)

# If the value of Con2 is allowed, convert to
   index notation and find B4
if abs(Con2) <= Jay && abs(Con2-floor(Con2)) ==
   Jay-floor(Jay)

   y0 = 1+Jay

   con2 = Int(-Con2+y0)

   # b4 = -a7 - b6 + b5 + a1 - con2

   B4 = -Mmg5[a7] - Mmg4[b6] + Mmg5[b5] + Mmg1[
      a1] + Mmg4[con2]

   Jay = ((dim4[2]-1)/2)

   # If the value of B4 is allowed, convert to
      index notation and find Con3
   if abs(B4) <= Jay && abs(B4-floor(B4)) == Jay
      -floor(Jay)

      y0 = 1+Jay

      b4 = Int(-B4+y0)

      # con3 = -c1 - c2 + a6 + b4 + b1

      Con3 = -Mmg1[c1] - Mmg2[c2] + Mmg3[a6] +
         Mmg1[b4] + Mmg2[b1]

      Jay = ((dim5[1]-1)/2)
```

```
# If the value of Con3 is allowed,
   convert to index notation and find C4
if abs(Con3) <= Jay && abs(Con3-floor(
   Con3)) == Jay-floor(Jay)

   y0 = 1+Jay

   con3 = Int(-Con3+y0)

   # c3 = a3 + b5 + c4 - con3 - b2

   C3 = Mmg3[a3] + Mmg5[b5] + Mmg6[c4] -
       Mmg3[con3] - Mmg6[b2]

   Jay = ((dim6[5]-1)/2)

   # If the value of C3 is allowed,
      convert to index notation and find
       Con4
   if abs(C3) <= Jay && abs(C3-floor(C3)
      ) == Jay-floor(Jay)

      y0 = 1+Jay

      c3 = Int(-C3+y0)

      # con4 = -a10 - a5 + c4 + b6 + c2

      Con4 = -Mmg6[a10] - Mmg2[a5] +
         Mmg6[c4] + Mmg4[b6] + Mmg2[c2]

      Jay = ((dim7[2]-1)/2)

      # If the value of C3 is allowed,
         convert to index notation and
         contract
      if abs(Con4) <= Jay && abs(Con4-
         floor(Con4)) == Jay-floor(Jay)
```

```
                                    y0 = 1+Jay

                                    con4 = Int(-Con4+y0)

                                    SUM += Int1[con1,a1,a2,a3,a4,
                                       a5]*Int2[a6,a7,a8,con1,a9,
                                       a10]*
                                        Int3[a8,con2,b1,b2,b3,a2]*
                                           Int4[a7,b4,con2,b5,a1,
                                           b6]*Int5[a6,c1,b1,con3,
                                           b4,c2]*
                                        Int6[a3,b5,b2,con3,c3,c4]*
                                           Int7[a10,b6,a5,c4,con4,
                                           c2]*int8[a9,a4,b3,c3,c1
                                           ,con4]

                                end

                            end

                        end

                    end

                end

            end

        end

    end

    return(SUM)

end
```

D.2 Code for contracting in steps

```
module performContraction

using LinearAlgebra

export zr
function zr(num::Float64)
    if abs(num) < 2*(10^-7)
        return 0
    else
        return num
    end
end

export magIndices
function magIndices(dim::Int)
    J = (dim-1)/2
    m = -J:J
    return -m
end

export contract1
function contract1(inter1,inter2)

    # Find size dimensions of the intertwiners and create a new
        array to hold the contraction

    dim1 = size(inter1)
    dim2 = size(inter2)

    arr = complex(zeros(dim1[2],dim1[3],dim1[4],dim1[5],dim1
        [6],
                dim2[1],dim2[2],dim2[3],dim2[5],dim2[6]))

    # Contract over the index "con"
```

```
Mm3 = magIndices(dim1[3])
Mm4 = magIndices(dim1[4])
Mm5 = magIndices(dim1[5])
Mm6 = magIndices(dim1[6])
Mm7 = magIndices(dim2[1])
Mm8 = magIndices(dim2[2])
Mm9 = magIndices(dim2[3])
Mm11 = magIndices(dim2[5])
Mm12 = magIndices(dim2[6])

for i3 = 1:dim1[3], i4 = 1:dim1[4], i5 = 1:dim1[5], i6 = 1:
   dim1[6],
    i7 = 1:dim2[1], i8 = 1:dim2[2], i9 = 1:dim2[3], i11 =
       1:dim2[5], i12 = 1:dim2[6]

    # Set the index I2 (a2 in working by hand) equal to the
        sum of the other magnetic indices

    I2 = -(Mm3[i3]-(Mm4[i4]+Mm5[i5]+Mm6[i6])+Mm7[i7]+Mm11[
       i11]+
       Mm9[i9]-(Mm8[i8]+Mm12[i12]))

    # Set the the index i2 equal to the correct value in
       code notation and perform contraction over the index
        con

    Jay = ((dim1[2]-1)/2)

    if abs(I2) <= Jay && abs(I2-floor(I2))== Jay-floor(Jay)

        y0 = 1+Jay

        i2 = Int(-I2+y0)

        for con = 1:dim1[1]

            arr[i2,i3,i4,i5,i6,i7,i8,i9,i11,i12] += (inter1[
               con,i2,i3,i4,i5,i6]*inter2[i7,i8,i9,con,i11,
```

```
                i12])

            end

        end

    end

    return arr
end

export contract2
function contract2(inter3,inter4)

    # Find size dimensions of the intertwiners and create a new
        array to hold the contraction

    dim1 = size(inter3)
    dim2 = size(inter4)

    arr = complex(zeros(dim1[2],dim1[3],dim1[4],dim1[5],dim1
        [6],
                dim2[1],dim2[2],dim2[3],dim2[5],dim2[6]))

    # Contract over the index "con"

    Mmc3 = magIndices(dim1[3])
    Mmc4 = magIndices(dim1[4])
    Mmc5 = magIndices(dim1[5])
    Mmc6 = magIndices(dim1[6])
    Mmd1 = magIndices(dim2[1])
    Mmd2 = magIndices(dim2[2])
    Mmd4 = magIndices(dim2[4])
    Mmd5 = magIndices(dim2[5])
    Mmd6 = magIndices(dim2[6])

    for c3=1:dim1[3], c4 = 1:dim1[4], c5 = 1:dim1[5], c6 = 1:
        dim1[6],
```

```
        d1 = 1:dim2[1], d2 = 1:dim2[2], d4 = 1:dim2[4], d5 = 1:
           dim2[5], d6 = 1:dim2[6]

        # Set the index I2 (a2 in working by hand) equal to the
            sum of the other magnetic indices

        C1 = Mmc3[c3]+(Mmc4[c4]-Mmc5[c5]-Mmc6[c6])+Mmd1[d1]+
           Mmd2[d2]+
            Mmd6[d6]-Mmd4[d4]-Mmd5[d5]

        # Set the the index i2 equal to the correct value in
           code notation and perform contraction over the index
            con

        Jay = ((dim1[1]-1)/2)

        if abs(C1) <= Jay && abs(C1-floor(C1))== Jay-floor(Jay)

            y0 = 1+Jay

            c1 = Int(-C1+y0)

            for con = 1:dim1[2]

                arr[c1,c3,c4,c5,c6,d1,d2,d4,d5,d6] += (inter3[c1
                   ,con,c3,c4,c5,c6]*inter4[d1,d2,con,d4,d5,d6])

            end

        end

    end

    return arr
end

export contract3
function contract3(inter5,inter6)
```

```
# Find size dimensions of the intertwiners and create a new
    array to hold the contraction

dim1 = size(inter5)
dim2 = size(inter6)

arr = complex(zeros(dim1[2],dim1[3],dim1[4],dim1[5],dim1
   [6],
           dim2[1],dim2[2],dim2[3],dim2[5],dim2[6]))

# Contract over the index "con"

Mme2 = magIndices(dim1[2])
Mme3 = magIndices(dim1[3])
Mme5 = magIndices(dim1[5])
Mme6 = magIndices(dim1[6])
Mmf1 = magIndices(dim2[1])
Mmf2 = magIndices(dim2[2])
Mmf3 = magIndices(dim2[3])
Mmf5 = magIndices(dim2[5])
Mmf6 = magIndices(dim2[6])

for e2 = 1:dim1[2], e3 = 1:dim1[3], e5 = 1:dim1[5], e6 = 1:
   dim1[6], f1 = 1:dim2[1],
    f2 = 1:dim2[2], f3 = 1:dim2[3], f5 = 1:dim2[5], f6 = 1:
       dim2[6]

    # Set the index I2 (a2 in working by hand) equal to the
        sum of the other magnetic indices

    E1 = Mme2[e2]-Mme3[e3]-Mme5[e5]+Mme6[e6]+
        Mmf1[f1]+Mmf2[f2]-Mmf3[f3]-Mmf5[f5]+Mmf6[f6]

    # Set the the index i2 equal to the correct value in
       code notation and perform contraction over the index
        con
```

```
        Jay = ((dim1[1]-1)/2)

        if abs(E1) <= Jay && abs(E1-floor(E1))== Jay-floor(Jay)

            y0 = 1+Jay

            e1 = Int(-E1+y0)

            for con = 1:dim1[4]

            arr[e1,e2,e3,e5,e6,f1,f2,f3,f5,f6] += (inter5[e1,e2,
               e3,con,e5,e6]*inter6[f1,f2,f3,con,f5,f6])

            end

        end

    end

    return arr
end

export contract4
function contract4(inter7,inter8)

    # Find size dimensions of the intertwiners and create a new
        array to hold the contraction

    dim1 = size(inter7)
    dim2 = size(inter8)

    arr = complex(zeros(dim1[2],dim1[3],dim1[4],dim1[5],dim1
       [6],
               dim2[1],dim2[2],dim2[3],dim2[5],dim2[6]))

    # Contract over the index "con"

    Mmg2 = magIndices(dim1[2])
```

```
Mmg3 = magIndices(dim1[3])
Mmg4 = magIndices(dim1[4])
Mmg6 = magIndices(dim1[6])
Mmh1 = magIndices(dim2[1])
Mmh2 = magIndices(dim2[2])
Mmh3 = magIndices(dim2[3])
Mmh4 = magIndices(dim2[4])
Mmh5 = magIndices(dim2[5])

for g2 = 1:dim1[2], g3 = 1:dim1[3], g4 = 1:dim1[4], g6 = 1:
   dim1[6], h1 = 1:dim2[1],
    h2 = 1:dim2[2], h3 = 1:dim2[3], h4 = 1:dim2[4], h5 = 1:
       dim2[5]

    # Set the index I2 (a2 in working by hand) equal to the
        sum of the other magnetic indices

    G1 = Mmg2[g2]-Mmg3[g3]+Mmg4[g4]+Mmg6[g6]+
        Mmh1[h1]-Mmh2[h2]-Mmh3[h3]-Mmh4[h4]+Mmh5[h5]

    # Set the the index i2 equal to the correct value in
       code notation and perform contraction over the index
        con

    Jay = ((dim1[1]-1)/2)

    if abs(G1) <= Jay && abs(G1-floor(G1))== Jay-floor(Jay)

        y0 = 1+Jay

        g1 = Int(-G1+y0)

        for con = 1:dim1[4]

        arr[g1,g2,g3,g4,g6,h1,h2,h3,h4,h5] += (inter7[g1,g2,
           g3,g4,con,g6]*inter8[h1,h2,h3,h4,h5,con])

        end
```

```
        end

    end

    return arr
end

function stepContract(Int1::Array{Complex{Float64}},Int2::Array
   {Complex{Float64}},Int3::Array{Complex{Float64}},Int4::
   Array{Complex{Float64}},Int5::Array{Complex{Float64}},Int6
   ::Array{Complex{Float64}},Int7::Array{Complex{Float64}},
   int8::Array{Complex{Float64}})

    # Round the intertwiners to remove entries that are close
       to zero.

    Int1new = zr.(real.(Int1))+zr.(imag(Int1))*im
    Int2new = zr.(real.(Int2))+zr.(imag(Int2))*im
    Int3new = zr.(real.(Int3))+zr.(imag(Int3))*im
    Int4new = zr.(real.(Int4))+zr.(imag(Int4))*im
    Int5new = zr.(real.(Int5))+zr.(imag(Int5))*im
    Int6new = zr.(real.(Int6))+zr.(imag(Int6))*im
    Int7new = zr.(real.(Int7))+zr.(imag(Int7))*im
    Int8new = zr.(real.(int8))+zr.(imag(int8))*im

    # Perform the contraction between each pair of intertwiners

    Int12 = contract1(Int1new,Int2new)
    Int34 = contract2(Int3new,Int4new)
    Int56 = contract3(Int5new,Int6new)
    Int78 = contract4(Int7new,Int8new)

    # Find size dimensions of the intertwiners and create
       variable "sum" to hold the contraction

    dim1 = size(Int12)
    dim2 = size(Int34)
```

```
    dim3 = size(Int56)
    dim4 = size(Int78)

    SUM = 0

    # Perform the contraction over all remaining indices

    for a1 = 1:dim1[1],a2 = 1:dim1[2],a3 = 1:dim1[3],a4 = 1:
       dim1[4],a5 = 1:dim1[5],a6 = 1:dim1[6],a7 = 1:dim1[7],
        a8 = 1:dim1[8],a9 = 1:dim1[9],a10 = 1:dim1[10],b1 = 1:
           dim2[2],b2 = 1:dim2[3],b3 = 1:dim2[4],b4=1:dim2[7],
        b5 = 1:dim2[8],b6 = 1:dim2[10],c1 = 1:dim3[2],c2 = 1:
           dim3[5],c3 = 1:dim3[9],c4 = 1:dim3[10]

        SUM += Int12[a1,a2,a3,a4,a5,a6,a7,a8,a9,a10]*Int34[a8,
           b1,b2,b3,a2,a7,b4,b5,a1,b6]*
            Int56[a6,c1,b1,b4,c2,a3,b5,b2,c3,c4]*Int78[a10,b6,a5
               ,c4,c2,a9,a4,b3,c3,c1]

    end

    return SUM
end
```

D.3 Unoptimized code for performing contraction

```
function contractOld()

    sum = 0

    for i1 = 1:2, i2 = 1:2, i3 = 1:2, i4 = 1:2, i5 = 1:2, i6 =
       1:2, i7 = 1:2, i8 = 1:2, i9 = 1:2,
        i10 = 1:2, i11 = 1:2, i12 = 1:2, i13 = 1:2, i14 = 1:2,
           i15 = 1:2, i16 = 1:2, i17 = 1:2,
        i18 = 1:2, i19 = 1:2, i20 = 1:2, i21 = 1:2, i22 = 1:2,
           i23 = 1:2, i24 = 1:2

        sum += (Int1[i1,i2,i3,i4,i5,i6]*Int2[i7,i8,i9,i1,i10,
           i11]*Int3[i11,i12,i6,i13,i14,i15]*
               Int4[i8,i16,i17,i18,i2,i12]*Int5[i19,i16,i20,i7,
                  i21,i15]*Int6[i4,i18,i22,i19,i23,i13]*
               Int7[i9,i17,i20,i22,i24,i3]*Int8[i10,i5,i24,i23,
                  i21,i14])

    end

    return sum

end
```

E Code for `toTxt()` and `fromTxt()`

```
function toTxt(inter::Array{Complex{Float64},6},name::String)

    dims = size(inter)
    inter = string.(inter)

    io = open(name,"w") # open file named "name"

    # write entries of intertwiner to the file

    for i = 1:dims[1],j = 1:dims[2],k = 1:dims[3],l=1:dims[4],m
       =1:dims[5],n=1:dims[6]
        write(io,inter[i,j,k,l,m,n])
        write(io, " ")
    end

    close(io) # close the file
end

function fromTxt(name::String,j::Array{Int})

    io = open(name,"r") # open file named "name"
    s = read(io,String) # save file contents as string s
    close(io) # close the file

        # split the string into a vector of strings of
           individual numbers

    s = split(s)
    s = String.(s)
    sz = Int(size(s)[1]/3)
    s = reshape(s,3,sz)
    s = s[1,:].*s[2,:].*s[3,:]

        # convert the strings to complex numbers
```

```
    vec = [parse(Complex{Float64},ss) for ss in s]

        # arrange the vector into an array of the appropriate
           shape given by j

    arr = permutedims(reshape(vec,j[3],j[2],j[1],j[3],j[2],j
       [1]),[6,5,4,3,2,1])
    return arr
end
```

F Sample Batch File

```
#!/bin/bash

#SBATCH -p debugq # Indicate the partition the job should run
   on
#SBATCH --time=1:00:00 # Approximate time the job will take
#SBATCH --nodes=1 # Number of compute nodes the job will
   require
#SBATCH --job-name=test # Name of the job
#SBATCH --output=/home/callen/test.out # Outputs will be saved
   to this file
#SBATCH --error=/home/callen/Trials1Same/test.err # Errors will
    be saved to this file

# Output to indicate the job has started
echo "Starting"
hostname
date

module load julia

# The running the job
echo "Starting Simulation"

time julia test.jl

# Output to indicate the job has stopped
echo "Stopping"

# Output to indicate all jobs are done
echo "Done."
date
```