2021

Using Numerical Relativity to Explore Strong Gravity and Develop Force-Free Electrodynamics Simulation Software with Best-Practice Development

Patrick E. Nelson
West Virginia University, penelson@mix.wvu.edu

Using Numerical Relativity to Explore Strong Gravity and Develop Force-Free Electrodynamics Simulation Software with Best-Practice Development

Patrick E. Nelson, B.S.

Dissertation submitted
to the Eberly College of Arts and Sciences
at West Virginia University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in
Physics

Zachariah B. Etienne, Ph.D., Chair
Adam Halasz, Ph.D.
Maura A. McLaughlin Ph.D.
Sean T. McWilliams, Ph.D.

Department of Physics and Astronomy

Morgantown, West Virginia
2021

Keywords: numerical analysis, general relativity, gravitational waves, metaprogramming, GRFFE

ABSTRACT

Using Numerical Relativity to Explore Strong Gravity and Develop Force-Free Electrodynamics Simulations with Best-Practice Software Development

Patrick E. Nelson

In this dissertation, we explore the effects of extremely strong gravitational and electrodynamic fields using the techniques of numerical relativity. We use the existing black hole simulation software in the Einstein Toolkit to compute the spin-up of two initially nonspinning black holes as they pass by each other in space. The angular momentum is imparted by the tidal interaction between the two black holes, in a parallel to Earth's tides, as described by classical mechanics, which also transfer angular momentum between the Earth's rotation and the Moon's orbit. The largest observed dimensionless spin observed was 0.20 with an initial boost of $0.78c$, and we conclude that higher spin-ups may be possible with larger initial boosts.

We also use NRPy+ to generate C code for new simulations. The first two are written as thorns, or modules, for the Einstein Toolkit (ETK). The first is a simple scalar wave solver, which has proven useful as an introduction to NRPy+ for new users who wish to write their own ETK thorns. The second is an open-source alternative to the Kranc-generated `WeylScal4`, which calculates the Weyl scalars that are needed to extract gravitational wave information from a simulation. We also port `GiRaFFE` to NRPy+. While this was originally intended to be another ETK thorn, as the project progressed, it ultimately became a standalone simulation. The individual modules of this code also have their own unit tests as an additional validation step. These unit tests also use continuous integration to ensure that bugs are not unknowingly introduced into the code. In the future, this code will be modified to be able to use arbitrary coordinate systems.

Table of Contents

Acknowledgements

I would like to thank my advisor, Dr. Zachariah B. Etienne, for guiding me forward in my research and constantly pushing me to improve. I would also like to thank Dr. Sean T. McWilliams for his guidance, especially for the analysis in my first project. I am also grateful to Drs. Mew-Bing Wan and Maria C. Babiuc for their advice as we collaborated.

I would also like to thank my family, especially my mother, who copyedited this dissertation.

I also owe a tremendous debt to the many other teachers I have had over the years, whose lessons ultimately form the basis of this research, as well as the doctors and nurses who have overseen my care these past few months.

Chapter 1

Introduction and Motivation

1.1 A Brief History of Gravity

Sir Isaac Newton's theories of motion and gravity proved to be quite successful; for centuries, they provided accurate predictions of the motions of planets and the ocean's tides. Newtonian mechanics represented an important step forward in human thinking; they introduced the key concept of an inertial frame. Experiments carried out in an inertial frame yield the same results, and measurements from one inertial frame can be easily transformed into their counterparts taken in another frame as long as the relative velocity between the two frames is known. This is known as the principle of relativity. For example, if two inertial observers were to observe the same experiment in which two masses collided inelastically with each other, they might disagree on what the momenta of the objects were, but they will always agree that total momentum was conserved. If either observer knows the velocity of the other observer relative to themselves, they will be able to calculate what the other observed for the momenta of the objects.

However, as humanity tested these predictions to ever-increasing precision, holes began to appear in these theories. One particular issue that arose was the precession of the perihelion of Mercury; that is, the point in Mercury's orbit at which it is closest to the Sun shifts slightly with each orbit. Although a small shift was expected because of the gravitational influence of the other planets, the measured precession was still greater than what was predicted by accounting for the slight tugs from each of the other planets.

The behavior of light also differed from what the Newtonian theories predicted. Because light displays many wave-like properties, physicists expected that it would display others. Namely, they supposed that light must travel through a medium (dubbed the luminiferous aether), and that its measured speed would change depending on the motion of an observer through that medium. However, these assumptions would not hold as another blow was dealt to the Newtonian theories by the Michelson-Morley experiment. By building a large interferometer, Michelson and Morley hoped to measure a difference in the speed of light traveling in two perpendicular directions. As the earth orbited the Sun throughout the year, the directions of the arms and their speeds relative to the luminiferous aether would change, revealing the properties of the aether. Instead, they found that the speed of light was invariant in reference frames.

It was by considering these two ideas—the principle of relativity and the invariance of the speed of light in a vacuum—as postulates that Albert Einstein discovered the special theory of relativity. This theory leads to some strange results; we find that space and time are not fixed. For example, suppose a scientist sets up an experiment in which a train moves past a train platform at extremely high speeds. An observer on the platform would measure the length of the train to be shorter than it was while the train was stationary (known as length contraction) and would note that clocks on the train tick more slowly than those on the platform (known as time dilation).

Later, Einstein managed to generalize this theory to include gravity by considering the equivalence principle. According to this postulate, there is no difference to an observer between the acceleration of an object due to gravity and an apparent acceleration caused by an accelerating reference frame. The usual thought experiment here is that of a scientist dropping an object in a sealed box. If the scientist measures the acceleration of that dropped object and finds that it is 9.8 $\mathrm{m/s^2}$ (i.e., the usual acceleration due to gravity on Earth's surface), they cannot tell if they are stationary on Earth's surface or in a rocket ship in deep space accelerating at 9.8 $\mathrm{m/s^2}$.

Ultimately, this led Einstein to the theory of general relativity (GR), in which space and time are not separate, but parts of a whole, which we call spacetime. This spacetime can be warped and bent by matter and energy, and this curvature in spacetime influences the

motions of objects. In view of this, objects orbiting the planet are simply moving in straight lines through curved spacetime. Additionally, GR accurately predicts that large masses such as the Sun will bend their paths.

This curvature of spacetime is represented by the Einstein equations,

$$G_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu}.$$

Here, $T_{\mu\nu}$ is the stress-energy tensor; its form will vary depending on the system in question. It can encode pressures, densities, and electromagnetic fields; in empty space and black hole spacetimes, it is simply zero. The Einstein tensor $G_{\mu\nu}$ encodes information about the curvature of spacetime.

These theories accurately explained several of the issues that bothered physicists, such as precession of the perihelion. The theories also produced several startling predictions. In eccentric orbits close to a black hole, the peribothron can precess so far during a single orbit that the orbiting particle traces out clover-leaf-shaped patterns. In a parallel to the electromagnetic waves produced by accelerating charges, explained by Maxwell's equations, accelerating masses (such as two orbiting objects) were predicted to radiate gravitational waves. The effect was first confirmed by noting the decreasing orbital period of pulsars [1]. They were then directly observed by the Laser Interferometer Gravitational-Wave Observatory (LIGO) in 2015 [2]. Later, LIGO observed the gravitational signal of two neutron stars coalescing, which was also observed by observatories around the world and across the electromagnetic spectrum, heralding the beginning of the era of multimessenger astronomy and highlighting the importance of numerical simulations in interpreting these observations [3].

1.2 General Relativity

In this dissertation, we will use geometrized units. In this system, we set Newton's gravitational constant and the speed of light $G = c = 1$, allowing us to express lengths, times, and masses using the same units. This helps illustrate the interconnectedness of time and space and allows us to more easily write equations. Einstein's equations become

$$G_{\mu\nu} = 8\pi T_{\mu\nu}.$$

We have already referred to $G_{\mu\nu}$ and $T_{\mu\nu}$ as tensors; a tensor is similar to a matrix, except a tensor can have a rank other than two (i.e., it can have more indices) and a tensor must obey specific transformation laws.

1.2.1 Einstein Notation

It is also important to note the indices in these equations. Greek indices represent spacetime quantities and run from 0 to 4, where 0 represents the time component of a vector or tensor, and the numbers 1 through 3 represent spatial components. Latin indices represent purely spatial quantities and run from 1 to 3. Repeated indices imply summation; for example, the inner product of the electric and magnetic field vectors is written as

$$E_\mu B^\mu = E_0 B^0 + E_1 B^1 + E_2 B^2 + E_3 B^3.$$

A particularly important tensor is called the metric tensor $g_{\mu\nu}$; as the name implies, it encodes information about how we measure distances. Recall that in Euclidean geometry, the distance element in Cartesian coordinates can be written as $ds^2 = dx^2 + dy^2 + dz^2$. In special relativity, we must also consider the relationship between space and time; the length element in flat spacetime is written as $ds^2 = -dt^2 + dx^2 + dy^2 + dz^2$. This corresponds to a metric tensor (often written as $\eta_{\mu\nu}$ in this special case) of

$$\eta_{\mu\nu} = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

In general spacetimes and arbitrary coordinate systems, a metric tensor $g_{\mu\nu}$ corresponds to a length element of $ds^2 = g_{\mu\nu} dx^\mu dx^\nu$. If we write the summation out explicitly, it becomes

$$ds^2 = \sum_{\mu=0}^{3} \sum_{\nu=0}^{3} g_{\mu\nu} dx^\mu dx^\nu.$$

Here, dx^μ is a vector of differentials

$$dx^\mu = \begin{pmatrix} dx^0 \\ dx^1 \\ dx^2 \\ dx^3 \end{pmatrix}$$

that can be written in Cartesian coordinates (as an example) as

$$dx^\mu = \begin{pmatrix} dt \\ dx \\ dy \\ dz \end{pmatrix}.$$

Partial derivatives are often represented using commas:

$$g_{\mu\nu,\alpha\beta} = \frac{\partial}{\partial x^\beta}\frac{\partial}{\partial x^\alpha} g_{\mu\nu}.$$

Note how the order of the indices changes and how a superscripted index in the denominator corresponds to a subscripted index in the numerator. Partial derivatives are only one kind of derivatives that we will need; in a general spacetime, a "tensor" formed using a partial derivative operator may not be invariant under coordinate transformations (and thus, not actually a tensor). On the other hand, the covariant derivative is invariant by construction and is represented by a nabla or a semicolon:

$$\nabla_\mu A_\nu = A_{\nu;\mu}.$$

We will elaborate on the covariant derivative further in the next subsection.

Additionally, the metric tensor can be used to lower an index (e.g., $g_{\mu\nu}E^\mu = E_\nu$), and the inverse metric tensor $g^{\mu\nu}$ can be used to raise an index (e.g., $g^{\mu\nu}E_\mu = E^\nu$).

1.2.2 The Einstein Equations

As we can see from the indices on the Einstein tensor and stress-energy tensor, $G_{\mu\nu} = 8\pi T_{\mu\nu}$ represents 16 equations. However, due to the symmetry of both tensors, only 10 of these

equations are unique. This provides us one glimpse into how compact Einstein notation is; the picture only becomes clearer as we dig into the definition of the Einstein tensor. We will start from the derivatives of the metric tensor and build up definitions from there until we have arrived back at the Einstein tensor.

Thus, we will first introduce the Christoffel symbol $\Gamma^{\sigma}_{\mu\nu}$, which depends on the first derivative $g_{\mu\nu,\delta}$ of the metric. Note in particular that we are calling it a symbol and not a tensor. By definition, tensors must follow a specific coordinate transformation law:

$$T^{\mu'_1\cdots\mu'_m}_{\nu'_1\cdots\nu'_n} = \frac{\partial x^{\mu'_1}}{\partial x^{\mu_1}}\cdots\frac{\partial x^{\mu'_m}}{\partial x^{\mu_m}}\frac{\partial x^{\nu_1}}{\partial x^{\nu'_1}}\cdots\frac{\partial x^{\nu_n}}{\partial x^{\nu'_n}}T^{\mu_1\cdots\mu_m}_{\nu_1\cdots\nu_n}.$$

While the Christoffel symbol does not obey this law, the symbol is critical to the definition of the covariant derivative. The covariant derivative is a type of derivative that is invariant under coordinate transformations; it reduces to the partial derivative in flat spacetimes.

We will next consider the second derivative of the metric, written as $g_{\mu\nu,\alpha\beta}$; with it, we will construct the Riemann curvature tensor. This tensor is defined as

$$R^{\alpha}_{\ \beta\gamma\delta} = \Gamma^{\alpha}_{\ \beta\delta,\gamma} - \Gamma^{\alpha}_{\ \beta\gamma,\delta} + \Gamma^{\alpha}_{\ \mu\gamma}\Gamma^{\mu}_{\ \beta\delta} - \Gamma^{\alpha}_{\ \mu\delta}\Gamma^{\mu}_{\ \beta\gamma}$$

and describes the deformation of spacetime (in this definition, the $g_{\mu\nu,\alpha\beta}$ terms are encapsulated within the derivatives of the Christoffel symbol). The Ricci tensor is then defined as the trace of the Riemann tensor $R_{\alpha\beta} = R^{\mu}_{\ \alpha\mu\beta} = g^{\mu\nu}R_{\mu\alpha\nu\beta}$ and encodes information about how the volume (but not shape) of a body changes throughout the spacetime. The Ricci scalar is the trace of the Ricci tensor, $R = R^{\mu}_{\ \mu}$.

The Einstein tensor is a function of the metric tensor, its inverse, and its first and second spatial derivatives and can be written as

$$G^{\mu}_{\ \nu} = R^{\mu}_{\nu} - \frac{1}{2}\delta^{\mu}_{\ \nu}R,$$

connecting geometry of a particular spacetime to the arrangement of mass and energy it contains. Thus, the Einstein equations are a set of coupled, second-order partial differential equations [4].

1.2.3 Relevance to Astrophysical Systems

Stars form from the gravitational collapse of massive clouds of hydrogen gas. Gravity crushes this gas until the necessary pressures are reached for nuclear fusion to begin. The energy released in the form of photons by the fusion of hydrogen into helium leads to an outward pressure, opposing the inward pull of gravity. The star exists in this state as helium accumulates in the star's core. Eventually, stars with enough mass will begin to fuse this helium into carbon; even more massive stars will eventually fuse even heavier elements as well. Regardless of the star's mass, however, all stars will eventually run out of fuel. The outward pressure from fusion will cease, and gravity will crush inwards. This implosion results in the star shedding its outer layers and leaving behind a compact object. In heavier stars, this implosion results in a supernova; in lower mass stars, the result is a planetary nebula.

The star's mass determines the mass of the stellar remnant left over after the star's death throes; in turn, the stellar remnant's mass determines its type. The least massive stars will become white dwarfs. For these stellar remnants, the collapse caused by the inward gravitational pull is eventually stopped by electron degeneracy pressure. More massive stars will become neutron stars, which can support incredibly strong magnetic fields. They are supported by neutron degeneracy pressure. For the most massive stars, nothing is able to resist the inward crush of gravity, resulting in a black hole. The mass is concentrated at a point known as the singularity. The gravitational pull from such a dense object is so extreme that there exists a radius (called the event horizon) at which the escape velocity exceeds the speed of light. Because these stellar remnants are incredibly dense, their gravitational fields are so strong that general relativity is necessary to accurately model them.

The matter surrounding such extreme objects is another source of interest. Binary systems of compact objects are particularly interesting for gravitational wave and multimessenger astrophysics. The gravity exerted by each object influences the other, transferring energy and momentum and emitting gravitational waves that carry information about the system. The extreme magnetic fields and rotation speeds of some neutron stars (known as pulsars) create intense beams of radiation that can serve as incredibly precise clocks [5]. They can also be used to detect gravitational waves [6]. The magnetic fields also influence the matter

surrounding the neutron star; for diffuse plasmas (like the medium that permeates most space in the universe), the magnetic field and strong gravity dominate the dynamics in the region appropriately known as the magnetosphere [5]. One particular phenomenon that occurs in such an environment is known as a current sheet. This is a flat disc of current that results in a sharp discontinuity in the magnetic field [7].

While nothing can escape from within the event horizon of a black hole, matter can escape from near it. As matter accretes onto a compact object, the dynamics of the resulting accretion disk result in the emission of photons and high-energy particles. Such emissions are another important source of information about stellar remnantse [5].

1.3 Numerical Relativity

If we want to perform simulations of black holes (BHs) on a computer, we must first find a way to represent these equations on the computer and represent the system that we wish to study as an initial value problem. In doing so, we will build the spacetime slice-by-slice and construct a stack of three-dimensional hypersurfaces. We thus need to turn our representation of the four-dimensional spacetime into representations of the hypersurfaces and new functions to describe the relationship between the hypersurfaces.

We do this by decomposing the four dimensional equations into a 3+1 decomposition. In this representation, the four-dimensional spacetime metric $g_{\mu\nu}$ is broken down into a three-dimensional spacetime metric γ_{ij}, the shift β^i, and the lapse α. For instance, in the Arnowitt-Deser-Misner (ADM) formalism [8], the four-metric is represented as

$$g_{\mu\nu} = \begin{pmatrix} -\alpha^2 + \beta^k\beta_k & \beta_j \\ \beta_i & \gamma_{ij} \end{pmatrix}.$$

Essentially, we slice the four-dimensional spacetime into a series of three-dimensional spaces that evolve through time. Then, the three-metric γ_{ij} represents the curvature of an individual slice of the spacetime, the shift β^i represents the change in the position of a given point during a timestep, and the lapse α represents how much time elapses at a point during the timestep. We can then decompose Einstein's equations into a set of constraint equations (governing the behavior of the spacetime fields within a slice) and a set of evolution equations (governing

how those fields change between timesteps) [9]. This allows us to use the method of lines, wherein we represent the spatial derivatives using standard finite difference methods and then advance the simulation forward in time using explicit Runge-Kutta methods [10].

When conducting any simulation, the choice of resolution is vitally important. Although a higher resolution allows the simulation to more accurately model the physical features within the system, it can also be very computationally intensive. We use the adaptive mesh refinement (AMR) grids infrastructure provided by the Einstein Toolkit (ETK) to mitigate the downsides of high- and low-resolution grids. This technique uses multiple nested boxes, with the higher-resolution grids that are necessary to accurately model the highly curved spacetime in the strong-field region near the black holes and with the lower-resolution grids farther out where the spacetime fields are smoother. The grids are overlapped, allowing data to be interpolated between them as necessary. By doing this, we get the benefits of high-resolution grids while mitigating their downsides. In the ETK, this functionality is provided by a thorn (that is, a module) called `Carpet` [11].

1.3.1 Automatic Code Generation for Einstein Toolkit Thorns

In numerical relativity, we simulate incredibly complicated equations on the computer, as can be seen in the definitions in Appendix B and Appendix C. Coding up these equations can represent a significant challenge, because this complexity introduces ample opportunity for human error. This complexity creep is especially apparent as we translate the equations from the elegant and compact Einstein notation to a much more explicit representation on the computer.

However, it is possible to simplify this using automatic code generation and computer algebra systems. By using a computer algebra system, we can input the expressions we need to code up in a format that is similar to how we write them in papers. By using arrays, we can even store entire tensors and access them using code that is similar to the elegant and compact Einstein notation. Automatic code generation can then be used to export these symbolic expressions as efficient code in a faster language, such as C. Code bases that do this, such as Wolfram Mathematica [12] and SymPy [13], can also use other techniques such as common subexpression elimination [13] to make the resulting code even more efficient.

For instance, if we wanted to code the expression $x = \sin(ab) + \cos(ab) + \log(ab)$, a naïve way to code this might be (in pseudocode)

```
x = sin(a*b) + cos(a*b) + log(a*b)
```

which requires six floating point operations (FLOPs) (here, the operations are assignment `=`, addition `+`, and multiplication `*`), neglecting the additional computational cost of the transcendental functions. However, note that we computed the product ab three times. If we instead coded the expression as

```
tmp = a*b
x = sin(tmp) + cos(tmp) + log(tmp)
```

we use only five FLOPs, again neglecting the transcendental functions, which are unchanged. If we allow the computer to hunt for such optimizations, it can be a big improvement to the efficiency of our code.

One such code base that does this is Kranc [14]; however, it uses Wolfram Mathematica, which requires an expensive license to use. On the other hand, NRPy+ is written in Python using SymPy, which is open source and completely free [15]. It is with this codebase that we created the code used in Chapters 3 and 4.

1.3.2 Convergence Testing

It is of critical importance in numerical relativity to validate our codes so that we can be certain that they accurately model physics. The simplest way to do this is by comparing the numerical solution to the exact solution, but we still must define some standard to tell us just how close we are. We do so by running a test several times at different resolutions and computing the convergence order.

We calculate this by first assuming that each approximate numerical solution is related to the exact solution F by $F_i = F + (\Delta x_i)^k$, where k is the convergence order and Δx_i is the resolution of the i^{th} test, as F_i is the approximate solution at that resolution. For simplicity, we let

$$\Delta x_2 = \frac{\Delta x_1}{2}.$$

Then, solving for the convergence order k, we find that

$$k = \log_2 \left(\frac{F - F_1}{F - F_2} \right),$$

where k is the convergence order, F is the exact solution, F_1 is the approximate solution on the coarser grid with resolution Δx, and F_2 is the approximate solution on the finer grid with resolution $\Delta x/2$.

We can use a similar technique when we do not know the exact solution by running a simulation three times at resolutions $\Delta x_i = 2^{1-i} \Delta x_1$. In this case, we find that the convergence order is

$$k = \log_2 \left(\frac{F_1 - F_2}{F_2 - F_3} \right).$$

Regardless of which method we use, we must compare our actual convergence order to the algorithm's theoretical convergence order. This must be calculated for any given algorithm [10].

1.4 General Relativistic Force-Free Electrodynamics

General relativistic force-free electrodynamics (GRFFE) is a set of equations that govern the behavior of diffuse plasmas in strongly curved spacetimes. It is a special case of ideal general relativistic magnetohydrodynamics (GRMHD), in which we consider three sets of equations: Einstein's equations of general relativity, Maxwell's equations of electromagnetism, and the hydrodynamics equations. In the force-free limit, the pressure and density of the plasma approach zero, so the magnetic pressure dominates the gas pressure.

With such a complex set of equations to solve, there are many different variables of which we must keep track. We divide these into two groups: the primitive variables and the conservative variables. Generally speaking, the conservative variables will be what we evolve directly using conservation equations, and the primitive variables correspond more closely to what we might measure in a physical experiment. So, during a timestep, we first compute the new value of the conservative variables using their evolution equations. We will then update the primitive variables from the new values of the conservative variables and apply boundary conditions. Because GRFFE is a special case of GRMHD, we are able

to make several simplifications. In particular, the conservative-to-primitive solver is greatly simplified, and the necessary equations can easily be analytically inverted.

Some of the first work done with these equations was done in the 1970s by Wald, who found several analytic solutions for black hole magnetospheres [16]. One particularly interesting solution is known as "Exact Wald," which consists of a spinning black hole in a uniform magnetic field parallel to the black hole's spin axis. This creates a system that is static in time. Another solution is the "Magnetospheric Wald," which is similar to the first, but represents a plasma-filled system that results in a current sheet forming around the black hole, in which there is a sharp change in the magnetic field. It has been called the "ultimate Rosetta stone" by Komissarov for black hole electrodynamics [7]. These systems have become important tests for GRFFE codes. The Exact Wald system is static, which allows us to use the initial data as an exact solution. Although there is no static solution for the Magnetospheric Wald system, its current sheet is an important physical phenomenon that we want to study, so we need to make sure our code does not dissipate it too rapidly.

The Blandford and Znajek mechanism is an early analytic solution to the GRFFE equations [17]. It was motivated by the study of active galactic nuclei when it became apparent that these objects could not be powered by a large pulsar or group of pulsars. This model consists of a Kerr black hole surrounded by an accretion disc through which an external current is flowing. This current creates a magnetic field which threads the black hole; the plasma is perfectly conducting, allowing $\vec{E} \times \vec{B} = 0$ to hold. This causes some of the energy from the black hole to be emitted as light. This has been confirmed numerically by Spitkovsky for both the aligned and oblique cases, referring to the alignment of the black hole spin and magnetic field through the black hole [18].

Another important system governed by GRFFE is the magnetosphere of a neutron star. There are toy models that represent such objects, such as the aligned rotator, which are used for code tests; recently, more detailed and realistic analysis has shown that after thousands of years, these systems can become unstable as the FFE conditions break down, releasing large bursts of energy that might be seen as transients [19].

As previously mentioned, Blandford and Znajek provided a simple tool for understanding black hole accretion disks by modeling a steady-state, axisymmetric magnetosphere and a

method for extracting energy from the black hole spin [17]. It is also the proposed mechanism for many high-energy transients over a wide range (i.e., a factor of 20) of time scales [20]. It is also useful for explaining relativistic jets that are observed coming from many galaxies, such as M87 [21]. This galaxy has been of particular interest lately, because the supermassive black hole at the galaxy's center was directly imaged in 2017 [22].

Many of the GRFFE systems have some symmetry; thus, it would be more efficient to simulate them using spherical coordinates. Because NRPy+ allows one to easily use many different coordinate systems, we will port `GiRaFFE` to NRPy+ in this dissertation. Other groups have also worked on writing GRFFE simulations in other coordinate systems. In July 2020, another group published a code that solves the GRFFE equations in spherical coordinates using the ETK [23]. Although there are many similarities between their project and ours, there are also several differences. Perhaps the biggest is the formalism used. We mentioned earlier that the equations we use are a special case of ideal GRMHD; on the other hand, their formalism starts from Einstein field equations and adds Maxwell's equations. This leads to the use of different conserved variables. While we use the vector potential and Poynting flux, they use the electric and magnetic fields. This formalism was developed by Komisarrov in 2004 [7] and was partially motivated by the Blandford-Znajek mechanism.

Chapter 2

Induced Spins from Scattering Experiments of Initially Nonspinning Black Holes[1]

2.1 Abstract

When two relativistically boosted, nonspinning black holes pass by one another on a scattering trajectory, we might expect the tidal interaction to spin up each black hole. We present the first exploration of this effect, appearing at fourth post-Newtonian order, with full numerical relativity calculations. The basic setup for the calculations involves two free parameters: the initial boost of each black hole and the initial angle between the velocity vectors and a line connecting the centers of the black holes, with zero angle corresponding to a head-on trajectory. To minimize gauge effects, we measure final spins only if the black holes reach a final separation of at least $20M$. Fixing the initial boost, we find that as the initial angle decreases toward the scattering/nonscattering limit, the spin-up grows nonlin-

[1]As first author, Patrick E. Nelson drafted this paper and led the project, analyzing the scattering experiments with the assistance of coauthors Zachariah B. Etienne, Sean T. McWilliams, and Viviana Nguyen. Note that this version of the paper has been lightly edited and reformatted. Citation: P. E. Nelson, Z. B. Etienne, S. T. McWilliams, and V. Nguyen. Induced Spins from Scattering Experiments of Initially Nonspinning Black Holes. *Physical Review D*, 100, 124045, September 2019 [24].

early. In addition, as initial boosts are increased from $0.42c$ to $0.78c$, the largest observed final dimensionless spin on each black hole increases nonlinearly from 0.02 to 0.20. Based on these results, we conclude that much higher spin-ups may be possible with larger boosts, although achieving this will require improved numerical techniques.

2.2 Introduction

Inelastic scattering experiments in high-energy physics have deepened our understanding of nonlinear interactions between quarks within hadrons and mesons [25, 26]. Analogously, we would expect black hole scattering experiments performed in full numerical relativity to provide insights into strong-field gravity at its strongest and most dynamical. This paper presents such experiments set up in a way that effectively eliminates gauge ambiguities. Specifically, we will address to what degree two relativistically boosted, equal-mass, non-spinning black holes on a scattering trajectory induce spins on each other.

The notion that a black hole's spin may be influenced by a distant object is not a new one; for example in 1974, Hartle [27] demonstrated with analytical arguments that a stationary, coplanar moon far from a spinning black hole will act to spin down the hole. More relevant to our work, a similar effect was studied by Campanelli, Lousto, and Zlochower for black hole binaries in quasicircular orbits [28]. They also derived a formula for the radiated angular momentum based on the Weyl scalar ψ_4 [29]. However, our work focuses on scattering systems.

Our experimental technique is as follows. We first uniquely specify initial black hole trajectories with Brandt-Brügmann initial data parameters, then perform scattering experiments in full numerical relativity, and finally measure the final black hole spins at large separations. Final spins are measured with both the isolated horizon formalism [30] and the Christodoulou formula (involving the ratio of proper equatorial to polar circumferences of the apparent horizons) [31].

The final spins encode important information about the strong-field interaction, which first appears at fourth order in a post-Newtonian expansion in terms that account for the flux of angular momentum at the event horizon [32]. This work therefore describes a potential

way to validate post-Newtonian calculations at very high order with numerical relativity.

Although this work appears to be the first to focus on the final spin of scattering black hole encounters, it is not the first to explore black holes on highly eccentric or hyperbolic trajectories. In one class of such interactions, orbits undergo "zooms" and "whirls"—that is, they whirl close to each other before zooming out to a larger separation, in an extreme example of the precession of the peribothron that occurs near the separatrix of bound and unbound orbits. This class of orbits has been studied in stationary black hole spacetimes [33, 34, 35], in the post-Newtonian limit [36, 37], and in extreme-mass-ratio systems [38, 39, 40, 41], and is related to the homoclinic orbits observed in stationary black hole spacetimes [42] and in the post-Newtonian limit [43]. Khurana and Pretorius were the first to show evidence of zoom-whirl orbits in full numerical relativity, for equal-mass, nonspinning binaries. They simulated up to five zoom-whirls, but showed that many more may be possible, since the number of orbits exhibits critical phenomenon behavior, with exponential scaling in the number of orbits n with the critical impact parameter b^*: $e^n \propto |b - b^*|^\gamma$ [44]. Although Khurana and Pretorius fine-tuned their initial data in an attempt to maximize the number of zoom-whirls, Healy, Levin, and Shoemaker found up to three orbits without fine-tuning [45].

Other studies of zoom-whirl behavior in equal-mass, nonspinning binaries have been performed (see, e.g., Refs. [46, 47]. These trajectories have also been studied for comparable-mass Kerr black holes [36] and large-mass-ratio Kerr black holes [48], as well as for binary systems that contain one spinning black hole and one nonspinning black hole [37]. Zooms and whirls have even been observed for hyperbolic orbits. Gold and Brügmann showed orbits that exhibited a single whirl followed by a zoom to infinity [49]. Our focus will be on highly eccentric encounters similar to these, with the black holes reaching large separations quickly after a close encounter. Our spin measures assume isolated black holes, so to minimize potential gauge or strong-field effects on these measurements, we only present final spin measures if, after a strong-field encounter, the final separation of the black holes remains at least $20M$ through the end of the calculation; we refer to encounters that satisfy this requirement as "scattering" events, whereas all other encounters are referred to as "nonscattering" or "merger" events. Throughout this work, we will work in geometrized units where $G = c = 1$.

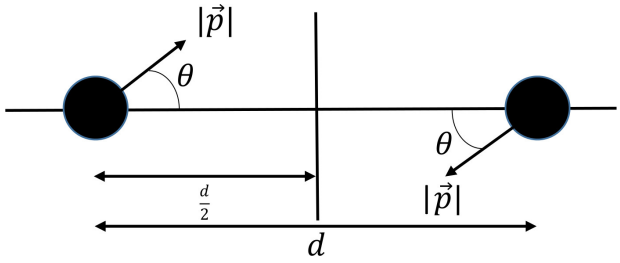


Figure 2.1: Schematic of the chosen initial black hole configuration. $|\vec{p}|$ is the magnitude of the Brandt-Brügmann momentum, θ is the shooting angle formed with the x axis, and d is the initial orbital separation. For calculations presented here, we choose $d = 100$ in code units, corresponding to initial separations between $52.4M$ in the case of an initial $v = 0.77$ boost and $96.6M$ in the case of an initial $v = 0.42$ boost ($c = 1$ units; M is the initial ADM mass), as summarized in Table 2.2.

As illustrated in Fig. 2.1, our experimental setup is quite similar to those adopted in the zoom-whirl literature for equal-mass black holes [44, 45, 46, 47], with the key exception that we start with a much larger separation of 100 in code units. This translates to between $50M$ separation with the highest boosts chosen up to $96M$ separation with the smallest initial boost, as the initial Arnowitt-Deser-Misner (ADM) mass increases monotonically with the initial boost. The actual values are summarized in Table 2.2. We find the choice of large initial separations to be especially important, as it gives the gauge fields ample time to settle and enables junk radiation to propagate away so that the gravitational-wave signal due to the dynamical interaction can be analyzed.

At fixed Brandt-Brügmann momentum magnitude $|\vec{p}|$, we find the induced spin-up increases monotonically as θ decreases toward the scattering/nonscattering separatrix. In addition, the maximum measured spin-up is found to be 0.02, 0.06, 0.11, and 0.20 with initial boosts of $v = 0.42$, 0.56, 0.66, and 0.78, respectively. This represents a significantly nonlinear trend as the initial boost parameter is increased. While the induced spins may not seem particularly large for astrophysically motivated boosts such as those potentially seen in hierarchical triple systems, we note that for bound systems, even small induced spins in each encounter may accumulate and significantly impact the dynamics over many orbital time scales.

The rest of the paper is organized as follows: Sec. 2.3 discusses the methods used for both

the simulations and data analysis; Sec. 2.4 describes the initial conditions of each simulation; Sec. 2.5 presents our results; and Sec. 2.6 contains our conclusions and plans for future work.

2.3 Numerical Approach and Diagnostics

We adopt standard moving puncture techniques, combining the 3+1 Baumgarte-Shapiro-Shibata-Nakamura (BSSN) formalism [50, 51] with moving puncture gauge conditions to construct the spacetime. We choose $W = e^{-2\phi}$ as our evolved conformal variable, and standard `1+log` lapse and Gamma-driver shift conditions.

Initial data are set up and evolutions performed using open-source tools within the Einstein Toolkit (ETK) [52, 53] infrastructure. In particular, constraint-satisfying Brandt-Brügmann two-black-hole initial data are generated using the `TwoPunctures` module [54] with optimized spectral interpolation [55]. The evolution of the BSSN equations was performed with the `McLachlan` [56, 14, 57] module. `QuasiLocalMeasures` was applied to calculate black hole spins [58], `AHFinderDirect` to measure horizon centroids and circumferences [59, 60], `WeylScal4` to compute Ψ_4 [61], as well as the Cactus Computational Toolkit [62, 63, 64] and Carpet [65, 11] for the adaptive mesh refinement grid infrastructure.

2.3.1 AMR Grid Structure

The adaptive-mesh-refinement (AMR) grids generated by Carpet use half-side lengths of 0.75×2^n, for $n = \{0, \ldots, 6\}, \{8, \ldots, 10\}$ in code units. The skip between $n = 6$ and $n = 8$ ensures a large region within which gravitational waves can be extracted at uniform resolution. All simulations enforce reflection symmetry across the orbital plane to minimize computational expense. The outermost boundary has a half-side length of 768, which ensures that approximate outer boundary conditions can have no causal impact on the spin measures. The three resolutions used for the most refined grid are $\Delta x = \{1/56, 1/66.\overline{6}, 1/85.\overline{3}\}$; henceforth, these shall be referred to as low, medium, and high resolution, respectively. Note that these resolutions are also given in code units, so that higher-boost cases will have higher resolutions (the initial ADM mass M for each boost is given in code units in Table 2.1). For boosts other than 0.66, only the "low" resolution was used. Finally, the Carpet parame-

ter `time_refinement_factors`, which controls how often a time step is performed on each refinement level, is set to `[1,1,1,1,2,4,8,16,32,64]`, so that the coarsest four grids are updated at the same rate, and each refinement level finer than that is updated twice as often as the next-coarser grid. This minimizes interpolation errors on the coarsest levels by disabling time prolongation between them.

2.3.2 Dimensionless Spin

The dimensionless spin of each black hole, a/M, is calculated using two approaches. First is the Christodolou spin, which is given by solving Eq. 5.2 of Ref. [31],

$$C_r = \frac{1+\sqrt{1-(a/M)^2}}{\pi} E\left(-\frac{(a/M)^2}{\left(1+\sqrt{1-(a/M)^2}\right)^2}\right), \tag{2.1}$$

where $C_r = C_p/C_e$ is the ratio of the polar and equatorial horizon circumferences, and $E(x)$ is the complete elliptic integral of the second kind,

$$E(k) = \int_0^{\pi/2} \sqrt{1-k\sin^2\theta}\, d\theta. \tag{2.2}$$

Second, the isolated horizon formulation is adopted as an alternative spin measure, which is provided by the ETK `QuasiLocalMeasures` module.

We also employ standard numerical surface integrals for evaluating the ADM mass $M_{\rm ADM}$ and angular momentum $J_{\rm ADM}$ in the Cartesian basis to measure mass and angular momentum lost to gravitational waves and spin-ups. Additionally, we define the "final angular momentum" and "final spin" to be a time average over the values of angular momentum and spin *after* the black holes have reached the cutoff separation $r_{\rm cutoff} = 20M$ as they move away from each other after the encounter.

Our measures of the final spin parameter assume that each black hole is far from another strong-field source and generally on an unbound trajectory. To confirm that our black hole spin measures are reliable, we only include cases in which the holes reach a final separation of at least $r_{\rm cutoff}$ through the end of the calculation. In Table 2.1, we compare the agreement of Christodolou spins calculated with different cutoff radii with the spin calculated at our chosen $r_{\rm cutoff} = 20M$; we see good agreement between cutoffs of $15M$ and $20M$, showing

$\lvert\vec{p}\rvert$	$M_{\rm ADM}$	SDA @ $\tilde{r}_{\rm cutoff} = 5.0$	SDA @ $\tilde{r}_{\rm cutoff} = 10.0$	SDA @ $\tilde{r}_{\rm cutoff} = 15.0$	SDA @ $\tilde{r}_{\rm cutoff} = 20.0$	Maximum $\tilde{r}_{\rm merge}$
0.490	1.13560	3.6 (3.4)	4.4 (3.7)	4.6 (3.7)	— (3.8)	3.41
0.735	1.28385	3.8 (3.8)	4.5 (4.0)	5.0 (4.1)	— (4.2)	3.95
0.980	1.46473	3.7 (3.8)	4.4 (4.1)	4.9 (4.5)	— (4.5)	4.36
1.500	1.90902	3.8 (4.0)	4.8 (4.4)	5.6 (4.7)	— (4.7)	3.75

Table 2.1: Validation of $r_{\rm cutoff}$ choice. Larger choices of $r_{\rm cutoff}$ ensure that the black holes are sufficiently far apart for an accurate final spin measurement and that they are unlikely to merge. The columns are described as follows. $|\vec{p}|$ is the Brandt-Brügmann initial momentum's magnitude, and $M_{\rm ADM}$ is the initial ADM mass in code units. The next four columns give the mean number of significant digits of agreement (SDA) between the Christodolou spin calculated with that cutoff radius and the Christodolou spin calculated with $r_{\rm cutoff} = 20.0 M_{\rm ADM}$; the em dashes in the $\tilde{r}_{\rm cutoff} = 20.0$ column indicate that the SDA between a measurement and itself is not meaningful. In parentheses, we give the mean SDA between the Christodolou and isolated horizon formalism dimensionless spin measures at points closest to the listed cutoff separation. For brevity we define tilded r quantities in this table to be normalized by $M_{\rm ADM}$; e.g., $\tilde{r}_{\rm cutoff} = r_{\rm cutoff}/M_{\rm ADM}$. Finally, "Maximum $\tilde{r}_{\rm merge}$" is the maximum separation observed between the two holes for cases in which, after a strong-field encounter, they ultimately merged at the given boost.

that we have a stable spin measurement. We also compare the Christodolou spin and the isolated horizon formalism measures of spin with varying choices of $r_{\rm cutoff}$, recording at each $r_{\rm cutoff}$ and for each initial boost the significant digits of agreement between the two spin measures (these are the values in parentheses in Table 2.1). The table confirms that at our fiducial $r_{\rm cutoff} = 20M$, the independent black hole spin measures agree to better than four significant digits across most cases. As a point of comparison, we find that over all chosen boost magnitudes and angles, no black holes return to merge after reaching about $r = 5M$ separation.

2.3.3 Parametrizing the Initial Boost

The initial Brandt-Brügmann boost parameter $\vec{p}_{\rm BB}/m_{\rm BB}$ provides an unambiguous measure of the initial boost of each black hole. However, we would expect that the junk radiation associated with the assumption of conformal flatness in Brandt-Brügmann initial data in-

creases as this momentum increases, acting to reduce the momentum of each black hole prior to the interaction more and more as we increase the initial boost [66].

To parametrize our runs in a way that is insensitive to the presence of junk radiation, we define the initial boost to be the coordinate speed of a *single* black hole imparted with the same initial Brandt-Brügmann momentum, in the limit $t \to \infty$. In particular, at each initial momentum chosen, we perform a dedicated numerical relativity calculation of a single black hole with this initial momentum, traveling along the x axis. We choose the AMR grids to be identical to the low-resolution experiments, except instead of two AMR grid hierarchies tracking one black hole each, we only need a single hierarchy to track the single black hole.

Plotting the position of the black hole in this calculation versus time, we find that it starts from near zero speed (due to the initial shift β^i being zero) and accelerates towards some constant speed. However, attempts at a linear fit to the late-time data revealed that the black hole coordinate acceleration on the numerical grids was still nonzero at the end of the calculation, when we were forced to terminate to ensure a valid AMR hierarchy. Therefore, instead of fitting the speed at the end of the calculation, we consider model functions that have asymptotes so the entire time series can be used to estimate the speed that our initial conditions represent.

A hyperbola in the x-t plane is a convenient choice here, so we fit hyperbolae to the data by minimizing the residual norm cost function

$$f(a,b,h,k) = \sqrt{\sum_{i=1}^{N} \left(x_i - g(a,b,h,k,t_i)\right)^2},$$

where x_i and t_i are the position and time of the black hole at each of N measurements of position, respectively. $g(a,b,h,k,t)$ is the hyperbola

$$g(a,b,h,k,t) = k - \sqrt{a^2\left(1 + \frac{(t-h)^2}{b^2}\right)},$$

where a is the semimajor axis, b is the semiminor axis, and (h,k) are the coordinates of the center. We then take the slope of the asymptote a/b as the actual preinteraction boost that corresponds to the input momentum. As illustrated in Fig. 2.2, the hyperbolic fit to data plotted at these late times is quite good.

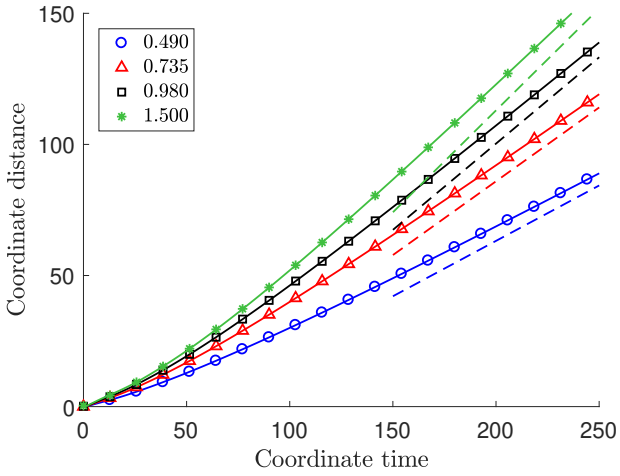


Figure 2.2: Parametrizing the initial boost: Late-time data of dedicated single-black-hole runs to find the initial boost. Here, $t = 0$ corresponds to an arbitrarily chosen later time, and all of these boost calibration calculations were performed to a final time of $\approx 250M$. The dashed lines show the asymptotic slope, solid lines show the actual data from the numerical relativity calculation, and the points show the hyperbolic fit.

This hyperbolic fitting method produces values of $v = 0.42$, 0.56, 0.66, and 0.78, corresponding to magnitudes of the initial Brandt-Brügmann momentum of $|\vec{p}| = 0.49$, 0.735, 0.98, and 1.50, respectively. Interestingly, these estimates for the initial boost are quite close to the respective values of $|\vec{p}|/M_{\rm ADM} = 0.43$, 0.57, 0.67, and 0.79. Notice however that our measured asymptotic coordinate boost is always slightly lower than the Brandt-Brügmann momentum-based measure, presumably due to the emission of junk radiation.

2.3.4 Radiated Angular Momentum

As the black holes are initially nonspinning, the orbital angular momentum contributes the entirety of the system's initial angular momentum. As the numerical relativity calculation progresses, however, this orbital contribution decreases as the black holes are spun up and gravitational waves carry away angular momentum. The radiated angular momentum $J_{\rm GW}$ is calculated using

$$J_{\rm GW} = \frac{r_{\rm ext}^2}{16\pi} \sum_{l,m} \int_{-\infty}^{t} -m(\dot{h}_+ h_\times - \dot{h}_\times h_+) dt, \tag{2.3}$$

Boost (v)	$\lvert\vec{p}\rvert$	Resolution	$d/M_{\rm ADM}$	$\theta_{\rm N-S}$	$N_{\theta_{\rm N-S}}$	$\theta_{\rm S}$	$N_{\theta_{\rm S}}$
0.42	0.490	Low	96.6	$\{5.600\times10^{-2}, 6.040\times10^{-2}\}$	11	$\{6.050\times10^{-2}, 7.100\times10^{-2}\}$	23
0.56	0.735	Low	77.9	$\{5.400\times10^{-2}, 5.440\times10^{-2}\}$	3	$\{5.460\times10^{-2}, 6.500\times10^{-2}\}$	31
0.66	0.980	Low	68.2	$\{5.400\times10^{-2}, 5.420\times10^{-2}\}$	3	$\{5.430\times10^{-2}, 5.800\times10^{-2}\}$	28
0.66	0.980	Medium	68.2	$\{5.400\times10^{-2}, 5.420\times10^{-2}\}$	3	$\{5.430\times10^{-2}, 5.800\times10^{-2}\}$	28
0.66	0.980	High	68.2	$\{5.418\times10^{-2}, 5.424\times10^{-2}\}$	3	$\{5.430\times10^{-2}, 5.690\times10^{-2}\}$	17
0.77	1.500	Low	52.4	$\{3.000\times10^{-2}, 5.950\times10^{-2}\}$	9	$\{5.960\times10^{-2}, 7.000\times10^{-2}\}$	21

Table 2.2: Initial conditions for black hole scattering experiments. $|\vec{p}|$ is the magnitude of the initial Brandt-Brügmann momentum; $\theta_{\rm N-S}$ and $\theta_{\rm S}$ are the ranges of the initial shooting angles (in radians) of the Brandt-Brügmann momentum for cases exhibiting scattering and nonscattering ("merging") behavior, respectively; and $N_{\theta_{\rm N-S}}$ and $N_{\theta_{\rm S}}$ are the number of simulations in those ranges.

derived from Eq. 24 of Ref. [67]. In all cases we measure the strain from the outgoing Weyl scalar ψ_4 at $r_{\rm ext} = 67.88M$, which is sufficiently far from the binary but in a high-enough resolution region to yield a reliable result.

2.4 Initial Conditions

For all sets of simulations, the initial separation was set to $d = 100$. Such large initial separations allow more time for the junk-radiation-induced perturbation on each black hole to settle before the holes strongly interact, and ensure that a sufficient interval of time exists between the junk and the gravitational radiation from the dynamical interaction.

In a given simulation with $|\vec{p}|$ and θ (whether bound or unbound), we set $p_x = \pm|\vec{p}|\cos(\theta)$ and $p_y = \mp|\vec{p}|\sin(\theta)$, giving rise to the configuration shown in Fig. 2.1 (this figure is not to scale).

Table 2.2 presents a complete list of calculations performed in this work. Notice that runs for $|\vec{p}| = 0.980$ were carried out at three different resolutions, with grids as specified in Sec. 2.3.1. Upon finding that results (i.e., final measured spin parameters) were not significantly improved at higher resolutions, other runs were carried out at low resolution.

The range of angles (given in radians) and number of runs performed in each set of experiments are also given; these sets are divided into subsets corresponding to bound and unbound trajectories. For instance, when setting the momentum magnitude to 0.490, we

find that the initial boost is $v = 0.42$ and that the angle $\theta = 6.050 \times 10^{-2}$ marks the boundary between scattering and nonscattering (i.e. "merging") cases. We performed 23 simulations between that angle and $\theta = 7.100 \times 10^{-2}$ and an additional 11 simulations satisfying $5.600 \times 10^{-2} \leq \theta \leq 6.040 \times 10^{-2}$ as we searched for the transition between scattering and nonscattering trajectories. Note that the separatrix angle between scattering/nonscattering is not monotonic in boost; this happens because the fixed initial separation in code units d combined with the larger initial ADM mass M in the higher-boost cases decreases the initial dimensionless separation d/M, so the angles are not immediately comparable.

2.5 Results

2.5.1 Trajectory Morphologies

The chosen set of initial conditions results in a variety of scattering and nonscattering (i.e. "merging") trajectories, as shown in Fig. 2.3. It is clear from the plots that this strong-field scattering exhibits much richer morphology than its classical analogue; some trajectories are zoom-whirl-like, as can be seen in the left and middle plots in Fig. 2.3. Likewise, while the right panel does not show a true "zoom," it is far from Keplerian due to the emission of gravitational waves and black hole spin-ups.

The plots of Fig. 2.3 also relate to how initial conditions are chosen: we first selected four Brandt-Brügmann momenta magnitudes corresponding to initial boosts of $v \approx 0.42, 0.56, 0.66$, and 0.78 (as measured in Sec. 2.3.3). At each of these magnitudes, we reduced the shooting angle until immediate merger occurred. This resulted in three classes of trajectories: immediate mergers (not of interest to this study), marginal scattering (which may subsequently merge, but the holes remain isolated long enough for a reliable spin measurement), and scattering; these are illustrated in the left, middle, and right plots in Fig. 2.3, respectively.

2.5.2 $v_{\rm boost} = 0.66$ Spin-Up Study

The left panel of Fig. 2.4 presents final spin data from the $v_{\rm boost} = 0.66$ case carried out at three resolutions. As can be seen, spin-up occurs in *every* case, and measures at different

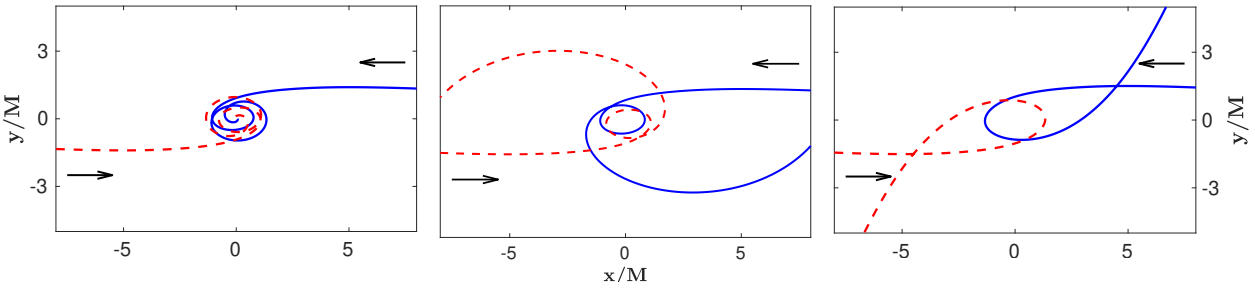


Figure 2.3: Nonscattering (left), marginally scattered (middle), and scattered (right) black hole trajectories. Arrows indicate the direction each puncture is initially traveling and M is the initial ADM mass of the system.

numerical resolutions agree well with each other: the relative error between spins at medium and low resolution never exceeds 1.4%, and the relative error between high- and medium-resolution spins never exceeds 0.6%. Comparing the Christodolou and the isolated horizon formalism [30] measures of dimensionless spin J/M^2 for each hole, we find agreement to within one percent in the worst case; typically the two measures agree to three to five significant digits.

The spin-up increases as the shooting angle decreases towards the separatrix between scattering and nonscattering trajectories. This increase is nonlinear and concave-up, reaching a maximum of 0.11 [2].

2.5.3 Maximum Spin-Up

Expanding our analysis to the other boost cases, we again find that in scattering cases, as the separatrix is approached, the spin-up increases to its maximum. This study provides the additional insight that the maximum spin-up depends strongly on the initial boost, as shown in the right panel of Fig. 2.4. As we increase the initial boost of the BHs, the maximum final BH spin obtainable at that boost increases; again, this increase is significant and nonlinear. The maximum spin for $v = 0.42$ was 0.02, and for $v = 0.56$, we found 0.06. The maximum

[2]Note that the shooting angle θ is related to the impact parameter via $b = d \sin\theta$, where d is the initial separation between the holes. Since the shooting angles we use are small, the small-angle approximation gives $b = d\theta$, so the left panel of Fig. 2.4 would remain almost entirely unchanged if the impact parameter were plotted instead of the shooting angle.

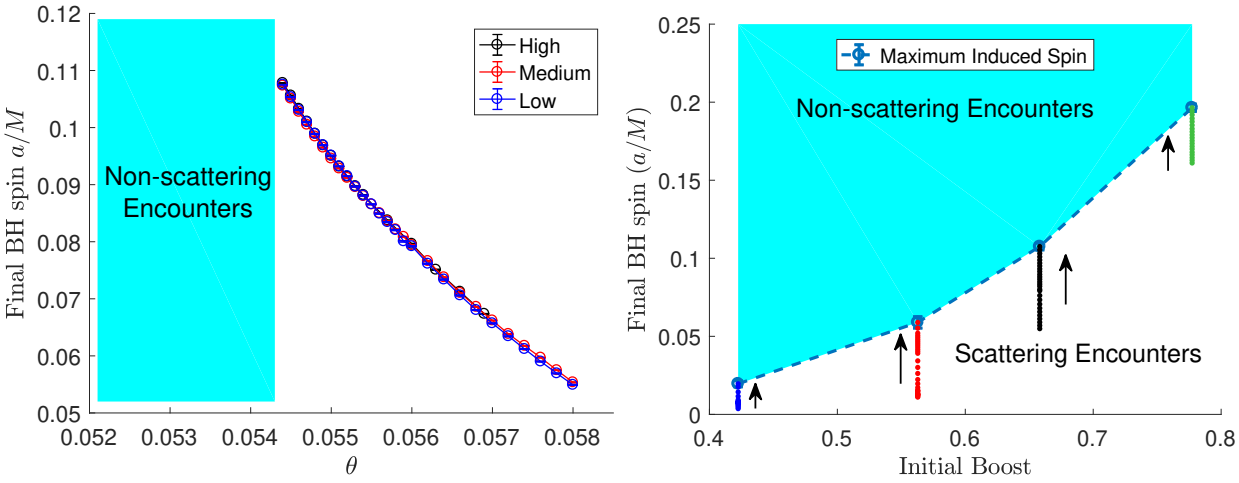


Figure 2.4: **Left panel:** The final spin a/M is computed by averaging the Christodoulou spin over late times (once the black holes are sufficiently far apart as to be weakly interacting, which we have defined as $20M$ separation), and is here plotted against the shooting angle θ, which increases with impact parameter.

Right panel: The maximum spin-up obtained at a given initial boost is plotted using open circles; the error bars on these points do not exceed 10% of the point's value. Other spin-ups are displayed as points; thus, the shooting angle increases downwards on this plot for a given set of points. The shaded region above the dashed line indicates the regime of immediate mergers. Black arrows on this plot indicate the direction of decreasing θ.

induced spin parameter we found overall was 0.20, for $v = 0.78$.

As we demonstrated previously, the spin measures are quite reliable, and in fact the largest uncertainty in maximum induced spin at a given initial boost comes from whether or not we truly found the shooting angle closest to an immediate merger. Therefore, in Fig. 2.4, we base the error bars on sampling resolution in shooting angle. (The error bars are similar in size to the open circles, and become difficult to distinguish for some of the points.) The error bars in the right panel of Fig. 2.4 are calculated as the difference between the highest and second-highest spin measured at a given boost. Again, the error bars are smaller than the data points themselves.

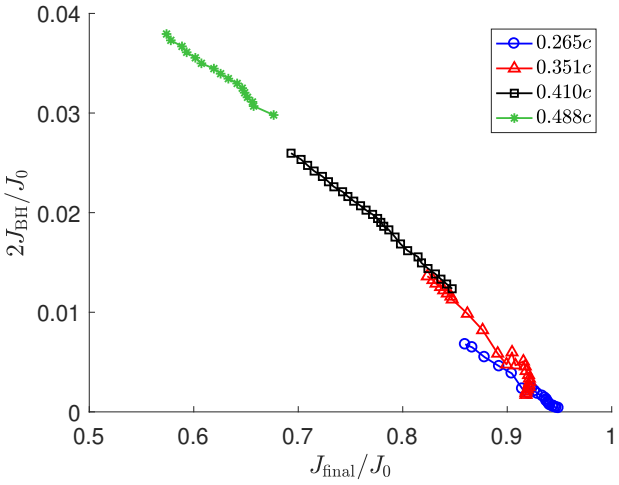


Figure 2.5: Spin-up efficiency, measured as the fraction of the initial angular momentum, J_0, that was transferred to the spins of the two black holes, $2J_{\rm BH}$.

2.5.4 Spin-Up Efficiency

The angular momentum spinning up the holes originates from the orbital angular momentum in the initial data. We would like to identify what fraction of this initial angular momentum is transferred into spin angular momentum. Figure 2.5 plots the efficiency of spin-up, $2J_{\rm BH}/J_0$, and the proportion of orbital angular momentum remaining in the system after the encounter, $J_{\rm final} = (J_0 - J_{\rm GW} - 2J_{\rm BH})/J_0$. Here, we define $J_{\rm BH}$ to be the dimensionful angular momentum as calculated using the isolated horizon formalism, and $J_{\rm GW}$ is defined in Eq. 2.3.

Fixing the initial boost, the smallest angle resulting in a scattering trajectory corresponds to the largest $J_{\rm BH}$ and, therefore, the largest spin-up efficiency. Further, we find that higher boosts increase the spin-up efficiency as well. We conclude that this spin-up effect has the potential to significantly impact the orbital evolution of a binary. This is not taken into account in current, commonly used post-Newtonian (PN) approximants, as the spin-up is not modeled until 4PN order [32]. Further, adding spin-spin and spin-orbit interaction terms to initially nonspinning configurations is not yet a standard approach in PN. Adding such terms would be necessary to use these results to validate PN theory.

2.6 Conclusions

We found that decreasing the shooting angle at a given initial boost increases the spin induced onto black holes that undergo a scattering trajectory. Further, as the initial boost is increased, the maximum spin-up and spin-up efficiency (i.e. the fraction of the initial angular momentum transferred to the black holes' spins) increases nonlinearly.

The maximum spin-up observed was $a/M = 0.20$ imparted on each hole, which occurs with an initial boost of $0.78c$ (the maximum boost chosen). This also corresponds to the maximum spin-up efficiency observed, of $2J_{BH}/J_0 = 3.9\%$. Once post-Newtonian theory has been completed at 4PN order for this type of interaction, this work will provide an exciting new avenue for validating PN theory directly with numerical relativity calculations.

These results indicate that the spin-ups and spin-up efficiencies may increase significantly with larger initial boosts, which will be a focus of future work. However, this work presents boosts near the upper limit allowed by conformally flat Brandt-Brügmann initial data, so larger initial boosts will require improved initial data *a la* Ruchlin, Healy, Lousto, and Zlochower [66], as well as higher numerical resolutions.

This is a very rich problem with a massive parameter space left to explore, and we hope to eventually study cases with varying initial spins and unequal mass ratios. With so many options available, we plan to add black hole scattering experiments to BlackHoles@Home, a distributed computing project enabling numerical relativity calculations of black hole interactions to be performed on consumer-grade desktop or laptop computers [68, 15, 69].

In future work we would also like to extract more gauge-invariant quantities from the gravitational-wave data, such as the peak frequency of the radiation, $f_{\rm peak}$. We attempted to do so with these data, but they were too noisy to obtain reliable measures. It would also be useful as an additional validation to measure the conservation of total J in the system by, e.g., directly computing the ADM $J_{\rm final}$ and comparing it to the proxy $J_0 - J_{\rm GW} - 2J_{\rm BH}$ that we used. Verifying conservation of total $M_{\rm ADM}$ will be useful as well.

More broadly speaking, we hope that the calculations performed here will allow us to characterize a subtle strong-field effect that could nonetheless bring about observable effects in future high-precision gravitational-wave observations.

Acknowledgments

We thank I. Ruchlin for helpful discussions. This work was supported by NSF awards OIA-1458952, PHY-1607405 and PHY-1912497, as well as NASA awards ISFM-80NSSC18K0538 and TCAN-80NSSC18K1488. Computational resources were provided by West Virginia University's Spruce Knob high-performance computing cluster, funded in part by NSF EPSCoR Research Infrastructure Improvement Cooperative Agreement No. 1003907, the state of West Virginia (WVEPSCoR via the Higher Education Policy Commission), and West Virginia University.

Chapter 3

Einstein Toolkit Thorns Using NRPy+

3.1 The Scalar Wave Equation

The first ETK thorns written in NRPy+ were `WaveToyNRPy` and `IDScalarWaveNRPy`, a NRPy+-based implementation of the ETK thorn `WaveToyC`, which solves the basic scalar wave equation

$$\partial_t^2 u = c^2 \nabla^2 u,$$

where the solution $u = u(t, x, y, z) = u(t, \vec{x})$ in Cartesian coordinates. This simple hyperbolic differential equation is a useful test of the underlying code infrastructure because many of its exact solutions are very simple. The particular exact solution we chose to solve was the monochromatic plane wave

$$u(t, \vec{x}) = \sin(\hat{k} \cdot \vec{x} - ct),$$

where $\hat{k}$ is a unit vector. With this analytic solution in hand to check our numerical work, we now turn to our numerical scheme.

To solve this equation numerically, we will first reduce this second-order partial differential equation (PDE) into a set of two coupled first-order PDEs by introducing the variable $v = \partial_t u$. It then follows that if we take the partial derivative with respect to t of both sides

we obtain

$$\partial_t v = \partial_t^2 u \tag{3.1}$$

$$= c^2 \nabla^2 u. \tag{3.2}$$

So, we are left with the equations

$$\partial_t u = v \tag{3.3}$$

$$\partial_t v = c^2 \nabla^2 u. \tag{3.4}$$

We then use the method of lines to evolve some initial data forward in time. Because we have an exact solution, the initial data are simply that exact solution at time $t = 0$; that is,

$$u(0, \vec{x}) = \sin(\hat{k} \cdot \vec{x}).$$

In the method of lines, the spatial derivatives are handled using standard finite-difference approaches; NRPy+ generates these for us to arbitrary order. The temporal derivatives are then handled using the usual techniques for ordinary differential equations; in this case, we use the ETK's built-in RK4 time-stepping thorn.

To solve these equations, we write two separate thorns that work in tandem. The thorn `IDScalarWaveNRPy` is run first to set the initial data at time $t = 0$. The second `WaveToyNRPy` is then run at each timestep to evolve those data forward in time. The process of writing each thorn proceeds similarly.

First, C-code kernels are generated using NRPy+. These kernels contain the code instructions to loop over the appropriate section of the grid (the whole grid for the initial data and only the grid interior for the time evolution) and set and perform the calculations to either set u and v at time $t = 0$ in `IDScalarWaveNRPy` or their time derivatives in `WaveToyNRPy`. These are then saved as header files. We then write a `.C` file which contains the functions called by the toolkit. In each thorn, one function simply includes the generated kernels. The others exist to perform other useful tasks within the simulation, such as ensuring that the parameters take on allowed values or telling other ETK thorns how we want them to work (e.g., setting symmetries, selecting boundary condition types, and making sure the MoL thorn knows which gridfunctions are which).

Then we write the `.ccl` files, which are responsible for defining how this module interacts and interfaces with the ETK infrastructure. We handwrote these following the example set by the `WaveToyC` thorn; however, since writing these thorns, the infrastructure has been developed to generate these automatically. The file `interface.ccl` defines the gridfunction groups needed, and provides keywords denoting what this thorn provides and what it should inherit from other thorns. This file governs the interaction between this thorn and others. The file `param.ccl` specifies free parameters within the thorn, enabling them to be set at runtime. It is required to provide allowed ranges and default values for each parameter. The file `schedule.ccl` allocates storage for gridfunctions, defines how the thorn's functions should be scheduled in a broader simulation, and specifies the regions of memory written to or read from gridfunctions. The final file that needs to be generated is `make.code.defn`, which simply lists all files that need to be compiled for a given thorn.

The notebook that generates `WaveToyNRPy` is included as Appendix A. Other notebooks that generate ETK thorns follow its example.

With these files all generated, the last thing we need to do is create a parameter file (where we can overwrite default parameter values from `param.ccl` if necessary). Then, we can run the simulation; we do so at multiple resolutions to check for convergence. This can be seen at the end of Appendix A.

3.2 Weyl Scalars and Invariants

A major goal of numerical relativity is the prediction of the gravitational wave signals that various astronomical systems would emit. A key component of this goal is the computation of the Weyl scalars.

The Weyl scalars are a set of five complex scalars, numbered 0 through 4, that are contractions of the Weyl tensor with some null tetrad. The Weyl Tensor itself is a rank-4 tensor that encodes information about the curvature of spacetime, specifically the change in a body's shape; we use contractions of this tensor to generate predictions about gravitational waves. It is a rank-4, 4-dimensional tensor with 20 independent components, making for some of the most complex expressions used in numerical relativity. In particular, we are interested

in ψ_4 because its real and imaginary components represent the second time derivatives of the strain measured by gravitational wave detectors.

To calculate the Weyl scalars, we begin by constructing a null tetrad $l^a, n^a, m^a, \overset{*}{m}{}^a$; we follow the choice of the original `WeylScal4` [70] and choose the quasi-Kinnersley tetrad. We use the overset $*$ to represent the complex conjugate to avoid possible confusion with conformal quantities used in the BSSN formalism. We also also contruct the Christoffel symbols

$$\Gamma^i_{kl} = \frac{1}{2}\gamma^{im}(\gamma_{mk,l} + \gamma_{ml,k} - \gamma_{kl,m}),$$

the Riemann curvature tensor

$$R_{abcd} = \frac{1}{2}(\gamma_{ad,cb} + \gamma_{bc,da} - \gamma_{ac,bd} - \gamma_{bd,ac}) + \gamma_{je}\Gamma^j_{bc}\Gamma^e_{ad} - \gamma_{je}\Gamma^j_{bd}\Gamma^e_{ac},$$

and the trace of the extrinsic curvature tensor $K = \gamma^{ij}K_{ij}$, because these are frequently used in the definitions of the Weyl scalars.

With these quantities in place, we are now free to construct the Weyl scalars. We precompute several frequently reused terms to help keep the code readable. We also break each definition into lines and then construct the scalars line by line, taking advantage of the Jupyter notebooks to allow us to keep the LaTeX-ed equations right next to the corresponding code, further increasing readability. We also calculate the Weyl invariants [71], which are specific combinations of the Weyl scalars, to fully reproduce the functionality of the original `WeylScal4` [70]. This process can be seen in full in Appendix B.

With these expressions coded, we now need to verify them. Because of the espressions' complexity, before diving into a numerical test, we decided to use Mathematica to compare each expression with the corresponding expression in the original `WeylScal4` analytically. In doing so, we found that our expressions were identical to those in the original Kranc-generated thorn.

For a numerical test, construction of the thorn proceeds in much the same way. We use SymPy and NRPy+ to generate a C-code kernel; we then write the function to use that kernel and the three `.ccl` files and `make.code.defn`. Then, a parameter file can be written to perform a basic simulation (we chose a binary black hole head-on collision) and output the Weyl scalars and invariants using both the original `WeylScal4` and our new `WeylScal4NRPy`.

We found that the relative error between the two versions for any quantity was at most 10^{-7}.

3.3 GRFFE in NRPy+ Using the Einstein Toolkit

The next thorns attempted to convert the ETK thorns `GiRaFFE` and `GiRaFFEfood` to NRPy+. The thorn `GiRaFFE` is built to evolve the GRFFE system of equations, and the thorn `GiRaFFEfood` provides initial data for it.

The original `GiRaFFE` code, as presented in [72], exists as a significant modification to `IllinoisGRMHD`. As such, it used a third-order reconstruction algorithm with a slope limiter and Riemann solver [73, 74, 75] to handle spatial derivatives in the general relativistic force-free electrodynamics (GRFFE) equations. However, because the GRFFE equations do not generally permit shocks, we thought a more optimal approach might involve finite-differencing *all* derivatives in the GRFFE equations. This approach, in addition to being a far simpler algorithm in general, would also allow us to take advantage of NRPy+'s ability to generate C code to generate finite-difference spatial derivatives at any user-specified order.

Thus, we sought to rewrite the equations of GRFFE as used in the original `GiRaFFE` code so that all derivatives that appear are represented numerically as finite-difference derivatives. As we will see, the largest complication stems from derivatives of magnetic fields—requiring judicious application of the chain and product rules.

The GRFFE evolution equations (from Eq. 13 of [72]) that we want to code in the NRPy+ version of `GiRaFFE` are as follows:

$$\partial_t \tilde{S}_i = -\partial_j \left(\alpha\sqrt{\gamma} T^j_{\mathrm{EM}i}\right) + \frac{1}{2}\alpha\sqrt{\gamma} T^{\mu\nu}_{\mathrm{EM}} \partial_i g_{\mu\nu} \tag{3.5}$$

$$\partial_t A_i = \epsilon_{ijk} v^j B^k - \partial_i(\alpha\Phi - \beta^j A_j) \tag{3.6}$$

$$\partial_t[\sqrt{\gamma}\Phi] = -\partial_j(\alpha\sqrt{\gamma}A^j - \beta^j[\sqrt{\gamma}\Phi]) - \xi\alpha[\sqrt{\gamma}\Phi] \tag{3.7}$$

Here, the densitized spatial Poynting flux one-form $\tilde{S}_i = \sqrt{\gamma}S_i$ (and S_i comes from $S_\mu - n_\nu T^\nu_{\mathrm{EM}\mu}$), and (Φ, A_i) is the vector potential. We will solve these PDEs using the method of lines, where the right-hand sides of these equations (involving no explicit time derivatives) will be constructed using NRPy+. Two versions of this approach were created: In the

first, the derivatives of the terms on the right-hand side of the $\tilde{S}_i$ evolution equation were taken analytically so that the only derivatives of stored quantities existed, which are easily represented as finite-difference derivatives by NRPy+. In the second, the term $\alpha\sqrt{\gamma}T^j_{\mathrm{EM}i}$ is stored as its own gridfunction. The derivative of that gridfunction was then calculated by NRPy+ using finite-difference derivatives. In both versions, the terms $\alpha\Phi - \beta^j A_j$ and $\alpha\sqrt{\gamma}A^j - \beta^j[\sqrt{\gamma}\Phi]$ were stored and finite-differenced directly. Full details can be found in Appendix C.

After coding up all the above expressions in NRPy+, we proceeded as before in constructing the ETK thorn. We also coded a thorn using NRPy+ to set the initial data for the Exact Wald test. However, try as we might, these thorns simply would not yield data consistent with those from the original `GiRaFFE` code. Thus, we decided to use a new approach, as described in the next chapter. We shifted away from using the Einstein Toolkit, as NRPy+ was now mature enough to perform simulations on its own. We also began to implement the more advanced algorithms used by the original `GiRaFFE`.

Chapter 4

Rewriting GiRaFFE Using NRPy+

It had become apparent that the initial attempt to rewrite the code without the shock-capturing techniques of the original would not work, and that the errors we were seeing were symptomatic of that and not human error in typing the equations. So, these techniques were introduced into the code.

In the next attempt, we reconstructed the primitive variables to cell faces using the piecewise-parabolic method (PPM) [73], then used the Riemann solver of Harten, Lax, van Leer [74], and Einfeldt [75] (HLLE) to compute the contribution of the flux terms in the equations. This was a definite improvement, as we were now able to successfully propagate some simple one-dimensional wave tests. But we still observed oscillations near the wave, which resembled the Gibbs phenomenon. These oscillations were ultimately eliminated by reintroducing the staggered grids from the original GiRaFFE.

Using NRPy+ over the ETK also presents other benefits. One advantage in particular is its ability to easily perform simulations in other coordinate systems using a reference metric formalism. It is helpful to choose the correct coordinate system when solving any physics problem; we ideally want to choose a coordinate system that exploits the symmetries of the system. The systems we want to simulate usually have some symmetries. By simulating them on grids that exploit these symmetries, we can significantly reduce the number of gridpoints we need while maintaining the same accuracy.

Our code thus presents two different GRFFE prescriptions—one on staggered grids, as in the original GiRaFFE, and one on unstaggered grids, as in WhiskyMHD [76]. While the

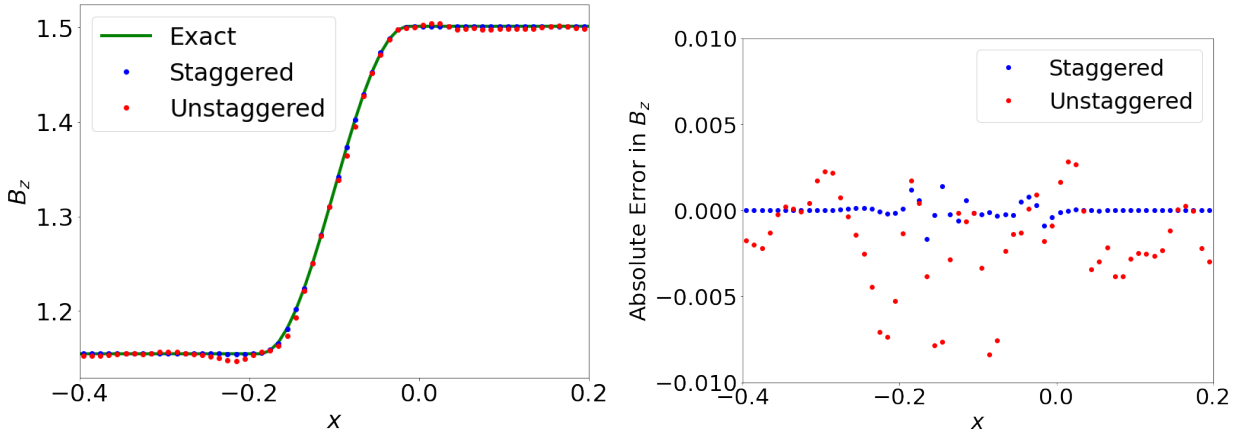


Figure 4.1: **Left panel:** This plot shows a simulation of the Alfvén wave initial data, one of the characteristic modes of GRFFE. It has been evolved to time $t = 0.2$ and shows the exact solution as well as approximate solutions computed with both the staggered and unstaggered prescriptions. Note the additional oscillations near the wave in the unstaggered solution.
Right panel: This plot shows the absolute error between the exact solution and the approximate solutions obtained with both prescriptions.

unstaggered code does not, in general, produce results as reliably as the staggered code at the same resolution, its simplicity can still be beneficial; for instance, in non-Cartesian coordinate systems, coordinate singularities can be easier to avoid when all physical quantities are sampled at the same point within the cell.

4.1 Basic Variables

To describe spacetime, we use a standard 3+1 decomposition in which the line element is given as

$$ds^2 = -\alpha^2 dt^2 + \gamma_{ij}\left(dx^i + \beta^i dt\right)\left(dx^j + \beta^j dt\right),$$

which corresponds to the four-metric $g_{\mu\nu}$ presented in Chapter 1 . Given two adjacent spatial hypersurfaces with coordinate times t_0 and $t_0 + dt$, αdt is the proper time interval between the two hypersurfaces, β^i represents the magnitude of the spatial coordinate shift between them, and γ_{ij} is the spatial three metric within a hypersurface. The initial data that we currently have use either flat spacetime or the shifted Kerr-Schild spacetime.

The GRFFE quantities are divided into two categories. The conservative variables are the densitized Poynting flux $\tilde{S}_i$, the vector potential A_i, and the scalar potential $\psi^6\Phi$; these are our evolved quantities, and their evolution equations are given in conservation form. Our primitive variables are the magnetic field B^i and the Valencia three-velocity $\bar{v}^i$. As part of our shock-capturing scheme, these are reconstructed to cell faces and edges using the piecewise-parabolic method (PPM). In a departure from the old `GiRaFFE`, we use the Valencia three-velocity used by `HydroBase` instead of the drift velocity $v^i = \frac{1}{\alpha}\left(\bar{v}^i + \beta^i\right)$ [77] used by `IllinoisGRMHD`. This does not affect the results, but it does simplify several expressions. For example,

$$\tilde{S}_i = \gamma_{ij}\frac{(v^j + \beta^j)\sqrt{\gamma}B^2}{4\pi\alpha} = \gamma_{ij}\frac{\bar{v}^j\sqrt{\gamma}B^2}{4\pi}.$$

When written in terms of the Valencia velocity, fewer computations are needed to calculate $\tilde{S}_i$.

4.2 The GRFFE Equations

In GRFFE, we use the pure electromagnetic stress energy tensor

$$T^{\mu\nu}_{\rm EM} = b^2u^\mu u^\nu + \frac{1}{2}b^2g^{\mu\nu} - b^\mu b^\nu,$$

where

$$\sqrt{4\pi}b^0 = B^0_{\rm (u)} = \frac{u_jB^j}{\alpha}, \tag{4.1}$$

$$\sqrt{4\pi}b^i = B^i_{\rm (u)} = \frac{B^i + (u_jB^j)u^i}{\alpha u^0}, \tag{4.2}$$

and u^μ is the four-velocity. We do not couple this stress-energy tensor to the Einstein Tensor because the plasma we consider is extremely diffuse.

In place of the magnetic field B^i, we choose to evolve the vector potential A_i. By doing so, we guarantee that the magnetic field is divergenceless to round-off level. We also evolve the Poynting flux $\tilde{S}_i$, which in GRFFE, represents the momentum flux in the plasma, and $\psi^6\Phi$, where $\psi^6 = \sqrt{\gamma}$ and γ is the determinant of the three-metric.

We thus have the following set of equations to evolve:

$$\partial_t \tilde{S}_i + \partial_j \left(\alpha\sqrt{\gamma} T^j_{\mathrm{EM}i}\right) = \frac{1}{2}\alpha\sqrt{\gamma} T^{\mu\nu}_{\mathrm{EM}} \partial_i g_{\mu\nu} \tag{4.3}$$

$$\partial_t[\sqrt{\gamma}\Phi] = -\partial_j(\alpha\sqrt{\gamma}A^j - \beta^j[\sqrt{\gamma}\Phi]) - \xi\alpha[\sqrt{\gamma}\Phi] \tag{4.4}$$

$$\partial_t A_i = \epsilon_{ijk} v^j B^k - \partial_i(\alpha\Phi - \beta^j A_j); \tag{4.5}$$

The gauge condition on the evolution equation for A_i comes from [78].

Here, the four-velocity u^μ is the velocity of the plasma as measured by a normal observer. There are several possible choices of three-velocity that can be made; the ones that we will use here are the drift velocity v^i and the Valencia three-velocity $\bar{v}^i$. Although we have chosen to use the Valencia three-velocity in `GiRaFFE_NRPy` to match the `HydroBase` ETK thorn, we have also frequently made use of a relationship expressing this in terms of the drift velocity, which was used in the original `GiRaFFE`. The usefulness of this relationship to drift velocity extends beyond merely translating the original code. As discussed in [79] (Sec. III.A just above Eq. 45, with a proof in their Appendix A), there is a one-parameter family of velocity definitions that fulfills the GRFFE conditions. The drift velocity sets this parameter to 0, which minimizes the Lorentz factor and *guarantees* that the four-velocity and magnetic fields are orthogonal to each other. This simplifies the form of b^μ and quantities that depend on it.

This is the equation used by `GiRaFFE`, pulled from Eqs. 47 and 85 of [79]:

$$\tilde{S}_i = \gamma_{ij}\frac{\bar{v}^j\sqrt{\gamma}B^2}{4\pi},$$

or

$$\tilde{S}_i = \gamma_{ij}\frac{(v^j + \beta^j)\sqrt{\gamma}B^2}{4\pi\alpha},$$

where $\bar{v}^j$ is the Valencia 3-velocity and v^j is the drift velocity.

We also use the Valencia three-velocity $\bar{v}^i$ and the magnetic field B^i as primitives, which can be defined in terms of the conservative variables:

$$B^i = \epsilon^{ijk}\partial_j A_k \tag{4.6}$$

$$\bar{v}^i = 4\pi\frac{\gamma^{ij}\tilde{S}_j}{\sqrt{\gamma}B^2} \tag{4.7}$$

In this staggered prescription, several gridfunctions are sampled at the cell face, edge, or vertex instead of the center. In particular, at code index `i,j,k`, we sample the following quantities at the listed locations:

- Centers: $B^x(i,j,k)$, $B^y(i,j,k)$, $B^z(i,j,k)$
- Faces: $B^x(i+1/2,j,k)$, $B^y(i,j+1/2,k)$, $B^z(i,j,k+1/2)$
- Edges: $A_x(i,j+1/2,k+1/2)$, $A_y(i+1/2,j,k+1/2)$, $A_z(i+1/2,j+1/2,k)$
- Vertices: $[\sqrt{\gamma}\Phi](i+1/2,j+1/2,k+1/2)$

This requires several portions of the code to be rewritten, as detailed below.

`GiRaFFE_NRPy` only supports uniform Cartesian grids at the moment, but we plan on expanding to the curvilinear coordinate systems that NRPy+ offers in the near future. The grids are, unless otherwise specified, cell-centered. We employ two different prescriptions; in the first, all quantities are sampled at cell centers. In the second, the vector potential, magnetic field, and scalar potential are staggered. Hereinafter, we refer to them as unstaggered and staggered, respectively.

Time evolution is accomplished through the method of lines (MoL) with a Runge-Kutta fourth-order scheme and finite-difference methods and an approximate Riemann solver for spatial derivatives.

Second-order finite-differencing methods are used in the computation of the magnetic field as the curl of the vector potential, as well as in the gauge term of the vector potential evolution equation, $\partial_i(\alpha\Phi - \beta^j A_j)$, and the evolution equation for the scalar potential Φ.

4.3 Overview of the Algorithm

4.3.1 Initial Data

Initial data are computed by the `GiRaFFEfood_NRPy` modules and set the vector potential A_i and Valencia three-velocity $\bar{v}^i$. The scalar potential Φ is always set to 0.

The physical scenarios for which we generate initial data are presented as expressions for the vector potential A_i and electric field E_i. After setting the vector potential directly, the

velocity is calculated as

$$\bar{v}^i = \frac{\epsilon^{ijk} E_j B_k}{B^2},$$

where $\epsilon^{ijk} = [ijk]/\sqrt{\gamma}$, γ is the three-metric determinant, and $B_k = \gamma_{ik} B^i$ is calculated analytically as $B^i = \epsilon^{ijk} \partial_j A_k$.

We then numerically set B^i according to the same formula as above, but using second-order finite-differencing. Finally, the densitized Poynting flux is set numerically as

$$\tilde{S}_i = \frac{\bar{v}_i \sqrt{\gamma} B^2}{4\pi}.$$

4.3.2 Evolution Equations—Unstaggered

In the unstaggered prescription, the gauge terms are calculated first: the gauge term (sometimes called the source term) of the vector potential evolution equation, $\partial_i(\alpha\Phi - \beta^j A_j)$, and the evolution equation for the scalar potential Φ. These are both done by first computing the operand, storing this result, and then computing the derivative operation using second-order finite-differencing.

However, for the flux terms in the evolution equations for the vector potential and densitized Poynting flux, we use something a bit more complicated. The methods we use here involve computing the terms on cell faces and either differencing (for $\tilde{S}_i$) or averaging (for A_i) the values on opposing faces. To do this, we first interpolate the metric quantities to cell faces in direction i with basic, third-order-accurate polynomial interpolation. Then, we reconstruct the primitive variables on the cell faces in the same direction using PPM. The $\tilde{S}_i$ source term $\frac{1}{2}\alpha\sqrt{\gamma} T^{\mu\nu}_{\mathrm{EM}} \partial_i g_{\mu\nu}$ is computed, using the interpolated grid function values to compute the metric derivatives $\partial_i g_{\mu\nu}$.

Then, we solve the one-dimensional Riemann problem approximately on the cell faces using HLLE.

4.3.3 Evolution Equations—Staggered

In the staggered prescription, the steps are done in a slightly different order. Because Φ is sampled at cell vertices and A_i is sampled at cell edges, updating the gauge terms requires

us to interpolate gridfunctions and reconstruct primitives to the appropriate locations; as such, it is performed after the flux terms are calculated.

The right-hand side of the vector potential A_i is calculated using a two-dimensional approximate Riemann solver described by Del Zanna, Bucciantini, and Londrillo [80].

4.3.4 Boundary Conditions—Vector Potential

We do not have exact boundary conditions available for the problems we would like to solve, so we choose simple boundary conditions to preserve numerical stability. For the vector potential, we use simple linear extrapolation boundary conditions.

4.3.5 Primitive Recovery

Once the vector potential has been updated everywhere, the magnetic field is computed on the whole grid using the same function that was used during the initial data step.

We enforce constraints on the densitized Poynting flux; we force it to be strictly orthogonal to the magnetic field and guarantee that the speed of light is not violated (note that $\tilde{S}_i$ is very closely related to $\bar{v}^i$). We then calculate the Valencia three-velocity $\bar{v}^i$. Note that this step is far simpler than in full ideal GRMHD because $\tilde{S}_i\left(\bar{v}^i\right)$ can be inverted analytically.

We then apply an algorithm to the newly updated velocities to preserve any current sheets that form perpendicular to the z-axis and recalculate $\tilde{S}_i$ using the same method we used during the initial data step.

4.3.6 Boundary Conditions—Three-Velocity

Now that the Valencia three-velocity has been updated, we apply copy/outflow boundary conditions. If the velocity is directed inward, it is set to 0; otherwise, it is set to the value at the nearest point (e.g., on the $+x$ face, we set v^i to the values at `i-1`, but if $v^x < 0$ there, we set $v^x = 0$). This helps prevent error from flowing into our simulation. We also choose to apply the boundary condition to the primitive $\bar{v}^i$ instead of the conservative $\tilde{S}_i$ because small errors in conservative quantities can be greatly amplified in our conservative-to-primitive solver, which can lead to unphysical primitives requiring unnecessary fixes.

4.4 GiRaFFEfood_NRPy

Initial data for GiRaFFE_NRPy is provided by GiRaFFEfood_NRPy. This series of modules allows one to easily set up several common GRFFE tests, which we use to validate this code against the original GiRaFFE. The infrastructure is also easily extensible and allows for other initial data to be set up, given the initial vector potential and electric field everywhere. The C code resulting from this then sets the vector potential directly and computes the Valencia three-velocity from the magnetic and electric fields (the magnetic field can either be computed by SymPy from the vector or be provided directly by the user). With these quantities set, all other quantities can be calculated on the grid. Currently, the provided initial data can be either in a spherical or a Cartesian basis, and the resulting code computes the data on a Cartesian grid. However, this is also easily extensible; NRPy's built-in reference metric handling provides the tools necessary to easily use any basis and coordinate system.

4.5 Comparison with GiRaFFE

We validated our new GiRaFFE_NRPy by comparing it with the old GiRaFFE. Because GiRaFFE passes a robust suite of GRFFE tests [72], we can take a "shortcut" in the validation of the new code base and test the agreement between the staggered GiRaFFE and GiRaFFE_NRPy.

In flat spacetime tests, we find that all quantities agree to roundoff error after the first timestep. After subsequent timesteps, the results continue to diverge as the errors accumulate, but this is expected, and both solutions agree at the same expected order.

In curved spacetimes, there are some discrepancies. These are caused by calculations involving the metric, namely, its determinant, inverse, and spatial derivatives thereof. By using SymPy, we obtain code blocks that calculate these quantities more efficiently than they might be written by hand. These slight differences add up and introduce discrepancies between the two solutions, even though all the steps involved are commutative in the continuum limit.

By introducing workarounds to ensure that all calculations are done in the same order, we are able to show that all quantities agree to roundoff error after the first timestep, and

that the results continue to agree well at later times. When we remove these workarounds, the results continue to agree well in spite of the differing amounts of accumulated roundoff error. In fact, in the strong-field region, we see that the new code produces more reasonable results near the horizon.

4.6 Module Validation and Continuous Integration

In addition to our validation of the code as a whole, we implemented unit tests on most of the modules that compose `GiRaFFE_NRPy`. Each unit test exists as its own, separate notebook that validates a portion of our GRFFE algorithm. Because `GiRaFFE_NRPy` itself exists as several Jupyter notebooks, most of the unit tests simply test a single corresponding notebook. The exceptions to this are `GiRaFFEfood_NRPy`, in which the various initial data share a unit test, and the module that calculates the characteristic speeds. That module is very simple and used in the calculation of the fluxes of both $\tilde{S}_i$ and A_i, so any unit test on those modules necessarily validates this one as well.

These unit tests check that the modules produce the expected output in one of two different ways. In either case, we start by generating arbitrary but physically reasonable test data (e.g., input velocities are subluminal, the GRFFE conditions are met). If the old `GiRaFFE` contains a function that we expect to behave identically, we feed the data into both the old and new versions of the function and make sure that we have roundoff-level agreement.

We do not always have a corresponding function from the old `GiRaFFE` with which to compare, however. In some cases, this happens because of the different limitations of NRPy+ and the ETK; in others, namely for modules written for unstaggered grids, sampling quantities at different points within the cell means that we cannot reasonably expect the output to have roundoff-level agreement. In these cases, we check the convergence of the function as described in the introduction 1.

We use continuous integration services to run these unit tests whenever a new commit is pushed to the repository to ensure that the modules are not inadvertently altered.

Chapter 5

Conclusions and Future Work

The potential of the work presented in this dissertation is not limited to the conclusions already presented. We have opened up a new direction of research in black hole binary interactions with our simulations of hyperbolic encounters. We have also furthered development on NRPy+, which has numerous potential uses in the field of numerical relativity.

5.1 Induced Spins from Black Hole Scattering Experiments

In Chapter 2, we investigated an effect that had not previously been considered in full numerical relativity, finding that initially nonspinning black holes can acquire a spin from hyperbolic encounters. Such encounters could occur in globular clusters and galactic centers, so understanding this effect is important to future black hole science. Such systems also provide an important test of post-Newtonian approximations to general relativity because this transfer of momentum to initially nonspinning black holes is only seen at very high order in those approximations.

These simulations show that these close encounters between black holes can induce astrophysically relevant spins; we found spins of up to $a/M = 0.2$ in our simulations. We also showed that these induced spins increase when the initial boost of the black holes is increased or when the impact parameter of their initial trajectories is decreased.

There is a very broad parameter space left to explore. Beyond the already mentioned higher initial boosts and initially spinning black holes, Jaraba and Garcia-Bellido have examined the interaction between black holes of different masses. They have found even higher final spins for the more massive black hole in such a system [81].

Further research in this area could also be supported by adding these initial data to BlackHoles@Home, which would allow us to run the simulations on consumer-grade desktops, ultimately increasing the number of simulations that can be carried out [68, 15, 69].

5.2 Einstein Toolkit Thorns Using NRPy+

The first thorn, `WaveToyNRPy`, is a basic scalar wave solver for the Einstein Toolkit, and propagates monochromatic plane wave and spherical gaussian data in Cartesian coordinates. It provides a basic example of how to write an ETK thorn and has become a useful tutorial for newer users who wish to create an ETK thorn.

The thorn created from `WeylScal4NRPy` follows its example, generating a thorn that computes all the Weyl scalars (which are critical for gravitational wave extraction) as well as several invariants defined in terms the Weyl scalars. The original thorn, `WeylScal4`, was used to validate the expressions generated by `WeylScal4NRPy`. In turn, `WeylScal4NRPy` has helped to form the basis for the current version used in NRPy+ in curvilinear coordinates and was used as a validation test.

5.3 Rewriting `GiRaFFE` using NPRy+

The original `GiRaFFE` is an effective tool for solving the GRFFE equations, but it is limited to Cartesian coordinates. By porting it to NRPy+, we give ourselves a foundation for generalizing the code to arbitrary coordinate systems, thus exploiting the symmetries of the astrophysical systems we wish to study. It also affords us the opportunity to better document the code.

The resulting code, `GiRaFFE_NRPy`, no longer relies on the ETK and has extensive documentation for its algorithms. Along with this documentation, it also has dedicated unit

tests for most of its functions that help validate the code. The code is also validated against the original `GiRaFFE`.

In the course of developing `GiRaFFE_NRPy`, we have stressed the abilities of NRPy+, advancing the state of the project as a whole. Bugs have been uncovered and fixed as part of this development. Quirks and limitations of SymPy have been discovered, which have led us to introduce new techniques into NRPy+ to work around these limitations to implement the algorithms required by `GiRaFFE_NRPy`.

In the course of implementing our unit testing infrastructure for `GiRaFFE_NRPy`, we have set a standard for NRPy+ modules going forward. By implementing best practices, we have been able to validate components of `GiRaFFE_NRPy` individually and monitor them as we implement changes, making it much easier to uncover and eliminate bugs. The unit tests can also be used as templates for future users to write their own unit tests. The unit tests created for `GiRaFFE_NRPy` also provide a guide for future users to implement best practices for themselves to more easily validate their code and to iterate more quickly to improve their codes.

There is also work that remains to be done before `GiRaFFE_NRPy` can fully support curvilinear coordinates. The equations must be rewritten in a covariant way and boundary conditions need to be handled differently in order to avoid placing gridpoints at coordinate singularities. Terrence Pierre Jacques, a WVU graduate student, has already begun to work on this. When it is finished, systems that were previously too memory-intensive to simulate can be studied, such as binary neutron stars whose spin and magnetic axes are misaligned. Such simulations could also possibly be added to BlackHoles@Home, allowing us to use consumer desktops to increase the rate at which we explore the parameter space of any system we study [68, 15, 69].

Appendix A: `WaveToyNRPy`

This appendix contains the notebook that documents the generation of an Einstein Toolkit (ETK) thorn that solves the scalar wave equation, formatted as a PDF. The other ETK thorns are created with similar notebooks. Note that the notebook has been output in landscape mode and that two pages have been included per page of this dissertation.

Tutorial-ETK_thorn-WaveToyNRPy

July 30, 2021

-1 `WaveToyNRPy`: A Simple Einstein Toolkit Thorn for Solving the (Scalar) Wave Equation

-1.1 Authors: Patrick Nelson, Terrence Pierre Jacques, & Zach Etienne

-1.1.1 Formatting improvements courtesy Brandon Clark

Notebook Status: Validated

Validation Notes: As demonstrated in the plot below, the numerical error converges to zero at the expected rate.

-1.1.2 NRPy+ Source Code for this module:

- ScalarWave/ScalarWave_RHSs.py **[tutorial]**
- ScalarWave/InitialData_PlaneWave.py **[tutorial]**

-1.2 Introduction:

This tutorial notebook focuses on how to construct an Einstein Toolkit (ETK) thorn (module) that will set up the expressions for the right-hand sides of $\partial_t u$ and $\partial_t v$ for the scalar wave equation, as defined in the `Tutorial-ScalarWave.ipynb` NRPy+ tutorial notebook. In that module, we used NRPy+ to contruct the SymPy expressions for these scalar wave "time-evolution equations." This thorn is largely based on and should function similarly to the `WaveToyC` thorn included in the Einstein Toolkit (ETK) `CactusWave` arrangement. Furthermore, we generate the C kernels for a number of finite difference orders, so that users are free to use a given option at runtime.

When interfaced properly with the ETK, this module will propagate the initial data for u and v defined in IDScalarWaveNRPy forward in time by integrating the equations for $\partial_t u$ and $\partial_t v$ subject to spatial boundary conditions. The time evolution itself is handled by the `MoL` (Method of Lines) thorn in the `CactusNumerical` arrangement, and the boundary conditions by the `Boundary` thorn in the `CactusBase` arrangement. Specifically, we implement ETK's `NewRad` boundary condition driver, i.e., a radiation (Sommerfeld) boundary condition.

Similar to the IDScalarWaveNRPy module, we will construct the WaveToyNRPy module in two steps.

1. Call on NRPy+ to convert the SymPy expressions for the evolution equations into one C-code kernel.
2. Write the C code and linkages to the Einstein Toolkit infrastructure (i.e., the .ccl files) to complete this Einstein Toolkit module.

In this Jupyter notebook we parallelize the code generation process for the optimized C code kernels needed for different finite difference options from which users may choose at run time. Thus, this Jupyter notebook does not itself run Python code in the relevant C code generation directly. Instead * code blocks needed to generate the C code kernels (i.e., the C-code representations of the basic equations) are simply output to the `$ScalarWaveETKdir/WaveToyETK_C_kernels_codegen.py` file using the `%%writefile` IPython magic command;

0 Table of Contents

This notebook is organized as follows

6. Step 6: Output this notebook to LaTeX-formatted PDF file

1 Step 1: Initialize needed Python/NRPy+ modules [Back to top]

[1]:
```
ScalarWaveETKdir = "ScalarWave_ETK_py_dir"
import cmdline_helper as cmd  # NRPy+: Multi-platform Python command-line interface
import os, sys, shutil        # Standard Python modules for multiplatform OS-level functions

cmd.mkdir(os.path.join(ScalarWaveETKdir))
# Write an empty __init__.py file in this directory so that Python2 can load modules from it.
with open(os.path.join(ScalarWaveETKdir, "__init__.py"), "w") as file:
    pass
```

[2]:
```
%%writefile $ScalarWaveETKdir/WaveToyETK_C_kernels_codegen.py

# Step 1: Import needed core NRPy+ modules
from outputC import lhrh         # NRPy+: Core C code output module
import finite_difference as fin  # NRPy+: Finite difference C code generation module
import NRPy_param_funcs as par   # NRPy+: Parameter interface
import grid as gri               # NRPy+: Functions having to do with numerical grids
import loop as lp                # NRPy+: Generate C code loops
import indexedexp as ixp         # NRPy+: Symbolic indexed expression (e.g., tensors, vectors, etc.) support
import reference_metric as rfm   # NRPy+: Reference metric support
import os, sys                   # Standard Python modules for multiplatform OS-level functions
import time                      # Standard Python module; useful for benchmarking
import ScalarWave.ScalarWaveCurvilinear_RHSs as swrhs

def WaveToyETK_C_kernels_codegen_onepart(params=
        "ThornName=WaveToyNRPy,FD_order=4"):
    # Set default parameters
    ThornName = "WaveToyNRPy"
    FD_order = 4

    import re
```



```
    if params != "":
        params2 = re.sub("^,","",params)
        params = params2.strip()
        splitstring = re.split("=|,", params)

#         if len(splitstring) % 2 != 0:
#             print("outputC: Invalid params string: "+params)
#             sys.exit(1)

        parnm = []
        value = []
        for i in range(int(len(splitstring)/2)):
            parnm.append(splitstring[2*i])
            value.append(splitstring[2*i+1])

        for i in range(len(parnm)):
            parnm.append(splitstring[2*i])
            value.append(splitstring[2*i+1])

        for i in range(len(parnm)):
            if parnm[i] == "ThornName":
                ThornName = value[i]
            elif parnm[i] == "FD_order":
                FD_order = int(value[i])
            else:
                print("WaveToyETK Error: Could not parse input param: "+parnm[i])
                sys.exit(1)

    # Set output directory for C kernels
    outdir = os.path.join(ThornName, "src")  # Main C code output directory

    # Set spatial dimension (must be 3 for WaveToy)
    par.set_parval_from_str("grid::DIM",3)

    # Step 2: Set some core parameters, including CoordSystem MoL timestepping algorithm,
    #                                FD order, floating point precision, and CFL factor:
```

```
    # Choices are: Spherical, SinhSpherical, SinhSphericalv2, Cylindrical, SinhCylindrical,
    #              SymTP, SinhSymTP
    # NOTE: Only CoordSystem == Cartesian makes sense here; new
    #       boundary conditions are needed within the ETK for
    #       Spherical, etc. coordinates.
    CoordSystem     = "Cartesian"

    par.set_parval_from_str("reference_metric::CoordSystem",CoordSystem)
    rfm.reference_metric() # Create ReU, ReDD needed for rescaling B-L initial data, generating WaveToy RHSs, etc.

    # Set the gridfunction memory access type to ETK-like, so that finite_difference
    #    knows how to read and write gridfunctions from/to memory.
    par.set_parval_from_str("grid::GridFuncMemAccess","ETK")
```

```
Overwriting ScalarWave_ETK_py_dir/WaveToyETK_C_kernels_codegen.py
```

2 Step 2: NRPy+-generated C code kernels for solving the Scalar Wave Equation [Back to top]

2.1 Step 2.a: Scalar Wave RHS expressions [Back to top]

Here, we simply call the ScalarWaveCurvilinear.ScalarWaveCurvilinear_RHSs.py; **[tutorial]** NRPy+ Python module to generate the symbolic expressions and then output the finite-difference C code form of the equations using NRPy+'s finite_difference (**tutorial**) C code generation module.

[3]:
```
%%writefile -a $ScalarWaveETKdir/WaveToyETK_C_kernels_codegen.py

    def WaveToy_RHSs__generate_symbolic_expressions():
        ###################################
        # START: GENERATE SYMBOLIC EXPRESSIONS
        print("Generating symbolic expressions for WaveToy RHSs...")
        start = time.time()
```



```
        par.set_parval_from_str("reference_metric::enable_rfm_precompute","True")
        par.set_parval_from_str("reference_metric::rfm_precompute_Ccode_outdir",os.path.join(outdir,"rfm_files/"))

        swrhs.ScalarWaveCurvilinear_RHSs()

        end = time.time()
        print("(BENCH) Finished WaveToy RHS symbolic expressions in "+str(end-start)+" seconds.")
        # END: GENERATE SYMBOLIC EXPRESSIONS
        ###################################

        # Step 2: Register uu_rhs and vv_rhs gridfunctions so
        # they can be written to by NRPy.

        uu_rhs,vv_rhs = gri.register_gridfunctions("AUX",["uu_rhs","vv_rhs"])

        WaveToy_RHSs_SymbExpressions = [\
                       lhrh(lhs=gri.gfaccess("out_gfs","uu_rhs"),rhs=swrhs.uu_rhs),\
                       lhrh(lhs=gri.gfaccess("out_gfs","vv_rhs"),rhs=swrhs.vv_rhs)]

        return WaveToy_RHSs_SymbExpressions

    def WaveToy_RHSs__generate_Ccode(all_RHSs_exprs_list):

        print("Generating C code for WaveToy RHSs (FD_order="+str(FD_order)+") in "+par.
    ↪parval_from_str("reference_metric::CoordSystem")+" coordinates.")
        start = time.time()

        # Store original finite-differencing order:
        FD_order_orig = par.parval_from_str("finite_difference::FD_CENTDERIVS_ORDER")
        # Set new finite-differencing order:
        par.set_parval_from_str("finite_difference::FD_CENTDERIVS_ORDER", FD_order)

        WaveToy_RHSs_string = fin.FD_outputC("returnstring",all_RHSs_exprs_list,
                                             params="outCverbose=False,SIMD_enable=True")
```

```
        filename = "WaveToy_RHSs"+"_FD_order_"+str(FD_order)+".h"
        with open(os.path.join(outdir,filename), "w") as file:
            file.write(lp.
↪loop(["i2","i1","i0"],["cctk_nghostzones[2]","cctk_nghostzones[1]","cctk_nghostzones[0]"],
          ["cctk_lsh[2]-cctk_nghostzones[2]","cctk_lsh[1]-cctk_nghostzones[1]","cctk_lsh[0]-cctk_nghostzones[0]"],
                               ["1","1","SIMD_width"],
                                ["#pragma omp parallel for",
                                 "#include \"rfm_files/rfm_struct__SIMD_outer_read2.h\"",
                                 r"""    #include "rfm_files/rfm_struct__SIMD_outer_read1.h"
    #if (defined __INTEL_COMPILER && __INTEL_COMPILER_BUILD_DATE >= 20180804)
        #pragma ivdep         // Forces Intel compiler (if Intel compiler used) to ignore certain SIMD vector␣
↪dependencies
        #pragma vector always // Forces Intel compiler (if Intel compiler used) to vectorize
    #endif"""],"",
                     "#include \"rfm_files/rfm_struct__SIMD_inner_read0.h\"\n"+WaveToy_RHSs_string))

        # Restore original finite-differencing order:
        par.set_parval_from_str("finite_difference::FD_CENTDERIVS_ORDER", FD_order_orig)

        end = time.time()
        print("(BENCH) Finished WaveToy_RHS C codegen (FD_order="+str(FD_order)+") in " + str(end - start) + "␣
↪seconds.")
```

```
Appending to ScalarWave_ETK_py_dir/WaveToyETK_C_kernels_codegen.py
```

2.2 Step 2.b: Generate algorithm for calling corresponding function within `WaveToyETK_C_kernels_codegen_onepart()` to generate C code kernel [Back to top]

[4]:
```
%%writefile -a $ScalarWaveETKdir/WaveToyETK_C_kernels_codegen.py

    exprs  = WaveToy_RHSs__generate_symbolic_expressions()

    WaveToy_RHSs__generate_Ccode(exprs)
```



```
    # Store all NRPy+ environment variables to an output string so NRPy+ environment from within this subprocess␣
↪can be easily restored
    import pickle
    # https://www.pythonforthelab.com/blog/storing-binary-data-and-serializing/
    outstr = []
    outstr.append(pickle.dumps(len(gri.glb_gridfcs_list)))
    for lst in gri.glb_gridfcs_list:
        outstr.append(pickle.dumps(lst.gftype))
        outstr.append(pickle.dumps(lst.name))
        outstr.append(pickle.dumps(lst.rank))
        outstr.append(pickle.dumps(lst.DIM))

    outstr.append(pickle.dumps(len(par.glb_params_list)))
    for lst in par.glb_params_list:
        outstr.append(pickle.dumps(lst.type))
        outstr.append(pickle.dumps(lst.module))
        outstr.append(pickle.dumps(lst.parname))
        outstr.append(pickle.dumps(lst.defaultval))

    outstr.append(pickle.dumps(len(par.glb_Cparams_list)))
    for lst in par.glb_Cparams_list:
        outstr.append(pickle.dumps(lst.type))
        outstr.append(pickle.dumps(lst.module))
        outstr.append(pickle.dumps(lst.parname))
        outstr.append(pickle.dumps(lst.defaultval))
    return outstr
```

```
Appending to ScalarWave_ETK_py_dir/WaveToyETK_C_kernels_codegen.py
```

2.3 Step 2.c: Generate C code kernels for `WaveToyNRPy` [Back to top]

Here we generate the C code kernels (i.e., the C-code representation of the equations needed) for `WaveToyNRPy`.

2.3.1 Step 2.c.i: Set compile-time and runtime parameters for `WaveToyNRPy` [Back to top]

NRPy+ is a code generation package that is designed to offer maximum flexibility *at the time of C code generation*. As a result, although NRPy+ can in principle output an infinite variety of C code kernels for solving systems of partial differential equations, generally free parameters in each kernel steerable at *runtime* are restricted to simple scalars. This leads to more optimized kernels, but at the expense of flexibility in generating multiple kernels (e.g., one per finite-differencing order). Reducing the number of kernels and adding more flexibility at runtime will be a focus of future work.

For now, `WaveToyNRPy` supports the following runtime options:

- `WaveToyNRPy`: Evolution of the Scalar Wave equation.
  - Finite differencing of orders 2, 4, 6, and 8 via runtime parameter `FD_order`

Next we set up the default parameter lists for `WaveToyETK_C_kernels_codegen_onepart()` for the `WaveToyNRPy` thorn. We set these parameter lists as strings to make parallelizing the code generation far easier (easier to pass a list of strings than a list of function arguments to Python's `multiprocessing.Pool()`).

[5]:
```
# Step 2.e.i: Set compile-time and runtime parameters for WaveToy
#         Runtime parameters for
#         WaveToy:       FD_orders = [2,4,6,8]

paramslist = []
FD_orders = [2,4,6,8]
WhichParamSet = 0
ThornName = "WaveToyNRPy"
for FD_order in FD_orders:
    paramstr= "ThornName="+ThornName+","
    paramstr+= "FD_order="+str(FD_order)+","
    paramslist.append(paramstr)
    WhichParamSet = WhichParamSet + 1

paramslist.sort() # Sort the list alphabetically.
```

2.3.2 Step 2.c.ii: Generate all C-code kernels for `WaveToyNRPy`, in parallel if possible [Back to top]

[6]:
```
nrpy_dir_path = os.path.join(".")
if nrpy_dir_path not in sys.path:
    sys.path.append(nrpy_dir_path)
```



```
# Create all output directories if they do not yet exist
import cmdline_helper as cmd    # NRPy+: Multi-platform Python command-line interface
import shutil                   # Standard Python module for multiplatform OS-level functions
for ThornName in ["WaveToyNRPy"]:
    outrootdir = ThornName
    cmd.mkdir(os.path.join(outrootdir))
    outdir = os.path.join(outrootdir,"src") # Main C code output directory

    # Copy SIMD/SIMD_intrinsics.h to $outdir/SIMD/SIMD_intrinsics.h, replacing
    #   the line "#define REAL_SIMD_ARRAY REAL" with "#define REAL_SIMD_ARRAY CCTK_REAL"
    #   (since REAL is undefined in the ETK, but CCTK_REAL takes its place)
    cmd.mkdir(os.path.join(outdir,"SIMD"))
    import fileinput
    f = fileinput.input(os.path.join(nrpy_dir_path,"SIMD","SIMD_intrinsics.h"))
    with open(os.path.join(outdir,"SIMD","SIMD_intrinsics.h"),"w") as outfile:
        for line in f:
            outfile.write(line.replace("#define REAL_SIMD_ARRAY REAL", "#define REAL_SIMD_ARRAY CCTK_REAL"))

    # Create directory for rfm_files output
    cmd.mkdir(os.path.join(outdir,"rfm_files"))

# Start parallel C code generation (codegen)
# NRPyEnvVars stores the NRPy+ environment from all the subprocesses in the following
#     parallel codegen
NRPyEnvVars = []

import time   # Standard Python module for benchmarking
import logging
start = time.time()
if __name__ == "__main__":
    try:
        if os.name == 'nt':
            # Windows & Jupyter multiprocessing do not mix, so we run in serial on Windows.
            #  Here's why: https://stackoverflow.com/questions/45719956/
 ↪python-multiprocessing-in-jupyter-on-windows-attributeerror-cant-get-attribut
            raise Exception("Parallel codegen currently not available in Windows")
```

```
        # Step 3.d.ii: Import the multiprocessing module.
        import multiprocessing
        print("**************************************")
        print("Starting parallel C kernel codegen...")
        print("**************************************")

        # Step 3.d.iii: Define master function for parallelization.
        #           Note that lambdifying this doesn't work in Python 3
        def master_func(i):
            import ScalarWave_ETK_py_dir.WaveToyETK_C_kernels_codegen as WCk
            return WCk.WaveToyETK_C_kernels_codegen_onepart(params=paramslist[i])

        # Step 3.d.iv: Evaluate list of functions in parallel if possible;
        #           otherwise fallback to serial evaluation:
        pool = multiprocessing.Pool() #processes=len(paramslist))
        NRPyEnvVars.append(pool.map(master_func,range(len(paramslist))))
        pool.terminate()
        pool.join()
    except:
        logging.exception("Ignore this warning/backtrace if on a system in which serial codegen is necessary:")
        print("**************************************")
        print("Starting serial C kernel codegen...")
        print("(If you were running in parallel before,")
        print(" this means parallel codegen failed)")
        print("**************************************")
        # Steps 3.d.ii-iv, alternate: As fallback, evaluate functions in serial.
        #       This will happen on Android and Windows systems
        import ScalarWave_ETK_py_dir.WaveToyETK_C_kernels_codegen as WCk
        import grid as gri
        for param in paramslist:
            gri.glb_gridfcs_list = []
            WCk.WaveToyETK_C_kernels_codegen_onepart(params=param)
        NRPyEnvVars = [] # Reset NRPyEnvVars in case multiprocessing wrote to it and failed.
# # Steps 3.d.ii-iv, alternate: As fallback, evaluate functions in serial.
# #       This will happen on Android and Windows systems
# for param in paramslist:
```



```
#     import grid as gri
#     gri.glb_gridfcs_list = []
#     import ScalarWave_ETK_py_dir.WaveToyETK_C_kernels_codegen as WCk
#     NRPyEnvVars.append(WCk.WaveToyETK_C_kernels_codegen_onepart(params=param))

print("(BENCH) Finished C kernel codegen for WaveToy in "+str(time.time()-start)+" seconds.")
```

```
**************************************
Starting parallel C kernel codegen...
**************************************
Generating symbolic expressions for WaveToy RHSs...Generating symbolic
expressions for WaveToy RHSs...

Generating symbolic expressions for WaveToy RHSs...
Generating symbolic expressions for WaveToy RHSs...
(BENCH) Finished WaveToy RHS symbolic expressions in 0.01595902442932129
seconds.(BENCH) Finished WaveToy RHS symbolic expressions in 0.01606130599975586
seconds.

Generating C code for WaveToy RHSs (FD_order=2) in Cartesian
coordinates.Generating C code for WaveToy RHSs (FD_order=4) in Cartesian
coordinates.

(BENCH) Finished WaveToy RHS symbolic expressions in 0.01991891860961914
seconds.
Generating C code for WaveToy RHSs (FD_order=6) in Cartesian coordinates.
(BENCH) Finished WaveToy RHS symbolic expressions in 0.020931720733642578
seconds.
Generating C code for WaveToy RHSs (FD_order=8) in Cartesian coordinates.
(BENCH) Finished WaveToy_RHS C codegen (FD_order=2) in 0.052909135818481445
seconds.
(BENCH) Finished WaveToy_RHS C codegen (FD_order=4) in 0.08585810661315918
seconds.
(BENCH) Finished WaveToy_RHS C codegen (FD_order=6) in 0.0957636833190918
seconds.
(BENCH) Finished WaveToy_RHS C codegen (FD_order=8) in 0.15320992469787598
seconds.
```

```
(BENCH) Finished C kernel codegen for WaveToy in 0.5363039970397949 seconds.
```

[7]:
```
# Store all NRPy+ environment variables to file so NRPy+ environment from within this subprocess can be easily␣
 ↪restored
import pickle                   # Standard Python module for converting arbitrary data structures to a uniform␣
 ↪format.
import grid as gri              # NRPy+: Functions having to do with numerical grids
import NRPy_param_funcs as par  # NRPy+: Parameter interface

if len(NRPyEnvVars) > 0:
    # https://www.pythonforthelab.com/blog/storing-binary-data-and-serializing/
    grfcs_list = []
    param_list = []
    Cparm_list = []

    for WhichParamSet in NRPyEnvVars[0]:
        # gridfunctions
        i=0
        # print("Length of WhichParamSet:",str(len(WhichParamSet)))
        num_elements = pickle.loads(WhichParamSet[i]); i+=1
        for lst in range(num_elements):
            grfcs_list.append(gri.glb_gridfc(gftype=pickle.loads(WhichParamSet[i+0]),
                                             name  =pickle.loads(WhichParamSet[i+1]),
                                             rank  =pickle.loads(WhichParamSet[i+2]),
                                             DIM   =pickle.loads(WhichParamSet[i+3]))) ; i+=4
        # parameters
        num_elements = pickle.loads(WhichParamSet[i]); i+=1
        for lst in range(num_elements):
            param_list.append(par.glb_param(type      =pickle.loads(WhichParamSet[i+0]),
                                            module    =pickle.loads(WhichParamSet[i+1]),
                                            parname   =pickle.loads(WhichParamSet[i+2]),
                                            defaultval=pickle.loads(WhichParamSet[i+3]))) ; i+=4
        # Cparameters
        num_elements = pickle.loads(WhichParamSet[i]); i+=1
        for lst in range(num_elements):
            Cparm_list.append(par.glb_Cparam(type      =pickle.loads(WhichParamSet[i+0]),
                                             module    =pickle.loads(WhichParamSet[i+1]),
```



```
                                             parname   =pickle.loads(WhichParamSet[i+2]),
                                             defaultval=pickle.loads(WhichParamSet[i+3]))) ; i+=4
    grfcs_list_uniq = []
    for gf_ntuple_stored in grfcs_list:
        found_gf = False
        for gf_ntuple_new in grfcs_list_uniq:
            if gf_ntuple_new == gf_ntuple_stored:
                found_gf = True
        if found_gf == False:
            grfcs_list_uniq.append(gf_ntuple_stored)

    param_list_uniq = []
    for pr_ntuple_stored in param_list:
        found_pr = False
        for pr_ntuple_new in param_list_uniq:
            if pr_ntuple_new == pr_ntuple_stored:
                found_pr = True
        if found_pr == False:
            param_list_uniq.append(pr_ntuple_stored)

    # Set glb_paramsvals_list:
    # Step 1: Reset all paramsvals to their defaults
    par.glb_paramsvals_list = []
    for parm in param_list_uniq:
        par.glb_paramsvals_list.append(parm.defaultval)

    Cparm_list_uniq = []
    for Cp_ntuple_stored in Cparm_list:
        found_Cp = False
        for Cp_ntuple_new in Cparm_list_uniq:
            if Cp_ntuple_new == Cp_ntuple_stored:
                found_Cp = True
        if found_Cp == False:
            Cparm_list_uniq.append(Cp_ntuple_stored)

    gri.glb_gridfcs_list = []
```

```
    par.glb_params_list  = []
    par.glb_Cparams_list = []

    gri.glb_gridfcs_list = grfcs_list_uniq
    par.glb_params_list  = param_list_uniq
    par.glb_Cparams_list = Cparm_list_uniq
```

3 Step 3: ETK `ccl` file generation [Back to top]

3.1 Step 3.a: `param.ccl`: specify free parameters within `WaveToyNRPy` [Back to top]

All parameters necessary for the computation of the Scalar Wave right-hand side (RHS) expressions are registered within NRPy+; we use this information to automatically generate `param.ccl`. NRPy+ also specifies default values for each parameter.

More information on `param.ccl` syntax can be found in the official Einstein Toolkit documentation.

[8]:
```
def keep_param__return_type(paramtuple):
    keep_param = True # We'll not set some parameters in param.ccl;
                      #   e.g., those that should be #define'd like M_PI.
    typestring = ""
    # Separate thorns within the ETK take care of grid/coordinate parameters;
    #   thus we ignore NRPy+ grid/coordinate parameters:
    if paramtuple.module == "grid" or paramtuple.module == "reference_metric":
        keep_param = False

    partype = paramtuple.type
    if partype == "bool":
        typestring += "BOOLEAN "
    elif partype == "REAL":
        if paramtuple.defaultval != 1e300: # 1e300 is a magic value indicating that the C parameter should be␣
 ↪mutable
            typestring += "CCTK_REAL "
        else:
```



```
            keep_param = False
    elif partype == "int":
        typestring += "CCTK_INT "
    elif partype == "#define":
        keep_param = False
    elif partype == "char":
        # FIXME: char/string parameter types should in principle be supported
        print("Error: parameter "+paramtuple.module+"::"+paramtuple.parname+
              " has unsupported type: \""+ paramtuple.type + "\"")
        sys.exit(1)
    else:
        print("Error: parameter "+paramtuple.module+"::"+paramtuple.parname+
              " has unsupported type: \""+ paramtuple.type + "\"")
        sys.exit(1)
    return keep_param, typestring

with open(os.path.join(outrootdir,"param.ccl"), "w") as file:
    file.write("""
# This param.ccl file was automatically generated by NRPy+.
#   You are advised against modifying it directly.

restricted:
CCTK_REAL wavespeed "The speed at which the wave propagates"
{
 *:* :: "Wavespeed as a multiple of c"
} 1.0

restricted:

CCTK_INT FD_order "Finite-differencing order"
{\n""")
    FDorders = []
    for _root, _dirs, files in os.walk(os.path.join(ThornName,"src")): # _root,_dirs unused.
        for Cfilename in files:
            if (".h" in Cfilename) and ("RHSs" in Cfilename) and ("FD" in Cfilename) and ("intrinsics" not in␣
 ↪Cfilename):
```

```
            array = Cfilename.replace(".","_").split("_")
            FDorders.append(int(array[-2]))
    FDorders.sort()
    for order in FDorders:
        file.write(" "+str(order)+":"+str(order)+"   :: \"finite-differencing order = "+str(order)+"\"\n")
    FDorder_default = 4
    if FDorder_default not in FDorders:
        print("WARNING: 4th-order FD kernel was not output!?! Changing default FD order to "+str(FDorders[0]))
        FDorder_default = FDorders[0]
    file.write("} "+str(FDorder_default)+ "\n\n") # choose 4th order by default, consistent with ML_WaveToy
```

3.2 Step 3.b: `interface.ccl`: define needed gridfunctions; provide keywords denoting what this thorn provides and what it should inherit from other thorns [Back to top]

`interface.ccl` declares all gridfunctions and determines how `WaveToyNRPy` interacts with other Einstein Toolkit thorns.

The official Einstein Toolkit documentation defines what must/should be included in an `interface.ccl` file.

[9]:
```
evol_gfs_list    = []
for i in range(len(gri.glb_gridfcs_list)):
    if gri.glb_gridfcs_list[i].gftype == "EVOL":
        evol_gfs_list.append(   gri.glb_gridfcs_list[i].name+"GF")

# NRPy+'s finite-difference code generator assumes gridfunctions
#    are alphabetized; not sorting may result in unnecessary
#    cache misses.
evol_gfs_list.sort()

with open(os.path.join(outrootdir,"interface.ccl"), "w") as file:
    file.write("""
# With "implements", we give our thorn its unique name.
implements: WaveToyNRPy

# By "inheriting" other thorns, we tell the Toolkit that we
#   will rely on variables/function that exist within those
```



```
#   functions.
inherits: Boundary grid MethodofLines

# Needed functions and #include's:
USES INCLUDE: Symmetry.h
USES INCLUDE: Boundary.h

# Needed Method of Lines function
CCTK_INT FUNCTION MoLRegisterEvolvedGroup(CCTK_INT IN EvolvedIndex,                                                 ␣
 ↪CCTK_INT IN RHSIndex)
REQUIRES FUNCTION MoLRegisterEvolvedGroup

# Needed Boundary Conditions function
CCTK_INT FUNCTION GetBoundarySpecification(CCTK_INT IN size, CCTK_INT OUT ARRAY nboundaryzones, CCTK_INT OUT ARRAY␣
 ↪is_internal, CCTK_INT OUT ARRAY is_staggered, CCTK_INT OUT ARRAY shiftout)
USES FUNCTION GetBoundarySpecification

CCTK_INT FUNCTION SymmetryTableHandleForGrid(CCTK_POINTER_TO_CONST IN cctkGH)
USES FUNCTION SymmetryTableHandleForGrid

CCTK_INT FUNCTION Boundary_SelectVarForBC(CCTK_POINTER_TO_CONST IN GH, CCTK_INT IN faces, CCTK_INT IN␣
 ↪boundary_width, CCTK_INT IN table_handle, CCTK_STRING IN var_name, CCTK_STRING IN bc_name)
USES FUNCTION Boundary_SelectVarForBC

# Needed for EinsteinEvolve/NewRad outer boundary condition driver:
CCTK_INT FUNCTION                          \
    NewRad_Apply                           \
        (CCTK_POINTER_TO_CONST IN cctkGH, \
         CCTK_REAL ARRAY IN var,           \
         CCTK_REAL ARRAY INOUT rhs,        \
         CCTK_REAL IN var0,                \
         CCTK_REAL IN v0,                  \
         CCTK_INT IN radpower)
REQUIRES FUNCTION NewRad_Apply

# Tell the Toolkit that we want all gridfunctions
```

```
#    to be visible to other thorns by using
#    the keyword "public". Note that declaring these
#    gridfunctions *does not* allocate memory for them;
#    that is done by the schedule.ccl file.

# FIXME: add info for symmetry conditions:
#    https://einsteintoolkit.org/thornguide/CactusBase/SymBase/documentation.html

""")

# Next we declare gridfunctions based on their corresponding gridfunction groups as registered within NRPy+

    def output_list_of_gfs(gfs_list,description="User did not provide description"):
        gfsstr = "    "
        for i in range(len(gfs_list)):
            gfsstr += gfs_list[i]
            if i != len(gfs_list)-1:
                gfsstr += "," # This is a comma-separated list of gridfunctions
            else:
                gfsstr += "\n} \""+description+"\"\n\n"
        return gfsstr
    # First EVOL type:
    file.write("public:\n")
    file.write("cctk_real scalar_fields type = GF Timelevels=3 tags=\'tensortypealias=\"Scalar\"\'\n{\n")
    file.write(output_list_of_gfs(evol_gfs_list,"The evolved scalar fields"))
    # Second EVOL right-hand-sides
    file.write("public:\n")
    file.write("cctk_real scalar_fields_rhs type = GF Timelevels=3 tags=\'tensortypealias=\"Scalar\"\'\n{\n")
    rhs_list = []
    for gf in evol_gfs_list:
        rhs_list.append(gf.replace("GF","")+"_rhsGF")
    file.write(output_list_of_gfs(rhs_list,"The rhs of the scalar fields"))
```


3.3 Step 3.c: `schedule.ccl`: schedule all functions used within `WaveToyNRPy`, specify data dependencies within said functions, and allocate memory for gridfunctions [Back to top]

Official documentation on constructing ETK `schedule.ccl` files is found here.

[10]:
```
with open(os.path.join(outrootdir,"schedule.ccl"), "w") as file:
    file.write("""
# Allocate storage for all 2 gridfunction groups used in WaveToyNRPy
STORAGE: scalar_fields_rhs[3]
STORAGE: scalar_fields[3]

# The following scheduler is based on Lean/LeanBSSNMoL/schedule.ccl

schedule WaveToyNRPy_Banner at STARTUP
{
  LANG: C
  OPTIONS: meta
} "Output ASCII art banner"

schedule WaveToyNRPy_Symmetry_registration at BASEGRID
{
  LANG: C
  OPTIONS: Global
} "Register symmetries, the CartGrid3D way."

schedule WaveToyNRPy_zero_rhss at BASEGRID after WaveToyNRPy_Symmetry_registration
{
  LANG: C
} "Idea from Lean: set all rhs functions to zero to prevent spurious nans"


schedule GROUP ApplyBCs as WaveToyNRPy_ApplyBCs at CCTK_INITIAL
{
} "Apply boundary conditions"
```

```
# MoL: registration

schedule WaveToyNRPy_MoL_registration in MoL_Register
{
  LANG: C
  OPTIONS: META
} "Register variables for MoL"


# MoL: compute RHSs, etc

schedule WaveToyNRPy_set_rhs as WaveToy_Evolution IN MoL_CalcRHS
{
  LANG: C
  READS: uuGF(Everywhere)
  READS: vvGF(Everywhere)
  WRITES: uu_rhsGF(Interior)
  WRITES: vv_rhsGF(Interior)
} "Evolution of 3D wave equation"

schedule WaveToyNRPy_NewRad in MoL_CalcRHS after WaveToyNRPy_RHS
{
  LANG: C
} "NewRad boundary conditions, scheduled right after RHS eval."

schedule WaveToyNRPy_BoundaryConditions_register_evolved_gfs in MoL_PostStep
{
  LANG: C
  OPTIONS: LEVEL
  SYNC: scalar_fields
} "Apply boundary conditions and perform AMR+interprocessor synchronization"

schedule GROUP ApplyBCs as WaveToyNRPy_ApplyBCs in MoL_PostStep after␣
 ↪WaveToyNRPy_BoundaryConditions_register_evolved_gfs
{
```



```
} "Group for applying boundary conditions"
""")
```

4 Step 4: C driver functions for ETK registration & NRPy+-generated kernels [Back to top]

Now that we have constructed the basic C code kernels and the needed Einstein Toolkit `ccl` files, we next write the driver functions for registering `WaveToyNRPy` within the Toolkit and the C code kernels. Each of these driver functions is called directly from schedule.ccl.

Step 4.a: Needed ETK functions: Banner, Symmetry registration, Parameter sanity check, Method of Lines (`MoL`) registration, Boundary condition [Back to top]

4.0.1 To-do: Parameter sanity check function. E.g., error should be thrown if `cctk_nghostzones[]` is set too small for the chosen finite-differencing order within NRPy+.

[11]:
```
make_code_defn_list = []
def append_to_make_code_defn_list(filename):
    if filename not in make_code_defn_list:
        make_code_defn_list.append(filename)
    return os.path.join(outdir,filename)
```

[12]:
```
# First the ETK banner code, proudly showing the NRPy+ banner
import NRPy_logo as logo

with open(append_to_make_code_defn_list("Banner.c"),"w") as file:
    file.write("""
#include <stdio.h>

void WaveToyNRPy_Banner()
{
    """)
    logostr = logo.print_logo(print_to_stdout=False)
    file.write("printf(\"WaveToyNRPy: another Einstein Toolkit thorn generated by\\n\");\n")
    for line in logostr.splitlines():
```

```
        file.write("    printf(\""+line+"\\n\");\n")
    file.write("}\n")
```

[13]:
```
# Next register symmetries
with open(append_to_make_code_defn_list("Symmetry_registration_oldCartGrid3D.c"),"w") as file:
    file.write("""
#include "cctk.h"
#include "cctk_Arguments.h"
#include "cctk_Parameters.h"
#include "Symmetry.h"

void WaveToyNRPy_Symmetry_registration(CCTK_ARGUMENTS)
{
  DECLARE_CCTK_ARGUMENTS;
  DECLARE_CCTK_PARAMETERS;

  // Stores gridfunction parity across x=0, y=0, and z=0 planes, respectively
  int sym[3];

  // Next register parities for each gridfunction based on its name
  //    (to ensure this algorithm is robust, gridfunctions with integers
  //     in their base names are forbidden in NRPy+).
""")
    full_gfs_list = []
    full_gfs_list.extend(evol_gfs_list)
    for gf in full_gfs_list:
        file.write("""
  // Default to scalar symmetry:
  sym[0] = 1; sym[1] = 1; sym[2] = 1;
  // Now modify sym[0], sym[1], and/or sym[2] as needed
  //    to account for gridfunction parity across
  //    x=0, y=0, and/or z=0 planes, respectively
""")
        # If gridfunction name does not end in a digit, by NRPy+ syntax, it must be a scalar
        if gf[len(gf) - 1].isdigit() == False:
            pass # Scalar = default
        elif len(gf) > 2:
```



```
            # Rank-1 indexed expression (e.g., vector)
            if gf[len(gf) - 2].isdigit() == False:
                if int(gf[-1]) > 2:
                    print("Error: Found invalid gridfunction name: "+gf)
                    sys.exit(1)
                symidx = gf[-1]
                file.write("  sym["+symidx+"] = -1;\n")
            # Rank-2 indexed expression
            elif gf[len(gf) - 2].isdigit() == True:
                if len(gf) > 3 and gf[len(gf) - 3].isdigit() == True:
                    print("Error: Found a Rank-3 or above gridfunction: "+gf+", which is at the moment unsupported.
 ↪")
                    print("It should be easy to support this if desired.")
                    sys.exit(1)
                symidx0 = gf[-2]
                file.write("  sym["+symidx0+"] *= -1;\n")
                symidx1 = gf[-1]
                file.write("  sym["+symidx1+"] *= -1;\n")
        else:
            print("Don't know how you got this far with a gridfunction named "+gf+", but I'll take no more of this␣
 ↪nonsense.")
            print("   Please follow best-practices and rename your gridfunction to be more descriptive")
            sys.exit(1)
        file.write("  SetCartSymVN(cctkGH, sym, \"WaveToyNRPy::"+gf+"\");\n")
    file.write("}\n")
```

[14]:
```
# Next register symmetries
import loop as lp                # NRPy+: Generate C code loops
with open(append_to_make_code_defn_list("zero_rhss.c"),"w") as file:
    file.write("""
#include "cctk.h"
#include "cctk_Arguments.h"
#include "cctk_Parameters.h"
#include "Symmetry.h"

void WaveToyNRPy_zero_rhss(CCTK_ARGUMENTS)
```

```
{
  DECLARE_CCTK_ARGUMENTS;
  DECLARE_CCTK_PARAMETERS;
""")
    set_rhss_to_zero = ""
    for gf in rhs_list:
        set_rhss_to_zero += gf+"[CCTK_GFINDEX3D(cctkGH,i0,i1,i2)] = 0.0;\n"

    file.write(lp.loop(["i2","i1","i0"],["0", "0", "0"],
                       ["cctk_lsh[2]","cctk_lsh[1]","cctk_lsh[0]"],
                       ["1","1","1"],
                        ["#pragma omp parallel for","","",],"",set_rhss_to_zero))
    file.write("}\n")
```

[15]:
```
# Next registration with the Method of Lines thorn
with open(append_to_make_code_defn_list("MoL_registration.c"),"w") as file:
    file.write("""
//------------------------------------------------------------------------
// Register with the Method of Lines time stepper
// (MoL thorn, found in arrangements/CactusBase/MoL)
// MoL documentation located in arrangements/CactusBase/MoL/doc
//------------------------------------------------------------------------
#include <stdio.h>

#include "cctk.h"
#include "cctk_Parameters.h"
#include "cctk_Arguments.h"

#include "Symmetry.h"

void WaveToyNRPy_MoL_registration(CCTK_ARGUMENTS)
{
  DECLARE_CCTK_ARGUMENTS;
  DECLARE_CCTK_PARAMETERS;

  CCTK_INT ierr = 0, group, rhs;
```



```
  // Register evolution & RHS gridfunction groups with MoL, so it knows

  group = CCTK_GroupIndex("WaveToyNRPy::scalar_fields");
  rhs = CCTK_GroupIndex("WaveToyNRPy::scalar_fields_rhs");
  ierr += MoLRegisterEvolvedGroup(group, rhs);

  if (ierr) CCTK_ERROR("Problems registering with MoL");
}
""")
```

[16]:
```
# Next register with the boundary conditions thorns.
# PART 1: Set BC type to "none" for all variables
# Since we choose NewRad boundary conditions, we must register all
#   gridfunctions to have boundary type "none". This is because
#   NewRad is seen by the rest of the Toolkit as a modification to the
#   RHSs.

# This code is based on Kranc's McLachlan/ML_BSSN/src/Boundaries.cc code.
with open(append_to_make_code_defn_list("BoundaryConditions_register_evolved_gfs.c"),"w") as file:
    file.write("""
#include "cctk.h"
#include "cctk_Arguments.h"
#include "cctk_Parameters.h"
#include "cctk_Faces.h"
#include "util_Table.h"
#include "Symmetry.h"

// Set `none` boundary conditions on RHSs, as these are set via NewRad.
void WaveToyNRPy_BoundaryConditions_register_evolved_gfs(CCTK_ARGUMENTS)
{
  DECLARE_CCTK_ARGUMENTS;
  DECLARE_CCTK_PARAMETERS;

  CCTK_INT ierr CCTK_ATTRIBUTE_UNUSED = 0;
""")
    for gf in evol_gfs_list:
        file.write("""
```

```
  ierr = Boundary_SelectVarForBC(cctkGH, CCTK_ALL_FACES, 1, -1, "WaveToyNRPy::"""+gf+""""", "none");
  if (ierr < 0) CCTK_ERROR("Failed to register BC for WaveToyNRPy::"""+gf+"""!");

""")
    file.write("}")

# PART 2: Set C code for calling NewRad BCs
#   As explained in lean_public/LeanBSSNMoL/src/calc_bssn_rhs.F90,
#   the function NewRad_Apply takes the following arguments:
#   NewRad_Apply(cctkGH, var, rhs, var0, v0, radpower),
#     which implement the boundary condition:
#       var  =  var_at_infinite_r + u(r-var_char_speed*t)/r^var_radpower
#  Obviously for var_radpower>0, var_at_infinite_r is the value of
#    the variable at r->infinity. var_char_speed is the propagation
#    speed at the outer boundary, and var_radpower is the radial
#    falloff rate.

with open(append_to_make_code_defn_list("BoundaryCondition_NewRad.c"),"w") as file:
    file.write("""
#include <math.h>

#include "cctk.h"
#include "cctk_Arguments.h"
#include "cctk_Parameters.h"

void WaveToyNRPy_NewRad(CCTK_ARGUMENTS) {
  DECLARE_CCTK_ARGUMENTS;
  DECLARE_CCTK_PARAMETERS;

  const CCTK_REAL uu_at_infinite_r = 1.0;
  const CCTK_REAL vv_at_infinite_r = 0.0;
  const CCTK_REAL var_char_speed    = 1.0;
  const CCTK_REAL var_radpower      = 3.0;

  NewRad_Apply(cctkGH, uuGF, uu_rhsGF, uu_at_infinite_r, var_char_speed, var_radpower);
  NewRad_Apply(cctkGH, vvGF, vv_rhsGF, vv_at_infinite_r, var_char_speed, var_radpower);
```



```
}
""")
```

4.1 Step 4.b: Evaluate scalar wave right-hand-sides (RHSs) [Back to top]

[17]:
```
##########################
# WaveToy_RHSs
##########################
common_includes = """
#include <math.h>
#include "cctk.h"
#include "cctk_Arguments.h"
#include "cctk_Parameters.h"
#include "SIMD/SIMD_intrinsics.h"

"""
common_preloop = """
DECLARE_CCTK_ARGUMENTS;
const CCTK_REAL NOSIMDinvdx0 = 1.0/CCTK_DELTA_SPACE(0);
const REAL_SIMD_ARRAY invdx0 = ConstSIMD(NOSIMDinvdx0);
const CCTK_REAL NOSIMDinvdx1 = 1.0/CCTK_DELTA_SPACE(1);
const REAL_SIMD_ARRAY invdx1 = ConstSIMD(NOSIMDinvdx1);
const CCTK_REAL NOSIMDinvdx2 = 1.0/CCTK_DELTA_SPACE(2);
const REAL_SIMD_ARRAY invdx2 = ConstSIMD(NOSIMDinvdx2);
"""
```

[18]:
```
path = os.path.join(ThornName,"src")
WaveToyETK_src_filelist = []
for _root, _dirs, files in os.walk(path):  # _root, _dirs unused.
    for filename in files:
        WaveToyETK_src_filelist.append(filename)
WaveToyETK_src_filelist.sort() # Sort the list in place.

WaveToy_FD_orders_output = []
for filename in WaveToyETK_src_filelist:
```

```
    if "WaveToy_RHSs_" in filename:
        array = filename.replace(".","_").split("_")
        FDorder =  int(array[-2])
        if FDorder not in WaveToy_FD_orders_output:
            WaveToy_FD_orders_output.append(FDorder)
WaveToy_FD_orders_output.sort()

##########################
# Output WaveToy RHSs driver function
outstr = common_includes
for filename in WaveToyETK_src_filelist:
    if ("WaveToy_RHSs_" in filename) and (".h" in filename):
        outstr += """extern void """+filename.replace(".h", "(CCTK_ARGUMENTS);") + "\n"

outstr += """
void WaveToyNRPy_set_rhs(CCTK_ARGUMENTS) {
DECLARE_CCTK_ARGUMENTS;

const CCTK_INT *FD_order = CCTK_ParameterGet("FD_order","WaveToyNRPy",NULL);

"""
for filename in WaveToyETK_src_filelist:
    if ("WaveToy_RHSs_" in filename) and (".h" in filename):
        array = filename.replace(".", "_").split("_")
        outstr += "    if(*FD_order == " + str(array[-2]) + ") {\n"
        outstr += "        " +filename.replace(".h", "(CCTK_PASS_CTOC);") + "\n"
        outstr += "    }\n"
outstr += "} // END FUNCTION\n"
# Add C code string to dictionary (Python dictionaries are immutable)
with open(append_to_make_code_defn_list("ScalarWave_RHSs.c"),"w") as file:
    file.write(outstr)

def SIMD_declare_C_params():
    SIMD_declare_C_params_str = ""
    for i in range(len(par.glb_Cparams_list)):
        # keep_param is a boolean indicating whether we should accept or reject
```



```
        #    the parameter. singleparstring will contain the string indicating
        #    the variable type.
        keep_param, singleparstring = keep_param__return_type(par.glb_Cparams_list[i])

        if (keep_param) and ("CCTK_REAL" in singleparstring):
            parname = par.glb_Cparams_list[i].parname
            SIMD_declare_C_params_str += "    const "+singleparstring + "*NOSIMD"+parname+\
                                         " = CCTK_ParameterGet(\""+parname+"\",\"WaveToyNRPy\",NULL);\n"
            SIMD_declare_C_params_str += "    const REAL_SIMD_ARRAY "+parname+" = ConstSIMD(*NOSIMD"+parname+");\n"
    return SIMD_declare_C_params_str

# # Create functions for the largest C kernels (WaveToy RHSs and Ricci) and output
# #    the .h files to .c files with function wrappers; delete original .h files
path = os.path.join(ThornName, "src")
for filename in WaveToyETK_src_filelist:
    if ("WaveToy_RHSs_" in filename) and (".h" in filename):
        outstr = common_includes + "void "+filename.replace(".h","")+"(CCTK_ARGUMENTS) {\n"
        outstr += common_preloop+SIMD_declare_C_params()
        with open(os.path.join(path,filename), "r") as currfile:
            outstr += currfile.read()
         # Now that we've inserted the contents of the kernel into this file,
        #     we delete the file containing the kernel
        os.remove(os.path.join(path,filename))
        outstr += "} // END FUNCTION\n"
        # Add C code string to dictionary (Python dictionaries are immutable)
        with open(append_to_make_code_defn_list(filename.replace(".h",".c")),"w") as file:
            file.write(outstr)
```

4.2 Step 4.b: `make.code.defn`: List of all C driver functions needed to compile `WaveToyNRPy` [Back to top]

When constructing each C code driver function above, we called the `append_to_make_code_defn_list()` function, which built a list of each C code driver file. We'll now add each of those files to the `make.code.defn` file, used by the Einstein Toolkit's build system.

[19]:
```
with open(os.path.join(outdir,"make.code.defn"), "w") as file:
    file.write("""
```

```
# Main make.code.defn file for thorn WaveToyNRPy

# Source files in this directory
SRCS =""")
    filestring = ""
    for i in range(len(make_code_defn_list)):
        filestring += "      "+make_code_defn_list[i]
        if i != len(make_code_defn_list)-1:
            filestring += " \\\n"
        else:
            filestring += "\n"
    file.write(filestring)
```

5 Step 5: Code validation, convergence tests [Back to top]

We have performed a number of convergence tests with WaveToyNRPy and IDScalarWaveNRPy within the ETK, which are presented below.

5.1 Step 5.a: Monochromatic plane wave convergence tests [Back to top]

Three-dimensional scalar wave equation code tests, adopting fourth-order finite differencing, coupled to RK4 method-of-lines for time integration

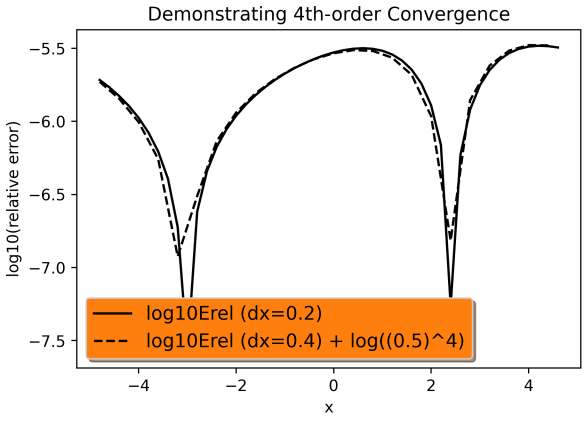
Inside the directory *WaveToyNRPy/example_parfiles/4thOrder_ConvergenceTests/* are the files used for this convergence test: 1. **planewave_along_3D_diagonal*.par** : ETK parameter files needed for performing the 3D tests. These parameter files set up a sinusoidal plane wave propagating along the x=y=z diagonal of a 3D numerical grid that extends from -12.8 to $+12.8$ along the x-, y-, and z-axes (in units of $\omega = k = c = 1$). The parameter files are identical, except one has grid resolution that is twice as high (so the errors should drop in the higher resolution case by a factor of 2^4, because we adopt fourth-order-convergent timestepping and spatial finite differencing). 1. **runscript.sh** : Runs the cactus executable (assumed to be named *cactus_etilgrmhd-FD4*) for all of the above parameter files. 1. **convert_IOASCII_1D_to_gnuplot.sh** : Used by **runscript.sh** to convert the 1D output from the execution into a format that gnuplot can recognize. 1. **gnuplot_script** : Script for creating code validation convergence plots with gnuplot.

Fourth-order code validation test results:

The plot below shows the discrepancy between numerical and exact solutions to the scalar wave equation at two different resolutions: the dashed line is low resolution ($\Delta x_{\rm low} = 0.4$) and the solid line is high resolution ($\Delta x_{\rm high} = 0.2$). Because this test adopts fourth-order finite differencing for spatial derivatives and fourth-order Runge-Kutta (RK4) for timestepping, we would expect this error to drop by a factor of approximately $(\Delta x_{\rm low}/\Delta x_{\rm high})^4 = (0.4/0.2)^4 = 2^4 = 16$ when going from low to high resolution, and after rescaling the error in the low-resolution case, we see that indeed it overlaps the low-resolution result quite nicely, confirming fourth-order convergence.

[20]:
```
from IPython.display import Image
Image("./WaveToyNRPy/example_parfiles/4thOrder_ConvergenceTests/convergence-RK4-FD4-3D.png", width=500, height=500)
```

[20]:



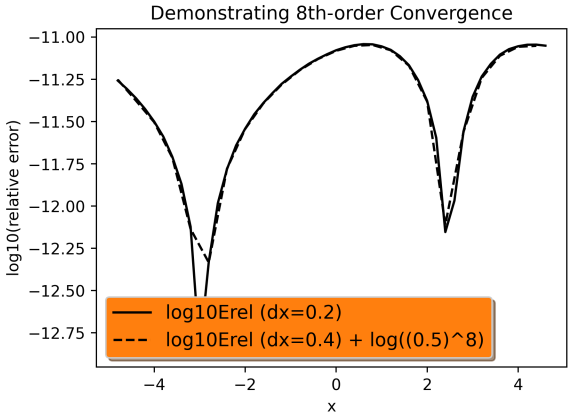
Three-dimensional scalar wave equation code tests for monochromatic plane wave initial data adopting *eighth*-order finite differencing, coupled to *RK8* method-of-lines for time integration Inside the directory *WaveToyNRPy/example_parfiles/8thOrder_ConvergenceTests/* are the files used for this convergence test: 1. **planewave_along_3D_diagonal*.par** : ETK parameter files needed for performing the 3D tests. These parameter files set up a sinusoidal plane wave propagating along the x=y=z diagonal of a 3D numerical grid that extends from -13.6 to $+13.6$ along the x-, y-, and

z-axes (in units of $\omega = k = c = 1$). The parameter files are identical, except one has grid resolution that is twice as high (so the errors should drop in the higher-resolution case by a factor of 2^8, because we adopt eighth-order-convergent timestepping and spatial finite differencing). 1. **runscript.sh** : Runs the cactus executable (assumed to be named *cactus_etilgrmhd-FD8*) for all of the above parameter files. 1. **convert_IOASCII_1D_to_gnuplot.sh** : Used by **runscript.sh** to convert the 1D output from the execution into a format that gnuplot can recognize. 1. **gnuplot_script** : Script for creating code validation convergence plots with gnuplot.

Eighth-order code validation test results:

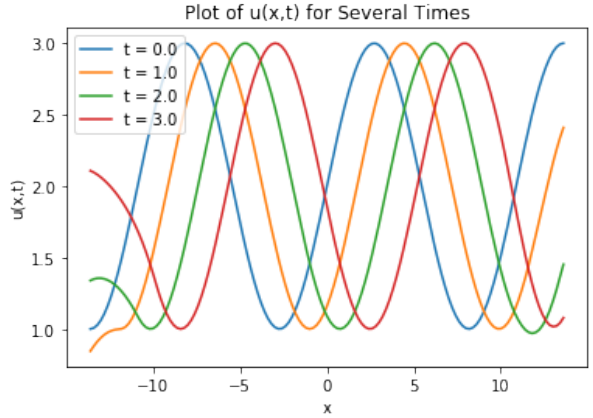
The plot below shows the discrepancy between numerical and exact solutions to the scalar wave equation at two different resolutions: dashed is low resolution ($\Delta x_{\rm low} = 0.4$) and solid is high resolution ($\Delta x_{\rm high} = 0.2$). Because this test adopts **eighth**-order finite differencing for spatial derivatives and **eighth**-order Runge-Kutta (RK8) for timestepping, we would expect this error to drop by a factor of approximately $(\Delta x_{\rm low}/\Delta x_{\rm high})^8 = (0.4/0.2)^8 = 2^8 = 256$ when going from low to high resolution, and after rescaling the error in the low-resolution case, we see that indeed it overlaps the low-resolution result quite nicely, confirming eighth-order convergence.

[21]:
```
Image("WaveToyNRPy/example_parfiles/8thOrder_ConvergenceTests/convergence-RK8-FD8-3D.png", width=500, height=500)
```
[21]:



The NewRad boundary condition driver allows the wave within our numerical domain to propagate out through the boundary, with minimal reflections. Shown below is the full extent of the x-axis of our grid, $[-13.6, 13.6]$, and the wave amplitude at different times. We observe indeed that there are minimal reflections as the wave propagates to the right. Note the features on the left side of the wave that originate from the left boundary. These features are the resullt of the incorrect assumption that our wave propagates radially, as opposed to just along a single diagonal.

[22]:
```
Image("WaveToyNRPy/example_parfiles/8thOrder_ConvergenceTests/wavetimes.png", width=500, height=500)
```
[22]:



5.2 Step 5.b: Spherical Gaussian wave convergence tests [Back to top]

Three-dimensional scalar wave equation code tests, adopting fourth-order finite differencing, coupled to RK4 method-of-lines for time integration

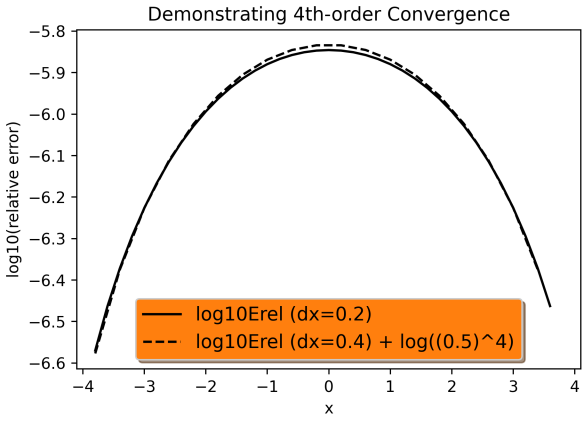
Inside the directory *WaveToyNRPy/example_parfiles/4thOrder_ConvergenceTests/* are the files used for this convergence test: **spherical_gaussian*.par** : ETK parameter files needed for performing the 3D tests. These parameter files set up a spherical gaussian wave propagating in a 3D numerical grid that extends from -13.0 to $+13.0$ along the x-, y-, and z-axes (in units of $c = 1$). The parameter files are identical, except one has grid resolution that is twice as high (so the errors should drop in the higher resolution case by a factor of 2^4, since we adopt eighth-order-convergent timestepping and spatial finite differencing).

Fourth-order code validation test results:

The plot below shows the discrepancy between numerical and exact solutions to the scalar wave equation at two different resolutions: the dashed line is low resolution ($\Delta x_{\rm low} = 0.4$) and the solid line is high resolution ($\Delta x_{\rm high} = 0.2$). Because this test adopts **fourth**-order finite differencing for spatial derivatives and **fourth**-order Runge-Kutta (RK4) for timestepping, we would expect this error to drop by a factor of approximately $(\Delta x_{\rm low}/\Delta x_{\rm high})^4 = (0.4/0.2)^4 = 2^4 = 16$ when going from low to high resolution, and after rescaling the error in the low-resolution case, we see that indeed it overlaps the low-resolution result quite nicely, confirming fourth-order convergence.

[23]:
```
Image("WaveToyNRPy/example_parfiles/4thOrder_ConvergenceTests/convergence-RK4-FD4-spherical_gaussian.png",␣
↪width=500, height=500)
```

[23]:



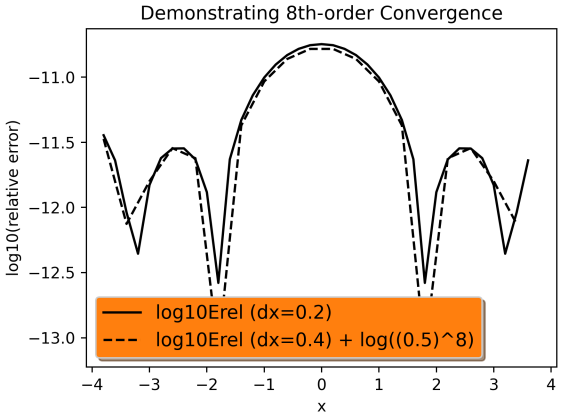
Three-dimensional scalar wave equation code tests for spherical gaussian initial data, adopting *eighth*-order finite differencing, coupled to *RK8* method-of-lines for time integration Inside the directory *WaveToyNRPy/example_parfiles/8thOrder_ConvergenceTests/* are the files used for this convergence test: **spherical_gaussian*.par** : ETK parameter files needed for performing the 3D tests. These parameter files set up a spherical gaussian wave propagating in a 3D numerical grid that extends from -13.8 to $+13.8$ along the x-, y-, and z-axes (in units of $c = 1$). The parameter files are identical, except one has grid resolution that is twice as high (so the errors should drop in the higher-resolution case by a factor of 2^8, because we adopt eighth-order-convergent timestepping and spatial finite differencing).

Eighth-order code validation test results:

The plot below shows the discrepancy between numerical and exact solutions to the scalar wave equation at two different resolutions: dashed is low resolution ($\Delta x_{\rm low} = 0.4$) and solid is high resolution ($\Delta x_{\rm high} = 0.2$). Because this test adopts **eighth**-order finite differencing for spatial derivatives and **eighth**-order Runge-Kutta (RK8) for timestepping, we would expect this error to drop by a factor of approximately $(\Delta x_{\rm low}/\Delta x_{\rm high})^8 = (0.4/0.2)^8 = 2^8 = 256$ when going from low to high resolution, and after rescaling the error in the low-resolution case, we see that indeed it overlaps the low-resolution result quite nicely, confirming eighth-order convergence.

[24]:
```
Image("WaveToyNRPy/example_parfiles/8thOrder_ConvergenceTests/convergence-RK8-FD8-spherical_gaussian.png",␣
↪width=500, height=500)
```

[24]:



6 Step 6: Output this notebook to LaTeX-formatted PDF file [Back to top]

The following code cell converts this Jupyter notebook into a proper, clickable LaTeX-formatted PDF file. After the cell is successfully run, the generated PDF may be found in the root NRPy+ tutorial directory, with filename `Tutorial-ETK_thorn-WaveToyNRPy.pdf` (Note that clicking on this link may not work; you may need to open the PDF file through another means.)

[25]:
```
import cmdline_helper as cmd    # NRPy+: Multi-platform Python command-line interface
cmd.output_Jupyter_notebook_to_LaTeXed_PDF("Tutorial-ETK_thorn-WaveToyNRPy")
```

```
Created Tutorial-ETK_thorn-WaveToyNRPy.tex, and compiled LaTeX file to PDF
    file Tutorial-ETK_thorn-WaveToyNRPy.pdf
```

Appendix B: `WeylScal4NRPy`

This appendix contains the code that generates the symbolic expressions for the Weyl scalars and invariants in Cartesian coordinates, as well as its documentation, as a PDF. Note that the notebook has been output in landscape mode and that two pages have been included per page of this dissertation.

Tutorial-WeylScalarsInvariants-Cartesian

July 30, 2021

-1 Using NRPy+ to Construct SymPy Expressions for Weyl Scalars and Invariants in Cartesian Coordinates

-1.1 Author: Patrick Nelson & Zach Etienne

-1.1.1 Formatting improvements courtesy Brandon Clark

-1.2 This module adopts the prescription of Baker, Campanelli, and Lousto. PRD 65, 044001 (2002) (hereafter, the "BCL paper") to construct the Weyl scalars ψ_0, ψ_1, ψ_2, ψ_3, and ψ_4 from an approximate Kinnersley tetrad.

-1.2.1 It also constructs the corresponding Weyl invariants adopting the same approach as Einstein Toolkit's Kranc-generated WeylScal4 diagnostic module. We will also follow that thorn's approach for other parts of this code.

Notebook Status: Validated

Validation Notes: The numerical implementation of expressions constructed in this module have been validated against a trusted code (the WeylScal4 Einstein Toolkit thorn).

-1.2.2 NRPy+ Source Code for this module:

- WeylScal4NRPy/WeylScalarInvariants_Cartesian.py
- WeylScal4NRPy/WeylScalars_Cartesian.py

-1.3 Introduction:

As this module is meant for Cartesian coordinates, all quantities are already rescaled. Furthermore, we assume that physical (as opposed to conformal) quantities, including the three-metric γ_{ij} and three-extrinsic curvature K_{ij} are provided as input gridfunctions.

0 Table of Contents

This notebook is organized as follows

1 Step 1: Set core NRPy+ parameters for numerical grids and reference metric [Back to top]

[1]:
```
# Step 1: import all needed modules from NRPy+:
import sympy as sp                # SymPy: The Python computer algebra package upon which NRPy+ depends
import indexedexp as ixp          # NRPy+: Symbolic indexed expression (e.g., tensors, vectors, etc.) support
import grid as gri                # NRPy+: Functions having to do with numerical grids
import NRPy_param_funcs as par    # NRPy+: Parameter interface
import sys                        # Standard Python modules for multiplatform OS-level functions
```

[2]:
```
# Step 2: Initialize WeylScalar parameters
thismodule = __name__
# Currently only one option: Approx_QuasiKinnersley = choice made in Baker, Campanelli, and Lousto. PRD 65, 044001␣
 ↪(2002)
par.initialize_param(par.glb_param("char", thismodule, "TetradChoice", "Approx_QuasiKinnersley"))
```

```
# The default value will output all psis
par.initialize_param(par.glb_param("bool", thismodule, "output_all_psis", False))
```

2 Step 2: Declare input and output variable gridfunctions [Back to top]

We declare our inputs as gridfunctions:

- the physical metric γ_{ij},
- the extrinsic curvature K_{ij},
- and the Cartesian coordinates (x, y, z).

Also, the output gridfunctions are as follows:

- the real and imaginary components of ψ_4, and
- the Weyl curvature invariants.

[3]:
```
# Step 3.a: Set spatial dimension (must be 3 for BSSN)
par.set_parval_from_str("grid::DIM",3)

# Step 3.b: declare the additional gridfunctions (i.e., functions whose values are declared
#           at every grid point, either inside or outside of our SymPy expressions) needed
#         for this thorn:
#           * the physical metric $\gamma_{ij}$,
#           * the extrinsic curvature $K_{ij}$,
#           * the real and imaginary components of $\psi_4$, and
#           * the Weyl curvature invariants:
gammaDD = ixp.register_gridfunctions_for_single_rank2("AUX","gammaDD", "sym01") # The AUX or EVOL designation is␣
 ↪*not*
                                                                                 # used in diagnostic modules.
kDD = ixp.register_gridfunctions_for_single_rank2("AUX","kDD", "sym01")
x,y,z = gri.register_gridfunctions("AUX",["x","y","z"])
psi4r,psi4i,psi3r,psi3i,psi2r,psi2i,psi1r,psi1i,psi0r,psi0i = gri.register_gridfunctions("AUX",["psi4r","psi4i",
                                                                                              "psi3r","psi3i",
                                                                                              "psi2r","psi2i",
                                                                                              "psi1r","psi1i",
                                                                                              "psi0r","psi0i"])
```



```
curvIr,curvIi,curvJr,curvJi,J1curv,J2curv,J3curv,J4curv = gri.register_gridfunctions("AUX",["curvIr","curvIi",
                                                                                            "curvJr","curvJi",
                                                                                            "J1curv","J2curv",
                                                                                            "J3curv","J4curv"])
```

3 Step 3: Defining the approximate quasi-Kinnersley tetrad [Back to top]

To define the Weyl scalars, first a tetrad must be chosen. Below, for compatibility with the WeylScal4 diagnostic module, we implement the approximate quasi-Kinnersley tetrad of the BCL paper.

We begin with the vectors given in Eqs. 5.6 and 5.7 of the BCL paper,

$$v_1^a = [-y, x, 0] \tag{1}$$
$$v_2^a = [x, y, z] \tag{2}$$
$$v_3^a = \det(g)^{1/2} g^{ad} \epsilon_{dbc} v_1^b v_2^c, \tag{3}$$

and carry out the Gram-Schmidt orthonormalization process. Note that these vectors are initially orthogonal to each other; one is in the ϕ direction, one is in r, and the third is the cross-product of the first two. The vectors w_i^a are placeholders in the code; the final product of the orthonormalization is the vectors e_i^a. So,

$$e_1^a = \frac{v_1^a}{\sqrt{\omega_{11}}} \tag{4}$$
$$e_2^a = \frac{v_2^a - \omega_{12} e_1^a}{\sqrt{\omega_{22}}} \tag{5}$$
$$e_3^a = \frac{v_3^a - \omega_{13} e_1^a - \omega_{23} e_2^a}{\sqrt{\omega_{33}}}, \tag{6}$$
$$\tag{7}$$

where $\omega_{ij} = v_i^a v_j^b \gamma_{ab}$ needs to be updated between steps (to save resources, we can get away with only calculating components as needed), and uses e_i^a instead of v_i^a if it has been calculated. Recall that γ_{ab} was declared as a gridfunction above.

Once we have orthogonal, normalized vectors, we can contruct the tetrad itself, again drawing on Eqs. 5.6 of the BCL paper. We could draw on SymPy's built-in tools for complex numbers to build the complex vectors m^a and $(m^*)^a$; however, the final expressions for the

Weyl scalars are complex enough that `sp.re()` and `sp.im()` are prohibitively time consuming. To get around this, we will define the real and imaginary components of $\overset{*}{m}^a$, and do the complex algebra by hand. Thus,

$$l^a = \frac{1}{\sqrt{2}} e_2^a \tag{8}$$

$$n^a = -\frac{1}{\sqrt{2}} e_2^a \tag{9}$$

$$m^a = \frac{1}{\sqrt{2}} (e_3^a + i e_1^a) \tag{10}$$

$$\overset{*}{m}^a = \frac{1}{\sqrt{2}} (e_3^a - i e_1^a) \tag{11}$$

In coding this procedure, we will follow the code from WeylScal4 very closely. We will also assume that $l^0 = n^0 = \frac{1}{\sqrt{2}}$ and that $m^0 = \overset{*}{m}^0 = 0$ (again, following the example of the Kranc-generated WeylScal4). This last assumption in particular will significantly reduce the terms needed to find ψ_4.

[4]:
```
# Step 4: Set which tetrad is used; at the moment, only one supported option

# The tetrad depends in general on the inverse 3-metric gammaUU[i][j]=\gamma^{ij}
#          and the determinant of the 3-metric (detgamma), which are defined in
#          the following line of code from gammaDD[i][j]=\gamma_{ij}.
tmpgammaUU, detgamma = ixp.symm_matrix_inverter3x3(gammaDD)
detgamma = sp.simplify(detgamma)
gammaUU = ixp.zerorank2()
for i in range(3):
    for j in range(3):
        gammaUU[i][j] = sp.simplify(tmpgammaUU[i][j])

if par.parval_from_str("TetradChoice") == "Approx_QuasiKinnersley":
    # Eqs 5.6 in https://arxiv.org/pdf/gr-qc/0104063.pdf
    xmoved = x# - xorig
    ymoved = y# - yorig
    zmoved = z# - zorig

    # Step 5.a: Choose 3 orthogonal vectors. Here, we choose one in the azimuthal
    #          direction, one in the radial direction, and the cross product of the two.
```



```
    # Eqs 5.7
    v1U = ixp.zerorank1()
    v2U = ixp.zerorank1()
    v3U = ixp.zerorank1()
    v1U[0] = -ymoved
    v1U[1] = xmoved# + offset
    v1U[2] = sp.sympify(0)
    v2U[0] = xmoved# + offset
    v2U[1] = ymoved
    v2U[2] = zmoved
    LeviCivitaSymbol_rank3 = ixp.LeviCivitaSymbol_dim3_rank3()
    for a in range(3):
        for b in range(3):
            for c in range(3):
                for d in range(3):
                    v3U[a] += sp.sqrt(detgamma) * gammaUU[a][d] * LeviCivitaSymbol_rank3[d][b][c] * v1U[b] *v2U[c]

    for a in range(3):
        v3U[a] = sp.simplify(v3U[a])

    # Step 5.b: Gram-Schmidt orthonormalization of the vectors.
    # The w_i^a vectors here are used to temporarily hold values on the way to the final vectors e_i^a

    # e_1^a &= \frac{v_1^a}{\omega_{11}}
    # e_2^a &= \frac{v_2^a - \omega_{12} e_1^a}{\omega_{22}}
    # e_3^a &= \frac{v_3^a - \omega_{13} e_1^a - \omega_{23} e_2^a}{\omega_{33}},

    # Normalize the first vector
    w1U = ixp.zerorank1()
    for a in range(3):
        w1U[a] = v1U[a]
    omega11 = sp.sympify(0)
    for a in range(3):
        for b in range(3):
            omega11 += w1U[a] * w1U[b] * gammaDD[a][b]
    e1U = ixp.zerorank1()
```

```
    for a in range(3):
        e1U[a] = w1U[a] / sp.sqrt(omega11)

    # Subtract off the portion of the first vector along the second, then normalize
    omega12 = sp.sympify(0)
    for a in range(3):
        for b in range(3):
            omega12 += e1U[a] * v2U[b] * gammaDD[a][b]
    w2U = ixp.zerorank1()
    for a in range(3):
        w2U[a] = v2U[a] - omega12*e1U[a]
    omega22 = sp.sympify(0)
    for a in range(3):
        for b in range(3):
            omega22 += w2U[a] * w2U[b] *gammaDD[a][b]
    e2U = ixp.zerorank1()
    for a in range(3):
        e2U[a] = w2U[a] / sp.sqrt(omega22)

    # Subtract off the portion of the first and second vectors along the third, then normalize
    omega13 = sp.sympify(0)
    for a in range(3):
        for b in range(3):
            omega13 += e1U[a] * v3U[b] * gammaDD[a][b]
    omega23 = sp.sympify(0)
    for a in range(3):
        for b in range(3):
            omega23 += e2U[a] * v3U[b] * gammaDD[a][b]
    w3U = ixp.zerorank1()
    for a in range(3):
        w3U[a] = v3U[a] - omega13*e1U[a] - omega23*e2U[a]
    omega33 = sp.sympify(0)
    for a in range(3):
        for b in range(3):
            omega33 += w3U[a] * w3U[b] * gammaDD[a][b]
    e3U = ixp.zerorank1()
```



```
    for a in range(3):
        e3U[a] = w3U[a] / sp.sqrt(omega33)

    # Step 5.c: Construct the tetrad itself.
    # Eqs. 5.6:
    # l^a &= \frac{1}{\sqrt{2}} e_2^a
    # n^a &= -\frac{1}{\sqrt{2}} e_2^a
    # m^a &= \frac{1}{\sqrt{2}} (e_3^a + i e_1^a)
    # \overset{*}{m}{}^a &= \frac{1}{\sqrt{2}} (e_3^a - i e_1^a)
    isqrt2 = 1/sp.sqrt(2)
    ltetU = ixp.zerorank1()
    ntetU = ixp.zerorank1()
    #mtetU = ixp.zerorank1()
    #mtetccU = ixp.zerorank1()
    remtetU = ixp.zerorank1() # SymPy did not like trying to take the real/imaginary parts of such a
    immtetU = ixp.zerorank1() # complicated expression, so we do it ourselves.
    for i in range(3):
        ltetU[i] = isqrt2 * e2U[i]
        ntetU[i] = -isqrt2 * e2U[i]
        remtetU[i] = isqrt2 * e3U[i]
        immtetU[i] = isqrt2 * e1U[i]
    nn = isqrt2

else:
    print("Error: TetradChoice == "+par.parval_from_str("TetradChoice")+" unsupported!")
    sys.exit(1)
```

4 Step 4: Building the Riemann and extrinsic curvature tensors [Back to top]

Now that we have the tetrad in place, we can contract it with the Weyl tensor to obtain the Weyl scalars. Naturally, we must first construct the Weyl tensor to do so; we will not do this directly, instead following the example of the BCL paper and the original WeylScal4. We will first build the Christoffel symbols,

$$\Gamma^i_{kl} = \frac{1}{2}\gamma^{im}(\gamma_{mk,l} + \gamma_{ml,k} - \gamma_{kl,m}).$$

[5]:
```
#Step 5: Declare and construct the second derivative of the metric.
gammaDD_dD = ixp.declarerank3("gammaDD_dD","sym01")

# Define the Christoffel symbols
GammaUDD = ixp.zerorank3(3)
for i in range(3):
    for k in range(3):
        for l in range(3):
            for m in range(3):
                GammaUDD[i][k][l] += (sp.Rational(1,2))*gammaUU[i][m]*\
                                     (gammaDD_dD[m][k][l] + gammaDD_dD[m][l][k] - gammaDD_dD[k][l][m])
```

We will also need the Riemann curvature tensor,

$$R_{abcd} = \frac{1}{2}(\gamma_{ad,cb} + \gamma_{bc,da} - \gamma_{ac,bd} - \gamma_{bd,ac}) + \gamma_{je}\Gamma^{j}_{bc}\Gamma^{e}_{ad} - \gamma_{je}\Gamma^{j}_{bd}\Gamma^{e}_{ac},$$

because several terms in our expression for ψ_4 are contractions of this tensor. To do this, we need second derivatives of the metric tensor, $\gamma_{ab,cd}$, using the finite differencing functionality in NRPy+.

[6]:
```
# Step 6.a: Declare and construct the Riemann curvature tensor:
gammaDD_dDD = ixp.declarerank4("gammaDD_dDD","sym01_sym23")
RiemannDDDD = ixp.zerorank4()
for a in range(3):
    for b in range(3):
        for c in range(3):
            for d in range(3):
                RiemannDDDD[a][b][c][d] += (gammaDD_dDD[a][d][c][b] +
                                            gammaDD_dDD[b][c][d][a] -
                                            gammaDD_dDD[a][c][b][d] -
                                            gammaDD_dDD[b][d][a][c]) * sp.Rational(1,2)
for a in range(3):
    for b in range(3):
        for c in range(3):
            for d in range(3):
                for e in range(3):
                    for j in range(3):
                        RiemannDDDD[a][b][c][d] +=  gammaDD[j][e] * GammaUDD[j][b][c] * GammaUDD[e][a][d] - \
                                                    gammaDD[j][e] * GammaUDD[j][b][d] * GammaUDD[e][a][c]
```


We will need the trace of the extrinsic curvature tensor K_{ij}, which can be computed as usual: $K = \gamma^{ij}K_{ij}$.

[7]:
```
# Step 6.b: We also need the extrinsic curvature tensor $K_{ij}$. This can be built from quantities from BSSN_RHSs
#           For now, we assume this is a gridfunction (We assume the ADM formalism for now).
#extrinsicKDD = ixp.zerorank2()
#for i in range(3):
#    for j in range(3):
#        extrinsicKDD[i][j] = (bssn.AbarDD[i][j] + sp.Rational(1,3)*gammaDD[i][j]*bssn.trK)/bssn.exp_m4phi
# We will, however, need to calculate the trace of K seperately:
trK = sp.sympify(0)
for i in range(3):
    for j in range(3):
        trK += gammaUU[i][j] * kDD[i][j]
```

5 Step 5: Putting it all together and calculating ψ_4 [Back to top]

We do not not need to explicitly build the Weyl tensor itself, because the BCL paper shows that, for the Weyl tensor C_{ijkl},

$$\begin{aligned}
\psi_4 &= C_{ijkl}n^i\bar{m}^j n^k\bar{m}^l && (12)\\
&= (R_{ijkl} + 2K_{i[k}K_{l]j})n^i\bar{m}^j n^k\bar{m}^l && (13)\\
&\quad - 8(K_{j[k,l]} + \Gamma^p_{j[k}K_{l]p})n^{[0}\bar{m}^{j]}n^k\bar{m}^l && (14)\\
&\quad + 4(R_{jl} - K_{jp}K^p_l + KK_{jl})n^{[0}\bar{m}^{j]}n^{[0}\bar{m}^{l]}. && (15)
\end{aligned}$$

Note here the brackets around pairs of indices. This indicates the antisymmetric part of a tensor; that is, for some arbitrary tensor A_{ij}, $A_{[ij]} = \frac{1}{2}(A_{ij} - A_{ji})$. This applies identically for indices belonging to separate tensors as well as superscripts in place of subscripts.

The other Weyl scalars from appendix A of the BCL paper that we may want to consider are

$$\begin{aligned}
\psi_3 &= (R_{ijkl} + 2K_{i[k}K_{l]j})l^i n^j \bar{m}^k n^l && (16)\\
&\quad - 4(K_{j[k,l]} + \Gamma^p_{j[k}K_{l]p})(l^{[0}n^{j]}\bar{m}^k n^l + l^k n^j \bar{m}^{[0}n^{l]}) && (17)\\
&\quad + 4(R_{jl} - K_{jp}K^p_l + KK_{jl})l^{[0}n^{j]}\bar{m}^{[0}n^{l]} && (18)\\
\psi_2 &= (R_{ijkl} + 2K_{i[k}K_{l]j})l^i m^j \bar{m}^k n^l && (19)\\
&\quad - 4(K_{j[k,l]} + \Gamma^p_{j[k}K_{l]p})(l^{[0}m^{j]}\bar{m}^k n^l + l^k m^l \bar{m}^{[0}n^{j]}) && (20)\\
&\quad + 4(R_{jl} - K_{jp}K^p_l + KK_{jl})l^{[0}m^{j]}\bar{m}^{[0}n^{l]} && (21)\\
\psi_1 &= (R_{ijkl} + 2K_{i[k}K_{l]j})n^i l^j m^k l^l && (22)\\
&\quad - 4(K_{j[k,l]} + \Gamma^p_{j[k}K_{l]p})(n^{[0}l^{j]}m^k l^l + n^k l^l m^{[0}l^{j]}) && (23)\\
&\quad + 4(R_{jl} - K_{jp}K^p_l + KK_{jl})n^{[0}l^{j]}m^{[0}l^{l]} && (24)\\
\psi_0 &= (R_{ijkl} + 2K_{i[k}K_{l]j})l^i m^j l^k m^l && (25)\\
&\quad - 8(K_{j[k,l]} + \Gamma^p_{j[k}K_{l]p})l^{[0}m^{j]}l^k m^l && (26)\\
&\quad + 4(R_{jl} - K_{jp}K^p_l + KK_{jl})l^{[0}m^{j]}l^{[0}m^{l]}. && (27)\\
& && (28)
\end{aligned}$$

To make it easier to track the construction of this expression, we will break it down into three parts, by first defining each of the parenthetical terms above separately. This is effectively identical to the procedure used in the Mathematica notebook that generates the original WeylScal4. That is, let

$$\text{GaussDDDD[i][j][k][l]} = R_{ijkl} + 2K_{i[k}K_{l]j}, \tag{29}$$

[8]:
```
# Step 7: Build the formula for \psi_4.
# Gauss equation: involving the Riemann tensor and extrinsic curvature.
GaussDDDD = ixp.zerorank4()
for i in range(3):
    for j in range(3):
        for k in range(3):
            for l in range(3):
                GaussDDDD[i][j][k][l] += RiemannDDDD[i][j][k][l] + kDD[i][k]*kDD[l][j] - kDD[i][l]*kDD[k][j]
```


$$\text{CodazziDDD[j][k][l]} = -2(K_{j[k,l]} + \Gamma^p_{j[k}K_{l]p}), \tag{30}$$

[9]:
```
3# Codazzi equation: involving partial derivatives of the extrinsic curvature.
# We will first need to declare derivatives of kDD
kDD_dD = ixp.declarerank3("kDD_dD","sym01")
CodazziDDD = ixp.zerorank3()
for j in range(3):
    for k in range(3):
        for l in range(3):
            CodazziDDD[j][k][l] += kDD_dD[j][l][k] - kDD_dD[j][k][l]

for j in range(3):
    for k in range(3):
        for l in range(3):
            for p in range(3):
                CodazziDDD[j][k][l] += GammaUDD[p][j][l]*kDD[k][p] - GammaUDD[p][j][k]*kDD[l][p]
```

and

$$\begin{aligned}
\text{RojoDD[j][l]} &= R_{jl} - K_{jp}K^p_l + KK_{jl} && (31)\\
&= \gamma^{pd}R_{jpld} - K_{jp}K^p_l + KK_{jl}, && (32)
\end{aligned}$$

[10]:
```
# Another piece. While not associated with any particular equation,
# this is still useful for organizational purposes.
RojoDD = ixp.zerorank2()
for j in range(3):
    for l in range(3):
        RojoDD[j][l] += trK*kDD[j][l]

for j in range(3):
    for l in range(3):
        for p in range(3):
            for d in range(3):
                RojoDD[j][l] += gammaUU[p][d]*RiemannDDDD[j][p][l][d] - kDD[j][p]*gammaUU[p][d]*kDD[d][l]
```

where these quantities are so named because of their relation to the Gauss-Codazzi equations. Then, we simply contract these with the tetrad we chose earlier to arrive at an expression for ψ_4. The barred Christoffel symbols and barred Ricci tensor have already been calculated by BSSN/BSSN_RHSs.py, so we use those values. So, our expression for ψ_4 has become

$$\psi_4 = (\text{GaussDDDD[i][j][k][l]}) n^i \bar{m}^j n^k \bar{m}^l \tag{33}$$
$$+ 2(\text{CodazziDDD[j][k][l]}) n^0 \bar{m}^j n^k \bar{m}^l \tag{34}$$
$$+ (\text{RojoDD[j][l]}) n^0 \bar{m}^j n^0 \bar{m}^l. \tag{35}$$

Likewise, we can rewrite the other Weyl scalars:

$$\psi_3 = (\text{GaussDDDD[i][j][k][l]}) l^i n^j \bar{m}^k n^l \tag{36}$$
$$+ (\text{CodazziDDD[j][k][l]}) (l^0 n^j \bar{m}^k n^l - l^j n^0 \bar{m}^k n^l - l^k n^j \bar{m}^l n^0) \tag{37}$$
$$- (\text{RojoDD[j][l]}) l^0 n^j \bar{m}^l n^0 - l^j n^0 \bar{m}^l n^0 \tag{38}$$
$$\psi_2 = (\text{GaussDDDD[i][j][k][l]}) l^i m^j \bar{m}^k n^l \tag{39}$$
$$+ (\text{CodazziDDD[j][k][l]}) (l^0 m^j \bar{m}^k n^l - l^j m^0 \bar{m}^k n^l - l^k m^l \bar{m}^l n^0) \tag{40}$$
$$- (\text{RojoDD[j][l]}) l^0 m^j \bar{m}^l n^0 \tag{41}$$
$$\psi_1 = (\text{GaussDDDD[i][j][k][l]}) n^i l^j m^k l^l \tag{42}$$
$$+ (\text{CodazziDDD[j][k][l]}) (n^0 l^j m^k l^l - n^j l^0 m^k l^l - n^k l^l m^j l^0) \tag{43}$$
$$- (\text{RojoDD[j][l]}) (n^0 l^j m^l l^0 - n^j l^0 m^l l^0) \tag{44}$$
$$\psi_0 = (\text{GaussDDDD[i][j][k][l]}) l^i m^j l^k m^l \tag{45}$$
$$+ 2(\text{CodazziDDD[j][k][l]}) (l^0 m^j l^k m^l + l^k m^l l^0 m^j) \tag{46}$$
$$+ (\text{RojoDD[j][l]}) l^0 m^j l^0 m^j. \tag{47}$$

We will start by setting the scalars to SymPy's 0 (this is done so that Python knows that the scalars are symbolic, not numeric, avoiding some potential bugs later on) and then performing the needed contractions of `RojoDD[j][l]`. Recall that the tetrad vectors were defined above and that we just built `RojoDD[j][l]` from the Ricci tensor and extrinsic curvature.

The relevant terms here are:

$$\psi_4: \ (\text{RojoDD[j][l]}) n^0 \bar{m}^j n^0 \bar{m}^l \tag{48}$$
$$\psi_3: \ -(\text{RojoDD[j][l]}) (l^0 n^j \bar{m}^l n^0 - l^j n^0 \bar{m}^l n^0) \tag{49}$$
$$\psi_2: \ -(\text{RojoDD[j][l]}) l^0 m^j \bar{m}^l n^0 \tag{50}$$
$$\psi_1: \ -(\text{RojoDD[j][l]}) (n^0 l^j m^l l^0 - n^j l^0 m^l l^0) \tag{51}$$
$$\psi_0: \ (\text{RojoDD[j][l]}) l^0 m^j l^0 m^j \tag{52}$$

[11]:
```
# Now we can calculate $\psi_4$ itself!
psi4r = sp.sympify(0)
psi4i = sp.sympify(0)
psi3r = sp.sympify(0)
psi3i = sp.sympify(0)
psi2r = sp.sympify(0)
psi2i = sp.sympify(0)
psi1r = sp.sympify(0)
psi1i = sp.sympify(0)
psi0r = sp.sympify(0)
psi0i = sp.sympify(0)
for l in range(3):
    for j in range(3):
        psi4r += RojoDD[j][l] * nn * nn * (remtetU[j]*remtetU[l]-immtetU[j]*immtetU[l])
        psi4i += RojoDD[j][l] * nn * nn * (-remtetU[j]*immtetU[l]-immtetU[j]*remtetU[l])
        psi3r +=-RojoDD[j][l] * nn * nn * (ntetU[j]-ltetU[j]) * remtetU[l]
        psi3i += RojoDD[j][l] * nn * nn * (ntetU[j]-ltetU[j]) * immtetU[l]
        psi2r +=-RojoDD[j][l] * nn * nn * (remtetU[l]*remtetU[j]+immtetU[j]*immtetU[l])
        psi2i +=-RojoDD[j][l] * nn * nn * (immtetU[l]*remtetU[j]-remtetU[j]*immtetU[l])
        psi1r += RojoDD[j][l] * nn * nn * (ntetU[j]*remtetU[l]-ltetU[j]*remtetU[l])
        psi1i += RojoDD[j][l] * nn * nn * (ntetU[j]*immtetU[l]-ltetU[j]*immtetU[l])
        psi0r += RojoDD[j][l] * nn * nn * (remtetU[j]*remtetU[l]-immtetU[j]*immtetU[l])
        psi0i += RojoDD[j][l] * nn * nn * (remtetU[j]*immtetU[l]+immtetU[j]*remtetU[l])
```

Now, we will add the contractions of `CodazziDDD[j][k][l]` to the Weyl Scalars. Again, we use the null tetrad we constructed and the tensor `CodazziDDD[j][k][l]` we constructed from the extrinsic curvature and Christoffel symbols.

The relevant terms here are:

$$\psi_4 : 2(\text{CodazziDDD[j][k][l]}) n^0 \bar{m}^j n^k \bar{m}^l \tag{53}$$
$$\psi_3 : \ (\text{CodazziDDD[j][k][l]}) (l^0 n^j \bar{m}^k n^l - l^j n^0 \bar{m}^k n^l - l^k n^j \bar{m}^l n^0) \tag{54}$$
$$\psi_2 : \ (\text{CodazziDDD[j][k][l]}) (l^0 m^j \bar{m}^k n^l - l^j m^0 \bar{m}^k n^l - l^k m^l \bar{m}^l n^0) \tag{55}$$
$$\psi_1 : \ (\text{CodazziDDD[j][k][l]}) (n^0 l^j m^k l^l - n^j l^0 m^k l^l - n^k l^l m^j l^0) \tag{56}$$
$$\psi_0 : 2(\text{CodazziDDD[j][k][l]}) (l^0 m^j l^k m^l + l^k m^l l^0 m^j) \tag{57}$$

[12]:
```
for l in range(3):
    for j in range(3):
        for k in range(3):
            psi4r += 2 * CodazziDDD[j][k][l] * ntetU[k] * nn * (remtetU[j]*remtetU[l]-immtetU[j]*immtetU[l])
            psi4i += 2 * CodazziDDD[j][k][l] * ntetU[k] * nn * (-remtetU[j]*immtetU[l]-immtetU[j]*remtetU[l])
            psi3r += 1 * CodazziDDD[j][k][l] * nn *␣
↪((ntetU[j]-ltetU[j])*remtetU[k]*ntetU[l]-remtetU[j]*ltetU[k]*ntetU[l])
            psi3i +=-1 * CodazziDDD[j][k][l] * nn *␣
↪((ntetU[j]-ltetU[j])*immtetU[k]*ntetU[l]-immtetU[j]*ltetU[k]*ntetU[l])
            psi2r += 1 * CodazziDDD[j][k][l] * nn *␣
↪(ntetU[l]*(remtetU[j]*remtetU[k]+immtetU[j]*immtetU[k])-ltetU[k]*(remtetU[j]*remtetU[l]+immtetU[j]*immtetU[l]))
            psi2i += 1 * CodazziDDD[j][k][l] * nn *␣
↪(ntetU[l]*(immtetU[j]*remtetU[k]-remtetU[j]*immtetU[k])-ltetU[k]*(remtetU[j]*immtetU[l]-immtetU[j]*remtetU[l]))
            psi1r += 1 * CodazziDDD[j][k][l] * nn *␣
↪(ltetU[j]*remtetU[k]*ltetU[l]-remtetU[j]*ntetU[k]*ltetU[l]-ntetU[j]*remtetU[k]*ltetU[l])
            psi1i += 1 * CodazziDDD[j][k][l] * nn *␣
↪(ltetU[j]*immtetU[k]*ltetU[l]-immtetU[j]*ntetU[k]*ltetU[l]-ntetU[j]*immtetU[k]*ltetU[l])
            psi0r += 2 * CodazziDDD[j][k][l] * nn * ltetU[k]*(remtetU[j]*remtetU[l]-immtetU[j]*immtetU[l])
            psi0i += 2 * CodazziDDD[j][k][l] * nn * ltetU[k]*(remtetU[j]*immtetU[l]+immtetU[j]*remtetU[l])
```

Finally, we will add the contractions of `GaussDDDD[i][j][k][l]` (from the Riemann tensor and extrinsic curvature, above) with the null tetrad.

The relevant terms here are:

$$\psi_4 : (\text{GaussDDDD[i][j][k][l]}) n^i \bar{m}^j n^k \bar{m}^l \tag{58}$$
$$\psi_3 : (\text{GaussDDDD[i][j][k][l]}) l^i n^j \bar{m}^k n^l \tag{59}$$
$$\psi_2 : (\text{GaussDDDD[i][j][k][l]}) l^i m^j \bar{m}^k n^l \tag{60}$$
$$\psi_1 : (\text{GaussDDDD[i][j][k][l]}) n^i l^j m^k l^l \tag{61}$$
$$\psi_0 : (\text{GaussDDDD[i][j][k][l]}) l^i m^j l^k m^l \tag{62}$$

[13]:
```
for l in range(3):
    for j in range(3):
        for k in range(3):
            for i in range(3):
                psi4r += GaussDDDD[i][j][k][l] * ntetU[i] * ntetU[k] *␣
↪(remtetU[j]*remtetU[l]-immtetU[j]*immtetU[l])
                psi4i += GaussDDDD[i][j][k][l] * ntetU[i] * ntetU[k] *␣
↪(-remtetU[j]*immtetU[l]-immtetU[j]*remtetU[l])
                psi3r += GaussDDDD[i][j][k][l] * ltetU[i] * ntetU[j] * remtetU[k] * ntetU[l]
                psi3i +=-GaussDDDD[i][j][k][l] * ltetU[i] * ntetU[j] * immtetU[k] * ntetU[l]
                psi2r += GaussDDDD[i][j][k][l] * ltetU[i] * ntetU[l] *␣
↪(remtetU[j]*remtetU[k]+immtetU[j]*immtetU[k])
                psi2i += GaussDDDD[i][j][k][l] * ltetU[i] * ntetU[l] *␣
↪(immtetU[j]*remtetU[k]-remtetU[j]*immtetU[k])
                psi1r += GaussDDDD[i][j][k][l] * ntetU[i] * ltetU[j] * remtetU[k] * ltetU[l]
                psi1i += GaussDDDD[i][j][k][l] * ntetU[i] * ltetU[j] * immtetU[k] * ltetU[l]
                psi0r += GaussDDDD[i][j][k][l] * ltetU[i] * ltetU[k] *␣
↪(remtetU[j]*remtetU[l]-immtetU[j]*immtetU[l])
                psi0i += GaussDDDD[i][j][k][l] * ltetU[i] * ltetU[k] *␣
↪(remtetU[j]*immtetU[l]+immtetU[j]*remtetU[l])
```

5.1 Step 5.a: Code validation against `WeylScal4NRPy.WeylScalars_Cartesian` NRPy+ Module [Back to top]

Here, as a code validation check, we verify agreement in the SymPy expressions for Weyl invariants between

1. this tutorial and
2. the NRPy+ WeylScal4NRPy.WeylScalars_Cartesian module.

[14]:
```
#psi4rb,psi4ib,psi3rb,psi3ib,psi2rb,psi2ib,psi1rb,psi1ib,psi0rb,psi0ib =␣
↪psi4r,psi4i,psi3r,psi3i,psi2r,psi2i,psi1r,psi1i,psi0r,psi0i
gri.glb_gridfcs_list = []
import WeylScal4NRPy.WeylScalars_Cartesian as weyl
par.set_parval_from_str("WeylScal4NRPy.WeylScalars_Cartesian::output_scalars","all_psis")
weyl.WeylScalars_Cartesian()

print("Consistency check between WeylScalars_Cartesian tutorial and NRPy+ module: ALL SHOULD BE ZERO.")

print("psi4r - weyl.psi4r = " + str(psi4r - weyl.psi4r))
print("psi4i - weyl.psi4i = " + str(psi4i - weyl.psi4i))
print("psi3r - weyl.psi3r = " + str(psi3r - weyl.psi3r))
print("psi3i - weyl.psi3i = " + str(psi3i - weyl.psi3i))
print("psi2r - weyl.psi2r = " + str(psi2r - weyl.psi2r))
print("psi2i - weyl.psi2i = " + str(psi2i - weyl.psi2i))
print("psi1r - weyl.psi1r = " + str(psi1r - weyl.psi1r))
print("psi1i - weyl.psi1i = " + str(psi1i - weyl.psi1i))
print("psi0r - weyl.psi0r = " + str(psi0r - weyl.psi0r))
print("psi0i - weyl.psi0i = " + str(psi0i - weyl.psi0i))
```

```
Consistency check between WeylScalars_Cartesian tutorial and NRPy+ module: ALL
SHOULD BE ZERO.
psi4r - weyl.psi4r = 0
psi4i - weyl.psi4i = 0
psi3r - weyl.psi3r = 0
psi3i - weyl.psi3i = 0
psi2r - weyl.psi2r = 0
psi2i - weyl.psi2i = 0
psi1r - weyl.psi1r = 0
psi1i - weyl.psi1i = 0
psi0r - weyl.psi0r = 0
psi0i - weyl.psi0i = 0
```


6 Step 6: The invariant scalars [Back to top]

We may also wish to compute the invariant scalars, whose value does not depend on the choice of the null tetrad. Although they are defined using the Weyl tensor, they can also be expressed in terms of the Weyl scalars. We will use those expressions for simplicity.

Following after the method used in the Kranc code, we will read in the already-computed values of the Weyl scalars to find the invariants instead of trying to make NRPy output a very large expression in terms of the metric and extrinsic curvature.

We will start with the invariants I and J, as defined in Eqs. (2.3a) and (2.3b) of arXiv:gr-qc/0407013. They are

$$I = 3\psi_2^2 - 4\psi_1\psi_3 + \psi_4\psi_0 \tag{63}$$

$$J = \begin{vmatrix} \psi_4 & \psi_3 & \psi_2 \\ \psi_3 & \psi_2 & \psi_1 \\ \psi_2 & \psi_1 & \psi_0 \end{vmatrix} \tag{64}$$

Here, because we can work in terms of the Weyl scalars themselves, we will use SymPy's built-in tools for handling complex numbers, which will not become overwhelmed as they did when computing the Weyl scalars.

[15]:
```
gri.glb_gridfcs_list = []
psi4r,psi4i,psi3r,psi3i,psi2r,psi2i,psi1r,psi1i,psi0r,psi0i = gri.register_gridfunctions("AUX",["psi4r","psi4i",
                                                                                             "psi3r","psi3i",
                                                                                             "psi2r","psi2i",
                                                                                             "psi1r","psi1i",
                                                                                             "psi0r","psi0i"])

psi4 = psi4r + sp.I * psi4i
psi3 = psi3r + sp.I * psi3i
psi2 = psi2r + sp.I * psi2i
psi1 = psi1r + sp.I * psi1i
psi0 = psi0r + sp.I * psi0i

curvIr = sp.re(3*psi2*psi2 - 4*psi1*psi3 + psi4*psi0)
curvIi = sp.im(3*psi2*psi2 - 4*psi1*psi3 + psi4*psi0)
curvJr = sp.re(psi4 * (psi2*psi0 - psi1*psi1) - \
               psi3 * (psi3*psi0 - psi1*psi2) +\
               psi2 * (psi3*psi1 - psi2*psi2) )
curvJi = sp.im(psi4 * (psi2*psi0 - psi1*psi1) - \
```

```
             psi3 * (psi3*psi0 - psi1*psi2) +\
             psi2 * (psi3*psi1 - psi2*psi2) )
```

We will now code the invariants J_1, J_2, J_3, and J_4, as found in Eqs. B5-B8 of arXiv:0704.1756. As with the other invariants, we will simply read in the values of the gridfunctions that we already calculated (that is, the Weyl scalars). These equations are based directly on those used in the Mathematica notebook that generates WeylScal4 (available at the ETK repository), modified so that Python can interpret them. Those equations were generated in turn using xTensor from Eqs. B5-B8.

[16]:
```
J1curv =-16*(3*psi2i**2-3*psi2r**2-4*psi1i*psi3i+4*psi1r*psi3r+psi0i*psi4i-psi0r*psi4r)

J2curv = 96*(-3*psi2i**2*psi2r+psi2r**3+2*psi1r*psi2i*psi3i+2*psi1i*psi2r*psi3i-psi0r*psi3i**2+
           2*psi1i*psi2i*psi3r-2*psi1r*psi2r*psi3r-2*psi0i*psi3i*psi3r+psi0r*psi3r**2-
           2*psi1i*psi1r*psi4i+psi0r*psi2i*psi4i+psi0i*psi2r*psi4i-psi1i**2*psi4r+psi1r**2*psi4r+
           psi0i*psi2i*psi4r-psi0r*psi2r*psi4r)

J3curv = 64*(9*psi2i**4-54*psi2i**2*psi2r**2+9*psi2r**4-24*psi1i*psi2i**2*psi3i+48*psi1r*psi2i*psi2r*psi3i+
           24*psi1i*psi2r**2*psi3i+16*psi1i**2*psi3i**2-16*psi1r**2*psi3i**2+
           24*psi1r*psi2i**2*psi3r+48*psi1i*psi2i*psi2r*psi3r-24*psi1r*psi2r**2*psi3r-64*psi1i*psi1r*psi3i*psi3r-
           16*psi1i**2*psi3r**2+16*psi1r**2*psi3r**2+6*psi0i*psi2i**2*psi4i-12*psi0r*psi2i*psi2r*psi4i-
           6*psi0i*psi2r**2*psi4i-8*psi0i*psi1i*psi3i*psi4i+8*psi0r*psi1r*psi3i*psi4i+8*psi0r*psi1i*psi3r*psi4i+
           8*psi0i*psi1r*psi3r*psi4i+psi0i**2*psi4i**2-psi0r**2*psi4i**2-6*psi0r*psi2i**2*psi4r-
           12*psi0i*psi2i*psi2r*psi4r+6*psi0r*psi2r**2*psi4r+8*psi0r*psi1i*psi3i*psi4r+8*psi0i*psi1r*psi3i*psi4r+
           8*psi0i*psi1i*psi3r*psi4r-8*psi0r*psi1r*psi3r*psi4r-4*psi0i*psi0r*psi4i*psi4r-psi0i**2*psi4r**2+
           psi0r**2*psi4r**2)

J4curv = -640*(-15*psi2i**4*psi2r+30*psi2i**2*psi2r**3-3*psi2r**5+10*psi1r*psi2i**3*psi3i+
             30*psi1i*psi2i**2*psi2r*psi3i-30*psi1r*psi2i*psi2r**2*psi3i-10*psi1i*psi2r**3*psi3i-
             16*psi1i*psi1r*psi2i*psi3i**2-3*psi0r*psi2i**2*psi3i**2-8*psi1i**2*psi2r*psi3i**2+
             8*psi1r**2*psi2r*psi3i**2-6*psi0i*psi2i*psi2r*psi3i**2+3*psi0r*psi2r**2*psi3i**2+
             4*psi0r*psi1i*psi3i**3+4*psi0i*psi1r*psi3i**3+10*psi1i*psi2i**3*psi3r-
             30*psi1r*psi2i**2*psi2r*psi3r-30*psi1i*psi2i*psi2r**2*psi3r+10*psi1r*psi2r**3*psi3r-
             16*psi1i**2*psi2i*psi3i*psi3r+16*psi1r**2*psi2i*psi3i*psi3r-6*psi0i*psi2i**2*psi3i*psi3r+
             32*psi1i*psi1r*psi2r*psi3i*psi3r+12*psi0r*psi2i*psi2r*psi3i*psi3r+6*psi0i*psi2r**2*psi3i*psi3r+
             12*psi0i*psi1i*psi3i**2*psi3r-12*psi0r*psi1r*psi3i**2*psi3r+16*psi1i*psi1r*psi2i*psi3r**2+
             3*psi0r*psi2i**2*psi3r**2+8*psi1i**2*psi2r*psi3r**2-8*psi1r**2*psi2r*psi3r**2+
             6*psi0i*psi2i*psi2r*psi3r**2-3*psi0r*psi2r**2*psi3r**2-12*psi0r*psi1i*psi3i*psi3r**2-
             12*psi0i*psi1r*psi3i*psi3r**2-4*psi0i*psi1i*psi3r**3+4*psi0r*psi1r*psi3r**3-
```



```
             6*psi1i*psi1r*psi2i**2*psi4i+2*psi0r*psi2i**3*psi4i-6*psi1i**2*psi2i*psi2r*psi4i+
             6*psi1r**2*psi2i*psi2r*psi4i+6*psi0i*psi2i**2*psi2r*psi4i+6*psi1i*psi1r*psi2r**2*psi4i-
             6*psi0r*psi2i*psi2r**2*psi4i-2*psi0i*psi2r**3*psi4i+12*psi1i**2*psi1r*psi3i*psi4i-
             4*psi1r**3*psi3i*psi4i-2*psi0r*psi1i*psi2i*psi3i*psi4i-2*psi0i*psi1r*psi2i*psi3i*psi4i-
             2*psi0i*psi1i*psi2r*psi3i*psi4i+2*psi0r*psi1r*psi2r*psi3i*psi4i-2*psi0i*psi0r*psi3i**2*psi4i+
             4*psi1i**3*psi3r*psi4i-12*psi1i*psi1r**2*psi3r*psi4i-2*psi0i*psi1i*psi2i*psi3r*psi4i+
             2*psi0r*psi1r*psi2i*psi3r*psi4i+2*psi0r*psi1i*psi2r*psi3r*psi4i+2*psi0i*psi1r*psi2r*psi3r*psi4i-
             2*psi0i**2*psi3i*psi3r*psi4i+2*psi0r**2*psi3i*psi3r*psi4i+2*psi0i*psi0r*psi3r**2*psi4i-
             psi0r*psi1i**2*psi4i**2-2*psi0i*psi1i*psi1r*psi4i**2+psi0r*psi1r**2*psi4i**2+
            ␣
↪2*psi0i*psi0r*psi2i*psi4i**2+psi0i**2*psi2r*psi4i**2-psi0r**2*psi2r*psi4i**2-3*psi1i**2*psi2i**2*psi4r+
             3*psi1r**2*psi2i**2*psi4r+2*psi0i*psi2i**3*psi4r+12*psi1i*psi1r*psi2i*psi2r*psi4r-
             6*psi0r*psi2i**2*psi2r*psi4r+3*psi1i**2*psi2r**2*psi4r-3*psi1r**2*psi2r**2*psi4r-
            ␣
↪6*psi0i*psi2i*psi2r**2*psi4r+2*psi0r*psi2r**3*psi4r+4*psi1i**3*psi3i*psi4r-12*psi1i*psi1r**2*psi3i*psi4r-
             2*psi0i*psi1i*psi2i*psi3i*psi4r+2*psi0r*psi1r*psi2i*psi3i*psi4r+2*psi0r*psi1i*psi2r*psi3i*psi4r+
             2*psi0i*psi1r*psi2r*psi3i*psi4r-psi0i**2*psi3i**2*psi4r+psi0r**2*psi3i**2*psi4r-
             12*psi1i**2*psi1r*psi3r*psi4r+4*psi1r**3*psi3r*psi4r+2*psi0r*psi1i*psi2i*psi3r*psi4r+
             2*psi0i*psi1r*psi2i*psi3r*psi4r+2*psi0i*psi1i*psi2r*psi3r*psi4r-2*psi0r*psi1r*psi2r*psi3r*psi4r+
             4*psi0i*psi0r*psi3i*psi3r*psi4r+psi0i**2*psi3r**2*psi4r-psi0r**2*psi3r**2*psi4r-
             2*psi0i*psi1i**2*psi4i*psi4r+4*psi0r*psi1i*psi1r*psi4i*psi4r+2*psi0i*psi1r**2*psi4i*psi4r+
             2*psi0i**2*psi2i*psi4i*psi4r-2*psi0r**2*psi2i*psi4i*psi4r-4*psi0i*psi0r*psi2r*psi4i*psi4r+
             psi0r*psi1i**2*psi4r**2+2*psi0i*psi1i*psi1r*psi4r**2-psi0r*psi1r**2*psi4r**2-
             2*psi0i*psi0r*psi2i*psi4r**2-psi0i**2*psi2r*psi4r**2+psi0r**2*psi2r*psi4r**2)

#cse_output = sp.cse(psi0i,sp.numbered_symbols("tmp"))
#for commonsubexpression in cse_output:
#    print("hello?",commonsubexpression)
#for commonsubexpression in cse_output[0]:
#    print((str(commonsubexpression[0])+" = "+str(commonsubexpression[1])+";").replace("**","^").replace("_d","d"))
#for i,result in enumerate(cse_output[1]):
#   print(("psi0iPy = "+str(result)+";").replace("**","^").replace("_d","d"))
# These replace commands are used to allow us to validate against Einstein Toolkit's WeylScal4 thorn in
 ↪Mathematica.
# Specifically, the first changes exponentiation to Mathematica's format, and the second strips the underscores
# that have a very specific meaning in Mathematica and thus cannot be used in variable names.
```

6.1 Step 6.a: Code validation against `WeylScal4NRPy.WeylScalarInvariants_Cartesian` NRPy+ module [Back to top]

Here, as a code validation check, we verify agreement in the SymPy expressions for Weyl invariants between

1. this tutorial and
2. the NRPy+ WeylScal4NRPy.WeylScalarInvariants_Cartesian module.

[17]:
```
# Reset the list of gridfunctions, as registering a gridfunction
#   twice will spawn an error.
gri.glb_gridfcs_list = []

import WeylScal4NRPy.WeylScalarInvariants_Cartesian as invar
invar.WeylScalarInvariants_Cartesian()

print("Consistency check between ScalarInvariants_Cartesian tutorial and NRPy+ module for invariant scalars: ALL␣
 ↪SHOULD BE ZERO.")

print("curvIr - weyl.curvIr = " + str(curvIr - invar.curvIr))
print("curvIi - weyl.curvIi = " + str(curvIi - invar.curvIi))
print("curvJr - weyl.curvJr = " + str(curvJr - invar.curvJr))
print("curvJi - weyl.curvJi = " + str(curvJi - invar.curvJi))
print("J1curv - weyl.J1curv = " + str(J1curv - invar.J1curv))
print("J2curv - weyl.J2curv = " + str(J2curv - invar.J2curv))
print("J3curv - weyl.J3curv = " + str(J3curv - invar.J3curv))
print("J4curv - weyl.J4curv = " + str(J4curv - invar.J4curv))
```

```
Consistency check between ScalarInvariants_Cartesian tutorial and NRPy+ module
for invariant scalars: ALL SHOULD BE ZERO.
curvIr - weyl.curvIr = 0
curvIi - weyl.curvIi = 0
curvJr - weyl.curvJr = 0
curvJi - weyl.curvJi = 0
J1curv - weyl.J1curv = 0
J2curv - weyl.J2curv = 0
J3curv - weyl.J3curv = 0
```



```
J4curv - weyl.J4curv = 0
```

7 Step 7: Output this notebook to LaTeX-formatted PDF file [Back to top]

The following code cell converts this Jupyter notebook into a proper, clickable LaTeX-formatted PDF file. After the cell is successfully run, the generated PDF may be found in the root NRPy+ tutorial directory, with filename Tutorial-WeylScalarsInvariants-Cartesian.pdf. (Note that clicking on this link may not work; you may need to open the PDF file through another means.)

[18]:
```
import cmdline_helper as cmd    # NRPy+: Multi-platform Python command-line interface
cmd.output_Jupyter_notebook_to_LaTeXed_PDF("Tutorial-WeylScalarsInvariants-Cartesian")
```

```
Created Tutorial-WeylScalarsInvariants-Cartesian.tex, and compiled LaTeX
    file to PDF file Tutorial-WeylScalarsInvariants-Cartesian.pdf
```

Appendix C: GiRaFFE_HO

This appendix is the notebook documenting the original attempt to port GiRaFFE to NRPy+, formatted as a PDF. Note that the notebook has been output in landscape mode and that two pages have been included per page of this dissertation.

Tutorial-GiRaFFE_Higher_Order

July 30, 2021

-1 GiRaFFE: General Relativistic Force-Free Electrodynamics

-1.1 Authors: Patrick Nelson, Zach Etienne, George Vopal, & Maria Babiuc-Hamilton

-1.1.1 Formatting improvements courtesy Brandon Clark

Notebook Status: In progress

Validation Notes: This module is under active development – do ***not*** use the resulting C code output for doing science.

-1.1.2 NRPy+ Source Code for this module: GiRaFFE_HO/GiRaFFE_Higher_Order.py

-1.2 Introduction:

The original GiRaFFE code, as presented in the original paper, exists as a significant modification to IllinoisGRMHD. As such, it used a third-order reconstruction algorithm with a slope limiter (Colella et al's piecewise parabolic method, or PPM) to handle spatial derivatives in the general relativistic force-free electrodynamics (GRFFE) equations. However, the GRFFE equations do not generally permit shocks, so a more optimal approach would involve finite-differencing all derivatives in the GRFFE equations. As it happens, NRPy+ was designed to generate C codes involving complex tensorial expressions and finite difference spatial derivatives, with finite-differencing order a freely specifiable parameter.

The purpose of this notebook is to rewrite the equations of GRFFE as used in the original GiRaFFE code so that all derivatives that appear are represented numerically as finite-difference derivatives. As we will see, the largest complication stems from derivatives of magnetic fields—requiring judicious application of the chain and product rules.

The GRFFE evolution equations (from Eq. 13 of the original GiRaFFE paper) we wish to encode in the NRPy+ version of GiRaFFE are as follows:

- $\partial_t \tilde{S}_i = -\partial_j \left(\alpha\sqrt{\gamma} T^j_{\rm EM i}\right) + \frac{1}{2}\alpha\sqrt{\gamma} T^{\mu\nu}_{\rm EM}\partial_i g_{\mu\nu}$: Link to Step 2
- $\partial_t A_i = \epsilon_{ijk} v^j B^k - \partial_i(\alpha\Phi - \beta^j A_j)$: Link to Step 5
- $\partial_t [\sqrt{\gamma}\Phi] = -\partial_j(\alpha\sqrt{\gamma}A^j - \beta^j[\sqrt{\gamma}\Phi]) - \xi\alpha[\sqrt{\gamma}\Phi]$: Link to Step 5

Here, the densitized spatial Poynting flux one-form $\tilde{S}_i = \sqrt{\gamma} S_i$ (and S_i comes from $S_\mu - n_\nu T^\nu_{{\rm EM}\mu}$), and (Φ, A_i) is the vector potential. We will solve these PDEs using the method of lines, where the right-hand sides of these equations (involving no explicit time derivatives) will be constructed using NRPy+.

-1.2.1 A Note on Notation:

As is standard in NRPy+,

- Greek indices refer to four-dimensional quantities in which the zeroth component indicates temporal (time) component.
- Latin indices refer to three-dimensional quantities. This is somewhat counterintuitive, because Python always indexes its lists starting from 0. As a result, the zeroth component of three-dimensional quantities will necessarily indicate the first *spatial* direction.

For instance, in calculating the first term of $T^{\mu\nu}_{\rm EM}$ (specifically, Term 1 $= b^2 u^\mu u^\nu$), we use Greek indices:

```
T4EMUU = ixp.zerorank2(DIM=4)
for mu in range(4):
    for nu in range(4):
        # Term 1: b^2 u^{\mu} u^{\nu}
        T4EMUU[mu][nu] = smallb2*u4U[mu]*u4U[nu]\
```

When we calculate $\beta_i = \gamma_{ij}\beta^j$, we use Latin indices:

```
betaD = ixp.zerorank1()
for i in range(DIM):
    for j in range(DIM):
        betaD[i] += gammaDD[i][j] * betaU[j]
```

As a corollary, any expressions involving mixed Greek and Latin indices will need to offset one set of indices by one: A Latin index in a four-vector will be incremented and a Greek index in a three-vector will be decremented (however, the latter case does not occur in this tutorial notebook). This can be seen when we handle the second term of $\partial_t \tilde{S}_i$ (or, more specifically, the second term thereof: $\frac{1}{2}\alpha\sqrt{\gamma} T^{\mu\nu}_{\rm EM}\partial_i g_{\mu\nu}$):

```
# The second term: \alpha \sqrt{\\gamma} T^{\mu \nu}_{\rm EM} \partial_i g_{\mu \nu} / 2
for i in range(DIM):
```

```
for mu in range(4):
    for nu in range(4):
        Stilde_rhsD[i] += alpsqrtgam * T4EMUU[mu][nu] * g4DDdD[mu][nu][i+1] / 2
```

0 Table of Contents:

This notebook is organized as follows:

1 Step 1: Set up the needed NRPy+ infrastructure and declare core gridfunctions used by `GiRaFFE` [Back to top]

1. Set some basic NRPy+ parameters, for example, set the spatial dimension parameter to 3 and the finite differencing order to 4.
2. Next, declare some gridfunctions that are provided as input to the equations:
    1. α, β^i, and γ_{ij}: These ADM 3+1 metric quantities are declared in the ADMBase Einstein Toolkit thorn, and are assumed to be made available to `GiRaFFE` at this stage.
    2. The Valencia three-velocity $v^i_{(n)}$ and vector potential A_i: Declared by `GiRaFFE`, and will have their initial values set in the separate thorn `GiRaFFEfood_HO`.
    3. The magnetic field as measured by a normal observer B^i: The quantities evolved forward in time in `GiRaFFE` do not include the Valencia three-velocity, so this quantity is not automatically updated. Instead, we compute it on the basis of the evolved quantity $\tilde{S}_i$ and $B^i = \epsilon^{ijk}\partial_j A_k$ (where A_k is another evolved quantity and ϵ^{ijk} is the Levi-Civita tensor). B^i is evaluated using finite differences of A_k in a separate function, though it can only be evaluated consistently on the interior of the grid. In the ghost zones, we will have to use lower-order derivatives.

[1]:
```
# Step 0: Add NRPy's directory to the path
# https://stackoverflow.com/questions/16780014/import-file-from-parent-directory
import os,sys
nrpy_dir_path = os.path.join("..")
if nrpy_dir_path not in sys.path:
    sys.path.append(nrpy_dir_path)
nrpy_dir_path = os.path.join("../..")
if nrpy_dir_path not in sys.path:
    sys.path.append(nrpy_dir_path)
nrpy_dir_path = os.path.join("../../..")
if nrpy_dir_path not in sys.path:
    sys.path.append(nrpy_dir_path)

import NRPy_param_funcs as par   # NRPy+: Parameter interface
import indexedexp as ixp         # NRPy+: Symbolic indexed expression (e.g., tensors, vectors, etc.) support
import grid as gri               # NRPy+: Functions having to do with numerical grids
import finite_difference as fin  # NRPy+: Finite difference C code generation module
# from outputC import *
import sympy as sp               # SymPy: The Python computer algebra package upon which NRPy+ depends
```

```
#Step 1.0: Set the spatial dimension parameter to 3.
par.set_parval_from_str("grid::DIM", 3)
DIM = par.parval_from_str("grid::DIM")

# Step 1.1: Set the finite differencing order to 4.
par.set_parval_from_str("finite_difference::FD_CENTDERIVS_ORDER", 4)

thismodule = "GiRaFFE_NRPy"

# M_PI will allow the C code to substitute the correct value
M_PI = par.Cparameters("#define",thismodule,"M_PI","")
# ADMBase defines the 4-metric in terms of the 3+1 spacetime metric quantities gamma_{ij}, beta^i, and alpha
gammaDD = ixp.register_gridfunctions_for_single_rank2("AUX","gammaDD", "sym01",DIM=3)
betaU   = ixp.register_gridfunctions_for_single_rank1("AUX","betaU",DIM=3)
alpha   = gri.register_gridfunctions("AUX","alpha")
# GiRaFFE uses the Valencia 3-velocity and A_i, which are defined in the initial data module(GiRaFFEfood)
ValenciavU = ixp.register_gridfunctions_for_single_rank1("AUX","ValenciavU",DIM=3)
AD = ixp.register_gridfunctions_for_single_rank1("EVOL","AD",DIM=3)
# B^i must be computed at each timestep within GiRaFFE so that the Valencia 3-velocity can be evaluated
BU = ixp.register_gridfunctions_for_single_rank1("AUX","BU",DIM=3)
```

2 Step 2: Build the four-metric $g_{\mu\nu}$, its inverse $g^{\mu\nu}$, and spatial derivatives $g_{\mu\nu,i}$ from ADM 3+1 quantities γ_{ij}, β^i, and α [Back to top]

Notice that the time evolution equation for $\tilde{S}_i$

$$\partial_t \tilde{S}_i = -\partial_j \left(\alpha \sqrt{\gamma} T^j_{\mathrm{EM}i} \right) + \frac{1}{2} \alpha \sqrt{\gamma} T^{\mu\nu}_{\mathrm{EM}} \partial_i g_{\mu\nu}$$

contains $\partial_i g_{\mu\nu} = g_{\mu\nu,i}$. We will now focus on evaluating this term.

The four-metric $g_{\mu\nu}$ is related to the three-metric γ_{ij}, index-lowered shift β_i, and lapse α by

$$g_{\mu\nu} = \begin{pmatrix} -\alpha^2 + \beta^k \beta_k & \beta_j \\ \beta_i & \gamma_{ij} \end{pmatrix}.$$

This tensor and its inverse have already been built by the u0_smallb_Poynting__Cartesian.py module (documented here), so we can simply load the module and import the variables.

[2]:
```
# Step 1.2: import u0_smallb_Poynting__Cartesian.py to set
#           the four metric and its inverse. This module also sets b^2 and u^0.
import u0_smallb_Poynting__Cartesian.u0_smallb_Poynting__Cartesian as u0b
u0b.compute_u0_smallb_Poynting__Cartesian(gammaDD,betaU,alpha,ValenciavU,BU)

betaD = ixp.zerorank1()
for i in range(DIM):
    for j in range(DIM):
        betaD[i] += gammaDD[i][j] * betaU[j]

# We will now pull in the four metric and its inverse.
import BSSN.ADMBSSN_tofrom_4metric as AB4m
gammaDD,betaU,alpha = AB4m.setup_ADM_quantities("ADM")
AB4m.g4DD_ito_BSSN_or_ADM("ADM",gammaDD,betaU,alpha)
g4DD = AB4m.g4DD
AB4m.g4UU_ito_BSSN_or_ADM("ADM",gammaDD,betaU,alpha)
g4UU = AB4m.g4UU
```

Next we compute spatial derivatives of the metric, $\partial_i g_{\mu\nu} = g_{\mu\nu,i}$, written in terms of the three-metric, shift, and lapse. Simply taking the derivative of the expression for $g_{\mu\nu}$ above, we find

$$g_{\mu\nu,l} = \begin{pmatrix} -2\alpha\alpha_{,l} + \beta^k_{\ ,l}\beta_k + \beta^k \beta_{k,l} & \beta_{j,l} \\ \beta_{i,l} & \gamma_{ij,l} \end{pmatrix}.$$

Notice the derivatives of the shift vector with its index lowered, $\beta_{i,j} = \partial_j \beta_i$. This can be easily computed in terms of the given ADMBase quantities β^i and γ_{ij} via:

$$\begin{aligned} \beta_{i,j} &= \partial_j \beta_i && (1)\\ &= \partial_j (\gamma_{ik} \beta^k) && (2)\\ &= \gamma_{ik} \partial_j \beta^k + \beta^k \partial_j \gamma_{ik} && (3)\\ \beta_{i,j} &= \gamma_{ik} \beta^k_{\ ,j} + \beta^k \gamma_{ik,j}. && (4) \end{aligned}$$

Because this expression mixes Greek and Latin indices, we will declare this as a four-dimensional quantity, but set only the three spatial components of its last index (that is, leaving $l = 0$ unset).

So, we will first set

$$g_{00,l} = \underbrace{-2\alpha\alpha_{,l}}_{\text{Term 1}} + \underbrace{\beta^k{}_{,l}\beta_k}_{\text{Term 2}} + \underbrace{\beta^k\beta_{k,l}}_{\text{Term 3}}.$$

[3]:
```
# Step 1.2, cont'd: Build spatial derivatives of the four metric
# Step 1.2.a: Declare derivatives of grid functions. These will be handled by FD_outputC
alpha_dD   = ixp.declarerank1("alpha_dD")
betaU_dD   = ixp.declarerank2("betaU_dD","nosym")
gammaDD_dD = ixp.declarerank3("gammaDD_dD","sym01")

# Step 1.2.b: These derivatives will be constructed analytically.
betaDdD    = ixp.zerorank2()
g4DDdD     = ixp.zerorank3(DIM=4)

for i in range(DIM):
    for j in range(DIM):
        for k in range(DIM):
            # \gamma_{ik} \beta^k_{,j} + \beta^k \gamma_{ik,j}
            betaDdD[i][j] += gammaDD[i][k] * betaU_dD[k][j] + betaU[k] * gammaDD_dD[i][k][j]

# Step 1.2.c: Set the 00 components
# Step 1.2.c.i: Term 1: -2\alpha \alpha_{,l}
for l in range(DIM):
    g4DDdD[0][0][l+1] = -2*alpha*alpha_dD[l]

# Step 1.2.c.ii: Term 2: \beta^k_{\ ,l} \beta_k
for l in range(DIM):
    for k in range(DIM):
        g4DDdD[0][0][l+1] += betaU_dD[k][l] * betaD[k]

# Step 1.2.c.iii: Term 3: \beta^k \beta_{k,l}
for l in range(DIM):
    for k in range(DIM):
        g4DDdD[0][0][l+1] += betaU[k] * betaDdD[k][l]
```

Now we will contruct the other components of $g_{\mu\nu,l}$. We will first construct

$$g_{i0,l} = g_{0i,l} = \beta_{i,l},$$

then

$$g_{ij,l} = \gamma_{ij,l}.$$

[4]:
```
# Step 1.2.d: Set the i0 and 0j components
for l in range(DIM):
    for i in range(DIM):
        # \beta_{i,l}
        g4DDdD[i+1][0][l+1] = g4DDdD[0][i+1][l+1] = betaDdD[i][l]

#Step 1.2.e: Set the ij components
for l in range(DIM):
    for i in range(DIM):
        for j in range(DIM):
            # \gamma_{ij,l}
            g4DDdD[i+1][j+1][l+1] = gammaDD_dD[i][j][l]
```

3 Step 3: $T^{\mu\nu}_{\rm EM}$ and its derivatives [Back to top]

Now that the metric and its derivatives are out of the way, we return to the evolution equation for $\tilde{S}_i$,

$$\partial_t \tilde{S}_i = -\partial_j \left(\alpha\sqrt{\gamma}T^j_{{\rm EM}i}\right) + \frac{1}{2}\alpha\sqrt{\gamma}T^{\mu\nu}_{\rm EM}\partial_i g_{\mu\nu}.$$

We turn our focus to $T^j_{{\rm EM}i}$ and its derivatives. To this end, we start by computing $T^{\mu\nu}_{\rm EM}$ (from Eq. 27 of Paschalidis & Shapiro's paper on their GRFFE code):

$$\boxed{T^{\mu\nu}_{\rm EM} = b^2 u^\mu u^\nu + \frac{b^2}{2} g^{\mu\nu} - b^\mu b^\nu.}$$

Notice that $T^{\mu\nu}_{\rm EM}$ is written in terms of

- b^μ, the four-component magnetic field vector, related to the comoving magnetic field vector $B^i_{(u)}$
- u^μ, the four-velocity
- $g^{\mu\nu}$, the inverse four-metric

However, GiRaFFE has access to only the following quantities, requiring in the following sections that we write the above quantities in terms of the following ones:

- γ_{ij}, the three-metric
- α, the lapse
- β^i, the shift
- A_i, the vector potential
- B^i, the magnetic field (we assume only in the grid interior, not the ghost zones)
- $\left[\sqrt{\gamma}\Phi\right]$, the zero-component of the vector potential A_μ, times the square root of the determinant of the three-metric
- $v^i_{(n)}$, the Valencia three-velocity
- u^0, the zero-component of the four-velocity

3.1 Step 3.a: u^i and b^i and related quantities [Back to top]

We begin by importing what we can from u0_smallb_Poynting__Cartesian.py. We will need the four-velocity u^μ, which is related to the Valencia 3-velocity $v^i_{(n)}$ used directly by GiRaFFE (see also Duez et al, eqs. 53 and 56)

$$u^i = u^0(\alpha v^i_{(n)} - \beta^i), \tag{5}$$
$$u_j = \alpha u^0 \gamma_{ij} v^i_{(n)}, \tag{6}$$

where $v^i_{(n)}$ is the Valencia three-velocity. These have already been constructed in terms of the Valencia 3-velocity and other 3+1 ADM quantities by the u0_smallb_Poynting__Cartesian.py module, so we can simply import these variables:

[5]:
```
# Step 2.0: u^i, b^i, and related quantities
# Step 2.0.a: import the four-velocity, as written in terms of the Valencia 3-velocity
uD = ixp.register_gridfunctions_for_single_rank1("AUX","uD")
uU = ixp.register_gridfunctions_for_single_rank1("AUX","uU")
u4upperZero = gri.register_gridfunctions("AUX","u4upperZero")

for i in range(DIM):
    uD[i] = u0b.uD[i].subs(u0b.u0,u4upperZero)
    uU[i] = u0b.uU[i].subs(u0b.u0,u4upperZero)
```


We also need the magnetic field four-vector b^μ, which is related to the magnetic field by Eqs. 23, 24, and 31 in Duez et al:

$$b^0 = \frac{1}{\sqrt{4\pi}} B^0_{(u)} = \frac{u_j B^j}{\sqrt{4\pi}\alpha}, \tag{7}$$
$$b^i = \frac{1}{\sqrt{4\pi}} B^i_{(u)} = \frac{B^i + (u_j B^j) u^i}{\sqrt{4\pi}\alpha u^0}, \tag{8}$$

where B^i is the variable tracked by the HydroBase thorn in the Einstein Toolkit. Again, these have already been built by the u0_smallb_Poynting__Cartesian.py module, so we can simply import the variables.

[6]:
```
# Step 2.0.b: import the small b terms
smallb4U = ixp.zerorank1(DIM=4)
smallb4D = ixp.zerorank1(DIM=4)
for mu in range(4):
    smallb4U[mu] = u0b.smallb4U[mu].subs(u0b.u0,u4upperZero)
    smallb4D[mu] = u0b.smallb4D[mu].subs(u0b.u0,u4upperZero)

smallb2 = u0b.smallb2etk.subs(u0b.u0,u4upperZero)
```

3.2 Step 3.b: Construct all components of the electromagnetic stress-energy tensor $T^{\mu\nu}_{\rm EM}$ [Back to top]

We now have all the pieces to calculate the stress-energy tensor,

$$T^{\mu\nu}_{\rm EM} = \underbrace{b^2 u^\mu u^\nu}_{\text{Term 1}} + \underbrace{\frac{b^2}{2} g^{\mu\nu}}_{\text{Term 2}} - \underbrace{b^\mu b^\nu}_{\text{Term 3}}.$$

Because u^0 is a separate variable, we will create a temporary variable $u^\mu = (u^0, u^i)$.

[7]:
```
# Step 2.1: Construct the electromagnetic stress-energy tensor
# Step 2.1.a: Set up the four-velocity vector
u4U = ixp.zerorank1(DIM=4)
u4U[0] = u4upperZero
for i in range(DIM):
    u4U[i+1] = uU[i]
```

```
# Step 2.1.b: Build T4EMUU itself
T4EMUU = ixp.zerorank2(DIM=4)
for mu in range(4):
    for nu in range(4):
        # Term 1: b^2 u^{\mu} u^{\nu}
        T4EMUU[mu][nu] = smallb2*u4U[mu]*u4U[nu]

for mu in range(4):
    for nu in range(4):
        # Term 2: b^2 / 2 g^{\mu \nu}
        T4EMUU[mu][nu] += smallb2*g4UU[mu][nu]/2

for mu in range(4):
    for nu in range(4):
        # Term 3: -b^{\mu} b^{\nu}
        T4EMUU[mu][nu] += -smallb4U[mu]*smallb4U[nu]
```

3.3 Step 3.c: Derivative components of the electromagnetic stress-energy tensor [Back to top]

If we look at the evolution equation, we see that we will need spatial derivatives of $T^{\mu\nu}_{\rm EM}$. When confronted with derivatives of complicated expressions, it is generally convenient to declare those expressions as gridfunctions themselves, allowing NRPy+ to take finite-difference derivatives of the expressions. This can even reduce the truncation error associated with the finite differences, because the alternative is to use a function of several finite-difference derivatives, allowing more error to accumulate than the extra gridfunction will introduce. Although we will use that technique for some of the subexpressions of $T^{\mu\nu}_{\rm EM}$, we do not want to rely on it for the whole expression; doing so would require us to take the derivative of the magnetic field B^i, which is itself found by finite-differencing the vector potential A_k. Thus B^i cannot be *consistently* defined in ghost zones. To potentially reduce numerical errors induced by inconsistent finite differencing, we will differentiate $T^{\mu\nu}_{\rm EM}$ term-by-term so that finite-difference derivatives of A_i appear.

We will now now take these spatial derivatives of $T^{\mu\nu}_{\rm EM}$, applying the chain rule until it is only in terms of basic gridfunctions and their derivatives: α, β^i, γ_{ij}, A_k, and the four-velocity u^i. Along the way, we will also set up useful temporary variables representing the steps of the chain rule. (Notably, *all* of these quantities will be written in terms of A_i and its derivatives:)

- B^i (already computed in terms of A_k, via $B^i = \epsilon^{ijk}\partial_j A_k$),
- $B^i_{,j}$,
- b^i and b_i (already computed),

- $b^i_{,k}$,
- b^2 (already computed),
- and $\left(b^2\right)_{,j}$.

(The variables not already computed will not be seen by the ETK, because they are written in terms of A_k and its derivatives; they simply help to organize the NRPy+ code.)

So then,

$$\partial_j T^j_{{\rm EM}i} = \partial_j (g_{\mu i} T^{\mu j}_{\rm EM}) \tag{9}$$
$$= \partial_j \left[g_{\mu i} \left(b^2 u^j u^\mu + \frac{b^2}{2} g^{j\mu} - b^j b^\mu \right) \right] \tag{10}$$
$$= \underbrace{g_{\mu i,j} T^{\mu j}_{\rm EM}}_{\text{Term A}} + g_{\mu i} \left(\underbrace{\partial_j \left(b^2 u^j u^\mu \right)}_{\text{Term B}} + \underbrace{\partial_j \left(\frac{b^2}{2} g^{j\mu} \right)}_{\text{Term C}} - \underbrace{\partial_j \left(b^j b^\mu \right)}_{\text{Term D}} \right) \tag{11}$$

Following the product and chain rules for each term, we find that

$$\text{Term B} = \partial_j (b^2 u^j u^\mu) \tag{12}$$
$$= \partial_j b^2 u^j u^\mu + b^2 \partial_j u^j u^\mu + b^2 u^j \partial_j u^\mu \tag{13}$$
$$= \underbrace{\left(b^2\right)_{,j} u^j u^\mu}_{\text{To Term 3 below}} + \underbrace{b^2 u^j_{,j} u^\mu + b^2 u^j u^\mu_{,j}}_{\text{To Term 2 below}} \tag{14}$$
$$\text{Term C} = \partial_j \left(\frac{b^2}{2} g^{j\mu} \right) \tag{15}$$
$$= \frac{1}{2} \left(\partial_j b^2 g^{j\mu} + b^2 \partial_j g^{j\mu} \right) \tag{16}$$
$$= \underbrace{\frac{1}{2} \left(b^2\right)_{,j} g^{j\mu}}_{\text{To Term 3 below}} + \underbrace{\frac{b^2}{2} g^{j\mu}_{,j}}_{\text{To Term 2 below}} \tag{17}$$
$$\text{Term D} = \partial_j (b^j b^\mu) \tag{18}$$
$$= \underbrace{b^j_{,j} b^\mu + b^j b^\mu_{,j}}_{\text{To Term 2 below}} \tag{19}$$

So,

$$\partial_j T^j_{\rm EM i} = g_{\mu i,j} T^{\mu j}_{\rm EM} \tag{20}$$

$$+ g_{\mu i}\left(\left(b^2\right)_{,j} u^j u^\mu + b^2 u^j_{,j} u^\mu + b^2 u^j u^\mu_{,j} + \frac{1}{2}\left(b^2\right)_{,j} g^{j\mu} + \frac{b^2}{2} g^{j\mu}_{,j} + b^j_{,j} b^\mu + b^j b^\mu_{,j}\right); \tag{21}$$

we will rearrange this once more, collecting the $\left(b^2\right)_{,j}$ terms together, noting that Term A above becomes Term 1 below:

$$\partial_j T^j_{\rm EM i} = \underbrace{g_{\mu i,j} T^{\mu j}_{\rm EM}}_{\text{Term 1}} \tag{22}$$

$$+ \underbrace{g_{\mu i}\left(b^2 u^j_{,j} u^\mu + b^2 u^j u^\mu_{,j} + \frac{b^2}{2} g^{j\mu}_{,j} + b^j_{,j} b^\mu + b^j b^\mu_{,j}\right)}_{\text{Term 2}} \tag{23}$$

$$+ \underbrace{g_{\mu i}\left(\left(b^2\right)_{,j} u^j u^\mu + \frac{1}{2}\left(b^2\right)_{,j} g^{j\mu}\right)}_{\text{Term 3}}. \tag{24}$$

List of Derivatives

Note that this is in terms of the derivatives of several other quantities:

- Step 3.c.i: $B^i_{,l}$: Because b^i is itself a function of B^i, we will first need the derivatives $B^i_{,l}$ in terms of the evolved quantity A_k (the vector potential).
- Step 3.c.ii: $b^i_{,k}$: Once we have $B^i_{,l}$ we can evaluate derivatives of b^i, $b^i_{,k}$
- Step 3.c.iii: The derivative of $b^2 = g_{\mu\nu} b^\mu b^\nu$, $\left(b^2\right)_{,j}$
- Step 3.c.iv: Derivatives of $g^{\mu\nu}$, $g^{\mu\nu}_{,k}$

3.3.1 Step 3.c.i: Derivatives of B^i [Back to List of Derivatives]

First, we will build the derivatives of the magnetic field. Because b^i is a function of B^i, we will start from the definition of B^i in terms of A_k, $B^i = \frac{[ijk]}{\sqrt{\gamma}}\partial_j A_k$ (Eq. 18 of the original GiRaFFE paper). We will first apply the product rule, noting that the symbol $[ijk]$ consists purely

of the integers -1, 0, and 1 and thus can be treated as a constant in this process.

$$B^i_{,l} = \partial_l\left(\frac{[ijk]}{\sqrt{\gamma}}\partial_j A_k\right) \tag{25}$$

$$= [ijk]\partial_l\left(\frac{1}{\sqrt{\gamma}}\right)\partial_j A_k + \frac{[ijk]}{\sqrt{\gamma}}\partial_l\partial_j A_k \tag{26}$$

$$= [ijk]\left(-\frac{\gamma_{,l}}{2\gamma^{3/2}}\right)\partial_j A_k + \frac{[ijk]}{\sqrt{\gamma}}\partial_l\partial_j A_k \tag{27}$$

$$= -\frac{\gamma_{,l}}{2\gamma}\left(\frac{[ijk]}{\sqrt{\gamma}}\partial_j A_k\right) + \frac{[ijk]}{\sqrt{\gamma}}\partial_l\partial_j A_k \tag{28}$$

Now, we will substitute back in for the definition of the Levi-Civita tensor: $\epsilon^{ijk} = [ijk]/\sqrt{\gamma}$. Then we will substitute the magnetic field B^i back in.

$$B^i_{,l} = -\frac{\gamma_{,l}}{2\gamma}\epsilon^{ijk}\partial_j A_k + \epsilon^{ijk}\partial_l\partial_j A_k \tag{29}$$

$$= -\frac{\gamma_{,l}}{2\gamma}B^i + \epsilon^{ijk}A_{k,jl}, \tag{30}$$

Thus, the expression we are left with for the derivatives of the magnetic field is:

$$B^i_{,l} = \underbrace{-\frac{\gamma_{,l}}{2\gamma}B^i}_{\text{Term 1}} + \underbrace{\epsilon^{ijk}A_{k,jl}}_{\text{Term 2}}, \tag{31}$$

where $\epsilon^{ijk} = [ijk]/\sqrt{\gamma}$ is the antisymmetric Levi-Civita tensor and γ is the determinant of the three-metric.

[8]:
```
# Step 2.2: Derivatives of the electromagnetic stress-energy tensor
ixp.register_gridfunctions_for_single_rank2("AUX","gammaUU","sym01")
gri.register_gridfunctions("AUX","gammadet")
gammaUU, gammadet = ixp.symm_matrix_inverter3x3(gammaDD)

# We already have a handy function to define the Levi-Civita symbol in indexedexp.py
# Initialize the Levi-Civita tensor by setting it equal to the Levi-Civita symbol
LeviCivitaSymbolDDD = ixp.LeviCivitaSymbol_dim3_rank3()
LeviCivitaTensorDDD = ixp.LeviCivitaTensorDDD_dim3_rank3(sp.sqrt(gammadet))
LeviCivitaTensorUUU = ixp.LeviCivitaTensorUUU_dim3_rank3(sp.sqrt(gammadet))
```

```
AD_dD = ixp.declarerank2("AD_dD","nosym")

# Step 2.2.a: Construct the derivatives of the magnetic field.
gammadet_dD = ixp.declarerank1("gammadet_dD")

AD_dDD = ixp.declarerank3("AD_dDD","sym12")
# The other partial derivatives of B^i
BUdD = ixp.zerorank2()
for i in range(DIM):
    for l in range(DIM):
        # Term 1: -\gamma_{,l} / (2\gamma) B^i
        BUdD[i][l] = -gammadet_dD[l]*BU[i]/(2*gammadet)

for i in range(DIM):
    for l in range(DIM):
        for j in range(DIM):
            for k in range(DIM):
                # Term 2: \epsilon^{ijk} A_{k,jl}
                BUdD[i][l] += LeviCivitaTensorUUU[i][j][k] * AD_dDD[k][j][l]
```

3.3.2 Step 3.c.ii: Derivatives of b^i [Back to List of Derivatives]

Starting from the definition

$$b^i = \frac{B^i + (u_j B^j) u^i}{\sqrt{4\pi}\alpha u^0},$$

we will now compose the spatial derivatives: $b^i{}_{,k}$.

First, we apply the quotient rule:

$$b^i_{,k} = \frac{\left(\sqrt{4\pi}\alpha u^0\right)\partial_k\left(B^i + (u_j B^j)u^i\right) - \left(B^i + (u_j B^j)u^i\right)\partial_k\left(\sqrt{4\pi}\alpha u^0\right)}{\left(\sqrt{4\pi}\alpha u^0\right)^2} \tag{32}$$

$$= \frac{1}{\sqrt{4\pi}}\frac{\left(\alpha u^0\right)\partial_k\left(B^i + (u_j B^j)u^i\right) - \left(B^i + (u_j B^j)u^i\right)\partial_k\left(\alpha u^0\right)}{\left(\alpha u^0\right)^2} \tag{33}$$

Note that $\left(\alpha u^0\right)$ is being used as its own gridfunction, so $\partial_k\left(\alpha u^0\right)$ will be finite-differenced by NRPy+ directly. We will also apply the product rule to the term $\partial_k\left(B^i + (u_j B^j)u^i\right) = B^i_{,k} + u_{j,k}B^j u^i + u_j B^j{}_{,k} u^i + u_j B^j u^i{}_{,k}$. So,

$$b^i_{,k} = \frac{1}{\sqrt{4\pi}}\frac{\left(\alpha u^0\right)\left(B^i_{,k} + u_{j,k}B^j u^i + u_j B^j{}_{,k} u^i + u_j B^j u^i{}_{,k}\right) - \left(B^i + (u_j B^j)u^i\right)\partial_k\left(\alpha u^0\right)}{\left(\alpha u^0\right)^2}.$$

It will be easier to code this up if we rearrange these terms to group together the terms that involve contractions over j. Doing that, we find

$$b^i_{,k} = \frac{\overbrace{\alpha u^0 B^i_{,k} - B^i\partial_k(\alpha u^0)}^{\text{Term Num1}} + \overbrace{\left(\alpha u^0\right)\left(u_{j,k}B^j u^i + u_j B^j{}_{,k} u^i + u_j B^j u^i{}_{,k}\right)}^{\text{Term Num2.a}} - \overbrace{\left(u_j B^j u^i\right)\partial_k\left(\alpha u^0\right)}^{\text{Term Num2.b}}}{\underbrace{\sqrt{4\pi}\left(\alpha u^0\right)^2}_{\text{Term Denom}}}.$$

[9]:
```
u0alpha = gri.register_gridfunctions("AUX","u0alpha")
u0alpha = alpha * u4upperZero
u0alpha_dD = ixp.declarerank1("u0alpha_dD")
uU_dD = ixp.declarerank2("uU_dD","nosym")
uD_dD = ixp.declarerank2("uD_dD","nosym")

# Step 2.2.b: Construct derivatives of the small b vector
# smallbUdD represents the derivative of smallb4U
smallbUdD = ixp.zerorank2()
for i in range(DIM):
    for k in range(DIM):
        # Term Num1: \alpha u^0 B^i_{,k} - B^i \partial_k (\alpha u^0)
        smallbUdD[i][k] += u0alpha*BUdD[i][k]-BU[i]*u0alpha_dD[k]

for i in range(DIM):
    for k in range(DIM):
        for j in range(DIM):
            # Term Num2.a: terms that require contractions over k, and thus an extra loop.
            # ( \alpha u^0 ) (  u_{j,k} B^j u^i
            #                 + u_j B^j_{,k} u^i
            #                 + u_j B^j u^i_{,k} )
            smallbUdD[i][k] += u0alpha*(uD_dD[j][k]*BU[j]*uU[i]\
```

```
                                        +uD[j]*BUdD[j][k]*uU[i]\
                                        +uD[j]*BU[j]*uU_dD[i][k])

for i in range(DIM):
    for k in range(DIM):
        for j in range(DIM):
            #Term 2.b (More contractions over k): ( u_j B^j u^i ) ( \alpha u^0 ),k
            smallbUdD[i][k] += -(uD[j]*BU[j]*uU[i])*u0alpha_dD[k]

for i in range(DIM):
    for k in range(DIM):
        # Term Denom: Divide the numerator by sqrt(4 pi) * (alpha u^0)^2
        smallbUdD[i][k] /= sp.sqrt(4*M_PI) * u0alpha * u0alpha
```

3.3.3 Step 3.c.iii: Derivative of b^2 [Back to List of Derivatives]

Here, we will take the derivative of $b^2 = g_{\mu\nu} b^\mu b^\nu$. Using the product rule,

$$\left(b^2\right)_{,j} = \partial_j \left(g_{\mu\nu} b^\mu b^\nu\right) \tag{34}$$

$$= g_{\mu\nu,j} b^\mu b^\nu + g_{\mu\nu} b^\mu_{,j} b^\nu + g_{\mu\nu} b^\mu b^\nu_{,j} \tag{35}$$

Recall that, by definition, the metric must be symmetric; that is, $g_{\mu\nu} = g_{\nu\mu}$. Also, consider that we can freely rename bound indices. So, if we swap μ and ν in the final term above, $g_{\mu\nu} b^\mu b^\nu_{,j} = g_{\nu\mu} b^\nu b^\mu_{,j}$ becomes $g_{\mu\nu} b^\nu b^\mu_{,j} = g_{\mu\nu} b^\mu_{,j} b^\nu$. This allows us to combine the final two terms:

$$\left(b^2\right)_{,j} = g_{\mu\nu,j} b^\mu b^\nu + 2 g_{\mu\nu} b^\mu_{,j} b^\nu. \tag{36}$$

We have already defined the spatial derivatives of the four-metric $g_{\mu\nu,j}$ in this section; we have also defined the spatial derivatives of spatial components of b^μ, $b^i_{,k}$ in this section. Notice the above expression for $\partial_j T^j_{\rm EMi}$ requires spatial derivatives of the *zeroth* component of

b^μ as well, $b^0_{,j}$, which we will now compute. Starting with the definition, and applying the quotient rule:

$$b^0 = \frac{u_k B^k}{\sqrt{4\pi}\alpha}, \tag{37}$$

$$\rightarrow b^0_{,j} = \frac{1}{\sqrt{4\pi}} \frac{\alpha \left(u_{k,j} B^k + u_k B^k_{,j}\right) - u_k B^k \alpha_{,j}}{\alpha^2} \tag{38}$$

$$= \frac{\alpha u_{k,j} B^k + \alpha u_k B^k_{,j} - \alpha_{,j} u_k B^k}{\sqrt{4\pi}\alpha^2}. \tag{39}$$

We will first code the numerator, and then divide through by the denominator.

[10]:
```
# Step 2.2.c: Construct the derivative of b^2
# First construct the derivative b^0_{,j}
# This four-vector will make b^2 simpler:
smallb4UdD = ixp.zerorank2(DIM=4)
# Fill in the zeroth component
for j in range(DIM):
    for k in range(DIM):
        # The numerator:  \alpha u_{k,j} B^k
        #               + \alpha u_k B^k_{,j}
        #               - \alpha_{,j} u_k B^k
        smallb4UdD[0][j+1] +=   alpha*uD_dD[k][j]*BU[k] \
                              + alpha*uD[k]*BUdD[k][j] \
                              - alpha_dD[j]*uD[k]*BU[k]
for j in range(DIM):
    # Divide through by the denominator: \sqrt{4\pi} \alpha^2
    smallb4UdD[0][j+1] /= sp.sqrt(4*M_PI)*alpha*alpha
```

At this point, both $b^0_{,j}$ and $b^i_{,j}$ have been computed, but one exists inconveniently in the 4×4 component `smallb4UdD[][]` and the other in the 3×3 component `smallbUdD[][]`. So that we can perform full implied sums over $g_{\mu\nu} b^\mu_{,j} b^\nu$ more conveniently, we will now store all information from `smallbUdD[i][j]` into `smallb4UdD[i+1][j+1]`:

[11]:
```
# Now, we'll fill out the rest of the four-vector with b^i_{,j} that we derived above.
for i in range(DIM):
    for j in range(DIM):
        smallb4UdD[i+1][j+1] = smallbUdD[i][j]
```

Using four-component (Greek-indexed) quantities, we can now complete our construction of

$$\left(b^2\right)_{,j} = g_{\mu\nu,j} b^\mu b^\nu + 2 g_{\mu\nu} b^\mu_{,j} b^\nu :$$

[12]:
```
smallb2_dD = ixp.zerorank1()
for j in range(DIM):
    for mu in range(4):
        for nu in range(4):
            #   g_{\mu\nu,j} b^\mu b^\nu
            # + 2 g_{\mu\nu} b^\mu_{,j} b^\nu
            smallb2_dD[j] +=   g4DDdD[mu][nu][j+1]*smallb4U[mu]*smallb4U[nu] \
                             + 2*g4DD[mu][nu]*smallb4UdD[mu][j+1]*smallb4U[nu]
```

3.3.4 Step 3.c.iv: Derivatives of $g^{\mu\nu}$ [Back to List of Derivatives]

We will need derivatives of the inverse four-metric, as well. Let us begin with g^{00}: since $g^{00} = -1/\alpha^2$ (Gourgoulhon, Eq. 4.49),

$$g^{00}_{,k} = \frac{2\alpha_{,k}}{\alpha^3}$$

[13]:
```
# Step 2.2.d: Construct derivatives of the components of g^{\mu\nu}
g4UUdD = ixp.zerorank3(DIM=4)

for k in range(DIM):
    # 2 \alpha_{,k} / \alpha^3
    g4UUdD[0][0][k+1] = 2*alpha_dD[k]/alpha**3
```

Now, we will code the $g^{i0}_{,k}$ and $g^{0j}_{,k}$ components. According to Gourgoulhon, Eq. 4.49, $g^{i0} = g^{0i} = \beta^i/\alpha^2$, so

$$g^{i0}_{,k} = g^{0i}_{,k} = \frac{\alpha^2 \beta^i_{,k} - 2\beta^i \alpha\alpha_{,k}}{\alpha^4}$$

by the quotient rule. So, we will code

$$g^{i0}_{,k} = g^{0i}_{,k} = \underbrace{\frac{\beta^i_{,k}}{\alpha^2}}_{\text{Term 1}} - \underbrace{\frac{2\beta^i\alpha_{,k}}{\alpha^3}}_{\text{Term 2}}.$$

[14]:
```
for k in range(DIM):
    for i in range(DIM):
        # Term 1:                                   \beta^i_{,k} / \alpha^2
        g4UUdD[i+1][0][k+1] = g4UUdD[0][i+1][k+1] = betaU_dD[i][k] / alpha**2

for k in range(DIM):
    for i in range(DIM):
        # Term 2:              -2 \beta^i \alpha_{,k} / \alpha^3
        g4UUdD[i+1][0][k+1] += -2 * betaU[i] * alpha_dD[k] / alpha**3
        g4UUdD[0][i+1][k+1] += -2 * betaU[i] * alpha_dD[k] / alpha**3
```

We will also need derivatives of the spatial part of the inverse four-metric: since $g^{ij} = \gamma^{ij} - \frac{\beta^i\beta^j}{\alpha^2}$ (Gourgoulhon, Eq. 4.49),

$$g^{ij}_{,k} = \gamma^{ij}_{,k} - \frac{\alpha^2\partial_k(\beta^i\beta^j) - \beta^i\beta^j\partial_k\alpha^2}{(\alpha^2)^2} \tag{40}$$

$$= \gamma^{ij}_{,k} - \frac{\alpha^2\beta^i\beta^j_{,k} + \alpha^2\beta^i_{,k}\beta^j - 2\beta^i\beta^j\alpha\alpha_{,k}}{\alpha^4}. \tag{41}$$

$$= \gamma^{ij}_{,k} - \frac{\alpha\beta^i\beta^j_{,k} + \alpha\beta^i_{,k}\beta^j - 2\beta^i\beta^j\alpha_{,k}}{\alpha^3} \tag{42}$$

$$g^{ij}_{,k} = \underbrace{\gamma^{ij}_{,k}}_{\text{Term 1}} - \underbrace{\frac{\beta^i\beta^j_{,k}}{\alpha^2}}_{\text{Term 2}} - \underbrace{\frac{\beta^i_{,k}\beta^j}{\alpha^2}}_{\text{Term 3}} + \underbrace{\frac{2\beta^i\beta^j\alpha_{,k}}{\alpha^3}}_{\text{Term 4}}. \tag{43}$$

[15]:
```
gammaUU_dD = ixp.declarerank3("gammaUU_dD","sym01")

# The spatial derivatives of the spatial components of the four metric:
# Term 1: \gamma^{ij}_{\ ,k}
for i in range(DIM):
    for j in range(DIM):
        for k in range(DIM):
            g4UUdD[i+1][j+1][k+1] = gammaUU_dD[i][j][k]

# Term 2: - \beta^i \beta^j_{,k} / \alpha^2
for i in range(DIM):
    for j in range(DIM):
```

```
        for k in range(DIM):
            g4UUdD[i+1][j+1][k+1] += -betaU[i]*betaU_dD[j][k]/alpha**2

# Term 3: - \beta^i_{,k} \beta^j / \alpha^2
for i in range(DIM):
    for j in range(DIM):
        for k in range(DIM):
            g4UUdD[i+1][j+1][k+1] += -betaU_dD[i][k]*betaU[j]/alpha**2

# Term 4: 2\beta^i \beta^j \alpha_{,k}\alpha^3
for i in range(DIM):
    for j in range(DIM):
        for k in range(DIM):
            g4UUdD[i+1][j+1][k+1] += 2*betaU[i]*betaU[j]*alpha_dD[k]/alpha**3
```

3.4 Step 3.d: Putting it together: Constructing $\partial_j T^j_{\rm EM}{}_i$ [Back to top]

So, we can now put it all together, starting from the expression we derived above in Step 3.c:

$$\partial_j T^j_{\rm EM}{}_i = \underbrace{g_{\mu i,j} T^{\mu j}_{\rm EM}}_{\text{Term 1}} \tag{44}$$

$$+ \underbrace{g_{\mu i}\left(b^2 u^j_{\ ,j} u^\mu + b^2 u^j u^\mu_{\ ,j} + \frac{b^2}{2} g^{j\mu}_{\ \ ,j} + b^j_{\ ,j} b^\mu + b^j b^\mu_{\ ,j}\right)}_{\text{Term 2}} \tag{45}$$

$$+ \underbrace{g_{\mu i}\left(\left(b^2\right)_{,j} u^j u^\mu + \frac{1}{2}\left(b^2\right)_{,j} g^{j\mu}\right)}_{\text{Term 3}}. \tag{46}$$

3.4.1 Step 3.d.i: Putting it together: Term 1 [Back to top]

We will now construct this term by term. Term 1 is straightforward:

$$\text{Term 1} = g_{\mu i,j} T^{\mu j}_{\rm EM}.$$

[16]:
```
# Step 2.2.e: Construct TEMUDdD_contracted itself
# Step 2.2.e.i
TEMUDdD_contracted = ixp.zerorank1()
for i in range(DIM):
    for j in range(DIM):
        for mu in range(4):
            # Term 1:                 g_{\mu i,j} T^{\mu j}_{\rm EM}
            TEMUDdD_contracted[i] += g4DDdD[mu][i+1][j+1] * T4EMUU[mu][j+1]
```

3.4.2 Step 3.d.ii: Putting it together: Term 2 [Back to top]

We will now add

$$\text{Term 2} = g_{\mu i}\left(\underbrace{b^2 u^j_{\ ,j} u^\mu}_{\text{Term 2a}} + \underbrace{b^2 u^j u^\mu_{\ ,j}}_{\text{Term 2b}} + \underbrace{\frac{b^2}{2} g^{j\mu}_{\ \ ,j}}_{\text{Term 2c}} + \underbrace{b^j_{\ ,j} b^\mu}_{\text{Term 2d}} + \underbrace{b^j b^\mu_{\ ,j}}_{\text{Term 2e}}\right)$$

to $\partial_j T^j_{\rm EM}{}_i$. These are the terms that involve contractions over k (but no metric derivatives as Term 1 had).

All terms have been computed in terms of gridfunctions and their derivatives, except we will find it convenient to define the derivative of the four-velocity `u4UdD[]` in terms of finite difference derivatives of $u^0 =$ `u4upperZero` and $u^i =$ `uU[]`:

[17]:
```
# Step 2.2.e.ii: Compose derivatives of u4U:
u4UdD = ixp.zerorank2(DIM=4)
u4upperZero_dD = ixp.declarerank1("u4upperZero_dD") # We defined u4upperZero as a separate gridfunction so as not␣
 ↪to confuse
                                                    # NRPy+'s finite difference generator code.
for i in range(DIM):
    u4UdD[0][i+1] = u4upperZero_dD[i]
for i in range(DIM):
    for j in range(DIM):
        u4UdD[i+1][j+1] = uU_dD[i][j]

# Step 2.2.e.iii: Add terms 2a--2e to TEMUDdD_contracted[i]
for i in range(DIM):
    for j in range(DIM):
```

```
        for mu in range(4):
            # Term 2a: g_{\mu i} b^2 u^j_{,j} u^\mu
            TEMUDdD_contracted[i] += g4DD[mu][i+1]*smallb2*uU_dD[j][j]*u4U[mu]

for i in range(DIM):
    for j in range(DIM):
        for mu in range(4):
            # Term 2b: g_{\mu i} b^2 u^j u^\mu_{,j}
            TEMUDdD_contracted[i] += g4DD[mu][i+1]*smallb2*uU[j]*u4UdD[mu][j+1]

for i in range(DIM):
    for j in range(DIM):
        for mu in range(4):
            # Term 2c: g_{\mu i} b^2 g^{j \mu}_{,j} / 2
            TEMUDdD_contracted[i] += g4DD[mu][i+1]*smallb2*g4UUdD[j+1][mu][j+1]/2

for i in range(DIM):
    for j in range(DIM):
        for mu in range(4):
            # Term 2d: g_{\mu i} b^j_{,j} b^\mu
            TEMUDdD_contracted[i] += g4DD[mu][i+1]*smallbUdD[j][j]*smallb4U[mu]

for i in range(DIM):
    for j in range(DIM):
        for mu in range(4):
            # Term 2e: g_{\mu i} b^j b^\mu_{,j}
            TEMUDdD_contracted[i] += g4DD[mu][i+1]*smallb4U[j+1]*smallb4UdD[mu][j+1]
```

3.4.3 Step 3.d.iii: Putting it together: Term 3 [Back to top]

Now, we will add

$$\text{Term 3} = g_{\mu i}\left(\underbrace{\left(b^2\right)_{,j}u^j u^\mu}_{\text{Term 3a}} + \underbrace{\frac{1}{2}\left(b^2\right)_{,j}g^{j\mu}}_{\text{Term 3b}}\right).$$

[18]:
```
# Step 2.2.e.iii
for i in range(DIM):
    for j in range(DIM):
        for mu in range(4):
            # Term 3a: g_{\mu i} ( b^2 )_{,j} u^j u^\mu
            TEMUDdD_contracted[i] += g4DD[mu][i+1]*smallb2_dD[j]*uU[j]*u4U[mu]

for i in range(DIM):
    for j in range(DIM):
        for mu in range(4):
            # Term 3b: g_{mu i} ( b^2 )_{,j} g^{j\mu} / 2
            TEMUDdD_contracted[i] += g4DD[mu][i+1]*smallb2_dD[j]*g4UU[j+1][mu]/2
```

4 Step 4: Construct the evolution equation for $\tilde{S}_i$ [Back to top]

Finally, we will return our attention to the time evolution equation (from Eq. 13 of the original GiRaFFE paper),

$$\partial_t \tilde{S}_i = -\partial_j\left(\alpha\sqrt{\gamma}T^j_{\text{EM}i}\right) + \frac{1}{2}\alpha\sqrt{\gamma}T^{\mu\nu}_{\text{EM}}\partial_i g_{\mu\nu} \tag{47}$$

$$= -T^j_{\text{EM}i}\partial_j(\alpha\sqrt{\gamma}) - \alpha\sqrt{\gamma}\partial_j T^j_{\text{EM}i} + \frac{1}{2}\alpha\sqrt{\gamma}T^{\mu\nu}_{\text{EM}}\partial_i g_{\mu\nu} \tag{48}$$

$$= \underbrace{-g_{i\mu}T^{\mu j}_{\text{EM}}\partial_j(\alpha\sqrt{\gamma})}_{\text{Term 1}} \underbrace{-\alpha\sqrt{\gamma}\partial_j T^j_{\text{EM}i}}_{\text{Term 2}} + \underbrace{\frac{1}{2}\alpha\sqrt{\gamma}T^{\mu\nu}_{\text{EM}}\partial_i g_{\mu\nu}}_{\text{Term 3}}. \tag{49}$$

We will first take derivatives of $\alpha\sqrt{\gamma}$, then construct each term in turn.

[19]:
```
# Step 3.0: Construct the evolution equation for \tilde{S}_i
# Here, we set up the necessary machinery to take FD derivatives of alpha * sqrt(gamma)
alpsqrtgam = gri.register_gridfunctions("AUX","alpsqrtgam")
alpsqrtgam = alpha*sp.sqrt(gammadet)
alpsqrtgam_dD = ixp.declarerank1("alpsqrtgam_dD")
```

```
Stilde_rhsD = ixp.zerorank1()
# The first term: g_{i\mu} T^{\mu j}_{\rm EM} \partial_j (\alpha \sqrt{\gamma})
for i in range(DIM):
    for j in range(DIM):
        for mu in range(4):
            Stilde_rhsD[i] += -g4DD[i+1][mu]*T4EMUU[mu][j+1]*alpsqrtgam_dD[j]

# The second term: \alpha \sqrt{\gamma} \partial_j T^j_{{\rm EM} i}
for i in range(DIM):
    Stilde_rhsD[i] += -alpsqrtgam * TEMUDdD_contracted[i]

# The third term: \alpha \sqrt{\gamma} T^{\mu \nu}_{\rm EM} \partial_i g_{\mu \nu} / 2
for i in range(DIM):
    for mu in range(4):
        for nu in range(4):
            Stilde_rhsD[i] += alpsqrtgam * T4EMUU[mu][nu] * g4DDdD[mu][nu][i+1] / 2
```

5 Step 5: Construct the evolution equations for A_i and $\left[\sqrt{\gamma}\Phi\right]$ [Back to top]

We will also need to evolve the vector potential A_i. This evolution is given as Eq. 17 in the GiRaFFE paper:

$$\boxed{\partial_t A_i = \epsilon_{ijk} v^j B^k - \partial_i (\underbrace{\alpha\Phi - \beta^j A_j}_{\text{AevolParen}}),}$$

where $\epsilon_{ijk} = [ijk]\sqrt{\gamma}$ is the antisymmetric Levi-Civita tensor, the drift velocity $v^i = u^i/u^0$, γ is the determinant of the three metric, B^k is the magnetic field, α is the lapse, and β is the shift. We will also need the scalar electric potential Φ, which is evolved by Eq. 19 in the GiRaFFE paper:

$$\boxed{\partial_t [\sqrt{\gamma}\Phi] = -\partial_j (\underbrace{\alpha\sqrt{\gamma}\gamma^{ij} A_i - \beta^j [\sqrt{\gamma}\Phi]}_{\text{PevolParenU[j]}}) - \xi\alpha[\sqrt{\gamma}\Phi],}$$

with ξ chosen as a damping factor.

5.1 Step 5.a: Construct some useful auxiliary gridfunctions for the other evolution equations [Back to top]

After declaring a some needed quantities, we will also define the parenthetical terms (underbrace above) of which we need to take derivatives. That way, we can take finite-difference derivatives easily. Note that the above equations incorporate the fact that γ^{ij} is the appropriate raising operator for A_i: $A^j = \gamma^{ij} A_i$. This is correct because $n_\mu A^\mu = 0$, where n_μ is a normal to the hypersurface, so $A^0 = 0$ (according to Sec. II, subsection C, of the "Improved EM gauge condition" paper of Etienne *et al*).

[20]:
```
# Step 4.0: Construct the evolution equations for A_i and sqrt(gamma)Phi
# Step 4.0.a: Construct some useful auxiliary gridfunctions for the other evolution equations
xi = par.Cparameters("REAL",thismodule,"xi",0.1) # The damping factor

# Define sqrt(gamma)Phi as psi6Phi
psi6Phi = gri.register_gridfunctions("EVOL","psi6Phi")
Phi = psi6Phi / sp.sqrt(gammadet)

# We'll define a few extra gridfunctions to avoid complicated derivatives
AevolParen  = gri.register_gridfunctions("AUX","AevolParen")
PevolParenU = ixp.register_gridfunctions_for_single_rank1("AUX","PevolParenU")

# {\rm AevolParen} = \alpha \Phi - \beta^j A_j
AevolParen = alpha*Phi
for j in range(DIM):
    AevolParen += -betaU[j] * AD[j]

# {\rm PevolParenU[j]} = \alpha\sqrt{\gamma} \gamma^{ij} A_i - \beta^j [\sqrt{\gamma} \Phi]
for j in range(DIM):
    PevolParenU[j] = -betaU[j] * psi6Phi
    for i in range(DIM):
        PevolParenU[j] += alpha * sp.sqrt(gammadet) * gammaUU[i][j] * AD[i]

AevolParen_dD  = ixp.declarerank1("AevolParen_dD")
PevolParenU_dD = ixp.declarerank2("PevolParenU_dD","nosym")
```

5.2 Step 5.b: Complete the construction of the evolution equations for A_i and $\left[\sqrt{\gamma}\Phi\right]$ [Back to top]

Now, we set the evolution equations (Eqs. 17 and 19 of the original GiRaFFE paper), recalling that the drift velocity $v^i = u^i/u^0$:

$$\partial_t A_i = \epsilon_{ijk} v^j B^k - \partial_i (\alpha\Phi - \beta^j A_j) \tag{50}$$

$$= \epsilon_{ijk} \frac{u^j}{u^0} B^k - \text{AevolParen\_dD[i]} \partial_t [\sqrt{\gamma}\Phi] \qquad = -\partial_j \left((\alpha\sqrt{\gamma}) A^j - \beta^j [\sqrt{\gamma}\Phi] \right) - \xi\alpha[\sqrt{\gamma}\Phi] \tag{51}$$

$$= -\text{PevolParenU\_dD[j][j]} - \xi\alpha[\sqrt{\gamma}\Phi]. \tag{52}$$

$$\tag{53}$$

[21]:
```
# Step 4.0.b: Construct the actual evolution equations for A_i and sqrt(gamma)Phi
A_rhsD = ixp.zerorank1()
psi6Phi_rhs = sp.sympify(0)

for i in range(DIM):
    A_rhsD[i] = -AevolParen_dD[i]
    for j in range(DIM):
        for k in range(DIM):
            A_rhsD[i] += LeviCivitaTensorDDD[i][j][k]*(uU[j]/u4upperZero)*BU[k]

psi6Phi_rhs = -xi*alpha*psi6Phi
for j in range(DIM):
    psi6Phi_rhs += -PevolParenU_dD[j][j]
```

6 Step 6: Code validation against GiRaFFE_HO.GiRaFFE_Higher_Order NRPy+ module [Back to top]

Here, as a code validation check, we verify agreement in the SymPy expressions for the GiRaFFE evolution equations and auxiliary quantities we intend to use between 1. this tutorial and 2. the NRPy+ GiRaFFE_HO.GiRaFFE_Higher_Order module.

[22]:
```
# Reset the list of gridfunctions, as registering a gridfunction
#   twice will spawn an error.
gri.glb_gridfcs_list = []

import GiRaFFE_Higher_Order as gho
```



```
gho.GiRaFFE_Higher_Order()

print("Consistency check between GiRaFFE_Higher_Order tutorial and NRPy+ module: ALL SHOULD BE ZERO.")

print("u0alpha - gho.u0alpha = " + str(u0alpha - gho.u0alpha))
print("alpsqrtgam - gho.alpsqrtgam = " + str(alpsqrtgam - gho.alpsqrtgam))
print("AevolParen - gho.AevolParen = " + str(AevolParen - gho.AevolParen))
print("psi6Phi_rhs - gho.psi6Phi_rhs = " + str(psi6Phi_rhs - gho.psi6Phi_rhs))
for i in range(DIM):
    print("uU["+str(i)+"] - gho.uU["+str(i)+"] = " + str(uU[i] - gho.uU[i]))
    print("uD["+str(i)+"] - gho.uD["+str(i)+"] = " + str(uD[i] - gho.uD[i]))
    print("PevolParenU["+str(i)+"] - gho.PevolParenU["+str(i)+"] = " + str(PevolParenU[i] - gho.PevolParenU[i]))
    print("Stilde_rhsD["+str(i)+"] - gho.Stilde_rhsD["+str(i)+"] = " + str(Stilde_rhsD[i] - gho.Stilde_rhsD[i]))
    print("A_rhsD["+str(i)+"] - gho.A_rhsD["+str(i)+"] = " + str(A_rhsD[i] - gho.A_rhsD[i]))
```

```
Consistency check between GiRaFFE_Higher_Order tutorial and NRPy+ module: ALL
SHOULD BE ZERO.
u0alpha - gho.u0alpha = 0
alpsqrtgam - gho.alpsqrtgam = 0
AevolParen - gho.AevolParen = 0
psi6Phi_rhs - gho.psi6Phi_rhs = 0
uU[0] - gho.uU[0] = 0
uD[0] - gho.uD[0] = 0
PevolParenU[0] - gho.PevolParenU[0] = 0
Stilde_rhsD[0] - gho.Stilde_rhsD[0] = 0
A_rhsD[0] - gho.A_rhsD[0] = 0
uU[1] - gho.uU[1] = 0
uD[1] - gho.uD[1] = 0
PevolParenU[1] - gho.PevolParenU[1] = 0
Stilde_rhsD[1] - gho.Stilde_rhsD[1] = 0
A_rhsD[1] - gho.A_rhsD[1] = 0
uU[2] - gho.uU[2] = 0
uD[2] - gho.uD[2] = 0
PevolParenU[2] - gho.PevolParenU[2] = 0
Stilde_rhsD[2] - gho.Stilde_rhsD[2] = 0
A_rhsD[2] - gho.A_rhsD[2] = 0
```

7 Step 7: Output this notebook to LaTeX-formatted PDF file [Back to top]

The following code cell converts this Jupyter notebook into a proper, clickable LaTeX-formatted PDF file. After the cell is successfully run, the generated PDF may be found in the root NRPy+ tutorial directory, with filename `Tutorial-GiRaFFE_Higher_Order.pdf` (Note that clicking on this link may not work; you may need to open the PDF file through another means.)

[23]:
```
import cmdline_helper as cmd    # NRPy+: Multi-platform Python command-line interface
cmd.output_Jupyter_notebook_to_LaTeXed_PDF("Tutorial-GiRaFFE_Higher_Order",location_of_template_file=os.path.
  ↪join(".."))
```

```
Created Tutorial-GiRaFFE_Higher_Order.tex, and compiled LaTeX file to PDF
    file Tutorial-GiRaFFE_Higher_Order.pdf
```

References

[1] R. A. Hulse and J. H. Taylor. Discovery of a pulsar in a binary system. *The Astrophysical Journal Letters*, 195:L51–L53, January 1975.

[2] B. P. Abbott et al. Observation of Gravitational Waves from a Binary Black Hole Merger. *Phys. Rev. Lett.*, 116(6):061102, 2016.

[3] B. P. Abbott et al. GW170817: Observation of Gravitational Waves from a Binary Neutron Star Inspiral. *Phys. Rev. Lett.*, 119(16):161101, 2017.

[4] John Archibald Wheeler Charles W. Misner, Kip S. Thorne. *Gravitation*. Princeton University Press, 2017.

[5] Stuart L Teukolsky, Saul A.; Shapiro. *Black holes, white dwarfs, and neutron stars : the physics of compact objects*. Physics textbook. Wiley-VCH, 2004.

[6] R N Manchester. The international pulsar timing array. *Classical and Quantum Gravity*, 30(22):224010, nov 2013.

[7] S. S. Komissarov. Electrodynamics of black hole magnetospheres. *Monthly Notices of the Royal Astronomical Society*, 350(2):427–448, 05 2004.

[8] R. Arnowitt, S. Deser, and C. W. Misner. Dynamical Structure and Definition of Energy in General Relativity. *Physical Review*, 116(5):1322–1330, December 1959.

[9] Eric Gourgoulhon. 3+1 formalism and bases of numerical relativity. 3 2007.

[10] Zachariah B. Etienne. Lecture notes, math 522, spring 2019 edition, February 2020. `http://astro.phys.wvu.edu/zetienne/MATH522-s2019/notes.pdf`.

[11] Carpet: Adaptive mesh refinement for cactus framework. `https://www.carpetcode.org`.

[12] Wolfram Research, Inc. Mathematica, Version 12.3.1. Champaign, IL, 2021.

[13] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.

[14] Kranc: Kranc assembles numerical code. `http://kranccode.org/`.

[15] Z. B. Etienne and I. Ruchlin. NRPy+: Code generator for Numerical Relativity. Astrophysics Source Code Library, July 2018.

[16] Robert M. Wald. Black hole in a uniform magnetic field. *Phys. Rev. D*, 10:1680–1685, Sep 1974.

[17] R. D. Blandford and R. L. Znajek. Electromagnetic extraction of energy from Kerr black holes. *Monthly Notices of the Royal Astronomical Society*, 179(3):433–456, 07 1977.

[18] Anatoly Spitkovsky. Time-dependent force-free pulsar magnetospheres: Axisymmetric and oblique rotators. *The Astrophysical Journal*, 648(1):L51–L54, aug 2006.

[19] Taner Akgün, Pablo Cerdá-Durán, Juan Antonio Miralles, and José A. Pons. Long-term evolution of the force-free twisted magnetosphere of a magnetar. *Mon. Not. Roy. Astron. Soc.*, 472(4):3914–3923, 2017.

[20] Kevin Thoelecke, Masaaki Takahashi, and Sachiko Tsuruta. Effects of inner Alfvén surface location on black hole energy extraction in the limit of a force-free magnetosphere. *Phys. Rev. D*, 95(6):063008, 2017.

[21] Masanori Nakamura et al. Parabolic Jets from the Spinning Black Hole in M87. *Astrophys. J.*, 868(2):146, 2018.

[22] Kazunori Akiyama et al. First M87 Event Horizon Telescope Results. I. The Shadow of the Supermassive Black Hole. *Astrophys. J. Lett.*, 875:L1, 2019.

[23] J. F. Mahlmann, M. A. Aloy, V. Mewes, and P. Cerdá-Durán. Computational General Relativistic Force-Free Electrodynamics: I. Multi-Coordinate Implementation and Testing. *Astron. Astrophys.*, 647:A57, 2021.

[24] Patrick E. Nelson, Zachariah B. Etienne, Sean T. McWilliams, and Viviana Nguyen. Induced Spins from Scattering Experiments of Initially Nonspinning Black Holes. *Phys. Rev. D*, 100(12):124045, 2019.

[25] E. D. Bloom, D. H. Coward, H. DeStaebler, J. Drees, G. Miller, L. W. Mo, R. E. Taylor, M. Breidenbach, J. I. Friedman, G. C. Hartmann, and H. W. Kendall. High-energy inelastic $e - p$ scattering at 6° and 10°. *Phys. Rev. Lett.*, 23:930–934, Oct 1969.

[26] M. Breidenbach, J. I. Friedman, H. W. Kendall, E. D. Bloom, D. H. Coward, H. DeStaebler, J. Drees, L. W. Mo, and R. E. Taylor. Observed behavior of highly inelastic electron-proton scattering. *Phys. Rev. Lett.*, 23:935–939, Oct 1969.

[27] James B. Hartle. Tidal shapes and shifts on rotating black holes. *Phys. Rev. D*, 9:2749–2759, May 1974.

[28] M. Campanelli, C. O. Lousto, and Y. Zlochower. Spin-orbit interactions in black-hole binaries. *Phys. Rev. D*, 74(8):084023, October 2006.

[29] Carlos O. Lousto and Yosef Zlochower. A Practical formula for the radiated angular momentum. *Phys. Rev.*, D76:041502, 2007.

[30] A. Ashtekar and B. Krishnan. Isolated and Dynamical Horizons and Their Applications. *Living Reviews in Relativity*, 7, December 2004.

[31] Miguel Alcubierre et al. Dynamical evolution of quasi-circular binary black hole data. *Phys. Rev.*, D72:044004, 2005.

[32] Eric Poisson and Misao Sasaki. Gravitational radiation from a particle in circular orbit around a black hole. 5: Black hole absorption and tail corrections. *Phys. Rev.*, D51:5753–5767, 1995.

[33] S Chandrasekhar. *The mathematical theory of black holes*. Oxford classic texts in the physical sciences. Oxford Univ. Press, Oxford, 2002.

[34] Karl Martel. Gravitational wave forms from a point particle orbiting a Schwarzschild black hole. *Phys. Rev.*, D69:044025, 2004.

[35] Janna Levin and Gabe Perez-Giz. A Periodic Table for Black Hole Orbits. *Phys. Rev.*, D77:103005, 2008.

[36] R. Grossman and J. Levin. Dynamics of black hole pairs. II. Spherical orbits and the homoclinic limit of zoom-whirliness. *Phys. Rev. D*, 79(4):043017, February 2009.

[37] J. Levin and R. Grossman. Dynamics of black hole pairs. I. Periodic tables. *Phys. Rev. D*, 79(4):043016, February 2009.

[38] K. Glampedakis and D. Kennefick. Zoom and whirl: Eccentric equatorial orbits around spinning black holes and their evolution under gravitational radiation reaction. *Phys. Rev. D*, 66(4):044002, August 2002.

[39] Steve Drasco and Scott A. Hughes. Gravitational wave snapshots of generic extreme mass ratio inspirals. *Phys. Rev.*, D73(2):024027, 2006. [Erratum: Phys. Rev.D90,no.10,109905(2014)].

[40] Roland Haas. Scalar self-force on eccentric geodesics in Schwarzschild spacetime: A Time-domain computation. *Phys. Rev.*, D75:124011, 2007.

[41] Scott A. Hughes, Steve Drasco, Éanna É. Flanagan, and Joel Franklin. Gravitational radiation reaction and inspiral waveforms in the adiabatic limit. *Phys. Rev. Lett.*, 94:221101, Jun 2005.

[42] L Bombelli and E Calzetta. Chaos around a black hole. *Classical and Quantum Gravity*, 9(12):2573–2599, dec 1992.

[43] Neil J. Cornish and Janna Levin. Lyapunov timescales and black hole binaries. *Classical and Quantum Gravity*, 20(9):1649–1660, May 2003.

[44] F. Pretorius and D. Khurana. Black hole mergers and unstable circular orbits. *Classical and Quantum Gravity*, 24:S83–S108, June 2007.

[45] J. Healy, J. Levin, and D. Shoemaker. Zoom-Whirl Orbits in Black Hole Binaries. *Physical Review Letters*, 103(13):131101, September 2009.

[46] R. Gold and B. Brügmann. Eccentric black hole mergers and zoom-whirl behavior from elliptic inspirals to hyperbolic encounters. *Phys. Rev. D*, 88(6):064051, September 2013.

[47] R. Gold and B. Brügmann. Radiation from low-momentum zoom-whirl orbits. *Classical and Quantum Gravity*, 27(8):084035, April 2010.

[48] J. Levin, S. T. McWilliams, and H. Contreras. Inspiral of generic black hole binaries: spin, precession and eccentricity. *Classical and Quantum Gravity*, 28(17):175001, September 2011.

[49] Roman Gold and Bernd Brügmann. Eccentric black hole mergers and zoom-whirl behavior from elliptic inspirals to hyperbolic encounters. *Phys. Rev.*, D88(6):064051, 2013.

[50] M. Shibata and T. Nakamura. Evolution of three-dimensional gravitational waves: Harmonic slicing case. *Phys. Rev. D*, 52:5428–5444, November 1995.

[51] T. W. Baumgarte and S. L. Shapiro. Numerical integration of Einstein's field equations. *Phys. Rev. D*, 59(2):024007, January 1999.

[52] F. Löffler, J. Faber, E. Bentivegna, T. Bode, P. Diener, R. Haas, I. Hinder, B. C. Mundim, C. D. Ott, E. Schnetter, G. Allen, M. Campanelli, and P. Laguna. The Einstein Toolkit: a community computational infrastructure for relativistic astrophysics. *Classical and Quantum Gravity*, 29(11):115001, June 2012.

[53] The Einstein Toolkit. `https://einsteintoolkit.org`.

[54] M. Ansorg, B. Brügmann, and W. Tichy. Single-domain spectral method for black hole puncture data. *Phys. Rev. D*, 70(6):064011, September 2004.

[55] V. Paschalidis, Z. B. Etienne, R. Gold, and S. L. Shapiro. An efficient spectral interpolation routine for the TwoPunctures code. *ArXiv e-prints*, April 2013.

[56] J. David Brown, Peter Diener, Olivier Sarbach, Erik Schnetter, and Manuel Tiglio. Turduckening black holes: an analytical and computational study. *Phys. Rev. D*, 79:044023, 2009.

[57] McLachlan, a public BSSN code. `http://www.cct.lsu.edu/~eschnett/McLachlan/`.

[58] O. Dreyer, B. Krishnan, D. Shoemaker, and E. Schnetter. Introduction to isolated horizons in numerical relativity. *Phys. Rev. D*, 67(2):024018, January 2003.

[59] J. Thornburg. A fast apparent horizon finder for three-dimensional Cartesian grids in numerical relativity. *Classical and Quantum Gravity*, 21:743–766, January 2004.

[60] Jonathan Thornburg. Finding apparent horizons in numerical relativity. *Phys. Rev. D*, 54:4899–4918, 1996.

[61] M. Zilhão and F. Löffler. An Introduction to the Einstein Toolkit. *International Journal of Modern Physics A*, 28:1340014–126, September 2013.

[62] Tom Goodale, Gabrielle Allen, Gerd Lanfermann, Joan Massó, Thomas Radke, Edward Seidel, and John Shalf. *High Performance Computing for Computational Science — VECPAR 2002: 5th International Conference Porto, Portugal, June 26–28, 2002 Selected Papers and Invited Talks*, chapter The Cactus Framework and Toolkit: Design and Applications, pages 197–227. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[63] Cactus computational toolkit. `https://www.cactuscode.org`.

[64] Cactus Computational Toolkit Prizes. `http://cactuscode.org/media/prizes/`.

[65] E. Schnetter, S. H. Hawley, and I. Hawke. Evolutions in 3D numerical relativity using fixed mesh refinement. *Classical and Quantum Gravity*, 21:1465–1488, March 2004.

[66] Ian Ruchlin, James Healy, Carlos O. Lousto, and Yosef Zlochower. Puncture Initial Data for Black-Hole Binaries with High Spins and High Boosts. *Phys. Rev.*, D95(2):024033, 2017.

[67] M. Campanelli and C. O. Lousto. Second order gauge invariant gravitational perturbations of a Kerr black hole. *Phys. Rev. D*, 59(12):124022, June 1999.

[68] Ian Ruchlin, Zachariah B. Etienne, and Thomas W. Baumgarte. SENR/NRPy+: Numerical Relativity in Singular Curvilinear Coordinate Systems. *Phys. Rev.*, D97(6):064036, 2018.

[69] I. Ruchlin, Z. B. Etienne, and T. W. Baumgarte. SENR: Simple, Efficient Numerical Relativity. Astrophysics Source Code Library, July 2018.

[70] John G. Baker, Manuela Campanelli, and Carlos O. Lousto. The Lazarus project: A Pragmatic approach to binary black hole evolutions. *Phys. Rev. D*, 65:044001, 2002.

[71] Jose M. Martin-Garcia, Renato Portugal, and Leon R. U. Manssur. The Invar Tensor Package. *Comput. Phys. Commun.*, 177:640–648, 2007.

[72] Zachariah B. Etienne, Mew-Bing Wan, Maria C. Babiuc, Sean T. McWilliams, and Ashok Choudhary. GiRaFFE: an open-source general relativistic force-free electrodynamics code. *Classical and Quantum Gravity*, 34(21):215001, November 2017.

[73] Phillip Colella and Paul R Woodward. The piecewise parabolic method (ppm) for gas-dynamical simulations. *Journal of Computational Physics*, 54(1):174–201, 1984.

[74] Amiram Harten, Peter Lax, and Bram van Leer. On upstream differencing and godunov-type schemes for hyperbolic conservation laws. *SIAM Rev*, 25:35–61, 01 1983.

[75] Bernd Einfeldt. On godunov-type methods for gas dynamics. *Siam Journal on Numerical Analysis - SIAM J NUMER ANAL*, 25:294–318, 04 1988.

[76] Bruno Giacomazzo and Luciano Rezzolla. WhiskyMHD: a new numerical code for general relativistic magnetohydrodynamics. *Classical and Quantum Gravity*, 24(12):S235–S258, June 2007.

[77] Matthew D. Duez, Yuk Tung Liu, Stuart L. Shapiro, and Branson C. Stephens. Relativistic magnetohydrodynamics in dynamical spacetimes: Numerical methods and tests. *Phys. Rev. D*, 72:024028, 2005.

[78] Zachariah B. Etienne, Vasileios Paschalidis, Yuk Tung Liu, and Stuart L. Shapiro. Relativistic MHD in dynamical spacetimes: Improved EM gauge condition for AMR grids. *Phys. Rev. D*, 85:024013, 2012.

[79] Vasileios Paschalidis and Stuart L. Shapiro. A new scheme for matching general relativistic ideal magnetohydrodynamics to its force-free limit. *Phys. Rev. D*, 88(10):104031, November 2013.

[80] L. Del Zanna, N. Bucciantini, and P. Londrillo. An efficient shock-capturing central-type scheme for multidimensional relativistic flows. II. Magnetohydrodynamics. *Astron. Astrophys.*, 400:397–413, March 2003.

[81] Santiago Jaraba and Juan Garcia-Bellido. Black hole induced spins from hyperbolic encounters in dense clusters. 6 2021.