Enabling Design-Oriented Fluid Simulations: Verification with Discontinuous Manufactured Solutions and Automatic Grid Generation with Moving Coordinates

by

C. Nathan Woods

B.S., Brigham Young University, 2008

M.S., University of Colorado Boulder, 2011

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Aerospace Engineering Sciences

2015

This thesis entitled:
Enabling Design-Oriented Fluid Simulations:
Verification with Discontinuous Manufactured Solutions and Automatic Grid Generation with Moving Coordinates
written by C. Nathan Woods
has been approved for the Department of Aerospace Engineering Sciences

Ryan P. Starkey

Dr. Brian Argrow

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Woods, C. Nathan (Ph.D., Aerospace Engineering)

Enabling Design-Oriented Fluid Simulations:

Verification with Discontinuous Manufactured Solutions and Automatic Grid Generation with Moving Coordinates

Thesis directed by Dr. Ryan P. Starkey

Computer simulations of complex mathematical models are a powerful tool for design, but they introduce uncertainties which can lead to poor design choices when simulation data is all that is available. Additionally, computational grid generation can dramatically increases the costs associated with initializing numerical simulations. Proper verification can help quantify the uncertainty in numerical simulations, and a new form of code verification is presented. This is based on the method of manufactured solutions for integral equations, which allows MMS to be used to verify shock-capturing codes. A procedure is presented for numerically evaluating the required integrals, and it is found to completely eliminate numerical error resulting from discontinuous integrand functions. Integral MMS is demonstrated, and it is found to yield convergence rates that differ by less than 5% from those obtained using differential MMS, and which match precisely with the theoretical rates for discontinuous solutions. This indicates that integral MMS can be used for code verification in place of differential MMS, which cannot be used with discontinuous solutions. Moving grids can be used to allow computed fluid motion to generate the computational grid automatically. The unique challenges associated with grid motion are explored, and multiple implementations are discussed. A software library for fluid-mechanical simulation in unsteady coordinates is also introduced. Preliminary verification of both the method and the library is discussed. The use of unsteady coordinates affects accuracy and grid convergence rates in complex ways. This work lays the foundation for future work on the use of moving grids in order to reduce the grid-generation burden for design-oriented computational fluid dynamics.

Dedication

To my loving wife, for all her patience, support, and countless hours as an editor.

Acknowledgements

I want to thank all of those members of the Department of Aerospace Engineering Sciences who contributed to this work. In particular, the members of my committee for the advice and insights they provided, Dr. John Evans, who was instrumental in providing me a window into the applied math world, and the many members of the Busemann Advanced Concepts Lab who were always willing to listen to me talk about my work, even when I was so far in the weeds I couldn't see the sky.

I also want to express my gratitude to my advisor, Dr. Ryan Starkey, for his continued support throughout this long process. He gave me my independence, worked with me tirelessly, and in every way enabled the production of this dissertation.

I would be remiss if I didn't also express my appreciation for my wife, who has kept the family in groceries and kept me in good spirits, all while acting as an excellent ad-hoc, professional, scientific editor.

Finally, I am especially grateful to God for all of the little things that didn't go wrong, for all the lucky decisions I made when I didn't know any better, for my good health throughout my dissertation, and for the circumstances of my life that enabled me to contemplate graduate school in the first place.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

The modern world, with all its conveniences and progress, has largely been shaped by successive attempts to design tools that are more powerful, more efficient, and more specialized. In the past, many of these designs have come from flashes of insight, mathematical analysis, or trial-and-error, but all of these sources have become less reliable as we have pushed the bounds of possibility and the acceptable design space has become more constrained. This can be clearly seen in the evolution of aircraft design. First, the Wright brothers tested airfoil shapes in order to achieve the first powered flight [1, 2]. Later designers used specialized wind tunnels and scale models along with simple mathematical tools in order to design the aircraft that fought both world wars. As system performance requirements increased, ever more detailed tests and models were required, to the point where current system design cycles may last for decades, and yet still have serious design issues that are only discovered in testing of the full, integrated system [3]. A similar trend can be seen across every field of human endeavor, from business to sports to politics, where increasingly complex designs are required in order to expand the performance envelope. Since it is unlikely that performance requirements will ever go down, it is critical that we develop ways to incorporate quantitative, high-fidelity tests and models early in the design process.

The development and use of numerical simulation tools during the twentieth century represented a revolution in engineering design [4, 5]. Numerical simulations allowed analysts and designers to use physics-based mathematical models to predict the behavior of complex systems directly, rather than relying on the greatly simplified models that were required in order to obtain analytic

solutions [6]. These improved models can reduce the uncertainty in predictive analytics, thus allowing users to identify workable design points more quickly in an ever-more-constrained design space [4, 5]. If this can be done early in the project lifecycle, then the information provided can be used to dramatically reduce the incidence of missed deadlines, cost overruns, and other setbacks that can lead to project failure in severe cases [7].

The lifecycle of a complex project can be divided into several distinct steps, as shown in figure 1.1, ranging from initial conception through development and on to logistics and maintenence. Many of the most critical design decisions are made during the early design phases (Concept of Operations, System and Sub-system Requirements). These decisions have a tremendous effect on the overall success of the project, yet they must be made in an extremely information-poor environment. The development and regular use of powerful predictive tools may allow designers to somewhat remedy this information deficit, provided the tools are accessible and reliable enough to incorporate early in the normal development process [6].

Unfortunately, powerful computational tools have historically been unwieldy and have often required specialized experts to use them [6]; it has not been uncommon for a dedicated team to spend weeks simply setting up a numerical simulation, much less the time required to obtain a solution. In the early design stages, one frequently desires to simulate several different designs simultaneously, and one may need to turnaround the results overnight [6]. Under such constraints, detailed numerical simulations simply do not make sense for many applications.

Additionally, the codes used to perform numerical simulations are quite feature-rich and complex, providing functionality for distributed-memory parallelism, data I/O, and pre- and post-processing, in addition to the numerical solution being implemented. As a result, these codes are normally too complex for developers to easily and reliably verify–that is, it is difficult to establish definitively the accuracy with which these codes are solving the underlying mathematical model upon which they are based [9]. This is extremely important, as it allows project managers to make critical design decisions based on the results of numerical simulations, with a reasonable knowledge of the error that these simulations are expected to contain.

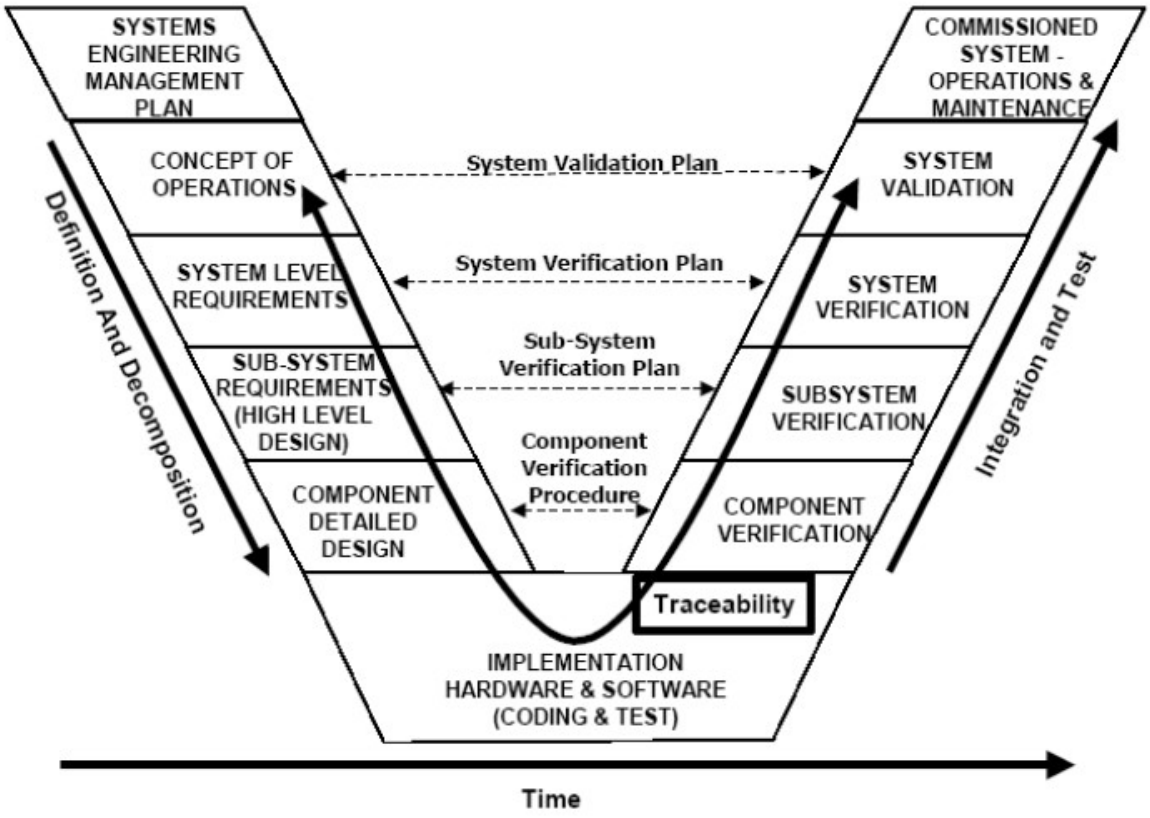


Figure 1.1: The Project Life Cycle [8]. At present, most high-fidelity tools for predictive analysis are used primarily at the Component Detailed Design level and later, because of the effort that they require. Unfortunately, many critical decisions are made earlier, during the Conceptual and Requirements phases. Making numerical simulations more reliable and simpler to use will help them to be more useful in the very early stages of design.

In this dissertation, I have investigated and developed solutions that aim to address these issues. First, I have developed a new method for establishing reliably the accuracy with which a given numerical simulation tool solves its computational model, even in the presence of discontinuous solutions. This is a critical capability in many fields, including especially high-speed aerodynamics, because it allows developers to quickly, easily, and thoroughly test their codes after every algorithm change or feature addition [6, 9]. Second, I have developed a software package for simulation of inviscid, compressible fluid flows without the prior generation of a computational grid. Because grid generation accounts for a large portion of the difficulty of running numerical simulations [6], it is anticipated that eliminating this step will greatly simplify the use of simulation tools in the early stages of product design cycles, where they would be the most valuable.

1.1 Integrative, Shock-Capturing, Method of Manufactured Solutions

Establishing the accuracy of numerical simulation tools is an important step in the advancement of numerical methods, and much work has been done in this field over the years. [9, 10] As a result, the method of manufactured solutions has developed into a widely accepted, general method for establishing the accuracy with which a given numerical simulation code solves its underlying equations. Verification using manufactured solutions is now accepted as the gold-standard method for demonstrating that a code correctly implements the numerical models upon which it based [11], and this has allowed developers to, for the first time, establish a finite testing regime that provides rigorous estimates for the error levels one should expect to find in solutions the code provides [9,12]. Unfortunately, the method of manufactured solutions has historically been useable only for verification of smooth problems [9,12,13]. As a result, codes which simulated important problems involving shocks, discontinuous interfaces, or other non-smooth solution behavior were not able to be verified in this way.

The method of manufactured solutions has historically been inherently unsuited to the verification of shock-capturing codes because it explicitly requires that the verification solutions be everywhere differentiable. [9] A code that is intended to simulate discontinuous systems such as

supersonic, inviscid fluid flows cannot be verified against smooth solutions alone, because smooth solutions will not exercise the aspects of the code that deal with discontinuities, such as flux limiters. Several researchers including myself have focused on this problem in recent years, [14–16] with an emerging consensus that integral methods provide a direct extension of the method of manufactured solution to the verification of shock-capturing and other discontinuous numerical simulations.

1.1.1 Integral Manufactured Solutions

Integral manufactured solutions result from a straightforward application of the method of manufactured solutions to the weak, or integral form of systems of differential equations [15–17]. Unfortunately, practical implementation difficulties have precluded their use until recently. Differential manufactured solutions are implemented by analytically operating a system of differential equations on a given manufactured solution, in order to obtain analytic source terms that can be used to drive the code being verified [9, 13, 18]. This naturally causes problems when applied to discontinuous solutions, such as those encountered in shocked flows, because these solutions are, by definition, non-differentiable. Operation of an integral system of equations on a discontinuous solution presents no such difficulty, as there is no explicit requirement that the solution be differentiable in order to integrate it with respect to its arguments. As a result, it is possible to directly apply the methodology of manufactured solutions to shocked and other discontinuous solutions, provided that an integral mathematical model is used [15, 16].

1.1.2 Evaluation of Multidimensional Integrals of Discontinuous Functions

Unfortunately, integral equations are much less amenable to analytic evaluation than are differential equations. For complex mathematical models and the complex manufactured solutions required to verify the codes that solve them, analytic evaluation of the multidimensional integrals involved is impossible, and one must resort to quadrature techniques for numerical integration [15, 16]. As the results of these integrals are used to prove/disprove the accuracy of other codes, it is critical that the integrals be evaluated as accurately as possible. As many such integrals must

be evaluated, computational efficiency is also crucial. Widely available numerical integration tools are poorly suited to accurate, efficient evaluation of multidimensional integrals of discontinuous functions when grid faces are not shock-aligned [15]. In chapter 2, I present a new method for accurately evaluating such integrals, based on a new technique for mapping discontinuities through one-dimensional integration operators. I also present a verification of this integration method which demonstrates the extreme accuracy that it obtains. Finally, I present the results of timing tests which firmly establish the utility of this new method.

1.1.3 Implementing Integral Manufactured Solutions

Once the problem of integration is resolved, implementation of integral manufactured solutions can be addressed. Verification using manufactured solutions has typically relied on symbolic algebra software and code generation in order to manage the complexity of the symbolic manipulations involved [9, 19]. Integral manufactured solutions benefit from the same treatment; symbolic manipulators and code generation can be used to dramatically reduce the burden of preparing integrals for numerical evaluation. In chapter 3, I present a software package that I developed using open-source computational tools in order to automate the generation of both differential and integral manufactured source terms for general systems of equations. I also show how this package can be used in the verification of numerical simulation software, by presenting the verification of an in-house Euler equations solver. Verification is performed using differential and integral manufactured solutions, exact solutions, and also an independent, third-party code. The results of these are compared, and integral manufactured solutions are found to compare favorably with differential manufactured solutions for smooth problems. For discontinuous problems, integral manufactured solutions encounter no difficulties, and perform at a level that is on par with exact discontinuous solutions, where these are available.

1.2 Elimination of Mesh Generation Via Hui's Unified Coordinates

Making numerical simulation tools available to early-stage designers is important to the success of future design projects, but accomplishing this requires a clear understanding of the underlying problems. Numerical simulation tools have made tremendous strides in model accuracy, algorithm efficiency, and problem size, yet these have resulted in design improvements only later the project life cycle. The critical difficulty that must be overcome is no longer the numerical simulation itself, but rather the time and effort required to set up a simulation in the first place [6]. Experience shows that, for designers who are not expert in numerical simulations, the set-up costs may be substantially higher than the eventual run-time costs. In the early stages of a project, when several designs may need to be evaluated each day, this initial startup cost us simply unacceptable, and so projects must do without the additional information that modeling and simulation could provide, until after many of the most critical design decisions have been made.

One of most important startup costs inherent in most numerical simulations is the generation of a computational grid upon which to represent the mathematical model equations [6]. Grid generation, which essentially represents a particular discretization of space and time (the independent variables in a simulation), can have unexpectedly large effects on the results of any given simulation. That is, a poor choice of computational grid can dramatically increase error in the results of the simulation, even if the equations remain unchanged [20, 21]. Therefore, it is critical that the model equations be discretized onto a grid that is "good". Unfortunately, "good" essentially means "yields computational results with the expected level of accuracy", and is often only recognizable a posteriori. Many techniques have been proposed to automate the process of generating high quality grids, yet all of these require some level of end-user involvement in order to obtain satisfactory results.

One class of techniques for automatic grid generation that shows some promise defines the grid together with the system being modeled. That is, the spatial discretization is chosen based on the state of the simulation variables. In the field of fluid mechanics, this can be done by allowing

the grid to move in some way that is correlated with the motion of the fluid itself. Defining an unsteady grid in this way allows the bulk of the grid to be automatically generated and tailored to the particular flow being modeled, which can result in a higher-accuracy simulation overall. In chapter 4, I present a software package that I developed to numerically model inviscid fluid flows on moving computational grids. The structured-grid method I used is based on Hui's unified coordinate system [21], which uses grid points that move with a velocity vector proportional to the fluid velocity vector. The software package presented in chapter 4 has been through several development versions, and I have included a brief discussion of these in order to present the insights and lessons learned from this development process. I also present several demonstration problems, in order to showcase how Hui's unified coordinates may eliminate some of the startup costs associated with numerical simulations, and thus make fluid simulation tools available much earlier in the project design process.

Chapter 2

Multidimensional Integration of Discontinuous Functions

As introduced in chapter 1, one of the key requirements for the use of numerical simulations in early design stages is a means of verifying the accuracy of the computer codes that perform these simulations. For many important classes of simulations, integral manufactured solutions provide a natural extension of the differential method of manufactured solutions that has been used to great success in many fields of numerical simulation, yet they encounter no problems when used to verify the accuracy of a code in the presence of discontinuous solutions. Unfortunately, integral manufactured solutions do require the evaluation of multidimensional integrals of complicated, discontinuous functions to a high degree of accuracy, and this must be done many times. To do this in general requires the use of a numerical integration routine that can efficiently evaluate multidimensional integrals of discontinuous functions to a high degree of accuracy.

Software that is used for the verification of computer codes faces special challenges with respect to overall accuracy, because of the nature of software verification. In effect, verification software is used to prove the inaccuracy of codes written by others. In such an application, accuracy is absolutely critical. Any inaccuracy in the verification software undermines the overall trust in the verification procedure itself, which may lead to false-positive and false-negative verifiaction tests, which can undermine the entire procedure of software verification, essentially leading researchers to discount the procedure completely.

To resolve this difficulty, traditional MMS has relied on the inherent differentiability of manufactured source terms using symbolic algebra tools. [9,19] MMS libraries may even include arbitrary-

precision arithmetic in order to further guarantee that the manufactured source terms are computed as accurately as possible. [19,22] Unfortunately, this is not typically an option for integral manufactured solutions. The complex functions that arise from the operation of many mathematical models on sufficiently general manufactured solutions simply do not admit analytic integration solutions to be found. As a result, one must instead rely on numerical integration tools.

The tools used in numerical integration yield, of necessity, approximate solutions to integrals. As a result, introducing numerical integration into the MMS algorithm must therefore introduce error into the manufactured source terms. Because this error leads to ambiguity in the reliability of MMS verification, it must be kept small. Research into integral MMS is still in its early stages [15–17], and so the exact amount of error that can be tolerated is not yet well-understood. As a result, I have chosen to require the highest level of accuracy that can be obtained using widely available tools for smooth functions, and a negligible increase in error for discontinuous functions. Although this may be computationally inefficient, it will provide a reference against which to compare future algorithms that take a more balanced approach to compational accuracy and efficiency.

The remainder of this chapter will be structured as follows. First, I will discuss some of the readily available computational tools for numerical integration using quadrature, as well as some of the newer algorithms that have particular bearing on the calculation of integral manufactured solutions. Second, I will discuss a method for handling discontinuous integrand functions in such a way as to allow standard quadrature tools to integrate them accurately. This will include a conceptual overview of the method, extensions to the SciPy quadrature package that were required in order to implement it, and an additional symbolic manipulation package in Sympy that automates the process of integrating discontinuous functions. Third, I will demonstrate both the accuracy and the computational performance of this integration procedure using several real-world problems drawn from the context of integral manufactured solutions.

2.1 Background on Numerical Integration and Available Tools

2.1.1 Monte Carlo Methods vs. Quadrature

Numerical integration methods can be organized into two broad categories, based on how they fundamentally approach the evaluation of integrals. The first of these categories contains the various Monte Carlo methods. These methods, are extremely simple, and they are also incredibly robust. In addition, they are inherently multidimensional. Monte Carlo methods evaluate integrals by exploiting the statistical property given in equation (2.1), where x_{jR} are random numbers chosen from within the integration volume V.

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} f\left(x_{0R}, x_{1R}, ..., x_{mR}\right) = \int_V f\left(x_0, x_1, ..., x_m\right) dx_0 dx_1 ... dx_m \tag{2.1}$$

In effect, this means that one can evaluate a multidimensional simply by randomly sampling the integrand function within the integration domain and averaging the results. Additionally, there is no particular additional cost associated with the integration of higher-dimensional functions, beyond the generation of additional random numbers. Because of this multidimensionality, as well as the ease with which integrals can be computed for poorly behaved functions or over complicated integration domains, Monte Carlo methods are widely used throughout applied mathematics. Unfortunately, these benefits come with a cost. Monte Carlo methods, while robust, are not very accurate, and the numerical approximation to the integral converges to the exact answer at a rate proportional to $\sqrt{n}$. For applications where high accuracy is required, such as verification using manufactured solutions, this rate is simply too slow to be efficient.

The second primary category of numerical integration tools is quadrature methods. Unlike Monte Carlo methods, quadrature methods generally evaluate integrals by fitting the integrand function to a polynomial, and then computing the integral analytically. As a result of this different approach, quadrature schemes are sensitive to the smoothness of the underlying integrand function, and they are also generally (but not always) defined in terms of one-dimensional integration operators. As a result, evaluating multidimensional integrals with quadrature methods typically

requires a number of function evaluations that grows exponentially with the dimensionality of the integral. Despite these limitations, quadrature methods are extremely useful for many applications, because they are significantly more accurate for smooth functions. Provided that care is taken with discontinuities, quadrature methods are sufficient for evaluating the numerical integrals that are typical in the context of integral manufactured solutions.

2.1.2 Available Software Tools for Quadrature

Development of a new technique for numerical integration is a significant investment, and one that should not be undertaken lightly. There are many different computational tools for numerical integration that are both available and well established, including general mathematical libraries, specific packages for multidimensional integration, and proprietary software environments. These tools use a variety of algorithms and provide a wide array of capabilities which form a strong foundation for any project that relies on accurate numerical integration. Unfortunately, none of them provides the total capability required for accurate integration of multidimensional functions in the context of integral manufactured solutions. Specifically, for integrand functions containing arbitrary embedded surfaces where the integrand might jump arbitrarily, there was no tool available that was capable of evaluating multidimensional integrals of these functions to a high degree of accuracy and within a reasonable time frame, that was also conveniently incorporated with external C and Fortran code.

Through the course of scientific and mathematical research, software has been developed to solve many different mathematical problems under various constraints. Much of this software never leaves the immediate research environment in which it was created, and this software remains largely inaccessible to the wider scientific community, especially outside of the specific application area for which it was developed. Other software may find its way into commonly used libraries, such as `SLATEC` (named for Sandia, Los Alamos, Air Force Weapons Laboratory Technical Exchange Committee) or the `GSL` (the GNU Scientific Library). Software in these general-purpose libraries often becomes foundational code upon which more advanced codes and algorithms rely. Occasion-

ally, research software will be packaged and released independently by the groups responsible for its creation Many such codes are recorded in the NIST Guide to Available Mathematical Software [23]. Because packaging and maintaining research software for external users imposes a substantial burden on the code developers, such codes are often difficult to use and/or unmaintained. Finally, commercial providers of scientific computing environments such as Matlab and Mathematica will compile and maintain packages of useful mathematical and scientific functions in order to add value to their products. I will discuss each of these, and show how they have been insufficient for the intended application, namely fast, accurate evaluation of multidimensional integrals of discontinuous integrand functions in an academic research environment.

2.1.3 General Mathematical Libraries

The first category of mathematical software that is available for integration is that found in large, general-purpose software libraries. Of these, the archetypical example is `SLATEC`, a publicly available library which was developed jointly among the U. S. Department of Energy and the Air Force to provide a set of common mathematical routines. The `netlib` repository [24] houses the official reference implementation of `SLATEC`, including the official reference implementation for `BLAS` (Basic Linear Algebra Subprograms). In particular, `SLATEC` [25] also contains the `QUADPACK` [26] subpackage, which provides a variety of routines for numerical integration of one-dimensional functions.

Because the various `SLATEC` subpackages have been publicly available for so long, they have been implemented in a wide variety of languages and for many different environments, and the basic application program interface (API) forms a de facto standard for mathematical software. The `QUADPACK` subpackage exists largely unchanged in the GNU Scientific Library (GSL) [27], and also within the SciPy library [28]. `QUADPACK` provides integration routines based largely on adaptive quadrature. These include special algorithms for infinite intervals, weighted integrals, integrals with known internal singularities, and others. Many of these are based on Gauss-Kronrod quadrature and yield highly accurate results for most integrals.

Unfortunately, the QUADPACK routines require that the integrand be provided as a function $y = f(x)$, with no provision for additional arguments. This greatly complicates the integration of multivariate functions. This same difficulty is encountered in many variants of QUADPACK, including the GSL.

The SciPy Python library provides a mechanism to work around the univariate limitation in QUADPACK, which it uses QUADPACK as a back-end for the `scipy.integrate` package. This makes it much easier to evaluate integrals of multivariate functions in SciPy, rather than using QUADPACK directly. Unfortunately, the default routines for multidimensional integration in SciPy, `dblquad` and `tplquad`, do not allow for detailed control of the integration, and they are limited to two and three dimensions, respectively.

Commercial libraries are available that provide much more powerful mathematical tools than those contained in open-source repositories such as SLATEC and the GSL. Unfortunately, many of these are too expensive for the typical academic research budget to accommodate, and they typically come with licensing restrictions that can limit how widely they are used. As a result, high-end libraries are typically unsuitable for the purposes of academic research development, especially when there is no clear evidence that they will be better able to solve the specific problems associated with discontinuous integrand functions.

To summarize, most general-purpose mathematical libraries offer a set of numerical integration techniques that is similar to or derived from the QUADPACK reference library. As a result, they are quite capable of evaluating one-dimensional integrals of many different mathematical functions. Unfortunately, they often face substantial difficulties for multidimensional integrals. Commercial software libraries are much better in this regard, but they are typically unavailable to most researchers.

2.1.4 The Adaptive Genz-Malik Algorithm

Most deterministic routines for evaluation of multidimensional integrals are based on one-dimensional integration rules such as those discussed in section 2.1.3. However, the Genz-Malik

scheme is an important algorithm that attempts to reduce the number of required function evaluations by approaching the multidimensional integration problem directly. The principal innovations of the Genz-Malik scheme are a reduced number of points at which a function must be evaluated, and a multidimensional adaptation scheme that progressively refines the integration domain in a fully multidimensional way. Grid refinement is done by progressively halving the integration domain along coordinate axes until the required accuracy is achieved, resulting in a refinement scheme that is conceptually similar to the quadtree scheme shown in figure 2.1. Unfortunately, this does not allow Genz-Malik to fit discontinuous integrand with critical surfaces that do not align with coordinate axes.

The Genz-Malik algorithm has been incorporated into many mathematical libraries, including the NAG Scientific Library [29] and the Mathematica software environment. [30] It is also publicly available as part of the Cuba [31] and Cubature [32] libraries. Although extremely effective in many cases, the Genz-Malik algorithm is not able to precisely subdivide integration regions along general discontinuous surfaces, because of how it refines the integration by halving along coordinate axes.

2.1.5 Mathematical Software Environments

Perhaps the most widely known methods for numerical integration are those encountered in dedicated mathematical software environments, such as Matlab, and Mathematica. These environments are fully self-contained programs that are designed principally to enable rapid development of powerful mathematical software. Matlab is more principally concerned with numerical computations, while Mathematica is best known for its symbolic math capabilities, but there is a great deal of overlap between the various packages. In particular, both environments contain various algorithms for numerical integration of functions, and the widespread availability of both Matlab and Mathematica in academic environments makes them attractive use in scientific computing.

The Matlab environment is best known for its linear algebra capabilities, but it includes many additional features, including the `integrate` function for one-dimensional numerical integration. Unfortunately, it is difficult to determine exactly what methods the Matlab integration

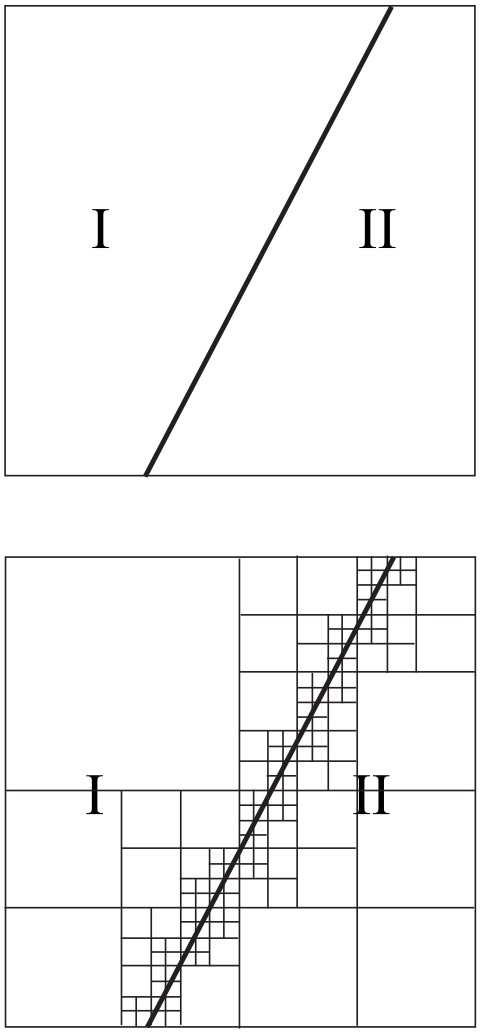


Figure 2.1: Quadtree-based Automatic Grid Refinement. An example multidimensional, discontinuous function can be given by two smooth regions I and II, separated by a critical surface where the function jumps discontinuously. One technique for evaluating such integrals is based on automatic grid refinement, as shown. The integral is evaluated, and an estimate of the error is obtained. If the error falls above a given threshold, the integration domain is subdivided along coordinate axes. This process is repeated until the estimated error falls below the threshold. Although accurate, this process is computationally expensive, because each subdivision of the integration domain must be treated as a separate integration problem.

routines are using. Based on the Matlab product documentation, it is clear that `integrate` is some kind of adaptive integrator, with the capacity for user-defined subdivisions much like that found in `QUADPACK`. Matlab's multidimensional routines also apparently include some kind of two-dimensional integration as an option, as well as the more common iterated integration. However, these multidimensional routines do not have any provision for user-defined subdivisions as the one-dimensional routines do, and so it is clear is that Matlab does not have the capability to naturally handle curvilinear discontinuities in functions. It is roughly comparable to the subset of `QUADPACK` routines that does not include weighting functions, except that it is proprietary, and it is more difficult to integrate with external software.

Wolfram's Mathematica software is an extremely powerful tool for mathematical computation. It is best known for its ability to handle complex symbolic mathematics and it also includes many numerical tools for when symbolic solutions cannot be found. For integration, this is the `NIntegrate` function.

Mathematica's symbolic features make it the most powerful tool available for the purposes of integrating discontinuous functions. Because the discontinuity is encoded within the function being evaluated, Mathematica can analyze that integrand directly, and process it accordingly in order to speed up computation. As is the case with Matlab, the exact mechanisms through which Mathematica does this are not publicized. A few bullet points of explanation are given in the Wolfram Language & System Documentation Center [30]: "NIntegrate uses symbolic preprocessing to resolve function symmetries, expand piecewise functions into cases, and decompose regions specified by inequalities into cells."; and the `MultiDimensional` integration method use the adaptive Genz-Malik algorithm. Access to the symbolic form of the integrand function and the ability to pre-process it to improve integration efficiency is an extremely powerful ability for numerical integration. Unfortunately, like Matlab, the proprietary nature of Mathematica makes it more difficult to integrate with external software.

2.1.6 Mapping Smooth Regions

There is one final method that must be mentioned, particularly within the present context of software verification. Grier et al. published a paper in August, 2014 [15] that specifically discusses methods for integration for discontinuous integrand functions. Their method relies on identifying the shock topology in a given integration domain, and mapping the smooth regions to regular geometric shapes. This method is exciting and promising, as it effectively fits the discontinuities that must be integrated, which leads to extremely accurate results. Unfortunately, the details of their approach have only been demonstrated for two-dimensional problems, and complex interactions among multiple discontinuous surfaces are not yet possible. Therefore, although extremely interesting, their method is not yet well-developed enough for the three- and four-dimensional integrals that I must consider.

Table 2.1: Comparison of Quadrature Libraries and Algorithms. In order to accurately evaluate integrals of discontinuous functions, a quadrature function must have several major characteristics: it must be able to evaluate multidimensional integrals; it must have some provision for handling points where the integrand may jump discontinuously; it must be freely available to the users who drive adoption; it must provide detailed access to the integration procedure. Detailed access means that the user can appropriately provide the integrator with available knowledge about the integrand function in order to speed evaluation and improve accuracy. Free availability of the library or algorithm is important because the intended application of this integrator, verification of codes, needs to be accessible to all potential users. Only Grier's algorithm provides all of these, but it has not been proven for 3- and 4-dimensional problems.

		Multi-dimensional	Discontinuity Handling	Freely Available	Detailed Integration Control
Libraries	QUADPACK		*	*	*
	GSL		*	*	*
	SciPy	*	*	*	
	Matlab	*	*		
	Mathematica	*	*		*
Algorithms	Genz-Malik	*		*	
	Grier	*	*	*	*

2.1.7 Conclusion

Of the many computational tools available for numerical evaluation of integrals, several offer significant promise for use with discontinuous integrand functions. One might attempt to build upon the foundation of basic, one-dimensional rules such as those found it `QUADPACK`, one might try to use one of the Genz-Malik implementations, or one could build the capability around the use of proprietary software such as Mathematica. The correct choice will naturally be application-dependent. For my purposes, the intended application is to enable fast, accurate evaluation of multidimensional integrals of discontinuous functions, so that one might implement an integral form of the method of manufactured solutions for software verification. In order to allow such verification software to be widely used, the tools required must be available under a permissive license and in a widely used environment. As a result, I have chosen to build a new integration tool based on the `QUADPACK` library which underlies the `scipy.integrate` package.

2.2 High-Performance, High-Accuracy Evaluation of Multidimensional Integrals of Discontinuous Functions

Despite these challenges, fast, accurate evaluation of multidimensional integrals of discontinuous functions is absolutely required in order to apply the method of manufactured solutions to integral, shock-capturing codes. Therefore, I have designed, developed, implemented, and tested an n-dimensional integration algorithm that is both fast and highly accurate, even when integrating discontinuous functions.

2.2.1 Conceptual Overview

It is helpful to understand the concepts behind this n-dimensional integration algorithm, before discussing the details. The design is actually quite simple, and requires an understanding of only three important concepts: iterative evaluation of multidimensional integrals with one-dimensional tools; the one-dimensional integration operator; and the effects the one-dimensional integration operator has on critical hypersurfaces of discontinuous functions.

2.2.1.1 Iterative Integration

Iterative integration itself is quite simple. It is simply the concept that, in many circumstances, one may evaluate a multiple integral as a series of one-dimensional integrals, as in equation (2.2).

$$\iint f(x,y) \Rightarrow \int \left(\int f(x,y)\, dx \right) dy = \int \left(\int f(x,y)\, dy \right) dx \tag{2.2}$$

As discussed in section 2.1.2, there are ways to numerically evaluate multiple integrals without using iterated integrals, and these can provide better computational performance. In the presence of discontinuous hypersurfaces, however, iterative integration is advantageous because of the greatly simplified topological problem that must be solved.

Consider the integral of a one-dimensional, discontinuous function such as that given in equation (2.3).

$$\int_{a<x_0}^{b>x_0} f(x)\, dx \qquad\qquad f(x) = \begin{cases} f_1(x) & : \quad x < x_0 \\ f_2(x) & : \quad else \end{cases} \tag{2.3}$$

Such an integral may be evaluated by dividing it into two smooth pieces, as in equation (2.4).

$$\int_{a<x_0}^{b>x_0} f(x)\, dx = \int_a^{x_0} f(x)\, dx + \int_{x_0}^{b} f(x)\, dx \tag{2.4}$$

That is, the integral in equation (2.3) has a critical point at $x = x_0$, and the integral may be subdivided at that critical point to yield two smooth integrals. The generalization to more than one critical point is obvious.

This process is simple enough, but it becomes more complex in higher-dimensional problems. Consider a similar two-dimensional integral, as in equation (2.5), and shown in figure 2.1.

$$\int f(x,y)\, dA \qquad\qquad f(x,y) = \begin{cases} f_1(x,y) & : \quad g(x,y) < 0 \\ f_2(x,y) & : \quad else \end{cases} \tag{2.5}$$

Conceptually, nothing has changed. One need simply integrate the function over the two smooth regions I and II, and then sum the result. Rather than a critical point, however, the discontinuity is represented as a critical line. No restrictions are placed on this critical line, so it may bend,

curve, and self-intersect. This more complex behavior dramatically increases the computational difficulty associated with subdividing the integral. The presence of additional critical lines only complicates the matter further. In higher dimensions, the result is a topological nightmare, as has been demonstrated by the experiences of the CFD community with shock-fitting schemes.

Adaptive integration schemes attempt to resolve these higher-dimensional discontinuities by progressively subdividing the integration domain until the divisions with large error are small enough to not substantially affect the overall accuracy of the integral. An example quadtree approach is shown in figure 2.1, but adaptive schemes require an excessive number of divisions in order to return the level of accuracy required for use with manufactured solutions, which severely limits computational performance.

As mentioned, the one-dimensional case is much simpler. A one-dimensional integral is subdivided at one or more zero-dimensional critical points, corresponding to critical hypersurfaces. Provided with a means to identify these points, integration of discontinuous functions becomes trivial. Since iterative integration allows one to reduce multiple integrals to a series of one-dimensional problems in most cases of engineering interest, I use iterative integration to avoid the higher-dimensional difficulties entirely.

2.2.1.2 The One-dimensional Integration Operator

Having chosen to use iterated, one-dimensional integration as the basis of my algorithm for multidimensional integration of discontinuous functions, it is important to have an understanding of the one-dimensional integration operator and its effects on functions.

The one dimensional definite integration operator is a linear operator $\mathcal{L}_{\S_{\Updownarrow}}$ that maps n-dimensional functions f of variables $x_0 ... x_n$ to $(n-1)$-dimensional functions of variables $y_0 ... y_{n-1}$ by integrating the function along the variable x_m between the bounds x_{mi} and x_{mf}, as in equation (2.6).

$$\mathcal{L}_{x_m}\left(f, x_{mi}, x_{mf}\right) \equiv \int_{x_{mi}}^{x_{mf}} f\left(x_0, ... x_n\right) dx_m \tag{2.6}$$

This mapping is the key to understanding how to quickly and accurately evaluate iterated

integrals of discontinuous functions. Each application of the one-dimensional integration operator results in a new function of one less variable. This new function can be considered a separate function in its own right, connected to the original function only by the mapping of the integration operator. Specifically, this new function will have its own continuity properties, which may be considered separately from the original function.

For example, consider the function given by equation (2.7).

$$F_0(x_0, x_1, x_2) \equiv \begin{cases} 1 & : \quad x_0^2 + x_1^2 - x_2 < 0 \\ 0 & : \quad else \end{cases} \tag{2.7}$$

Application of the one-dimensional integration operator $\mathcal{L}_{x_0}$ with the integration range $[-1, 1]$ maps F_0 to the new function F_1, given in equation (2.8).

$$F_1(x_1, x_2) \equiv \mathcal{L}_{x_0}(F_0, -1, 1) = \int_{-1}^{1} F_0(x_0, x_1, x_2)\, dx_0 \tag{2.8}$$

Application of a different operator $\mathcal{L}_{x_2}$ with the integration range $[0, 2]$ to this same function results in a different mapping, given in equation (2.9).

$$F_{12}(x, y) \equiv \mathcal{L}_{x_2}(F_0, 0, 2) = \int_{0}^{2} F_0(x_0, x_1, x_2)\, dx_2 \tag{2.9}$$

The results of one mapping can be operated on again to produce further new functions, with one example given in equation (2.10).

$$F_2(z) \equiv \mathcal{L}_{x_1}(F_1, -1, 1) = \int_{-1}^{1} F_1(x_1, x_2)\, dx_1 \tag{2.10}$$

Figure 2.2 shows each of these functions graphically, in the integration domain given by $\{x_0, x_1, x_2\} = \{[-1, 1], [-1, 1], [0, 2]\}$. The three-dimensional function F_0 contains one two-dimensional, discontinuous surface, f_{00}, given by equation (2.11).

$$f_{00} : 0 = x_0^2 + x_1^2 - x_2 \tag{2.11}$$

The two-dimensional function F_1 contains two one-dimensional, non-smooth curves, f_{10} and

f_{11}, given by equation (2.12).

$$f_{10} : 0 = x_1^2 - x_2 \tag{2.12}$$
$$f_{11} : 0 = x_1^2 - x_2 + 1$$

Finally, F_2 contains two zero-dimensional, non-smooth points, given by equation (2.13).

$$f_{20} : 0 = x_2 \tag{2.13}$$
$$f_{21} : 0 = x_2 + 1$$

The two-dimensional function F_{12}, on the other hand, contains no critical curves at all.

The objective of this exercise is to highlight the fact that the integration operator maps n-dimensional functions to $(n-1)$-dimensional output functions. These output functions can themselves be analyzed and integrated, and may be discontinuous, non-smooth, or neither, depending on the form of the input function and the particular choice of the integration operator.

2.2.1.3 Integration and Critical Hypersurfaces

As has been discussed, the key to accurately evaluating integrals of discontinuous functions is to subdivide the integration domain at the position of the discontinuity. This same procedure can be applied to multidimensional integrals by making use of iterated integration routines, which apply several integration operators to the integrand in sequence. Each of these operators maps their input function to a new, lower-dimensional, output function, with its own properties and discontinuities. Therefore, one must be able to accurately predict how discontinuities will be affected by the mapping of an integration operator, and the functional form of any discontinuities in the resulting function. That is, given a function with a set of critical surfaces and a particular integration operator, one must be able to accurately predict the critical surfaces of the function that results from apply that integration operator to the initial function.

If we consider again the example given in figure 2.2, one must be able to derive , given the initial function F_0, the initial discontinuity F_{00}, and the integration volume V, and the sequence of

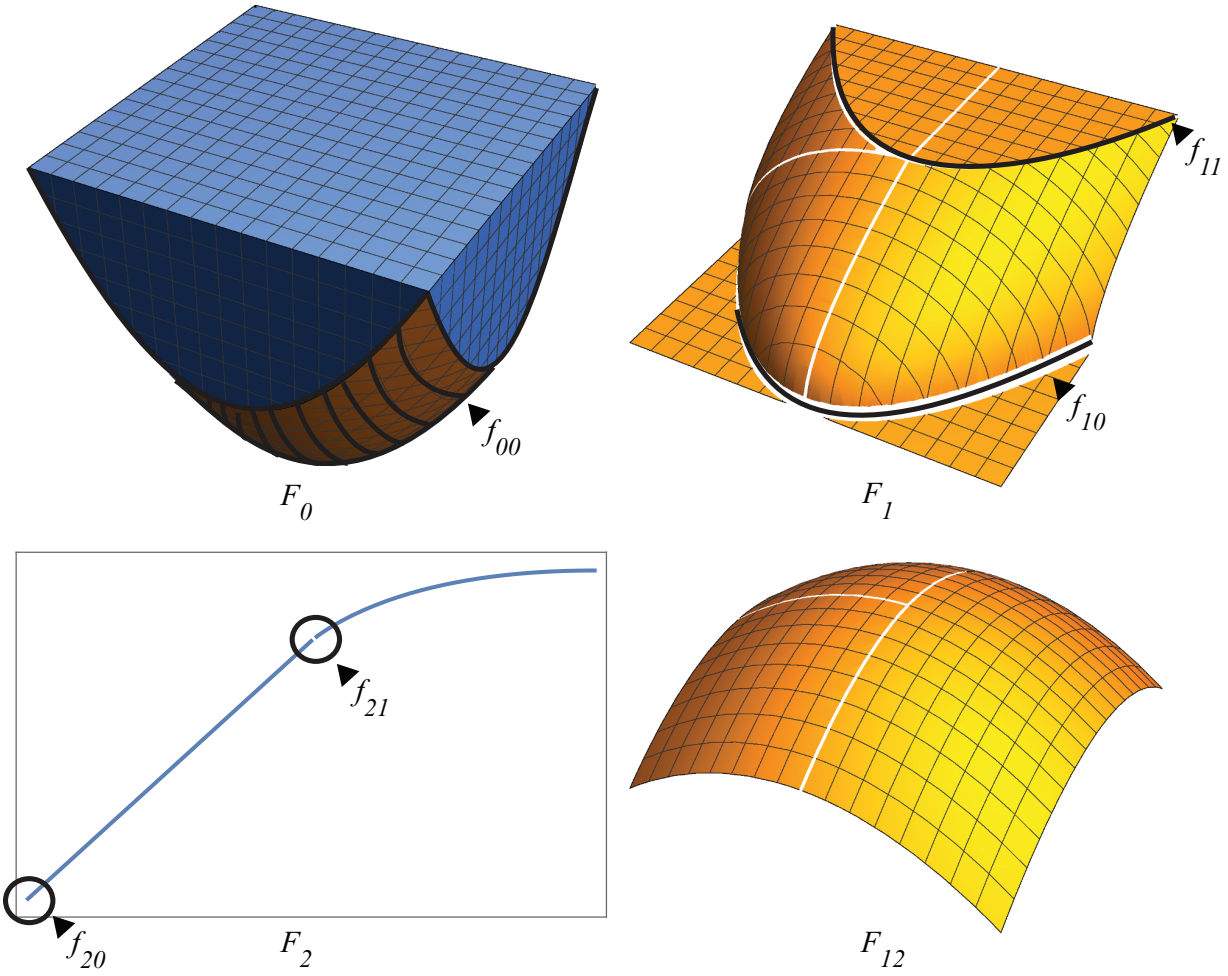


Figure 2.2: Iterative Integration of a Three-dimensional Integrand. A graphical representation can provide insight to the process of iterative integration with a discontinuous integrand function. In this problem, one must essentially compute the volume of the shape bounded by the function F_0, which consists of the intersection between a cubic integration domain and a paraboloid, which is shown with lines. Integrating this function crosswise results in the two-dimensional function F_1. Integrating this crosswise again results in the one dimensional function F_2. Both F_1 and F_2 have critical surfaces where the derivative of the function changes discontinuously, and these are marked and labeled. Order of integration also matters; a different choice of the initial integration of F_0 results in F_{12}, which has no critical surfaces at all.

integration operators $\mathcal{L}_{x_m}$, the critical surfaces of the intermediate integration functions f_{10}, f_{11}, f_{20}, and f_{21}.

As has been seen, the function that results from the application of the one-dimensional integration operator to a discontinuous function may include zero or more critical surfaces. Because the integration operator has a smoothing effect on functions, these critical surfaces are often non-smooth, rather than discontinuous, and a careful choice of integration operator may result in an output that has no critical surfaces at all.

Critical surfaces in the output of a one-dimensional integration operator arise when the smoothing effect of the integration is insufficient to fully smooth the underlying function. This is generally caused by one of two mechanisms. First, a discontinuous surface may be normal to the direction of integration for the particular operator being applied. If a function has critical surfaces given by the functions $f_{0i}(x_0, ..., x_n)$, and the integration operator $\mathcal{L}_{x_m}$ is applied to this function, then the resulting function will have one or more critical surfaces corresponding to the zeros of the the directional derivative. These zeros may then be evaluated in order to yield the values of the integration variable x_m that correspond to these zeros, as shown in equation (2.14).

$$\frac{\partial f_{0i}(x_0, ...x_n)}{\partial x_m} = 0 \Rightarrow x_m^{**} \tag{2.14}$$

These values may then be substituted into the initial critical surface in order to yield the new surfaces for the output function, as in equation (2.15).

$$f_{1j}(x_0, ..., x_{m-1}, x_{m+1}, ..., x_n) = f_{0i}(x_0, ..., x_m^{**}, ..., x_n) \tag{2.15}$$

Second, one surface may intersect with the boundary of the integration domain V. In particular, if an integration operator $\mathcal{L}_{x_m}$ has functional integration boundaries $x_{m\,min}$ and $x_{m\,max}$, then these boundaries can themselves be described by functions of the form $f(x_0, ...x_n) = 0$, and critical surfaces in the output of the integration operator may be derived from the intersections between critical surfaces and these boundaries.

New critical surfaces arising from either of these mechanisms must be accounted for in order to accurately evaluate multidimensional integrals of discontinuous functions. In most cases,

this this is done by solving the boundary surface equations f_{ij} for the integration variable x_m corresponding to a particular integration operator $\mathcal{L}_{x_m}$ in order to yield a set of critical values $x_m^*(x_0, ..., x_{m-1}, x_{m+1}, ..., x_n)$ at which the integration domain should be subdivided. This process is applied recursively throughout the iterated integration process.

It should be reemphasized that the number and type of critical surfaces generated by an integration operator is dependent on the particular operator chosen (i.e., the order of integration), as well as the integration domain. The careful choice of integration order can sometimes even eliminate critical surfaces entirely from the bulk of the iterated integration levels, which is both simpler and computationally beneficial.

The ability to map a function's discontinuous surfaces through a one-dimensional integration operator is crucial for the accurate evaluation of multidimensional integration of discontinuous functions. It allows the integration domain to be appropriately subdivided at all levels of iterated integration, effectively fitting the discontinuous surfaces exactly, which effectively reduces the problem to a collection of smooth integrals. As a result, the accuracy of the overall integration is equivalent to that obtained for smooth functions.

2.2.1.4 Conceptual Overview–Conclusion

In summary, I have devised a new method for accurate evaluation of multidimensional integrals of discontinuous functions. This method uses iterated one-dimensional integration operators for multidimensional integration, which greatly simplifies the treatment of discontinuities. These operators map multidimensional functions to lower-dimensional functions, corresponding to integration over one of the variables. My key contribution to the state-of-the-art is a method for determining, based on the information available prior to the integration, what the continuity properties of the resultant function will be. This allows the entire multidimensional integral to be subdivided in such a way as to perfectly fit the discontinuous surface, thus allowing accuracy levels that are equivalent to those that would be expected for continuous integrand functions.

Having explained the concepts behind my new method for accurate evaluation of multidi-

mensional integrals of discontinuous functions, I will discuss in detail the actual computational implementation of this method, as used in `BACL-IMMS`, a software package for code verification, which will be discussed in greater detail in chapter 3. This can be broken into two major sections. First, I will discuss the routine I developed for iterative integration of n-dimensional integrals. Second, I will discuss the high-level interface that automatically transforms an integrand function, a list of discontinuous surfaces, and an integration domain into the appropriate form for numerical integration. These tools have proven themselves invaluable in enabling a simple implementation of my research into methods for scientific software verification.

2.2.2 `nquad`: Adaptive, Iterative, Controllable Integration in Python

In order to implement a multidimensional integration routine based on the concepts outlined in section 2.2.1, one must have access to an n-dimensional, iterative integration tool that allows for user-defined subdivision of the integration domain at all levels of the iteration. Unfortunately, no such tools were readily available, so I developed `nquad`, an n-dimensional, recursive integration routine providing the appropriate level of control.

There are many libraries available for numerical evaluation of integrals. Some of these are simple left-, right-, and midpoint-rule Riemann sums, while others are much more sophisticated. Many of the best libraries are proprietary, and many are difficult to incorporate into new software projects. The SciPy [33] library provided the nearest match to the specific requirements of this project, wrapping the well-known and well-established `QUADPACK` [24] library for use in Python programs. `QUADPACK` is a Fortran library for numerical evaluation of one-dimensional integrals. It provides various routines for both fixed-order and adaptive integration of univariate functions, as well as routines that allow for user-specified critical points and weight functions. The SciPy Python wrapper enhances these capabilities with a unified interface to most of the routines, as well as the ability to handle multivariate function signatures. These improvements greatly simplified the process of evaluating multidimensional integrals using iterative integration, but the library lacked the level of control required in order to apply the method for accurately integrating discontinuous

functions described in section 2.2.1.3. The development of `nquad` was undertaken in order to address these shortcomings.

Making this tool publicly available was a major focus, and a concerted effort was made to make it useful in a wide range of applications. `nquad` was incorporated into the SciPy library in version 0.13, and further performance optimizations were included in version 0.15.

2.2.2.1 `nquad` Interface and Algorithm

`Nquad` was developed as a Python wrapper to `scipy.integrate.quad`, following the model of the already extant `scipy.integrate.dblquad` function. The internal structure is illustrated in figure 2.3.

`nquad` has a simple user interface that nonetheless allows for fine control of the integration processes if desired. It is called as a function of the form:

```
nquad(f,ranges,args,opts)
```

where `f` is the function to be integrated, `ranges` is a list of the ranges of integration, `args` is any additional arguments required by `f` beyond the integration variables, and `opts` contains the integration options corresponding to the underlying levels of integration.

This interface is a simple wrapper to the underlying machinery. It processes the arguments it receives into the appropriate form, and initializes an `_NQuad` object. This object provides the `integrate` method, which is called recursively to evaluate the multidimensional integral.

Using the information contained in the `_NQuad` state, the `integrate` method determines whether or not it is a call corresponding to the innermost level of integration. If it is not, then it defines a temporary integrand function, which is itself a call to the `integrate` method. If it does correspond to the innermost integration level, then the temporary integrand function becomes an alias for the integrand function provided by the initial `nquad` call and `_NQuad` initialization.

Once the integrand function has been defined, it is passed on to the one-dimensional integration routine, along with the appropriate integration range and options. If the integrand is a call to `_NQuad.integrate`, then an evaluation of the integrand will result in another call to `quad`, and so

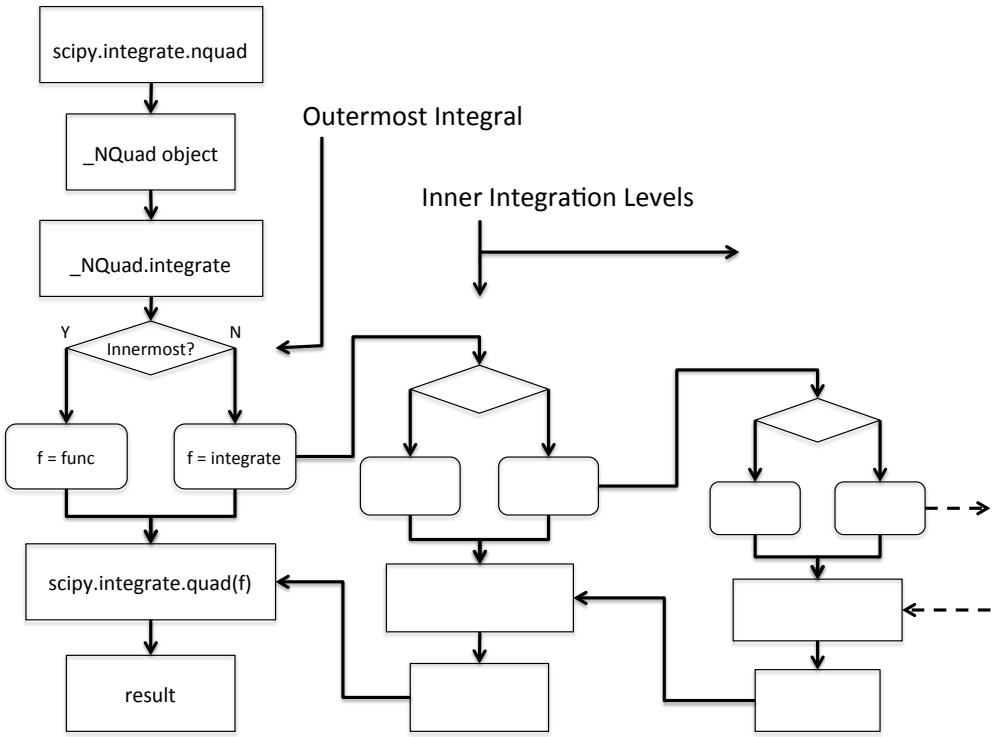


Figure 2.3: `nquad` Flow Diagram. `nquad`, the SciPy routine for multidimensional quadrature works by recursively applying the one-dimensional quadrature rules available in `scipy.integrate.quad`. In order to make this process as simple as possible for the user, it is presented as a single function: `scipy.integrate.nquad`. This interface function creates an `_NQuad` object, which is initialized with the integrand function, integration ranges, and any extra arguments or integration options. Integration ranges and options are distributed to the appropriate level of integration, which allows `scipy.integrate.quad` to be called with different integration options at each level, a capability that was missing from previous SciPy multidimensional integrators.

on until the innermost level of integration is reached.

`scipy.integrate.quad` is used as the one-dimensional integration routine. It is a flexible interface to the `QUADPACK` library and contains many different algorithms, the choice among which is determined by the options it receives. The two most important algorithms for the purposes of the present discussion are the default adaptive Gauss-Kronrod quadrature method, and the variant that allows for externally supplied information on internal discontinuities, singularities, and other difficulties of the integrand function [24].

As described in the `QUADPACK` documentation and code [24], the default integration method for `scipy.integrate.quad` is based on a 22-point integration rule. The results of applying this rule are then compared with those of a 15-point rule whose points are co-located with those of the 22-point rule, and the difference between them is evaluated. If this error is below a given (user-specified) tolerance, then the result is returned. If not, then the integration domain is bisected, and the integral is reevaluated for each segment. This process continues until the required tolerance is reached.

When the user provides the locations of known critical points to `scipy.integrate.quad` through the `points` argument, the integration domain is automatically subdivided at those critical points before any adaptive subdivision takes place, thus controlling the difficulties associated with singularities and removing the effects of discontinuities.

It is important to note that both integration ranges and options are defined as functions in the `nquad` interface. That is, the integration ranges and critical points corresponding the the inner loops may be functions of any outer integration variables. Specifically, if `nquad` is used to integrate a function $f\left(x_0, ..., x_n, t_0, ..., t_m\right)$ over some integration domain in $x_0, ..., x_n$, with additional parameters $t_0, ..., t_m$ provided via the `args` argument, then both the `ranges` argument and the `opts` argument may contain a list of functions $f_0\left(x_1, ..., x_n, t_0, ..., t_m\right), f_1\left(x_2, ..., x_n, t_0, ..., t_m\right), ...$ that return the appropriately formed `range` and `opts` structures for a given level of integration. This is a critical feature, as it allows integration of domains that are not hyper-cubic, and it also allows `points` to be specified such that they lie on some curved hypersurface.

2.2.2.2 Optimizing the innermost integration level

The nature of `nquad` is such that the integrand function must be evaluated many times by the underlying Fortran library. In their simplest form, `nquad` and `quad` provide these integrands as Python functions with the signature `f(x0,...,xn,t0,...,tm)`, where `x0,...,xn` are the integration variables, and `t0,...,tm` are any additional arguments which the function requires. This form is preferred in most circumstances, as it interfaces very well with the overall Python environment. However, defining the integrand as a Python function entails performance sacrifices, both due to the relatively slow evaluation of complex Python functions and the repeated callbacks to Python that must be made from within the `QUADPACK` integration library. Since these performance sacrifices can be severe for some problems, it is beneficial to provide the user with options for performance optimization.

The most straight-forward way to improve performance of the overall `nquad` integration is by using a compiled C function for the integrand. Such a function can be optimized for performance at compile-time, and also accessed natively by the `QUADPACK` library, thus avoiding callbacks. This would affect function calls at the innermost level of integration, where the bulk of the function evaluations are made.

The principal difficulty with this approach is the structure of the `QUADPACK` library, which is written for univariate functions only. The SciPy library worked around this limitation for Python functions, but provided no interface for doing the same with C functions. Brian Newsom, a student funded through the Discovery Learning Apprenticeship program at CU-Boulder, rewrote the interface code connecting `scipy.integrate.quad` with `QUADPACK` during the 2013-2014 school year, to support this functionality. As a result of his work, it is now possible to improve computational performance by defining `f` as a C function with the signature `f(n+m,[x0,...,xn,t0,...,tm])`. This function is then compiled, loaded into Python using the `ctypes` module from the Python standard library, and the resulting object can be passed to `quad` or any of the functions that wrap it, including `nquad`. The performance improvements from this optimization are dependent on the

complexity of the integrand function. For simple integrands, 2x speedups are common. For complex integrands that benefit from aggressive compile-time optimization, 10x speedups are more typical.

2.2.3 Automatic Evaluation of Multidimensional Integrals of Discontinuous Functions using `scipy.integrate.nquad`

An iterative integration tool that offers fine-level control such as `nquad` is an important enabling technology for high-performance, highly accurate evaluation of multidimensional integrals of discontinuous functions, but the process of preparing an integral for evaluation in `nquad` can be complex in practice. One must prepare the functional integrand, derive all of the necessary critical surfaces, and convert these to the appropriate computational forms. A mistake in any one of these steps can result in inefficient, inaccurate, or even erroneous results. Therefore, I have developed a software tool to automate the process of preparing discontinuous integrals for evaluation by `nquad`.

Automating this process requires three separate steps: mapping critical surfaces through integration operators; organizing the resulting surfaces as required for iterative integration; and generating integrand and option functions for use in `nquad`. As a result, the tool I developed must include capabilities for symbolic mathematics, general computer programming, and code generation. The Python programming language is an excellent environment in which to develop this kind of software, and I have used the Sympy package to provide both symbolic math and code generation capabilities.

The usage of this software tool is as follows:

(1) Define the integrand function symbolically, as a Sympy expression. Theoretically, discontinuous functions should be definable using `sympy.Piecewise` objects, but I have found greater success using Heaviside step functions from `sympy.special.delta_functions.Heaviside` in current versions of Sympy (0.7.6).

(2) Record the critical surface. If an integrand function has critical surfaces defined by expressions of the form $f(x_0, ..., x_n) = 0$, then one must provide a list of Sympy expressions

$f(x_0, ..., x_n)$.

(3) Provide the integration domain in a symbolic form. Currently, only hypercubic integration domains are supported, and these must be given as a list of structures in the form: `[[var,varmin,varmax],...]` where `var` is the Sympy `Symbol` object used as the integration variable in the integrand function, and `varmin` and `varmax` are the bounds of integration.

Given these three pieces of input data, the software computes all of the necessary critical surfaces, arranges them in the appropriate order for iterated integration, and puts them into a computational form suitable for use by general Python functions such as `nquad`. It does this through the action of three different Python objects, as shown in figure 2.4: `Discontinuity` objects, which map the critical surfaces and generate code, `Discontinuities` objects, which aggregate collections of `Discontinuity` objects into the form required for iterated integration, and `IntegrableFunction` objects, which provide a convenient user-interface to the integration process.

2.2.3.1 Discontinuity Objects

`Discontinuity` objects are primarily responsible for two things: converting symbolic representations of critical surfaces into functions that can be used by `nquad`; and mapping critical surfaces through integration. `Discontinuity` objects are initialized with the Sympy expression for a critical surface, along with the integration bounds. Once initialized, they provide a `__call__` method, which returns a list of critical points for use by `nquad`.

It is important to recall at this time that critical surfaces $\mathcal{S}$ are typically defined by functions of several variables, as in equation (2.16):

$$S : f(x_0, ...x_n) = 0 \tag{2.16}$$

If one wishes to know x_m values for this critical surface, then one may solve equation (2.16) for these critical values, as in equation (2.17).

$$x_m^*(x_0, ..., x_{m-1}, x_{m+1}, ..., x_n) \equiv \texttt{sympy.Solve}(f(x_0, ..., x_n), x_m) \tag{2.17}$$

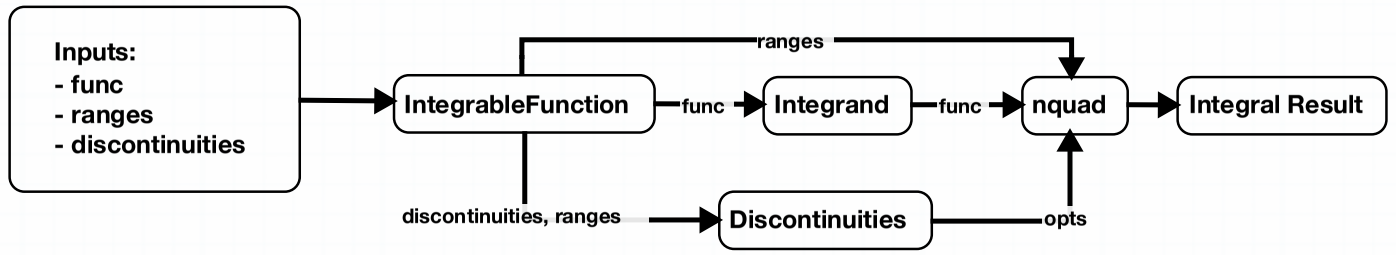


Figure 2.4: Automatic Evaluation of Discontinuous Integrals using `nquad`. The use of an integration tool that provides detailed control, such as `nquad`, makes it possible to accurately evaluate multidimensional integrals of discontinuous functions, but it does not necessarily make it easy. Much of the work required in order to use nquad can be automated using symbolic algebra packages, so software was developed in order to do just that. The combined action of an `IntegrableFunction` object, an `Integrand` object, and a `Discontinuities` object results in a simple procedure which converts an integrand function, integration ranges, and critical surfaces, into a highly accurate result for the integral.

These critical points will, of necessity, depend on the remaining $n-1$ variables. As a result, the `__call__` method for a given `Discontinuity` object must use these as arguments in order to return the list of critical points x_m^*.

The `__call__` function is actually itself an abstraction. Sympy provides several methods for evaluating symbolic expressions, and these form the various back-ends available to the `__call__` function. The simplest approach is to use the `subs` method available to Sympy objects, which substitutes numerical values for the `sympy.Symbol` objects in an expression. This approach is the simplest and the most flexible, because it maintains the underlying structure of a Sympy expression. Unfortunately, this also means that using `subs` is computationally expensive. For any function that must be evaluated many times, the cost of evaluating symbolic expressions using `subs` is too high to be practical.

A better approach is to take advantage of the various code-generation capabilities Sympy includes. There are two main options for this. One can use `sympy.utilities.lambdify.lambdify` to generate Python functions from Sympy expressions, or else one can use the code-generation routines in `sympy.utilities.codegen.codegen` to generate actual C code from those same expressions. Each approach has advantages and disadvantages. Python functions generated by `lambdify` are inherently somewhat slower to execute, but C code generated by `codegen` must be compiled and linked, which generally leads to higher initialization costs and a more fragile build environment. I will discuss the use of `codegen` more thoroughly in the context of integrand functions in section 2.2.3.3, but `lambdify` is the preferred method for use with `Discontinuity` objects.

Once initialization of the `__call__` method is complete for a given critical surface, that surface must be mapped through the integration operators. `Discontinuity` objects, like `nquad` itself, use the convention that the order of integration can be read from the specification of the integration domain in order from first to last. That is, if the integration domain is specified as

```
[[x0,x0min,x0max],[x1,x1min,x1max],...,[xn,xnmin,xnmax]],
```

then the iterated integration will be carried out as in equation (2.18).

$$\mathcal{L}_{x_n}...\mathcal{L}_{x_1}\mathcal{L}_{x_0}f\left(x_0,...,x_n\right) \tag{2.18}$$

Therefore, initialization of the `Discontinuity` object with full integration domain and the initial critical surface generates a `__call__` function corresponding to the $\mathcal{L}_{x_0}$ operator, which takes the values for x_1,...,x_n as arguments. This critical surface is then mapped through the $\mathcal{L}_{x_0}$ operator to create some number of new `Discontinuity` objects corresponding to the $\mathcal{L}_{x_1}$ operator. This is done by creating and storing additional `Discontinuity` objects as `Discontinuity.children`.

`Discontinuity.children` are generated as discussed in section 2.2.1.3. First, the symbolic discontinuity is differentiated with respect to the integration variable (x_0, as discussed here), and the zeros of the resulting expression are computed, yielding some number of solutions defined by sets of arguments ($x_0...x_n$). The various x_0 values from these solutions are then substituted into the initial critical surface function $f\left(x_0,...x_n\right)$, and the resulting functions are themselves used to initialize new `Discontinuity` objects.

Second, the intersections between the the x_0 integration bounds and the initial critical surface are computed by substituting x_{0min} and x_{0max} into $f\left(x_0,...,x_n\right)$ and using these to initialize any additional `Discontinuity` objects.

The resulting list of `Discontinuity` objects contains all of the critical surfaces that result from mapping the initial critical surface through the $\mathcal{L}_{x_0}$ operator. Each of these will likewise have `children` objects, resulting in a linked list structure where each critical surface in the integrand function contains pointers to each critical surface that derives from it, as shown in figure 2.5.

2.2.3.2 The Discontinuities Object

As discussed in section 2.2.3.1 and shown in figure 2.5, critical surfaces that arise from the action of integration operators are organized based on the root critical surface from which they are derived. This organizational scheme is quite natural and well-suited for deriving these surfaces, but it is inappropriate for numerical integration using `nquad`. As discussed in section 2.2.2.1, `nquad`

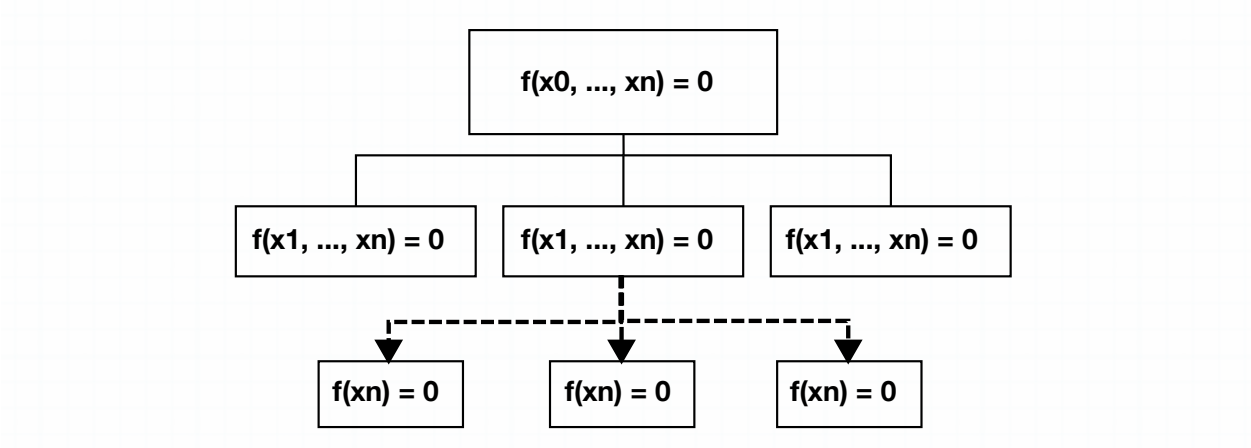


Figure 2.5: Discontinuities and their Children. When a one-dimensional integration operator is applied to a function containing a critical surface such as a discontinuity, that surface may map to one or more new critical surfaces in the resulting function. Therefore, when a `Discontinuity` object is created, it must likewise account for any new critical surfaces it may create by spawning additional children `Discontinuity` objects. This results in a simple tree-like structure, with the initial `Discontinuity` at the root, and derived `Discontinuity` objects branching out from it.

expects to receive the locations of critical surfaces as a list of functions $f_0(x_1, ..., x_n, t_0, ..., t_m)$, $f_1(x_2, ..., x_n, t_0, ..., t_m), ...$ that return the appropriate critical values for each integration operator. Essentially, one must step through all of the `Discontinuity` objects and their `children`, and then one must sort these according to the specific integration operator to which they pertain, and provide an `nquad`-compatible interface. This is the task of the `Discontinuities` object, shown in figure 2.6.

`Discontinuities` objects are initialized in much the same way as `Discontinuity` objects. The principal difference between the two is that a `Discontinuities` object is initialized with a list of several critical surfaces.

Each element of this list of surfaces is used to initialize a root `Discontinuity` object, with all of the corresponding `children`. The resulting tree structure is sorted by integration level using a breadth-first tree traversal, and the resulting data structure is then purged of duplicate critical surfaces.

Finally, a Python functional interface is generated for each level of integration. This interface provides access to a function with the appropriate function signature, that returns a list of critical points for a given level of integration.

In summary, a `Discontinuities` object accepts a list of several symbolic critical surfaces defined by $f(x_0, ..., x_n) = 0$ and given as Sympy expressions f, and it converts these into a list of n functions $x_0^*(x_1, ..., x_n), ..., x_n()$ that each returns a list of critical points at which the integrand function being mapped by the appropriate integration operator may be ill-behaved. This list of functions is essential for accurate, fast evaluation of multidimensional integrals of discontinuous functions, but one further step is required in order to make this information available to the `nquad` integration routine.

2.2.3.3 IntegrableFunction Objects

The discussion in section 2.2.3.1 and in section 2.2.3.2 has explained the mechanisms by which symbolic critical surface functions may be converted into lists of critical points for use in `scipy.integrate.nquad`. The final step is to define a method for numerically integrating a discon-

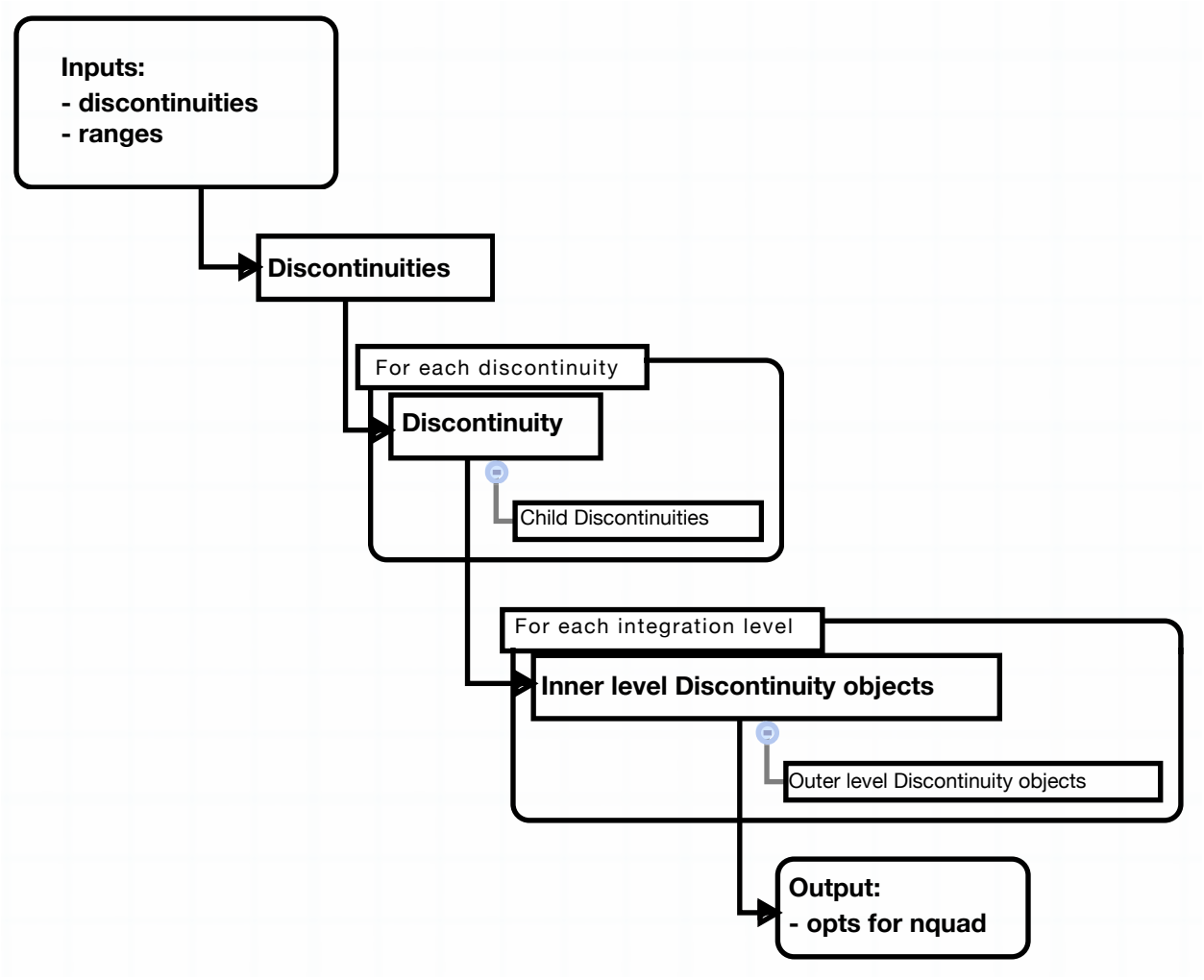


Figure 2.6: `Discontinuities` Object Flow Diagram. A single integral may involve many `Discontinuity` objects. Unfortunately, the tree structure in which these naturally occur is not amenable to integration using `nquad`. As a result, the various `Discontinuity` objects must be created, and then they must be organized into levels, corresponding to the various levels of iterative integration. This process is handled by a `Discontinuities` object.

tinuous function quickly and accurately, using the tools developed here. The `IntegrableFunction` object does exactly that.

Although `Discontinuity` and `Discontinuities` objects are fairly well-developed and can be used independently, the average user is concerned only with the results and performance of the integration. Additionally, hiding the intricacies of integration in this way is important, because it greatly reduces the incidence of mistakes in preparation of the integrals. The `IntegrableFunction` object is designed to take the symbolic representations of the integrand function, the integration domain, and the discontinuous surfaces, and process this information such that the integral can be computed quickly and accurately with a single function call.

The first step is to convert the symbolic integrand function to a form that can be evaluated efficiently by `nquad`. This is a very similar problem to that encountered for critical surfaces in section 2.2.3.1. The symbolic integrand is used to initialize an `Integrand` object, which uses Sympy's `lambdify` and `codegen` tools to generate Python and C functions that evaluate the integrand. The efficiency tradeoffs can be different for the integrand function, as the number of function evaluations scales as $\mathcal{O}^n$, where n is the dimensionality of the integral. In contrast, the number of evaluations of `Discontinuity` functions scales only as $\mathcal{O}^{n-1}$. Therefore, in some instances, it can be preferable to use the C functions here, especially with complicated integrand functions.

The second step is to create a `Discontinuities` object from the symbolic representations of the critical surfaces, as discussed in section 2.2.3.2.

The third, and final step is to use the `Discontinuities` object to create a list of callable `opts` structures for `nquad` that provides the appropriate critical values of the integration. This is done using an `OptionsDict` object, which is essentially a wrapper for the `Discontinuities` function that returns critical values. Additional options can be easily included in the future as they are needed.

To summarize, it may be helpful to show an example. Consider the critical surface defined by equation (2.19).

$$f(x_0, x_1) \equiv x_0^2 + x_1^2 - 1 \tag{2.19}$$

One may define a simple, discontinuous integrand function based on this surface, as in equation (2.20).

$$F\left(x_0, x_1\right) \equiv H\left(f\left(x_0, x_1\right)\right) \tag{2.20}$$

The integrable of this function over the computational volume $[-0.25, 1.25] \times [-0.25, 1.25]$ may be defined, as in equation (2.21).

$$\int_{-0.25}^{1.25} \int_{-0.25}^{1.25} F\left(x_0, x_1\right) dx_0 dx_1 \tag{2.21}$$

The integral defined in equation (2.21) may be initialized computationally as follows:

```
x0, x1 = [sympy.Symbol(var,real=True) for var in ['x0, x1']]
ranges = [[x0, -0.25, 1.25], [x1, -0.25, 1.25]]
disc = [x ** 2 + y ** 2 - 1]
H = sympy.special.delta_functions.Heaviside
integral = IntegrableFunction(H(disc[0]),ranges,disc)
```

The integral may then be evaluated using `nquad`:

```
integral.integrate()
```

2.3 Accuracy and Performance of the Integration Procedure

As with any other kind of scientific software, it is critical to verify the accuracy of the integration scheme against the kinds of problems it may be expected to encounter during real-world usage. As discussed in section 1.1, the primary motivation for developing this integration routine was to enable evaluation of integrals arising from weak formulations of mathematical conservation laws. Therefore, I have chosen to demonstrate the accuracy and computational performance of this integration procedure using problems that arise from two such laws: the linear heat equation; and the Euler equations of inviscid, compressible fluid dynamics.

The linear heat equation with constant material properties is given in equation (2.22).

$$\begin{aligned} 0 &= \frac{\partial}{\partial t} c_P \rho u - k \nabla^2 u \Rightarrow \\ 0 &= \int_{\partial V} \left[c_P \rho u dx_1 dx_2 dx_3 - k \frac{\partial u}{\partial x_1} dx_0 dx_2 dx_3 - k \frac{\partial u}{\partial x_2} dx_0 dx_1 dx_3 - k \frac{\partial u}{\partial x_3} dx_0 dx_1 dx_2 \right] \end{aligned} \tag{2.22}$$

In equation (2.22), c_P is defined as the specific heat at constant pressure, ρ as mass density, k as thermal conductivity, and u as heat energy. Some exact solutions to the linear heat equation are given by Polyanin [34], and reproduced in equation (2.23) through equation (2.28).

$$u = A\xi + B \tag{2.23}$$

$$u = A\left(\xi^2 + 2at\right) + B \tag{2.24}$$

$$u = A\left(\xi^3 + 6at\right) + B \tag{2.25}$$

$$u = A\left(\xi^4 + 12at\xi^2 + 12a^2t^2\right) + B \tag{2.26}$$

$$u = A\exp\left(-a\mu^2 t\right)\cos\left(\mu\xi + B\right) + C \tag{2.27}$$

$$u = A\left(-\mu\xi\right)\cos\left(\mu\xi - 2a\mu^2 t + B\right) + C \tag{2.28}$$

In these solutions, A, B, C, and μ are arbitrary constants, n is a positive integer, a is the thermal diffusivity given by $a \equiv \frac{k}{c_P\rho}$, and ξ is a three-dimensional, spatial rotation of the coordinate x defined by $\xi \equiv \cos(\theta)\,x_0 + \sin(\theta)\cos(\phi)\,x_1 + \sin(\theta)\sin(\phi)\,x_2$. Given a suitable choice of solution parameters, these solutions can then be used to generate verification tests based on the linear heat equation.

The linear heat equation provides a good first test of `BACL-MMS` integration, but the system is too simple to adequately exercise many important features of an integration routine for use in software verification. In particular, the linear heat equation does not allow for discontinuities, and discontinuous solutions are a key part of the verification of `BACL-MMS`. The Euler equations of fluid dynamics, however, are both more complex and allow discontinuous solutions, making them an attractive system for more strenuous testing. If ρ is defined as mass density, u_i are the Cartesian velocity components, e is the specific internal energy, p is static pressure, and δ_{ij} is the Kronecker

delta, then the Euler equations are given in equation (2.29).

$$0 = \frac{\partial}{\partial x^0}\boldsymbol{F}_0 + \sum_{j=1}^{3}\frac{\partial}{\partial x^j}\boldsymbol{F}_j = \frac{\partial}{\partial x^0}\begin{pmatrix}\rho\\ \rho u_i\\ \rho e\end{pmatrix} + \sum_{j=1}^{3}\frac{\partial}{\partial x^j}\begin{pmatrix}\rho u_j\\ \rho u_i u_j + \delta_{ij}p\\ u_j\left(\rho e + p\right)\end{pmatrix} \tag{2.29}$$

$$0 = \oint \left[\boldsymbol{F}_0 dx_1 dx_2 dx_3 + \boldsymbol{F}_1 dx_0 dx_2 dx_3 + \boldsymbol{F}_2 dx_0 dx_1 dx_3 + \boldsymbol{F}_3 dx_0 dx_1 dx_2\right]$$

The pressure and internal energy are related by the ideal gas equation of state. Defining $\gamma \equiv \frac{7}{5}$, this can be written as in equation (2.30).

$$e = \frac{1}{2}\sqrt{\sum_{j=1}^{3} u_i u_j} + \frac{p}{\left(\gamma - 1\right)\rho} \tag{2.30}$$

Exact solutions to the Euler equations are more difficult to find than solutions to the linear heat equation, particularly when they must involve discontinuous flow features. Unsteady Riemann problems are one important class of such solutions that can be used to form a suite of problems to test the accuracy of integration routines.

An unsteady Riemann problem is a one-dimensional problem given by an initial condition of two constant states $\boldsymbol{W} \equiv (p, \rho, u_1, u_2, u_3)$ such that:

$$\boldsymbol{W}|_{t=t_0} = \begin{cases}\boldsymbol{W}_L & ; \quad \xi \leq \xi_0\\ \boldsymbol{W}_R & ; \quad \xi > \xi_0\end{cases} \tag{2.31}$$

At time t_0, the two states begin to interact, generating three nonlinear waves, one of which is a linearly degenerate slip line while the other two may be either shocks or rarefaction waves. The choice of initial states determines the final character of the Riemann problem.

Four such problems are given by Toro [35], and shown in table 2.2. The problems are rotated into three-dimensional space as with the heat equation, with ξ representing a three-dimensional rotation of the x coordinate axis. Since each of these rotated Riemann problems contains between 3 and 5 critical surfaces, they form an excellent suite of verification tests for the integration routine for the Euler equations.

2.3.1 The Integration Testing Procedure

I have tested the accuracy and performance of the various integration techniques for both the linear heat equation and the Euler equations. by computing the flux functions $\boldsymbol{F}_\mu$ using randomized sets of parameters and evaluating the integral conservation laws defined in equation (2.22) and equation (2.29). Due to the inaccuracies inherent in numerical integration, the sum of the flux integrals will not be exactly zero, and the value of this residual R can be used as a metric for the accuracy of a particular numerical integration routine.

2.3.1.1 Linear Heat Equation

Integration of solutions of the heat equation is straightforward, because the solutions are smooth. Specific solutions were chosen for the heat equation by selecting random values for the solution parameters used in equation (2.23) through equation (2.28). The random values were selected from parameter ranges that were chosen to avoid integrand functions that span many orders of magnitude across the integration domain, because accurate integration of such functions requires special care.

A given set of random parameters defines a specific solution, and this solution is substituted into equation (2.22). The flux integrals are evaluated and summed, and the resulting residual R is recorded. This process is repeated many times, and visualization of such high-dimensional data can be difficult. Parallel coordinates plots are one useful way to visualize such high-dimensional data sets.

In parallel coordinates plots, each line represents a specific test case. Discrete locations on the x-axis represent the various parameters of the data, along with the units in which these are expressed. Each line will connect a series of points whose x-coordinate corresponds to a given parameter and whose y-coordinate corresponds to the value of that parameter for that test case. For the heat equation, each test case contains eight separate parameters. These correspond to the parameters of the heat equation solutions in equation (2.23) through equation (2.28) and the angles

θ and ϕ by which the solution is rotated to make it three-dimensional.

The results of 1000 separate tests are shown in figure 2.7. As discussed, each line represents a single test case, with the specific solution parameters and the resulting residual represented by the y-coordinate values through which the line passes. As figure 2.7 shows, the residual R was less than $\mathcal{O}\left(10^{-20}\right)$ for all cases. From this result, it is clear that the routines for computing integral manufactured source terms are working as expected for smooth problems.

2.3.1.2 The Euler Equations

For the Euler equations, specific solutions were chosen by applying a rotation to one of the four available Riemann problems from table 2.2. Integration without specifying any critical points was impractically slow for any discontinuous, multidimensional problem, but it was possible to obtain useful results when specifying the critical points for only the innermost level of integration. This is akin to using the initial discontinuity functions, without mapping them through the integration operator. This allowed useful evaluation of up to three-dimensional integrals.

For innermost-only tests, 100 specific solutions were chosen by subjecting each of the Riemann problems to rotation around the z-axis in increments of 5°. Results and performance were much improved compared to integration without critical point specification, and two-dimensional unsteady problems almost always yielded residuals for the quantities of mass m, x-momentum px, y-momentum py, and energy e that were less than 10^{-5}, as seen in figure 2.8. Although encouraging, this accuracy broke down for three-dimensional, unsteady problems, for which full mapping of critical surfaces was required.

By specifying the full complement of critical points at all levels of integration, accuracy is improved to near machine precision, and four-dimensional space-time integrations are possible. The results from 100 such tests are shown in figure 2.9. When three-dimensional, unsteady Riemann problems with the included z-momentum pz were rotated by two randomly chosen angles θ and ϕ, the computed residuals were of order $\mathcal{O}^{-20}$, or essentially machine-precision. Without intelligently mapping critical surfaces, the presence of discontinuous surfaces in the integrand introduces signif-

Table 2.2: Test Riemann Problem Initial States. Riemann problems are composed of two flow states, separated by an initial interface. The exact flow states chosen determine the character of the resulting solution. These four Riemann problems are used to provide test problems for evaluation of the integration procedure.

	Left State			Right State		
Problem	p	ρ	u	p	ρ	u
1	1.0	1.0	0.0	0.1	0.125	0.0
2	0.4	1.0	-2.0	0.4	1.0	2.0
3	1000.0	1.0	0.0	0.01	1.0	0.0
4	0.1	1.0	0.0	100.0	1.0	0.0

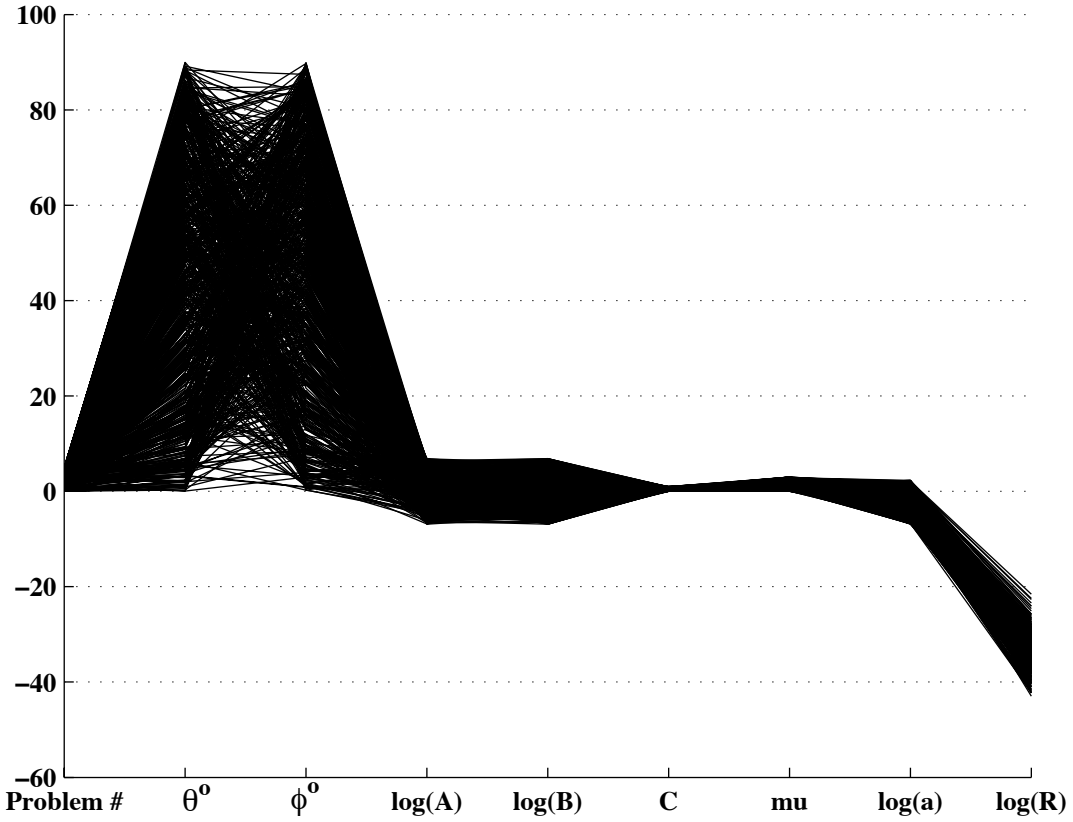


Figure 2.7: Integration Testing Using the Heat Equation. Computing manufactured source terms based on exact solutions results in complex integrals that must evaluate to zero. Because the resulting data is nine-dimensional, it is most easily represented using parallel-coordinates. In this representation, each line represents a single test case, the discrete locations on the x-axis represent specific parameters, as defined in equation (2.23) through equation (2.28), and the numerical values on the y-axis represent the values taken by those parameters. The most important thing to note here, is that the integral test evaluated exactly, with residuals of $O(10^{-20})$, for all 1,000 of the random combinations of parameters tested.

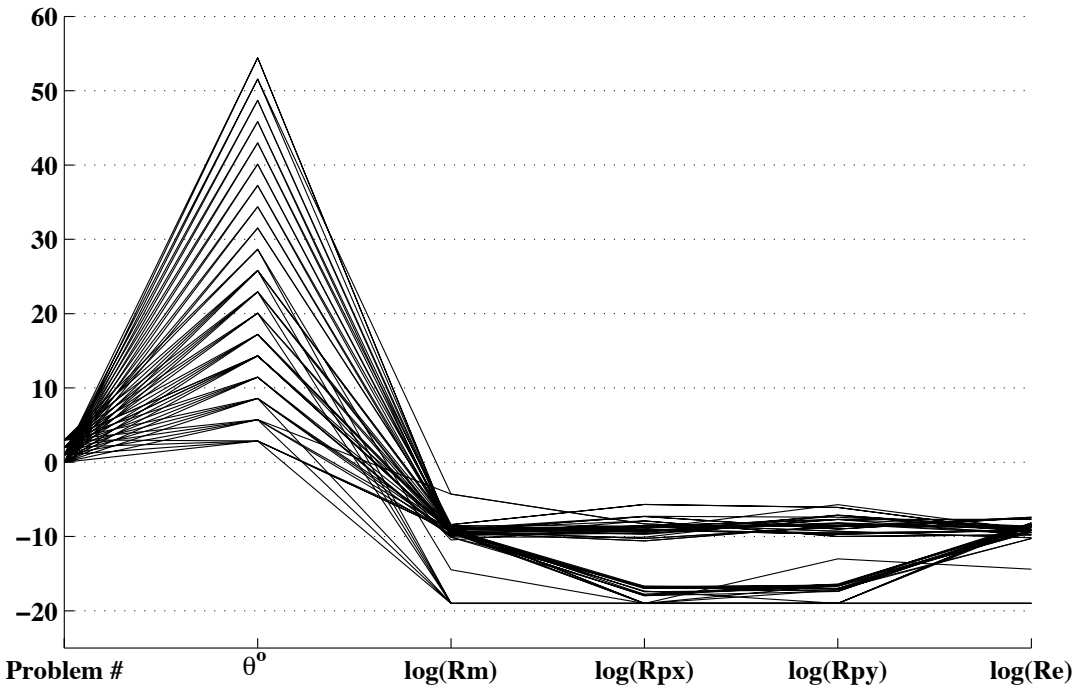


Figure 2.8: Integration Testing Using the Euler Equations, pt. 1. As with the heat equation, computing manufactured source terms for the Euler equations based on exact solutions results in complex integrals that must evaluate to zero. For this test, where critical points were specified only at the innermost level of integration, only two solution parameters were used: a specific Riemann problem from table 2.2; and an angle by which this problem was rotated into two dimensions. However, the vector nature of the equations results in four residual terms which must be evaluated separately, for a total of six dimensions of data. This is again shown in a parallel coordinates representation. In this instance, residuals were much higher, including some as high as 10^{-3} for density. This is poor performance for quadrature, and completely unacceptable for verification of codes.

icant errors that can undermine the credibility of the integration, especially in accuracy-dependent applications such as the method of manufactured solutions. Intelligently mapping critical surfaces through integration operators not only makes such accuracy achievable, but it also greatly reduces the computational cost of obtaining these results.

2.3.2 Integration Timing Tests

For many applications, the number of calls to `scipy.integrate.nquad` may be quite large. Therefore, it is important to understand the various performance optimizations that are available for evaluating these integrals. I have compared the performance effects of critical surface mapping on the evaluation of a single multidimensional integral, given in equation (2.32).

$$\int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} f\left(\xi\right) dx_1 dx_2 dx_3 \tag{2.32}$$

$$f\left(\xi\right) \equiv \begin{cases} 1 & ; \quad \xi < -1.18 \\ \left(0.833 - 0.0334x_1 - 0.0496x_2 - 0.128x_3\right)^5 & ; \quad \xi < -0.0703 \\ 0.426 & ; \quad \xi < 0.927 \\ 0.266 & ; \quad \xi < 1.75 \\ 0.125 & ; \quad \xi \geq 1.75 \end{cases}$$

$$\xi = \cos\left(\theta\right) x_1 + \sin\left(\theta\right)\cos\left(\phi\right) x_2 + \sin\left(\theta\right)\sin\left(\phi\right) x_3$$

$$\theta = 1.33 \qquad \phi = 1.20$$

Integration using the `IntegrableFunction` object consists of two primary step, initialization of the object, which processes the symbolic expressions into the form required for SciPy, and the resulting call to `nquad`. I averaged the timing results over 100 `nquad` evaluations of this integral on a 2012 MacBook Pro with a 2.5 GHz Intel Core i5 processor and 8 GB 1600 MHz DDR3 RAM. The results are given in table 2.3. From these values, it is clear that intelligent specification of critical points in the integration has a dramatic effect on computational performance, resulting in a speedup of more than 60x. Optimizing the innermost integration loop into C code yielded a further

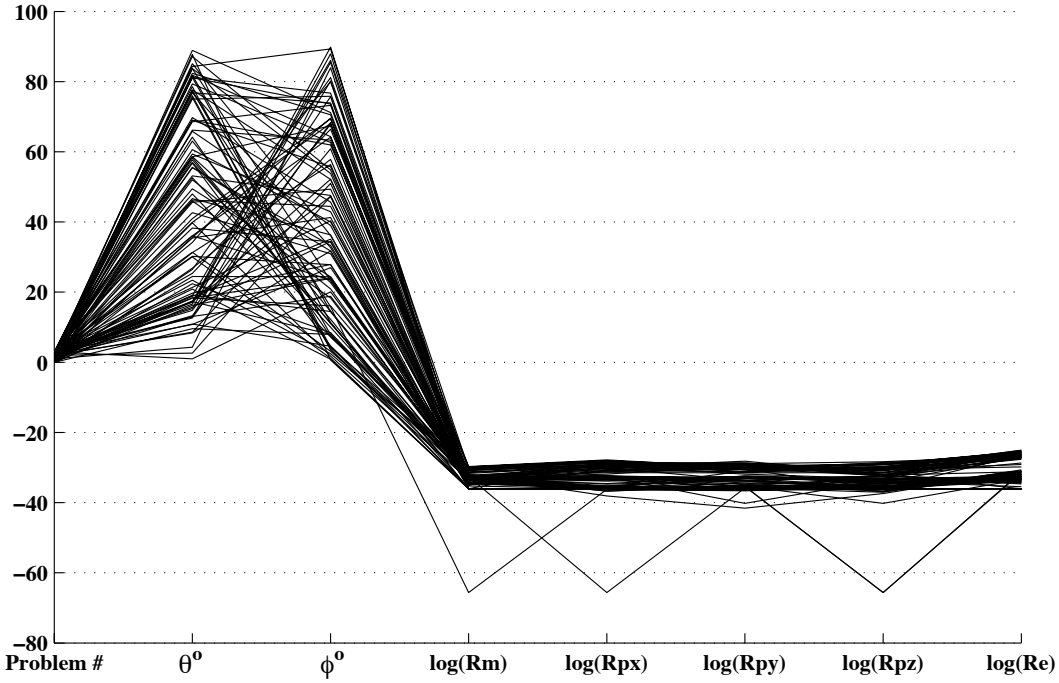


Figure 2.9: Integration Testing Using the Euler Equations, pt. 2. In this test, an additional rotation angle ϕ is introduced in order to allow for testing of three-dimensional integrals, and critical surfaces are both specified and mapped through integration operators to yield of full complement of critical surfaces at all levels of integration. As a result of this mapping, the error resulting from integration of discontinuous functions is completely eliminated, with residuals remaining below 10^{-20} for all tests.

Table 2.3: Integration of Discontinuous Functions – Timing. Integration with `nquad` is subject to many different performance optimizations, including the specification of critical points, as well as optimization of the integrand function. Of these, critical points specification resulted in a 60X speedup, while optimizing the integrand resulted in only 2X improvement.

	Time (seconds)
Python function, no points specified	40
Python function, innermost points specified	12
Python function, all points specified	0.63
Compiled C function, all points specified	0.33

speedup of approximately 2x. The increased speed arises because the adaptive integrator no longer attempts to fully resolve the discontinuities as smooth polynomials, and it is able to concentrate solely on resolving the integrand in its smooth regions. If the critical points were not specified, then the adaptive algorithm used bisection to steadily refine the integral around the critical point. In these tests, this frequently resulting in warnings about integration failing to achieve the required accuracy within the designated maximum number of bisections.

It is also of interest to know how the integration algorithm used by `nquad` scales with increasing dimensionality. `nquad` was therefore tested with the integral of the hyperspheroid step funciton, given by equation (2.33).

$$F_0(n) = \int_V \begin{cases} 1 & ; \quad \sum_{i=1}^{n} \left(x_i^2\right) - 1 < 0 \\ 0 & ; \quad else \end{cases} dx_1 ... dx_n \tag{2.33}$$

This function was integrated over the computational volume $V = [0,\ 1.25]^n$, and the time to compute the integral was measured in seconds. The results of these tests are shown in table 2.4 and in figure 2.10. From these results, it can be clearly seen that the `nquad` algorithm suffers from the "curse of dimensionality", and rapidly becomes unwieldy at very high dimensionalities. In order to evaluate higher-dimensional integrals of discontinuous functions, it will be necessary to replace the simple `nquad` integration routine with a more efficient, sparse integration.

The results given in table 2.3 and table 2.4 are extremely promising, but they cover only the computational costs associated with actually evaluating the numerical integral. The costs of

Table 2.4: `nquad` Integration Dimensional Scaling. `nquad` was used to integrate the volume of an n-dimensional hypersphere, with critical points specified at all integration levels. The integration process was timed at each level, up to five dimensions. Values were extrapolated to six and seven dimensions.

# dimensions	Time (s)
1	3×10^{-5}
2	8×10^{-3}
3	1×10^{-1}
4	$3 \times 10^{+1}$
5	$6 \times 10^{+2}$
6	(1×10^{3})
7	(5×10^{3})

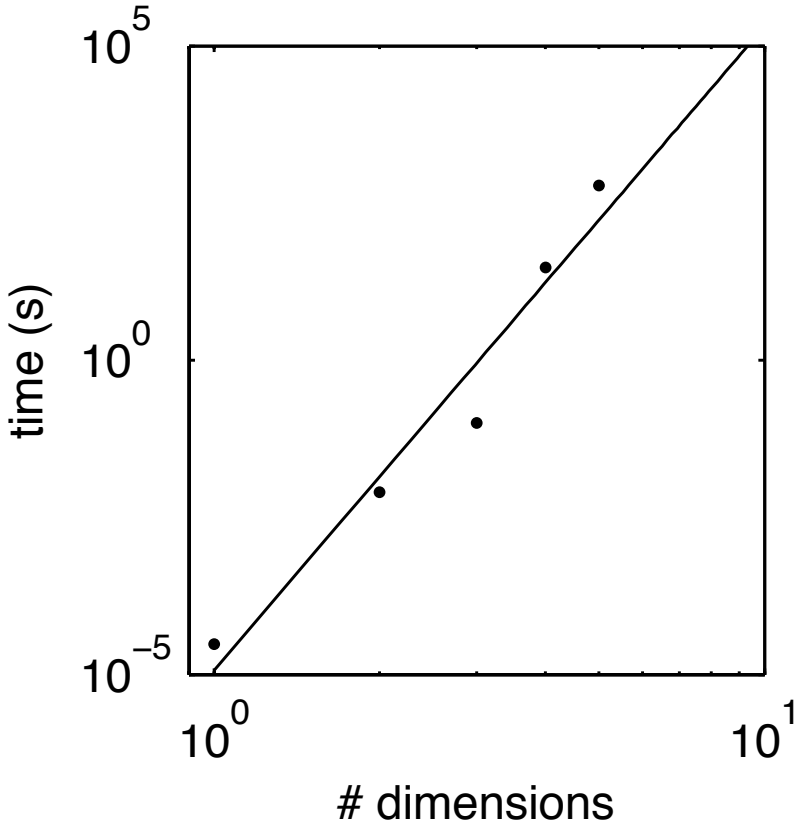


Figure 2.10: `nquad` Integration Dimensional Scaling. `nquad` was used to integrate the volume of an n-dimensional hypersphere, with critical points specified at all integration levels. The integration process was timed at each level, up to five dimensions.

initializing the `IntegrableFunction` object are also significant. Because of the symbolic manipulation and code generation that initialization entails, these costs can actually be quite high. It is therefore advantageous to maximize the utility of `IntegrableFunction` objects as much as possible. In principle, it is possible to use a single `IntegrableFunction` object to represent any integral of a given integrand function, however, this is complicated for discontinuous integrands. For discontinuous integrands, the results of mapping critical surfaces through integration are dependent on a specific integration domain. Therefore, using a single `IntegrableFunction` object in this way requires a more general specification of integration domains. If the bounds of integration can be specified symbolically, then one can use these to define an `IntegrableFunction` object that works over multiple integration domains. Work in this area is ongoing.

2.4 Conclusion

In order to enable reliable verification of numerical simulation codes that may encounter discontinuous solutions, it was necessary to develop fast, accurate tools for accurately evaluating multidimensional integrals of discontinuous functions. In this chapter, I have presented the development of one such tool, and demonstrated how it can be used in the context of verification. I have shown both the accuracy levels that can be expected from the use of this integration tool, as well as the performance increases that can be achieved compared with basic quadrature methods. Both the accuracy and the performance levels achieved are extremely high for multidimensional integrals of discontinuous functions, suggesting that this method will be suitable for use in verification of codes by integral manufactured solutions. Implementation and evaluation of integral manufactured solutions themselves will be discussed in detail in chapter 3.

Chapter 3

Verification and the Integrative Method of Manufactured Solutions

One of the most important challenges facing early-stage designers is that tremendous amount of uncertainty with which they must operate. New projects of any level of complexity require numerous design decisions to be made based on predictions, historical precedents, and educated guesses. Although these uncertainties are an unavoidable characteristic of design, they can be understood, bounded, and reduced using past experience, analytic models, and numerical simulation. The use of these tools allows designers to better predict the behavior and performance of a system, but the decisions made on the basis of the knowledge these tools provide can have far-reaching consequences. As a result, the precise definition of "better predict" can be extremely important; that is, designers must have a quantitative understanding of the level of uncertainty that can be expected from their tools, so that they can make informed design decisions based on a clear understanding of the limitations of their knowledge.

In order to provide a means for better understanding of the uncertainties inherent in numerical simulations, the National Research Council [11] called for improved methods for quantifying the error and uncertainty in the results of a numerical simulation with respect to the mathematical model it implements. The process of quantifying the error in a simulation with respect to the underlying mathematical model is known as verification, and the NRC specifically identified the method of manufactured solutions (MMS) as a key tool in this process. Unfortunately, existing implementations of MMS for engineering software verification are insufficient for the verification of codes that may encounter discontinuous solutions.

Having developed an accurate, efficient means of numerical integration in chapter 2, it is now possible to discuss in detail the integral extensions to MMS that I have developed in order to enable the reliable, robust verification of codes that may encounter discontinuous solutions. The remainder of this chapter will be structured as follows. First, I will introduce the important background concepts that are required in order to discuss code verification in general, and MMS in particular. Second, I will present the theoretical foundations of integral MMS and show how its implementation differs from traditional MMS. Third, I will discuss the use of MMS for verification. Fourth, I will give a brief overview of the `BACL-manufactured` software package, which provides a general-purpose implementation of both differential and integral MMS with a very simple user-interface for a variety of equation sets. Finally, I will demonstrate the use of `BACL-manufactured` in the verification of a compressible, inviscid fluid dynamics solver. I will show that integral MMS yields convergence rates that are equivalent to those obtained using both differential MMS and verification using exact solutions. This will show that integral MMS is ready and able to perform the same rigorous code verification duties for discontinuous solutions that discontinuous MMS has performed for smooth solutions, and finally enable the rigorous, one-step verification of codes in the presence of discontinuities.

3.1 Background

The uncertainties inherent in numerical simulation arise naturally out of the physical and numerical approximations that are used, and so it is important to both understand and quantify the errors introduced by these approximations. Roache [9] classified simulation errors as:

(1) Errors that are a result of modeling approximations, such as fluid incompressibility, etc.

(2) Errors that are ordered by some measure of the problem discretization.

(3) Errors that are the result of some other non-physical approximation, e.g. far-field boundary conditions.

(4) Errors in programming (mistakes, or bugs).

(5) Errors that result from the representation of numbers on a computer.

The study and quantification of modeling errors for a given application is known as model validation, while the study of the remaining mathematical, numerical, and programming errors is known as verification [9]. Verification is further subdivided into code verification and solution verification, where code verification is used to show that the code or software solves a given mathematical model correctly within some domain of inputs, while solution verification estimates the expected numerical error in a solution to a specific problem or application.

The most powerful, accurate, and reliable method of code verification is known as code order verification [18]. For a code that is working as intended, error is expected to be dominated by discretization errors, which scale as Δx^n, where n is determined by the algorithms in use by the code. In code order verification, the code is used to compute a solution to a problem with a known exact solution under successive refinements of the code discretization. The actual rate at which the code converges to the correct solution is computed based on the error between the computational results and the exact solution, and this is compared to the nominal expected value. If the observed convergence rate matches the theoretical convergence rate, then the code is considered verified.

One of the most powerful features of code verification is the ability to analyze the errors resulting from programming mistakes directly. Software debugging is the most difficult and time-consuming part of scientific code development, and modern scientific computing codes are so complex that it is impossible to reliably eliminate all coding mistakes. Thorough code verification allows scientists to at least eliminate any mistakes that introduce error in the computed solution. Unfortunately, thorough verification, or verification that exercises all aspects of a code, can be difficult.

Code order verification, while clearly defined and very reliable, has traditionally suffered from the scarcity of exact solutions for systems of interest. The method of manufactured solutions (MMS) was developed to resolve this problem, by allowing the simple, direct generation of complex solutions for the purpose of code verification [9,10,13,18,19,36]. MMS works by operating a system

of differential equations on some user-defined, "manufactured" solution to derive source terms that may be used to drive a code return that same manufactured solution. By doing this, one is able to verify codes with solutions containing any complex behavior the user may desire.

The measurement of convergence rates to complex, manufactured solutions is the best technique currently available [9, 10, 13, 18, 36], but it is limited in scope, and many important physical systems are simply unsuitable for verification with manufactured solutions as currently available. One of the principal limitations of MMS is that it cannot be applied directly to problems that admit discontinuous solutions [18, 36]. Because it works by operating differential equations on manufactured solutions, the manufactured solutions themselves must necessarily be differentiable in order for MMS to be used, and discontinuous solutions clearly are not differentiable. This is a known limitation of MMS, and researchers have commented on it periodically for many years [18, 19, 36]. Various work-arounds are available, such as using rapidly varying, but smooth, solutions, or splitting verification into parts that separately verify smooth solutions and the treatment of shocks, but a comprehensive approach to the problem of discontinuous solutions has not been found.

Discontinuous solutions are important because they arise in many branches of physics and engineering, and they are an important aspect of many scientific models. These discontinuities can be part of the problem specification, such as material interfaces, or they can arise naturally from the mathematical model, such as aerodynamic shock waves. Regardless of the source of the discontinuities, they render problems unverifiable by differential MMS [9, 10, 13, 18, 36]. In order to establish trust in the codes used to study these phenomena, it is absolutely essential to develop verification tools and techniques that can be directly applied to simulations that contain them.

The root of the problem is that discontinuous solutions to systems of differential equations are not, in fact, solutions to differential equations. Rather, they are solutions to related systems of integral equations, and many computational frameworks for solving systems of this kind are themselves integral in nature. Therefore, in order to correctly verify codes that solve integral equations, one must necessarily use formulation of MMS that is designed for use with integral equations and integral solvers, which will automatically accept discontinuous manufactured solutions.

3.2 Theoretical Framework

As discussed, discontinuous, or weak, solutions are not actually solutions to systems of differential equations. Rather, they are solutions to the integral, or weak, form of those same equations. By approaching the issue of applying MMS to discontinuous solutions as an integral problem, rather than a differential one, many of the difficulties that have traditionally been encountered are resolved automatically. Roy et al. [17] took this approach in their application of MMS to finite-volume codes, though they considered only smooth solutions, and Grier et al [15] also used an integral form of MMS, although their integration procedure is specific to two-dimensional problems.

Traditional MMS expects a code to have governing equations that may be expressed in strong form as a system of differential equations. Such a system may be written as:

$$\sum_{\mu} \frac{\partial F_{\mu}\left(U\right)}{\partial x^{\mu}} + S\left(U\right) = 0 \tag{3.1}$$

The system in equation (3.1) is defined in terms of conservative flux functions F_{μ}, a non-conservative source function S, dependent variables U, and independent variables x^{μ}. The index μ sums over the dimensionality of the problem. Greek letters are used when the summation may include the 0 index to represent time. Latin letters are used to index summations starting from 1.

The relationship in equation (3.1) describes what is known as a differential balance law, and is general enough to be applied in many different contexts. For instance, one common application is to fluid dynamical problems in four-dimensional space-time, where the dependent variables U are the primitive flow quantities of mass, velocity, and pressure, and the independent variables x^{μ} are time and the three dimensions of space. In such an application, the temporal independent variable is represented as x_0. Other applications may include both lower- and higher-dimensional problems, perhaps by taking advantage of symmetries, or operating in phase space. For instance, spherically symmetric inviscid fluid problems will satisfy the Euler equations written in polar coordinates [35]:

$$\frac{\partial F_0\left(U\right)}{\partial x^0} + \frac{\partial F_1\left(U\right)}{\partial x^1} + S\left(U\right) = 0 \tag{3.2}$$

In equation (3.2), x^0 represents time, and x^1 represents the radial distance from the origin. The

vectors of primitive variables, fluxes, and sources are given in equation (3.3).

$$U \equiv \begin{bmatrix} \rho \\ u \\ p \end{bmatrix}; \quad F_0(U) \equiv \begin{bmatrix} \rho \\ \rho u \\ E \end{bmatrix}; \quad F_1(U) \equiv \begin{bmatrix} \rho u \\ \rho u^2 + p \\ u(E+p) \end{bmatrix}; \quad S \equiv \frac{2}{x^1} \begin{bmatrix} \rho u \\ \rho u^2 \\ u(E+p) \end{bmatrix} \tag{3.3}$$

For any system of strong balance laws, one may choose an arbitrary solution U_m and substitute it into equation (3.1). Such an arbitrary solutions will not generally satisfy the balance law, and this will result in a residual term. In the context of manufactured solutions, this residual is typically referred to as a manufactured source term S_m, as in equation (3.4).

$$\sum_{\mu} \frac{\partial F_\mu(U)}{\partial x^\mu} + S(U) = 0 \Rightarrow \sum_{\mu} \frac{\partial F_\mu(U_m)}{\partial x^\mu} + S(U_m) = S_m \tag{3.4}$$

Once computed, this manufactured source term can be used to drive the code being verified so that it returns the manufactured solution. The error in this manufactured solution is then analyzed as part of the verification process.

In contrast, the weak, or integral formulation of MMS expects a code to have governing equations that are expressed as a system of integral equations, rather than a system of differential equations. The weak form of an n-dimensional system of equations of the form given in equation (3.1) can be obtained by integration over some suitable computational volume and application of Stokes theorem, which is the n-dimensional generalization of the three-dimensional divergence theorem. By integrating equation (3.1) over the n-dimensional volume V and applying Stokes' theorem to convert the flux derivatives into integrals over the volume boundary ∂V, one can obtain an integral form of this same law as in equation (3.5). Equations of this form can be used to model many different types of problems. Of particular interest for the purposes of verification, this weak form does not explicitly require that the integrand functions be differentiable.

$$\sum_{\mu} \left(\oint_{\partial V} F_\mu(U) \right) + \int_V S(U) = 0 \tag{3.5}$$

In many instances, one of the independent variables can be positively identified as a "marching" coordinate, along which information propagates only in one direction. Examples include the

downstream coordinate in space-marching schemes for supersonic fluid flows, and the time coordinate in unsteady problems. For such marching problems, it is often possible to derive an alternative weak formulation, where integration is carried out over a computational volume V' composed only of the non-marching coordinates, and the marching coordinate is left in differential form. If x_0 is taken to be the marching coordinate, then this can be written as in equation (3.6).

$$\int_{V'} \frac{\partial F_0(U)}{\partial x^0} + \sum_i \left(\oint_{\partial V'} F_i(U) \right) + \int_{V'} S(U) = 0 \tag{3.6}$$

If the boundaries $\partial V'$ of the integration domain are constant with respect to the marching variable, then the marching derivative may be moved outside of the integral, and the resulting term may be discretized using finite differences. In the case of the unsteady Euler equations in polar coordinates given in equation (3.2), this formulation can be written as in equation (3.7).

$$\frac{d}{dx^0} \int_{x_L^1}^{x_R^1} F_0(U)\, dx^1 + F_1\left(U\left(x_R^1\right)\right) - F_1\left(U\left(x_L^1\right)\right) + \int_{x_L^1}^{x_R^1} S(U)\, dx^1 = 0 \tag{3.7}$$

For problems where it is appropriate, this second formulation is beneficial because it reduces the dimensionality of the integrals that must be evaluated when computing source terms. As a result, it is widely used. However, it can be problematic if $\int_{V'} F_0(U)$ is not everywhere differentiable with respect to the marching coordinate, or if the integration volume is not constant with respect to the marching coordinate. Such difficulties are quite common, even for smooth problems. They can arise from any simulation involving a spatial discretization that varies with respect to time, or with respect to space in a space-marching scheme. As a result, I have focused my attention on the more general formulation in equation (3.5).

Integral manufactured source terms are computed in the same manner as for the differential formulation, by substituting a manufactured solution U_m into the weak system, though the source terms must be computed with respect to a specific computational volume, as seen in equation (3.8). The introduction of this computational volume V does create new difficulties. In practice, source terms are computed based on computational cells, thus making a particular collection of manufac-

tured source terms dependent on a given computational mesh.

$$\sum_{\mu}\left(\oint_{\partial V} F_{\mu}\left(U\right)\right)+\int_{V} S\left(U\right)=0 \Rightarrow \sum_{\mu}\left(\oint_{\partial V} F_{\mu}\left(U_{m}\right)\right)+\int_{V} S\left(U_{m}\right)=\int_{V} S_{m} \tag{3.8}$$

With this new formulation of MMS, it is now possible to more naturally verify scientific codes based on integral equations, especially those involving discontinuous solutions. Such codes are very common, since both finite-volume and finite-element codes are inherently based on weak governing equations, so the weak form of MMS is more naturally applicable to these codes than the strong form [17]. The absence of derivatives in the weak form of MMS makes it inherently permissive of discontinuous solutions, because there is no requirement that the solution U_m be differentiable for the computation of source terms. As a result, equation (3.8) is perfectly applicable to the piecewise-continuous solutions that are common when dealing with high-speed aerodynamic shocks and other discontinuous solutions. From here, it is a simple matter to adapt the traditional MMS procedure for use with integral equations.

3.3 Manufacturing Integral Solutions

The method of manufactured solutions has been in use for some time and the basic methodology is well-established, but its integral form requires a few minor changes. The basic algorithm for finding integral manufactured source terms is as follows:

(1) Choose an appropriate manufactured solution.

(2) Compute flux and source functions.

(3) Integrate to find source terms.

3.3.1 Choice of Manufactured Solutions

As with traditional manufactured solutions, the test solution may be chosen more-or-less arbitrarily, with the added freedom to choose a solution that is only piecewise continuous. Discontinuous solutions may be defined either as piecewise functions or as step functions, but regardless the functional form of the discontinuity is known at this point.

In order to be reliable, code verification must use solutions that fully exercise all important aspects of the code. Knupp [18] gives the following guidelines for constructing traditional manufactured solutions:

(1) Manufactured solutions should be sufficiently smooth on the problem domain that the theoretical order-of-accuracy can be matched by the observed order-of-accuracy obtained from the test.

(2) The solutions should be general enough that they exercise every term in the governing equations.

(3) The solutions should have a sufficient number of nontrivial derivatives.

(4) Solution derivatives should be bounded by a small constant.

(5) The manufactured solutions should not prevent the code from running successfully to completion during testing, since code robustness is not considered part of code verification.

(6) Manufactured solutions should be composed of simple analytic functions.

(7) The solution should be constructed in such a manner that any operators in the governing equations make sense.

(8) Solutions should not grow exponentially in time.

Because the integral form of MMS explicitly allows for discontinuous solutions, items 1 and 4 may be relaxed such that the solution is smooth and the derivatives are bounded, except along certain critical hypersurfaces where the solution may be jump-discontinuous.

For the most part, these guidelines are concerned with the equation set being solved, not with the specific code to be verified. The Center for Predictive Engineering and Computational Sciences (PECOS), part of the Institute for Computational Engineering and Sciences (ICES) at the University of Texas at Austin developed a centralized repository for MMS code that has since been made publicly available as the Manufactured Analytical Solution Abstraction (MASA) library

[19,22]. The MASA project provides "an openly available, application-independent software package that provides generated MMS source terms, solutions, etc."

In particular, MASA provides sample manufactured solutions for a variety of equation sets, including the heat equation, Laplace's equation, the Euler equations (with or without thermal equilibrium chemistry), the Navier-Stokes equations, and the Reynolds-Averaged Navier-Stokes equations with various turbulence models [22]. For the Euler equations, the authors follow the recommendation of Roy et al. [37], and recommend a manufactured solution composed of sines and cosines, as given in equation (3.9), reproduced from the MASA documentation [22].

$$\phi(x,y,z) = \phi_0 + \phi_x\, f_s\left(\frac{a_{\phi x}\pi x}{L}\right) + \phi_y\, f_s\left(\frac{a_{\phi y}\pi y}{L}\right) + \phi_z\, f_s\left(\frac{a_{\phi z}\pi z}{L}\right), \tag{3.9}$$

where ϕ represents the primitive flow variables ρ, u, v, w or p, $f_s(\cdot)$ functions denote either sine or cosine functions, and the remaining variables are user-defined constants.

A manufactured solution such as that in equation (3.9) provides a good test for the steady-state Euler equations under conditions which lead to smooth solutions. In order to define a manufactured solution that also test the shock-capturing capabilities of Euler solvers, I propose to use a solution of the kind shown in equation (3.10), where H is the Heaviside step function defined in equation (3.11), and ϕ_H is the strength of the discontinuity.

$$\phi\left(t,x,y,z\right) = \phi_0 + \sum_{\mu=t,x,y,z} \phi_\mu f_s\left(\frac{a_\mu \pi x_\mu}{L_\mu}\right) + \phi_H H\left(f\left(t,x,y,z\right)\right) \tag{3.10}$$

$$H\left(f\left(x_0,...,x_n\right)\right) \equiv \begin{cases} 0 & ; \quad f\left(x_0,...,x_n\right) < 0 \\ \frac{1}{2} & ; \quad f\left(x_0,...,x_n\right) = 0 \\ 1 & ; \quad f\left(x_0,...,x_n\right) > 0 \end{cases} \tag{3.11}$$

3.3.2 Computation of Flux and Source Functions

Once the manufactured solution is defined, the system of integral equations is then operated on it to obtain the various flux and source integrand functions. This may be a very complex task

if the manufactured solution is sufficiently general, especially for nonlinear differential equations. The use of computer algebra systems to aid in this task is usually advisable.

An example from the MASA project [19] will serve to illustrate the complexity of source terms for general manufactured solutions. For the steady-state, one-dimensional Euler equations, they choose the specific manufactured solution given by equation (3.12).

$$\begin{aligned} \rho(x) &= \rho_0 + \rho_x \sin\left(\frac{a_{\rho x}\pi x}{L}\right) \\ u(x) &= u_0 + u_x \sin\left(\frac{a_{ux}\pi x}{L}\right) \\ p(x) &= p_0 + p_x \cos\left(\frac{a_{px}\pi x}{L}\right) \end{aligned} \tag{3.12}$$

They computed the manufactured source terms by applying the Euler equations from equation (2.29) to this solution. This was done using the Maple software package [38]. Defining a number of auxiliary variables helps improve the readability of the final solution.

$$\begin{aligned} R_1 &= \rho_0 + \rho_x \sin\left(\frac{a_{\rho x}\pi x}{L}\right) \\ U_1 &= u_0 + u_x \sin\left(\frac{a_{ux}\pi x}{L}\right) \\ P_1 &= p_0 + p_x \cos\left(\frac{a_{px}\pi x}{L}\right) \end{aligned}$$

The analytic source terms for the equations governing conservation of mass, momentum and energy are given in equation (3.13). Clearly, additional dimensionality will only complicate the problem further.

$$\begin{aligned} Q_\rho =& \frac{a_{\rho x}\pi\rho_x U_1}{L}\cos\left(\frac{a_{\rho x}\pi x}{L}\right) + \frac{a_{ux}\pi u_x R_1}{L}\cos\left(\frac{a_{ux}\pi x}{L}\right) \\ Q_u =& \frac{a_{\rho x}\pi\rho_x U_1^2}{L}\cos\left(\frac{a_{\rho x}\pi x}{L}\right) - \frac{a_{px}\pi p_x}{L}\sin\left(\frac{a_{px}\pi x}{L}\right) + \frac{2a_{ux}\pi u_x R_1 U_1}{L}\cos\left(\frac{a_{ux}\pi x}{L}\right) \\ Q_{e_t} =& \frac{a_{\rho x}\pi\rho_x U_1^3}{2L}\cos\left(\frac{a_{\rho x}\pi x}{L}\right) - \frac{\gamma}{\gamma-1}\frac{a_{px}\pi p_x U_1}{L}\sin\left(\frac{a_{px}\pi x}{L}\right) \\ &+ \frac{\gamma}{\gamma-1}\frac{a_{ux}\pi u_x P_1}{L}\cos\left(\frac{a_{ux}\pi x}{L}\right) + \frac{3a_{ux}\pi u_x R_1 U_1^2}{2L}\cos\left(\frac{a_{ux}\pi x}{L}\right) \end{aligned} \tag{3.13}$$

Generating flux and source functions for integral manufactured solutions is somewhat simpler, as derivatives do not need to be taken. However, these functions must still be integrated. It is quite difficult to find manufactured solutions that generate flux and source functions that are analytically integrable for systems of interest, and so one must typically rely on approximate methods for numerical integration.

3.3.3 Evaluation of Integrals

Numerical integration methods are a reliable way to evaluate integrals of functions that do not have an analytic antiderivative, but they do introduce challenges of their own. First, like any other numerical approximation technique, numerical integration algorithms introduce some amount of error. This is crucial for code verification, which relies on the analysis of simulation errors to establish code correctness. Any additional error introduced by the verification technique can dramatically complicate this process. Therefore, in order to trust the results of numerical integration in verification, 1) the order of accuracy of the numerical integration routine must be much higher than that of the integral equation solver, and 2) the error introduced by the discontinuities must be small.

Second, because numerical integration techniques do not compute an antiderivative explicitly, a new integral must be evaluated for each and every change in integration boundaries. For verification applications, one must typically evaluate numerical integrals for a large number of different boundaries, and so computational performance becomes critical.

Once the manufactured solution has been chosen, flux and source integrands have been determined, and these integrands have been put into an appropriate form for numerical integration, one must evaluate the integrals. As discussed in section 3.2, a given set integral manufactured source terms is dependent on a specific computational volume. As a result, the number of numerical integrations that must be carried out may be quite large. For a mathematical model of q quantities and dimensionality d, and a regular structured grid with nx_i cells in each dimension, the number

of flux integrals of dimensionality $d-1$ scales as in equation (3.14).

$$q \prod_{j=1}^{d} \prod_{j \neq j} (nx_i + 1)\, nx_j \tag{3.14}$$

The number of d-dimensional volume integrals scales as in equation (3.15).

$$\prod_{i=1}^{d} nx_i \tag{3.15}$$

For example, a simulation of the Euler equations ($q = 5$) that is unsteady in three dimensions ($d = 4$) on a grid that consists of $100 \times 100 \times 100$ cells for 100 time steps would require the evaluation of $5 \cdot 100^3 \cdot 101 \cdot 4 = 2.02 \cdot 10^9$ three-dimensional hypersurface integrals and $5 \cdot 100^4 = 5 \cdot 10^8$ four-dimensional hypervolume integrals. The sheer scale of this problem means that the performance of the numerical integration routine is critical.

Third, most computational tools for numerical integration require the integrand function to be available to the tool in a specific functional form, which is usually not the same as required by computer algebra systems. Therefore, one must either write a function based on the symbolic form of the integrand, or else rely on code generation tools to do so automatically. The availability of such code generation tools can play a significant role in the success of verification by the method of manufactured solutions, as a result.

The fast, accurate evaluation of flux and source integrals is the most difficult aspect of integral manufactured solutions. This is especially true when the integrand functions may contain discontinuities, because the degree to which an integration technique tolerates discontinuous integrands is usually inversely proportional to the computational efficiency of that routine.

Monte Carlo methods provide the simplest approach to integration of multidimensional functions, and they are typically the most efficient when the dimensionality of the problem is high. Monte Carlo methods work by simply sampling the value of the function at pseudo-random points throughout the integration domain, and averaging the results. Because no assumptions are made about the underlying form of the integrand, Monte Carlo schemes work equally well for smooth and discontinuous integrand functions. Unfortunately, accuracy for these methods scales as the

square-root of the number of function evaluations, and high-accuracy results are often difficult to obtain [39]. Despite their amenability to discontinuous integrand functions, this lack of accuracy means that Monte Carlo methods are unsuitable for MMS applications.

Quadrature methods are somewhat more complex than Monte Carlo, but they have much higher rates of convergence for smooth problems. One-dimensional quadrature methods work by sampling the function at a series of points and using these to fit a polynomial, which can then be integrated analytically. This results in an integration technique that is much more accurate than Monte Carlo, for a given number of sample points, provided that the integrand function is well-approximated by polynomials. While this is typically the case for smooth functions, discontinuities in the integrand can lead to greatly reduced accuracy around the jump.

The most common way to accurately integrate discontinuous functions using quadrature is to take advantage of the linearity of the integration operator to subdivide the integration domain into smooth regions, apply quadrature to these parts separately, and sum the results. Subdividing the domain in this way effectively eliminates the effects of the discontinuity from the resulting integration, and is a highly effective method for problems where the functional form of the integrand is known a priori. Failure to account for the discontinuities can result in either error levels that are unacceptable, or computational costs that are too high, or both.

The details of applying quadrature methods to discontinuous integrand functions are discussed in greater detail in chapter 2, but a brief summary of the approach I have chosen will be provided here.

Evaluating multidimensional integrals of discontinuous functions is a challenging problem. Although Monte Carlo methods continue to work, mostly without modification, they remain inaccurate and unsuitable for use in code verification applications. Quadrature methods can also be readily extended to multiple dimensions, but they still require special care when the integrand functions involved may be discontinuous. These difficulties can be addressed by subdividing the integration domain, as in the one-dimensional case, but doing so becomes more complicated because of the more complex topology inherent in multidimensional spaces.

Multidimensional subdivision of integration domains can be entirely avoided by mapping the discontinuous critical surfaces of the integrand function through the various one-dimensional integration operators as these are applied. Doing so gives up any efficiency gains that can be achieved by using inherently multidimensional integration routines, but these are more than compensated by the performance benefits of accurately subdividing the integration domain along critical surfaces. Therefore, this is the scheme that I have used in my investigations into the behavior of integral manufactured solutions.

3.4 Computational Implementation

MMS is an invaluable tool for computational researchers, but it is not without its limitations. It requires the generation of additional source terms, which can be quite complex, and these terms must then be converted into a form that the original code can access and use. I have used the Sympy and SciPy software packages to simplify and automate this process, and I have packaged this as the `BACL-manufactured` software package.

In order to design software to automate the implementation of integral manufactured source terms, it is important to define the expected behavior of this software. This software must accept as inputs a manufactured solution, a specific integral balance law, and a set of computational volumes. Using these, it must return as outputs a set of numbers representing the integral manufactured source terms. Because it is a relatively simple addition, this software will also provide access to differential source terms based on these same inputs.

`BACL-manufactured` is a Python package consisting of three interdependent parts that act together to compute manufactured source terms: a module for numerical integration; a generic equation class that defines how to process fluxes and sources into manufactured source terms; and specific equation classes that define the flux and source terms for specific balance laws, as well as example manufactured solutions. The numerical integration module in `BACL-manufactured` is discussed in section 2.2.3, but the other two sections will be discussed here.

`BACL-manufactured` defines a `SympyEquation` base class to describe the behavior of both

integral and differential balance laws. This is an abstract class, and `SympyEquation` objects cannot be directly instantiated. Rather, `SympyEquation` forms the base class from which equation-specific subclasses are derived. It provides common methods that are required for the computation of manufactured source terms for general balance laws. Some of these are:

- `balance_integrate` - Returns integral form of manufactured source terms, integrated over a specified computational volume. Includes options for shared-memory parallelism.
- `flux_integrate` - Evaluates flux integrals over the surface of the computational volume.
- `source_integrate` - Evaluates volume integral of any source terms inherent in the balance law.
- `balance_diff` - Returns symbolic representation of differential manufactured source terms.
- `balance_lambda_init` - Convert symbolic, differential manufactured source terms to native Python "lambda" for improved computational performance.
- `vector_diff` - Differentiates a list of symbolic expressions and returns the result as a Sympy Matrix.

A related function, scheduled for incorporation into the `SympyEquation` class in the future, is `list_integral`, which evaluates a list of integrals. This is intended to provide functionality that is similar to what the `SympyEquation.vector_diff` method does for derivatives.

A `SympyEquation` subclass-object is instantiated with a Python `dict` containing Sympy representations of the integration variables, the manufactured solution to be used in computing the source terms, and symbolic representations of any discontinuities the solution contains. These are saved as object data, and then the subclass `setup` method is called on the input`dict` to generate symbolic flux and source terms from the manufactured solution and the specific balance law implemented in the subclass.

Once the object is instantiated, source terms may be computed as needed. For use in code verification, there are two principal methods that are of interest. `balance_lambda_init` uses Sympy

code generation tools to generate native Python functions that return the differential manufactured source terms for a given set of computational coordinates. This allows the use of traditional, strong-form MMS for code verification.

`balance_integrate` is called with a specific integration domain and an optional symbolic representation of discontinuities, and it evaluates the integral manufactured source terms over that integration domain. It does this by processing the symbolic integrand, ranges, and discontinuities into a set of `IntegrableFunction` objects (described in section 2.2.3), and then evaluating these individually.

Overall, the `SympyEquation` abstract class greatly simplifies the incorporation of MMS verification into scientific and engineering codes. Using the highly capable Sympy library for symbolic mathematics, it provides all the tools necessary to generate either differential or integral manufactured source terms, given only a manufactured solution, a set of balance law equations, and a list of any discontinuities in the manufactured solution. All of these are made available to the user with just two functions, thus hiding all of the complexities of computing these source terms and providing a simple, capable interface.

The `SympyEquation` class is an abstract class, and is not intended for direct use. Instead, subclasses are used to define specific balance laws to be used in generating source terms. These subclasses are responsible for providing the `setup` method, which defines how to convert a manufactured solution into flux and source terms for differentiation or integration. Implementations are available for the linear heat equation and for the UCS Euler equations described in chapter 4. The files in which these subclasses are defined are also a convenient location for sample manufactured solutions to be stored, as a given manufactured solution is often applicable to many different codes without modification, and only needs to be considered in the context of a single set of balance law equations. This organization allows for easy expansion of the method to different balance laws.

Because so much critical functionality is available in the `SympyEquation` class, the subclasses can be very simple. For the linear heat equation, the entire subclass can be reproduced simply and easily:

```
class HeatEquation(SympyEquation):
    def setup(self, **kwargs):
        self.rho = kwargs['rho']
        self.cp = kwargs['cp']
        self.k = kwargs['k']
        self.fluxes = [sympy.Matrix([self.rho * self.cp * self.sol[0]]),
                       sympy.Matrix(
                           [-self.k * sympy.diff(self.sol[0], self.vars_[1])]),
                       sympy.Matrix(
                           [-self.k * sympy.diff(self.sol[0], self.vars_[2])]),
                       sympy.Matrix(
                           [-self.k * sympy.diff(self.sol[0], self.vars_[3])])]
        self.source = sympy.Matrix([Zero])
```

The design of these subclasses is intended to allow the greatest possible freedom to developers to implement subclasses in whatever way is best-suited to a specific equation set. The interface is as simplified as possible, and the entirety of the Python solution `dict` with which the object is instantiated is available to the subclass as `self.sol`. This has disadvantages, as it is difficult to tell from the interface alone what specific arguments are needed for a given subclass, but it allows experimentation among developers and reduces the friction of adoption for new systems of balance laws.

The only critical aspects of a subclass are that it define the `setup` method with the given function signature (`setup(self, **kwargs)`), and that it define the two objects `self.fluxes` and `self.source`. `self.fluxes` must be a Python list of Sympy Matrix objects of length one or more, each containing a symbolic description of one of the flux terms for the balance law being implemented. `self.source` is similar, except that it is a single Sympy Matrix object, rather than a list.

Given a symbolic representation of the flux and source vectors, as well as a particular manufactured solution, it is a simple matter to compute symbolic representations of the functions which must be integrated. The `integration` package uses Sympy tools to convert these symbolic integrands to more computationally efficient Python functions, with an additional option to generate C functions.

3.5 Demonstration

With the tools and methods developed in this chapter, it is possible to use integral MMS to verify the accuracy of scientific and engineering codes. I have demonstrated the use and utility of integral MMS, by performing a demonstration verification of the `BACL-Streamer` package, which is a code developed in the Busemann Advanced Concepts Lab for compressible, inviscid fluid dynamics. `BACL-Streamer` will be discussed in greater detail in section 3.5.3 and in chapter 4.

3.5.1 Code-order Verification in the Presence of Discontinuities

Code-order verification remains the best available technique for establishing the mathematical accuracy of engineering codes, yet there are some special difficulties that arise when performing code-order verification in the presence of discontinuous solutions. Code-order verification works by measuring the observed rate at which a code converges to a known solution, one can make reliable assertions about the mathematical accuracy of that code. Specifically, if the observed convergence rate matches the nominal order of accuracy of the algorithms the code implements, then it can be asserted that the code does, in fact, implement those algorithms correctly. Discontinuities present a special difficulty for this process, because they can affect the expected rate of convergence of the code. Even more challenging, different types of discontinuities can affect overall convergence rates in different ways.

In fluid dynamical simulations modeled by the Euler equations, there are two primary types of discontinuities that can arise. These are classified based on the properties of the various characteristic fields that cause them. [40] If these fields are genuinely nonlinear (that is, the matrix product of the eigenvector with the gradient of the corresponding eigenvalue is nonzero for all space), then the characteristic field gives rise to a nonlinear wave, such as a shock or a rarefaction wave. If the field is linearly degenerate (the matrix product of the eigenvector with the gradient of the eigenvalue is zero for all space), then the field gives rise to a linearly degenerate slip line (contact discontinuity). These two types of waves have been shown to affect overall convergence

rates differently.

This is a problem, because it can be difficult to distinguish the effects of discontinuities from the effects of simple programming mistakes. Banks et al. [41] have shown that convergence rates may decrease to one-half order due to the effects of linearly degenerate waves, while Sabac [42] showed that it was possible to construct initial data for which monotone schemes converge at a rate no better than one-half. Popov [43] showed a similar effect in higher-order schemes based on the minmod limiter. This is problematic for verification, because it can be unclear what the expected rate of convergence for a given problem should be.

3.5.2 NUMERICA

In order to provide some form of computational baseline against which to compare verification results using integral MMS, I have first performed a sample verification of an independent, established CFD code.

`NUMERICA` is a library of source codes developed by Euletorio Toro and based on the contents of his book on Riemann solvers and numerical simulation of compressible fluid dynamic problems [40]. It contains a fully functional fluid mechanical solver for the unsteady, one-dimensional Euler equations, based on a first-order Godunov method. As the `NUMERICA` code does not allow for the direct solution of problems including source terms, it is unsuitable for verification using MMS. Therefore, I have used the `NUMERICA` solver to model a known, exact solution. I have used the well-known Sod's test, given by the initial data in equation (3.16). The solution to this problem consists of a rarefaction wave, a slip line, and a shock, and therefore it helps to establish the sort of convergence one might expect from a first-order Godunov method in the presence of discontinuous solutions.

Sod's problem is solved over a range of grid resolutions, and convergence rates are computed based on the L_1, L_2, and L_∞ norms. These are shown in table 3.1, and plots for pressure are shown in figure 3.1.

$$(\rho, p, u)\,|_{t=t_0} = \begin{cases} (1.0, 1.0, 0.0) & ; \quad x \leq 0 \\ (0.1, 0.125, 0.0) & ; \quad x > 0 \end{cases} \tag{3.16}$$

From these results, it is clear that the slip line contained in Sod's test does indeed result in half-order convergence. The L_1 norm yielded the highest overall convergence rates, although these are somewhat inconsistent among flow variables. The L_2 and L_∞ norms yielded consistent convergence rates for different flow quantities, but they were also lower than those obtained using the L_1 norm. The L_1 norm is the most widely reported in the literature on verification of codes, and so I have followed suit in the discussion that follows.

3.5.3 BACL-Streamer

In my time at the Busemann Advanced Concepts Lab, I have developed a code for solving the Euler equations as part of my research into the unified coordinate system [44, 45]. `BACL-Streamer` is a finite-volume code that uses the same first-order, exact Godunov method used by NUMERICA to solve the conservative Euler equations. The code uses the alternative finite-volume formulation discussed in equation (3.6), where the time derivative is left in strong form. This derivative is advanced using a simple forward-Euler scheme. Arbitrary source terms are included through the use of partial time steps as described in Toro [40]. For integral MMS, source terms are computed using the four-dimensional space-time cell associated with the evolution of a spatial volume over one time step, as described in equation (3.5), and then incorporated directly with the numerical fluxes as in equation (3.17). For differential MMS, source terms are handled using the SciPy differential equation solvers.

$$\int_{\partial V} \mathbf{F_0}|_{x_0=x_{0-}+\Delta x_0} dx_1 dx_2 dx_3 = \int_{\partial V} \mathbf{F_0}|_{x_0=x_{0-}} dx_1 dx_2 dx_3 + \oint_{\partial V} \mathbf{F}_i d\mathbf{A}_i\,(x_0, x_1, x_2, x_3) + \int_V \mathbf{S}_m dx_0 dx_1 dx_2 dx_3 \tag{3.17}$$

Table 3.1: Convergence of NUMERICA for Various Flow Variables and Norms. For a given code, and a given verification solution, convergence rates may depend on what variable is being tested, as well as what norm is being used to represent the error. Convergence rates were computed for density, velocity, and pressure using various norms for solutions obtained using the NUMERICA [35, 40] software. The L_1 norm was the least consistent among flow variables, but most theoretical work on convergence is based on this norm, which is also widely used in the literature.

	L_1 error	L_2 error	L_3 error
Density	0.52	0.48	0.13
Velocity	0.71	0.48	0.13
Pressure	0.58	0.48	0.13

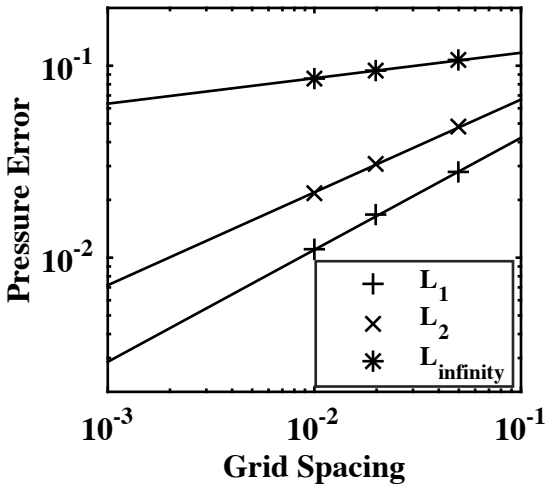


Figure 3.1: NUMERICA Grid Convergence of Various Norms. The NUMERICA code was tested against a Riemann problem for a variety of grid resolutions. The resulting error was computed using the L_1, L_2, and L_∞ norms. These resulted in different convergence rates. L_1 is the most widely used in the literature.

3.5.4 Verification Solutions

I performed a sample verification of BACL-Streamer using exact solutions, differential MMS, and integral MMS. I measured convergence rates using both differential and integral MMS for smooth problems and using integral MMS and exact solutions for discontinuous problems. The exact solution chosen was Sod's test (equation (3.16)), as was done for NUMERICA. I did this in order to establish two distinct aspects of integral manufactured solutions. First, one must demonstrate that the measured convergence to smooth manufactured solutions is equivalent whether a differential or integral algorithm is used. Second, one must demonstrate that the measured convergence to discontinuous solutions is the same for both manufactured and exact solutions. The two cases were considered separately.

3.5.4.1 Smooth Solutions

Following the example of the MASA library [19], I chose a smooth solution given by a scaled cosine function, in equation (3.18).

$$p = \rho = u = v = w = 1.1 + 0.5 \cos\left(\frac{2\pi x}{N_x}\right) \tag{3.18}$$

This solution is a simple, powerful choice for use with MMS. It is completely non-physical, and it therefore provides a non-trivial test of the accuracy of the fluid mechanical solver with included source terms.

3.5.4.2 Discontinuous Solutions

In order to test the shock-capturing capabilities of `BACL-Streamer`, while maintaining the ability to compare results with smooth problems, I chose a discontinuous manufactured solution based on a modification of equation (3.18). This is shown in equation (3.19). No effort was made to ensure that the discontinuity satisfied any form of the Rankine-Hugoniot jump conditions. Both the smooth and discontinuous manufactured solutions are shown in figure 3.2.

Table 3.2: Choice of Grid Refinement

	Grid spacing	Grid sizes	x ranges
Manufactured Solutions	$\Delta x = 1, 2, 5$	$nx = 101, 51, 21$	$\{0, 100\}$
Sod's problem	$\Delta x = 0.01, 0.02, 0.05$	$nx = 100, 50, 20$	$\{0, 1\}$

$$p = \rho = u = v = w = 1.1 + 0.5 \cos\left(\frac{2\pi x}{N_x}\right) + \begin{cases} 0 & 1 \quad x \leq N_x/2 \\ 1 & ; \quad x > N_x/2 \end{cases} \tag{3.19}$$

3.5.5 Measured Convergence Rates

Measurement of a code's order of convergence requires the use of multiple grids at different levels of refinement. For structured grids, such as those used in `BACL-Streamer`, it is a simple matter to begin with a fine grid, and obtain coarser grids by neglecting multiples of two, three, etc., or else to obtain a finer grid by simply doubling the number of points, as shown in table 3.2. For the manufactured cases, we began with a 101-point cell-edge grid and progressively defined 51- and 21-point grids using $\Delta x = 1, 2, 5$. For the Riemann problems, we began with a 100-point cell-center grid and used $\Delta x = 0.01, 0.02, 0.05$. For all of these, the time step was chosen for stability and scaled along with the grid spacing so that halving the resolution of the computational grid corresponded to doubling the time step used.

Having defined the grids to be used, I computed the numerical solutions, and the L_1 norm of the pointwise-error between the numerical solutions and the known, exact solutions. I then fit the results to the function $f(\Delta x) = A\Delta x^n$. The results and their fits are shown in figure 3.3, figure 3.4, and table 3.3.

It can be seen from the function fits given in table 3.3 that `BACL-Streamer` did, in fact, converge at the expected first-order for smooth manufactured solutions, regardless of whether differential or integral solutions were used. This is excellent, as it shows that integral MMS performs as well as the current gold-standard traditional MMS for smooth problems. Discontinuous manufactured solutions also resulted in first-order convergence. First-order results are excellent for a

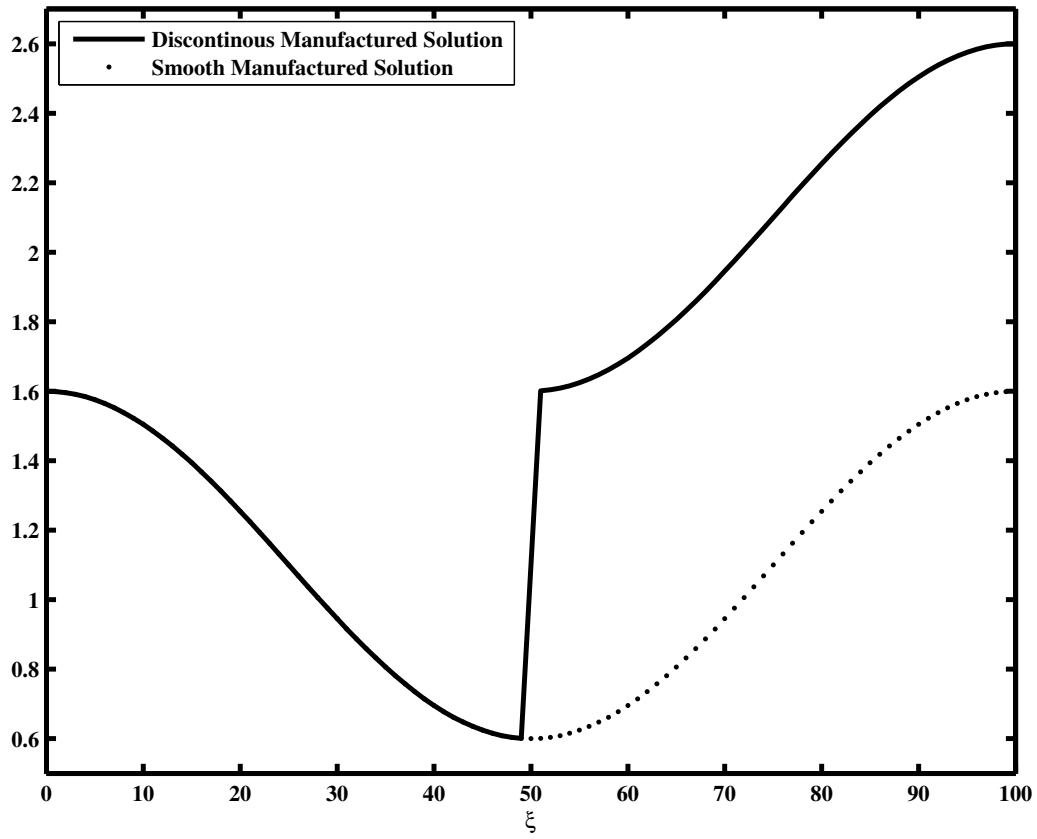


Figure 3.2: 1-Dimensional Smooth and Discontinuous Manufactured Solutions. These two manufactured solutions are used to verify the performance of `BACL-Streamer` for one-dimensional flows. The discontinuous solution is equal to the smooth solution plus a unit step function.

Table 3.3: Measured Convergence Rates. Convergence rates were computed using `NUMERICA` and `BACL-Streamer`, with both manufactured and exact solutions. Rates were computed using the L_1 norm based on error in pressure. For the smooth solutions, first-order convergence was expected. For Sod's test, one-half-order convergence was expected. Discontinuous MMS was expected to lie somewhere between these two.

	Function fit	Convergence rate
Smooth, differential, manufactured solution	$y = 3 \times 10^{-4} \Delta x^{1.0}$	1.0
Smooth, integral, manufactured solution	$y = 2 \times 10^{-3} \Delta x^{0.96}$	0.96
Discontinuous, integral, manufactured solution	$y = 5 \times 10^{-3} \Delta x^{0.95}$	0.95
`NUMERICA`, Sod's test	$y = 2 \times 10^{-1} \Delta x^{0.58}$	0.58
`BACL-Streamer`, Sod's test	$y = 4 \times 10^{-1} \Delta x^{0.64}$	0.64

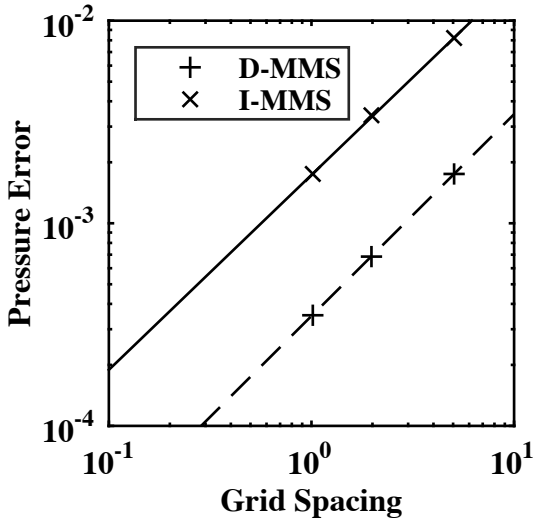


Figure 3.3: Smooth MMS Convergence. When `BACL-Streamer` was verified using a smooth manufactured solution, both differential and integral methods returned nearly identical first-order convergence rates. The error coefficient was slightly different, but this is not typically used in code verification.

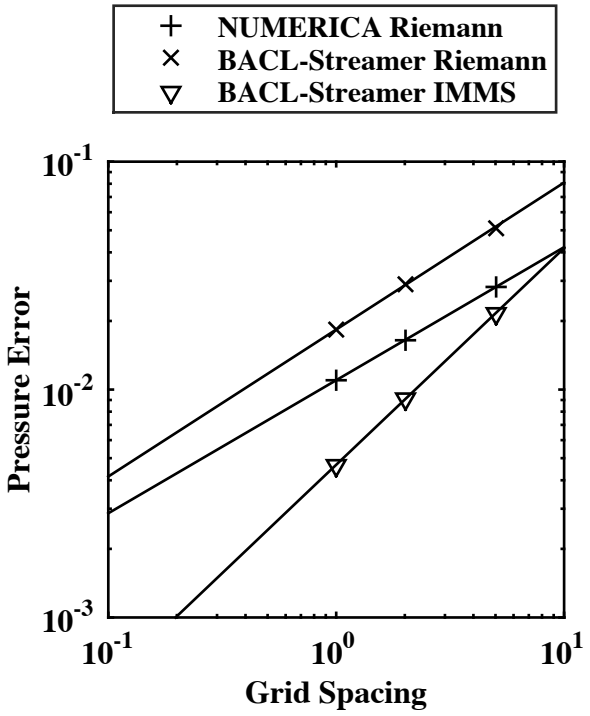


Figure 3.4: Discontinuous Convergence. Because there is no way to use differential MMS to compute convergence rates for discontinuous solutions, the results of integral MMS were compared to convergence rates for exact, discontinuous solutions. These rates are complicated by the fact that some types of discontinuities can lower convergence rates below first-order, as seen in the exact solutions, which converge at only one-half order. The arbitrarily chosen discontinuity used in the MMS solution has no such difficulty, converging at the same first order as was shown for smooth solutions.

manufactured solution that largely ignores the physics of the underlying system of equations, as this one ignores the Rankine-Hugoniot jump conditions, although it remains unclear exactly what convergence rates one should expect for other problems of this kind. Finally, the Riemann solution also yielded the expected convergence rate of one-half, demonstrating close agreement with both `NUMERICA` and the expected convergence rate for linearly degenerate problems.

3.6 Conclusion

The accurate verification of computer codes based on numerical methods is critical to the advance of computational science and engineering. Although MMS has done much to improve verification for codes solving differential equations, it is insufficient for the verification of codes based on integral equations. This is true even when these are simply weak forms of differential equations, if those equations and the codes solving them may be reasonably expected to encounter discontinuous solutions. In order to verify such codes, it is necessary to use an integral form of MMS, as I and Grier [15] have both shown. This method must be numerically accurate, to far greater precision than would be expected from the codes it is intended to verify.

I have demonstrated one form of MMS for use with integral equations with discontinuities. I have developed a technique for mapping critical surfaces through integration operators, and I have used this technique to develop a discontinuous integration algorithm that is both fast and accurate. I have also shown the results of a sample verification carried out on a compressible fluid mechanical code. In that verification, no significant difference can be seen in the convergence rates of codes verified with integral manufactured solutions and codes verified with exact solutions or differential manufactured solutions. As a result, I recommend the use of integral manufactured solutions for the verification of scientific codes based on integral equations, especially if they may be expected to encounter discontinuous solutions.

Future work in this area should include optimizations to the integration algorithms and code to further improve computational efficiency, and further research into the propagation of discontinuities through nested levels of integration.

Chapter 4

The Unified Coordinate System

As discussed in chapter 1, powerful tools for numerical simulation have historically been unwieldy and difficult to use for non-experts. It has not been uncommon for a complex simulation to require weeks of time from a dedicated, specialized team, just to set the simulation up. In early stages of project design, where resources are limited and rapid turnarounds essential, the use of numerical simulations has therefore been highly constrained. This has certainly been the case in the field of fluid dynamics.

Computational fluid dynamics, or CFD, dramatically altered the field of aerodynamics during the twentieth century, and continues to be an essential part of aerodynamic analysis around the world, both as a supplement to wind-tunnel testing and as a replacement when circumstances require. Unfortunately, CFD is not without its costs, especially for design work. To quote Jameson, "CFD is still not being exploited as effectively as one would like in the design process. This is partially because of the long set-up times and high costs, both human and computational, of complex flow simulations." [6]

Automatic grid generation for complex problems would greatly reduce the set-up time and human costs, and good progress has been made on this front using unstructured grids. However, as illustrated in the AIAA Drag Prediction Workshop series, the exact form of the computational grid can have a dramatic effect on overall simulation accuracy. In their summary on grid quality issues from the DPW, Mavriplis et al. pointed out that the most reliable way to gauge the effects of design changes on aerodynamic quantities was to apply mesh deformation techniques to an existing

computational grid with identical geometric topology. [46] This unfortunately requires the prior availability of a high-quality grid which can be deformed.

In addition to these challenges, it can also be desirable to avoid the use of unstructured grids in favor of structured grids, which offer significant advantages in terms of memory use, grid adaptation, and algorithm maturity (see [47] [48] [49] and [50]), have so far resisted automation. Additionally, quadrilateral and hexahedral cells such as those used in structured grids are "widely acknowledged to be superior to unstructured meshes (tet cells)." [51]. This is especially true near flow boundaries, because of the greater ability of hexahedral cells to capture strong gradients, align with flow geometries, and define grid normal vectors [51], and many codes use structured grids near boundaries for just this reason [52].

W.H Hui et al. developed and demonstrated the unified coordinate system (UCS) as a coordinate framework that would unify the strengths of Lagrangian and Eulerian coordinates, namely, sharp resolution of contact discontinuities and a robust, non-deforming grid. UCS was developed as a fully conservative system of conservation laws that contained both Lagrangian and Eulerian coordinates as special cases, and that also allowed for intelligent control of grid motion in order to obtain desired grid properties [21, 53–55]. A fortuitous result of Hui's work was the development of a methodology for computational fluid dynamics that provided automatic, structured, grid generation as part of the solution of the flow. Computational grid points could be generated at the upstream boundary and allowed to flow through the simulation region in order to fill the area of interest. This eliminated the need for non-trivial grid generation entirely, and allowed complex, body-fitted grids to arise naturally out of the flow solution.

Hui's methodology provided other benefits, as well. In particular, it was able to simulate flow around supersonic bodies extremely efficiently. Using the UCS automatic grid generation, as well as the use of space-marching schemes for steady-state supersonic flows, Hui demonstrated a solution for flow around a diamond airfoil, from start to finish, in 1.8 seconds [55]. Steady-state schemes could not be used the comparable simulations in Eulerian coordinates, because the flow velocity in the direction of marching became subsonic. As a result, obtaining the solution to this

problem using Eulerian coordinates required the use of a time-marching method (2393 s), as well as generation of a body-fitted mesh (2180 s), and the result was substantial smearing of both the shocks and the expansion waves. [55]

The actual grid motion is completely specified by requiring the grid velocity to preserve some quantity of interest, such as skewness or jacobian. This eliminates the major problems of skewness and arbitrarily small jacobian [56] associated with grid-point-movement methods of solution-adaptive grid refinement, while simultaneously yielding the excellent resolution of slip lines that is characteristic of a streamline-oriented grid.

In this chapter, I will show the theoretical underpinnings of the unified coordinate system, I will discuss various means of implementing the system, and I will present example problems that demonstrate the power and utility of the UCS method. First, I will discuss the UCS transformation itself. Second, I will show how the transformation can be applied to the Euler equations of compressible, inviscid, fluid dynamics. Third, I will discuss several ways in which UCS grid motion can be controlled, and provide some comments based on my experiences with UCS over the years. Fourth, I will discuss the Godunov method for solving the fluid equations. Fifth, I will show how the Euler equations in unified coordinates may be solved using the Godunov method by deriving a solution to the Riemann problem in the unified coordinates. Sixth, I will present two versions of a complete computational algorithm for simulation of flows using UCS. Seventh, I will demonstrate several example applications which showcase the ability of UCS to bypass traditional grid generation processes. Eighth, I will discuss briefly the various versions of UCS code that I have developed, with their strengths and weaknesses. And ninth, I will discuss some of the more interesting options that are available for improving the accuracy and reliability of UCS for general fluid flows.

This work lays the theoretical and computational foundation for future projects, and establishes preliminary confidence in Hui's method for design-oriented fluid simulations.

4.1 Background on the Unified Coordinates

The unified coordinate system arose from a series of studies that used Lagrangian coordinates to clearly resolve slip lines in supersonic flows, and it is best understood in that context. Traditional computational fluid dynamics (CFD) is a field theory, in that it models the behavior of a continuous fluid at discrete locations in space and time. The fundamental quantities in this system are scalar and vector fields representing some complete set of flow quantities such as the set of pressure, temperature, and velocity, or the set of mass, momentum and energy. This arrangement is known as an Eulerian coordinate system. In contrast, a Lagrangian coordinate system is a particle theory, in that it models the behavior of discrete blobs (particles) of fluid as they move, distort, and interact with each other. In a Lagrangian coordinate system, flow quantities are a function of specific fluid particles, rather than of space and time.

In many ways, Lagrangian coordinates result in a simpler form of physics. Newton's laws of motion, for instance, describe the motion of particles under the influence of forces, whereas the advective terms in the fluid equations vanish in Lagrangian coordinates. Unfortunately, the distortion that affects fluid particles of finite size makes it very difficult to compute approximate numerical solutions for the fluid equations in Lagrangian coordinates. As a result, Eulerian fluid dynamics has become the industry standard. The Unified Coordinate System (UCS) is one attempt to capture the most important benefits of Lagrangian coordinates, while still maintaining the ability to solve the equations computationally, as with Eulerian coordinates.

This is done by beginning with the fluid equations in Eulerian coordinates, and then introducing a generic, time-dependent coordinate transformation, which results in an Eulerian computational grid in computational space, that moves freely in physical space. At its most basic, the computational grid moves in the same direction as the fluid, but the speed of the grid is constrained in order to control grid distortion. This additional freedom introduced by allowing the speed of the grid points to vary allows the simulation to capture many useful properties of the Lagrangian coordinate system, such as automatic grid generation and excellent slip line resolution,

while also maintaining a structured, regular (orthogonal, if the flow permits) grid. In particular, the grid follows fluid pathlines, automatically conforms to solid boundaries, and resolves slip lines nearly perfectly. It does this at the cost of requiring the dynamic creation and destruction of nodes at in- and out-flow simulation boundaries, and the computational costs associated with solving a linear, first order, ordinary differential equation in space at each time step.

Because of their roots in lagrangian fluid dynamics, these lagrangian-esque moving meshes invite comparison with Arbitrary-Lagrangian-Eulerian (ALE) schemes common in fluid-structure interaction research, but the two are in fact quite distinct. Although the present work does involve a grid that is, in a sense, a combination of Lagrangian and Eulerian grids, the implementation of ALE schemes is typically done very differently. In most ALE schemes, a fully Lagrangian grid is advanced in time for some arbitrary number of time steps before being interpolated onto an Eulerian grid to control distortion. This methodology is unsuitable for grid generation, because it requires an initial grid itself, and the interpolation routine is a source of diffusion that can eliminate many of the accuracy advantages Lagrangian coordinates provide [21].

The unified coordinates were first used to solve time-dependent flows in 1999 [53], where they were applied to the two-dimensional, inviscid, Euler equations for an ideal gas. In that paper, a Godunov scheme with a flux-limited MUSCL extension to second order was used to solve a variety of problems, including a steady Riemann problem, flow through a transonic duct, Mach reflection of a traveling shock wave, and an implosion/explosion problem. In 2001 [54], a similar scheme was applied in three dimensions to the steady Riemann problem and to supersonic corner flow. Two-dimensional, inviscid, external flows around both steady and oscillating airfoils were presented later [57]. During its original development by Hui et al., it was found that uniform Cartesian grids could be generated at the upstream simulation boundary, and allowed to "flow" through the simulation region, automatically conforming to simulation boundaries and fitting itself to body surfaces, as shown in Fig. 4.1. Unsteady boundary conditions were also easily modeled with UCS, as in Fig. 4.2.

Hui and his team also successfully applied the unified coordinate system to the Navier-Stokes

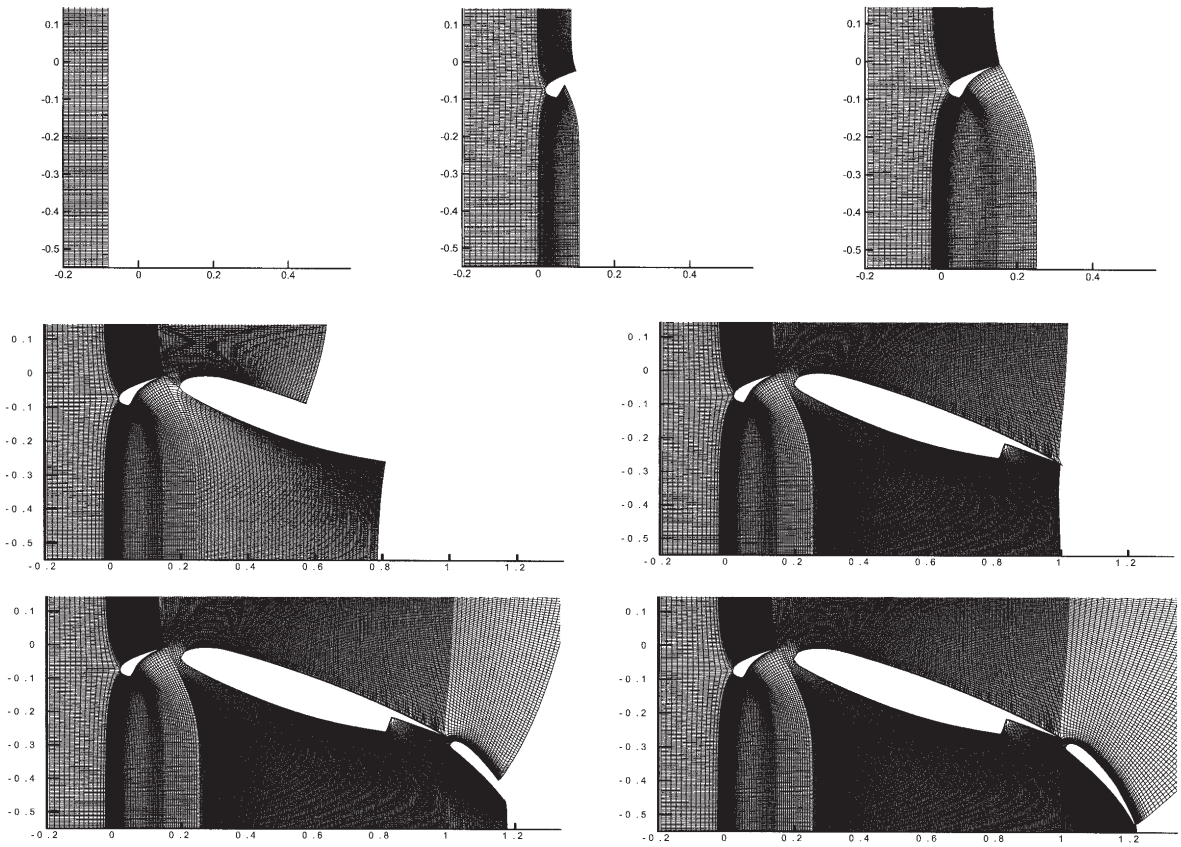


Figure 4.1: Automatically Generated Grid Around a Complex Body [58]. One of the primary challenges of CFD, especially in early design, is the tedious process of grid generation, especially for a complicated geometric body. [55] Hui demonstrated how the unified coordinates could be used to compute flow around a multi-element airfoil, demonstrating how UCS could automatically generate body-fitted computational grids for complex geometric bodies.

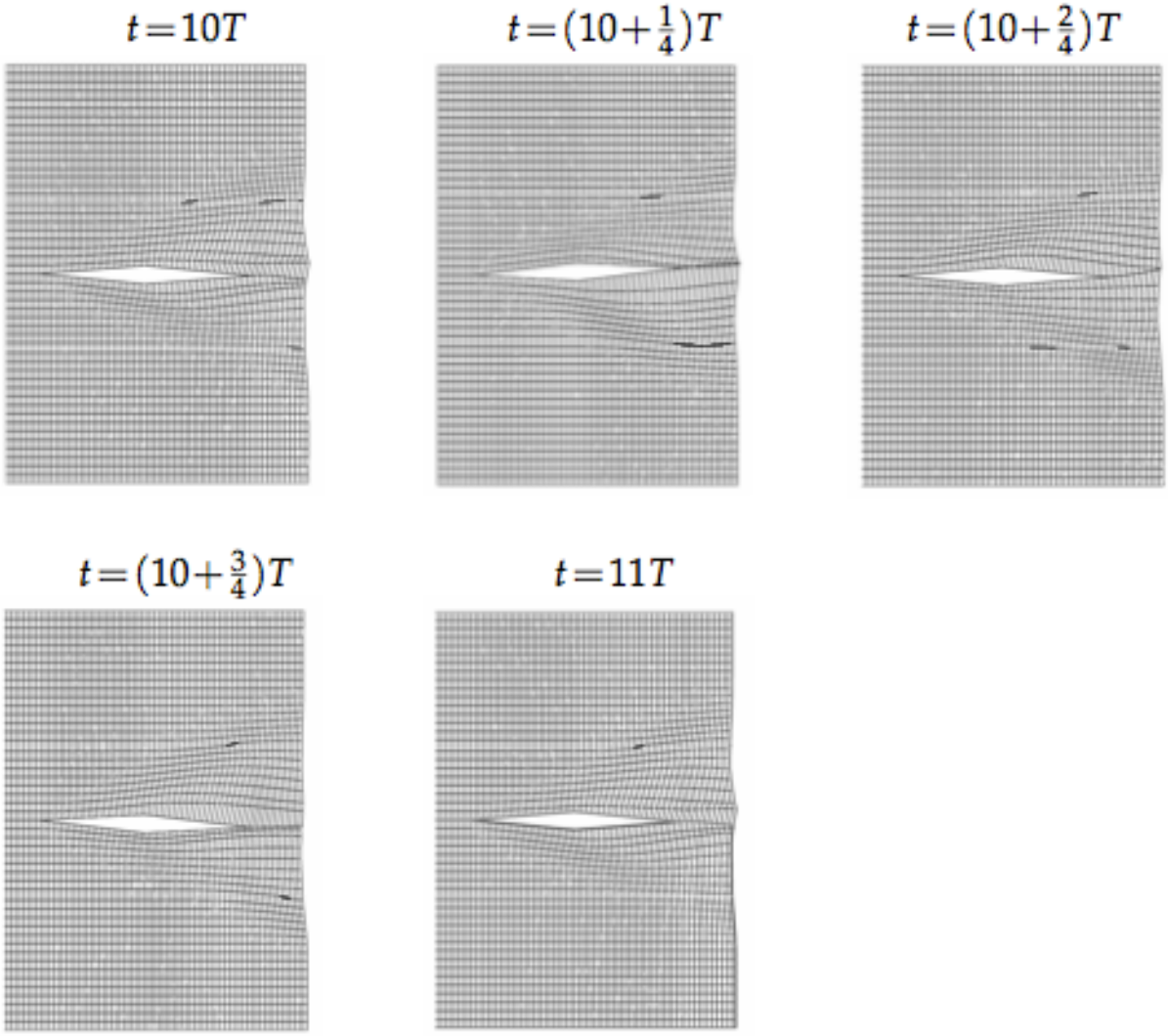


Figure 4.2: Automatically Generated Grid Around an Oscillating Airfoil [21]. The unsteady nature of the UCS grid is also particularly amenable to modeling unsteady geometries, such as those that arise from unsteady boundary conditions. Hui demonstrated this capability by modeling the flow around an oscillating diamond airfoil.

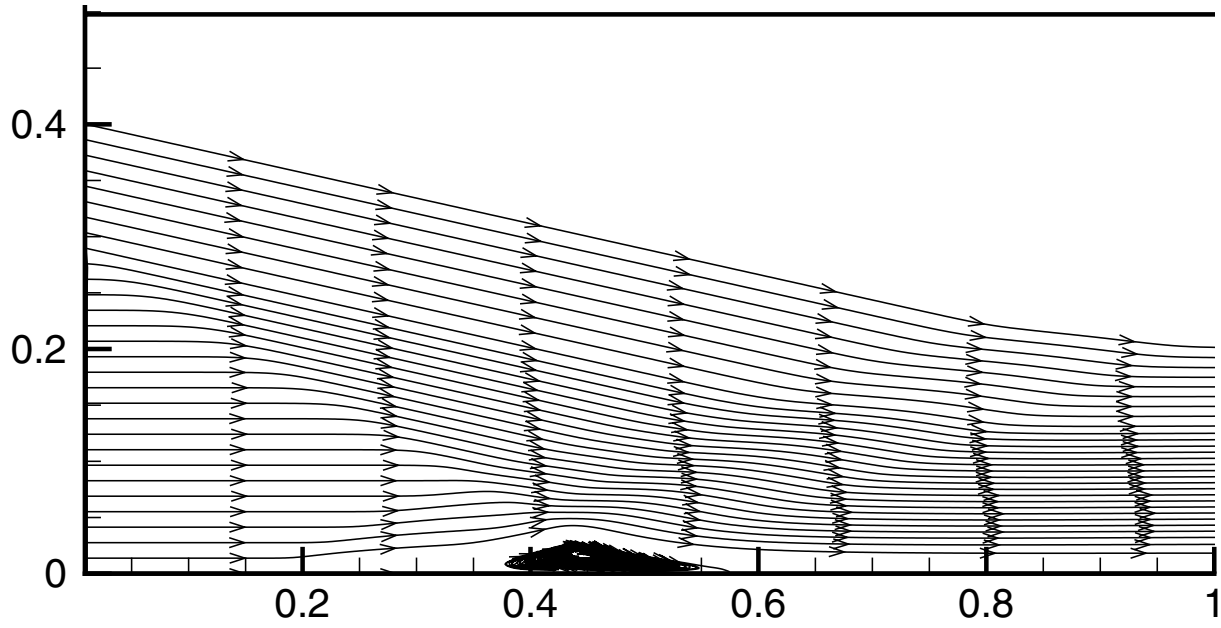


Figure 4.3: Shock-induced Boundary-layer-separation and Recirculation [59]. One of the major concerns about a Lagrangian-based grid such as UCS is that it will be unable to deal effectively with recirculating flows. Hui showed this was not necessarily a problem by computing the flow through a recirculating boundary-layer-separation bubble.

equations in two dimensions. Their principal applications were to boundary-layer flows. The first was a shock-boundary-layer interaction problem, composed of an incoming oblique shock impinging on a boundary layer to induce separation. They found that the unified coordinate system was quite capable of reproducing recirculating flow fields, despite its Lagrangian origins, as shown in figure 4.3. The second problem was a shock-shock-interaction problem in a dual-ramp channel. They found that the unified coordinates produced results that were more accurate than a simulation using stationary, Eulerian coordinates, and did so with fewer grid points. Finally, they tested the Blasius solution, in order to obtain a verification problem with which to quantify the accuracy of their method.

UCS has been applied to more than just fluid mechanics. Additional models to which UCS has been applied include multi-material flows [60], magneto-hydrodynamics [61], and gas-kinetic BGK simulations of a freely falling plate [62] [63] . UCS provides several advantages over traditional CFD, such as better slip line resolution and streamline-oriented grids, but our interest in the system arises primarily from its automatic generation of a computational grid. Design optimization codes will produce many different design options (thousands), and there is a need for a low-level CFD program that is reasonably fast, reliably accurate, and requires little interaction on the part of the user to validate these designs. The automatically generated, streamline-aligned coordinate system UCS provides has the potential to satisfy these requirements, if it can be shown to work reliably and in an intuitive way for a wide range of simulations.

In summary, the unified coordinate system has been used and applied in a wide variety of fields. It possesses two primary advantages: first, automatic generation of a structured, body-fitted grid; second, streamline-alignment of the grid. Streamline-alignment leads to many useful properties, including excellent resolution of contact discontinuities.

4.2 The Euler Equations in Unified Coordinates

The unified coordinate system (UCS) represents a specific class of unsteady, curvilinear, coordinate transformations. As such, one must be able to verify that the transformation itself is

well-behaved, and results in a conservation set of equations, as one would expect. We here present the method for deriving the Unified Coordinate System (UCS) transformation, and we also show how to transform the conservation form of the Euler equations from Cartesian to UCS coordinates.

The application of the UCS transformation to the Euler equations has been previously shown in the various published works by Hui et al., e. g. [21, 53, 54]. However, because the transformation can be quite involved, it was not discussed in sufficient detail, except in the masters and doctoral theses of Hui's various students [64, 65]. Comparison of the original UCS papers [53, 54] with the later review article [21] and monograph [55] clearly demonstrates an evolution in the understanding of the mathematics behind the UCS method over the course of the intervening years.

The original UCS publications, including the theses where the UCS derivation is described in detail, are based upon the idea that grid-point velocity should differ from fluid velocity only by a scalar factor h. For steady-state problems, this results in a formulation that varies constantly between Eulerian and Lagrangian coordinates [53]. For unsteady problems, the form of the grid is less clear. This difficulty can be avoided through the more recent formulation of UCS in terms of grid velocity components U, V, and W, as presented in later publications [21, 55].

When the motion of the computational grid is handled using grid velocity components, it is possible to define the computational coordinates η and ζ to be material coordinates [21]. This effectively creates a grid where fluid particles travel exclusively down streamtubes bounded by constant values of η and ζ, independent of the steadiness of the flow. This formulation also simplifies the schemes that are used to control grid motion. Whereas grid distortion can be controlled using both h and $\vec{U}$ schemes, this requires the solution of an additional partial differential equation for h [53, 54], while it requires only an ordinary differential equation for $\vec{U}$ [21].

I have chosen to derive these equations under the more recent formalism, as done in [21, 55]. This provides a unified, coherent description of the UCS transformation, that is consistent with the ideas and notation of more recent works in the subject, while also giving a higher level of detail that has been available in previously published works in this formalism.

4.2.1 Mathematical Background

Although simple coordinate transformations are ubiquitous in the mathematical sciences and engineering, the unsteady UCS coordinate transformation is far from simple. In order to discuss the UCS transformation, as well as its weak and strong forms, it is important to have a basic understanding of Einstein summation notation, the Levi-Civita tensor and Kronecker delta, n-dimensional Euclidean space, embedded manifolds with boundaries, and Stokes's theorem. As many of these concepts are unfamiliar to most engineers, it is important to review them briefly before presenting the UCS transformation in detail. The following discussion will draw heavily from books by Weintraub, Flanders, and Preston. [66–68]

4.2.1.1 Einstein summation (index) notation

The Einstein system of summation notation is a powerful form of shorthand for vector calculus. It is commonly taught in graduate courses in the mathematical sciences, but there are a few conventions that are less well-known, and a brief overview is helpful.

At its most basic, summation notation is a shorthand form of representing sums. It is especially common in the mathematics of tensors, where it replaces the usual matrix products, dot products, and so on, as in equation (4.1). This has the effect of dramatically reducing the complexity of tensor calculations.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} \Rightarrow \sum_{k=1}^{3} a_{ik} b_{kj} \tag{4.1}$$

In practice, many fields where summation notation is used are concerned principally with sums over the dimensionality of the problem. As a result, one can often neglect to explicitly write the summation symbol, and allow the reader to assume the appropriate upper and lower indices, as in equation (4.2). In problems which consider both three-dimensional space and four-dimensional space-time, a common convention is to use latin indices such as i, j, k to represent sums over

three-dimensional space ($i = 1, 2, 3$), while using greek indices such as μ, ν to represent sums over space-time, where the zero index represents the time dimension ($\mu = 0, 1, 2, 3$). As UCS occasionally requires both three- and four-dimensional sums, this is the convention that will be used here.

$$\sum_{i=1}^{n} a_i b_i \Rightarrow a_i b_i \tag{4.2}$$

There are two additional symbols commonly used in calculations which rely on summation notation. The first is the Kronecker delta δ_{ij}, which is equal to one when its two indices i, j are equal, and zero otherwise. This makes the Kronecker delta an index representation of the identity matrix. The second is the Levi-Civita tensor ϵ_{ijk}, which is a three-dimensional, antisymmetric tensor that is commonly used to represent vector cross-products. The Levi-Civita tensor has a value of one whenever its indices are an even permutation of $1, 2, 3$, a value of negative one whenever they are an odd permutation, and a value of zero everywhere else. This means that, for instance, $\epsilon_{ijk} = -\epsilon_{jik} = \epsilon_{kij}$. As a result, the cross product of two vectors can be expressed as in equation (4.3).

$$\vec{a} \times \vec{b} = \epsilon_{ijk} a_j b_k \tag{4.3}$$

4.2.1.2 Four-dimensional, Euclidean Spaces

The concept of three-dimensional space underlies nearly everything that is done in science and engineering. Unfortunately, this means that most researchers will avoid any forays into higher-dimensional spaces, because the mathematical environment is foreign to them. As the UCS transformation is inherently four-dimensional, it is necessary to briefly describe this environment, as well as the notation used to describe these spaces.

Four-dimensional, Euclidean space-time consists of the $\mathbb{R}^4$ vector space defined by all sets of real numbers (t, x, y, z), or equivalently (x_0, x_1, x_2, x_3). Defining the space in this way also suggests a global coordinate system for the space, with the origin at $(x_0, x_1, x_2, x_3) = (0, 0, 0, 0)$, and the usual unit vectors $\vec{x_0}, \vec{x_1}, \vec{x_2}, \vec{x_3}$. In the language of differential geometry, vectors are written differently in order to simplify the notation. These basis vectors are written in this form as derivatives:

$\frac{\partial}{\partial x^0}, \frac{\partial}{\partial x^1}, \frac{\partial}{\partial x^2}, \frac{\partial}{\partial x^3}$. This notation arises from the definition of vectors on manifolds, and it is also a helpful visual mnemonic for coordinate transforms.

Coordinate transformations are conceptually quite simple, when considered in this way, as the rules for a coordinate transformation are equivalent to an application of the derivative chain rule. That is, given an additional coordinate system which maps points in $\mathbb{R}^4$ to coordinates $(\xi_0, \xi_1, \xi_2, \xi_3)$, then one can transform from one to the other using only the partial derivatives, as in equation (4.4).

$$dx_\mu = \frac{\partial x_\mu}{\partial \xi_\nu} d\xi_\nu \tag{4.4}$$

The transformation for basis vectors is perfectly analogous:

$$\frac{\partial}{\partial x_\mu} = \frac{\partial \xi_\nu}{\partial x_\mu} \frac{\partial}{\partial \xi_\nu} \tag{4.5}$$

Here, it is possible to see the power of the visual mnemonic. In traditional matrix notation, the transformations are rather more complicated:

$$\begin{bmatrix} dx_0 \\ dx_1 \\ dx_2 \\ dx_3 \end{bmatrix} = \mathbf{A} \begin{bmatrix} d\xi_0 \\ d\xi_1 \\ d\xi_2 \\ d\xi_3 \end{bmatrix} \Rightarrow \begin{bmatrix} \frac{\partial}{\partial x_0} \\ \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \\ \frac{\partial}{\partial x_3} \end{bmatrix} = \left((\mathbf{A})^{-1} \right)^\top \begin{bmatrix} \frac{\partial}{\partial \xi_0} \\ \frac{\partial}{\partial \xi_1} \\ \frac{\partial}{\partial \xi_2} \\ \frac{\partial}{\partial \xi_3} \end{bmatrix} \tag{4.6}$$

In order to consider weak balance laws in four-dimensional space, it is important to understand the concepts of computational volumes and their integrals. A four-volume, as generally considered in the context of integration, consists of some subset of $\mathbb{R}^4$ that is bounded by a closed, oriented, three-dimensional manifold. In three dimensions, a volume might be a sphere, bounded by a two-dimensional surface manifold, or it might be a cube, bounded by the two-dimensional manifold composed of the six faces. The same concept applies in four dimensions, except that the boundary manifold is three-dimensional. As an example, one might consider the boundary surface defined by the four-dimensional integral boundaries in equation (4.7).

$$\int_0^1 \int_0^1 \int_0^1 \int_0^1 \left(1 - \sum_{\mu=0}^{3} x_\mu * *2\right) dx_0 dx_1 dx_2 dx_3 \tag{4.7}$$

The definition of surfaces in four-space is important, because there is an analogue to the three-dimensional divergence theorem that is used to derive weak form balance laws. The details of Stokes's theorem require an understanding of differential forms, and are beyond the scope of this overview, but the basic concept, that the integral of a vector field over some computational volume is equal to the flux of that field through the hypersurface that bounds that volume. This can be written explicitly for a conservation law integrated over the unit hypercube, as in equation (4.8).

$$\begin{aligned}\int_{\mathcal{V}} \left(\sum_{\mu=0}^{3} \frac{\partial F_\mu}{\partial x_\mu}\right) = \oint_{\partial V} \sum_{\mu=0}^{3} F_\mu dx_\mu = \\ \int_0^1 \int_0^1 \int_0^1 F_0|_0^1 dx_1 dx_2 dx_3 + \int_0^1 \int_0^1 \int_0^1 F_1|_0^1 dx_0 dx_2 dx_3 \\ + \int_0^1 \int_0^1 \int_0^1 F_2|_0^1 dx_0 dx_1 dx_3 + \int_0^1 \int_0^1 \int_0^1 F_3|_0^1 dx_0 dx_1 dx_2\end{aligned} \tag{4.8}$$

For more complete information about any of these topics, the reader is referred to the excellent works by [66–68].

4.2.2 The Unified Coordinate System

UCS is defined by the coordinate transformation:

$$\begin{bmatrix} dt \\ dx \\ dy \\ dz \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ U & A & L & P \\ V & B & M & Q \\ W & C & N & R \end{bmatrix} \cdot \begin{bmatrix} d\lambda \\ d\xi \\ d\eta \\ d\zeta \end{bmatrix} \tag{4.9}$$

Equation (4.9) may be written more succinctly as $dx_\alpha = \frac{\partial x_\alpha}{\partial \xi^\beta} d\xi_\beta$, which naturally leads to the corresponding transformation for vectors and derivatives which can be written as $\frac{\partial}{\partial x^\alpha} = \frac{\partial \xi_\beta}{\partial x^\alpha} \frac{\partial}{\partial \xi^\beta}$,

or in expanded form as:

$$\begin{bmatrix} \frac{\partial}{\partial t} \\ \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} = \begin{bmatrix} 1 & -U_\xi & -U_\eta & -U_\zeta \\ 0 & \frac{MR-NQ}{J} & \frac{CQ-BR}{J} & \frac{BN-CM}{J} \\ 0 & \frac{NP-LR}{J} & \frac{AR-CP}{J} & \frac{CL-AN}{J} \\ 0 & \frac{LQ-MP}{J} & \frac{BP-AQ}{J} & \frac{AM-BL}{J} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial}{\partial \lambda} \\ \frac{\partial}{\partial \xi} \\ \frac{\partial}{\partial \eta} \\ \frac{\partial}{\partial \zeta} \end{bmatrix} \tag{4.10}$$

In Eq. 4.10, the following relations are defined:

$$\begin{aligned} U_{\xi_i} &\equiv (U,V,W) \cdot \nabla_{\vec{x}} \xi_i \\ \nabla_{\vec{x}} \xi &\equiv \tfrac{1}{J} \left(MR - NQ, NP - LR, LQ - MP\right) \\ \nabla_{\vec{x}} \eta &\equiv \tfrac{1}{J} \left(CQ - BR, AR - CP, BP - AQ\right) \\ \nabla_{\vec{x}} \zeta &\equiv \tfrac{1}{J} \left(BN - CM, CL - AN, AM - BL\right) \end{aligned} \tag{4.11}$$

$$J \equiv \begin{vmatrix} A & L & P \\ B & M & Q \\ C & N & R \end{vmatrix} = \varepsilon_{ijk} A_i L_j P_k$$

$$\vec{U} \equiv (U,V,W)\,; \;\; \vec{A} \equiv (A,B,C)\,; \;\; L \equiv (L,M,N)\,; \;\; \vec{P} \equiv (P,Q,R)$$

In particular, the above definitions provide the formulas for projection of a vector in global Cartesian coordinates onto the computational coordinate vectors, as well as providing a useful shorthand for later derivations, as discussed in section 4.2.1.

The UCS transformation in Eq. 4.9 is not complete as written, however. In order for the transformation to be well-behaved, it is necessary to impose additional conditions on the components of the transformation. In particular, it is necessary that partial derivatives commute:

$$\frac{\partial}{\partial \xi^\gamma} \left(\frac{\partial x_\alpha}{\partial \xi^\beta} \right) = \frac{\partial}{\partial \xi^\beta} \left(\frac{\partial x_\alpha}{\partial \xi^\gamma} \right) \tag{4.12}$$

Upon expansion, and using the notation of Eq. 4.9, this constraint is equivalent to the following compatibility conditions:

$$\begin{array}{lll}
\frac{\partial A}{\partial \lambda} = \frac{\partial U}{\partial \xi} & \frac{\partial L}{\partial \lambda} = \frac{\partial U}{\partial \eta} & \frac{\partial P}{\partial \lambda} = \frac{\partial U}{\partial \zeta} \\
\frac{\partial B}{\partial \lambda} = \frac{\partial V}{\partial \xi} & \frac{\partial M}{\partial \lambda} = \frac{\partial V}{\partial \eta} & \frac{\partial Q}{\partial \lambda} = \frac{\partial V}{\partial \zeta} \\
\frac{\partial C}{\partial \lambda} = \frac{\partial W}{\partial \xi} & \frac{\partial N}{\partial \lambda} = \frac{\partial W}{\partial \eta} & \frac{\partial R}{\partial \lambda} = \frac{\partial W}{\partial \zeta} \\
\frac{\partial A}{\partial \eta} = \frac{\partial L}{\partial \xi} & \frac{\partial A}{\partial \zeta} = \frac{\partial P}{\partial \xi} & \frac{\partial L}{\partial \zeta} = \frac{\partial P}{\partial \eta} \\
\frac{\partial B}{\partial \eta} = \frac{\partial M}{\partial \xi} & \frac{\partial B}{\partial \zeta} = \frac{\partial Q}{\partial \xi} & \frac{\partial M}{\partial \zeta} = \frac{\partial Q}{\partial \eta} \\
\frac{\partial C}{\partial \eta} = \frac{\partial N}{\partial \xi} & \frac{\partial C}{\partial \zeta} = \frac{\partial R}{\partial \xi} & \frac{\partial N}{\partial \zeta} = \frac{\partial R}{\partial \eta}
\end{array} \tag{4.13}$$

The conditions of Eq. 4.13 are not independent. In particular, if the nine purely spatial conditions are satisfied at some initial time λ, then it follows that the 9 conditions involving the temporal derivative $\frac{\partial}{\partial \lambda}$ are sufficient to ensure that the full compatibility conditions will be met at all other times. If these are compatibility conditions are not met, then the resulting coordinate transformation will not be single-valued. [21]

Furthermore, the parameters related to grid motion, U, V, W, remain unspecified. As a result, these may be chosen freely by the user, provided that they do not cause the coordinate transformation to become singular.

4.2.3 The Three-dimensional Euler Equations

The three-dimensional, unsteady, Euler equations are a system of five nonlinear equations defined on four-dimensional space-time ($\mathbb{R}^4$) by the fluxes:

$$F_0 \equiv \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e \end{bmatrix} ; \; F_1 \equiv \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uw \\ \rho u \left(e + p/\rho\right) \end{bmatrix} ; \; F_2 \equiv \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho vw \\ \rho v \left(e + p/\rho\right) \end{bmatrix} ; \; F_3 \equiv \begin{bmatrix} \rho w \\ \rho uw \\ \rho vw \\ \rho w^2 + p \\ \rho w \left(e + p/\rho\right) \end{bmatrix} \tag{4.14}$$

where ρ represents mass density, u,v, and w are the Cartesian velocity components, p is static pressure, e is the specific energy given by the ideal gas equation of state $e = \frac{1}{2}\left(u^2 + v^2 + w^2\right) +$

$\frac{p}{(\gamma-1)\rho}$, and γ is the ratio of specific heats of the fluid, which for calorically perfect, dry air can be considered to be $\frac{7}{5}$.

If time t is grouped with the spatial coordinates as $x_0 \equiv t$, then the Euler equations themselves can be written in weak conservation form as:

$$\oint_{\partial V} F_\mu = 0 \tag{4.15}$$

where ∂V signifies integration over the oriented boundary of the 4-volume V, just as in a three-dimensional flux integral.

If the fluxes are differentiable with respect to the coordinate directions, then it is possible to use Stokes' theorem to express this in strong conservation form as:

$$\frac{\partial}{\partial x^\mu} F_\mu = 0 \tag{4.16}$$

Under the UCS transformation defined in Eqs. 4.9 and 4.10, Eq. 4.16 becomes:

$$\left(\frac{\partial}{\partial \lambda} - U_i \frac{\partial \xi_j}{\partial x^i} \frac{\partial}{\partial \xi^j} \right) F_0 + \frac{\partial \xi_j}{\partial x^i} \frac{\partial}{\partial \xi^j} F_i = 0 \tag{4.17}$$

Multiplying this by the jacobian and applying the differential product rule yields the following identities:

$$\begin{aligned}
J \tfrac{\partial F_0}{\partial \lambda} &= \tfrac{\partial (J\, F_0)}{\partial \lambda} - F_0 \tfrac{\partial J}{\partial \lambda} \\
J U_i \tfrac{\partial \xi_j}{\partial x^i} \tfrac{\partial}{\partial \xi^j} F_0 &= \tfrac{\partial}{\partial \xi^j} \left(J\, U_i \tfrac{\partial \xi_j}{\partial x^i} F_0 \right) - F_0 \left(J\, U_i \tfrac{\partial \xi_j}{\partial x^j} \right) \\
J \tfrac{\partial \xi_j}{\partial x^i} \tfrac{\partial}{\partial \xi^j} F_i &= \tfrac{\partial}{\partial \xi^j} \left(J \tfrac{\partial \xi_j}{\partial x^i} F_i \right) - F_i \tfrac{\partial}{\partial \xi^j} \left(J \tfrac{\partial \xi_j}{\partial x^i} \right)
\end{aligned} \tag{4.18}$$

Using these identities, it is possible to show:

$$\frac{\partial (J\, F_0)}{\partial \lambda} - F_0 \frac{\partial J}{\partial \lambda} - \frac{\partial}{\partial \xi^j} \left(J\, U_i \frac{\partial \xi_j}{\partial x^i} F_0 \right) + F_0 \frac{\partial}{\partial \xi^j} \left(J\, U_i \frac{\partial \xi_j}{\partial x^i} \right) + \frac{\partial}{\partial \xi^i} \left(J \frac{\partial \xi_i}{\partial x^j} F_j \right) - F_j \frac{\partial}{\partial \xi^i} \left(J \frac{\partial \xi_i}{\partial x^j} \right) \tag{4.19}$$

Upon collecting terms, this becomes:

$$\frac{\partial\left(J\,F_0\right)}{\partial\lambda}+\frac{\partial}{\partial\xi^j}\left(J\frac{\partial\xi_j}{\partial x^i}\left(F_i-U_iF_0\right)\right)-F_0\left(\frac{\partial J}{\partial\lambda}-\frac{\partial}{\partial\xi^j}\left(J\,U_i\frac{\partial\xi_j}{\partial x^i}\right)\right)-F_i\frac{\partial}{\partial\xi^j}\left(J\frac{\partial\xi_j}{\partial x^i}\right) \tag{4.20}$$

The next step is to eliminate the remaining non-conservative terms. $\frac{\partial J}{\partial\lambda}-\frac{\partial}{\partial\xi^j}\left(J\,U_i\frac{\partial\xi_j}{\partial x^i}\right)$ represents conservation of volume, and is therefore identically zero, as shown by Thomas [69]. This leaves only the $F_i\frac{\partial}{\partial\xi^j}\left(J\frac{\partial\xi_j}{\partial x^i}\right)$ term to eliminate. This term also vanishes; the general idea is to use the compatibility conditions to show that the term is identically zero. The details are as follows:

It is possible to write the inverse of a 3x3 matrix such as $\frac{\partial\xi_j}{\partial x^i}$ in terms of vector cross products:

$$\mathbf{A}=\left[\frac{\partial x_i}{\partial\xi^1},\ \frac{\partial x_i}{\partial\xi^2},\ \frac{\partial x_i}{\partial\xi^3}\right]\Rightarrow\mathbf{A}^{-1}=\frac{1}{|\mathbf{A}|}\begin{bmatrix}\left(\frac{\partial x_i}{\partial\xi^2}\times\frac{\partial x_i}{\partial\xi^3}\right)^T\\ \left(\frac{\partial x_i}{\partial\xi^3}\times\frac{\partial x_i}{\partial\xi^1}\right)^T\\ \left(\frac{\partial x_i}{\partial\xi^1}\times\frac{\partial x_i}{\partial\xi^2}\right)^T\end{bmatrix} \tag{4.21}$$

This becomes:

$$\frac{\partial\xi_1}{\partial x^i}=\frac{1}{J}\varepsilon_{ijk}\frac{\partial x_j}{\partial\xi^2}\frac{\partial x_k}{\partial\xi^3};\quad\frac{\partial\xi_2}{\partial x^i}=\frac{1}{J}\varepsilon_{ijk}\frac{\partial x_j}{\partial\xi^3}\frac{\partial x_k}{\partial\xi^1};\quad\frac{\partial\xi_3}{\partial x^i}=\frac{1}{J}\varepsilon_{ijk}\frac{\partial x_j}{\partial\xi^1}\frac{\partial x_k}{\partial\xi^2} \tag{4.22}$$

It is then possible to rewrite:

$$\begin{aligned}&\tfrac{\partial}{\partial\xi^j}\left(J\tfrac{\partial\xi_j}{\partial x^i}\right)=\tfrac{\partial}{\partial\xi^1}\left(\varepsilon_{ijk}\tfrac{\partial x_j}{\partial\xi^2}\tfrac{\partial x_k}{\partial\xi^3}\right)+\tfrac{\partial}{\partial\xi^2}\left(\varepsilon_{ijk}\tfrac{\partial x_j}{\partial\xi^3}\tfrac{\partial x_k}{\partial\xi^1}\right)+\tfrac{\partial}{\partial\xi^3}\left(\varepsilon_{ijk}\tfrac{\partial x_j}{\partial\xi^1}\tfrac{\partial x_k}{\partial\xi^2}\right)\\&=\varepsilon_{ijk}\left(\tfrac{\partial^2x_j}{\partial\xi^1\partial\xi^2}\tfrac{\partial x_k}{\partial\xi^3}+\tfrac{\partial x_j}{\partial\xi^2}\tfrac{\partial^2x_k}{\partial\xi^1\partial\xi^3}\right)+\varepsilon_{ijk}\left(\tfrac{\partial^2x_j}{\partial\xi^2\partial\xi^3}\tfrac{\partial x_k}{\partial\xi^1}+\tfrac{\partial x_j}{\partial\xi^3}\tfrac{\partial^2x_k}{\partial\xi^2\partial\xi^1}\right)+\varepsilon_{ijk}\left(\tfrac{\partial^2x_j}{\partial\xi^3\partial\xi^1}\tfrac{\partial x_k}{\partial\xi^2}+\tfrac{\partial x_j}{\partial\xi^1}\tfrac{\partial^2x_k}{\partial\xi^3\partial\xi^2}\right)\end{aligned} \tag{4.23}$$

By exploiting the anti-symmetry of the Levi-Civita tensor ε_{ijk}, this term vanishes identically. The strong conservation form of the Euler equations in unified coordinates is therefore:

$$\frac{\partial\left(J\,F_0\right)}{\partial\lambda}+\frac{\partial}{\partial\xi^j}\left(J\frac{\partial\xi_j}{\partial x^i}\left(F_i-U_iF_0\right)\right)=0 \tag{4.24}$$

The corresponding weak form may be derived similarly.

$$\begin{aligned}&\textstyle\int_{\partial V}J\,F_0d\xi_1d\xi_2d\xi_3+\int_{\partial V}J\frac{\partial\xi_1}{\partial x^i}\left(F_i-U_iF_0\right)d\lambda d\xi_2d\xi_3\\&\textstyle+\int_{\partial V}J\frac{\partial\xi_2}{\partial x^i}\left(F_i-U_iF_0\right)d\lambda d\xi_1d\xi_3+\int_{\partial V}J\frac{\partial\xi_3}{\partial x^i}\left(F_i-U_iF_0\right)d\lambda d\xi_1d\xi_2=0\end{aligned} \tag{4.25}$$

Eq. 4.24 describes the behavior of the physical flow quantities under the UCS transformation. It is known from Eq. 4.13 that evolution equations also exist for the grid metric components. These may be handled by appending the time-dependent compatibility conditions to Eq. 4.24 to yield an expanded equation set.

Using Eq. 4.10, it is possible to define:

$$\frac{D\xi_i}{Dt} \equiv \left(\frac{\partial}{\partial t} + u_j \frac{\partial}{\partial x^j} \right) \xi_i = (u_j - U_j) \frac{\partial}{\partial x^j} \xi_i \tag{4.26}$$

This allows one to define new flux vectors for the Euler equations in the unified coordinate system:

$$F_0 \equiv \begin{bmatrix} \rho J \\ \rho J u \\ \rho J v \\ \rho J w \\ \rho J e \\ A \\ B \\ C \\ L \\ M \\ N \\ P \\ Q \\ R \end{bmatrix} ; \; F_1 \equiv \begin{bmatrix} \rho J \frac{D\xi}{Dt} \\ \rho J \frac{D\xi}{Dt} u + J \frac{\partial \xi}{\partial x} p \\ \rho J \frac{D\xi}{Dt} v + J \frac{\partial \xi}{\partial y} p \\ \rho J \frac{D\xi}{Dt} w + J \frac{\partial \xi}{\partial z} p \\ \rho J \frac{D\xi}{Dt} e + J u_j \frac{\partial \xi}{\partial x^j} p \\ -U \\ -V \\ -W \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} ; \; F_2 \equiv \begin{bmatrix} \rho J \frac{D\eta}{Dt} \\ \rho J \frac{D\eta}{Dt} u + J \frac{\partial \eta}{\partial x} p \\ \rho J \frac{D\eta}{Dt} v + J \frac{\partial \eta}{\partial y} p \\ \rho J \frac{D\eta}{Dt} w + J \frac{\partial \eta}{\partial z} p \\ \rho J \frac{D\eta}{Dt} e + J u_j \frac{\partial \eta}{\partial x^j} p \\ 0 \\ 0 \\ 0 \\ -U \\ -V \\ -W \\ 0 \\ 0 \\ 0 \end{bmatrix} ; \; F_3 \equiv \begin{bmatrix} \rho J \frac{D\zeta}{Dt} \\ \rho J \frac{D\zeta}{Dt} u + J \frac{\partial \zeta}{\partial x} p \\ \rho J \frac{D\zeta}{Dt} v + J \frac{\partial \zeta}{\partial y} p \\ \rho J \frac{D\zeta}{Dt} w + J \frac{\partial \zeta}{\partial z} p \\ \rho J \frac{D\zeta}{Dt} e + J u_j \frac{\partial \zeta}{\partial x^j} p \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -U \\ -V \\ -W \end{bmatrix} \tag{4.27}$$

Using these redefined flux vectors, along with the spatial compatibility conditions of Eq. 4.13, the strong and weak forms of the transformed Euler equations are given exactly as before:

$$\frac{\partial}{\partial \xi^\mu} F_\mu = 0 \tag{4.28}$$

$$\oint_{\partial V} F_\mu = 0 \tag{4.29}$$

4.3 Unsteady Grids and Grid Motion Control

The equations defined by the UCS flux vectors in Eq. 4.27 describe the evolution through time of fourteen quantities:

- The five physical quantities: mass; three components of momentum; energy.
- The nine spatial derivatives of the UCS transformation.

Eq. 4.27 contains additional variables beyond these, however. The grid velocity components U, V, and W are unspecified at this point, which allows the user to choose them in such a way as to yield a grid that has desirable properties, such as an orthogonal grid or one that conforms to fluid streamlines. The major constraint on such choices is only that the UCS transformation does not become singular, or that J remains positive for all time.

Some very useful grid properties can be obtained through judicious choice of grid velocity. One useful approach is to require that η and ζ shall be material coordinates:

$$\frac{D\eta}{Dt} = \frac{D\zeta}{Dt} = 0 \tag{4.30}$$

Using Eq. 4.10, this may be written:

$$(u_i - U_i)\frac{\partial \eta}{\partial x^i} = (u_i - U_i)\frac{\partial \zeta}{\partial x^i} = 0 \tag{4.31}$$

This requirement is equivalent to requiring that material particles moving with the fluid velocity u shall not cross lines of constant η and ζ.

Having thus constrained V and W, the grid will be forced to move along the same path as the computed motion of the fluid particles. This provides many of the most important advantages of the unified coordinate system, while leaving U unspecified to allow further control of the grid.

The simplest way to specify U is to set it equal to hu where h is some value between 0 and 1. The choice $h = 1$ is equivalent to requiring that ξ also be a material coordinate, returning the traditional Lagrangian coordinate system, but this choice has the unfortunate effect of rendering

the system of equations weakly hyperbolic, lacking a complete set of eigenvectors. As a result, if Lagrangian behavior is desired, it is preferable to set h to some constant value less than 1 instead. $h = 0.999$ works well for most purposes.

4.3.1 Grid-angle Preservation

For two-dimensional flows, it is possible to choose U such that the angle of intersection between lines of constant ξ and lines of constant η is preserved. That is:

$$\frac{\partial}{\partial \lambda}\left(\nabla_{\vec{x}}\xi \cdot \nabla_{\vec{x}}\eta\right) = 0 \tag{4.32}$$

For two dimensional flow, $C = N = P = Q = 0$ and $R = 1$, so Eq. 4.32 becomes:

$$\frac{\partial}{\partial \lambda}\left(\frac{AL + BM}{\sqrt{A^2 + B^2}\sqrt{L^2 + M^2}}\right) = 0 \tag{4.33}$$

By defining $S \equiv \sqrt{L^2 + M^2}$ and $T \equiv \sqrt{A^2 + B^2}$, one can write $\dot{S} = \frac{L\dot{L} + M\dot{M}}{S}$ and $\dot{T} = \frac{A\dot{A} + B\dot{B}}{T}$, which lead to $\frac{\partial}{\partial \lambda}(ST) = \left(L\dot{L} + M\dot{M}\right)\frac{T}{S} + \left(A\dot{A} + B\dot{B}\right)\frac{S}{T}$

Eq. 4.33 can then be rewritten:

$$\begin{gathered}
0 = \left(\dot{A}L + A\dot{L} + \dot{B}M + B\dot{M}\right)S^2T^2 - (AL + BM)\left[\left(L\dot{L} + M\dot{M}\right)T^2 + \left(A\dot{A} + B\dot{B}\right)S^2\right] \\
\Downarrow \\
\begin{aligned}
0 = {} & \dot{A}S^2\left[T^2L - A(AL + BM)\right] + \dot{B}S^2\left[T^2M - B(AL + BM)\right] \\
& + \dot{L}T^2\left[S^2A - L(AL + BM)\right] + \dot{M}T^2\left[S^2B - M(AL + BM)\right]
\end{aligned}
\end{gathered} \tag{4.34}$$

which leads to:

$$0 = S^2\left(B\dot{A} - A\dot{B}\right) + T^2\left(L\dot{M} - M\dot{L}\right) \tag{4.35}$$

At this point, the compatibility conditions of Eq. 4.13 can be applied:

$$\begin{aligned}
\dot{A} &= \tfrac{\partial U}{\partial \xi} \quad \dot{B} = \tfrac{\partial V}{\partial \xi} \\
\dot{L} &= \tfrac{\partial U}{\partial \eta} \quad \dot{M} = \tfrac{\partial V}{\partial \eta}
\end{aligned}$$

which leads to:

$$0 = S^2\left(B\frac{\partial U}{\partial \xi} - A\frac{\partial V}{\partial \xi}\right) + T^2\left(L\frac{\partial V}{\partial \eta} - M\frac{\partial U}{\partial \eta}\right) \tag{4.36}$$

At this point, one might choose to solve Eq. 4.36 for U in terms of grid velocity magnitude and flow velocity angle θ, defining:

$$\begin{gathered} U = \exp(g)\cos\theta;\;\; V = \exp(g)\sin\theta \\ \Downarrow \\ \frac{\partial U}{\partial \xi} = \exp(g)\,\frac{\partial g}{\partial \xi}\cos(\theta) - \exp(g)\sin(\theta)\,\frac{\partial \theta}{\partial \xi} \\ \frac{\partial V}{\partial \xi} = \exp(g)\,\frac{\partial g}{\partial \xi}\sin(\theta) + \exp(g)\cos(\theta)\,\frac{\partial \theta}{\partial \xi} \end{gathered} \tag{4.37}$$

This allows one to write Eq. 4.36 as:

$$\begin{gathered} 0 = S^2\left(A\sin(\theta) - B\cos(\theta)\right)\frac{\partial g}{\partial \xi} + T^2\left(M\cos(\theta) - L\sin(\theta)\right)\frac{\partial g}{\partial \eta} \\ +S^2\left(A\cos(\theta) + B\sin(\theta)\right)\frac{\partial \theta}{\partial \xi} - T^2\left(M\sin(\theta) + L\cos(\theta)\right)\frac{\partial \theta}{\partial \eta} \end{gathered} \tag{4.38}$$

Eq. 4.38 may then be solved using a variety of techniques.

Alternatively, one may return to Eq. 4.36, and derive instead an equation for U. Based on Eqs. 4.31 and 4.13, one can write:

$$\begin{gathered} V = v - \tfrac{B}{A}(u - U) \\ \Downarrow \\ V_{\xi^i} = v_{\xi^i} + \tfrac{B_{\xi^i}A + BA_{\xi^i}}{A^2}(U - u) + \tfrac{B}{A}\left(U_{\xi^i} - u_{\xi^i}\right) \end{gathered} \tag{4.39}$$

Substituting this into Eq. 4.36 yields:

$$\begin{gathered} 0 = -S^2BU_\xi + S^2A\left(v_\xi + \tfrac{B_\xi A + BA_\xi}{A^2}(U - u) + \tfrac{B}{A}(U_\xi - u_\xi)\right) \\ +T^2MU_\eta - T^2L\left(v_\eta + \tfrac{B_\eta A + BA_\eta}{A^2}(U - u) + \tfrac{B}{A}(U_\eta - u_\eta)\right) \end{gathered} \tag{4.40}$$

This equation is simplified by collecting terms of U, U_ξ and U_η, whereupon the U_ξ terms vanish, leaving:

$$\begin{gathered} 0 = T^2\left(M - \tfrac{BL}{A}\right)U_\eta + \left(S^2\tfrac{B_\xi A + BA_\xi}{A} - T^2L\tfrac{B_\eta A + BA_\eta}{A^2}\right)(U - u) \\ +S^2A\left(v_\xi - \tfrac{B}{A}u_\xi\right) - T^2L\left(v_\eta - \tfrac{B}{A}u_\eta\right) \end{gathered} \tag{4.41}$$

This can finally be written:

$$\begin{aligned} 0 = U_\eta + \tfrac{S^2 A}{T^2 J}\left(A v_\xi - B u_\xi\right) - \tfrac{L}{J}\left(A v_\eta - B u_\eta\right) \\ + \left(\tfrac{S^2}{T^2 J}\left(B_\xi A + B A_\xi\right) - \tfrac{L}{A J}\left(B_\eta A + B A_\eta\right)\right)\left(U - u\right) \end{aligned} \tag{4.42}$$

In some situations, this form may be more advantageous. In particular, equation (4.42) is an ordinary differential equation in the cross-stream direction η, rather than a partial differential equation in ξ and η, which makes it simpler to solve numerically.

4.3.2 Jacobian Preservation

For general three-dimensional flows, no orthogonal grid exists, and so grid-angle preservation is no longer preferred [21]. One alternative approach is to preserve the jacobian of the transformation:

$$\frac{\partial J}{\partial \lambda} = 0 \tag{4.43}$$

It is helpful to define a few useful variables:

$$\begin{bmatrix} \vec{J}_1 \\ \vec{J}_2 \\ \vec{J}_3 \end{bmatrix} \equiv J \begin{bmatrix} \nabla_{\vec{x}}\xi \\ \nabla_{\vec{x}}\eta \\ \nabla_{\vec{x}}\zeta \end{bmatrix} \Rightarrow \begin{bmatrix} J_{1i} \\ J_{2i} \\ J_{3i} \end{bmatrix} = \varepsilon_{ijk} \begin{bmatrix} L_j P_k \\ P_j A_k \\ A_j L_k \end{bmatrix} \tag{4.44}$$

and

$$\begin{gathered} \vec{A} \equiv (A, B, C)\,; \;\; \vec{L} \equiv (L, M, N)\,; \;\; \vec{P} \equiv (P, Q, R) \\ \Downarrow \\ J = \vec{A} \cdot \vec{J}_1 = \vec{L} \cdot \vec{J}_2 = \vec{P} \cdot \vec{J}_3 \end{gathered} \tag{4.45}$$

Rewriting the material coordinates from Eq. 4.31:

$$\begin{aligned} J \tfrac{D\eta}{Dt} = \varepsilon_{ijk}\left(u_i - U_i\right) P_j A_k = 0 \\ J \tfrac{D\zeta}{Dt} = \varepsilon_{ijk}\left(u_i - U_i\right) A_j L_k = 0 \end{aligned} \tag{4.46}$$

and using the compatibility conditions from Eq. 4.13, the equation for preservation of the jacobian can be written as:

$$\frac{\partial}{\partial \lambda}\left(\varepsilon_{ijk} A_i L_j P_k\right) = 0 \tag{4.47}$$

which leads to

$$\frac{\partial U_i}{\partial \xi}\left(\varepsilon_{ijk} L_j P_k\right) + \frac{\partial U_i}{\partial \eta}\left(\varepsilon_{ijk} A_k P_j\right) + \frac{\partial U_i}{\partial \zeta}\left(\varepsilon_{ijk} A_j L_k\right) \tag{4.48}$$

Differentiating Eq. 4.46 results in

$$\begin{aligned} \tfrac{\partial u_i}{\partial \eta}\varepsilon_{ijk} P_j A_k + \left(u_i - U_i\right)\tfrac{\partial}{\partial \eta}\left(\varepsilon_{ijk} P_j A_k\right) = 0 \\ \tfrac{\partial u_i}{\partial \zeta}\varepsilon_{ijk} A_j L_k + \left(u_i - U_i\right)\tfrac{\partial}{\partial \zeta}\left(\varepsilon_{ijk} A_j L_k\right) = 0 \end{aligned} \tag{4.49}$$

Combining all of these results in an equation for the jacobian:

$$\frac{\partial U_i}{\partial \xi} J_{1i} + \frac{\partial u_i}{\partial \eta} J_{2i} + \frac{\partial u_i}{\partial \zeta} J_{3i} + \left(u_i - U_i\right)\left(\frac{\partial}{\partial \eta} J_{2i} + \frac{\partial}{\partial \zeta} J_{3i}\right) \tag{4.50}$$

Jacobian-preserving grid motion also results in an ODE. In contrast with equation (4.42), however, the jacobian-preserving equation is an ODE in the streamwise direction, ξ.

4.3.3 Comments on Grid Motion Control

Both of these forms of grid motion control have their uses, and they all provide an acceptable answer to the problem of how to control the grid distortion inherent in Lagrangian coordinate systems, but they also suffer from one glaring limitation. They do not have any knowledge of the physical location of boundaries, and are completely defined by their action on the components of the flow velocity and grid metric. It will be seen later (Sec. 4.8.3) that this can cause unphysical behavior of the grid near boundaries. A better approach would control this behavior, perhaps by incorporating a "grid pressure" term that tended to equalize grid-point distribution over time.

4.4 Solving the UCS Equations

The equations defined by Eqs. 4.25 and 4.27 are complex, perhaps hopelessly so when the equations of grid motion control in Eqs. 4.26 are included. It is necessary to make simplifying approximations in order to compute practical solutions. The first and most essential approximation is to treat grid velocity as a parameter that is set at each time step, rather than an integral part of the evolution. The second is to decouple the equations that control the evolution of the grid metric

from those that govern the fluid flow. The net result of these two approximations is three sets of equations:

(1) Five fluid evolution equations, equivalent to the standard Euler equations in curvilinear coordinates, except for use of relative velocity components.

(2) Nine simple geometric evolution equations.

(3) One spatial differential equation to solve for U, and formulas for computing V and W.

Each of these sets of equations is solved independently, taking the results of the other sets as constant parameters. This technique was dubbed the time-step-Eulerian (TSE) approximation by Hui [53]. By solving the equations in this way, the problem is reduced to that of solving the Euler equations in curvilinear coordinates, along with a simple grid update step.

One of the principal challenges and benefits in the use of UCS to model physical systems, is that the computational grid upon which the equations are solved is steadily moving through the physical space. Therefore, in order to use UCS to simulate fluid flow in a physical system, a form of computational "window" must be defined. The grid is allowed to move forward from the upstream edge of this window, and then it is no longer used after it has moved beyond the downstream edge. The most common way to accomplish this is by creating Cartesian grid points at the upstream boundary whenever the upstream edge of the grid has moved a downstream distance of Δx, and then removing points as they pass out of the region. This approach is extremely beneficial, because it allows the user to forego the normally tedious grid-generation process. It also provides unique computational challenges for UCS codes, which must make some provision for efficiently handling the unsteadiness in flow data structures.

In this work, an exact Godunov method is used to solve the fluid evolution equations. Godunov methods work by treating nodes as computational volumes of constant state, and the boundaries between nodes as Riemann problems which can be solved exactly. This exact solution is then used to compute the fluid state at the boundary interface, which is then used to compute the

boundary flux. Applying the Godunov method to multidimensional problems is straightforward, but does require some special considerations.

4.4.1 Multidimensional Considerations

There are many algorithms available for solving the Euler equations that can be equally applied to the UCS system, but many of these are inherently one-dimensional. It is therefore necessary to extend these algorithms to handle multidimensional flows. The two simplest methods are dimensional splitting approximations and finite-volume methods. Finite-element methods are also widely used, but will not be discussed here.

The simplest form of dimensional splitting approximation consists of breaking the various fluxes apart, and solving the resulting one-dimensional equations in sequential steps, as:

$$Q^{\xi_0^0+\Delta\xi^0} = \mathcal{L}^{\Delta\xi^0} Q^{\xi_0^0} \approx \prod_{i=1}^{n} \mathcal{L}_{\xi^i}^{\Delta\xi^0} Q^{\xi_0^0} \tag{4.51}$$

for some set of variables Q and some linear operator $\mathcal{L}^{\Delta\xi^0}$ that advances the variables in ξ^0 from ξ_0^0 by $\Delta\xi^0$, and $\mathcal{L}_{\xi^i}$ is derived from $\mathcal{L}$ by ignoring all fluxes except those corresponding to the ξ^0 and ξ^i dimensions. The Godunov splitting described above is first-order accurate. Other splitting algorithms are possible, which improve accuracy through the use of fractional time steps, but these will not be considered here.

The finite-volume approach is more easily expressed in terms of integral flux equations. In this approach, for a given computational volume, the flux through each boundary face is computed independently, and these fluxes are combined to advance the state within the volume.

4.5 The Riemann Problem

The Godunov method for solving partial differential equations is a nonlinear, monotonic method that has proved very successful in the solution of compressible fluid dynamic flows. It represents a flow field as a collection of adjacent cells, each with a state approximated by its

average over the whole cell. That is, for a cell of volume V, the flow state $\mathbf{W}$ would be given by

$$\overline{\mathbf{W}} = \frac{1}{V}\int_V \mathbf{W}\left(\xi,\ \eta,\ \zeta\right) dV \tag{4.52}$$

The boundaries between cells are naturally regions where the approximated flow state is discontinuous, and the Godunov method treats these discontinuities as actual flow features which can be solved exactly to find the intercellular fluxes. For more information about the Godunov method and compressible, computational fluid dynamics in general, the reader is referred to the excellent book by Toro [35].

4.5.1 Transformation to grid components

Godunov's method requires the solution of various one-dimensional Riemann problems, where the initial condition is given by the discontinuous interface between two adjoining cells. Therefore, in order to apply the Godunov method to solve the equations defined by the UCS fluxes in Eq. 4.27, it is necessary to express the velocity vector in terms of components that are normal and tangential to the cell interface. If we define the vectors $\hat{e}_1,\ \hat{e}_2,\ \hat{e}_3$ as an orthonormal basis where $\hat{e}_1$ is normal to the cell interface, then we may write

$$u_i = \frac{\partial x_i}{\partial \hat{e}^j}\omega_j$$

Under this transformation, the tangential derivatives vanish at cell interfaces, and the remaining unsteady, one-dimensional equations become [21]:

$$\frac{\partial F'_0}{\partial \lambda} + \frac{\partial F'_1}{\partial \xi} = S_0$$

$$F'_0 = \begin{bmatrix} \rho J \\ \rho J\omega \\ \rho J\tau_1 \\ \rho J\tau_2 \\ \rho Je \\ A \\ B \\ C \end{bmatrix}; \quad F'_1 = \begin{bmatrix} \rho S(\omega - \Omega) \\ \rho S(\omega - \Omega)\omega + p \\ \rho S(\omega - \Omega)\tau_1 \\ \rho S(\omega - \Omega)\tau_2 \\ \rho S(\omega - \Omega)e + \omega p \\ -U \\ -V \\ -W \end{bmatrix}; \quad S_0 = \begin{bmatrix} 0 \\ s_{11} \\ s_{12} \\ s_{13} \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{4.53}$$

$$\vec{s}_{1i} = (f_1,\ f_2,\ f_3) \cdot \frac{\partial \hat{e}_i}{\partial \xi}$$

$$f_i = \rho J(\omega - \Omega)u_i + pJ\frac{\partial \xi}{\partial x^i}$$

For the sake of brevity, we examine only the first dimensional case, though the rest are similar. The non-conservative source terms are a result of the non-inertial velocity components, and are analogous to the centrifugal and Coriolis force terms encountered in the physics of rotating coordinate systems. These source terms are non-zero only at the cell boundaries.

There is no general solution to Eq. 4.53 unless $\frac{\partial \xi}{\partial x^i}$ is constant across the cell boundary. It is therefore necessary to choose some suitable average to be applied at the cell boundary:

$$\overline{\frac{\partial \hat{e}_1}{\partial x^i}} = \frac{\left(\left(\frac{\partial \xi}{\partial x^i}\right)_L + \left(\frac{\partial \xi}{\partial x^i}\right)_R\right)}{\left\|\left(\frac{\partial \xi}{\partial x^i}\right)_L + \left(\frac{\partial \xi}{\partial x^i}\right)_R\right\|} \tag{4.54}$$

Both flow and grid velocity components are transformed using Eq. 4.54. Once transformed, the normal grid velocity component must also be averaged: $\overline{\Omega} = \frac{1}{2}(\Omega_L + \Omega_R)$. This averaged value is used both in the solution of the Riemann problem and in the computation of intercellular fluxes.

4.5.2 Eigensystem of the grid-aligned equations

In order to compute the solution to the one-dimensional Riemann problem defined by Eq. 4.53 subject to the initial conditions:

$$\mathbf{W} = \begin{cases} \mathbf{W}_L;\ \xi < 0 \\ \mathbf{W}_R;\ \xi \geq 0 \end{cases} ;\ \mathbf{W}_i = \begin{bmatrix} p \\ \rho \\ \omega \\ \tau_1 \\ \tau_2 \\ A \\ B \\ C \\ L \\ M \\ N \\ P \\ Q \\ R \end{bmatrix}$$

one must first compute the eigenvalues and eigenvectors of the system. The first step is to compute the derivative matrices (under the time-step-Eulerian approximation, where grid derivatives are

assumed to be constant):

$$\frac{\partial F'_o}{\partial W} = \begin{bmatrix} J & 0 & 0 & 0 & 0 & \rho J\frac{\partial \xi}{\partial x} & \rho J\frac{\partial \xi}{\partial y} & \rho J\frac{\partial \xi}{\partial z} \\ J\omega & 0 & \rho J & 0 & 0 & \rho J\frac{\partial \xi}{\partial x}\omega & \rho J\frac{\partial \xi}{\partial y}\omega & \rho J\frac{\partial \xi}{\partial z}\omega \\ J\tau_1 & 0 & 0 & \rho J & 0 & \rho J\frac{\partial \xi}{\partial x}\tau_1 & \rho J\frac{\partial \xi}{\partial y}\tau_1 & \rho J\frac{\partial \xi}{\partial z}\tau_1 \\ J\tau_2 & 0 & 0 & 0 & \rho J & \rho J\frac{\partial \xi}{\partial x}\tau_2 & \rho J\frac{\partial \xi}{\partial y}\tau_2 & \rho J\frac{\partial \xi}{\partial z}\tau_2 \\ J\left(e+\frac{p}{(\gamma-1)\rho}\right) & \frac{J}{\gamma-1} & \rho J\omega & \rho J\tau_1 & \rho J\tau_2 & \rho\frac{\partial \xi}{\partial x}e & \rho\frac{\partial \xi}{\partial y}e & \rho\frac{\partial \xi}{\partial z}e \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.55}$$

$$\frac{\partial F'_1}{\partial W} = \begin{bmatrix} S(\omega-\Omega) & 0 & S\rho & 0 & 0 & 0 & 0 & 0 \\ S\omega(\omega-\Omega) & S & S\rho(2\omega-\Omega) & 0 & 0 & 0 & 0 & 0 \\ S\tau_1(\omega-\Omega) & 0 & S\rho\tau_1 & S\rho(\omega-\Omega) & 0 & 0 & 0 & 0 \\ S\tau_2(\omega-\Omega) & 0 & S\rho\tau_2 & 0 & S\rho(\omega-\Omega) & 0 & 0 & 0 \\ S(\omega-\Omega)\left(e-\frac{p}{(\gamma-1)\rho}\right) & \frac{S(\gamma\omega-\Omega)}{\gamma-1} & S(p+\rho(e+\omega(\omega-\Omega))) & S\rho(\omega-\Omega)\tau_1 & S\rho(\omega-\Omega)\tau_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.56}$$

where $S \equiv \delta_{ij}\frac{\partial \xi}{\partial x^i}\frac{\partial \xi}{\partial x^j}$

The eigenvalues can be computed as solutions of the equation: [21]

$$\begin{aligned} &\left(\sigma\frac{\partial F'_0}{\partial \mathbf{W}} - \frac{\partial F'_1}{\partial \mathbf{W}}\right) = 0 \\ &\Downarrow \\ &\frac{\rho^3\sigma^3\hat{\sigma}^3\left(S^2\frac{\gamma p}{\rho}-\hat{\sigma}^2\right)}{\gamma-1} \\ &\text{where } \hat{\sigma} \equiv J\sigma - S(\omega-\Omega) \\ &\Downarrow \\ &\sigma_1 = 0(\text{multiplicity of 3}) \\ &\sigma_2 = \frac{S}{J}(\omega-\Omega)(\text{multiplicity of 3}) \\ &\sigma_\pm = \frac{S}{J}\left(\omega-\Omega\pm\sqrt{\frac{\gamma p}{\rho}}\right) \end{aligned} \tag{4.57}$$

The corresponding eigenvectors are:

$$
\begin{aligned}
r_1 &= \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} ; \quad r_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} ; \quad r_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \\
r_4 &= \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} ; \quad r_5 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} ; \quad r_6 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \\
r_+ &= \begin{bmatrix} 1 \\ \frac{1}{a^2} \\ -\frac{1}{a\rho} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} ; \quad r_- = \begin{bmatrix} 1 \\ \frac{1}{a^2} \\ \frac{1}{a\rho} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}
\end{aligned}
\tag{4.58}
$$

From these, it is possible to compute both the Riemann invariants, which govern flow across smooth waves, and the Rankine-Hugoniot relations, which govern flow across shocks.

4.5.3 Riemann invariants and rarefaction wave relations

The generalized Riemann invariants are relations that hold true across smooth waves, and are given by the relations [35]:

$$\begin{gathered}\frac{dw_1}{{r_i}^1} = \frac{dw_2}{{r_i}^2} = \cdots = \frac{dw_m}{{r_i}^m} \\ \Downarrow \\ \frac{\partial\rho}{\partial p} = \frac{1}{a^2};\ \frac{\partial\omega}{\partial p} = \frac{\pm 1}{\rho a};\ \frac{\partial\tau_1}{\partial p} = 0;\ \frac{\partial\tau_2}{\partial p} = 0;\end{gathered} \tag{4.59}$$

Relations for flow quantities may be derived based on an upstream state given by p_0, ρ_0, ω_0, τ_{10}, τ_{20}. Beginning with ρ, it can be shown:

$$\begin{gathered}\frac{\partial\rho}{\partial p} = \frac{\rho}{\gamma p} \Rightarrow \frac{\partial\rho/\partial p}{\rho} = \frac{1}{\gamma p} \Rightarrow \ln(\rho) = \frac{\ln(p)}{\gamma} + C_1 \Rightarrow \rho = C_1 p^{1/\gamma} \Rightarrow \rho = \rho_0 \alpha^{\frac{1}{\gamma}} \\ \alpha \equiv \frac{p}{p_0}\end{gathered} \tag{4.60}$$

Normal velocity may then be similarly derived:

$$\begin{gathered}\frac{\partial\omega}{\partial p} = \pm\frac{1}{\rho a} = \pm\left(a_0\rho_0 p_0 \alpha^{\frac{\gamma+1}{2\gamma}}\right)^{-1} \Rightarrow \frac{\partial\omega}{\partial\alpha} = \pm\frac{a_0}{\gamma}\alpha^{\frac{-(\gamma+1)}{2\gamma}} \Rightarrow \omega = \pm\frac{2\gamma}{\gamma-1}\frac{a_0}{\gamma}\alpha^{\frac{\gamma-1}{2\gamma}} + C_1 \\ \Downarrow \\ \omega = \omega_0 \pm \frac{2a_0}{\gamma-1}\left(\alpha^{\frac{\gamma-1}{2\gamma}} - 1\right)\end{gathered} \tag{4.61}$$

The tangential velocities are simple:

$$\frac{\partial\tau_1}{\partial p} = \frac{\partial\tau_2}{\partial p} = 0 \tag{4.62}$$

If the state downstream of the rarefaction wave is denoted by p_*, ρ_*, ω_*, τ_{1*}, τ_{2*}, then the rarefaction head and tail speeds are also known, given by the eigenvalues from Eq. 4.57:

$$S_H = \frac{S}{J}\left(\omega_0 - \bar{\Omega} \pm a_0\right) \tag{4.63}$$

$$S_T = \frac{S}{J}\left(\omega_* - \bar{\Omega} \pm a_*\right) \tag{4.64}$$

It is finally necessary to compute pressures for points within the rarefaction wave. The slope of a characteristic for a rarefaction wave is:

$$\frac{\xi}{\lambda} = \frac{S}{J}\left[(\omega - \Omega) \pm a\right] \Rightarrow \omega = \frac{J}{S}\left(\frac{\xi}{\lambda}\right) \mp a + \Omega \tag{4.65}$$

Eqs. 4.61 and 4.65 can be combined to solve for the pressure ratio $\alpha \equiv \frac{p}{p_0}$:

$$\begin{gathered}
\frac{J}{S}\left(\frac{\xi}{\lambda}\right) \mp a + \Omega = \omega_0 \pm \frac{2a_0}{\gamma-1}\left(\alpha^{\frac{\gamma-1}{2\gamma}} - 1\right) \Rightarrow (\omega_0 - \Omega) \pm \frac{\gamma+1}{\gamma-1} \mp \frac{2a_0}{\gamma-1} - \frac{J}{S}\frac{\xi}{\lambda} = 0 \\
\Downarrow \\
\pm\left(\omega_0 - \Omega - \frac{J}{S}\frac{\xi}{\lambda}\right) + \frac{\gamma+1}{\gamma-1}a - \frac{2a_0}{\gamma-1} = 0 \Rightarrow \frac{2}{\gamma+1} \mp \frac{\gamma-1}{a_0(\gamma+1)}\left(\omega_0 - \Omega - \frac{J}{S}\frac{\xi}{\lambda}\right) = \alpha^{\frac{\gamma-1}{2\gamma}} \\
\Downarrow \\
\alpha = \left[\frac{2}{\gamma+1} \mp \frac{\gamma-1}{a_0(\gamma+1)}\left(\omega_0 - \Omega - \frac{J}{S}\frac{\xi}{\lambda}\right)\right]^{\frac{2\gamma}{\gamma-1}}
\end{gathered} \tag{4.66}$$

4.5.4 The Rankine-Hugoniot conditions and shock wave relations

The Riemann invariants do not hold across discontinuous waves such as shocks. For such waves, the Rankine-Hugoniot conditions must be used [35]:

$$F_{1R} - F_{1L} = S\left(F_{0R} - F_{0L}\right)$$

The derivation is somewhat tedious, but straightforward. The general idea is to use a Galilean velocity transformation to a frame where the shock speed is zero, and then solve the left-hand-side to find the relations between flow variables and the shock speed. This yields the relations:

$$\begin{aligned}
\rho &= \rho_0 \tfrac{\alpha(\gamma+1)+(\gamma-1)}{\alpha(\gamma-1)+(\gamma+1)} \\
\omega &= \omega_0 \pm \frac{(\alpha-1)a_0}{\sqrt{\frac{1}{2}\gamma[(\gamma+1)\alpha+(\gamma-1)]}} \\
\tau_1 &= \tau_{10} \\
\tau_2 &= \tau_{20} \\
S &= \tfrac{S}{J_0}\left[\omega_0 - \Omega \pm a_0\sqrt{\tfrac{\gamma+1}{2\gamma}(\alpha-1)+1}\right]
\end{aligned} \tag{4.67}$$

4.5.5 Slip lines

The third type of wave is the linearly degenerate slip line. This discontinuous wave moves at the normal speed of the fluid, and pressure and normal velocity are constant across the wave while density and tangential velocity may jump discontinuously.

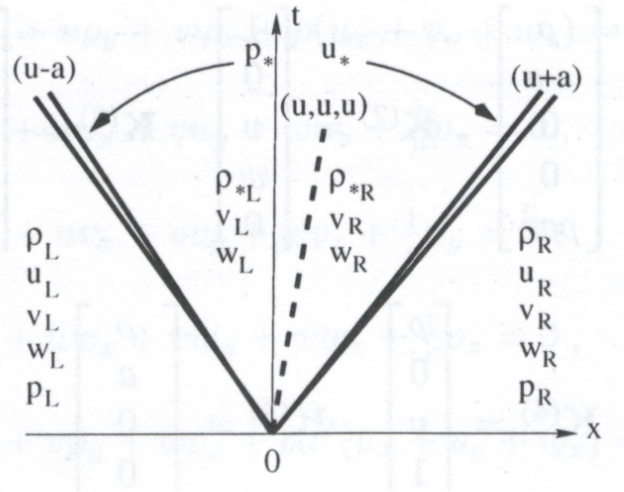


Figure 4.4: General Structure of the 1D, Unsteady Riemann Problem. The unsteady, one-dimensional Riemann problem for the Euler equations is given by two discontinuous states, separated from one another at an initial time t_0, and allowed to interact. For the Euler Equations, this interaction results in three non-linear waves which propagate outward from the initial point of discontinuity. The middle wave is always a slip line, a linearly degenerate wave across which pressure and velocity normal to the wave are constant, but across which density and tangential velocity components may jump discontinuously. The two outer waves are true nonlinear waves, and may be either shocks or expansion waves, depending on the specific character of the problem. [35]

4.5.6 The one-dimensional Riemann problem in the unified coordinates

The boundary between two adjacent cells can be represented as a one-dimensional Riemann problem, as in figure 4.4. The Riemann problem consists of 3 waves: a central, linearly degenerate, slip line, across which pressure and normal velocity are constant while density and tangential velocity may jump discontinuously, and two nonlinear waves which may be either rarefaction waves or shocks, and across which Using Eqs. 4.60, 4.61, and 4.67, it is possible to define a function of pressure representing the jump in normal velocity across the central slip line:

$$f\left(p_*\right) \equiv \omega\left(\mathbf{W}_R\right) - \omega\left(\mathbf{W}_L\right) = 0 \tag{4.68}$$

where $\omega\left(\mathbf{W}_i\right)$ represents the normal velocity computed across the i^{th} wave given a central pressure p_*. This yields a nonlinear equation that can be solved for the pressure between the two nonlinear waves. From this, the rest of the flow variables can be computed directly. The solution of this nonlinear equation for pressure can be computed by iteration, and is the most computationally expensive step in the traditional Godunov method. The present work uses a Newton-Rhapson solver for this purpose. Approximate Riemann solvers, which do not depend on iterative solution schemes, offer substantial performance gains, but they are not discussed here.

4.5.7 Spatial accuracy and boundary interpolation

The Godunov method is inherently first-order accurate, but it is possible to boost the order of spatial accuracy using MUSCL interpolation [21, 35, 53] to reconstruct the left and right boundary states used in the Riemann problem. In particular, for the boundary between the cells i and $i+1$, and for flow variable w, we have:

$$\begin{aligned}
&w_R = w_{i+1} - \tfrac{1}{2}\left(w_{i+2} - w_{i+1}\right)\phi\left(\tfrac{w_{i+1}-w_i}{w_{i+2}-w_{i+1}}\right)\\
&w_L = w_i + \tfrac{1}{2}\left(w_i - w_{i-1}\right)\phi\left(\tfrac{w_{i+1}-w_i}{w_i-w_{i-1}}\right)\\
&\text{where } \phi \text{ is the minmod limiter given by :}\\
&\phi\left(w\right) = \max\left(0, \min\left(1, w\right)\right)
\end{aligned} \tag{4.69}$$

4.6 Algorithms

Hui [53] uses a dimensional splitting technique to solve the Euler equations, as follows:

4.6.1 Dimensional Splitting

- For each coordinate direction n:
  - For each cell i, j, k:
    - For each interface $+$, $-$:
      - Apply MUSCL reconstruction using Eq. 4.69.
      - Transform flow and grid velocity to normal and tangential components:

$$\frac{\partial \hat{e}_{+i}}{\partial x^j} = \frac{\left(\frac{\partial x_i}{\partial \xi^j}\right)^{-1} + \left(\frac{\partial x_{i+1}}{\partial \xi^j}\right)^{-1}}{\left\| \left(\frac{\partial x_i}{\partial \xi^j}\right)^{-1} + \left(\frac{\partial x_{i+1}}{\partial \xi^j}\right)^{-1} \right\|}; \quad \frac{\partial \hat{e}_{-i}}{\partial x^j} = \frac{\left(\frac{\partial x_{i-1}}{\partial \xi^j}\right)^{-1} + \left(\frac{\partial x_i}{\partial \xi^j}\right)^{-1}}{\left\| \left(\frac{\partial x_{i-1}}{\partial \xi^j}\right)^{-1} + \left(\frac{\partial x_i}{\partial \xi^j}\right)^{-1} \right\|}$$

      - Solve the Riemann problems as described in section 4.5 to find the flow variables at each interface:

$$p_{\pm},\ \rho_{\pm},\ \omega_{\pm},\ \tau_{1\pm},\ \tau_{2\pm}$$

      - Transform interface velocity back to Cartesian components:

$$\frac{\partial x_{+i}}{\partial \hat{e}^j} = \left(\frac{\partial \hat{e}_{+i}}{\partial x^j}\right)^{-1}$$

    - Update coordinate-appropriate grid metric components:

$$\left(\frac{\partial x_l}{\partial \xi^n}\right)_i = \left(\frac{\partial x_l}{\partial \xi^n}\right)_i + \int \left(\frac{\Omega_l}{\omega_l}\right)_i (u_+ - u_-)\, dt$$

    - Compute interface fluxes using interface flow variables and central metric variables, e.g.: $F_{1\rho+} = \rho_+ \sqrt{{L_i}^2 + {M_i}^2}\,(\omega_+ - \Omega_i)$
    - Compute new conserved quantities F_0 using Eqs. 4.16 or 4.15, and updated metric components.
    - Update conserved variables using (e.g.):

$$\int {F_0}^{t_0+\Delta t} dxdydz - \int {F_0}^{t_0} dxdydz + \oint F_n \partial V_n$$
$$\Downarrow$$
$${F_0}^{t_0+\Delta t} = {F_0}^{t_0} - \tfrac{\Delta t}{\Delta \xi_n} F_n$$

    - Convert updated conserved variables to updated primitive variables.

The dimensionally split algorithm in section 4.6 suffers from two major drawbacks. First, it is difficult to choose adaptive time steps accurately. In the Godunov method, the temporal stability condition is dependent on the maximum wave speed present in the problem. This is known only after the solution of all the Riemann problems at all cell interfaces, so it is impossible to compute directly for dimensional splitting algorithms.

Second, and more importantly, the manner in which fluxes are computed using cell-specific metric components makes it impossible to enforce strong conservation in the algorithm. A better approach would be to rather implement a finite-volume algorithm, as follows:

4.6.2 Finite Volume

- Compute all cell interface fluxes
  - For each interface
    - Perform MUSCL interpolation if applicable.
    - Transform velocity vectors to normal and tangential components.
    - Solve Riemann problem to find interface variables.
    - Transform interface velocity back to Cartesian components.
    - Compute interface flux vector using Riemann interface variables, average metric components, and average grid velocity.
- Use maximum Riemann wave speed to determine maximum time step as:

$$\Delta t = CFL \frac{\min\left(\Delta\xi, \Delta\eta, \Delta\zeta\right)}{wavespeed_{\max}}$$

- Compute conserved variables.
- Update conserved variables using computed flux vectors.
- Compute updated primitive variables.
- Compute updated grid metric components.
- Compute updated grid velocity components.

4.7 Example Applications

A few examples are useful to showcase the potential benefits of UCS, particularly the automatic generation of curvilinear grids appropriate to specific problems.

4.7.1 Diamond shock train

The unified coordinate system is especially useful for problems where the physical boundaries themselves are unknown, such as occurs with pressure boundary conditions. Consider the nozzle plume flow in Fig. 4.5, which was generated with constant pressure boundary conditions. An efficient, flow-fitted grid has been automatically generated, without any user input other than the pressure at the boundaries. The grid has simply flowed to fill the streamtube defined by the nozzle, freeing the user from defining where those boundaries might lie. Using traditional methods, the simulation would have had to be sized such that it was larger than some estimated size of the streamtube. Such an approach requires simulation of additional nodes beyond those required for an understanding of the problem.

4.7.2 Transonic duct flow

UCS can also be used to generate a grid that conforms to some solid body, as in the case of the transonic duct shown in Fig. 4.6. This problem is identical to the one given by Hui [53], and is characterized by the formation of a mach stem and the resulting subsonic region and slip line. The grid simply flows through the duct, following the fluid as it conforms to the solid wall boundaries. Inflow conditions are given by equation (4.70), and the lower boundary has a 15° ramp from $x = 0.5$ to $x = 1.0$.

$$(p, \rho, M, \theta)\,|_{x=0} = (1, 1, 1.8, 0) \tag{4.70}$$

It can also be seen in Fig. 4.7 that using a stationary grid with the same code fails to capture the formation of the slip line at low grid resolutions, and also shows slightly less accurate prediction of shock locations when compared with a much higher resolution solution. UCS, on the other hand,

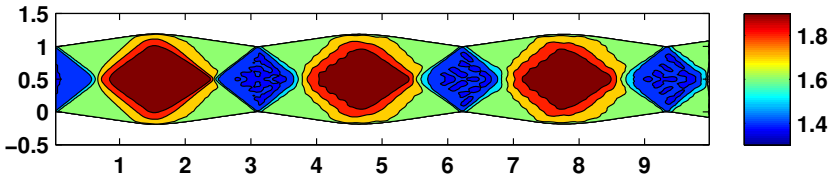


Figure 4.5: Computed Mach Number for Under-Expanded Nozzle Flow. UCS provides a powerful means of computing flows with poorly defined boundaries. Constant pressure boundary conditions are perfectly acceptable for the Euler equations, but they can be difficult to implement computationally, because the location of the boundary is not known a priori. This is no problem in UCS, as the grid will naturally move to the appropriate position as it follows the flow of the fluid, as shown in this example of constant pressure flow in a diamond shock train resulting from an under expanded nozzle.

does resolve the slip line, and makes slightly more accurate predictions of shock locations, at the cost of a less-well-resolved expansion corner.

4.7.3 Basic boundary-layer effects

Many complex flow phenomena are a result of the interaction of the viscous boundary layer with the inviscid flow. As a result, some method of accounting for viscous effects is almost a requirement for hypersonic flows, and boundary- layer methods provide this at minimal cost. A crude, prototypical implementation uses the turbulent, flat-plate, constant pressure formula given in Schlichting [70]:

$$\frac{\delta u_\infty}{\nu} = 0.14 \frac{\mathrm{Re}_x}{\log \mathrm{Re}_x} G\left(\log \mathrm{Re}_x\right)$$

where G is taken to be the limiting value of 1. This serves as a useful proof-of-concept for the method, and is a valuable step toward future incorporation of a boundary-layer solver. In the inviscid simulation, boundary-layer effects are included by enforcing a solid wall condition that aligns with the boundary-layer displacement thickness.

Results from one such proof-of-concept test are shown in Fig. 4.8. The presence of the boundary layer deflects the flow away from the channel walls, causing the channel to constrict. This causes the formation of an oblique shock train, much as that seen in figure 4.5 with the under-expanded nozzle.

4.7.4 Model inlet

One of the principal attractions of the unified coordinates method is the potential to represent complex flow geometries in a simple, intuitive way and without grid generation. To better illustrate this feature, a more complex inlet model was chosen, based off of publicly available sketches [71] of the inlet of the now retired USAF F-14. This inlet is approximately two-dimensional and is designed to provide subsonic flow to the engine at freestream Mach number in the range $0 < M < 2.3$. This is accomplished through the use of internal, variable ramps, as shown in Fig. 4.9.

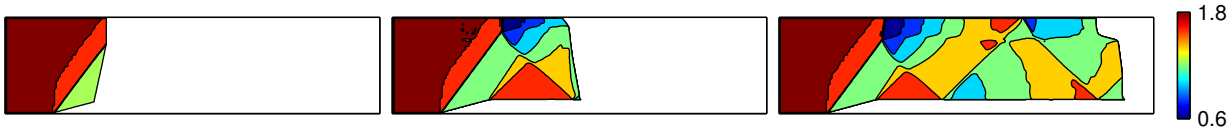
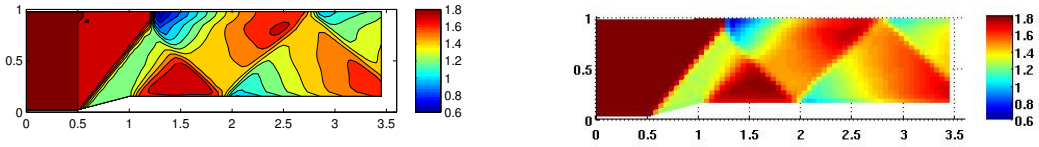
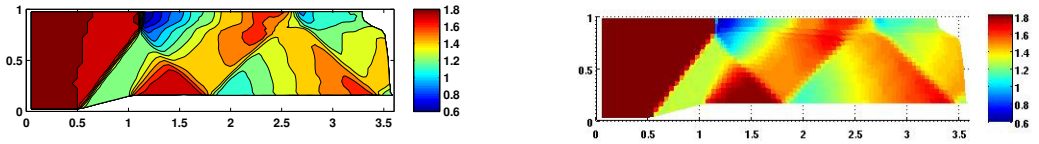


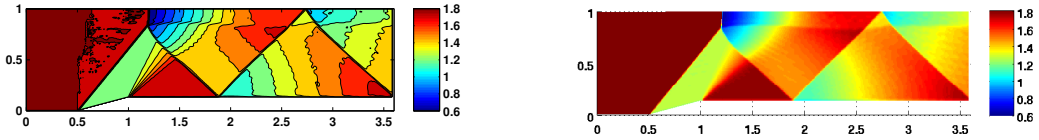
Figure 4.6: Time-Lapse Images of Transonic Duct Flow. The unsteady nature of UCS is such that it is most easily understood in a time-dependent context. The grid is generated at the upstream boundary and allowed to flow downstream to fill the channel boundaries. An oblique shock is formed as the grid encounters the ramp in the lower duct boundary. Where this shock impacts the upper wall, a Mach stem is formed, with a subsonic region directly behind the stem, bounded between the wall and a slip line. The reflected shock, as well as the expansion fan created by the end of the ramp, form a shock train that propagates down the channel. The subsonic region, with its elliptic character, takes some time to converge to a steady-state solution. This necessitates the periodic removal of grid points as they flow out of the simulation region.



(a) $h = 0$, nominal grid size: 72×20



(b) $h = 0.25$, nominal grid size: 72×20



(c) $h = 0$, nominal grid size: 360×100

Figure 4.7: Accuracy Comparison Between Moving and Stationary Grids. One of the advantages of the Lagrangian-style unsteady coordinate system used by UCS is that slip line resolution is dramatically improved. This can be seen in a grid resolution comparison of the transonic duct problem. The same simulation is run three times, with both stationary and moving grids and with high and low resolutions. At low resolutions, the slip line is completely smeared for the stationary grid case, but it is clearly visible in unified coordinates and in stationary coodrinates at higher resolution. The UCS simulation (b) has grid motion specified by $(U, V) = (0.25u, 0.25v)$.

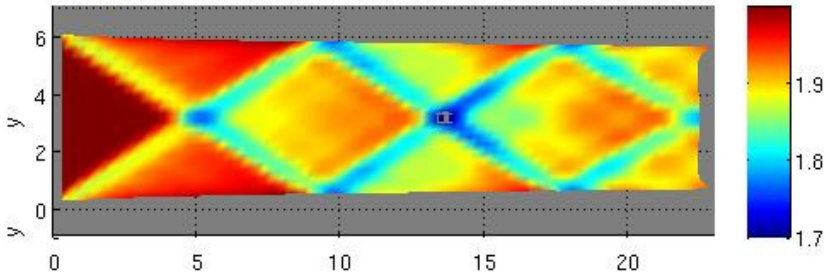


Figure 4.8: Channel Flow With a Turbulent Boundary Layer. UCS can also be used to simulate more complex flows, with some additional approximations. It is possible to model a boundary-layer driven shock train in an otherwise parallel channel, by using the turbulent-boundary momentum thickness to create an artificial "wall", and using UCS to model the inviscid core flow. This suggests many future applications, including integration of UCS with boundary-layer solvers, full Navier-Stokes/RANS codes, and others.

This is exactly the kind of problem the unified coordinates can excel at. Defining an approximate flow geometry is simple, and the automatic grid generation allows for quick solutions at different freestream conditions, and the correspondingly different inlet geometries. An example based on the F-14 inlet is shown in Fig. 4.10, but the results are only prototypical. Bleed flows, in particular, require special code features to effectively handle downstream boundaries as the grid reaches them. Further investigation into these types of problems is needed.

4.8 Code Development and Versions

Much work remains to be done with the unified coordinate system, but many of the desired new features are already under development in the latest iteration of the `BACL-Streamer` code.

4.8.1 Streamer v1.0

Initial development of `BACL-Streamer` began in 2009 with simple one-dimensional tests and demonstrations in Matlab, before moving to fully two-dimensional flows in the latter half of that year. Development continued until the Fall of 2010, when development began on v2.0.

Version 1.0 stored simulation variables, including conserved quantities, geometric coefficients, and grid motion parameters, as multidimensional arrays. To date, v1.0 has primarily served as a lessons-learned version, and informed many of the design choices in versions 2.0 and 3.0.

4.8.2 Streamer v2.0

Development of `BACL-Streamer` v2.0 began in 2010, and continued until moving to v3.0 in December 2011. It is most notable for its heavy use of Fortran object-oriented programming and for its implementation of the primary data structure as a linked list, rather than as an array. These design changes were made primarily as a means to improve on the performance of v1.0, which required extensive reallocation and copying of simulation data whenever grid points had to be added or removed. Unfortunately, many of these design changes also had a direct negative effect of simulation performance, and this, along with the need for a three-dimensional code and better

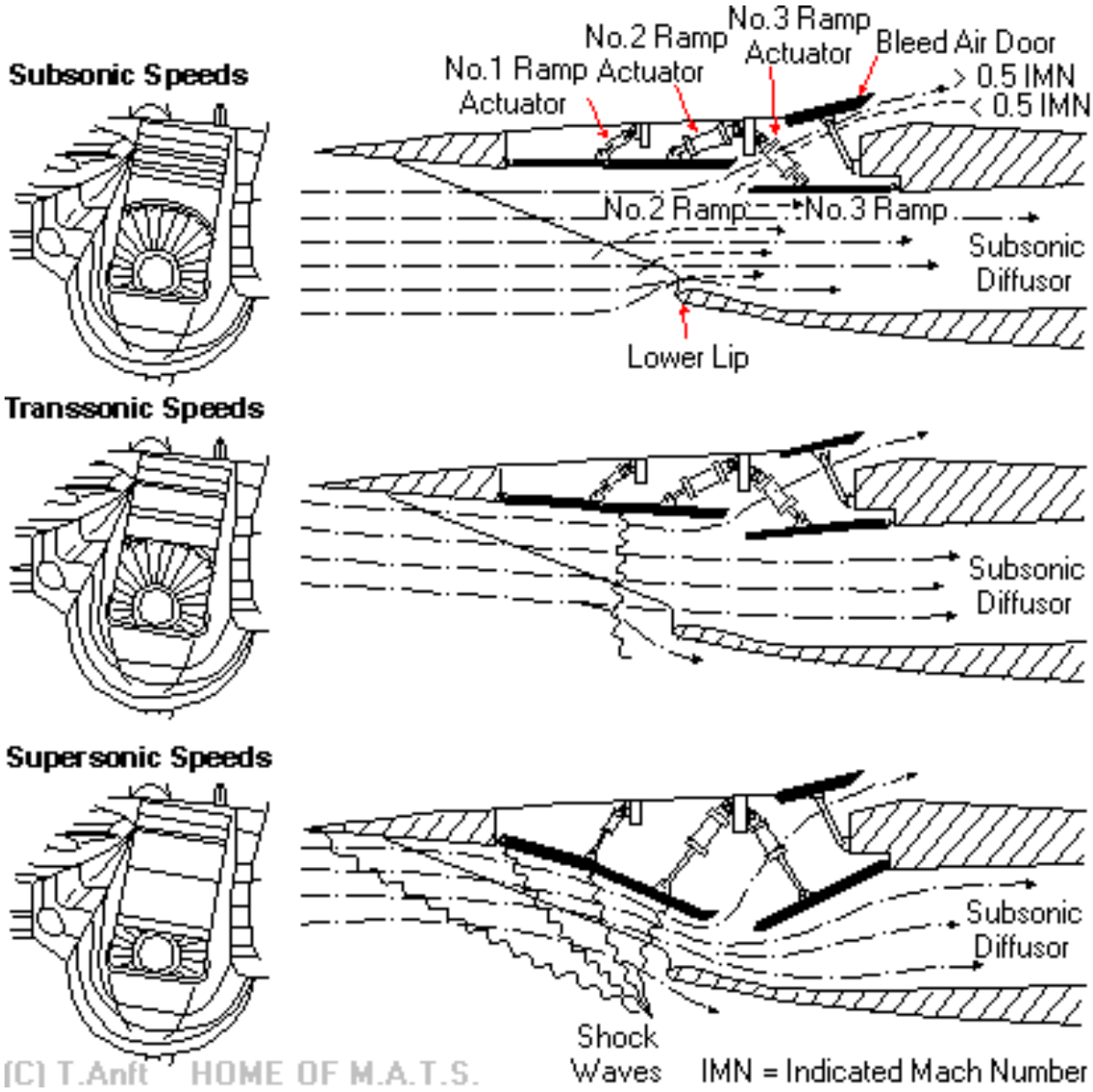


Figure 4.9: Diagram of the USAF F-14 Tomcat [71]. The ability of UCS to rapidly simulate flows in different physical geometries makes a problem such as this variable-geometry inlet a very attractive demonstration. Ideally, UCS can rapidly simulate the flow conditions in this inlet under a variety of configurations, as well as the transition modes from one state to the next.

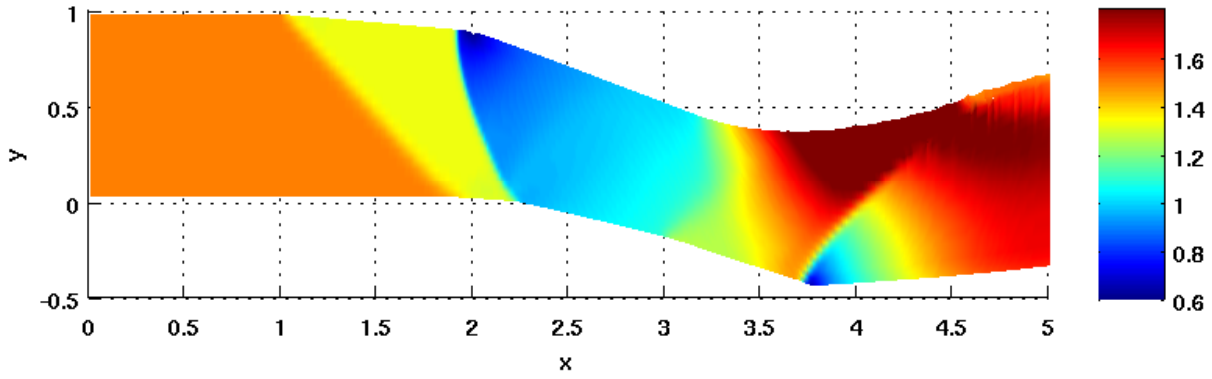


Figure 4.10: Mach number in F-14 Inlet. As a preliminary demonstration problem for UCS, flow through a geometry roughly approximating that in figure 4.9 was simulated. In this case, the flow conditions were chosen such that the inlet flow remained supersonic throughout, and bleed flows were neglected. This greatly simplified the problem, as it was unnecessary to devise a method for handling multiple fluid streams. These preliminary results are promising, and future iterations of this problem with representative inlet conditions, compression angles, and bleed flows may prove extremely valuable.

verification options, led directly to the development of v3.0.

`BACL-Streamer` v2.0 is structured as a group of interacting Fortran modules. It makes heavy use of object-oriented features introduced in Fortran 2003, including objects (Fortran derived types with bound methods), private variables, and the linked-list structure that replaces the multidimensional arrays used to store simulation variables in v1.0.

4.8.2.1 Node objects

There are two fundamental objects used in v2.0: `node` objects, which are containers for the simulation variables p, ρ, u, v, A, B, L, M, x, y, and so on, and the `node_array`, which is a linked list joining many nodes together into a two-dimensional grid. These two objects are fundamental to understanding the workings of v2.0, and their definitions are contained in `types.f90`, and `node_array`, respectively.

In addition to containing simulation variables, `nodes` also provide a variety of functions and routines for working with `node` objects directly, including arithmetic operators, and equivalence functions, and velocity component transformations. Additionally, each node contains pointers to each directly neighboring node, which are used to implement the linked list.

The linked list itself requires more explanation. In this implementation, the list is built from a single head `node` pointer, corresponding to the grid coordinates $i = 1$, $j = 1$. Specific nodes are accessed using the `get_node` function, which begins at the head and steps through the list to the appropriate node. `set_node` works in a similar fashion. The `node_array` module also contains a variety of functions that simplify the process of working adding and removing nodes, and populating them based on boundary conditions.

Once the particulars of communicating with the linked-list structure are understood, the rest of v2.0 is a relatively straightforward implementation of the basic UCS algorithm. The driver program is the aptly named `main.f90`, which reads a single input filename as a command line argument, initializes the simulation from this file using the `read_boundary` subroutine, located in the `boundary_conditions_init` module, and then calls the various modules of the code as

necessary, until the desired output time has been reached.

The `read_boundary` routine defines four `boundary_master` objects, defined in `types.f90`, describing the left, right, bottom, and top boundary conditions, along with specifications for the computational region of interest, the dimensionality of the initial array, and the length of the computational differentials `dxi` and `deta`, as well as a variety of flags that are used to control various parts of the simulation.

4.8.3 Streamer v2.0 Verification

Verification is a critical piece in the development of any scientific code, and `BACL-Streamer` v2.0 is no exception. As the code has no provision for the computation of source terms, as would be required for the method of manufactured solutions, verification is restricted to order-of-accuracy convergence using exact solutions to the Euler equations: a steady, supersonic Riemann problem, a wall-induced oblique shock, and a wall-induced Prandtl-Meyer expansion fan.

4.8.3.1 Two-dimensional Riemann problem

The two-dimensional, steady, Riemann problem is a direct analogue to the one-dimensional, unsteady, Riemann problems that are discussed in Sec. 4.5, and is especially useful as a verification test because of the stress it places on solvers. The particular problem used here is given by Hui [53], and consists of an expansion fan, a slip line, and a shock. These waves converge to a singularity located at the upstream boundary. The upstream conditions are given by:

$$(p, \rho, M, \theta) = \begin{cases} (0.25, 0.5, 7, 0) & : y > 0 \\ (1, 1, 2.4, 0) & : y < 0 \end{cases} \tag{4.71}$$

where M is the flow Mach number and θ is the flow angle, measured from horizontal. This problem admits a similarity solution, much as the one-dimensional problem does, and the results from plotting this solution are shown in Figs. 4.12 and 4.13. In particular, it should be noted that the slip line is fully resolved in the moving grid case. This is a well known effect of Lagrangian-esque simulations [21, 53, 54], and one of the beneficial features of UCS.

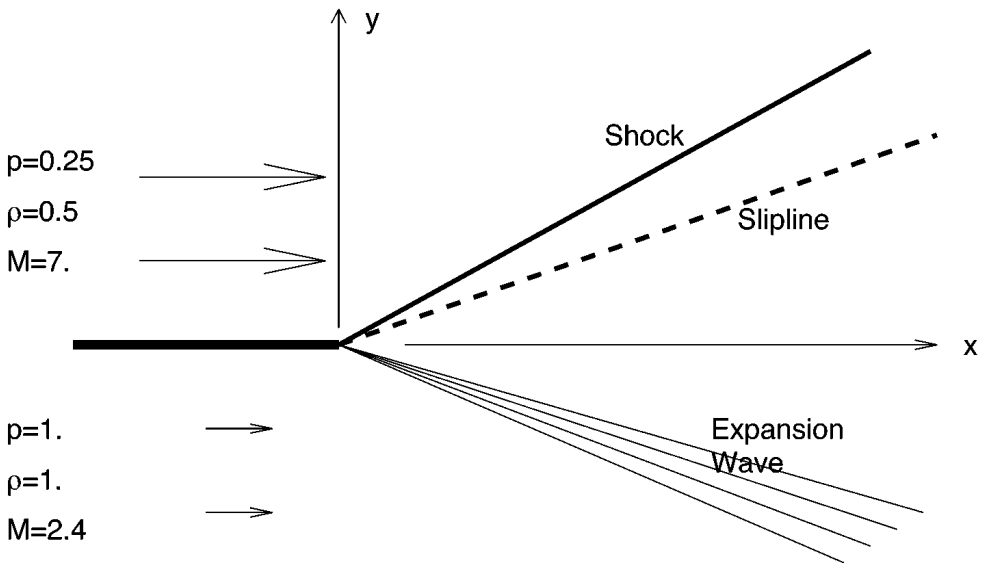


Figure 4.11: Two-dimensional, Steady-state Riemann Problem. For steady-state, supersonic flows, the two-dimensional Euler Equations reduce to a one-dimensional system which admits the definition of a Riemann problem with an exact, similarity solution, just as with the unsteady case. In this case, two supersonic streams come into contact at x_0, and three nonlinear waves are formed. The central wave is a slip line, while the character of the outer two waves is determined by the specific problem. This particular problem is used by Hui [53] in order to demonstrate the utility of the UCS method.

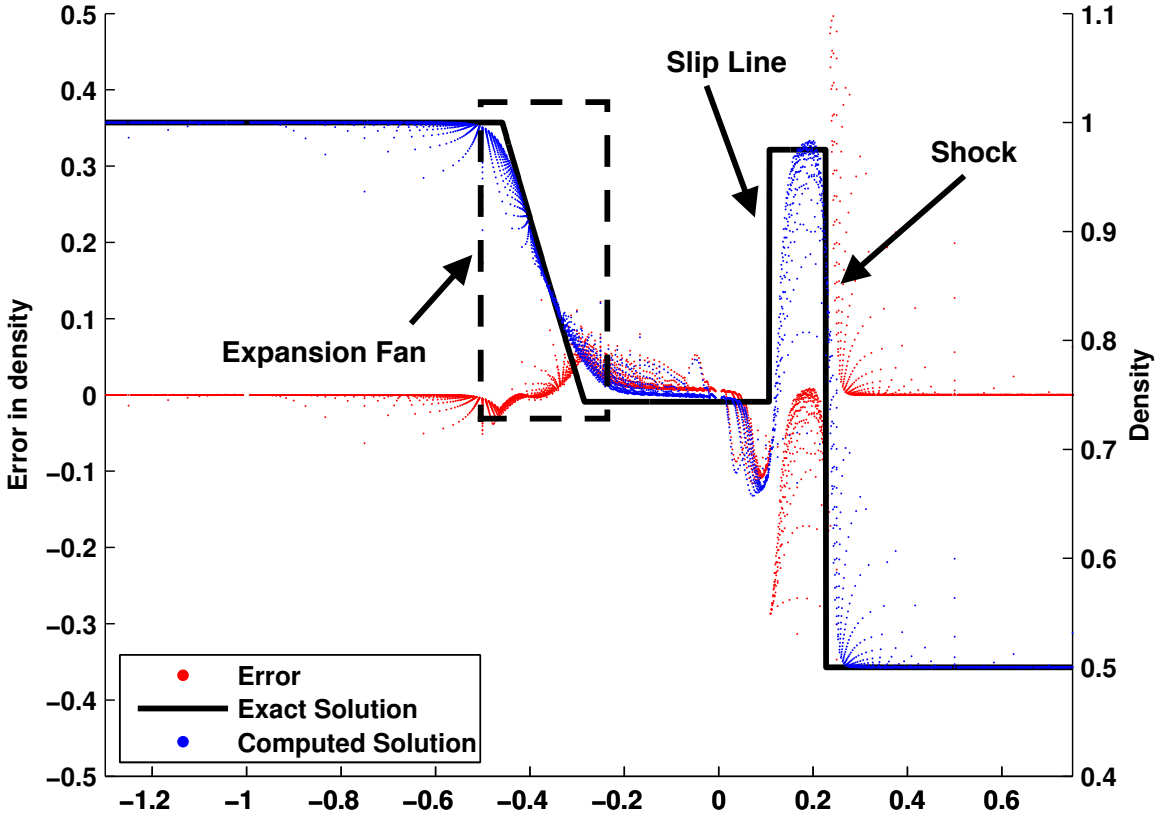


Figure 4.12: The 2-D Riemann problem with corresponding numerical error, stationary grid. One important characteristic of Riemann problems is that they can be solved with a similarity solution. For this two-dimensional problem, a change of variables to $\frac{y}{x}$ results in a one-dimensional solution which can be represented exactly. When this problem is solved numerically, each grid point corresponds to a particular $\frac{y}{x}$ value, and the value of the solution at that point can be visualized with a scatter plot. Comparing the computed solutions with the exact solution yields a scatter plot of error, as well. These results are here shown for density, as solved on a traditional, stationary, Eulerian grid. The smoothing effect of the Godunov solver has smeared out the corners on the expansion fan, as well as the shock, and a large oscilation is also visible at the central slip line.

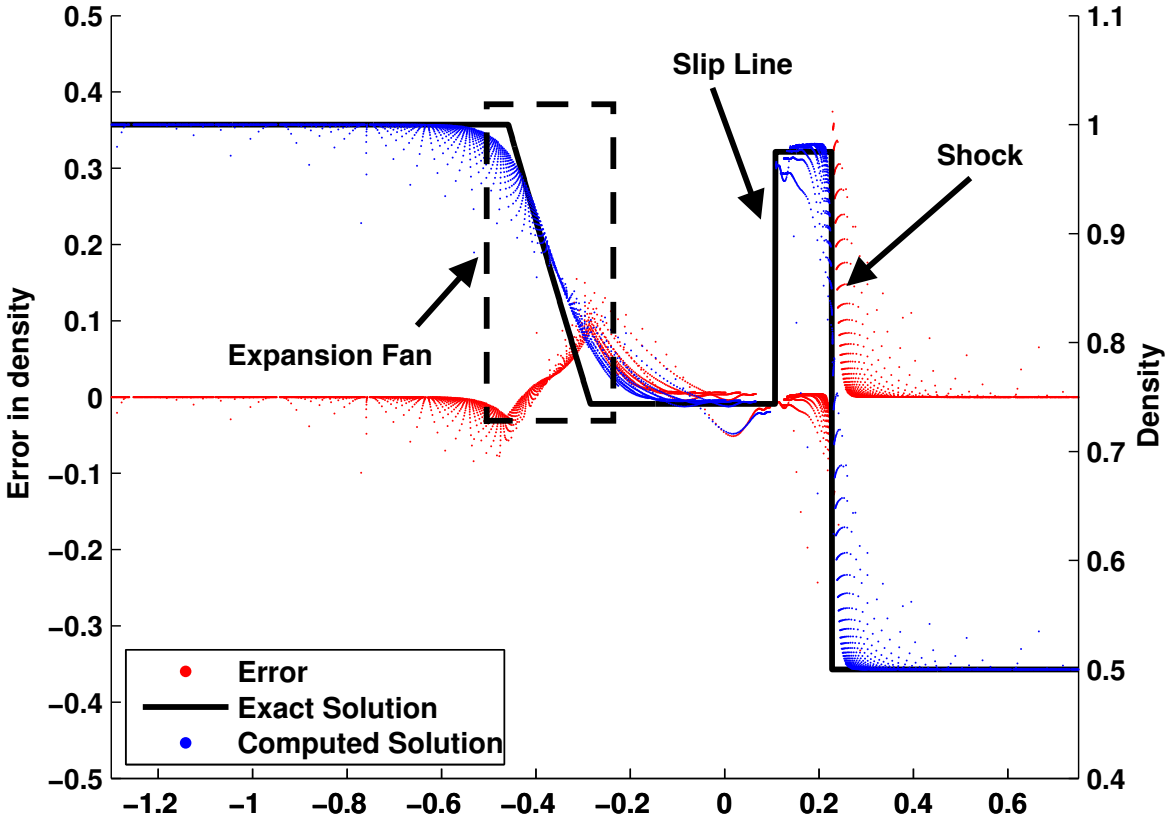


Figure 4.13: The 2-D Riemann problem with corresponding numerical error, moving grid. The same plot can be created for a simulation using the UCS grid, instead. Grid motion is given by $(U, V) = (0.999u, 0.999v)$. When compared with figure 4.12, the smearing oscillation of the slip line has almost entirely vanished, and the shock smearing appears to be somewhat reduced, as well. These improvements come at the cost of a slightly increased smearing of the expansion fan. Further analysis is required in order to determine the actual benefits of UCS, but improved shock and slip line resolution is very promising.

The two-dimensional Riemann problem can be solved exactly [72], and so convergence rates can be measured for this problem. Convergence is tested for both moving and stationary grids, with and without the 2^{nd}-order MUSCL update. These convergence rates are shown in Fig. 4.14. It is clear that the algorithm converges for at least three different test configurations, however this highlights one of the difficulties associated with convergence testing with shock-capturing codes, in that the presence of discontinuities generally reduces the observed order of convergence dramatically. [42] [43]

Unfortunately, these tests also expose drawbacks of the unified coordinates. While the moving grid simulation provides much higher accuracy at the slip line, overall accuracy actually decreases, and the rate of convergence drops measurably, as well. This may be due to the unsteady representation of the singularity at the upstream boundary. As the grid moves downstream, any point with an x value that places it upstream of the boundary will have a state defined by the boundary conditions, while any with an x value that places it downstream will be evolving in time. This effectively creates unsteadiness in the application of the singularity as the first grid points are located varying distances downstream of the boundary. The two-dimensional Riemann problem is defined entirely by that singularity, and unsteadiness could easily cause this increased error throughout the simulation.

4.8.3.2 Wall-induced shock wave

The next test was designed to provide better coverage of the different boundary conditions. Supersonic flow past a corner is well-defined, both for induced shocks and expansions, and provides a convenient test of solid wall boundary conditions. The upstream condition is given by:

$$(p, \rho, M, \theta) = (1, 1, 1.8, 0) \tag{4.72}$$

The wall angle is chosen to yield a downstream flow angle of 45^o. The grid is generated automatically, with grid velocity being set to 99.9% of flow velocity ($h = 0.999$). Grid convergence rates are computed and convergence is shown in figure 4.15 again fall short of expectations. Like the Riemann

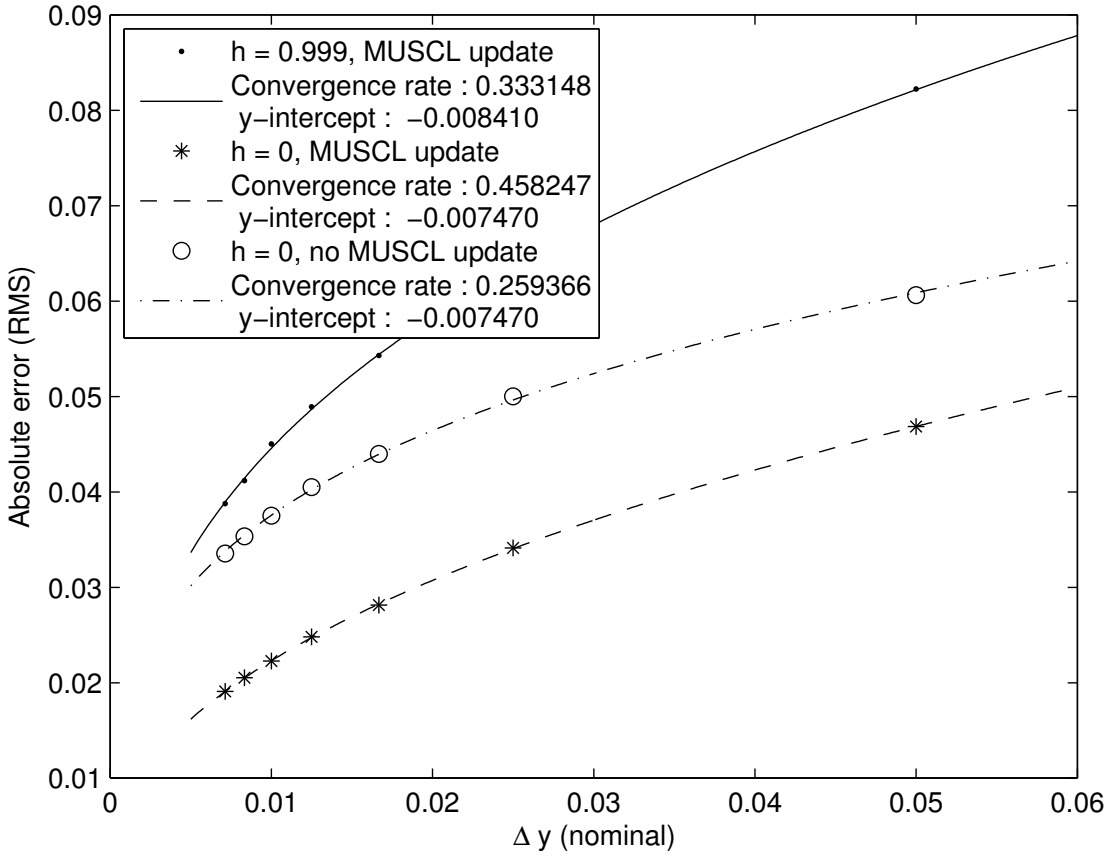


Figure 4.14: Streamer 2.0, 2-D Riemann Problem, Convergence. A grid refinement study was performed using Streamer v2.0, with the two-dimensional Riemann problem given in equation (4.71). Convergence rates were measured for both stationary and moving grids, based on the behavior of the simulation L_2 norm for density. These were tested using both the first-order Godunov solver, as well as the second-order MUSCL update scheme. Convergence for the UCS grid was found to be somewhat lower than for the stationary grid. It is unknown if this is an artifact of the incomplete Streamer v2.0, if it is caused by the use of the L_2 norm, or if it is an inherent result of the UCS method, when applied as has been done here.

problem, wall-induced shock flow is defined by the location of the critical point in the wall boundary condition, and it is unsurprising that an unsteady apparent location for this critical point would negatively affect error and convergence. Additionally, unsteady oscillations arise at higher grid resolutions, though preliminary investigations suggest that these may be due to inappropriately large time steps.

4.8.3.3 Wall-induced expansion

Wall-induced expansion fans, in contrast, offer a smooth solution, though one that is also dependent on a single critical point. Convergence rates were measured as with the shock, and with similar results, as seen in Fig. 4.17. A study was also done on flow behavior under varying expansion angles, which yielded important insights: in the presence of strong expansion waves, the computational grid tends to pull away from the wall, as seen in Fig. 4.18. It is important to note that this behavior was not observed in Hui's work on complex bodies, such as the multi-element airfoil shown in figure 4.1.

4.8.3.4 Conclusion

Overall, this type of verification has highlighted the difficulties that must be overcome by any UCS program. The very simple implementation of moving grids that was used here appears to be insufficient, leading to errors in grid motion and lower rates of convergence, but Hui's later work suggests that a more sophisticated treatment of grid motion can overcome at least some of these. Even without further refinements of the method, the increase in error is seen to be quite small, and the simplicity of the grid generation process in UCS may well be worth sacrifices in accuracy.

Unfortunately, it was impossible to conclusively establish verification for `BACL-Streamer` v2.0. The available verification solutions simply incorporated too many difficult features, and it was impossible to separate errors due to the solution algorithm from errors due to the implementation of boundary conditions or errors in the coding itself. To solve this problem, it was necessary to turn to the method of manufactured solutions. This, along with the performance issues encountered in

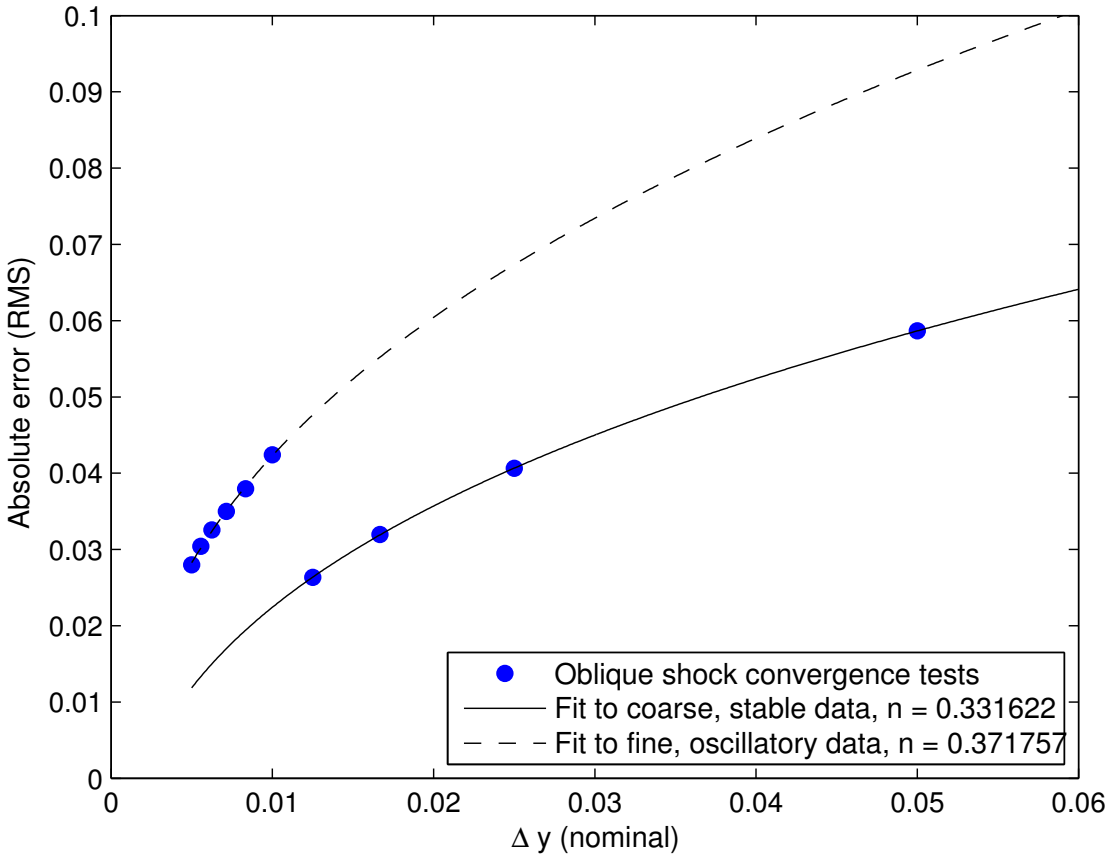


Figure 4.15: Streamer 2.0, L_2 Error for an Oblique Shock. A grid refinement study was performed using Streamer v2.0, with a simple, wall-induced, oblique shock wave. This problem was chosen in order to more fully test the implementation of boundary conditions in Streamer v2.0, especially those corresponding to flows around solid bodies. For this problem, refinement beyond a certain point caused the solution to become unstable, with strong oscillations that propagated downstream from the shock wave. The appearance of these strong oscillations caused the grid convergence study to split into two distinct curves. Although it is possible that this is a direct result of the unsteadiness of UCS, some evidence exists to indicate that it is actually a result of an incorrect CFD condition. Work is now being done using Streamer v3.0 in order to comfirm this hypothesis.

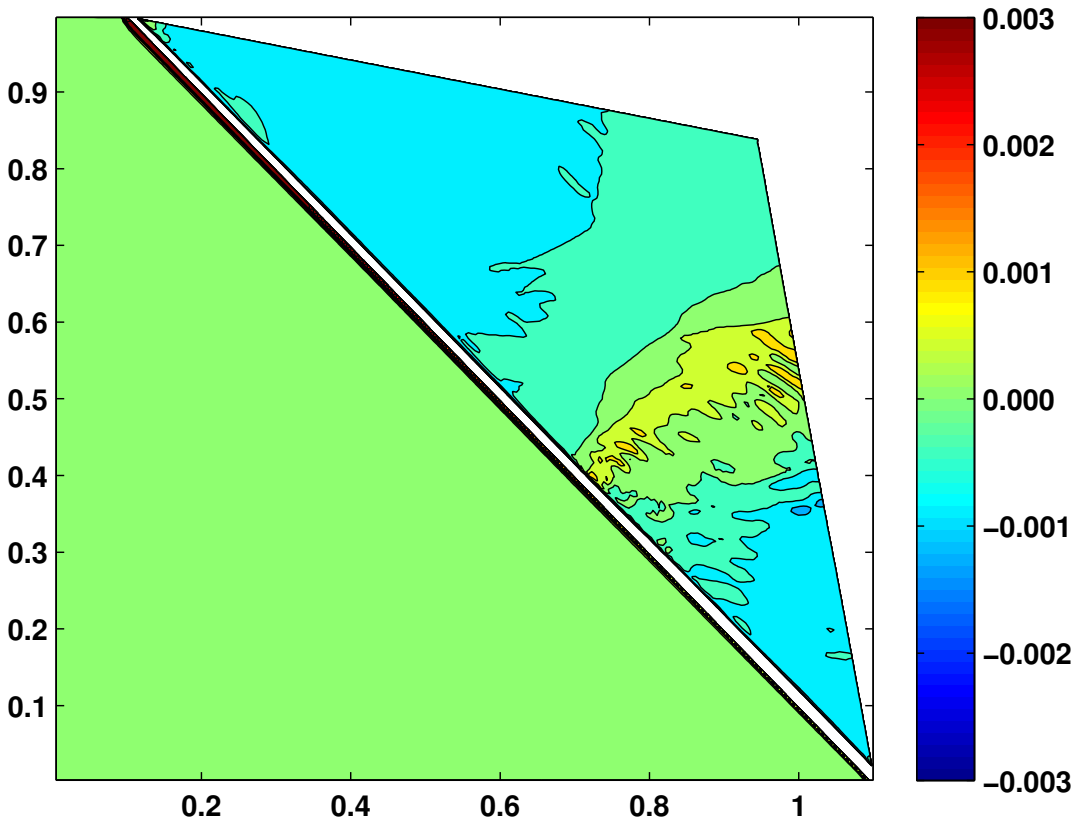


Figure 4.16: Streamer 2.0, Oblique Shock, Pressure Error. For computational grids with a grid spacing $\Delta\xi$ that is equal to 0.01 or smaller, unsteady oscillations appear in the solution of the wall-induced oblique shock wave. These oscilations, although small, can be clearly seen in plots of the error in the pressure field. The appear to originate in the shock wave and propagate downstream, rather than dissipating as one would expect. The appearance of these oscillations corresponds to an overall change in the rate of convergence of Streamer v2.0. Preliminary investigations suggest that this may be simply the result of an innappropriate CFL condition, which hypothesis is to be confirmed in Streamer v3.0.

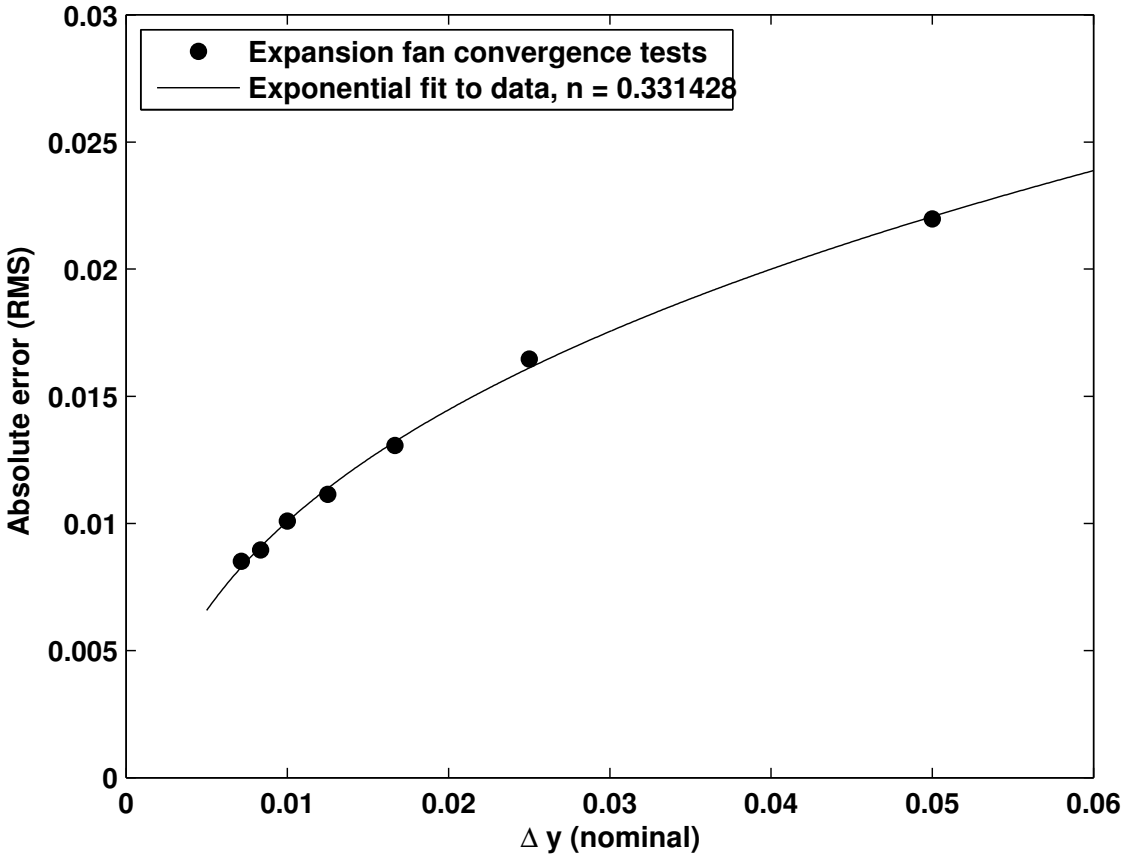


Figure 4.17: Streamer 2.0, Prandtl-Meyer Expansion Wave Convergence. A grid-refinement study was conducted using a wall-induced Prandtl-Meyer expansion fan as an exact solution. This problem did not appear to suffer from oscillatory behavior as was found in the oblique shock, though overall convergence rates were somewhat lower than expected.

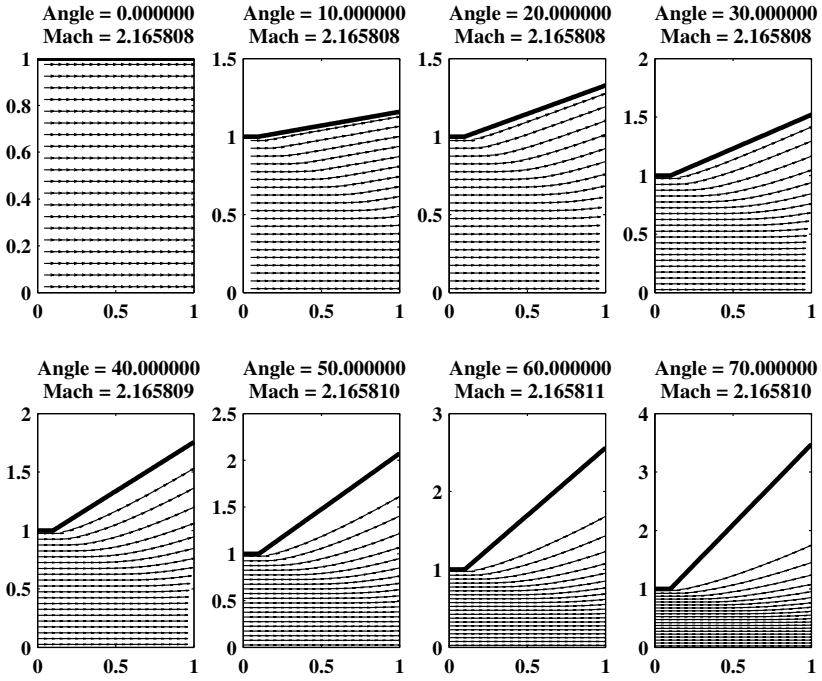


Figure 4.18: Streamer 2.0, Prandtl-Meyer Expansion Wave Streamlines. A troubling feature of UCS is the inability of the coordinate system to apply boundary conditions based on the physical location of the boundary. Because the relative positions of grid points and boundaries are never used, it is possible to encounter situations where the simulated flow field pulls away from the boundary, as is seen in this study of UCS flow over an expansion corner of increasingly sharp angle. It is unclear exactly how important this behavior is, as realistic flows over corners of this kind would most likely involve boundary-layer separation and recirculation, anyway. Additionally, Hui has demonstrated that these particular difficulties can be overcome, as seen in figure 4.1. Angles are given in degrees.

v2.0 and the need for a three-dimensional solver, led directly to the development of v3.0.

4.8.4 Streamer v3.0

`BACL-Streamer`v3.0 was developed to remedy many of the difficulties encountered during development of version 2.0. It abandoned the object-oriented style, and reverted to the use of multidimensional arrays to track flow states. It is also the first version to attempt to handle the increasingly difficult problem of boundary condition specification and implementation, while also building a fully three-dimensional framework for later expansion. Finally, v3.0 is the first version to include modern software quality controls, including an expansive suite of unit tests and built-in order-of-accuracy convergence testing. Development was begun in late 2011, and has continued to the present time.

`BACL-Streamer`v3.0 can be roughly divided into two parts: a high-performance core, and developer-friendly scripts. The interface between these divisions is roughly equivalent to a division between the time-marching step, which is handled in script, and the space-marching computations, which are handled in core. Within a given time step, core functionality may be used extensively, but everything else is handled at the scripting level. The decision was made early in development to implement these as a hybrid Fortran/Python code, with the core being developed in Fortran 90, and the scripts developed in Python 2.x. The interface between the core and the scripts is provided by the `f2py` package [73], which is included as part of the SciPy Python library.

4.8.4.1 Software prerequisites

`BACL-Streamer` v3.0 has made a concerted effort to leverage available software tools, while minimizing the number of special software packages that are required. Unfortunately, as of this writing, several of these packages provide required functionality only in the development branch, and so it is necessary to compile them from the `git` repositories. This is noted, when applicable.

- Build tools

    - Fortran90 compiler, such as GNU Fortran 4.2.3 (OS X).
    - C compiler, such as Apple LLVM version 5.1.
    - LAPACK library (required for test routines)
    - standard GNU build tools e.g. `make`, `ar`, etc.

- Python environment
    - Python 2.7.x
    - SciPy library, for numerical integration. This functionality is currently available only on the `master` branch, currently slated for v0.15, so SciPy must be built from source.
    - NumPy package for array-based computation in Python. Version must be sufficiently new to build SciPy master, such as v1.8.1.
    - Cython package, required to build NumPy and SciPy.
    - Sympy package for use with manufactured solutions. Must also be built from the git development branch, currently v0.7.5.
    - Nose testing package, useful for ensuring proper build of SciPy and Sympy.

The build process will also compile several files from the `MINPACK` library, which are used for some tests. These are distributed with the package, and so are not considered prerequisites.

4.8.4.2 Fortran core

The Fortran Core itself consists of three main sections. There are the underlying library functions which solve Riemann problems, implement the Godunov method, and so on. There are the interface driver functions, which are linked using `f2py` [73] and provide the primary access to the library. Finally, there are the test functions, which implement everything from individual unit tests to full time-dependent convergence testing of the UCS system. These three components are all built through the Makefile included in the `Streamer` source directory.

The most important unit of data in the Fortran Core is the `main_data` array. This is a four-dimensional array of double-precision floating point numbers, and contains all of the most important information needed for solution of the Euler equations in unified coordinates. Each computational node stores a flow state, $\mathbf{W}_{ijk}$, represented by 21 distinct numbers (the innermost array dimension), corresponding to the variables p, ρ, u, v, w, A, B, C, L, M, N, P, Q, R, U, V, W, x, y, z, J. This is very nearly the minimum set of numbers required to define the system; J can be computed from the metric components, but it is used so frequently that it is useful to maintain a copy.

`main_data` is also sized to be larger than is needed for holding array data. Boundary conditions in `BACL-Streamer` are implemented using ghost points, as shown in figure 4.19. Implementing boundary conditions in this way greatly simplifies the Core, and allows the difficult task of implementing boundary conditions to be handled in Python.

The primary task of the library functions in the Fortran Core is to time-advance the flow quantities in the `main_data` object from time step n to time step $n + 1$, as shown in figure 4.20. Although the `f2py` software is mature and well-established, the Python-Fortran interface is most simple when the surface of the Fortran library is tightly constrained. As a result, access to the Core is restricted to interface functions in order to reduce that surface. The Python script calls these driver functions with the `main_data` array, a preliminary time step Δt, and an integer array of options that can be used to control the action of the Core library. This limits the interaction between Python and Fortran to three inputs: a four-dimensional array of double-precision numbers; a double-precision preliminary time step; and a one-dimensional array of integers. The Godunov library computes an actual time step based on the preliminary one and the values in the options array. This actual time step is returned, along with a `main_data` array in which the variables except for the grid velocity components U, V, and W have been advanced by the actual time step. These are used as inputs to the `grid_motion_driver`, which updates the grid velocity components, completing the update to the time-step-evolved flow state $\mathbf{W}_{ijk}^{n+1}$.

Core library

`libStreamer` is built using `make all`, and provides everything from basic matrix operations,

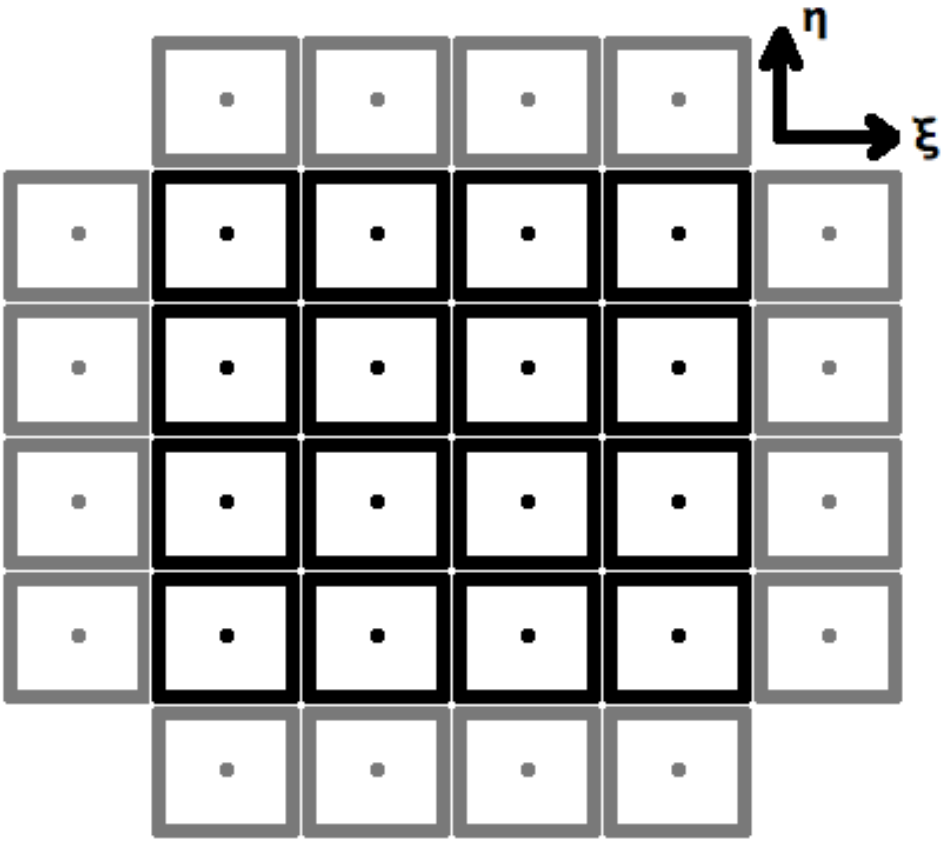


Figure 4.19: Two-dimensional, Cell-centered Grid with Ghost Points. Ghost points are one method for implementing boundary conditions in finite-volume fluid mechanical solvers that allows for a very simple flow solver. In this system, the solver has no specific knowledge of the simulation boundaries. Rather, "ghost" points are populated such that the action of the flow solver on the interior of the grid enforces the boundary conditions implicitly. For inviscid fluid simulations with a solid wall condition, for instance, the flow state of the ghost points is a mirrored reflection of the interior points, while a supersonic upstream condition is simply a specified Dirichlet condition.

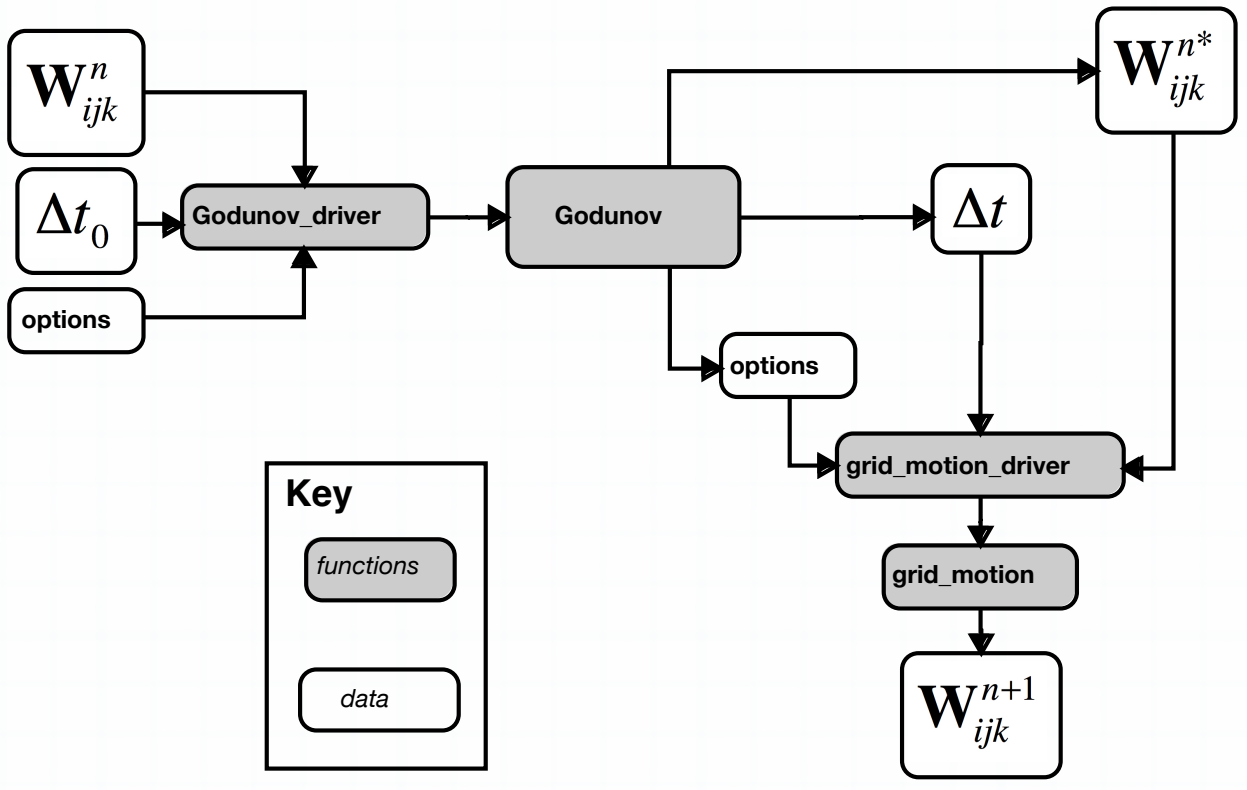


Figure 4.20: The Flow of Information in the Streamer v3.0 Fortran Core. The Fortran Core of Streamer v3.0 contains the code that implements the Godunov method for the Euler equations in the unified coordinates. This Fortran library is primarily used to accelerate the Godunov method relative to a pure Python implementation, and many of its design decisions are motivated by this mixed-language architecture. The `main_data` array W_{ijk}^n is passed from the driving Python script to the library interface functions, along with simulation options and a preliminary time step, and an actual time step is returned, along with a time-evolved `main_data` array.

to special utilities for working with the UCS flow state vectors, to solving Riemann problems, to using the Godunov method to advance a UCS state from one time step to the next. The files which are compiled into `libStreamer` are:

- `GeneralUtilities.f90` - Useful utilities for working with UCS flow states.
- `Riemann.f90` - The function `riemann_solve`, which solves one-dimensional UCS Riemann problems.
- `TimeAdvancementStuff.f90` - Utilities for managing the unsteady nature of UCS grids, as well as an I/O routine for interfacing with Matlab.
- `Godunov.f90` - The function `prim_update`, which advances a UCS flow state from one time step to the next.
- `FortranNormalVectors.f90` - Routines for use in applying solid wall boundary conditions. Experimental.
- `grid_motion.f90` - Routines for solving the grid control equations. At present, only Eq. 4.38 is implemented.

Library interface

Though the `f2py` tool greatly simplifies the process of communicating data between Python and Fortran codes, it is nonetheless advantageous to restrict and simplify the interface between the two divisions as much as possible. Driver modules for the `Godunov` and `grid_motion` modules provide this kind of interface, and all calls to the underlying library functions pass through these drivers. As this interface project is ongoing, no such drivers exist for the other library modules at present.

Control of the underlying library routines is provided by the integer `options` array. The values of different elements of this array determine behaviors such as the particular Godunov algorithm to use, or the number of ghost points needed to specify boundary conditions. While the library remains

under active development, the exact behavior of these options remains in flux, but a snapshot of various option values and their meanings may still be helpful.

```
Options meanings
[1-2]: controls which prim_update algorithm to use
[3-5]: sets values for dxi, deta, dzeta (0=>1., 1=>0.5, 2=>0.25)
[6-7]: controls grid motion.
[6 ]: 0=>Eulerian, 1=>Lagrangian-esque, 2=>Constant, 3=>Angle-preserving
[7 ]: h0 = (1=>.25, 2=>.5, 3=>.999)
[101]: reports how many boundary ghost points are present
[102]: controls spatial order of accuracy
[104]: Controls type of time step (constant or CFL)
[301]: Controls type of grid motion
```

This form of algorithm control is crude, but highly extensible. It is assumed that a more user-friendly interface will eventually be implemented.

Unit testing and library verification

The most difficult part of any code development project is ensuring the correctness of the code. To this end, tests have been developed that provide extensive coverage of the `libStreamer` functionality. As much as possible, these tests rely on mathematics to evaluate functionality directly. For instance, if a function computes the inverse of a matrix, then the matrix product of a random initial matrix and the result of that function should be the identity matrix. Where necessary, the tests may be written based on sample problems where the results are known. Finally, the testing library provides routines for testing the overall convergence of the Godunov solver to exact solutions to the Euler equations, and comparing the measured rate of convergence with that returned by other codes.

The test suite is composed of additional Fortran modules, one for each module being tested. These contain both a battery of tests and a routine for interpreting the error messages that might result. A test suite driver program is also available. Compilation and execution of the full testing

executable is done by executing `make check` in the `Streamer` source directory.

The coverage of the test suite has been made as broad as possible, but there are known areas where testing has not yet been possible. As of this writing, testing of the Euler equations, where the grid transformation is the identity, has been implemented and achieved up to order-of-accuracy convergence for one-dimensional problems. Visual checks show agreement for two-dimensional problems, as well. Testing of code behavior with moving, curvilinear grids (full UCS) remains incomplete.

4.8.4.3 Python scripts

While the Fortran Core provides most of the basic functionality required for a working UCS code, Python scripting handles many of the more complex tasks. It consists primarily of two files. `main.py` covers initialization of the overall simulation, and it also handles time-advancement of the solution, application of source terms, data output, and computation of simulation error for verification problems. `BoundaryConditions.py` is an experimental boundary conditions implementation.

main.py

Execution of a simulation program from Python is done by running `python main.py <input file>`. The script reads data from the input file, and then initializes `Stream` objects based on that data. This object defines the simulation boundary conditions, as well as options that allow for control of how the object behaves. This design is chosen to allow for future expansion, including multiple streams that interact through boundary conditions, specification of different flow solvers, incorporation of non-conservative source terms, and so on. Each `Stream` object also includes the required tools to advance its state from one time step to another.

Once any streams have been initialized with appropriate boundary conditions, initial conditions, and options, a time-stepping simulation is run to advance these streams forward.

Input files

Input files for `BACL-Streamer` are Python files. They must contain an importable `init` function returning three objects:

- `bounds_init` - Used by `BoundaryCondtions.py` to initialize `Bounds` objects.

- `initial_conds` - An array to initialize `main_data`. If empty, then boundary conditions are used for initialization.

- `stream_options` - A Python dict, containing options, additional information, and other directives required by the `Stream`. For instance, source functions for the method of manufactured solutions are included here. Also includes the `solver_options` array, which is passed to the Fortran Core.

For more information on the effects of these objects, consult the source code for the `main.py` and `BoundaryConditions.py` files.

BoundaryConditions.py

`BoundaryConditions.py` is an experimental method for implementing boundary conditions in an extensible way. The module organizes boundary condition data into `Patch` objects, which are themselves organized into `Face` and `Bounds` objects.

Each `Patch` represents a single, distinct, boundary condition, such as a planar wall with a specific normal vector, or a constant pressure boundary. These patches are organized into `Face` objects, representing the six logical boundaries of the `main_data` array, and everything is brought together in the `Bounds` object, one of which corresponds to a `Stream`.

`BoundaryCondtions.py` and `BoundaryConditionsStuff.f90` work together with the code-generated `FortranNormalVectors.f90` to implement various types of boundary conditions, with varying degrees of success. The most difficult problems are a direct result of the unsteady nature of the UCS grid, which makes it difficult to determine how to best assign grid points to the appropriate boundary patches. Research in this area is ongoing.

4.8.4.4 BACL-Manufactured and the Method of Manufactured Solutions

`BACL-Streamer` v3.0 is also integrated with `BACL-manufactured`, a verification package available in Python. `BACL-manufactured` is an add-on tool that greatly simplifies the process of computing source terms for use in the method of manufactured solutions using either differential or

integral methods.

4.8.5 Streamer v3.0 Verification

Verification of `BACL-Streamer` v3.0 is ongoing, and has been done only in preliminary cases for the basic Euler equations, as was described in chapter 3. This preliminary verification has demonstrated the accuracy of the underlying solvers for stationary, Cartesian grids, but more work is needed in order to extend this confidence to the full UCS equations.

4.9 Future Developments

UCS has shown promise for reducing the costs of CFD while raising accuracy around shocks and slip lines [21, 53, 54, 59], but more work needs to be done before it can be considered a mature methodology. The preliminary results have been encouraging, but a mature code base is badly needed for both verification tests and demonstration problems. The most difficult aspects of this are the implementation of boundary conditions. Connecting the unsteady, computational coordinates in the which the UCS equations are solved with the steady, physical coordinates in which the boundary conditions are defined has proven quite challenging.

4.9.1 A first cut at better boundary conditions

Consider a two-dimensional duct flow containing a ramp, as in Fig. 4.6. How would one apply boundary conditions for this problem? The inflow, outflow, and top wall conditions are simple enough, and can be easily implemented as array operations:

$$\mathbf{W}(0,j) = \mathbf{W}_{in}(1,j) \tag{4.73}$$

$$\mathbf{W}(ni+1,j) = \mathbf{W}(ni,j) \tag{4.74}$$

$$\mathbf{W}(nj+1,i) = refl(\mathbf{W}(nj,i), \theta = 0) \tag{4.75}$$

where $refl$ indicates vector reflection across the wall boundary, imposing a symmetry condition at the top wall at zero degrees from horizontal.

Things become more difficult at the bottom wall. For example, one might try:

```
do i = 1, ni
if(cell(i)%x < start_ramp .or. &
cell(i)%x > end_ramp)then
```

$$\mathbf{W}\left(ni+1,j\right)=\mathbf{W}\left(ni,j\right);\ \mathbf{W}\left(nj+1,i\right)=refl\left(\mathbf{W}\left(nj,i\right),\theta=0\right)$$

```
else
```

$$\mathbf{W}\left(ni+1,j\right)=\mathbf{W}\left(ni,j\right);\ \mathbf{W}\left(nj+1,i\right)=refl\left(\mathbf{W}\left(nj,i\right),\theta=\theta_{ramp}\right)$$

```
end if
end do
```

This is functional for simple geometries, but it quickly becomes unmanageable for complex geometries containing many different surfaces with varying reflection angles, and it is also computationally expensive, requiring conditional evaluation of each and every cell that lies on a boundary.

These problems can be managed simultaneously, by exploiting some useful features of the unified coordinate system, which does not specify anything in particular about the spatial step size in computational coordinates. In particular, it is possible to require

$$\Delta\xi=\Delta\eta=\Delta\zeta=1 \tag{4.76}$$

It is of course possible to use any other constant as well, but a value of one is particularly convenient, because it provides a direct mapping from array indices to computational coordinates. For example, given an array of defined shape and starting coordinates $\mathbf{W}_{ijk};\ (\xi_{111},\ \eta_{111},\ \zeta_{111})=(\xi_0,\ \eta_0,\ \zeta_0)$ it is possible to write a relation allowing the computational coordinates of a cell to be determined solely from the array indices:

$$(\xi,\eta,\zeta)_{ijk}=(i+\xi_0,\ j+\eta_0,\ k+\zeta_0) \tag{4.77}$$

Using this approach, the code needs only to convert the physical coordinate representation of the boundary conditions to unsteady computational coordinates by extrapolating the grid metric from

the nearest cells. This results in a loop over boundary conditions to compute updated computational coordinates, rather than a loop over cells to evaluate complex conditional statements. Once the computational coordinates of the boundary conditions are determined, they may be applied directly to the appropriate cells using the computational coordinates as defined in Eq. 4.77. This may be done as a simple array slice operation, with no conditional evaluation required. The principal difficulty here lies in accurately estimating the computational coordinates of boundary condition elements, and in appropriately assigning elements to grid cells. Some preliminary work in this direction has been attempted, but it remains highly experimental.

4.9.2 Singular points in UCS flows

An additional difficulty is introduced to simulations as a result of the unsteady grid used by the unified coordinate system. Considering again the ramp problem, it is quickly apparent that the location of the leading edge of the ramp has a great effect on the entire downstream flow. The same situation occurs with the two-dimensional Riemann problem in figure 4.11. Unfortunately, with the grid moving at an unpredictable rate, these singular points never align exactly with grid points, which automatically introduces error $O(\Delta\xi)$ in the flow. Various techniques are possible to reduce this error. One is to choose time step values and grid velocities such that the singular point always coincides with a grid point. Another is to introduce an intermediate grid point that remains at the singular point. A third is to manipulate grid velocity so that the grid point at the singular point becomes stationary after the grid has been generated, similar to what Hui does with the viscous boundary layer [59]

4.9.3 Accurate adherence to boundary surfaces

A final difficulty with the unified coordinates is that there is no guarantee that grid points will remain close to simulation boundaries. Since grid points are mobile, and the evolution of the grid metric depends only on the grid velocity, it is very possible for the physical coordinates of the grid to move quite far away from the actual physical boundaries, as will be discussed further in

Section 4.8.3. The remedy is to apply grid motion controls that take account of the actual position of the boundary conditions. It should be possible, for instance, to include a small forcing term that applies pressure to the grid to fill out boundary conditions. If it is important that grid cells be exactly coincident with boundary surfaces, then grid points can be constrained to move along the boundary surface itself, or the metric components can be adjusted to fit.

4.9.4 Dynamic grid separation into structured blocks

The UCS transformation itself is highly specific to structured grids, which provides many advantages, however it does complicate matters when dealing with irregular geometries. Because of the inherent flow-oriented nature of the grid, UCS is principally suited to H-type grids rather than C- or O- type. This means that the use of block-structured grids is inevitable for many kinds of flows, including flow around an embedded surface such as a wing, or flow through a round channel. In particular, flow around embedded surfaces requires the dynamic detection of the leading edge geometry and the division of the grid into separate blocks on the fly. This is made more difficult by the fact that the advancing flow at the advancing edge of the grid is typically unsteady, and it is not always apparent which grid cells should pass on which side of the embedded surface.

4.9.5 Verification of Streamer v3.0

Preliminary results from verification of `BACL-Streamer` v3.0 have been very encouraging, and provide confidence in the accuracy of the code within the coverage of the tests. The principal next steps for this work are a more complete code verification of `BACL-Streamer` v3.0, followed by an in-depth investigation into the effects of using the UCS method to find solutions to fluid dynamical systems. The accuracy of the underlying time-step-eulerian approximation, the suitability of different boundary condition implementations, and the effects of different grid motion schemes, all need to be studied and understood, independently of the underlying flow solvers. This knowledge of how UCS affects simulation accuracy will guide the development of new solution algorithms, and it will provide designers with enough confidence in the method to use it on new projects.

4.10 Conclusion

UCS continues to show promise for enabling fast, accurate, design-oriented CFD computations, by eliminating the need for explicit grid generation and by improving the resolution of flow discontinuities. UCS also introduces several unique challenges for CFD codes, as a result of the moving computational grid. Overcoming these implementation challenges is critical to the eventual success of UCS for design CFD, and requires further investigation. The research and development of `BACL-Streamer` demonstrates several different approaches to this problem, as well as a steady investigation of the effects of using unsteady coordinates to model physical problems.

Chapter 5

Conclusion

Thousands of years of successive design successes have given us the modern world we live in today. These designs, arising from bursts of inspiration, mathematical analysis, or trial-and-error, have allowed humanity to achieve ever-greater things, from cell-phones to the United Nations to the International Space Station. However, as the demands we place on our devices, systems, and organizations continue to become more stringent, we are discovering that we need better and better design tools in order to develop workable solutions in an increasingly constrained design space. This is simply the result of the millennia we have spent pushing outward against the limits of possibility. Fortunately, better design tools continue to become available.

The development of numerical simulation tools during the twentieth century allowed complex mathematical models to be applied to design problems for the first time, and has led to a dramatic increase in the use of physics-based modeling to directly predict the behavior of complex systems. The reduction in uncertainty that these tools have provided has allowed us to identify workable design points in constrained design spaces in the fields of physics, engineering, economics, business analytics, politics, and many more. In turn, this has allowed us to accomplish things that would have never been possible before. Unfortunately, these tools are not currently being used early in project design cycles where they could be extremely useful.

Numerical simulation tools have traditionally suffered from two challenges that have prevented them from being used in early-stage design work. First, they are typically unwieldy, requiring the attention of dedicated experts in order to use them effectively. These large startup costs make

the use of numerical simulations difficult when design requirements are fluid and turnaround times are critical. Second, numerical simulation tools are only able to reduce uncertainty, not eliminate it, and it can be very difficult to know exactly how much uncertainty remains. This makes it difficult to rely on the results of simulations in order to make design decisions that can directly lead to project success or failure. In this work, I have sought to address both of these issues directly, by developing tools for the direct measurement of error in numerical simulation tools and by developing a method for computational fluid dynamics that dispenses with the otherwise arduous process of grid generation.

5.1 Verification of Numerical Simulations

Since the initial forays into numerical simulation of mathematical models nearly a century ago, researchers have been working steadily to increase both the resolution and the fidelity of the results. Unfortunately, the related work on verification of numerical simulations (evaluating their accuracy) has lagged somewhat behind these advances. It is only recently that techniques have been developed to directly measure the accuracy with which a given simulation code obtains solutions to the underlying mathematical model. In particular, the method of manufactured solutions (MMS) has been especially influential, allowing the general-purpose verification of a broad class of numerical simulation codes.

Unfortunately, MMS was unable to provide verification of codes in the presence of discontinuous solutions, such as those that are encountered in compressible fluid flows. The work done by Grier [15] and the separate project done by me (see chapter 3) represent the first successful extensions of MMS to include discontinuous solutions, and they therefore provide the first complete, general-purpose verification technique for shock-capturing fluid codes. Both of these projects rely on an integral formulation of MMS, and they therefore require the use of fast, accurate, numerical quadrature routines.

My work differs from that of Grier primarily in the means by which the MMS integral fluxes are evaluated. Multidimensional integrals of discontinuous functions are difficult to evaluate ac-

curately, and we have both devised ways to alleviate this difficulty. Grier et al demonstrate their technique for two-dimensional problems by subdividing an integration domain within which the integrand is discontinuous into subdomains that can be mapped to quadrilateral and triangular elements, upon which the smooth portions of the integral can be evaluated. In contrast, I use iterated one-dimensional integration operators for multidimensional integration, which allows me to leverage the properties of one-dimensional space to simplify the problem of of subdividing integration domains. Both schemes have the intended effect of fitting the discontinuities exactly, but it is unclear how Grier's approach will generalize to higher-dimensional integrals. In contrast, my integration approach has already been demonstrated with integrals up to five dimensions, and is limited in higher dimensions only by available computational power.

I have also demonstrated the use of integral manufactured solutions in the verification of the `BACL-Streamer` code, comparing computed convergence rates with those obtained by traditional MMS and those obtained using discontinuous, exact solutions. I have demonstrated that integral MMS is equivalent to traditional MMS for smooth problems, and I have shown that it performs equally well for discontinuous problems. Additional testing is needed to further prove the utility of integral MMS, but I am confident in recommending the technique for verification of codes where traditional MMS is not sufficient to represent the underlying mathematical model, as is the case with discontinuous solutions.

There are two issues with my implementation of integral MMS that need to be addressed in future work. First, my current implementation of discontinuous integration, described in chapter 2, is inefficient. This causes problems when attempting to compute integral manufactured solutions over a large computational grid, and these inefficiencies should be resolved in order to boost the speed of integral MMS for large verification problems. Second, the integral form of manufactured solutions needs to be made publicly available for developers. The best way to accomplish this is to incorporate integral MMS into the established MASA [19] library for MMS verification. These projects are connected, as integral optimizations will likely ease the process of adapting my code to fit more naturally into the MASA library.

5.2 Automatic Grid Generation and UCS

If verification procedures for numerical simulations have lagged behind increases in resolution and fidelity, the same is no less true of procedures for simplifying the process of setting up numerical simulations for computation, especially the process of grid generation. Unfortunately, this has been a much more difficult problem to solve, and the solutions that have been developed so far must make many compromises. As a result, numerical simulation tools continue to require dedicated, hands-on work from specialists in order to set up a simulation that will yield accurate results. In contrast to verification, however, there is no clear path forward on this issue, which means that we must continue to experiment with various possible strategies in order to find a solution. My work with the unified coordinates (UCS) provides one such experiment, in the context of computational fluid dynamics.

UCS works by using an unsteady, structured-grid transformation in order to avoid the need for grid generation entirely. By tying grid motion to the motion of the fluid itself, a trivial grid at the upstream boundary will eventually flow to fill the entire simulation region, without any input on the part of the user. UCS has the beneficial side effect of providing a grid that is aligned with fluid streamlines, which can lead to dramatic increases in solution accuracy for some flows, particularly those dominated by slip lines.

Unfortunately, the primary developer behind UCS retired shortly after publishing his initial findings, and much of the institutional knowledge contained in that group has been lost. As a result, much of my research has consisted of rediscovering the characteristics, techniques, and experience required in order to use UCS effectively for fluid simulations. I have also developed several implementations of UCS in order to resolve to unique difficulties that arise when dealing with computation on unsteady grids. I have demonstrated the ability of UCS to forego grid generation with several example problems, and I have also presented the first preliminary verification of UCS code using code order verification.

Future work with UCS should be based around three principal ideas. First, the simple method

of controlling grid motion first presented by Hui [53] encounters difficulties when used to simulate flows around expansion corners, and a more robust grid motion scheme is needed. Second, boundary conditions can be problematic in general, multidimensional simulations with unsteady grids. There is a definite need for preprocessing techniques that simplify the specification of simulation boundary conditions. Finally, the actual convergence properties of simulations based on UCS are unknown in general. UCS badly needs rigorous verification in order to quantify the error that is introduced into simulations through the use of unified coordinates. As before, these needs are linked, as changes in grid motion will necessarily affect both code verification and boundary condition specification.

5.3 Conclusion

In summary, my contributions to the state-of-the-art for design-oriented simulations are:

(1) I designed a new algorithm for fast, accurate evaluation of multidimensional integrals of discontinuous functions.

(2) I designed, developed, and published an extension to the SciPy integration library that enables the recursive evaluation of n-dimensional integrals with full user control at all levels of integration.

(3) I designed and developed code that automates the entire process of computing manufactured source terms for both differential and integral MMS.

(4) I used the code I developed to verify the stationary-grid functionality of the `BACL-Streamer` software package using both differential and integral manufactured solutions.

(5) I compared the results of MMS verification to verification of `BACL-Streamer` with exact solutions, and also with verification of an independent, third-party flow solver.

(6) I have established the theoretical fundamentals of UCS for solving the inviscid fluid equations by presenting, in detail, the derivations that are hinted at in previous, published work.

(7) I have explored various ways to implement working code that handles the unique requirements of unsteady grids such as those used in UCS.

(8) I have demonstrated the ways in which UCS can be used to eliminate the need for explicit grid generation in computational fluid dynamics.

(9) I have identified the unique challenges faced by any UCS code, and laid a solid foundation for future work in this area.

The future of the world, like the past, will be built on the work of designers who create ever-more-efficient tools to magnify the efforts of ordinary people. As the march of progress continues to push the boundaries of possibility, the design environment in which these men and women work will continue to become progressively more constrained, and the risks of failure will continue to increase. In order to mitigate these risks, designers will come to depend more and more on sophisticated tools for predicting system behavior, and these tools will need to be both seamless to use and computationally reliable. Through the work I present here, I have contributed to this goal in my own small way. The future of the world may be built on designers, but the designers themselves will build on the tools we create to help them do their work.

Bibliography

[1] Culick, F. E. C., "The Wright Brothers: First Aeronautical Engineers and Test Pilots," AIAA Journal, Vol. 41, No. 6, 2003, pp. 985–1006, Paper No. AIAA 2003-3385.

[2] Howard, F., Wilbur and Orville: A Biography of the Wright Brothers, Ballantine Books, New York, 1987.

[3] Muller, G. and Gill, E., "Applied Space Systems Engineering," Applied Space Systems Engineering, edited by W. J. Larson, D. Kirkpatrick, J. J. Sellers, L. D. Thomas, and D. Verma, chap. 10, McGraw Hill, Boston, 1st ed., 2009, pp. 351–384.

[4] Wendt, J. F., editor, Computational Fluid Dynamics: An Introduction, Springer, 2009.

[5] Hirsch, C., Numerical Computation of Internal and External Flows, Butterworth-Heinemann, Oxford, 2nd ed., 2007.

[6] Jameson, A., "Re-Engineering the Design Process Through Computation," Journal of Aircraft, Vol. 36, 1997, pp. 36–50.

[7] Martins, J. R. R. a. and Lambe, A. B., "Multidisciplinary Design Optimization: A Survey of Architectures," AIAA Journal, Vol. 51, No. 9, 2013, pp. 2049–2075.

[8] Washington State Department of Transportation, "Systems Engineering for Intelligent Transportation Systems," http://www.wsdot.wa.gov/publications/manuals/fulltext/M22-01/SystemsEngineering.pdf, 2005, Accessed: 2015-04-17.

[9] Roache, P. J., Verification and Validation in Computational Science and Engineering, Hermosa Publishers, 1998.

[10] Oberkampf, W. L. and Roy, C. J., Verification and Validation in Scientific Computing, Cambridge University Press, Cambridge, UK, 2010.

[11] Committee on Mathematical Foundations of Verification Validation and Uncertainty Quantification, "Assessing the Reliability of Complex Models: Mathematical and Statistical Foundations of Verification, Validation, and Uncertainty Quantification," Tech. rep., National Research Council, 2012.

[12] Roache, P. J., "Quantification of Uncertainty in Computational Fluid Dynamics," Annual Review of Fluid Mechanics, Vol. 29, 1997, pp. 123–160.

[13] Roy, C. J., "Review of code and solution verification procedures for computational simulation," Journal of Computational Physics, Vol. 205, No. 1, May 2005, pp. 131–156.

[14] Malaya, N., "Personal Communication," 2014.

[15] Grier, B., Alyanak, E., White, M., Camberos, J., and Figliola, R., "Numerical integration techniques for discontinuous manufactured solutions," Journal of Computational Physics, Vol. 278, 2014, pp. 193–203.

[16] Woods, C. N. and Starkey, R. P., "Verification of Fluid - Dynamic Codes in the Presence of Shocks and Other Discontinuities," Journal of Computational Physics, Vol. 294, Aug. 2015, pp. 312–328.

[17] Derlaga, J. M., Phillips, T. S., Roy, C. J., and Tech, V., "SENSEI Computational Fluid Dynamics Code : A Case Study in Modern Fortran Software Development," 21st AIAA Computational Fluid Dynamics Conference, 2013, Paper No. AIAA 2013-2450.

[18] Knupp, P. and Salari, K., Verification of Computer Codes in Computational Science and Engineering, Chapman and Hall/CRC, 2002.

[19] Malaya, N., Estacio-Hiroms, K. C., Stogner, R. H., Schulz, K. W., Bauman, P. T., and Carey, G. F., "MASA : a library for verification using manufactured and analytical solutions," Engineering with Computers, Vol. 29, No. 4, Oct. 2012, pp. 487–496.

[20] Habashi, H. G., Dompierre, J., Bourgault, Y., Ait-Ali-Yahia, D., Fortin, M., Vallet, M. G., and Habashi, W. G., "Anisotropic mesh adaptation: towards user-indipendent, mesh-indipendent and solver-indipendent CFD. Part I: General principles," Int. J. Numer. Meth. Fluids, Vol. 32, 2000, pp. 725–744.

[21] Hui, W. H., "The Unified Coordinate System in Computational Fluid Dynamics," Communications in Computational Physics, Vol. 2, No. 4, 2007, pp. 577–610.

[22] Malaya, N., Estacio-Hiroms, K. C., Stogner, R. H., Schulz, K. W., Bauman, P. T., Carey, G. F., and Al, E., "MASA: Manufactured Analytical Solution Abstraction Repository," https://github.com/manufactured-solutions/MASA.

[23] Boisvert, R. F., Howe, S. E., Kahaner, D. K., and Springmann, J. L., "Guide to Available Mathematical Software," http://gams.nist.gov, 1990, Accessed: 2015-02-18.

[24] AT&T Bell Laboratories, University of Tennessee, and Oak Ridge National Laboratory, "Netlib," http://www.netlib.org/, Accessed: 2015-14-17.

[25] Vandevender, W. H. and Haskell, K. H., "The SLATEC Mathematical Subroutine Library," ACM SIGNUM Newsletter, Vol. 17, No. 3, 1982, pp. 16–21.

[26] Piessens, R., de Doncker-Kapenga, E., Uberhuber, C. W., and Kahaner, D. K., Quadpack: A Subroutine Package for Automatic Integration, Springer-Verlag, Berlin Heidelberg, 1983.

[27] Gough, B., GNU Scientific Library Reference Manual, Network Theory Ltd., 3rd ed., 2009.

[28] SciPy, "SciPy GitHub Repository," https://github.com/scipy/scipy.

[29] Phillips, J., The NAG Library: A Beginners Guide, Oxford University Press, Inc., New York, 1987.

[30] Wolfram Language and System Documentation Center, "Notes on Internal Implementation," http://reference.wolfram.com/language/tutorial/SomeNotesOnInternalImplementation.html, Accessed: 2015-03-16.

[31] Hahn, T., "Cuba - a library for multidimensional numerical integration," Computer Physics Communications, Vol. 168, No. 2, April 2004, pp. 78–95.

[32] Johnson, S. G., "Cubature (Multi-dimensional Integration)," http://ab-initio.mit.edu/wiki/index.php/Cubature, Accessed: 2015-03-16.

[33] Oliphant, T. E., "SciPy: Open source scientific tools for Python," Computing in Science and Engineering, Vol. 9, 2007, pp. 10–20.

[34] Polyanin, A. D., Handbook of Linear Partial Differential Equations for Engineers and Scientists, Chapman and Hall/CRC, Boca Raton, FL, 2002.

[35] Toro, E., Riemann Solvers and Numerical Methods for Fluid Dynamics, Springer, 2009.

[36] Roache, P. J., "Code Verification by the Method of Manufactured Solutions," Journal of Fluids Engineering, Vol. 124, No. 1, 2002, pp. 4.

[37] Roy, C., Ober, C., and Smith, T., "Verification of a Compressible CFD Code Using the Method of Manufactured Solutions," 32nd AIAA Fluid Dynamics Conference and Exhibit, 2002, Paper No. AIAA 2002-3110.

[38] Char, B. W., Geddes, K. O., Gonnet, G. H., Leong, B. L., Monagan, M. B., and Watt, S. M., Maple V: Language Reference Manual, Springer, New York, 1991.

[39] Caflisch, R. E., "Monte Carlo and quasi-Monte Carlo methods," Acta Numerica, Vol. 7, 1998, pp. 1–49.

[40] Toro, E. F., "NUMERICA: a Library of Source Codes for Teaching, Research and Applications. HYPER-EUL. Methods for the Euler equations." http://www.ing.unitn.it/ toroe/terms.html, 1999.

[41] Banks, J., Aslam, T., and Rider, W., "On sub-linear convergence for linearly degenerate waves in capturing schemes," Journal of Computational Physics, Vol. 227, No. 14, July 2008, pp. 6985–7002.

[42] Sabac, F., "The Optimal Convergence Rate of Monotone Finite Difference Methods for Hyperbolic Conservation Laws," SIAM Journal on Numerical Analysis, Vol. 34, No. 6, 1997, pp. 2306–2318.

[43] Popov, B. and Trifonov, O., "Order of convergence of second order schemes based on the minmod limiter," Mathematics of Computation, Vol. 75, No. 256, 2006, pp. 1735–1753.

[44] Woods, C. N. and Starkey, R. P., "Verification and Application of the Unified Coordinate System in Preliminary Design," 20th AIAA Computational Fluid Dynamics Conference, No. June, 2011, Paper No. AIAA 2011-3383.

[45] Woods, C. N. and Starkey, R. P., "A Unified CFD Approach to High-Speed Component Design," 47th AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit, No. August, 2011, Paper No. AIAA 2011-6109.

[46] Mavriplis, D. J., Vassberg, J. C., Tinoco, E. N., Mani, M., Brodersen, O. P., Eisfeld, B., Wahls, R. a., Morrison, J. H., Zickuhr, T., Levy, D., and Murayama, M., "Grid Quality and Resolution Issues from the Drag Prediction Workshop Series," Journal of Aircraft, Vol. 46, No. 3, 2009, pp. 935–950.

[47] Steinthorsson, E. and Modiano, D., "Advanced Methodology for Simulation of Complex Flows Using Structured Grid Systems," Surface Modeling, Grid Generation, and Related Issues in Computational Fluid Dynamic (CFD) Solutions, 1995, pp. 697–710.

[48] Johnson, F. T., Tinoco, E. N., and Yu, N. J., "Thirty years of development and application of CFD at Boeing Commercial Airplanes, Seattle," Computers and Fluids, Vol. 34, No. 10, 2005, pp. 1115–1151.

[49] Badcock, K. J., Richards, B. E., and Woodgate, M. A., "Elements of computational fluid dynamics on block structured grids using implicit solvers," Progress in Aerospace Sciences, Vol. 36, No. 5, 2000, pp. 351–392.

[50] Garretson, D., Mair, H., Martin, C., Sullivan, K., and Teichman, J., "Review of CFD Capabilities," Tech. rep., Institute for Defense Analyses, Science & Technology Division, Alexandria, VA, 2005.

[51] Chawner, J., "Quality and Control - Two Reasons Why Structured Grids Aren't Going Away," http://www.pointwise.com/theconnector/March-2013/Structured-Grids-in-Pointwise.shtml, 2013, Accessed: 2015-04-17.

[52] Wang, T.-S., "Multidimensional Unstructured-Grid Liquid Rocket-Engine Nozzle Performance and Heat Transfer Analysis," Journal of Propulsion and Power, Vol. 22, No. 1, 2006, pp. 78–84.

[53] Hui, W., Li, P., and Li, Z., "A unified coordinate system for solving the two-dimensional Euler equations," Journal of Computational Physics, Vol. 153, 1999, pp. 596–637.

[54] Hui, W. and Kudriakov, S., "A Unified Coordinate System for Solving the Three-Dimensional Euler Equations," Journal of Computational Physics, Vol. 172, No. 1, Sept. 2001, pp. 235–260.

[55] Hui, W.-H. and Xu, K., Computational Fluid Dynamics Based on the Unified Coordinates, Springer, Heidelberg, 1st ed., 2012.

[56] Venkatakrishnan, V., "Perspective on Unstructured Grid Flow Solvers," AIAA Journal, Vol. 34, No. 3, March 1996, Paper No. AIAA 95-0667.

[57] Hui, W. H., Hu, J. J., and Zhao, G. P., "Gridless Computation Using the Unified Coordinates," Computational Fluid Dynamics 2004: Proceedings of the Third Integrational Conference on Computational Fluid Dynamics, ICCFD3, Toronto, 12-16 July 2004, No. 2, Springer, Berlin Heidelberg, 2006, pp. 503–508.

[58] Hui, W. H., Zhao, G. P., Hu, J., and Zheng, Y., "Gridless Computation of Two Dimensional Flow Using the Unified Coordinates," April 2005, Preprint submitted to El Sevier Science.

[59] Hui, W. H., Wu, Z. N., and Gao, B., "Preliminary Extension of the Unified Coordinate System Approach to Computation of Viscous Flows," Journal of Scientific Computing, Vol. 30, No. 2, Oct. 2007, pp. 301–344.

[60] Jia, P., Jiang, S., and Zhao, G., "Two-Dimensional Compressible Multimaterial Flow Calculations in a Unified Coordinate System," Computers & Fluids, Vol. 35, No. 2, Feb. 2006, pp. 168–188.

[61] Zhilkin, a. G., "A dynamic mesh adaptation method for magnetohydrodynamics problems," Computational Mathematics and Mathematical Physics, Vol. 47, No. 11, Nov. 2007, pp. 1819–1832.

[62] Jin, C. and Xu, K., "A unified moving grid gas-kinetic method in Eulerian space for viscous flow computation," Journal of Computational Physics, Vol. 222, No. 1, March 2007, pp. 155–175.

[63] Jin, C. and Xu, K., "Numerical Study of the Unsteady Aerodynamics of Freely Falling Plates 1 Introduction," Communications in Computational Physics, Vol. 3, No. 4, 2008, pp. 834–851.

[64] Kudriakov, S., Unified Coordinates and Resolution of Discontinuities for Euler and Shallow Water Equations, Ph.D. thesis, Hong Kong University of Science and Technology, 2000.

[65] Leung, B. and Yu, S., Numerical Experiments on the Unified Coordinates System for Two Dimensional Steady Flow, Ph.D. thesis, Hong Kong University of Science and Technology, 2001.

[66] Weintraub, S. H., Differential Forms–A Complement to Vector Calculus, Academic Press, 1997.

[67] Flanders, H., Differential Forms with Applications to the Physical Sciences, Academic Press, 1963.

[68] Preston, S. C., "An Introduction to Differential Geometry," http://math.colorado.edu/ prestos/math6230/diffgeotext.pdf, 2013, Accessed 2015-04-17.

[69] Thomas, P. D. and Lombard, C. K., "Geometric Conservation Law and Its Application to Flow Computations on Moving Grids," AIAA Journal, Vol. 17, No. 10, Oct. 1979, pp. 1030–1037, Paper No. AIAA 78-1208R.

[70] Schlichting, H. and Gersten, K., Boundary Layer Theory, Springer-Verlag, New York, 8th ed., 2000.

[71] M.A.T.S., "F-14 Inlet," http://www.anft.net/f-14/f14-detail-airintake.htm, Accessed: 2011-11-25.

[72] Toro, E., "The development of a Riemann solver for the steady supersonic Euler equations," Aeronautical Journal, Vol. 98, 1994, pp. 325–339.

[73] Peterson, P., "F2PY: a tool for connecting Fortran and Python programs," International Journal of Computational Science and Engineering, Vol. 4, No. 4, 2009, pp. 296.

Appendix A

scipy.integrate.nquad

This is the source code for the `nquad` function, as documented in chapter 2 and included in SciPy as of version 0.15, and available at https://github.com/scipy/scipy.

```
def nquad(func, ranges, args=None, opts=None):
    """
    Integration over multiple variables.

    Wraps `quad` to enable integration over multiple variables.
    Various options allow improved integration of discontinuous functions, as
    well as the use of weighted integration, and generally finer control of the
    integration process.

    Parameters
    ----------
    func : callable
        The function to be integrated. Has arguments of ``x0, ... xn``,
        ``t0, tm``, where integration is carried out over ``x0, ... xn``, which
        must be floats.  Function signature should be
        ``func(x0, x1, ..., xn, t0, t1, ..., tm)``.  Integration is carried out
        in order.  That is, integration over ``x0`` is the innermost integral,
        and ``xn`` is the outermost.
        If performance is a concern, this function may be a ctypes function of
        the form::

            f(int n, double args[n])

        where ``n`` is the number of extra parameters and args is an array
        of doubles of the additional parameters.  This function may then
        be compiled to a dynamic/shared library then imported through
        ``ctypes``, setting the function's argtypes to ``(c_int, c_double)``,
        and the function's restype to ``(c_double)``.  Its pointer may then be
        passed into `nquad` normally.
```

```
    This allows the underlying Fortran library to evaluate the function in
    the innermost integration calls without callbacks to Python, and also
    speeds up the evaluation of the function itself.
ranges : iterable object
    Each element of ranges may be either a sequence  of 2 numbers, or else
    a callable that returns such a sequence.  ``ranges[0]`` corresponds to
    integration over x0, and so on.  If an element of ranges is a callable,
    then it will be called with all of the integration arguments available.
    e.g. if ``func = f(x0, x1, x2)``, then ``ranges[0]`` may be defined as
    either ``(a, b)`` or else as ``(a, b) = range0(x1, x2)``.
args : iterable object, optional
    Additional arguments ``t0, ..., tn``, required by `func`.
opts : iterable object or dict, optional
    Options to be passed to `quad`.  May be empty, a dict, or
    a sequence of dicts or functions that return a dict.  If empty, the
    default options from scipy.integrate.quadare used.  If a dict, the same
    options are used for all levels of integraion.  If a sequence, then each
    element of the sequence corresponds to a particular integration. e.g.
    opts[0] corresponds to integration over x0, and so on. The available
    options together with their default values are:

      - epsabs = 1.49e-08
      - epsrel = 1.49e-08
      - limit  = 50
      - points = None
      - weight = None
      - wvar   = None
      - wopts  = None

    The ``full_output`` option from `quad` is unavailable, due to the
    complexity of handling the large amount of data such an option would
    return for this kind of nested integration.  For more information on
    these options, see `quad` and `quad_explain`.

Returns
-------
result : float
    The result of the integration.
abserr : float
    The maximum of the estimates of the absolute error in the various
    integration results.

See Also
--------
quad : 1-dimensional numerical integration
dblquad, tplquad : double and triple integrals
fixed_quad : fixed-order Gaussian quadrature
```

```
quadrature : adaptive Gaussian quadrature

Examples
--------
>>> from scipy import integrate
>>> func = lambda x0,x1,x2,x3 : x0**2 + x1*x2 - x3**3 + np.sin(x0) + (
...                                 1 if (x0-.2*x3-.5-.25*x1>0) else 0)
>>> points = [[lambda (x1,x2,x3) : 0.2*x3 + 0.5 + 0.25*x1], [], [], []]
>>> def opts0(*args, **kwargs):
...     return {'points':[0.2*args[2] + 0.5 + 0.25*args[0]]}
>>> integrate.nquad(func, [[0,1], [-1,1], [.13,.8], [-.15,1]],
...                 opts=[opts0,{},{},{}])
(1.5267454070738633, 2.9437360001402324e-14)

>>> scale = .1
>>> def func2(x0, x1, x2, x3, t0, t1):
...     return x0*x1*x3**2 + np.sin(x2) + 1 + (1 if x0+t1*x1-t0>0 else 0)
>>> def lim0(x1, x2, x3, t0, t1):
...     return [scale * (x1**2 + x2 + np.cos(x3)*t0*t1 + 1) - 1,
...             scale * (x1**2 + x2 + np.cos(x3)*t0*t1 + 1) + 1]
>>> def lim1(x2, x3, t0, t1):
...     return [scale * (t0*x2 + t1*x3) - 1,
...             scale * (t0*x2 + t1*x3) + 1]
>>> def lim2(x3, t0, t1):
...     return [scale * (x3 + t0**2*t1**3) - 1,
...             scale * (x3 + t0**2*t1**3) + 1]
>>> def lim3(t0, t1):
...     return [scale * (t0+t1) - 1, scale * (t0+t1) + 1]
>>> def opts0(x1, x2, x3, t0, t1):
...     return {'points' : [t0 - t1*x1]}
>>> def opts1(x2, x3, t0, t1):
...     return {}
>>> def opts2(x3, t0, t1):
...     return {}
>>> def opts3(t0, t1):
...     return {}
>>> integrate.nquad(func2, [lim0, lim1, lim2, lim3], args=(0,0),
                    opts=[opts0, opts1, opts2, opts3])
(25.066666666666666, 2.7829590483937256e-13)

"""
depth = len(ranges)
ranges = [rng if callable(rng) else _RangeFunc(rng) for rng in ranges]
if args is None:
    args = ()
if opts is None:
    opts = [dict([])] * depth
```

```
    if isinstance(opts, dict):
        opts = [opts] * depth
    else:
        opts = [opt if callable(opt) else _OptFunc(opt) for opt in opts]

    return _NQuad(func, ranges, opts).integrate(*args)


class _RangeFunc(object):
    def __init__(self, range_):
        self.range_ = range_

    def __call__(self, *args):
        """Return stored value.

        *args needed because range_ can be float or func, and is called with
        variable number of parameters.
        """
        return self.range_


class _OptFunc(object):
    def __init__(self, opt):
        self.opt = opt

    def __call__(self, *args):
        """Return stored dict."""
        return self.opt


class _NQuad(object):
    def __init__(self, func, ranges, opts):
        self.abserr = 0
        self.func = func
        self.ranges = ranges
        self.opts = opts
        self.maxdepth = len(ranges)

    def integrate(self, *args, **kwargs):
        depth = kwargs.pop('depth', 0)
        if kwargs:
            raise ValueError('unexpected kwargs')

        # Get the integration range and options for this depth.
        ind = -(depth + 1)
        fn_range = self.ranges[ind]
```

```
low, high = fn_range(*args)
fn_opt = self.opts[ind]
opt = dict(fn_opt(*args))

if 'points' in opt:
    opt['points'] = [x for x in opt['points'] if low <= x <= high]
if depth + 1 == self.maxdepth:
    f = self.func
else:
    f = partial(self.integrate, depth=depth+1)

value, abserr = quad(f, low, high, args=args, **opt)
self.abserr = max(self.abserr, abserr)
if depth > 0:
    return value
else:
    # Final result of n-D integration with error
    return value, self.abserr
```

Appendix B

BACL-Manufactured

I have developed a software package that greatly simplifies the computation of both integral and differential manufactured source terms for conservation and balance laws, as discussed in chapter 2 and chapter 3. The code is organized into separate modules that handle numerical integration, general balance laws, and specific balance laws. The most current code for these is available on `https://github.com/woodscn/BACL-MMS`, but a snapshot has been reproduced here.

B.1 `integration.py`

The `integration.py` module is responsible for quickly and accurately evaluating the multidimensional integral of a potentially discontinuous Sympy function using the numerical methods available in the SciPy library. It handles symbolic critical surface processing, which is used to generate function calls for `scipy.integrate.nquad`.

```
import os
import subprocess
import ctypes
import random
import sympy
import numpy
from sympy.utilities.lambdify import lambdify
from sympy.utilities.codegen import codegen
from sympy.core.cache import clear_cache
from scipy.integrate import nquad
H = sympy.special.delta_functions.Heaviside
from functools import partial
```

```
def list_integral(integrands, **kwargs):
    """
Map numeric integration to a list of symbolic integrands.
Numerically integrate a list of symbolic integrands over a single range and
with a single set of integration options.
This is a good spot to implement multiprocessing using Python's
multiprocessing module.

Parameters
----------
integrands : iterable of sympy expressions
    List of integrand expressions
sympy_ranges : iterable
    List of ranges and Symbols e.g. ((t,tmin,tmax),(x,xmin,xmax))
sympy_discontinuities : iterable of sympy expressions, optional
    List of symbolic discontinuities e.g. (x/t-0.75). The discontinuity is
    assumed to be at disc == 0, where disc is the sympy expression.
args : dict
    Substitution dict for any additional arguments required for evaluation of
    integrands beyond integration variables given in sympy_ranges.
integrator : not yet implemented
    --
opts : not yet implemented
    --

Returns
-------
iterable of floats
    Results of the integration of integrands over sympy_ranges.
"""
    multi_list_integral = partial(_func_integral, **kwargs)
    out = map(multi_list_integral, integrands)
    # # You have to choose where you will apply multiprocessing, either here or
    # # else in the computation of flux and source integrals. At present, this
    # # is done at the flux/source level.
    # pool = Pool()
    # out = pool.map(multi_list_integral,integrands)
    # pool.close()
    # pool.join()
    return out


def _func_integral(integrand, **kwargs):
    """
Necessary wrapper function to enable use of multiprocessing module.
"""
```

```
    out = IntegrableFunction(integrand, **kwargs).integrate()[0]
    return out


class IntegrableFunction(object):
    """
Short Summary: (delete this heading)

Extended Summary: (delete this heading)
Does not allow much in the way of integration options, only providing 'points'.

Parameters
----------
'sympy_function' : Sympy expression
    Sympy expression for a possibly vector-valued function.
'sympy_ranges' : iterable
    List of Sympy variable ranges in order e.g. ((x,0,1),(y,0,1)).
'sympy_discontinuities' : Sympy expression, optional
    Sympy expressions for various discontinuities.
'args' : iterable, optional
    Any additional arguments required by sympy_function.
'integrator' : optional
    Specify an integration method. Unused at present.
'opts' : optional
    Specify function options. Unused at present.

Attributes
----------
'int_variables' : iterable
    List of Sympy Symbols representing the variables of integration.
'ranges' : iterable
    List of ranges, of the form ((xmin,xmax),(ymin,ymax),...).
'function' : lambda or ctypes function
    Function with signature f(*int_variables,*args).
'integrate' : callable
    Abstracted integration function, currently quadrature.
'symbolic_discontinuities' : iterable
    List of symbolic expressions for discontinuities.

"""

    def __init__(self, sympy_function, sympy_ranges, sympy_discontinuities=(),
                 args={}, integrator=None, opts=None):
        self.sympy_ranges = sympy_ranges
        self.int_variables, self.min_ranges, self.max_ranges = zip(
            *sympy_ranges)
        self.int_variables = list(self.int_variables)
```

```
        self.ranges = zip(self.min_ranges, self.max_ranges)
        self.args = args
        self.sympy_variables = self.int_variables
        self.integrand = Integrand(
            sympy_function, self.sympy_variables, args=self.args)
        self.function = self.integrand.lambdified # ctypesified
        self.integrate = self.quad_integrate
        # Unpack sympy_discontinuities into a list of points for nquad.

        # In order to be used by nquad, discontinuities must be put into a
        # form of functions of the integration variables. One-dimensional
        # integration effectively smooths a discontinuity, provided the path
        # of integration crosses the discontinuity. Effectively, this means
        # that any discontinuity will be smoothed by integration over a
        # particular variable, provided that the function describing the
        # discontinuity is dependent on that variable. An example may help.

        # Assume three discontinuities: [x = 0, x*y = 1, y-1 = 0]. The form of
        # these discontiuities will depend on the order of integration given
        # to nquad. If the integration is done as int(int(f(x,y),dx),dy), then
        # nquad will need the discontinuities in the form:
        # [[lambda y : 0, lambda y : 1/y],[lambda : 1]].
        # Conversely, if the order of integration is reversed to
        # int(int(f(x,y),dy),dx), then the discontinuities must be
        #  [[lambda x : 1/x, lambda x : 1],[lambda : 0]].

        # This segment of code unpacks the list of discontinuities into the
        # correct form based on the order of integration given by ranges.

        discs = Discontinuities(
            sympy_discontinuities, self.sympy_ranges, self.args)
        self.opts = []
        for ind, level in enumerate(self.ranges):
            if discs.nquad_disc_functions:
                self.opts.append(OptionsDict(
                    points=discs.nquad_disc_functions[ind]))
            else:
                self.opts.append({})
        return None

    def quad_integrate(self):
        '''
Integration using scipy.integrate
'''
        out = nquad(self.function, self.ranges, opts=self.opts)
        # except:
        #     print "Some kind of error in the call to nquad!"
```

```
        #     import pdb;pdb.set_trace()
        return out

    def mc_integrate(self):
        """
Integration using Monte Carlo

Monte Carlo integration is a robust, though inaccurate, method of integration.
mc_integrate evaluates function f ncalls number of times at random points, and
uses the sum of these evaluations to compute the integral of function f over
the rectangular volume specified by ranges. The error of the integral scales
as 1/sqrt(ncalls)
"""
        f_sum = 0
        for n in xrange(ncalls):
            coords_lst = []
            for rng in ranges:
                coords_lst.append(rng[0] + rng[1] * numpy.random.random())
            f_sum += self.function(coords_lst)
        vol = numpy.prod([float(rng[1]) - float(rng[0]) for rng in ranges])
        return vol / calls * f_sum


class Integrand(object):
    def __init__(self, sympy_function, sympy_variables, args={}):
        self.sympy_function = sympy_function.subs(args)
        self.sympy_variables = sympy_variables
        self.lambdified = lambdify(self.sympy_variables, self.sympy_function)
        clear_cache()
        # I need a unique way to identify the integrand libraries I will be
        # generating. id(self) works sort of, but it may or may not be unique
        # I think. I'm going to try a hash of the underlying sympy function,
        # but I can't allow negative numbers (C gets confused), so I have to
        # generate a positive integer instead. Stackoverflow had this idea:
        self.unique_id = ctypes.c_size_t(hash(self.sympy_function)).value
        # stackoverflow.com/questions/18766535/...
        #   positive-integer-from-python-hash-function
        filename_prefix = os.path.join(
            'integrand_libs', 'integrand' + str(self.unique_id))
        libname_prefix = os.path.join(
            'integrand_libs', '' + str(self.unique_id))
# Enable the use of ctypes objects in nquad, once multivariate ctypes objects
# are appropriate arguments for the QUADPACK library functions.
#        try:
#            self.generated_code = codegen(
#                ('integrand',self.sympy_function),'C',filename_prefix,
#                argument_sequence=self.sympy_variables,to_files=True)
```

```
#        except IOError:
#            import pdb;pdb.set_trace()
#        args_str = ",".join(["args["+str(ind)+"]"
#                             for ind in range(len(self.sympy_variables))])
#        extra_c_code="".join([r"""
# double integrand_wrapper(int n, double args[n])
# {
#   return integrand(""",args_str,""");
#
# }
# """])
#        extra_h_code="""
#
# double integrand_wrapper(int n, double args[n]);
#
# """
#
#        f = open(filename_prefix+".c",'a')
#        f.write(extra_c_code)
#        f.close()
#        f = open(filename_prefix+".h",'a')
#        f.write(extra_h_code)
#        f.close()
#        cmd = ("gcc -dynamiclib -O3 -I. "+filename_prefix+".c -o "
#               +filename_prefix+".dylib")
#        subprocess.call(cmd,shell=True)
#        self.ctypeslib = ctypes.CDLL(filename_prefix+'.dylib')
#        self.ctypesified = self.ctypeslib.integrand_wrapper
#        self.ctypesified.restype = ctypes.c_double
#        # Test the reliability of ctypesified function. This should be
#        # disabled eventually.
#        self.ctypesified.argtypes = (ctypes.c_int,
#                                     len(self.sympy_variables)*ctypes.c_double)
#
#        test = []
#        for indx in range(0):
#            randargs = [random.random() for item in self.sympy_variables]
#            temp = (self.lambdified(*randargs)-
#                    self.ctypesified(
#                    ctypes.c_int(len(self.sympy_variables)),
#                    (len(self.sympy_variables)*ctypes.c_double)(*randargs)))
#            if temp**2 >.00000001:
#                test.append(temp)
#        if test:
#            raise IntegrationError("Ctypesified and lambdified do not match!")
#        self.ctypesified.argtypes = ctypes.c_int, ctypes.c_double
#        return None
```

```
    def __call__(self, *args):
        if len(args) != len(self.sympy_variables):
            print 'args = ', args
            print 'sympy_vars = ', self.sympy_variables
            raise Error('invalid argument list given in call to Integrand!')
        import pdb; pdb.set_trace()
        out = self.lambdified(*args)
        out1 = self.ctypesified(len(args), tuple(args))
        print out - out1
        import pdb; pdb.set_trace()
#        print (out-out2)**2
#        exit()
        clear_cache()
        return out


class IntegrationError(Exception):
    pass


class OptionsDict(object):
    def __init__(self, points):
        self.points = points

    def __call__(self, *args):
        if self.points:
            out = {"points": self.points(*args)}
        else:
            out = {}
        return out


class DiscontinuityError(Exception):
    pass


class EmptyDiscontinuity(UserWarning):
    pass


class Discontinuity(object):
    def __init__(self, disc, ranges, args={}, opts={}):
        self._disc = disc.subs(args)
        self._ranges = ranges
        self._args = args
        self._opts = opts
```

```
    # Eventually set self.method from opts
    self._method = "lambdified"
    self.sym_sols = self._solve_points()
    if not self.sym_sols:
        raise EmptyDiscontinuity()
    self.call_args, self._lambda_list = self._lambdify()
    self.children = self._spawn()

def __call__(self, *args):
    'Return list of points of discontinuity for given arguments.\n\n'
    if self._method == "lambdified":
        return self._lambdified(*args)
    else:
        raise DiscontinuityError("Undefined call method!")

def __eq__(self, other):
    try:
        out = self._key() == other._key()
    except(AttributeError):
        out = False
    return out

def __ne__(self, other):
    return not self.__eq__(other)

def _key(self):
    return(type(self).__name__, self._disc, self._ranges,
           self._args, self._opts)

def __hash__(self):
    return hash(self._key())

def _solve_points(self):
    try:
        sols = sympy.solve(self._disc, self._ranges[0][0])
    except(KeyError):
        # No solutions.
        sols = []
    return sols

def _lambdify(self):
    lambda_list = []
    vars = [range_[0] for range_ in self._ranges[1:]]
    for sym_sol in self.sym_sols:
        lambda_list.append(lambdify(vars, sym_sol))
    self.__call__.__func__.__doc__ += (
        'Function signature is f(' + ','.join([str(var) for var in vars])
```

```
            + ') \n')
        clear_cache()
        return vars, lambda_list

    def _lambdified(self, *args):
        return [lambda_(*args) for lambda_ in self._lambda_list]

    def _spawn_local_extrema(self):
        sols = sympy.solve(
            sympy.diff(self._disc, self._ranges[0][0]), self._ranges[0][0])
        new_discs = [self._disc.subs({self._ranges[0][0]:sol}) for sol in sols]
        out = []
        for disc in new_discs:
            try:
                out.append(Discontinuity(disc, self._ranges[1:]))
            except(EmptyDiscontinuity):
                continue
        return out

    def _spawn_boundary_intersections(self):
        new_discs = [self._disc.subs({self._ranges[0][0]: lim})
                     for lim in self._ranges[0][1:]]
        out = []
        for disc in new_discs:
            try:
                out.append(Discontinuity(disc, self._ranges[1:]))
            except(EmptyDiscontinuity):
                continue
        return out

    def _spawn(self):
        if len(self._ranges) > 1:
            out = (self._spawn_local_extrema() +
                   self._spawn_boundary_intersections())
        else:
            out = []
        return out


class Discontinuities(object):
    def __init__(self, discs, ranges, args={}, opts={}):
        self.ranges = ranges
        self.discs = []
        for disc in discs:
            try:
                self.discs.append(Discontinuity(disc, self.ranges, args, opts))
            except(EmptyDiscontinuity):
```

```
                continue
        self.leveled_discs = self._level_discs()
        self.nquad_disc_functions = self.nquad_discs_f()

    def _level_discs(self):
        # Organize the discontinuities according to their level of
        # integration.
        this_level = list(self.discs)
        out = []
        while not empty(this_level):
            out.append(this_level)
            next_level = []
            for disc in this_level:
                next_level.extend(disc.children)
            this_level = next_level
        # Need to eliminate duplicates
        for level in out:
            new_level = list(level)
            for item in level:
                if new_level.count(item) > 1:
                    new_level.remove(item)
            level[:] = new_level
        return out

    class NQuadDiscFunction(object):
        def __init__(self, level):
            self.level = level

        def __call__(self, *args):
            out = []
            for disc in self.level:
                try:
                    out.extend(disc(*args))
                except(ValueError):
                    out.extend([])
            return out

    def nquad_discs_f(self):
        out = [self.NQuadDiscFunction(level) for level in self.leveled_discs]
        return out


def empty(seq): # Thanks StackOverflow!
    # See: http://stackoverflow.com/questions/1593564/
    # python-how-to-check-if-a-nested-list-is-essentially-empty
    # Accessed 6 Jun 2014
    try:
```

```
        return all(map(empty, seq))
    except TypeError:
        return False


if __name__ == "__main__":
    import random
    # Test handling of symbolic discontinuities
    x, y, z = [sympy.Symbol(var, real=True) for var in ['x', 'y', 'z']]
    ranges = [[x, -.25, 1.25], [y, -.25, 1.25], [z, -.25, 1.25]]
    sym_disc = x ** 2 + y ** 2 + z ** 2 - 1
    disc = Discontinuity(sym_disc, ranges)
    eqns = [- sympy.sqrt(1 - y ** 2 - z ** 2), sympy.sqrt(1 - y ** 2 - z ** 2)]
    for eqn in eqns:
        for sym_sol in disc.sym_sols:
            if sympy.Equivalent(eqn - sym_sol, 0):
                break
        else:
            raise DiscontinuityError(
                "Discontinuity test returned incorrect symbolic solutions!")
    yrand, zrand = [.5 * random.random() - .25 for ind in [0, 1]]
    lambda_sol = disc._lambdified(yrand, zrand)
    subs_sol = [
        sym_sol.subs({y: yrand, z: zrand}) for sym_sol in disc.sym_sols]
    err = [((lambda_sol[ind] - subs_sol[ind]) ** 2) ** .5
           for ind in range(len(subs_sol))]
    if max(err) > 1 * 10 ** - 13:
        raise DiscontinuityError(
            "Lambdified solution does not match symbolic solution!")
    test_children = [y ** 2 + z ** 2 - 1, y ** 2 + z ** 2 - 15. / 16]
    for test in test_children:
        for child in disc.children:
            if sympy.Equivalent(test, child._disc):
                break
        else:
            raise DiscontinuityError(
                "Direct children do not match!")
    test_discs = [[disc._disc for disc in level] for level in
                  Discontinuities([sym_disc], ranges).leveled_discs]
    sol_discs = [[x ** 2 + y ** 2 + z ** 2 - 1],
                 [y ** 2 + z ** 2 - 1, y ** 2 + z ** 2 - 15. / 16],
                 [z ** 2 - 1, z ** 2 - 15. / 16, z ** 2 - 7. / 8]]
    for inda in range(len(test_discs)):
        if not set(test_discs[ind]) == set(sol_discs[ind]):
            raise DiscontinuityError("Levelled discontinuities do not match!")
    test = Discontinuities([sym_disc], ranges)
    test2 = test.nquad_disc_functions
```

```
    args_list = [[.5 * random.random() - .25 for ind in range(inds)]
                 for inds in [2, 1, 0]]
    vars_list = [[y, z], [z], []]
    for ind, level in enumerate(test.leveled_discs):
        subs_points = [sym_sol.subs(dict(zip(vars_list[ind], args_list[ind])))
                       for disc in level for sym_sol in disc.sym_sols]
        subs_points.sort()
        lambda_points = test2[ind](*args_list[ind])
        lambda_points.sort()
        err = [((subs_points[ind] - lambda_points[ind]) ** 2) ** .5
               < 10 ** (- 13) for ind in range(len(subs_points))]
        if [item for item in err if not item]:
            raise DiscontinuityError(
                "Lambdified functions do not match symbolic functions!")

    # Test integration of functions
    test_discs = [x ** 2 + y ** 2 - 1]
    # This takes too long at present. Only integrate over x, y.
    test4 = IntegrableFunction(H(test_discs[0]),
                               ranges[:-1], test_discs)
    # Check against evaluation in Mathematica.
    if ((test4.integrate()[0] - 0.907360054182972) ** 2) ** .5 > 10 ** (- 7):
        raise IntegrationError(
            "Incorrect integration result for 2-D integration!")
    import pdb; pdb.set_trace()
```

B.2 base_equation.py

base_equation.py provides the basic functionality required to compute manufactured source terms for conservation and balance laws. This is principally done through the use of the abstract SympyEquation class, as described in section 3.4.

```
"""
Base equation class, intended to be subclassed for specific systems of
equations, such as the linear heat equation, the Euler equations, and so on.
"""
import sympy
from sympy.utilities.lambdify import lambdify
import numpy as np
from integration import list_integral
from functools import partial
from multiprocessing import Pool
```

```
class SympyEquation(object):
    """
Base equation class providing for computation of manufactured source terms.

SympyEquation is an abstract superclass that provides the general routines
used to compute manufactured source terms given symbolic flux and source terms.

Parameters
----------
sol : dict
    Dictionary containing the entries:
        'vars' - list of sympy Symbols corresponding to integration variables.
        'sol' - sympy Matrix of integrand expressions.
        'discontinuities' - list of sympy expressions corresponding to known
            solution discontinuities e.g. (x/t-0.75). The discontinuity is
            assumed to be at disc == 0, where disc is the sympy expression.
        'eqn_kwargs' - dict containing any keyword arguments required by the
            specific equation subclass.
    'vars', 'sol', and 'discontinuities' are copied as object attributes.
    'eqn_kwargs' are used as arguments to the setup routine provided by the
    specific equation subclass.

Attributes
----------
vars : iterable
    List of Sympy Symbols corresponding to integration variables.
sol : sympy.Matrix
    Matrix containing the symbolic expression of the solution to be used. May
    be a manufactured solution, but may also be an exact solution.
discontinuities : iterable
    List of Sympy expressions corresponding to known solution discontinuities
    e.g. (x/t-0.75). The discontinuity is assumed to be at disc == 0, where
    disc is the sympy expression.
fluxes : list
    List of len('vars') of sympy Matrix objects. Each Matrix is composed of
    sympy expressions derived from the given solution 'sol' and the flux
    functions defined by the specific equation subclass and 'eqn_kwargs'.
source : sympy.Matrix
    Sympy Matrix composed of sympy expressions derived from the given solution
    'sol' and the source functions defined by the specific equation subclass
    and 'eqn_kwargs'.
source_diff_symbolic : sympy.Matrix
    symbolic differential source terms, computed by 'balance_diff'


Methods
```

```
-------
balance_diff()
    Return symbolic form of strong (differential) manufactured source terms.
balance_lambda_init()
    Return lambdified form of balance_diff.
balance_integrate(ranges,discs=())
    Return weak-form manufactured source terms, numerically integrated over
    given ranges, with optional discontinuities.
setup(**sol['eqn_kwargs'])
    Subclass-defined initialization routine which computes symbolic
    expressions of flux and source terms given the solution provided in 'sol'.
    Must set:
        self.fluxes : list of sympy Matrix objects containing sympy
            expressions.
        self.source : sympy Matrix object containing sympy expressions.
"""
    def __init__(self,sol):
        self.vars_ = sol['vars']
        self.sol = sol['sol']
        self.discontinuities = sol['discontinuities']
        self.setup(**sol['eqn_kwargs'])
    def __call__(self):
        return self.fluxes+[self.source]
    def setup(self):
        pass
    def dot_product(self,a,b):
        return sum(map(lambda x,y:x*y,a,b))
    def vector_diff(self,expr,var_in):
        vars_ = list(expr)
        for n in range(len(vars_)):
            vars_[n] = var_in
#        return sympy.Matrix(map(lambda u,x:sympy.diff(u,x),expr,vars_))
        return sympy.Matrix(map(lambda u,x=var_in:sympy.diff(u,x),expr,vars_))
    def balance_diff(self):
        """
Return symbolic form of strong (differential) manufactured source terms.

Compute strong manufactured source terms based on a differentiable solution,
flux functions, and source functions. The flux functions must be
differentiable, because symbolic derivatives are taken.

Returns
-------
sympy.Matrix
    Sympy Matrix object containing symbolic manufactured source terms.
"""
        terms = [self.vector_diff(flux,var)
```

```
                for (flux, var) in zip(self.fluxes,self.vars_)]+[self.source]
        self.source_diff_symbolic = reduce(sympy.Matrix.__add__,terms)
        return self.source_diff_symbolic
    def balance_lambda_init(self):
        """
Lambdify strong symbolic manufactured source terms.

Define python lambda function f(*vars) to evaluate manufactured source terms at
a given point.

Returns
-------
callable
    Python lambda function that evaluates symbolic manufactured source terms at
    a given point (set of 'vars').
"""
        self.balance_diff()
        self.balance_diff_lambda=lambdify(self.vars_,self.source_diff_symbolic)
        return self.balance_diff_lambda
    def flux_integrate(self,flux,area_ranges,point_range,discs=()):
        out = (np.array(list_integral(
                    flux,sympy_ranges=area_ranges,sympy_discontinuities=discs,
                    args={point_range[0]:point_range[2]}))-
               np.array(list_integral(
                    flux,sympy_ranges=area_ranges,sympy_discontinuities=discs,
                    args={point_range[0]:point_range[1]})))
        return out
    def source_integrate(self,source,ranges,discs):
        out = np.array(list_integral(
                source,sympy_ranges=ranges,sympy_discontinuities=discs))
        return out
    def balance_integrate(self,ranges,discs=()):
        """
Evaluate weak (integral) manufactured source terms over a given volume.

Call flux_integrate and source_integrate with the appropriate ranges to
evaluate the weak (integral) manufactured source terms based on the potentially
discontinuous 'sol' and the flux and source functions. Source terms are
integrated over the computational volume described by 'ranges'.

Parameters
----------
ranges : iterable
    List of ranges defining the computational volume over which the balance
    integral is to be evaluated e.g. ((t,0,1),(x,-0.5,0.5)) where t,x are sympy
    Symbols.
discs : iterable
```

```
    List of symbolic expressions that describe the location of any known
    discontinuities in the integrand. The discontinuity is assumed to be
    located where the expressions are equal to zero.

Returns
-------
numpy.array
    Array of floats containing the integrated source terms over 'ranges'.

Notes
-----
This is a promising opportunity for implementing multiprocessing, as the flux-
and source-integrals take some time to compute and are independent of one
another.
"""
        # Since we're multiprocessing at the list level, we can't
        # also multiprocess here. Pool workers can't spawn their
        # own pools.
        out = np.zeros(len(self.sol))
        wrapper = partial(flux_integrate_wrapper,obj=self,ranges=ranges,
                          discs=self.discontinuities)
#        pool = Pool()
#        out_list = []
#        out_list = pool.map(wrapper,range(len(ranges)))
#        pool.close()
#        pool.join()
        out_list = map(wrapper,range(len(ranges)))
#        print "done with flux integrals"
        out = out_list + [self.source_integrate(self.source,ranges,discs)]
#        print "done with source integral"
        out = sum(out)
        return out
def flux_integrate_wrapper(ind,obj,ranges,discs):
    """
Wrapper for flux_integrate for use in multiprocessing.
"""
    return obj.flux_integrate(
        obj.fluxes[ind],
        area_ranges=[item for item in ranges if item is not ranges[ind]],
        point_range=ranges[ind],discs=discs)

if __name__=="__main__":
    pass
```

B.3 Equation-specific Modules

The general-purpose machinery in `base_equation.py` is incomplete, and it is necessary to define subclasses based on `SympyEquation` in order to specify the specific flux and source terms that define a specific equation set. These subclasses are distributed in equation specific modules, which are also convenient locations to include useful manufactured solutions. At present, `BACL-MMS` contains two such modules, for the linear heat equation and for the Euler equations in the unified coordinate system, as described in chapter 4.

B.3.1 `heat_equation.py`

```
"""
The linear heat equation, including several solutions
"""
from itertools import product
import random

import sympy
from sympy.utilities.lambdify import lambdify
from sympy import sin, cos

import numpy
import numpy.random

from base_equation import SympyEquation
from recursive_derivative import recursive_derivative

Zero = sympy.singleton.S.Zero
t = sympy.Symbol('t')
x = sympy.Symbol('x')
y = sympy.Symbol('y')
z = sympy.Symbol('z')
xi= sympy.Symbol('xi')

class HeatEquation(SympyEquation):
    def setup(self,**kwargs):
        self.rho = kwargs['rho']
        self.cp = kwargs['cp']
        self.k = kwargs['k']
        self.fluxes = [sympy.Matrix([self.rho*self.cp*self.sol[0]]),
                       sympy.Matrix(
```

```
                [-self.k*sympy.diff(self.sol[0],self.vars_[1])]),
                       sympy.Matrix(
                [-self.k*sympy.diff(self.sol[0],self.vars_[2])]),
                       sympy.Matrix(
                [-self.k*sympy.diff(self.sol[0],self.vars_[3])])]
        self.source = sympy.Matrix([Zero])

def MASA_solution(Ax,At,By,Bt,Cz,Ct,Dt,rho,cp,k):
    """
Manufactured solution as given in MASA documentation.
"""
    return {'vars':[t,x,y,z],'eqn_kwargs':{'rho':rho,'cp':cp,'k':k},
                'sol':[sympy.cos(Ax*x+At*t)*sympy.cos(By*y+Bt*t)*
                       sympy.cos(Cz*z+Ct*t)*sympy.cos(Dt*t)],
                'discontinuities':[]}

def MASA_source(Ax,At,By,Bt,Cz,Ct,Dt,rho,cp,k):
    """
Analytic manufactured source term as given in MASA documentation.
"""
    out = (
        -(sin(Ax*x+At*t)*cos(By*y+Bt*t)*cos(Cz*z+Ct*t)*cos(Dt*t)*At+
          cos(Ax*x+At*t)*sin(By*y+Bt*t)*cos(Cz*z+Ct*t)*cos(Dt*t)*Bt+
          cos(Ax*x+At*t)*cos(By*y+Bt*t)*sin(Cz*z+Ct*t)*cos(Dt*t)*Ct+
          cos(Ax*x+At*t)*cos(By*y+Bt*t)*cos(Cz*z+Ct*t)*sin(Dt*t)*Dt)*rho*cp+
         (Ax**2+By**2+Cz**2)*cos(Ax*x+At*t)*cos(By*y+Bt*t)*cos(Cz*z+Ct*t)*
         cos(Dt*t)*k
         )
    return out

def MASA_source_lambda(**kwargs):
    """
Functional form of 'MASA_source'.

Parameters
----------
(all parameters are keyword arguments)
Ax, At, By, Bt, Cz, Ct, Dt : float
    Parameters used in the computation of 'MASA_source'
rho : float
    mass density of medium.
cp : float
    specific heat at constant pressure of medium.
k : float
    heat conductivity of medium.

Returns
```

```
-------
callable
    Returns Python lambda function f(t,x,y,z) to evaluate 'MASA_source' at the
    given point in space and time.
"""
    return lambdify((t,x,y,z),MASA_source(**kwargs))

def heat_exact_sol(**kwargs):
    """
Collection of exact solutions to the heat equation.

These exact solutions are inherently one-dimensional, and are rotated by two
angles in order to obtain solutions that three-dimensional (not aligned with
any three-dimensional coordinate direction).

Parameters
----------
(all parameters are keyword arguments)
n : int in range(8)
    Identifies specific desired exact solution.
theta : float
    Polar angle of solution rotation, measured from x-axis, in radians.
phi : float
    Aximuthal angle of solution rotation, measured from y-axis, in radians.
A, B, C, mu : float
    Exact solution parameters. Can be arbitrary constants.
a : float
    Thermal diffusivity, given by k/(cp*rho)

Returns
-------
dict, containing the fields:
    sol : sympy.Matrix
        Matrix containing symbolic representation of one exact solution.
    discontinuities : list
        Empty list; the linear heat equation does not admit discontinuities.
    vars : list
        List of sympy Symbols: t,x,y,z.
    eqn_kwargs : dict
        kwargs dict that sets 'rho', 'cp', 'k', such that the thermal
        diffusivity 'a' is obtained.
"""
    n, theta, phi, A, B, C, mu, a  = (
        kwargs['n'], kwargs['theta'], kwargs['phi'], kwargs['A'], kwargs['B'],
        kwargs['C'], kwargs['mu'], kwargs['a'] )
#    theta, phi = 0.*sympy.pi, 0.*sympy.pi
    space = (sympy.cos(theta)*x
```

```
              +sympy.sin(theta)*sympy.cos(phi)*y
              +sympy.sin(theta)*sympy.sin(phi)*z
              )
    # These solutions taken from:
    # http://eqworld.ipmnet.ru/en/solutions/lpde/lpde101.pdf
#    A, B, C, mu = 1,1,1,1
#    a = 1
    sols = [A*space+B,
            A*(space**2+2*a*t)+B,
            A*(space**3+6*a*t*space)+B,
            A*(space**4+12*a*t*space**2+12*a**2*t**2)+B,
            A*sympy.exp(a*mu**2*t+mu*space)+B,
            A*sympy.exp(a*mu**2*t-mu*space)+B,
            A*sympy.exp(-a*mu**2*t)*sympy.cos(mu*space+C)+B,
            A*sympy.exp(-mu*space)*sympy.cos(mu*space-2*a*mu**2*t+C)+B]
    out = {'sol':sympy.Matrix([sols[n]]),'discontinuities':[],
           'vars':[t,x,y,z],'eqn_kwargs':{'rho':1.,'cp':1.,'k':a}}
    return out

#def heat_exact_tests(ranges,nx):
#    """
#
#"""
#    angle_list = [range_*(.5*sympy.pi/(nx-1)) for range_ in range(nx)]
#    spaces = [sympy.cos(theta)*x
#              +sympy.sin(theta)*sympy.cos(phi)*y
#              +sympy.sin(theta)*sympy.sin(phi)*z for angle in angle_list]
#    return [[heat_exact_sol(xi,n),angle_list,
#             heat_exact_array(spaces,n,ranges)]
#            for n in range(8)]


def test_exact(ntests):
    """
Test the accuracy of numerical integration of the linear heat equation

Compute the weak manufactured source terms from exact solutions for the linear
heat equation, given randomized combinations of the various solution
parameters. Write the combinations of parameters and the resulting source term
to the file 'random_heat_exact.dat'.

Parameters
----------
ntests : int
    Number of random trials to compute.

Returns
```

```
-------
random_heat_exact.dat : ASCII file
    Contains solution parameters and corresponding source term values.
"""
    f = open('random_heat_exact.dat','w')
    f.write('%problem #, theta(deg), phi(deg), A, B, C, mu, a, source')
    ranges = ((t,0.,1.),(x,-.5,.5),(y,-.5,.5),(z,-.5,.5))
    random.seed(100)
    S_prime_list = []
    for indn in range(ntests):
        n_choices = [0,1,2,3,4,5,6,7]
        theta_min, theta_max = 0, numpy.pi*.5
        phi_min, phi_max = 0, numpy.pi*.5
        A_min, A_max = 0.001, 1000
        B_min, B_max = 0.001, 1000
        C_min, C_max = 0, 1
        mu_min, mu_max = 0, 3
        a_min, a_max = 0.001, 10
        n,theta,phi,A,B,C,mu,a = [random.choice(n_choices),
                                  random.random()*(theta_max-theta_min),
                                  random.random()*(phi_max-phi_min),
                                  10**(random.random()*(numpy.log10(A_max)
                                                        -numpy.log10(A_min))
                                       +numpy.log10(A_min)),
                                  10**(random.random()*(numpy.log10(B_max)
                                                        -numpy.log10(B_min))
                                       +numpy.log10(A_min)),
                                  random.random()*(C_max-C_min),
                                  random.random()*(mu_max-mu_min),
                                  10**(random.random()*(numpy.log10(a_max)
                                                        -numpy.log10(a_min))
                                       +numpy.log10(a_min))
                                  ]
        sol = heat_exact_sol(n=n,theta=theta,phi=phi,A=A,B=B,C=C,mu=mu,a=a)
        try:
            S_prime = HeatEquation(sol).balance_integrate(ranges)
            S_prime_list.append(S_prime)
            f.write('\n'+', '.join([str(item) for item in
                                    (n,theta/numpy.pi*180,phi/numpy.pi*180,
                                     A,B,C,mu,a,S_prime[0])]))
        except(OverflowError):
            print "Overflow Error!"
            print ('n = ',n,'theta = ',theta,'phi = ',phi,
                   'A = ',A,'B = ',B,'C = ',C,'mu = ',mu,'a = ',a)
    f.close()
    return S_prime_list
```

```
class HeatEquationError(Exception):
    pass

if __name__=="__main__":
    S_prime_list = test_exact(100)
    S_prime_log = [numpy.log10(numpy.abs(S)) for S in S_prime_list]
    high_error_ratio = (len([S for S in S_prime_log if S > -10])
                        /float(len(S_prime_log)))
    if high_error_ratio > 0.01:
        raise HeatEquationError('Unexpectedly high error in heat equation!')
    # You can load random_heat_exact.dat in Matlab to create a parallel
    # coordinates plot of the error.
    # If you run enough tests, you do find some odd explosions of error for
    # some cases. These are few (0.33% for 1000 cases with given seed).
    print "All Heat Equation tests passed!"
```

B.3.2 Euler_UCS.py

```
"""
Tools for computing manufactured solutions for Euler equations expressed in
Hui's Unified Coordinate System.
"""
import random
import numpy
import sympy
from sympy.utilities.lambdify import lambdify
H = sympy.special.delta_functions.Heaviside
from sympy import sin, cos
from functools import partial

from base_equation import SympyEquation

Zero = sympy.singleton.S.Zero
t = sympy.Symbol('t')
xi = sympy.Symbol('xi')
eta = sympy.Symbol('eta')
zeta = sympy.Symbol('zeta')
gamma = sympy.Rational(7,5)

#def H(S):
#    if S>0:
#        out = 1.
#    else:
#        if S<0:
#            out = 0.
```

```
#        else:
#            out = .5
#    return out

class Euler_UCS(SympyEquation):
    def setup(self,**kwargs):
        (self.pressure,self.density,                #
         self.vels_x,self.vels_y,self.vels_z,       # u, v, w
         self.dx_dxi,self.dy_dxi,self.dz_dxi,       # A, B, C
         self.dx_deta,self.dy_deta,self.dz_deta,    # L, M, N
         self.dx_dzeta,self.dy_dzeta,self.dz_dzeta, # P, Q, R
         self.dx_dt,self.dy_dt,self.dz_dt,          # U, V, W
         self.x,self.y,self.z) = self.sol
        self.gamma = gamma
        self.jacobian = (
            self.dx_dxi*
            (self.dy_deta*self.dz_dzeta-self.dz_deta*self.dy_dzeta) +
            self.dy_dxi*
            (self.dz_deta*self.dx_dzeta-self.dx_deta*self.dz_dzeta) +
            self.dz_dxi*
            (self.dx_deta*self.dy_dzeta-self.dy_deta*self.dx_dzeta) )
        self.dxi_dx = (self.dy_deta*self.dz_dzeta-self.dz_deta*self.dy_dzeta
                       )/self.jacobian
        self.dxi_dy = (self.dz_deta*self.dx_dzeta-self.dx_deta*self.dz_dzeta
                       )/self.jacobian
        self.dxi_dz = (self.dx_deta*self.dy_dzeta-self.dy_deta*self.dx_dzeta
                       )/self.jacobian
        self.deta_dx = (self.dz_dxi*self.dy_dzeta-self.dy_dxi*self.dz_dzeta
                        )/self.jacobian
        self.deta_dy = (self.dx_dxi*self.dz_dzeta-self.dz_dxi*self.dx_dzeta
                        )/self.jacobian
        self.deta_dz = (self.dy_dxi*self.dx_dzeta-self.dx_dxi*self.dy_dzeta
                        )/self.jacobian
        self.dzeta_dx = (self.dy_dxi*self.dz_deta-self.dz_dxi*self.dy_deta
                         )/self.jacobian
        self.dzeta_dy = (self.dz_dxi*self.dx_deta-self.dx_dxi*self.dz_deta
                         )/self.jacobian
        self.dzeta_dz = (self.dx_dxi*self.dy_deta-self.dy_dxi*self.dx_deta
                         )/self.jacobian
        self.Dxi_Dt_vec = (
            self.dot_product((self.vels_x-self.dx_dt,self.vels_y-self.dy_dt,
                              self.vels_z-self.dz_dt),
                             (self.dxi_dx,self.dxi_dy,self.dxi_dz)),
            self.dot_product((self.vels_x-self.dx_dt,self.vels_y-self.dy_dt,
                              self.vels_z-self.dz_dt),
                             (self.deta_dx,self.deta_dy,self.deta_dz)),
            self.dot_product((self.vels_x-self.dx_dt,self.vels_y-self.dy_dt,
```

```
                            self.vels_z-self.dz_dt),
                           (self.dzeta_dx,self.dzeta_dy,self.dzeta_dz)))
        self.vels_xi_vec = (
            self.dot_product((self.vels_x,self.vels_y,self.vels_z),
                             (self.dxi_dx,self.dxi_dy,self.dxi_dz)),
            self.dot_product((self.vels_x,self.vels_y,self.vels_z),
                             (self.deta_dx,self.deta_dy,self.deta_dz)),
            self.dot_product((self.vels_x,self.vels_y,self.vels_z),
                             (self.dzeta_dx,self.dzeta_dy,self.dzeta_dz)))
        self.energy = (sympy.Rational(1,2)*
                       (self.vels_x**2+self.vels_y**2+self.vels_z**2) +
                       self.pressure/((self.gamma-1)*self.density))
        self.grad_xi_vec = ((self.dxi_dx,self.dxi_dy,self.dxi_dz),
                            (self.deta_dx,self.deta_dy,self.deta_dz),
                            (self.dzeta_dx,self.dzeta_dy,self.dzeta_dz))
        self.fluxes = [self.cons()]+[self.flux(n) for n in range(3)]
        self.source = self.source_func()

    def cons(self):
        return sympy.Matrix([
            self.density*self.jacobian,
            self.density*self.jacobian*self.vels_x,
            self.density*self.jacobian*self.vels_y,
            self.density*self.jacobian*self.vels_z,
            self.density*self.jacobian*self.energy,
            self.dx_dxi,self.dy_dxi,self.dz_dxi,
            self.dx_deta,self.dy_deta,self.dz_deta,
            self.dx_dzeta,self.dy_dzeta,self.dz_dzeta,
            self.dx_dt,self.dy_dt,self.dz_dt,
            self.x,self.y,self.z])
    def flux(self,n):
        out =  [self.jacobian*(self.density*self.Dxi_Dt_vec[n]),
                self.jacobian*(self.density*self.Dxi_Dt_vec[n]*self.vels_x+
                               self.grad_xi_vec[n][0]*self.pressure),
                self.jacobian*(self.density*self.Dxi_Dt_vec[n]*self.vels_y+
                               self.grad_xi_vec[n][1]*self.pressure),
                self.jacobian*(self.density*self.Dxi_Dt_vec[n]*self.vels_z+
                               self.grad_xi_vec[n][2]*self.pressure),
                self.jacobian*(self.density*self.Dxi_Dt_vec[n]*self.energy+
                          self.vels_xi_vec[n]*self.pressure),
                0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
        out[5:8] = [item*sympy.Integer(-1*(1-n)*(1-n/2)) for item in
                    (self.dx_dt,self.dy_dt,self.dz_dt)]
        out[8:11] = [item*sympy.Integer(-1*(n)*(2-n)) for item in
                     (self.dx_dt,self.dy_dt,self.dz_dt)]
        out[11:14] = [item*sympy.Integer(-1*(n-1)*(n/2)) for item in
                      (self.dx_dt,self.dy_dt,self.dz_dt)]
```

```
        return sympy.Matrix(out)
    def source_func(self):
        return sympy.Matrix([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                self.dx_dt,self.dy_dt,self.dz_dt])

def exact_Euler_sols(case,opts={}):
    """
Various exact solutions for the Euler equations in Cartesian coordinates.

Returns solution objects for various exact solutions of the unsteady, Euler
equations. Includes both normal shocks and riemann problems.

Parameters
----------
case : string
    Key to cases dict. Can be 'normal' or 'riemann_problem'.
opts : dict, optional
    Additional solution options. Currently only chooses among riemann problems,
    and solution rotations. The riemann problem is set by the 'set_riemann'
    key, and the rotation angles are set by 'phi' and 'theta'. Rotation angles
    are used only  by 'riemann_problem'.

Returns
-------
dict

"""
    cases = {'simple':simple_case,
#             'two_shock':two_shock_case,
             'normal':normal_case,
             'riemann_problem':partial(riemann_problem_case,
                                       opts['set_riemann'],opts['theta'],
                                       opts['phi'])}
    out = {'vars':[t,xi,eta,zeta],'eqn_kwargs':{}}
    out.update(cases[case]())
    return out

def simple_case():
    return {'sol':sympy.Matrix([1,1,1,0,0,1,0,0,0,1,0,0,0,1,0,0,0,0,0,0]),
            'discontinuities':[]}

def normal_case():
    theta = sympy.pi*.25
    S = (sympy.cos(theta)*xi+sympy.sin(theta)*eta)/t
    shock_speed = .78931
    p1 = 460.894
    d1 = 5.99924
```

```
    umag1 = 19.5975
    M1rel = (umag1-shock_speed)/(gamma*p1/d1)**(.5)
    p2 = ((2.*gamma*M1rel**2-(gamma-1))/(gamma+1))*p1
    d2 = ((gamma+1.)*M1rel**2/((gamma-1.)*M1rel**2+2.))*d1
    umag2 = (1-d1/d2)*shock_speed+umag1*d1/d2
    print "p2,d2,u2 = ",p2,d2,umag2
    u1 = umag1*sympy.cos(theta)
    v1 = umag1*sympy.sin(theta)
    u2 = umag2*sympy.cos(theta)
    v2 = umag2*sympy.sin(theta)
    speeds = [shock_speed]
    states = [sympy.Matrix([p1,d1,u1,v1]),sympy.Matrix([p2,d2,u2,v2])]
    base_state = [0,1,0,0,0,1,0,0,0,1,0,0,0,0,0,0]
    out = {'sol':sympy.Matrix(list(states[0]+
                                   H(S-speeds[0])*(states[1]-states[0]))+
                               base_state),
           'discontinuities':[S-speed for speed in speeds]}
    return out

#def two_shock_case():
#    speeds = [0.78959391926443701,8.6897744116323814,12.250778123084338]
#    phi, theta = 0.,sympy.pi*.1#sympy.pi*.5, sympy.pi*.25
#    S = (sympy.cos(theta)*xi+
#         sympy.sin(theta)*sympy.cos(phi)*eta+
#         sympy.sin(theta)*sympy.sin(phi)*zeta)/t
#    yz_rotation_matrix = sympy.Matrix(
#        [[sympy.cos(theta),0,0],
#         [sympy.sin(theta)*sympy.cos(phi),0,0],
#         [sympy.sin(theta)*sympy.sin(phi),0,0]])
#
##    S = yz_rotation_matrix.dot(sympy.Matrix([xi,eta,zeta]))[0]/t
#    states = [sympy.Matrix([460.89400000000001,5.9992400000000000004,
#                            19.597500000000000,0.,0.]),
#              sympy.Matrix([1691.6469553991260,14.282349951978405,
#                            8.6897744116323814,0.,0.]),
#              sympy.Matrix([1691.6469553991260,31.042601641619882,
#                            8.6897744116323814,0.,0.]),
#              sympy.Matrix([46.09499999999999,5.99242000000000001,
#                            -6.19632999999997,0.,0.])]
#    for state in states:
#        vel = sympy.Matrix([state[2],state[3],state[4]])
#        new_vel = yz_rotation_matrix.dot(vel)
#        state[2],state[3],state[4] = new_vel
#    base_state = [1.,0.,0.,0.,1.,0.,0.,0.,1.,0.,0.,0.,0.,0.,0.]
#    out = {'sol':sympy.Matrix(
#            list(
##                (1-H(S-speeds[0]))*states[0]+
```

```
##                H(S-speeds[0])*states[1])+base_state),
#                (1-H(S-speeds[0]))*states[0]+
#                H(S-speeds[0])*(1-H(S-speeds[1]))*states[1]+
#                H(S-speeds[1])*(1-H(S-speeds[2]))*states[2]+
#                H(S-speeds[2])*states[3])+base_state),
#           'discontinuities':[S-speed for speed in speeds]}
#    return out

def riemann_problem_case(n,theta,phi):
#    n = 4
#    theta, phi = sympy.pi*.5, 0.
    tests = riemann_problem_init()
    states,speeds = ([sympy.Matrix(state) for state in tests[n]['states']],
                     tests[n]['speeds'])
    S = (sympy.cos(theta)*xi+
         sympy.sin(theta)*sympy.cos(phi)*eta+
         sympy.sin(theta)*sympy.sin(phi)*zeta)/t
    fan_state_L = inside_fan_state(states[0],speeds[1],speeds[0],S)
    fan_state_R = inside_fan_state(states[3],speeds[3],speeds[4],S)
    states.insert(1,fan_state_L)
    states.insert(-1, fan_state_R)
    yz_rotation_matrix = sympy.Matrix(
        [[sympy.cos(theta),0,0],
         [sympy.sin(theta)*sympy.cos(phi),0,0],
         [sympy.sin(theta)*sympy.sin(phi),0,0]])
    for state in states:
        vel = sympy.Matrix([state[2],state[3],state[4]])
        new_vel = yz_rotation_matrix.dot(vel)
        state[2],state[3],state[4] = new_vel
    base_state = [1.,0.,0.,0.,1.,0.,0.,0.,1.,0.,0.,0.,0.,0.,0.]

    prims = map(list,zip(*states))
    sol_list = []
    for i, prim in enumerate(prims):
        piecewise_args = []
        piecewise_args.append((prim[0],S<speeds[0]))
        if speeds[1] != speeds[0]:
            piecewise_args.append((prim[1],S<speeds[1]))
        piecewise_args.append((prim[2],S<speeds[2]))
        piecewise_args.append((prim[3],S<speeds[3]))
        if speeds[4] != speeds[3]:
            piecewise_args.append((prim[4],S<speeds[4]))
        piecewise_args.append((prim[5],True))
        sol_list.append(sympy.Piecewise(*piecewise_args))
    solution_state = sol_list+base_state
    import pdb;pdb.set_trace()
    out = {'sol':sympy.Matrix(solution_state),
```

```
            'discontinuities':[S-speed for speed in speeds]}
#    import matplotlib.pyplot as plt
#    from mpl_toolkits.mplot3d import Axes3D
#    fig = plt.figure()
#    ax = fig.add_subplot(111,projection='3d')
#    Axes3D.plot_surface(X,Y,Z)
    return out

def inside_fan_state(upwind,head,tail,S):
    if head > tail:
        plusminus = -1
    else:
        if head < tail:
            plusminus = 1
        else:
            return upwind

    p = upwind[0]*(2/(gamma+1)-plusminus*(gamma-1)/(
            (gamma+1)*sympy.sqrt(gamma*upwind[0]/upwind[1]))*(
            upwind[2]-S))**((2*gamma)/(gamma-1))
    d = upwind[1]*(2./(gamma+1)-plusminus*(gamma-1.)/(
            (gamma+1)*sympy.sqrt(gamma*upwind[0]/upwind[1]))*(upwind[2]-S)
            )**(2./(gamma-1.))
    u = 2/(gamma+1)*((gamma-1)*.5*upwind[2]+S
                     -plusminus*sympy.sqrt(gamma*upwind[0]/upwind[1]))
    return sympy.Matrix([p,d,u,upwind[3],upwind[4]])

def riemann_problem_init():
    tests = []
    tests.append({})
    tests[0]['states'] = [
        [1., 1., 0., 0., 0.],
        [0.30313017805064679, 0.42631942817849516, 0.92745262004894879, 0., 0.],
        [0.30313017805064679, 0.26557371170530708, 0.92745262004894879, 0., 0.],
        [.1, .125, 0., 0., 0.]]
    tests[0]['speeds'] = [-1.1832159566199232, -7.0272812561184450E-002,
                          0.92745262004894879, 1.7521557320301779,
                          1.7521557320301779]
    tests.append({})
    tests[1]['states'] = [
        [.4, 1., -2., 0., 0.],
        [1.8938734200547610E-003, 2.1852118206812810E-002, 0., 0., 0.],
        [1.8938734200547610E-003, 2.1852118206812810E-002, 0., 0., 0.],
        [.4, 1., 2., 0., 0.]]
    tests[1]['speeds'] = [-2.7483314773547880, -0.34833147735478825,
                          0., 0.34833147735478825, 2.7483314773547880]
    tests.append({})
```

```
    tests[2] = {}
    tests[2]['states'] = [
        [1000., 1., 0., 0., 0.],
        [460.89378749138365, 0.57506229847655554, 19.597451388723059, 0., 0.],
        [460.89378749138365, 5.9992407047962342, 19.597451388723059, 0., 0.],
        [.01, 1., 0., 0., 0.]]
    tests[2]['speeds'] = [-37.416573867739416, -13.899632201271743,
                          19.597451388723059, 23.517536966903236,
                          23.517536966903236]
    tests.append({})
    tests[3]['states'] = [
        [.01, 1., 0., 0., 0.],
        [46.095044248867971, 5.9924168635152260, -6.1963282497870367, 0., 0.],
        [46.095044248867971, 0.57511278978241231, -6.1963282497870367, 0., 0.],
        [100., 1., 0., 0., 0.]]
    tests[3]['speeds'] = [-7.4374762586943133, -7.4374762586943133,
                          -6.1963282497870367, 4.3965656664547872,
                          11.832159566199232]
    tests.append({})
    tests[4]['states'] = [
        [460.894, 5.99924, 19.5975, 0., 0.],
        [1691.6469553991260, 14.282349951978405, 8.6897744116323814, 0., 0.],
        [1691.6469553991260, 31.042601641619882, 8.6897744116323814, 0., 0.],
        [46.0950, 5.99242, -6.19633, 0., 0.]]
    tests[4]['speeds'] = [0.78959391926443701, 0.78959391926443701,
                          8.6897744116323814, 12.250778123084338,
                          12.250778123084338]
    return tests

def MASA_with_pinned_bounds(ranges,nxes=(100,1,1),
                            dxis=(1.,1.,1.),x_origin=(0.,0.,0.)):
    ((x0,xn),(y0,yn),(z0,zn)) = ranges
    out = MASA_solution_full(ranges,nxes,dxis,x0=x_origin)
    out_vars = list(out['vars'])
    out_vars.remove(t)
    valmax = 2.
    pinning_eqn = (4.*valmax/(xn-x0)**2*
                   (-out_vars[0]**2+(xn+x0)*out_vars[0]-x0*xn))
    out['sol'] = sympy.Matrix(
        [sol*pinning_eqn + .01 for sol in out['sol'][0:5]]+out['sol'][5:])
    return out

def MASA_solution_full(ranges,nxes=(100,1,1),dxis=(1.,1.,1.),x0=(0.,0.,0.),
                       disc=False):
    """
Manufactured solution as given in MASA documentation for Euler equations.
"""
```

```
    kwargs={'x0':1.1, # = -(xt,xx,xy,xz)+pinned_value
            'xx':.5,'ax':2.,'fx':sympy.cos,'Lx':100.,
            'xy':0,'ay':2.,'fy':sympy.cos,'Ly':100.,
            'xz':0,'az':2.,'fz':sympy.cos,'Lz':100.,
            'xt':0,'at':2.,'ft':sympy.cos,'Lt':100.,
            'shock_strength':0.,'shock_position':sympy.Float(50.)}
    if disc:
        kwargs['shock_strength']=1.
    dxes = [1,1,1]
    for ind in range(3):
        try:
            dxes[ind] = float(ranges[ind][1]-ranges[ind][0])/(nxes[ind]-1)
        except(ZeroDivisionError):
            dxes[ind] = float(ranges[ind][1]-ranges[ind][0])/nxes[ind]
#    dxes = [float((range_[1]-range_[0]))/(nx-1)
#            for (range_,nx) in zip(ranges,nxes)]
    prims = [MASA_full_var(**kwargs) for var in range(5)]
    grid_vels = [0.*.25*vel for vel in prims[2:]]
    dx_dxi = dx_dxi_f(dx_dlambda=grid_vels,
                      dx_dxi_0=[dxes[0]/dxis[0],0.,0.,
                                0.,dxes[1]/dxis[1],0.,
                                0.,0.,dxes[2]/dxis[2]],t0=0.)
    xes = x_f(dx_dxi,grid_vels,x0,dxis)
    return {'vars':[t,xi,eta,zeta],
            'sol':sympy.Matrix(prims+dx_dxi+grid_vels+xes),
            'discontinuities':[kwargs['shock_position']],'eqn_kwargs':{}}

def dx_dxi_f(dx_dlambda,dx_dxi_0,t0):
    change = [0 for ind in range(9)]
    out = list(change)
    for inda in range(3):
        for indb in range(3):
            change[inda*3+indb] = sympy.integrate(sympy.diff(
                    dx_dlambda[indb],[xi,eta,zeta][inda]),t)
            out[inda*3+indb] = ( change[inda*3+indb] + dx_dxi_0[inda*3+indb] -
                                 change[inda*3+indb].subs({t:t0}) )
    return out

def x_f(dx_dxi,dx_dlambda,x0,dxis):
    # Assumes a simple initial grid, where dx~dxi, dy~deta, dz~dzeta
    initial_xes = [dx_dxi[0]*xi+x0[0],dx_dxi[4]*eta*0+x0[1],
        dx_dxi[8]*zeta*0+x0[2]]
    initial_xes = [x.subs({t:0}) for x in initial_xes]
    out = [initial_xes[ind] +
           sympy.integrate(dx_dlambda[ind],t) for ind in range(3)]
    return out
```

```
def MASA_full_var(x0,xx,ax,fx,Lx,xy,ay,fy,Ly,xz,az,fz,Lz,xt,at,ft,Lt,
                  shock_strength,shock_position):
    out = (x0+
            xt*ft(at*sympy.pi*t/Lt)+
            xx*fx(ax*sympy.pi*xi/Lx)+
            xy*fy(ay*sympy.pi*eta/Ly)+
            xz*fz(az*sympy.pi*zeta/Lz))
    if shock_strength != 0.:
        out += shock_strength*H(xi-shock_position)
    return out

def test_riemann(ntests):
    """
Test the accuracy of numerical integration of the Euler equations.

Compute weak manufactured source terms from exact, discontinuous, solutions
for the Euler equations. In particular, use a collection of exact,
one-dimensional Riemann problems, rotated so as to be three-dimensional.

Parameters
----------
ntests : int
    Number of random trials to compute.

Returns
-------
random_euler_riemann.dat : ASCII file
    Contains solution parameters and corresponding source term values.
"""
    f = open('random_euler_riemann.dat','w')
#    f.write('\n')
    f.write('%problem #, theta(deg), phi(deg), '
            +'source_rho, source_p, source_u, source_v, source_w')
    ranges = [[t,0.1,1],[xi,-1,1],[eta,-1,1],[zeta,-1,1]]
    random.seed(100)
    S_prime_list = []
    for indn in range(ntests):
        print "Test case #"+str(indn)
        n_choices = [0,1,2,3]#,4]
        theta_min, theta_max = 0, numpy.pi*0.5
        phi_min, phi_max = 0, numpy.pi*0.5
        n,theta,phi = [random.choice(n_choices),
                       random.random()*(theta_max-theta_min),
                       random.random()*(phi_max-phi_min)]
        opts = {'set_riemann':n,'theta':theta,'phi':phi}
        sol = exact_Euler_sols('riemann_problem',opts)
```

```
        S_prime = Euler_UCS(sol).balance_integrate(ranges)
        S_prime_list.append(S_prime)
        f.write('\n'+', '.join([str(item) for item in (
                        n,theta/numpy.pi*180,phi/numpy.pi*180,
                        S_prime[0],S_prime[1],S_prime[2],
                        S_prime[3],S_prime[4])]))
    f.close()
    return S_prime_list
if __name__ == "__main__":
    S_prime_list = test_riemann(10)


#    eqn = Euler_UCS(MASA_with_pinned_bounds([[0,1],[0,1],[0,1]],nxes=(100,1,1)))
#    print eqn.balance_diff()[0]
#    out = eqn.balance_lambda_init()
#    import pdb;pdb.set_trace()
```

Appendix C

Streamer Fortran Core

Included here is the most critical code for the Fortran Core library included in Streamer v3.0, along with the associated Makefile and Python setup function. For all other code related to Streamer development, see the Github repository at tt https://github.com/woodscn.

C.1 Streamer v3.0

C.1.1 setup.py

This uses the Numpy `distutils` package to make the Fortran Core library importable into Python. It is run with the usual command, `python setup.py install`, after the library has been compiled.

```
addl_source_dir = "./f90src/"
def configuration(parent_package='',top_path=None,compile_type='debug'):
    from numpy.distutils.misc_util import Configuration
    from os import getcwd
    opts={'debug':'--debug --noopt --noarch','run':'--opt=-O3'}
    config = Configuration('Streamer',parent_package,top_path)
    config.add_extension('Godunov_driver','Godunov_driver.f90',
                         extra_compile_args=['-J./bindings'],
                         include_dirs=['/usr/local/include',
                                       '/usr/include','./bindings'],
#                         libraries=['lapack','recipes_f90','minpack','cgns',
#                                    'Streamer'],
                         libraries = ['Streamer'],
                         library_dirs=['/usr/local/lib','/usr/lib','.']
                         )
```

```
    config.add_extension("Source_functions",
                         addl_source_dir+'Source_functions.f90',
                         extra_compile_args=['-J./bindings'],
                         include_dirs=['/usr/local/include',
                                       '/usr/include','./bindings'],
#                         libraries=['lapack','recipes_f90','minpack','cgns',
#                                    'Streamer'],
                         libraries=['Streamer'],
                         library_dirs=['/usr/local/lib','/usr/lib','.']
                         )
    config.add_extension('grid_motion_driver','grid_motion_driver.f90',
                         extra_compile_args=['-J./bindings'],
                         include_dirs=['/usr/local/include',
                                       '/usr/include','./bindings'],
                         libraries = ['Streamer'],
                         library_dirs=['/usr/local/lib','/usr/lib','.']
                         )
    config.add_extension('BoundaryConditionsStuff',
                         addl_source_dir+'BoundaryConditionsStuff.f90',
                         extra_compile_args=["-fbounds-check",'-J./bindings'],
                         include_dirs=['/usr/local/include',
                                       '/usr/include','./bindings'],
                         libraries=['lapack','Streamer'],
                         library_dirs=['/usr/local/lib','/usr/lib','.']
                         )

    config.add_extension('TimeAdvancementStuff',
                         addl_source_dir+'TimeAdvancementStuff.f90',
                         extra_compile_args=["-fbounds-check",'-J./bindings'],
                         include_dirs=['/usr/local/include',
                                       '/usr/include','./bindings'],
                         libraries=['lapack'],
                         library_dirs=['/usr/local/lib','/usr/lib','.']
                         )

#    config.add_extension('STLA_IO',addl_source_dir+'stla_io.f90',
#                         extra_compile_args=["-fbounds-check"],
#                         include_dirs=['/usr/local/include','/usr/include'],
#                         libraries=['lapack'],
#                         library_dirs=['/usr/local/lib','/usr/lib']
#                         )


#    config.add_extension('CGNS_Interface',addl_source_dir+'cgns_interface.f90',
#                         include_dirs=['/usr/local/include','/usr/include'],
#                         libraries=['cgns','lapack'],
#                         library_dirs=['/usr/local/lib','/usr/lib'],
```

```
#                          extra_compile_args=["-fbounds-check"]
#                          )
    return config

def prepend_GU(fin):
    fout_name = addl_source_dir+fin.split('.')[0]+'GU.f90'
    fout = open(fout_name,'w')
    fout.write('''! This file is automatically generated from
! '''+fin.split('x')[0]+''' and from GeneralUtilities.f90
''')
    f=open(addl_source_dir+'GeneralUtilities.f90')
    for line in f:
        fout.write(line)
    f.close()
    f=open(addl_source_dir+fin)
    for line in f:
        fout.write(line)
    f.close()
    fout.close()
    return(fout_name)



if __name__ == "__main__":
    from numpy.distutils.core import setup
    setup(**configuration(top_path='').todict())
```

C.1.2 Makefile

The Fortran Core library itself is built using Make. The library can be compiled using `make all`, and it can also be tested using `make check`. The unit tests require access to some of the MINPACK files, available on netlib [24].

```
OS = $(shell uname)
RM = rm -f
AR = ar
ARFLAGS = rv
FC = gfortran
# Debug options for gfortran
FFLAGS = -I$(MODDIR) -I/usr/local/include -J$(MODDIR) \
   -fbounds-check -O0 -g3 -p -fPIC -arch x86_64 -ff2c \
   -framework Accelerate #-fopenmp
# # Release options for gfortran
```

```
# FFLAGS = -I$(MODDIR) -I/usr/local/include -J$(MODDIR) -O3
# # Profiling options for gfortran
# FFLAGS = -I$(MODDIR) -I/usr/local/include -J$(MODDIR) -O3 -p

# # Debug options for ifort
# FFLAGS = -I/usr/local/include -module $(MODDIR) -O0 -g3 -p -check bounds
# # Release options for ifort
# FFLAGS = -I/usr/local/include -module $(MODDIR) -fast #-fast-transcendentals
# # Profiling options for ifort
# FFLAGS = -I/usr/local/include -module $(MODDIR) -fast -p

LDFLAGS = -L./ -llapack -lStreamer -lStreamerTest \
   -lStreamer -lminpack -llapack#-lcgns

MODDIR = ./bindings
LIBSRCDIR = ./f90src
LIBTESTSRCDIR = ./f90src/test
MINPACKSRCDIR = ./minpack
DATDIR = ./dat
EXEDIR = ./exe
DRVDIR = .
LIBSOURCES = GeneralUtilities.f90 \
Riemann.f90 \
TimeAdvancementStuff.f90 \
Godunov.f90 \
ODE_Solvers.f90 \
BoundaryConditionsStuff.f90 \
FortranNormalVectors.f90 \
cgns_interface.f90
all: lib
.POSIX:
lib: libStreamer.a
# $(RM) $(LIBSRCDIR)/*.o
# @echo Free object files have been removed. \
   object files can be found in libStreamer.a
#drivers = $(
libStreamer.a: $(LIBSRCDIR)/GeneralUtilities.o \
$(LIBSRCDIR)/Riemann.o \
$(LIBSRCDIR)/TimeAdvancementStuff.o \
$(LIBSRCDIR)/Godunov.o \
$(LIBSRCDIR)/BoundaryConditionsStuff.o \
$(LIBSRCDIR)/FortranNormalVectors.o \
$(LIBSRCDIR)/grid_motion.o
# $(LIBSRCDIR)/cgns_interface.o
#       End dependencies
$(AR) $(ARFLAGS) $@ $?
$(LIBSRCDIR)/Godunov.o: $(LIBSRCDIR)/GeneralUtilities.o \
```

```
$(LIBSRCDIR)/Riemann.o
$(LIBSRCDIR)/ODE_Solvers.o: $(LIBSRCDIR)/Source_functions.o
$(LIBSRCDIR)/Source_functions.o: $(LIBSRCDIR)/GeneralUtilities.o
$(LIBSRCDIR)/BoundaryConditionsStuff.o: $(LIBSRCDIR)/GeneralUtilities.o
check: lib libStreamerTest.a libminpack.a \
$(LIBTESTSRCDIR)/test_program.f90 \
$(DRVDIR)/Godunov_driver.o \
$(DRVDIR)/grid_motion_driver.o
#  End dependencies
$(FC) $(FFLAGS) $(LIBTESTSRCDIR)/test_program.f90 \
$(DRVDIR)/Godunov_driver.o \
$(DRVDIR)/grid_motion_driver.o  -o $(LIBTESTSRCDIR)/f90tests $(LDFLAGS)
$(LIBTESTSRCDIR)/f90tests $(DATDIR)
libStreamerTest.a: $(LIBTESTSRCDIR)/GeneralUtilities_tester.o \
$(LIBTESTSRCDIR)/Godunov_tester.o \
$(LIBTESTSRCDIR)/Riemann_tester.o \
$(LIBTESTSRCDIR)/grid_motion_tester.o \
$(LIBTESTSRCDIR)/UCS_tester.o
# $(LIBTESTSRCDIR)/ODE_Solvers_tester.o
$(AR) $(ARFLAGS) $@ $?
libminpack.a: $(MINPACKSRCDIR)/lmder1.o \
$(MINPACKSRCDIR)/dpmpar.o \
$(MINPACKSRCDIR)/lmder.o \
$(MINPACKSRCDIR)/lmpar.o \
$(MINPACKSRCDIR)/qrsolv.o \
$(MINPACKSRCDIR)/enorm.o \
$(MINPACKSRCDIR)/qrfac.o
$(AR) $(ARFLAGS) $@ $?
$(LIBTESTSRCDIR)/Godunov_tester.o: $(DRVDIR)/Godunov_driver.o
$(LIBTESTSRCDIR)/grid_motion_tester.o: $(DRVDIR)/grid_motion_driver.o
$(LIBTESTSRCDIR)/UCS_tester.o: $(DRVDIR)/Godunov_driver.o \
$(DRVDIR)/grid_motion_driver.o
#$(EXEDIR)/test_program: $(SRCDIR)/test_program.f90 lib
# $(FC) $(FFLAGS) $(LDFLAGS) $< -o $@
%.o: %.f90
$(FC) -c $(FFLAGS) $< -o $@
clean:
$(RM) $(LIBSRCDIR)/*.o $(LIBTESTSRCDIR)/*.o $(DRVDIR)/Godunov_driver.o \
$(MODDIR)/*.mod libStreamer.a libStreamerTest.a *.so *.a *.mod *.o \
minpack/*.o
```

C.1.3 GeneralUtilities.f90

This module contains routines that are commonly used in UCS.

```
module GeneralUtilities
  real(8), parameter :: PI = 3.141592653589793
  real(8), parameter :: EPS = 5.d-15
  real(8), parameter :: EPSs = 1d-4
  real(8), parameter :: gamma_const = 1.4d0
  real(8), parameter :: gamma1 = 1.d0/(gamma_const-1.d0)
  real(8), parameter :: gamma2 = (gamma_const-1.d0)
  real(8), parameter :: gamma3 = (gamma_const - 1.d0)/(2.d0*gamma_const)
  real(8), parameter :: gamma4 = 1.d0/gamma3
  real(8), parameter :: gamma5 = (gamma_const-1.d0)/(gamma_const+1.d0)
  real(8), parameter :: gamma6 = 1.d0/(gamma_const+1.d0)
  real(8), parameter :: gamma7 = 1.d0/gamma3
  real(8), dimension(7), parameter :: dxi_a = [1., .5, .25, .2, 2., 4., 5.]
  real(8), dimension(7), parameter :: deta_a = [1., .5, .25, .2, 2., 4., 5.]
  real(8), dimension(7), parameter :: dzeta_a = [1., .5, .25, .2, 2., 4., 5.]
contains
  pure function MetricInverse(Metric)
    implicit none
    ! Given a list of metric variables:
    ! dx/dxi, dy/dxi, dz/dxi, dx/deta, dy/deta, dz/deta,
    ! dx/dzeta, dy/dzeta, dz/dzeta
    ! ( A, B, C, L, M, N, P, Q, R )
    ! Return the inverse of this list:
    ! dxi/dx, deta/dx, dzeta/dx, dxi/dy, deta/dy, dzeta/dy,
    ! dxi/dz, deta/dz, dzeta/dz.
    real(8), dimension(9), intent(in) :: Metric
    real(8), dimension(9) :: MetricInverse
    real(8) :: J

    J = Jacobian(Metric)
    MetricInverse = [&
         Metric(5)*Metric(9) - Metric(6)*Metric(8), &
         Metric(3)*Metric(8) - Metric(2)*Metric(9), &
         Metric(2)*Metric(6) - Metric(3)*Metric(5), &
         Metric(6)*Metric(7) - Metric(4)*Metric(9), &
         Metric(1)*Metric(9) - Metric(3)*Metric(7), &
         Metric(3)*Metric(4) - Metric(1)*Metric(6), &
         Metric(4)*Metric(8) - Metric(5)*Metric(7), &
         Metric(2)*Metric(7) - Metric(1)*Metric(8), &
         Metric(1)*Metric(5) - Metric(2)*Metric(4) ]/J
  end function MetricInverse

  function MetrictoMatrix(Metric)
    implicit none
    ! Given a 9-element, rank-1 array, convert it to a 3x3 matrix,
    ! such that the elements are ordered column-wise:
    ! [ A, B, C, L, M, N, P, Q, R ]
```

```
    !
    !              | |
    !             \   /
    !              \ /
    !
    !            A, L, P
    !            B, M, Q
    !            C, N, R
    !
    ! This is consistent with the statement:
    ! dx = matmul(MetrictoMatrix(Metric),dxi)
    real(8), dimension(9), intent(in) :: Metric
    real(8), dimension(3,3) :: MetrictoMatrix

    MetrictoMatrix = reshape(Metric,[3,3])
  end function MetrictoMatrix

!!$  function NormalizedMetric(in)
!!$    implicit none
!!$    real(8), dimension(9), intent(in) :: in
!!$    real(8), dimension(9) :: NormalizedMetric
!!$    integer :: n
!!$    do n = 1, 3
!!$       NormalizedMetric(n:9:3) = in(n:9:3)/sqrt(sum(in(n:9:3)**2))
!!$    end do
!!$  end function NormalizedMetric

  subroutine ComputationalGrads(metric,jac,grad_xi,grad_eta,grad_zeta)
    implicit none
    ! Metric has the form:
    !   1  2  3  4  5  6  7  8  9
    ! [ A, B, C, L, M, N, P, Q, R ]
    ! dx/dxi, dy/dxi, dz/dxi, dx/deta, dy/deta,
    ! dz/deta, dx/dzeta, dy/dzeta, dz/dzeta
    !
    ! ComputationalGrads returns:
    ! grad_xi   = [   dxi/dx,   dxi/dy,   dxi/dz ]
    ! grad_eta  = [  deta/dx,  deta/dy,  deta/dz ]
    ! grad_zeta = [ dzeta/dx, dzeta/dy, dzeta/dz ]
    real(8), intent(in), dimension(9) :: metric
    real(8), intent(in) :: jac
    real(8), intent(out), dimension(3) :: grad_xi
    real(8), intent(out), dimension(3) :: grad_eta
    real(8), intent(out), dimension(3) :: grad_zeta
    real(8), dimension(9) :: inv_metric

    inv_metric = MetricInverse(metric)
```

```
    grad_xi   = inv_metric(1:9:3)
    grad_eta  = inv_metric(2:9:3)
    grad_zeta = inv_metric(3:9:3)
  end subroutine ComputationalGrads

  real(8) pure function Jacobian(in)
    implicit none
    ! Computes the determinant of a 3 x 3 matrix using a brute-force method:
    !       | A L P |
    !  J =  | B M Q |
    !       | C N R |
    ! Assumes the structure of in(:) is :
    !   1  2  3  4  5  6  7  8  9
    ! [ A, B, C, L, M, N, P, Q, R ]
    real(8), dimension(9), intent(in) :: in
    Jacobian = &
         in(1)*in(5)*in(9) - in(1)*in(6)*in(8) + & ! A*M*R - A*N*Q
         in(2)*in(6)*in(7) - in(2)*in(4)*in(9) + & ! B*N*P - B*L*R
         in(3)*in(4)*in(8) - in(3)*in(5)*in(7)     ! C*L*Q - C*M*P
  end function Jacobian

!!$  function GradstoMatrix(Grad1,Grad2,Grad3)
!!$    !
!!$    ! matmul(GradstoMatrix,vector) ==
!!$    ! [ sum(GradstoMatrix(1,:)*vector),
!!$    !   sum(GradstoMatrix(2,:)*vector),
!!$    !   sum(GradstoMatrix(3,:)*vector) ]
!!$    implicit none
!!$    real(8), dimension(3), intent(in) :: Grad1, Grad2, Grad3
!!$    real(8), dimension(3,3) :: GradstoMatrix
!!$    GradstoMatrix = transpose(reshape([Grad1,Grad2,Grad3],[3,3]))
!!$  end function GradstoMatrix

  subroutine TwoDGradient(in,dx,dy,nx,ny,gradx,grady)
    implicit none
    ! Compute the 2-dimensional gradient of a matrix using central
    ! differencing wherever possible. Returns gradient matrices
    ! the same size and shape as the input matrix. If the input matrix
    ! has length 1 in either dimension, then it is assumed that the
    ! gradient in that dimension is 0.
    real(8), intent(in), dimension(nx,ny) :: in
    real(8), intent(in) :: dx, dy
    integer, intent(in) :: nx, ny
    real(8), intent(out), dimension(nx,ny) :: gradx, grady
    integer :: i, j

    if(nx>1)then
```

```
      ! Central differencing where possible
      gradx(2:nx-1,:) = .5d0*(in(3:nx,:)-in(1:nx-2,:))/dx
      ! Forward & backward differencing elsewhere
      gradx(1,:)  = (in( 2,:)-in(   1,:))/dx
      gradx(nx,:) = (in(nx,:)-in(nx-1,:))/dx
    else
      gradx(:,:) = 0.d0
    end if
    if(ny>1)then
      ! Central differencing where possible
      grady(:,2:ny-1) = .5d0*(in(:,3:ny)-in(:,1:ny-2))/dy
      ! Forward & backward differencing elsewhere
      grady(:,1)  = (in(:, 2)-in(:,   1))/dy
      grady(:,ny) = (in(:,ny)-in(:,ny-1))/dy
    else
      grady(:,:) = 0.d0
    end if
  end subroutine TwoDGradient

  function MatrixInverse(in)
    ! Compute the inverse of a 3x3 matrix
    implicit none
    real(8), dimension(3,3), intent(in) :: in
    real(8), dimension(3,3) :: MatrixInverse
    real(8) :: J

    J = ( &
         in(1,1)*in(2,2)*in(3,3) + &
         in(1,2)*in(2,3)*in(3,1) + &
         in(1,3)*in(2,1)*in(3,2) - &
         in(1,1)*in(2,3)*in(3,2) - &
         in(1,2)*in(2,1)*in(3,3) - &
         in(1,3)*in(2,2)*in(3,1) )

    MatrixInverse = transpose(reshape( [ &
         in(3,3)*in(2,2)-in(3,2)*in(2,3) , &
         in(3,2)*in(1,3)-in(3,3)*in(1,2) , &
         in(2,3)*in(1,2)-in(2,2)*in(1,3) , &
         in(3,1)*in(2,3)-in(3,3)*in(2,1) , &
         in(3,3)*in(1,1)-in(3,1)*in(1,3) , &
         in(2,1)*in(1,3)-in(2,3)*in(1,1) , &
         in(3,2)*in(2,1)-in(3,1)*in(2,2) , &
         in(3,1)*in(1,2)-in(3,2)*in(1,1) , &
         in(2,2)*in(1,1)-in(2,1)*in(1,2) ] &
         ,[3,3])/J)
!!$    if(.true. .and. maxval(matmul(in,MatrixInverse)&
!!$           -reshape([1,0,0,0,1,0,0,0,1],[3,3]))**2>1.d-15)then
```

```
!!$          write(*,*) "MatrixInverse failed!!"
!!$          write(*,*) matmul(in,MatrixInverse)
!!$          stop
!!$    end if
  end function MatrixInverse

  function vectorProjection(in,normal)
    implicit none
    real(8), intent(in), dimension(3) :: in
    real(8), intent(in), dimension(3) :: normal
    real(8), dimension(3) :: vectorProjection
    vectorProjection = normal*&
         (dot_product(in,normal)/dot_product(normal,normal))
  end function vectorProjection

  real(8) function SoundSpeed(point)
    implicit none
    real(8), intent(in), dimension(21) :: point
    SoundSpeed = sqrt(1.4d0*point(1)/point(2))
  end function SoundSpeed

end module GeneralUtilities
```

C.1.4 Riemann.f90

This module provides the machinery required in order to solve Riemann problems for the Euler Equations in the unified coordinates.

```
module Riemann
  use GeneralUtilities
  logical :: verbose = .false.
  logical :: riemann_test_flag
  logical :: test_flag = .false.
  real(8), dimension(4) :: test_sol
  real(8), dimension(:,:), allocatable :: exact_sol
contains

!!$  function fan_Jacobian(J0,psi,u0,u_grid,a0,EV,S,n)
!!$    real(8), intent(in) :: J0, psi, u0, u_grid, a0, EV, S
!!$    integer, intent(in) :: n ! 1 for left fan, 2 for right fan
!!$
!!$    real(8) :: bn, exp
!!$    integer :: pm
!!$
```

```
!!$    pm = (-1)**n
!!$    bn = -pm*(gamma_const+1d0)/((gamma_const-1d0)*(u0-ugrid)-pm*2d0*a0)
!!$    exp = (1d0-gamma_const)/(2d0*gamma_const)
!!$
!!$    fan_Jacobian = J0*((psi**exp-bn)/(psi**exp*(1d0-bn)))**(&
!!$         (4d0*a0*EV)/(S*(gamma_const+1d0)))
!!$  end function fan_Jacobian

  subroutine riemann_solve(left, right, dir, nx, x, out, max_wave_speed,&
       riemann_middle_states, riemann_wave_speeds,ierr_out)
    ! Riemann_solve accepts two physical flow states of the form:
    !     [ pressure, mass density, velocity_normal, velocity_tangential_1,
    !       velocity_tangential_2 ]
    !   as well as an average geometric state of the form:
    !     [ pressure, density, v_norm, v_tan1, v_tan2, A, B, C, L, M, N,
    !       P, Q, R, U, V, W, x, y, z, J ]
    ! Riemann_solve solves the 1-dimensional riemann problem given by
    !   the left and right states, as well as the grid motion U, V, W.
    !   It returns the state value [ p, rho, v_norm, v_tan1, v_tan2 ]
    !   at x/t = 0. It also updates the maximum wavespeed encountered,
    !   for use in computing an optimal time step.
    implicit none
    real(8), dimension(21), intent(in) :: left, right
    integer, intent(in) :: dir, nx
    real(8), dimension(nx), intent(in) :: x
    real(8), dimension(5,nx), intent(out) :: out
    real(8), intent(out) :: max_wave_speed
    real(8), dimension(4), intent(out), optional :: riemann_middle_states
    real(8), dimension(5), intent(out), optional :: riemann_wave_speeds
    integer, intent(out), optional :: ierr_out

    real(8) :: DL, PL, UL, VL, WL, AL
    real(8) :: DR, PR, UR, VR, WR, AR
    real(8) :: Uavg
    real(8), dimension(9) :: met_inv
    real(8) :: J, S
    real(8) :: Pstar, Ustar, DstarL, DstarR
    real(8), dimension(4) :: riemann_middle_states_temp
    real(8), dimension(5) :: riemann_wave_speeds_temp
    real(8), parameter :: tol = 1.d-10
    real(8) :: PsiL, PsiR, temp, fL, fR, dfL, dfR
    integer :: n, ierror

    ierror = 0
    temp = 1d0
    DL =  left(2); PL =  left(1);
    DR = right(2); PR = right(1);
```

```
!!$    select case(dir)
!!$    case(1)
       UL =  left(3); VL =  left(4); WL =  left(5)
       UR = right(3); VR = right(4); WR = right(5)
       Uavg = .5d0*(left(15)+right(15))
!!$    case(2)
!!$       UL =  left(4); VL =  left(5); WL =  left(3)
!!$       UR = right(4); VR = right(5); WR = right(3)
!!$       Uavg = .5d0*(left(16)+right(16))
!!$    case(3)
!!$       UL =  left(5); VL =  left(3); WL =  left(4)
!!$       UR = right(5); VR = right(3); WR = right(4)
!!$       Uavg = .5d0*(left(17)+right(17))
!!$    end select
    AL = sqrt(gamma_const*PL/DL); AR = sqrt(gamma_const*PR/DR)

    met_inv = MetricInverse(.5d0*(left(6:14)+right(6:14)));
    J = Jacobian(.5d0*(left(6:14)+right(6:14)))
!!$    if(abs(J-1d-6)>1d-7)then
!!$       write(*,*) "J = ",J,"  Metric = ",.5d0*(left(6:14)+right(6:14))
!!$       read(*,*)
!!$    end if
    S = sqrt(sum((J*met_inv(dir:9:3))**2))
!    write(*,*) "S/J = ", S/J
    Pstar = guessp(left,right)
!!$    if(verbose)write(*,*) "Initial guess P = " , Pstar
    temp = 1.d0 ; n = 0
    do while(abs(temp) .gt. tol)
!!$       if(verbose)write(*,*) "Step ",n , "Pstar = " , Pstar;
       n = n + 1
       if(n .gt. 10)then
          ierror = 1
          exit
       end if
       PsiL = Pstar/PL
       call u_fun( left,Pstar,fL,dfL)
       PsiR = Pstar/PR
       call u_fun(right,Pstar,fR,dfR)
       temp = ( UR - UL + fR + fL )/( dfL + dfR )
       Pstar = max( Pstar - temp , tol )
!!$       if(verbose)write(*,*) "fL , fR , Pstar = " , fL , fR , Pstar
    end do
    Ustar = .5*(UR+fR+UL-fL)
    DstarL = beta(Pstar/PL)*DL
    DstarR = beta(Pstar/PR)*DR
    riemann_middle_states_temp = [Pstar,Ustar,DstarL,DstarR]
    riemann_wave_speeds_temp = wave_speeds(&
```

```
         left,right,dir,riemann_middle_states_temp,Uavg,J,S)
    max_wave_speed = maxval(abs(riemann_wave_speeds_temp))
!!$    if(verbose)write(*,*) Ustar , DstarL , DstarR
    do n = 1, nx
       call sample(x(n),left,right,riemann_middle_states_temp,&
            riemann_wave_speeds_temp,Uavg,J,S,out(:,n))
    end do
    if(present(riemann_middle_states))&
         riemann_middle_states = riemann_middle_states_temp
    if(present(riemann_wave_speeds))&
         riemann_wave_speeds = riemann_wave_speeds_temp
    if(present(ierr_out)) ierr_out = ierror
!!$    if(verbose)write(*,*)"RiemannSolve = ", out
  end subroutine riemann_solve

  function wave_speeds(left,right,dir,riemann_middle_states,Uavg,J,S)
    implicit none
    real(8), dimension(21), intent(in) :: left, right
    integer, intent(in) :: dir
    real(8), dimension(4), intent(in) :: riemann_middle_states
    real(8), intent(in) :: Uavg
    real(8), intent(in) :: J, S
    real(8), dimension(5) :: wave_speeds

    real(8), dimension(9) :: met_inv
    real(8) :: Pstar, Ustar, DstarL, DstarR

    wave_speeds = 0d0
    Pstar = riemann_middle_states(1)
    Ustar = riemann_middle_states(2)
    DstarL = riemann_middle_states(3)
    DstarR = riemann_middle_states(4)
    if(Pstar/left(1)>1d0)then
       wave_speeds(1) = S/J*(left(3)-Uavg-sqrt(left(1)*gamma_const/left(2))&
            *sqrt((gamma_const+1d0)/(2d0*gamma_const)*(Pstar/left(1)-1d0)+1d0))
    else
       wave_speeds(1) = S/J*(left(3)-Uavg-sqrt(left(1)*gamma_const/left(2)))
       wave_speeds(2) = S/J*(Ustar-Uavg-sqrt(Pstar*gamma_const/DstarL))
    end if
    wave_speeds(3) = S/J*(Ustar-Uavg)
    if(Pstar/right(1)>1d0)then
       wave_speeds(5) = S/J*(right(3)-Uavg+sqrt(right(1)*gamma_const/right(2))&
            *sqrt(&
            (gamma_const+1d0)/(2d0*gamma_const)*(Pstar/right(1)-1d0)+1d0))
    else
       wave_speeds(4) = S/J*(Ustar-Uavg+sqrt(Pstar*gamma_const/DstarR))
       wave_speeds(5) = S/J*(right(3)-Uavg+sqrt(right(1)*gamma_const/right(2)))
```

```
  end if
end function wave_speeds

real(8) pure function guessp(left,right)
  implicit none
  real(8), dimension(:), intent(in) :: left, right
  real(8) :: aL, aR, gL, gR, tol
  aL = sqrt(gamma_const* left(1)/ left(2))
  aR = sqrt(gamma_const*right(1)/right(2))
  tol = 1d-8
  ! Linearised guess
  guessp = .5*(left(1)+right(1))&
       -.125*(right(3)-left(3))*(left(2)+right(2))*(aL+aR)
  if(.not.( guessp .gt. min(left(1),right(1)) .and. &
       guessp .lt. max(left(1),right(1)) &
       .and. max(left(1),right(1))/min(left(1),right(1)) .le. 2.0))then
     if(guessp .lt. min(left(1),right(1)))then
        ! Two-rarefaction solution
        guessp = (&
             (aL+aR-.5*(gamma_const-1.)*(right(3)-left(3)))&
             /(aL/left(1)**((gamma_const-1.)/(2.*gamma_const))&
             +aR/right(1)**((gamma_const-1.)/(2.*gamma_const)))&
             )**(2.*gamma_const/(gamma_const-1.))
     else
        ! Two-shock solution
        gL=sqrt( 2./((gamma_const+1.)* left(2))/(guessp+ left(1)&
             *(gamma_const-1.)/(gamma_const+1.)))
        gR=sqrt( 2./((gamma_const+1.)*right(2))/(guessp+right(1)&
             *(gamma_const-1.)/(gamma_const+1.)))
        guessp = max(tol,(gL*left(1)+gR*right(1)-(right(3)-left(3)))/(gL+gR))
     end if
  end if
  !          guessp = .5*( PL + PR )
end function guessp

pure subroutine u_fun(in,pstar,f,df)
  real(8), dimension(:), intent(in) :: in
  real(8), intent(in) :: pstar
  real(8) , intent(out) :: f , df
  real(8) :: A, B, psi, a0
  psi = pstar/in(1)
  a0  = sqrt(gamma_const*in(1)/in(2))
  if( psi .gt. 1. )then
     A = 2.d0/((gamma_const+1.d0)*in(2))
     B = (gamma_const-1.d0)/(gamma_const+1.d0)*in(1)
     f = in(1)*(psi-1.d0)*sqrt(A/(in(1)*psi+B))
     df= sqrt(A/(B+in(1)*psi))*(1.d0-in(1)*(psi-1.d0)/(2.*(B+psi*in(1))))
```

```
    else
       f = 2.d0*a0/(gamma_const-1.d0)*(psi**((gamma_const-1.d0)/&
            (2.d0*gamma_const))-1.d0)
       df= 1.d0/(in(2)*a0)*psi**((gamma_const+1.d0)/(-2.d0*gamma_const))
    end if
  end subroutine u_fun

  pure function beta(psi)
    implicit none
    real(8), intent(in) :: psi
    real(8) :: beta
    if( psi .gt. 1 )then
       beta = ((gamma_const+1.)*psi+gamma_const-1.)/&
            (gamma_const+1.+(gamma_const-1.)*psi)
    else
       beta = psi**(1.d0/gamma_const)
    end if
  end function beta

  pure subroutine sample(x,left,right,riemann_middle_states,&
       riemann_wave_speeds,Uavg,J,S,out)
    implicit none
    real(8), dimension(:), intent(in) :: left, right, riemann_middle_states
    real(8), dimension(:), intent(in) :: riemann_wave_speeds
    real(8), intent(in) :: x, J, S, Uavg
    real(8), dimension(:) , intent(out) :: out
    real(8) :: PsiL , Pstar, Ustar, DstarL, DstarR
    real(8) :: PsiR , aL , aR , betaL , betaR
    real(8) :: cL , cR , cLT , cLH , cRT , cRH , h
    logical :: test_flag
!!$    test_flag = .false.
!!$    if(verbose) test_flag = .true.
    Pstar  = riemann_middle_states(1); Ustar  = riemann_middle_states(2)
    DstarL = riemann_middle_states(3); DstarR = riemann_middle_states(4)
    PsiL = Pstar/left(1)
    PsiR = Pstar/right(1)
    aL   = sqrt(gamma_const* left(1)/ left(2))
    aR   = sqrt(gamma_const*right(1)/right(2))
    betaL= DstarL/ left(2)
    betaR= DstarR/right(2)
    if( riemann_wave_speeds(3) .gt. x )then
!!$       write(*,*) "The boundary lies to the left of the contact wave"
       out(4) = left(4) ; out(5) = left(5)
       if( PsiL .gt. 1.d0 )then
!!$          if(test_flag) write(*,*) " Left shock"
          cL = riemann_wave_speeds(1)
!!$          if(test_flag) write(*,*) "Left shock speed = " , cL
```

```
         if( cL .gt. x )then
!!$            write(*,*) " The boundary lies to the left of the shock"
            out(1) = left(1)
            out(3) = left(3)
            out(2) = left(2)
         else
!!$            write(*,*) " The boundary lies in the left central region"
            out(1) = Pstar
            out(3) = Ustar
            out(2) = DstarL
         end if
      else
!!$         if(test_flag) write(*,*) "Left rarefaction wave"
         cLT = riemann_wave_speeds(1)
!!$         if(test_flag) write(*,*) "Left rarefaction tail speed = " , cLT
         cLH = riemann_wave_speeds(2)
!!$         if(test_flag) write(*,*) "Left rarefaction head speed = " , cLH
         if( cLT .gt. x )then
!!$            write(*,*) " The boundary lies to the left of the wave"
            out(1) = left(1)
            out(3) = left(3)
            out(2) = left(2)
         elseif( cLH .lt. x )then
!!$            write(*,*) "The boundary lies in the left central region"
            out(1) = Pstar
            out(3) = Ustar
            out(2) = DstarL
         else
!!$            write(*,*) "The boundary lies within the left expansion wave"
            out(1) = left(1)*(2d0*gamma6+gamma5/aL*&
                 (left(3)-Uavg-J/S*x))**gamma7
            out(3) = left(3) - 2.d0*aL/(gamma_const-1.d0)*((out(1)/left(1))&
                 **((gamma_const-1.d0)/(2.d0*gamma_const))-1.d0)
            out(2) = left(2)*(out(1)/left(1))**(1.d0/gamma_const)
         end if
      end if
   else
!!$      write(*,*) " The boundary lies to the right of the contact wave"
      out(4) = right(4); out(5) = right(5)
      if( PsiR .gt. 1.d0 )then
!!$         if(test_flag) write(*,*) " Right shock"
         cR = riemann_wave_speeds(5)
!!$         if(test_flag) write(*,*) " Right shock speed = " , cR
         if( cR .lt. x )then
!!$            write(*,*) " The boundary lies to the right of the shock"
            out(1) = right(1)
            out(3) = right(3)
```

```
            out(2) = right(2)
          else
!!$             write(*,*) " The boundary lies in the right central region"
            out(1) = Pstar
            out(3) = Ustar
            out(2) = DstarR
          end if
       else
!!$          if(test_flag) write(*,*) " Right rarefaction wave"
          cRT = riemann_wave_speeds(5)
!!$          if(test_flag) write(*,*) "Right rarefaction tail speed = " , cRT
          cRH = riemann_wave_speeds(4)
!!$          if(test_flag) write(*,*) "Right rarefaction head speed = " , cRH
          if( cRT .lt. x )then
!!$             write(*,*) " The boundary lies to the right of the wave"
            out(1) = right(1)
            out(3) = right(3)
            out(2) = right(2)
          elseif( cRH .gt. x )then
!!$             write(*,*) "The boundary lies in the right central region"
            out(1) = Pstar
            out(3) = Ustar
            out(2) = DstarR
          else
!!$             write(*,*) " The boundary lies within the right expansion wave"
            out(1) = right(1)*(2d0*gamma6-gamma5/&
                 aR*(right(3)-Uavg-J/S*x))**gamma7
            out(3) = right(3) + 2.d0*aR/(gamma_const-1.d0)*((out(1)/right(1))&
                 **((gamma_const-1.d0)/(2.d0*gamma_const))-1.d0)
            out(2) = right(2)*(out(1)/right(1))**(1.d0/gamma_const)
          end if
       end if
    end if
  end subroutine sample
end module Riemann
```

C.1.5 Godunov.f90

This module provides routines for advancing a UCS simulation forward one time step.

```
module Godunov
  use GeneralUtilities
  use Riemann
  implicit none
  real(8), parameter :: max_dt = 1.d0
```

```
!  real(8), dimension(7), parameter :: dxi_a = [1.d0, .5d0, .25d0, .2d0, &
!       2.d0, 4.d0, 5.d0]
!  real(8), dimension(7), parameter :: deta_a = [1.d0, .5d0, .25d0, .2d0, &
!       2.d0, 4.d0, 5.d0]
!  real(8), dimension(7), parameter :: dzeta_a = [1.d0, .5d0, .25d0, .2d0, &
!       2.d0, 4.d0, 5.d0]
  real(8) :: dxi, deta, dzeta, dxi_inv, deta_inv, dzeta_inv, dV_inv
!  real(8), parameter :: dxi   = 1.d0
!  real(8), parameter :: deta  = 1.d0
!  real(8), parameter :: dzeta = 1.d0
!  real(8), parameter :: dxi_inv   = 1.d0/dxi
!  real(8), parameter :: deta_inv  = 1.d0/deta
!  real(8), parameter :: dzeta_inv = 1.d0/dzeta
!  real(8), parameter :: dV_inv = dxi_inv*deta_inv*dzeta_inv
!  integer :: update_type = 1 ! 1 = FV, 2 = HUI3D

  interface primtocons
     module procedure primtoconsarray
     module procedure primtoconspoint
  end interface primtocons

  interface constoprim
     module procedure constoprimarray
     module procedure constoprimpoint
  end interface constoprim

contains
!  integer elemental function gt0(x)
!    implicit none
!    real(8), intent(in) :: x
!    gt0 = ishft( int(sign(1.d0,x) + 1) , -1 )
!  end function gt0

  ! Computes specific energy, given the array [ p, rho, u, v, w ]
  real(8) function energy_func(in)
    implicit none
    real(8), dimension(:), intent(in) :: in
    energy_func = 0.5d0*(in(3)**2+in(4)**2+in(5)**2) + in(1)/(in(2)*gamma2)
  end function energy_func

  subroutine primtoconsarray(main)
    ! Assumes the structure of prim(:) is :
    !   1   2   3  4  5
    ! [ p, rho, u, v, w ]-pressure, mass density, cartesian velocity components
    ! J is the Jacobian
    ! Returns the structure of cons(:) :
    !     1       2         3         4       5
```

```
  ! [ rho J, rho J u , rho J v , rho J w , J e ]
  !      e is energy, defined for the ideal gas law by :
  !      .5*rho*(u^2+v^2+w^2) + p/(gamma-1)
  real(8), dimension(:,:,:,:), intent(inout) :: main
  integer, dimension(4) :: en_shape
  real(8), allocatable, dimension(:,:,:) :: energy
  en_shape = shape(main)
  allocate(energy(en_shape(2), en_shape(3), en_shape(4)))
  energy = main(21,:,:,:)*( main(1,:,:,:)*gamma1 + 0.5d0*main(2,:,:,:)&
       *( main(3,:,:,:)**2 + main(4,:,:,:)**2 + main(5,:,:,:)**2 ) )
  main(1,:,:,:) = main(2,:,:,:)*main(21,:,:,:)
  main(2,:,:,:) = main(1,:,:,:)*main(3,:,:,:)
  main(3,:,:,:) = main(1,:,:,:)*main(4,:,:,:)
  main(4,:,:,:) = main(1,:,:,:)*main(5,:,:,:)
  main(5,:,:,:) = energy
  deallocate(energy)
end subroutine primtoconsarray

subroutine primtoconspoint(main)
  ! Assumes the structure of prim(:) is :
  !   1   2   3  4  5
  ! [ p, rho, u, v, w ] - pressure, mass density, cartesian velocity components
  ! J is the Jacobian
  ! Returns the structure of cons(:) :
  !     1       2         3         4       5
  ! [ rho J, rho J u , rho J v , rho J w , J e ]
  !      e is energy, defined for the ideal gas law by :
  !      .5*rho*(u^2+v^2+w^2) + p/(gamma-1)
  real(8), dimension(:), intent(inout) :: main
  real(8) :: energy, J
  J = Jacobian(main(6:14))
  energy = J*( main(1)*gamma1 + 0.5d0*main(2)&
       *( main(3)**2 + main(4)**2 + main(5)**2 ) )
  main(1) = main(2)*J
  main(2) = main(1)*main(3)
  main(3) = main(1)*main(4)
  main(4) = main(1)*main(5)
  main(5) = energy
end subroutine primtoconspoint

subroutine constoprimarray(main)
  ! Returns the structure of prim(:) :
  !   1   2   3  4  5
  ! [ p, rho, u, v, w ]-pressure, mass density, cartesian velocity components
  ! J is the Jacobian
  ! Assume the structure of cons(:) is :
  !     1       2         3         4       5
```

```
  ! [ rho J, rho J u , rho J v , rho J w , J e ]
  !      e is energy, defined for the ideal gas law by :
  !      .5*rho*(u^2+v^2+w^2) + p/(gamma-1)
  real(8), dimension(:,:,:,:), intent(inout) :: main
  integer, dimension(4) :: p_shape
  real(8), allocatable, dimension(:,:,:) :: temp1, temp2, p
  p_shape = [size(main,1),size(main,2),size(main,3),size(main,4)]
  allocate( temp1(p_shape(2), p_shape(3), p_shape(4)) )
  allocate( temp2(p_shape(2), p_shape(3), p_shape(4)) )
  allocate(     p(p_shape(2), p_shape(3), p_shape(4)) )
  temp1 = 1.d0/main(21,:,:,:)
  temp2 = 1.d0/main(1,:,:,:)
  p = gamma2*temp1*( main(5,:,:,:) - .5d0*temp2&
       *( main(2,:,:,:)**2 + main(3,:,:,:)**2 + main(4,:,:,:)**2 )&
       )
  main(5,:,:,:) = main(4,:,:,:)*temp2
  main(4,:,:,:) = main(3,:,:,:)*temp2
  main(3,:,:,:) = main(2,:,:,:)*temp2
  main(2,:,:,:) = main(1,:,:,:)*temp1
  main(1,:,:,:) = p
  deallocate( temp1, temp2, p )
end subroutine constoprimarray

subroutine constoprimpoint(main)
  ! Returns the structure of prim(:) :
  !   1   2   3  4  5
  ! [ p, rho, u, v, w ]-pressure, mass density, cartesian velocity components
  ! J is the Jacobian
  ! Assume the structure of cons(:) is :
  !     1       2         3         4       5
  ! [ rho J, rho J u , rho J v , rho J w , J e ]
  !      e is energy, defined for the ideal gas law by :
  !      .5*rho*(u^2+v^2+w^2) + p/(gamma-1)
  real(8), dimension(:), intent(inout) :: main
  real(8) :: temp1, temp2, p, J
  J = Jacobian(main(6:14))
  temp1 = 1.d0/J
  temp2 = 1.d0/main(1)
  p = gamma2*temp1*( main(5) - .5d0*temp2&
       *( main(2)**2 + main(3)**2 + main(4)**2 )&
       )
  main(5) = main(4)*temp2
  main(4) = main(3)*temp2
  main(3) = main(2)*temp2
  main(2) = main(1)*temp1
  main(1) = p
end subroutine constoprimpoint
```

```
  function invnorm3(in)
    real(8), dimension(3), intent(in) :: in
    real(8) :: invnorm3
    invnorm3 = 1.d0/sqrt( in(1)**2 + in(2)**2 + in(3)**2 )
  end function invnorm3

  subroutine grid_coords(grad, normal, tangential1, tangential2)
! Compute an orthonormal coordinate system given an initial,
! unnormalized vector.
    real(8), dimension(:), intent(in) :: grad
    real(8), dimension(3), intent(out) :: normal, tangential1, tangential2
    real(8) :: temp1, temp3
    real(8), dimension(3) :: temp2, temp4
    temp1 = invnorm3(grad)
    normal = grad*temp1
    if( grad(2)**2 + grad(3)**2 < EPS )then
       tangential1 = (/ 0.d0, 1.d0, 0.d0 /)
       tangential2 = (/ 0.d0, 0.d0, 1.d0 /)
    else
       temp2 = (/ 0.d0, -grad(3), grad(2) /)
       temp3 = invnorm3(temp2)
       temp4 = (/ grad(2)**2 + grad(3)**2, -grad(1)*grad(2), -grad(1)*grad(3) /)
       tangential1 = temp2*temp3
       tangential2 = temp1*temp3*temp4
    end if
  end subroutine grid_coords

  subroutine prim_update_HUI3D(main,dt_out,dt_in,CFL,nx,ny,nz,opts)
    implicit none
    integer, dimension(:), intent(in) :: opts
    real(8), dimension(21,-1*opts(101):nx+opts(101)-1,&
         -1*opts(101):ny+opts(101)-1,-1*opts(101):nz+opts(101)-1),&
         intent(inout) :: main
!f2py intent(in,out) :: main
    real(8), dimension(21,0:nx-1,0:ny-1,0:nz-1) :: main_temp
    real(8), intent(out) :: dt_out
    real(8), intent(in), optional :: dt_in
    real(8), intent(in), optional :: CFL
    integer, intent(in) :: nx,ny,nz
    ! Options values are used to activate specific routine options.
    ! - Options(202) controls the spatial order of accuracy.
    ! - Options(203) controls grid motion.
    ! - Options(204) controls the boundary conditions

    integer :: i, j, k, m, n, im, jm, km, ip, jp, kp, case_no
    real(8) :: area, dv_inv, max_wave_speed_temp
```

```
    real(8), dimension(5) :: cons, prim
    real(8), dimension(5) :: left_interface, right_interface
    real(8), dimension(5) :: left_flux, right_flux
    real(8), dimension(3) :: grid_vel, grid_pos
    real(8), dimension(3,3) :: row_ops_mat, vels_transform
    real(8), dimension(9) :: metric, metric_inverse
    real(8), dimension(21) :: temp, center, left, right, geom_avg
    integer, parameter :: splitting_type = 1
    real(8) :: dt, junk
    real(8), dimension(4) :: riemann_middle_states
    real(8), dimension(5) :: riemann_wave_speeds
    integer :: spatial_order
    integer :: grid_motion
    integer :: time_step_scheme
    real(8), dimension(3) :: normal, tan1, tan2
    real(8), parameter :: h=.999d0

    n = opts(2)
    spatial_order = opts(102)
    grid_motion = opts(103)
    time_step_scheme = opts(104)
!!$    select case(time_step_scheme)
!!$    case(1)
!!$       dt_out = dt_in
!!$    case default
!!$       write(*,*) "Invalid time_step_scheme!"
!!$       stop
!!$    end select
    dxi = dxi_a(opts(3))
    deta = deta_a(opts(4))
    dzeta = dzeta_a(opts(5))
    dxi_inv = 1d0/dxi;deta_inv=1d0/deta;dzeta_inv=1d0/dzeta
    dV_inv = dxi_inv*deta_inv*dzeta_inv
    dt = dt_in
    dt_out = dt_in
!    do n = 1, 3
       if( n .eq. 1 )then
          case_no = 1
          area = deta*dzeta
          im =-1; ip = 1; jm = 0; jp = 0; km = 0; kp = 0
       elseif( n .eq. 2 )then
          case_no = 2
          area = dxi*dzeta
          jm =-1; jp = 1; im = 0; ip = 0; km = 0; kp = 0
       elseif( n .eq. 3 )then
          case_no = 3
          area = dxi*deta
```

```
         km =-1; kp = 1; im = 0; ip = 0; jm = 0; jp = 0
      else
         write(*,*) "Error in prim_update, invalid value for n"
         stop
      end if

!!$      if(.not.opts(101)==1)stop
!!$      main(:,nx,:,:) = main(:,nx-1,:,:)
!!$      main(:,:,-1,:) = main(:,:,0,:)
!!$      main(:,:,ny,:) = main(:,:,ny-1,:)
!!$      main(:,:,:,-1) = main(:,:,:,0)
!!$      main(:,:,:,nz) = main(:,:,:,nz-1)
!!$      main(15:17,:,:,:) = h*main(3:5,:,:,:)
      do k = 0, nz-1
         do j = 0, ny-1
            do i = 0, nx-1
               center = main(:,i,j,k)
               left = main(:,i+im,j+jm,k+km)

               if(spatial_order .eq. 2.and.(&
                    (n==1.and.i>0.and.i<nx-1).or.&
                    (n==2.and.j>0.and.j<ny-1).or.&
                    (n==3.and.k>0.and.k<nz-1)))&
                    call MUSCL_HUI(main(1:5,i+2*im,j+2*jm,k+2*km),&
                    main(1:5,i+im,j+jm,k+km),main(1:5,i,j,k),&
                    main(1:5,i+ip,j+jp,k+kp),left(1:5),center(1:5))

               metric_inverse = .5d0*(MetricInverse(left(6:14))+&
                    MetricInverse(center(6:14)))
               grid_vel = .5d0*(left(15:17)+center(15:17))

               select case(case_no)
               case(1)
                  normal = metric_inverse(1:9:3)
                  tan1 = metric_inverse(2:9:3)
                  tan2 = metric_inverse(3:9:3)
               case(2)
                  normal = metric_inverse(2:9:3)
                  tan1 = metric_inverse(3:9:3)
                  tan2 = metric_inverse(1:9:3)
               case(3)
                  normal = metric_inverse(3:9:3)
                  tan1 = metric_inverse(1:9:3)
                  tan2 = metric_inverse(2:9:3)
               end select

               vels_transform(1,:) = normal/sqrt(sum(normal**2))
```

```
vels_transform(2,:) = tan1/sqrt(sum(tan1**2))
vels_transform(3,:) = tan2/sqrt(sum(tan2**2))

left(3:5) = matmul(vels_transform,left(3:5))
center(3:5) = matmul(vels_transform,center(3:5))
grid_vel = matmul(vels_transform,grid_vel)
geom_avg = 0d0
geom_avg(6:14) = center(6:14)
geom_avg(15:17) = grid_vel
left(6:21) = geom_avg(6:21); center(6:21) = geom_avg(6:21)
call riemann_solve(left,center,n,1,[0d0],left_interface,&
     max_wave_speed_temp)

vels_transform = MatrixInverse(vels_transform)
left_interface(3:5) = matmul(&
     vels_transform,left_interface(3:5))

center = main(:,i,j,k)
right = main(:,i+ip,j+jp,k+kp)
if(spatial_order .eq. 2.and.(&
     (n==1.and.i>0.and.i<nx-1).or.&
     (n==2.and.j>0.and.j<ny-1).or.&
     (n==3.and.k>0.and.k<nz-1)))&
     call MUSCL_HUI(main(1:5,i+im,j+jm,k+km),&
     main(1:5,i,j,k),main(1:5,i+ip,j+jp,k+kp),&
     main(1:5,i+2*ip,j+2*jp,k+2*kp),center(1:5),right(1:5))

metric = .5d0*(center(6:14)+right(6:14))
metric_inverse = .5d0*(MetricInverse(center(6:14))+&
     MetricInverse(right(6:14)))
grid_vel = .5d0*(center(15:17)+right(15:17))

select case(case_no)
case(1)
   normal = metric_inverse(1:9:3)
   tan1 = metric_inverse(2:9:3)
   tan2 = metric_inverse(3:9:3)
case(2)
   normal = metric_inverse(2:9:3)
   tan1 = metric_inverse(3:9:3)
   tan2 = metric_inverse(1:9:3)
case(3)
   normal = metric_inverse(3:9:3)
   tan1 = metric_inverse(1:9:3)
   tan2 = metric_inverse(2:9:3)
end select
```

```
   vels_transform(1,:) = normal/sqrt(sum(normal**2))
   vels_transform(2,:) = tan1/sqrt(sum(tan1**2))
   vels_transform(3,:) = tan2/sqrt(sum(tan2**2))

   right(3:5) = matmul(vels_transform,right(3:5))
   center(3:5) = matmul(vels_transform,center(3:5))
   grid_vel = matmul(vels_transform,grid_vel)
   geom_avg = 0d0
   geom_avg(6:14) = center(6:14)
   geom_avg(15:17) = grid_vel
   right(6:21) = geom_avg(6:21); center(6:21) = geom_avg(6:21)
   call riemann_solve(center,right,n,1,[0d0],right_interface,&
        max_wave_speed_temp)

   vels_transform = MatrixInverse(vels_transform)

   right_interface(3:5)=matmul(&
        vels_transform,right_interface(3:5))

   left = main(:,i+im,j+jm,k+km)
   center = main(:,i,j,k)
   right = main(:,i+ip,j+jp,k+kp)


   if(grid_motion .eq. 1)then
      if(n==1)then
         center(6:8) = center(6:8) + h*dt*area*dv_inv*&
              (right_interface(3:5)-left_interface(3:5))
      else if(n==2)then
         center(9:11) = center(9:11) + h*dt*area*dv_inv*&
              (right_interface(3:5)-left_interface(3:5))
      else if(n==3)then
         center(12:14) = center(12:14) + h*dt*area*dv_inv*&
              (right_interface(3:5)-left_interface(3:5))
      end if
   end if

   left_flux  = flux( left_interface,center,n)
   right_flux = flux(right_interface,center,n)
   call primtocons(center)
   center(1:5) = center(1:5) - dt*area*dv_inv*&
        (right_flux-left_flux)
   call constoprim(center)
   center(21) = Jacobian(center(6:14))

   main_temp(:,i,j,k) = center
end do
```

```
          end do
        end do
        main(:,0:nx-1,0:ny-1,0:nz-1) = main_temp
        if(n==3)&
             main(18:20,:,:,:) = main(18:20,:,:,:) + dt*main(15:17,:,:,:)

!       main(15:17,0:nx-1,0:ny-1,0:nz-1) = main(3:5,0:nx-1,0:ny-1,0:nz-1)*.0d0
!       main(18:20,0:nx-1,0:ny-1,0:nz-1) = main(18:20,0:nx-1,0:ny-1,0:nz-1)&
!            *dt*area*dv_inv*main(15:17,0:nx-1,0:ny-1,0:nz-1)
!    end do
  end subroutine prim_update_HUI3D

  subroutine prim_update_FV(main,dt_out,dt_in,CFL,nx,ny,nz,opts)
! Advance the solution using the integral form of the equations
! This subroutine assumes that main is the full array of primitive variables.
! main must also include the boundary cell values. That is, main contains a
! 0-index and an n + 1 index containing the contents prescribed by the
! boundary conditions.
    implicit none
    integer, dimension(:), intent(in) :: opts
!!$    type(prim_update_FV_options), intent(in) :: opts
!!$    real(8), dimension(21,&
!!$         -1*opts%ghost_pts:nx+opts%ghost_pts-1,&
!!$         -1*opts%ghost_pts:ny+opts%ghost_pts-1,&
!!$         -1*opts%ghost_pts:nz+opts%ghost_pts-1),intent(inout) :: main
    real(8), dimension(21,-1*opts(101):nx+opts(101)-1,&
         -1*opts(101):ny+opts(101)-1,-1*opts(101):nz+opts(101)-1),&
         intent(inout) :: main
!f2py intent(in,out) :: main
    real(8), intent(out) :: dt_out
    real(8), intent(in), optional :: dt_in
    real(8), intent(in), optional :: CFL
    integer, intent(in) :: nx,ny,nz
    integer :: spatial_order
    integer :: grid_motion
    integer :: time_step_scheme
    integer, dimension(2) :: splitting
    integer :: i, j, k
!    real(8), dimension(14,3,nx+1,ny+1,nz+1) :: fluxes
    real(8), dimension(14,0:nx,0:ny-1,0:nz-1) :: fluxx
    real(8), dimension(14,0:nx-1,0:ny,0:nz-1) :: fluxy
    real(8), dimension(14,0:nx-1,0:ny-1,0:nz) :: fluxz
    real(8), dimension(14,nx,ny,nz) :: temp
    real(8) :: max_wave_speed, dt, max_dt_grid
    real(8), dimension(3) :: gradXi, gradEta, gradZeta
    real(8), dimension(3) :: GradX, GradY, GradZ
    real(8), dimension(9) :: geom_avg
```

```
    real(8), dimension(3,3) :: dXidX, dXdXi
    real(8), dimension(5) :: interface_vars
    real(8), dimension(21) :: StateL, StateR
    real(8), dimension(14) :: junk
    splitting = opts(1:2)
    spatial_order = opts(102)
    grid_motion = opts(103)
    time_step_scheme = opts(104)
    ! Set module values for dxi, deta, dzeta, based on opts.
    dxi = dxi_a(opts(3))
    deta = deta_a(opts(4))
    dzeta = dzeta_a(opts(5))
    dxi_inv = 1.d0/dxi; deta_inv = 1.d0/deta; dzeta_inv = 1.d0/dzeta
    dV_inv = dxi_inv*deta_inv*dzeta_inv

  ! Riemann_solve expects the left and right states to express velocities in
  ! grid-oriented components: normal, tangential, tangential.
    fluxx = 0d0; fluxy = 0d0; fluxz = 0d0
!! Grid motion has been moved to grid_motion_driver.f90
!    select case(grid_motion)
!    case(0)
!       main(15:17,:,:,:) = 0d0
!    case(1)
!       main(15:17,:,:,:) = .999d0*main(3:5,:,:,:)
!    case(2)
!       main(15:17,:,:,:) = .5
!    case default
!       write(*,*) "Invalid grid motion specification!"
!       stop
!    end select
    if(splitting(1).eq. 1 .or.(splitting(1).eq. 2 .and.splitting(2).eq. 2 ))&
         then
       do k = 0, nz-1
          do j = 0, ny-1
             do i = 0, nx
                StateL = main(:,i-1,j,k)
                StateR = main(:,i,j,k)
                if(spatial_order .eq. 2 .and. i > 0 .and. i < nx)&
                     call MUSCL_HUI(main(1:5,i-2,j,k),main(1:5,i-1,j,k),&
                     main(1:5,i,j,k),main(1:5,i+1,j,k),StateL(1:5),StateR(1:5))
                call compute_fluxes_FV(StateL,StateR,fluxx(:,i,j,k),1,&
                     max_wave_speed,dt=dt,dV_in=[dxi,deta,dzeta])
             end do
          end do
       end do
    end if
    if(splitting(1).eq. 1 .or.(splitting(1).eq. 2 .and.splitting(2).eq. 2 ))&
```

```
          then
         do k = 0, nz-1
            do j = 0, ny
               do i = 0, nx-1
                  StateL = main(:,i,j-1,k)
                  StateR = main(:,i,j,k)
                  if(spatial_order .eq. 2 .and. j > 0 .and. j < ny)&
                       call MUSCL_HUI(main(1:5,i,j-2,k),main(1:5,i,j-1,k),&
                       main(1:5,i,j,k),main(1:5,i,j+1,k),StateL(1:5),StateR(1:5))
                  call compute_fluxes_FV(StateL,StateR,fluxy(:,i,j,k),2,&
                       max_wave_speed,dt=dt,dV_in=[dxi,deta,dzeta])
               end do
            end do
         end do
      end if
      if(splitting(1).eq. 1 .or.(splitting(1).eq. 2 .and.splitting(2).eq. 3))then
         do k = 0, nz
            do j = 0, ny-1
               do i = 0, nx-1
                  StateL = main(:,i,j,k-1)
                  StateR = main(:,i,j,k)
                  if(spatial_order .eq. 2 .and. k > 0 .and. k < nz)&
                       call MUSCL_HUI(main(:,i,j,k-2),main(:,i,j,k-1),&
                       main(:,i,j,k),main(:,i,j,k+1),StateL,StateR)
                  call compute_fluxes_FV(StateL,StateR,fluxz(:,i,j,k),3,&
                       max_wave_speed,dt=dt,dV_in=[dxi,deta,dzeta])
               end do
            end do
         end do
      end if
      max_wave_speed = max(max_wave_speed,EPS)
      select case(time_step_scheme)
      case(0)
         write(*,*) "Using given timestep", time_step_scheme
         dt = dt_in
      case(1)
!        dt = min(CFL/max_wave_speed,dt_in)
         dt = minval([dxi,deta,dzeta])/max_wave_speed*CFL
      case default
         write(*,*) "Error in Godunov prim_update, invalid time step flag!"
         stop
      end select
      dt_out = dt
!     write(*,*) "Got here -- prim_update"
      call primtocons(main(:,0:nx-1,0:ny-1,0:nz-1))
      main(1:14,0:nx-1,0:ny-1,0:nz-1) = main(1:14,0:nx-1,0:ny-1,0:nz-1) - (&
           (fluxx(:,1:nx,:,:)-fluxx(:,0:nx-1,:,:))*deta*dzeta + &
```

```
         (fluxy(:,:,1:ny,:)-fluxy(:,:,0:ny-1,:))*dxi*dzeta + &
         (fluxz(:,:,:,1:nz)-fluxz(:,:,:,0:nz-1))*dxi*deta &
         )*dt*dV_inv
! Update extra variables
    do k = 0, nz-1
       do j = 0, ny-1
          do i = 0, nx-1
             main(21,i,j,k) = Jacobian(main(6:14,i,j,k))
          end do
       end do
    end do
    call constoprim(main(:,0:nx-1,0:ny-1,0:nz-1))
! Update grid position
    main(18:20,0:nx-1,0:ny-1,0:nz-1) = main(18:20,0:nx-1,0:ny-1,0:nz-1) + &
         main(15:17,0:nx-1,0:ny-1,0:nz-1)*dt
!!$! Update extra variables
!!$  do k = 0, nz-1
!!$     do j = 0, ny-1
!!$        do i = 0, nx-1
!!$           main(21,i,j,k) = Jacobian(main(6:14,i,j,k))
!!$        end do
!!$     end do
!!$  end do
  end subroutine prim_update_FV

  subroutine compute_fluxes_FV(inL, inR, flux_vec, case_no,&
      max_wave_speed,dt,dV_in,debug_flag)
    implicit none
    real(8), dimension(21), intent(in) :: inL, inR
    real(8), dimension(:), intent(out) :: flux_vec
    integer, intent(in) :: case_no
    real(8), intent(inout) :: max_wave_speed
    real(8), intent(in) :: dt
    real(8), dimension(3),intent(in) :: dV_in
    logical, intent(in), optional :: debug_flag

    real(8), dimension(21) :: StateL, StateR
    real(8), dimension(21) :: geom_avg
    real(8) :: dA, dV_inv
    real(8), dimension(5) :: interface_vars
    real(8), dimension(3) :: GradXi, GradEta, GradZeta
    real(8), dimension(3) :: GradX, GradY, GradZ
    real(8), dimension(3,3) :: dX_dXi_u, dXi_dX_u
    real(8) :: temp_wave_speed
    integer, dimension(3,3) :: row_ops_mat
    real(8), dimension(9) :: metric, metric_inverse
    real(8), dimension(3,3) :: vels_transform
```

```
    real(8), dimension(3) :: grid_vel
    integer :: m
    real(8), dimension(9) :: metric_inv_l, metric_inv_r
    real(8), dimension(3) :: normal, tan1, tan2
    StateL = inL
    StateR = inR
    geom_avg = .5d0*(inL+inR)
    flux_vec = 0.d0
    dV_inv = 1.d0/(product(dV_in))
!    row_ops_mat = row_ops_mat_func(case_no)
    metric_inverse = MetricInverse(inL(6:14))+MetricInverse(inR(6:14))
    select case(case_no)
    case(1)
       normal = metric_inverse(1:9:3)
       tan1 = metric_inverse(2:9:3)
       tan2 = metric_inverse(3:9:3)
    case(2)
       normal = metric_inverse(2:9:3)
       tan1 = metric_inverse(3:9:3)
       tan2 = metric_inverse(1:9:3)
    case(3)
       normal = metric_inverse(3:9:3)
       tan1 = metric_inverse(1:9:3)
       tan2 = metric_inverse(2:9:3)
    end select

    vels_transform(1,:) = normal/sqrt(sum(normal**2))
    vels_transform(2,:) = tan1/sqrt(sum(tan1**2))
    vels_transform(3,:) = tan2/sqrt(sum(tan2**2))
!    vels_transform = transpose(vels_transform)

    StateL(3:5) = matmul(vels_transform,inL(3:5))
    StateR(3:5) = matmul(vels_transform,inR(3:5))
    StateL(15:17) = matmul(vels_transform,inL(15:17))
    StateR(15:17) = matmul(vels_transform,inR(15:17))

    call riemann_solve(StateL,StateR,case_no,1,[0d0],interface_vars,&
         temp_wave_speed)

    vels_transform = MatrixInverse(vels_transform)
!!$    do m = 1, 3
!!$       vels_transform(m,:) = vels_transform(m,:)&
!!$            /sqrt(sum(vels_transform(m,:)**2))
!!$    end do
    interface_vars(3:5) = matmul(vels_transform, interface_vars(3:5))
!    geom_avg(15:17) = .999d0*interface_vars(3:5)
    flux_vec(1:5) = flux(interface_vars,geom_avg,case_no)
```

```
    select case(case_no)
    case(1)
       flux_vec(6:8) = -geom_avg(15:17)
       dA = dV_in(2)*dV_in(3)
    case(2)
       flux_vec(9:11) = -geom_avg(15:17)
       dA = dV_in(1)*dV_in(3)
    case(3)
       flux_vec(12:14) = -geom_avg(15:17)
       dA = dV_in(1)*dV_in(2)
    case default
       write(*,*) "Invalid case_no in compute_fluxes -- case_no = ",case_no
       stop
    end select
!    flux_vec(6:14)=0d0
    max_wave_speed = max(max_wave_speed,temp_wave_speed)
  end subroutine compute_fluxes_FV

  function row_ops_mat_func(case_no)
    implicit none
    integer, dimension(3,3) :: row_ops_mat_func, row_ops_mat
    integer, intent(in) :: case_no
        select case(case_no)
    case(1)
       row_ops_mat = reshape([1,0,0,0,1,0,0,0,1],[3,3])
    case(2)
       row_ops_mat = reshape([0,1,0,1,0,0,0,0,1],[3,3])
    case(3)
       row_ops_mat = reshape([0,0,1,0,1,0,1,0,0],[3,3])
    case default
       write(*,*) "Invalid case_no in compute_fluxes -- case_no = ",case_no
       stop
    end select
    row_ops_mat_func = row_ops_mat
  end function row_ops_mat_func

  function flux(in,geom_avg,case_no)
    implicit none
    real(8), dimension(:), intent(in) :: in, geom_avg
    integer, intent(in) :: case_no
    real(8), dimension(5) :: flux
    real(8) :: D, grads(3,3), grad(3), J, Jinv
    J = Jacobian(geom_avg(6:14))
    Jinv = 1.d0/J
!    call ComputationalGrads(geom_avg(6:14),J,grads(:,1),grads(:,2),grads(:,3))
    grads = MetrictoMatrix(MetricInverse(geom_avg(6:14)))
    grad = grads(case_no,:)
```

```
    D = dot_product(in(3:5)-geom_avg(15:17),grad)
!sum(([ in(3), in(4), in(5) ] - &
!         [ geom_avg(15), geom_avg(16), geom_avg(17) ])*grad)
    flux = [&
         in(2)*J*D, &
         in(2)*J*D*in(3) + J*grad(1)*in(1), &
         in(2)*J*D*in(4) + J*grad(2)*in(1), &
         in(2)*J*D*in(5) + J*grad(3)*in(1), &
         in(2)*J*D*energy_func(in(1:5)) + J*sum(grad*in(3:5))*in(1) ]
  end function flux

  subroutine MUSCL_HUI(leftleft, left, right, rightright, outleft, outright)
    implicit none
    real(8), dimension(5), intent(in) :: leftleft, left, right, rightright
    real(8), dimension(5), intent(out) :: outleft, outright
    integer :: n

    do n = 1, 5
       outleft(n)  =  left(n) + 5d-1*( left(n) - leftleft(n))*minmod(&
            (right(n)-left(n))/( left(n) - leftleft(n)))
       outright(n) = right(n) - 5d-1*(rightright(n)-right(n))*minmod(&
            (right(n)-left(n))/(rightright(n)-right(n)))
    end do
    outleft  =  left + 5d-1*( left - leftleft)*&
         minmod((right-left)/( left - leftleft))
    outright = right - 5d-1*(rightright-right)*&
         minmod((right-left)/(rightright-right))

  contains
    elemental function minmod(in)
      implicit none
      real(8), intent(in) :: in
      real(8) :: minmod
      minmod = max(0d0,min(1d0,in))
    end function minmod
  end subroutine MUSCL_HUI

  subroutine FreeExitConditions(main,nx,ny,nz)
    implicit none
    real(8), dimension(21,nx,ny,nz) :: main
    integer, intent(in) :: nx, ny, nz

!!$    nx = size(main,2)
!!$    ny = size(main,3)
!!$    nz = size(main,4)
    main(:, 1,:,:) = main(:,   2,:,:)
    main(:,nx,:,:) = main(:,nx-1,:,:)
```

```
    main(:,:, 1,:) = main(:,:,   2,:)
    main(:,:,ny,:) = main(:,:,ny-1,:)
    main(:,:,:, 1) = main(:,:,:,   2)
    main(:,:,:,nz) = main(:,:,:,nz-1)
  end subroutine FreeExitConditions

end module Godunov
```

C.1.6 GeneralUtilitiesTester.f90

This module contains test routines for the `GeneralUtilities` module.

```
module GeneralUtilitiesTester
  use GeneralUtilities
  real(8), dimension(:), allocatable :: lstsqx, lstsqy
contains
  integer function GUErrorReader(in)
    implicit none
    integer, intent(in) :: in
    write(*,*) " GeneralUtilities.f90 diagnostic:"
    select case(in)
    case(0)
       write(*,*) "   All tests passed"
    case(1)
       write(*,*) "   Combined test of metrictomatrix and metricinverse failed"
    case(2)
       write(*,*) "   TwoDGradient does not converge at the expected rate"
    case(3)
       write(*,*) "   MetrictoMatrix does not transform correctly"
    case(4)
       write(*,*) "   MatrixInverse does not work properly"
    case default
       write(*,*) "   Unexpected error code"
    end select
    GUErrorReader = 0
  end function GUErrorReader

  integer function GUTest()
    ! Test the following routines:
    !   function MetricInverse(Metric)
    !   function MetrictoMatrix(Metric)
    !   subroutine ComputationalGrads(metric,jac,grad_xi,grad_eta,grad_zeta)
    !   function Jacobian(in)
    !   function GradstoMatrix(Grad1,Grad2,Grad3)
```

```
!   subroutine TwoDGradient(in,dx,dy,nx,ny,gradx,grady)

! Returns an integer error code.
! 0: All tests passed
! 1: Combined test of metrictomatrix and metricinverse failed
! 2: TwoDGradient does not converge at the expected rate
! 3: metrictomatrix does not transform correctly.
! 4: NormalizedMetric does not work properly
implicit none
real(8), dimension(9) :: metric
real(8), dimension(3,3) :: matrix
real(8), dimension(0:3) :: p1, p2, p3
integer :: out, i, j, k, n, nx, ny
real(8), dimension(400,400) :: test, grad_test_x, grad_test_y
real(8), dimension( 50, 50) :: num_test_1x, num_test_1y
real(8), dimension(100,100) :: num_test_2x, num_test_2y
real(8), dimension(200,200) :: num_test_3x, num_test_3y
real(8), dimension(400,400) :: num_test_4x, num_test_4y
real(8), dimension(4) :: dxes, rmserrors
real(8), dimension(2) :: fitted_poly
real(8) :: dx, dy
real(8) :: x, y

out = 0
! Try variable ranges on these random numbers to check to handle
! ill-conditioned matrices
call random_number(metric)
call random_number(p1)
call random_number(p2)
call random_number(p3)
matrix=matmul(metrictomatrix(metric),metrictomatrix(metricinverse(metric)))
if(maxval((matrix-reshape([1.,0.,0.,0.,1.,0.,0.,0.,1.],[3,3]))**2)>1.d-14)&
     out = 1
if(sqrt(sum((matmul(metrictomatrix(&
     [1d0,2d0,3d0,4d0,5d0,6d0,7d0,8d0,9d0])&
     ,[1.,2.,3.])-[30.,36.,42.])**2))>1d-13)out = 3
if(sqrt(sum((matmul(metrictomatrix(metric),&
     matrixinverse(metrictomatrix(metric)))-&
     reshape([1,0,0,0,1,0,0,0,1],[3,3]))**2))>1d-14)out = 4

nx = size(test,1)
ny = size(test,2)
dx = 12./nx
dy = 13./ny
do j = 1, ny
   do i = 1, nx
      x = i*dx
```

```
        y = j*dy
        test(i,j) = 0.d0
        grad_test_x(i,j) = 0.d0
        grad_test_y(i,j) = 0.d0
        do n = 0, size(p1) - 1
           test(i,j) = test(i,j) + p1(n)*x**n + p2(n)*y**n
           grad_test_x(i,j) = grad_test_x(i,j) + n*p1(n)*x**(n-1)
           grad_test_y(i,j) = grad_test_y(i,j) + n*p2(n)*y**(n-1)
        end do
     end do
  end do
  call TwoDGradient(test(1:nx:8,1:ny:8),dx*8,dy*8,nx/8,ny/8,&
       num_test_1x,num_test_1y)
  call TwoDGradient(test(1:nx:4,1:ny:4),dx*4,dy*4,nx/4,ny/4,&
       num_test_2x,num_test_2y)
  call TwoDGradient(test(1:nx:2,1:ny:2),dx*2,dy*2,nx/2,ny/2,&
       num_test_3x,num_test_3y)
  call TwoDGradient(test,dx,dy,nx,ny,num_test_4x,num_test_4y)

  dxes = [dx*8,dx*4,dx*2,dx]
  rmserrors = [sqrt(sum((num_test_1x-grad_test_x(1:nx:8,1:ny:8))**2)&
       /size(num_test_1x)),&
       sqrt(sum((num_test_2x-grad_test_x(1:nx:4,1:ny:4))**2)&
       /size(num_test_2x)),&
       sqrt(sum((num_test_3x-grad_test_x(1:nx:2,1:ny:2))**2)&
       /size(num_test_3x)),&
       sqrt(sum((num_test_4x-grad_test_x)**2)/size(num_test_4x))]

  fitted_poly = polyfit(log(dxes),log(rmserrors),1)
  if(fitted_poly(2) < 1.) out = 2
  GUTest = out

end function GUTest

integer function lstsq_init(x,y)
  implicit none
  real(8), dimension(:), intent(in) :: x,y
  integer :: nx
  nx = size(x,1)
  allocate(lstsqx(nx), lstsqy(nx))
  lstsqx = x ; lstsqy = y
  lstsq_init = 0
end function lstsq_init

integer function lstsq_close()
  leastsq_close = 0
  deallocate(lstsqx,lstsqy)
```

```
  end function lstsq_close

  subroutine minpack_function_fitting(xdat,ydat,fcn,x,fvec,fjac,tol,info)
    implicit none
    real(8), dimension(:), intent(in) :: xdat,ydat
    external fcn
    real(8), dimension(:), intent(inout) :: x
!    integer, intent(in) :: m, n, ldfjac
    real(8), dimension(:), intent(out) :: fvec
    real(8), dimension(:,:), intent(out) :: fjac
    real(8), intent(in) :: tol
    integer, intent(out) :: info
    integer, dimension(:), allocatable :: wa, ipvt
    integer :: out, m, n, ldfjac, lwa,iflag

iflag = 2
    m = size(xdat,1)
    n = size(x,1)
    ldfjac = m
    allocate(wa(2*(5*n+m)),ipvt(n))
    lwa = size(wa)
    out = lstsq_init(xdat,ydat)
    call exponential_with_y_offset(m,n,x,fvec,fjac,ldfjac,iflag)
    call lmder1(fcn,m,n,x,fvec,fjac,ldfjac,tol,info,ipvt,wa,lwa)
    out = lstsq_close()
    deallocate(wa,ipvt)
  end subroutine minpack_function_fitting

  subroutine exponential_with_y_offset(m,n,x,fvec,fjac,ldfjac,iflag)
    integer, intent(in) :: m
    integer, intent(in) :: n
    integer, intent(in) :: ldfjac
    real(8), dimension(n), intent(in) :: x
    real(8), dimension(m), intent(out) :: fvec
    real(8), dimension(ldfjac,n), intent(out) :: fjac
    integer, intent(inout) :: iflag
    integer i
    ! If iflag = 1 calculate the functions at x and return
    !     this vector in fvec. Do not alter fjac.
    ! If iflag = 2 calculate the jacobian at x and return
    !     this matrix in fjac. Do not alter fvec.
    ! The value of iflag should not be changed unless the
    !     user wants to terminate execution of the least-
    !     squares fit. In this case, set iflag < 0.
    if(.not.m==size(lstsqx,1))then
       write(*,*) "Error in least-squares fit!!"
       write(*,*) "  Incompatible dimensions passed to fcn!"
```

```
      stop
    end if
    select case(iflag)
    case(1)
      forall (i = 1: m)
        fvec(i) = x(1)*lstsqx(i)**x(2)+x(3)-lstsqy(i)
      end forall
    case(2)
      forall (i = 1: m)
        fjac(i,:) = [lstsqx(i)**x(2),x(1)*lstsqx(i)**x(2)*log(lstsqx(i)),1d0]
      end forall
    end select
  end subroutine exponential_with_y_offset
!!$interface ConvergenceFit
!!$   subroutine 4DConvergenceFit(numerical,exact,nmax,out)
!!$     implicit none
!!$     type(4d_array_pointer), dimension(:), intent(in) :: numerical
!!$     real(8), dimension(:,:,:,:), intent(in) :: exact
!!$     real(8), dimension(:,:,:,:), intent(out) :: out
!!$   end subroutine 4DConvergenceFit
!!$   subroutine 2DConvergenceFit(numerical,exact,nmax,out)
!!$     type(4d_array_pointer), dimension(:), intent(in) :: numerical
!!$     real(8), dimension(:,:,:,:), intent(in) :: exact
!!$     real(8), dimension(:,:), intent(out) :: out
!!$   end subroutine 2DConvergenceFit
!!$   subroutine 1DConvergenceFit(numerical,exact,nmax,out)
!!$     type(4d_array_pointer), dimension(:), intent(in) :: numerical
!!$     real(8), dimension(:,:,:,:), intent(in) :: exact
!!$     real(8), dimension(:), intent(out) :: out
!!$end interface ConvergenceFit
!!$
!!$subroutine 4DConvergenceFit(numerical,exact,nmax,out)
!!$  implicit none
!!$  type(4d_array_pointer), dimension(:), intent(in) :: numerical
!!$  real(8), dimension(:,:,:,:), intent(in) :: numerical, exact
!!$  integer, intent(in) :: nmax
!!$  real(8), dimension(:,:,:,:), intent(out) :: out
!!$  integer :: n
!!$  real(8), dimension(nmax) :: dxes
!!$
!!$
!!$  do n = 1, nmax
!!$
!!$
!!$
!!$end subroutine 4DConvergenceFit
```

```
! From Rosetta Code: rosettacode.org/wiki/PolynomialRegression#Fortran
! Requires LAPACK library
! Verified for single-case linear fit against Mathematica, 27 Aug 2012.
function polyfit(vx, vy, d)
  implicit none
  integer, intent(in)                   :: d
  integer, parameter                    :: dp = selected_real_kind(15, 307)
  real(dp), dimension(d+1)              :: polyfit
  real(dp), dimension(:), intent(in)    :: vx, vy

  real(dp), dimension(:,:), allocatable :: X
  real(dp), dimension(:,:), allocatable :: XT
  real(dp), dimension(:,:), allocatable :: XTX

  integer :: i, j

  integer     :: n, lda, lwork
  integer :: info
  integer, dimension(:), allocatable :: ipiv
  real(dp), dimension(:), allocatable :: work

  n = d+1
  lda = n
  lwork = n

  allocate(ipiv(n))
  allocate(work(lwork))
  allocate(XT(n, size(vx)))
  allocate(X(size(vx), n))
  allocate(XTX(n, n))

  ! prepare the matrix
  do i = 0, d
     do j = 1, size(vx)
        X(j, i+1) = vx(j)**i
     end do
  end do

  XT  = transpose(X)
  XTX = matmul(XT, X)

  ! calls to LAPACK subs DGETRF and DGETRI
  call DGETRF(n, n, XTX, lda, ipiv, info)
  if ( info /= 0 ) then
     print *, "problem"
     return
  end if
```

```
    call DGETRI(n, XTX, lda, ipiv, work, lwork, info)
    if ( info /= 0 ) then
       print *, "problem"
       return
    end if

    polyfit = matmul( matmul(XTX, XT), vy)

    deallocate(ipiv)
    deallocate(work)
    deallocate(X)
    deallocate(XT)
    deallocate(XTX)

  end function

end module GeneralUtilitiesTester
```

C.1.7 Riemann_tester.f90

This module contains a test suite for verifying the performance of `Riemann.f90`.

```
module Riemann_tester
  use Riemann
  real(8), dimension(5) :: riemann_sol
  real(8), dimension(21):: test_geom
  real(8) :: mws_test
  ! Though it's possible to do specific tests of the guessp algorithm,
  ! it seems less-effective to do so, since a bad guess will still converge.
contains
  integer function RieErrorReader(in)
    implicit none
    integer, intent(in) :: in
    select case(in)
    case(0)
       write(*,*) "   All tests passed"
    case(1)
       write(*,*) "   Failure in one of the Riemann Star state checks"
    case default
       write(*,*) "   Unexpected error code"
    end select
  end function RieErrorReader

  integer function RieTester(output_dir)
    implicit none
```

```
character(len=*), intent(in) :: output_dir
real(8) :: U, V, W
real(8), dimension(21) :: test_base
real(8), dimension(21) :: test_1_left
real(8), dimension(21) :: test_1_right
real(8), dimension(21) :: test_2_left
real(8), dimension(21) :: test_2_right
real(8), dimension(21) :: test_3_left
real(8), dimension(21) :: test_3_right
real(8), dimension(21) :: test_4_left
real(8), dimension(21) :: test_4_right
real(8), dimension(21) :: test_5_left
real(8), dimension(21) :: test_5_right
real(8), dimension(4) :: test_1_sol
real(8), dimension(4) :: test_2_sol
real(8), dimension(4) :: test_3_sol
real(8), dimension(4) :: test_4_sol
real(8), dimension(4) :: test_5_sol
real(8), dimension(5) :: test_1_speeds
real(8), dimension(5) :: test_2_speeds
real(8), dimension(5) :: test_3_speeds
real(8), dimension(5) :: test_4_speeds
real(8), dimension(5) :: test_5_speeds
real(8), dimension(21) :: moving_test_1_left
real(8), dimension(21) :: moving_test_1_right
real(8), dimension(21) :: complicated_grid_left
real(8), dimension(21) :: complicated_grid_right
real(8), dimension(4) :: moving_test_1_sol
real(8), dimension(5) :: moving_test_1_speeds
integer, parameter :: nx = 1
real(8), dimension(nx) :: x
real(8), dimension(5,nx) :: out
real(8) :: max_wave_speed
real(8), dimension(4) :: riemann_middle_states
real(8), dimension(5) :: riemann_wave_speeds
integer, parameter :: nxfull = 101
real(8), dimension(nxfull) :: xfull
real(8), dimension(5,nxfull) :: outfull
integer :: n, dir
RieTester = 0

test_base = [0d0,0d0,0d0,0d0,0d0,&
     1d0,0d0,0d0,0d0,1d0,0d0,0d0,0d0,1d0,&
     U,V,W,0d0,0d0,0d0,0d0]
test_1_left = test_base; test_1_right = test_base
test_2_left = test_base; test_2_right = test_base
test_3_left = test_base; test_3_right = test_base
```

```
    test_4_left = test_base; test_4_right = test_base
    test_5_left = test_base; test_5_right = test_base

    test_1_left(1:5)  = [1.d0, 1.d0, 0.d0, 0.d0, 0.d0]
    test_1_right(1:5) = [.1d0, .125d0, 0.d0, 0.d0, 0.d0]
    test_2_left(1:5)  = [.4d0, 1.d0, -2.d0, 0.d0, 0.d0]
    test_2_right(1:5) = [.4d0, 1.d0, 2.d0, 0.d0, 0.d0]
    test_3_left(1:5)  = [1.d3, 1.d0, 0.d0, 0.d0, 0.d0]
    test_3_right(1:5) = [.01d0, 1.d0, 0.d0, 0.d0, 0.d0]
    test_4_left(1:5)  = [.01d0, 1.d0, 0.d0, 0.d0, 0.d0]
    test_4_right(1:5) = [1.d2, 1.d0, 0.d0, 0.d0, 0.d0]
    test_5_left(1:5)  = [460.894d0, 5.99924d0, 19.5975d0, 0d0, 0d0]
    test_5_right(1:5) = [46.095d0, 5.99242d0, -6.19633d0, 0d0, 0d0]
    test_1_sol = [.30313d0, .92745d0, .42632d0, .26557d0]
    test_1_speeds = [-1.18322d0,-0.0702745d0,0.92745d0,0d0,1.75216d0]
    test_2_sol = [.00189d0, .00000d0, .02185d0, .02185d0]
    test_2_speeds = [-2.74833d0,-0.347992d0,0d0,0.347992d0,2.74833d0]
    test_3_sol = [460.894d0, 19.5975d0, .57506d0, 5.99924d0]
    test_3_speeds = [-37.4166d0,-13.8997d0,19.5975d0,0d0,23.5175d0]
    test_4_sol = [46.095d0, -6.19633d0, 5.99242d0, .57511d0]
    test_4_speeds = [-7.43747d0,0d0,-6.19633d0,4.39658d0,11.8322d0]
    test_5_sol = [1691.64d0, 8.68975d0, 14.2823d0, 31.0426d0]
    test_5_speeds = [0.789631d0,0d0,8.68975d0,0d0,12.2507d0]
    ! Test riemann_solve against the test problems from Toro. This only tests
    ! whether Pstar, Ustar, DstarL and DstarR are correct, and whether the
    ! computed wave speeds match as computed in Mathematica, and visually
    ! compared (eyeballed) with Toro's plots. This leaves the solutions within
    ! the expansion fan untested.
    test_flag = .true.
!    test_sol = test_1_sol
!      subroutine riemann_solve(left, right, dir, nx, x, out, max_wave_speed,&
!       riemann_middle_states, riemann_wave_speeds)
    do dir = 1, 3
       call riemann_solve(test_1_left,test_1_right,dir,1,[0d0],out,&
            max_wave_speed,riemann_middle_states,riemann_wave_speeds)
       if(.not.(maxval(abs(riemann_middle_states-test_1_sol))<5d-6))&
            RieTester = 1
       if(.not.(maxval(&
            abs((riemann_wave_speeds-test_1_speeds)/test_1_speeds))<5d-5))&
            RieTester = 1
       call riemann_solve(test_2_left,test_2_right,dir,1,[0d0],out,&
            max_wave_speed,riemann_middle_states,riemann_wave_speeds)
       if(.not.(maxval(&
            abs(riemann_middle_states-test_2_sol)/test_2_sol)<5d-3))&
            RieTester = 1
       if(.not.(maxval(abs(riemann_wave_speeds-test_2_speeds))<5d-4))&
            RieTester = 1
```

```
   call riemann_solve(test_3_left,test_3_right,dir,1,[0d0],out,&
        max_wave_speed,riemann_middle_states,riemann_wave_speeds)
   if(.not.(maxval(&
        abs(riemann_middle_states-test_3_sol)/test_3_sol)<5d-6))&
        RieTester = 1
   if(.not.(maxval(abs(riemann_wave_speeds-test_3_speeds))<7d-5))&
        RieTester = 1
   call riemann_solve(test_4_left,test_4_right,dir,1,[0d0],out,&
        max_wave_speed,riemann_middle_states,riemann_wave_speeds)
   if(.not.(maxval(&
        abs(riemann_middle_states-test_4_sol)/test_4_sol)<5d-6))&
        RieTester = 1
   if(.not.(maxval(abs(riemann_wave_speeds-test_4_speeds))<5d-5))&
        RieTester = 1
   call riemann_solve(test_5_left,test_5_right,dir,1,[0d0],out,&
        max_wave_speed,riemann_middle_states,riemann_wave_speeds)
   if(.not.(maxval(&
        abs(riemann_middle_states-test_5_sol)/test_5_sol)<5d-6))&
        RieTester = 1
   if(.not.(maxval(abs(riemann_wave_speeds-test_5_speeds))<8d-5))&
        RieTester = 1
end do
! Visual tests are possible to check that the solution within the fan is
!correct. This writes the density to a file that can then be plotted.
do n = 1, nxfull
   xfull(n) = 0d0+1d0/(nxfull-1d0)*(n-1)
end do
call riemann_solve(test_1_left,test_1_right,1,nxfull,(xfull-.5d0)/.15d0,&
     outfull,max_wave_speed,riemann_middle_states,riemann_wave_speeds)
open(unit=9492, file=trim(output_dir)//'riemann_test.dat')
write(9492,*) outfull(2,:)
write(9492,*) xfull
close(9492)

! I need to test this with more complex geometric variables
test_geom = [0d0,0d0,0d0,0d0,0d0,&
     1d0,0d0,0d0,0d0,1d0,0d0,0d0,0d0,1d0,&
     0d0,0d0,0d0,0d0,0d0,0d0,1d0]

complicated_grid_left = test_1_left
complicated_grid_left(6:21) = test_geom(6:21)
complicated_grid_right = test_1_right
complicated_grid_right(6:21) = test_geom(6:21)
do n = 1, nxfull
   xfull(n) = 0d0+1d0/(nxfull-1)*(n-1)
end do
call riemann_solve(complicated_grid_left,complicated_grid_right,1,nxfull,&
```

```
        (xfull-.5d0)/.15d0,outfull,max_wave_speed,riemann_middle_states,&
        riemann_wave_speeds)
    open(unit=9493,file=trim(output_dir)//'complicated_riemann_test.dat')
    write(9493,*)outfull(2,:)
    write(9493,*)xfull
    close(9493)

  end function RieTester
end module Riemann_tester
```

C.1.8 Godunov_tester.f90

This module provides a test suite for evaluating the accuracy of `Godunov.f90`. In particular, this runs full one-dimensional convergence testing on the Godunov method and evaluates the convergence.

```
module Godunov_tester
  ! Test the following routines:
  ! function riemann_solve(left, right, geom_avg,
  !     max_wave_speed, verbose_flag, t_out)
  ! function energy_func(in)
  ! subroutine primtocons(main,nx,ny,nz)
  ! subroutine constoprim(main,nx,ny,nz)
  ! function invnorm3(in)
  ! subroutine grid_coords(grad, normal, tangential1, tangential2)
  ! subroutine prim_update(main,bcextent,dt_in,CFL,nx,ny,nz)
  ! subroutine compute_fluxes(inL, inR, geom_avg, flux_vec, case_no,&
  !      max_wave_speed,dt,dV_in,debug_flag)
  ! function flux(in,geom_avg,case_no)
  use Godunov
  use GeneralUtilitiesTester
  use GodunovDriver
  use TimeAdvancementStuff
contains
  real(8) function norm2(in)
    implicit none
    real(8), dimension(:), intent(in) :: in
    norm2 = sqrt(sum(in**2))
  end function norm2
  integer function GodErrorReader(in)
    integer, intent(in) :: in
    write(*,*)
```

```
  select case(in)
  case(0)
     write(*,*) "   All tests passed"
  case(1)
     write(*,*) "   Primitive-Conservative mutual inverse test failed"
  case(2)
     write(*,*) "   Grid_coords failed to return an orthonormal system"
  case(3)
     write(*,*) "   Flux fails provided test problem"
  case(4)
     write(*,*) "   GodConvergenceTester1D returned unexpectedly
       low convergence rate"
  case default
     write(*,*) "   Unexpected error code"
  end select
  GodErrorReader = 0
end function GodErrorReader

integer function GodConvergenceTester1D(left,right,x0,t_out,dt,nmax,&
     prim_update_options,base_filename)
  implicit none
  real(8), dimension(5), intent(in) :: left, right
  real(8), dimension(5) :: left_init, right_init
  real(8), intent(in) :: x0, t_out, dt
  integer, intent(in) :: nmax
  integer, dimension(:), intent(in) :: prim_update_options
  character(len=*), intent(in), optional :: base_filename
  integer, parameter :: nx = 100
  real(8), dimension(0:10) :: dxes, rmserrors, fvec
  real(8), dimension(0:10,3) :: fjac
  integer :: nxmax
  real(8), dimension(:,:,:,:), allocatable :: Rie_1D_dir
  real(8), dimension(:,:), allocatable :: Rie_1D
  real(8), dimension(:,:), allocatable :: Rie_1D_exact
  integer :: filenum
  integer :: dir, m, n, i, j, k
  real(8), dimension(4) :: riemann_middle_states
  real(8), dimension(5) :: riemann_wave_speeds
  real(8) :: max_wave_speed, t
  real(8), dimension(3) :: fitted_poly, fitted_poly2
  integer :: info
  logical, parameter :: verbose = .true.
  real(8) :: maxvalue
  real(8), dimension(:), allocatable :: exact_x
  real(8), dimension(21) :: left_full, right_full
  integer, dimension(3) :: nxes
  real(8), dimension(3,3) :: row_ops_mat
```

```
      real(8) :: dt_out

      GodConvergenceTester1D = 0
      if(nmax>10)then
         write(*,*) "Error in GodConvergenceTester1D: Too many refinements!"
         stop
      end if
      maxvalue = 0d0
      do dir = 1, 1
         row_ops_mat = row_ops_mat_func(dir)
         left_init = left; left_init(3:5) = matmul(row_ops_mat,left(3:5))
         right_init = right; right_init(3:5) = matmul(row_ops_mat,right(3:5))
         do n = 0, nmax
            nxmax = nx*2**n
            allocate(Rie_1D_exact(5,0:nxmax-1),exact_x(0:nxmax-1),&
                 Rie_1D(21,-1:nxmax))
            select case(dir)
            case(1)
               allocate(Rie_1D_dir(21,-1:nxmax,-1:1,-1:1))
               nxes = [nx*2**n,1,1]
            case(2)
               allocate(Rie_1D_dir(21,-1:1,-1:nxmax,-1:1))
               nxes = [1,nx*2**n,1]
            case(3)
               allocate(Rie_1D_dir(21,-1:1,-1:1,-1:nxmax))
               nxes = [1,1,nx*2**n]
            end select
            call RieInit1D(left_init,right_init,x0,nxmax,dir,[0d0,1d0],Rie_1D_dir)
            t = 0d0
            select case(dir)
            case(1)
               exact_x = (Rie_1D_dir(18,0:nxmax-1,0,0)-x0)&
                    /Rie_1D_dir(6,0:nxmax-1,0,0)/t_out
            case(2)
               exact_x = (Rie_1D_dir(19,0,0:nxmax-1,0)-x0)&
                    /Rie_1D_dir(10,0,0:nxmax-1,0)/t_out
            case(3)
               exact_x = (Rie_1D_dir(20,0,0,0:nxmax-1)-x0)&
                    /Rie_1D_dir(14,0,0,0:nxmax-1)/t_out
            end select
!!$          call write_files_matlab(Rie_1D_dir,t,nxes(1),nxes(2),.true.)
            do
               call FreeExitConditions(Rie_1D_dir,size(Rie_1D_dir,2),&
                    size(Rie_1D_dir,3),size(Rie_1D_dir,4))
               call prim_update(Rie_1D_dir,dt_out,dt,&
                    .7d0,nxes(1),nxes(2),nxes(3),prim_update_options)
               t = t + dt_out
```

```
!!$             call write_files_matlab(Rie_1D_dir(:,:,:,1),&
!!$                  t,nxes(1),nxes(2),.false.)
            if(t .ge. t_out - 1d-13) exit
         end do
         select case(dir)
         case(1)
            Rie_1D = Rie_1D_dir(:,:,0,0)
            dxes(n) = Rie_1D(18,2) - Rie_1D(18,1)
            exact_x = ((Rie_1D(18,0:nxmax-1)-x0)/t_out&
                 -.5d0*(Rie_1D(15,0)+Rie_1D(15,nxmax-1)))&
                 /(.5d0*(Rie_1D(6,0)+Rie_1D(6,nxmax-1)))
         case(2)
            Rie_1D(:,:) = Rie_1D_dir(:,0,:,0)
            Rie_1D(3,:) = Rie_1D(4,:); Rie_1D(4,:) = 0d0;
            dxes(n) = Rie_1D(19,2) - Rie_1D(19,1)
            exact_x = ((Rie_1D(19,0:nxmax-1)-x0)/t_out&
                 -.5d0*(Rie_1D(16,0)+Rie_1D(16,nxmax-1)))/Rie_1D(10,0:nxmax-1)
         case(3)
            Rie_1D(:,:) = Rie_1D_dir(:,0,0,:)
            Rie_1D(3,:) = Rie_1D(5,:); Rie_1D(5,:) = 0d0;
            dxes(n) = Rie_1D(20,2) - Rie_1D(20,1)
            exact_x = ((Rie_1D(20,0:nxmax-1)-x0)/t_out&
                 -.5d0*(Rie_1D(17,0)+Rie_1D(17,nxmax-1)))/Rie_1D(14,0:nxmax-1)
         end select
         left_full = Rie_1D(:,-1)
         right_full = Rie_1D(:,nxmax)
         Rie_1D_exact = 0d0
!!$         write(*,*) " Left_full = ",left_full
!!$         write(*,*) "Right_full = ",right_full
         call riemann_solve(left_full,right_full,dir,nxmax,exact_x,&
              Rie_1D_exact,max_wave_speed,&
              riemann_middle_states,riemann_wave_speeds)
!!$         write(*,*) left_full(1:3)
!!$         write(*,*) right_full(1:3)
!!$         write(*,*) riemann_middle_states
!!$         write(*,*) riemann_wave_speeds
!!$         read(*,*)
         rmserrors(n) = sqrt(sum(((Rie_1D(2,0:nxmax-1)-Rie_1D_exact(2,:))/&
              maxval(abs(Rie_1D_exact(2,:))))**2)/nxmax)
         maxvalue = max(maxvalue,maxval(abs(Rie_1D(2,:))))
         if(present(base_filename))then
            filenum = 92920 + n
            open(unit=filenum,file=base_filename//achar(48+n)&
                 //"_"//achar(87+dir)//".dat")
            do m = 1, 20
               write(filenum,*) Rie_1D(m,0:nxmax-1)
            end do
```

```
            close(filenum)
            open(unit=92919,file=base_filename//achar(48+n)&
                 //"_"//achar(87+dir)//"_exact.dat")
            do m = 1, 5
               write(92919,*) Rie_1D_exact(m,:)
            end do
            close(92919)
         end if
         deallocate(Rie_1D,Rie_1D_exact,exact_x,Rie_1D_dir)
      end do
   fitted_poly(1:2) = polyfit(log(dxes(0:nmax)),log(rmserrors(0:nmax)),1)
   fitted_poly(3) = 0d0
!!$    call minpack_function_fitting(dxes(0:nmax),rmserrors(0:nmax),&
!!$         exponential_with_y_offset,fitted_poly,&
     fvec(0:nmax),fjac(0:nmax,:),1d-3,info)
   call ToroRiemannConvergence(fitted_poly2)
   info = 1
   if(info .eq. 1 .or. info .eq. 2 .or. info .eq. 3)then
      if(verbose)then
         write(*,*) "Simulation converges with order = ",fitted_poly(2)
         write(*,*) "Simulation converges to within ",&
              fitted_poly(3)/maxvalue*100,&
              "% of exact solution."
         write(*,*) "As a comparison, Numerica's hyper_eul converges with ",&
              "order = ", fitted_poly2(2)
         write(*,*) "CLAWPACK converges with order = 0.608111"
         write(*,*) "CLAWPACK converges to within 0.49% of exact solution"
      end if
   else
      write(*,*) "Function fitting returned unexpected value"
      write(*,*) "info = ", info
      write(*,*) "The meanings of various info values are found in lmder1.f"
      stop
   end if
   if(fitted_poly(2) < .5d0 .or. &
        fitted_poly(3)/maxvalue > .5d0)&
        GodConvergenceTester1D = 4
   end do
 end function GodConvergenceTester1D

 function RotateCoords(in,phi,theta)
   implicit none
   real(8), dimension(3), intent(in) :: in
   real(8), intent(in) :: phi, theta
   optional theta
   real(8), dimension(3) :: RotateCoords
```

```
    RotateCoords = matmul(reshape(&
         [cos(phi),sin(phi),0d0,&
         -sin(phi),cos(phi),0d0,&
         0d0,0d0,1d0],[3,3]),in)
    if(present(theta))then
       write(*,*) "Error in RotateCoords, 3-D rotations not yet supported!!!"
       stop
    end if
  end function RotateCoords

!!$  integer function GodConvergenceTester2D(&
!!$       left,right,t_out,dt,nmax,base_filename)
!!$    use GeneralUtilities, only : PI
!!$    use TimeAdvancementStuff
!!$    implicit none
!!$    real(8), dimension(5), intent(in) :: left, right
!!$    real(8), intent(in) :: t_out, dt
!!$    integer, intent(in) :: nmax
!!$    character(len=*), intent(in), optional :: base_filename
!!$    integer, parameter :: nx = 50
!!$    real(8), dimension(0:10) :: dxes, rmserrors, fvec
!!$    real(8), dimension(0:10,3) :: fjac
!!$    integer :: nxmax
!!$    real(8), dimension(:,:,:,:), allocatable :: Rie_2D
!!$    real(8), dimension(:,:,:), allocatable :: Rie_2D_exact
!!$    real(8), parameter :: phi = 0d0!PI*.5d0!25d0/180d0
!!$    real(8), dimension(2) :: xy
!!$    integer :: filenum
!!$    integer :: m, n, i, j, k
!!$    real(8), dimension(4) :: riemann_middle_states
!!$    real(8), dimension(5) :: riemann_wave_speeds
!!$    real(8) :: max_wave_speed, t
!!$    real(8), dimension(3) :: fitted_poly
!!$    integer :: info
!!$    real(8), dimension(3) :: rotated_coords, coords_shift
!!$    real(8) :: dt_out
!!$
!!$    GodConvergenceTester2D = 0
!!$    if(nmax>10)then
!!$       write(*,*) "Error in GodConvergenceTester2D: Too many refinements!"
!!$       stop
!!$    end if
!!$    do n = 0, nmax
!!$       nxmax = nx*2**n
!!$       allocate(Rie_2D(21,-1:nxmax,-1:nxmax,-1:1),&
!!$            Rie_2D_exact(5,0:nxmax-1,0:nxmax-1))
!!$       call RieInit2D(left,right,phi,nxmax,[0d0,1d0],Rie_2D)
```

```
!!$       t = 0d0
!!$!       call write_files_matlab(Rie_2D(:,0:nx-1,0:nx-1,0),t,nx,nx,.true.)
!!$       do
!!$          call prim_update(Rie_2D,dt_out,FreeExitConditions,&
!!$              1,dt,.7d0,nxmax,nxmax,1,[2,0])
!!$          t = t + dt
!!$          write(*,*) "t = ",t
!!$          if(t .ge. t_out) exit
!!$       end do
!!$       Rie_2D_exact = 0d0
!!$       do i = 0, nxmax-1
!!$          do j = 0, nxmax-1
!!$             coords_shift = .5d0*(maxval(maxval(Rie_2D(18:20,:,:,0),3),2)-&
!!$                  minval(minval(Rie_2D(18:20,:,:,0),3),2))
!!$             rotated_coords = RotateCoords((Rie_2D(18:20,i,j,0)-coords_shift)&
!!$                  /Rie_2D(6:14:4,i,j,0),phi)/t_out
!!$             call riemann_solve(Rie_2D(:,-1,-1,-1),&
!!$                  Rie_2D(:,nxmax,nxmax,-1),1,1,&
!!$                  rotated_coords,Rie_2D_exact(:,i,j),max_wave_speed)
!!$          end do
!!$       end do
!!$       dxes(n) = Rie_2D(18,1,0,0)-Rie_2D(18,0,0,0)
!!$       rmserrors(n) = sqrt(sum(((Rie_2D(2,0:nxmax-1,0:nxmax-1,0)&
!!$            -Rie_2D_exact(2,:,:))/&
!!$            maxval(abs(Rie_2D_exact(2,:,:))))**2)/nxmax**2)
!!$       if(present(base_filename))then
!!$          filenum = 93020 + n
!!$          open(unit=filenum,file=base_filename//achar(48+n)//".dat")
!!$          do m = 1, 20
!!$             write(filenum,*) Rie_2D(m,:,:,0)
!!$          end do
!!$          close(filenum)
!!$          if(n .eq. nmax)then
!!$             open(unit=93019,file=base_filename//achar(48+n)//"_exact.dat")
!!$             do m = 1, 5
!!$                write(93019,*) Rie_2D_exact(m,:,:)
!!$             end do
!!$             close(93019)
!!$          end if
!!$       end if
!!$       deallocate(Rie_2D,Rie_2D_exact)
!!$    end do
!!$    write(*,*) "dxes = ",dxes(0:nmax)
!!$    write(*,*) "rmserrors = ",rmserrors(0:nmax)
!!$    fitted_poly(1:2) = polyfit(log(dxes(0:nmax)),log(rmserrors(0:nmax)),1)
!!$    fitted_poly(3) = 0d0
!!$    call minpack_function_fitting(dxes(0:nmax),rmserrors(0:nmax),&
```

```
!!$         exponential_with_y_offset,fitted_poly,fvec(0:nmax),&
!!$              fjac(0:nmax,:),1d-3,info)
!!$    write(*,*) "2-D data"
!!$    if(info .eq. 1)then
!!$       write(*,*) "Simulation converges with order = ",fitted_poly(2)
!!$       write(*,*) "Simulation converges to within ",fitted_poly(3)*100,&
!!$            "% of exact solution."
!!$       write(*,*) "CLAWPACK converges with order = 0.608111"
!!$       write(*,*) "CLAWPACK converges to within 0.49% of exact solution"
!!$       stop
!!$    else
!!$       write(*,*) "Function fitting returned unexpected value"
!!$       write(*,*) "info = ", info
!!$       write(*,*) "The meanings of various info values are found in lmder1.f"
!!$       stop
!!$    end if
!!$    if(fitted_poly(2) < 1d0) GodConvergenceTester2D = 3
!!$
!!$  end function GodConvergenceTester2D

  integer function GodTester(output_dir)
    use TimeAdvancementStuff
    implicit none
    character(len=*), intent(in) :: output_dir
    real(8), dimension(21,3,4,3) :: main
    integer :: i, j, nx, ny
    real(8), dimension(5) :: riemann_test
    real(8), dimension(21) :: geom_test
    real(8), dimension(21,3,3,3) :: constoprimtest1, constoprimtest2
    real(8), dimension(21) :: constoprimtest3
    real(8), dimension(5) :: left_test, right_test
    real(8) :: max_wave_speed
    real(8), dimension(3) :: grad, norm, tan1, tan2
    integer, dimension(1000) :: prim_update_options
    real(8) :: test1, test2, test3, dt
    real(8), dimension(21), parameter :: flux_seed = [0.270991,0.0613936,&
         0.748627,0.24336,0.374409,-0.423353,0.323678,-0.333788,-0.53982,&
         -0.991525,0.75888,-0.921888,-0.631486,0.801117,0.634729,-0.750043,&
         0.412858,-0.0734701,-0.0563167,0.97121,1.]
    real(8), dimension(5), parameter :: flux_test_x = [-0.0171431,-0.0982248,&
         -0.0765652,-0.161747,-0.335386]
    real(8), dimension(5), parameter :: flux_test_y = [-0.0384556,-0.0419378,&
         -0.184655,-0.167707,-0.548871]
    real(8), dimension(5), parameter :: flux_test_z = [0.0285833,-0.00172451,&
         0.142847,0.171804,0.402354]
    real(8), dimension(21,62,102,3) :: riemann_test_array
    real(8), dimension(21,102,3,3) :: riemann_test_array_1d
```

```
real(8), dimension(5) :: left, right
integer :: n

GodTester = 0
! Check that primtocons and constoprim are mutual inverses
! Are there invertibility problems from non-physical variables,
! such as negative pressures, etc? Random_number only returns
! positive numbers, I suppose.
call random_number(constoprimtest1)
constoprimtest2 = constoprimtest1
call constoprim(constoprimtest1)
call primtocons(constoprimtest1)
if(sqrt(sum((constoprimtest1-constoprimtest2)**2)&
     /size(constoprimtest1))>EPS) GodTester = 1
! Also check against a Matlab solution.
constoprimtest3 = [&
     0.081125768865785,0.929385970968730,0.775712678608402,&
     0.486791632403172,0.435858588580919,0.446783749429806,&
     0.306349472016557,0.508508655381127,0.510771564172110,&
     0.817627708322262,0.794831416883453,0.644318130193692,&
     0.378609382660268,0.811580458282477,0.532825588799455,&
     0.350727103576883,0.939001561999887,0.875942811492984,&
     0.550156342898422,0.622475086001227,0.022367194643555]
call primtocons(constoprimtest3)
constoprimtest3=[&
     0.020787756911647,0.016125326596194,0.010119306121021,&
     0.009060522387274,0.015228249823238,0.446783749429806,&
     0.306349472016557,0.508508655381127,0.510771564172110,&
     0.817627708322262,0.794831416883453,0.644318130193692,&
     0.378609382660268,0.811580458282477,0.532825588799455,&
     0.350727103576883,0.939001561999887,0.875942811492984,&
     0.550156342898422,0.622475086001227,0.022367194643555]&
     -constoprimtest3
if(any(constoprimtest3**2>1d-15)) GodTester = 1
! Ensure that the coordinate system returned from grid_coords
! is orthonormal, and that norm is parallel to grad.

call random_number(grad)
grad = grad - .5d0
call grid_coords(grad, norm, tan1, tan2)
if(maxval(abs([norm2(norm), norm2(tan1), norm2(tan2)]-1.d0))>EPS)then
   GodTester = 2
end if
if(dot_product(grad/norm2(grad),norm)-1.d0>EPS) GodTester = 2
if(abs(dot_product(norm,tan1))>EPS) GodTester = 2
if(abs(dot_product(norm,tan2))>EPS) GodTester = 2
if(abs(dot_product(tan1,tan2))>EPS) GodTester = 2
```

```
    ! Test flux routine against Mathematica-generated test problem & solution
    ! Mathematica only returns to a set precision.
    if(sqrt(sum((flux(flux_seed(1:5),flux_seed,1)&
         - flux_test_x)**2)/5.)>EPSs)then
       GodTester = 3
    end if
    if(sqrt(sum((flux(flux_seed(1:5),flux_seed,2)&
         - flux_test_y)**2)/5.)>EPSs)then
       GodTester = 3
    end if
    if(sqrt(sum((flux(flux_seed(1:5),flux_seed,3)&
         - flux_test_z)**2)/5.)>EPSs)then
       GodTester = 3
    end if

    ! It is important to also test prim_update, though the only known way
    ! to do this is via specific convergence testing. For now, that testing
    ! must be approved manually.
    ! subroutine prim_update(main,bcextent,dt_in,CFL,nx,ny,nz)

    prim_update_options = 0
    prim_update_options(1) = 1
    prim_update_options(3:5) = [1,1,1]
    prim_update_options(101) = 1
    prim_update_options(102) = 1
    prim_update_options(103) = 0
    prim_update_options(104) = 1
    left(1:5)  = [1d0, 1d0, .75d0, 0d0, 0d0]
    right(1:5) = [.1d0, .125d0, 0d0, 0d0, 0d0]
    GodTester = GodConvergenceTester1D(&
         left,right,.3d0,.18d0,1d-4,3,&
         prim_update_options,&
         base_filename=trim(output_dir)//"1D_Rie_Test_1")
!!$    left(1:5)  = [.4d0, 1d0,-2d0, 0d0, 0d0]
!!$    right(1:5) = [.4d0, 1d0, 2d0, 0d0, 0d0]
!!$    GodTester = GodConvergenceTester1D(
!!$         left,right,.5d0,.12d0,1d-4,3,&
!!$         prim_update_options,&
!!$         base_filename=trim(output_dir)//"1D_Rie_Test_2")
!!$    left(1:5)  = [1000d0, 1d0, 0d0, 0d0, 0d0]
!!$    right(1:5) = [.01d0, 1d0, 0d0, 0d0, 0d0]
!!$    GodTester = GodConvergenceTester1D(&
!!$         left,right,.5d0,.01d0,1d-4,3,&
!!$         prim_update_options,&
!!$         base_filename=trim(output_dir)//"1D_Rie_Test_3")
!!$    left(1:5)  = [460.894d0, 5.99924d0, 19.5975d0, 0d0, 0d0]
```

```
!!$    right(1:5) = [46.095d0, 5.99242d0,-6.19633d0, 0d0, 0d0]
!!$    GodTester = GodConvergenceTester1D(&
!!$         left,right,.4d0,.03d0,1d-4,3,&
!!$         prim_update_options,&
!!$         base_filename=trim(output_dir)//"1D_Rie_Test_4")
!!$    left(1:5)  = [1000d0, 1d0,-19.59745d0, 0d0, 0d0]
!!$    right(1:5) = [.01d0, 1d0,-19.59745d0, 0d0, 0d0]
!!$    GodTester = GodConvergenceTester1D(&
!!$         left,right,.8d0,.01d0,1d-4,3,&
!!$         prim_update_options,&
!!$         base_filename=trim(output_dir)//"1D_Rie_Test_5")
  end function GodTester

  subroutine NormShockInit(nx,main)
    implicit none
    integer, intent(in) :: nx
    real(8), dimension(21,-1:nx,-1:1,-1:1), intent(out) :: main
    real(8), dimension(3) :: upstream, downstream
    real(8), dimension(2) :: xrange
    integer :: half_nx, i,j,k

    xrange = [main(18,0,0,0),main(18,nx-1,0,0)]
    upstream = [1d0,1d0,2d0*sqrt(1.4d0)]
    call NormalShockRelations(upstream,1.4d0,downstream)

    if(mod(nx,2).eq.0)then
       half_nx = nx/2
    else
       write(*,*) "Warning: NormShockInit called with an odd value for nx."
       write(*,*) "         Press any key to continue, but the normal shock"
       write(*,*) "         will be slightly offset from center."
       read(*,*)
       half_nx = nx/2
       write(*,*) "         The true halfway point is ",&
            .5*(xrange(2)-xrange(1))
       write(*,*) "         The initial shock location is ",&
            .5*(xrange(2)-xrange(1))-.5*(xrange(2)-xrange(1))/(nx-1)
    end if
    main = 0d0
    forall (i=-1:half_nx-1, j=-1:1, k=-1:1)
       main(1:3,i,j,k) = upstream
    end forall
    forall (i=half_nx:nx, j=-1:1, k=-1:1)
       main(1:3,i,j,k) = downstream
    end forall
    main(6,:,:,:) = 1d0
    main(10,:,:,:) = main(6,:,:,:)
```

```
    main(14,:,:,:) = main(6,:,:,:)
    forall (i=-1:nx,j=-1:1,k=-1:1)
       main(18,i,j,k) = real(i,8)
       main(21,i,j,k) = Jacobian(main(6:14,i,j,k))
    end forall
    main(18,:,:,:) = main(18,:,:,:) - .5d0*maxval(main(18,:,:,:))
  end subroutine NormShockInit

  subroutine NormalShockRelations(in,gamma,out)
    implicit none
    real(8), dimension(3), intent(in) :: in
    real(8), intent(in) :: gamma
    real(8), dimension(3), intent(out) :: out
    real(8) :: a1, M1, p, d, u, a2, M2

    p = in(1); d = in(2); u = in(3)
    a1 = sqrt(gamma*p/d); M1 = u/a1
    M2 = sqrt((1+((gamma-1d0)*.5d0)*M1**2)/(gamma*M1**2-.5d0*(gamma-1d0)))
    out(1) = p*(1d0 + 2d0*gamma/(gamma-1d0)*(M1**2-1d0))
    out(2) = d*((gamma+1d0)*M1**2)/(2d0 + (gamma-1d0)*M1**2)
    a2 = sqrt(gamma*out(1)/out(2))
    out(3) = M2*a2
  end subroutine NormalShockRelations

  subroutine RieInit1D(left,right,x0,nx,dir,xrange,main)
    implicit none
    real(8), dimension(5), intent(in) :: left, right
    real(8), intent(in) :: x0
    integer, intent(in) :: nx, dir
    real(8), dimension(2), intent(in) :: xrange
    real(8), dimension(:,:,:,:), intent(out) :: main
    real(8), dimension(21,-1:nx,-1:1,-1:1) :: main_temp
    integer :: i,j,k,half_nx
    real(8) :: dx
    if(mod(nx*x0,1d0).eq.0)then
       half_nx = int(floor(nx*x0))
    else
       write(*,*) "Warning: RieInit1D called with an odd value for nx."
       write(*,*) "         Press any key to continue, but the Riemann"
       write(*,*) "         problem will be slightly offset."
       read(*,*)
       half_nx = int(nx*x0)
!!$       write(*,*) "         The true halfway point is "&
!!$            ,.5*(xrange(2)-xrange(1))
!!$       write(*,*) "         The Riemann problem is centered at ",&
!!$            .5*(xrange(2)-xrange(1))-.5*(xrange(2)-xrange(1))/(nx-1)
    end if
```

```
    main = 0d0
    main_temp = 0d0
    forall (i=-1:half_nx-1, j=-1:1, k=-1:1)
       main_temp(1:5,i,j,k) = left
    end forall
    forall (i=half_nx:nx, j=-1:1, k=-1:1)
       main_temp(1:5,i,j,k) = right
    end forall
    dx = (xrange(2)-xrange(1))/(nx)
    main_temp(6,:,:,:) = dx
    main_temp(10,:,:,:) = dx
    main_temp(14,:,:,:) = dx
!!$    main_temp(6:14:4,:,:,:) = 1d0
    forall (i=-1:nx, j=-1:1, k=-1:1)
       main_temp(18,i,j,k) = dx*(i+.5) + xrange(1)
       main_temp(21,i,j,k) = Jacobian(main_temp(6:14,i,j,k))
    end forall
    select case(dir)
    case(1)
       main = main_temp
    case(2)
       forall (i=-1:nx,j=-1:1,k=-1:1)
          main(:,k+2,i+2,j+2) = main_temp(:,i,j,k)
       end forall
       main(19,:,:,:) = main(18,:,:,:)
       main(18,:,:,:) = 0d0
    case(3)
       forall (i=-1:nx,j=-1:1,k=-1:1)
          main(:,j+2,k+2,i+2) = main_temp(:,i,j,k)
       end forall
       main(20,:,:,:) = main(18,:,:,:)
       main(18,:,:,:) = 0d0
    case default
       write(*,*) "Error in RieInit1D, invalid dir value!!!"
       stop
    end select
  end subroutine RieInit1D

  subroutine RieInit2D(left,right,phi,nx,xrange,main)
    use GeneralUtilities, only : PI
    implicit none
    real(8), dimension(5) :: left, right
    real(8), intent(in) :: phi
    integer, intent(in) :: nx
    real(8), dimension(2), intent(in) :: xrange
    real(8), dimension(21,-1:nx,-1:nx,-1:1), intent(out) :: main
    integer :: i,j,k,half_nx
```

```
  real(8) :: dx
  real(8), dimension(3,3) :: vels_transform

  vels_transform = reshape(&
       [cos(phi),-sin(phi),0d0,sin(phi),cos(phi),0d0,0d0,0d0,1d0],[3,3])
  main = 0d0
  dx = (xrange(2)-xrange(1))/(nx-1)
  main(6,:,:,:) = dx
  main(10,:,:,:) = dx
  main(14,:,:,:) = main(6,:,:,:)
  left(3:5) = matmul(vels_transform,left(3:5))
  right(3:5) = matmul(vels_transform,right(3:5))
  do i = -1, nx
     do j = -1, nx
        do k = -1, 1
           main(18,i,j,k) = dx*i + xrange(1)
           main(19,i,j,k) = dx*j + xrange(1)
           main(21,i,j,k) = Jacobian(main(6:14,i,j,k))
           if( (main(19,i,j,k)-.5d0*(xrange(2)-xrange(1)))&
                - tan(phi-PI*.5d0)*( main(18,i,j,k)-.5d0&
                *(xrange(2)-xrange(1))) < 0d0)then
              main(1:5,i,j,k) = left
           else
              main(1:5,i,j,k) = right
           end if
        end do
     end do
  end do

end subroutine RieInit2D

subroutine GodRieInit(main)
  implicit none
  real(8), dimension(21,62,102,3), intent(inout) :: main
  integer :: i, j

  main( 1,:,:,:) = 1.d0
  main( 2,:,:,:) = 1.d0
  main( 3,:,:,:) = 2.4d0*sqrt(1.4d0*main(1,:,:,:)/main(2,:,:,:))
  main( 4,:,:,:) = 0.d0
  main( 5,:,:,:) = 0.d0
  main( 6,:,:,:) = 1.d0/99.d0
  main( 7,:,:,:) = 0.d0
  main( 8,:,:,:) = 0.d0
  main( 9,:,:,:) = 0.d0
  main(10,:,:,:) = 1.d0/99.d0
  main(11,:,:,:) = 0.d0
```

```
  main(12,:,:,:) = 0.d0
  main(13,:,:,:) = 0.d0
  main(14,:,:,:) = 1.d0
  main(15,:,:,:) = .25d0*main(3,:,:,:)
  main(16,:,:,:) = 0.d0
  main(17,:,:,:) = 0.d0
  main(18,:,:,:) = 0.d0
  do j = 2, size(main,3)-1
     do i = 2, size(main,2)-1
        main(19,i,j,2) = 1.d0/100.d0*(i-1.5d0)
        main(20,i,j,2) = 0.d0
        main(21,i,j,2) = Jacobian(main(6:14,i,j,2))
     end do
  end do
  main(1,:,size(main,3)/2+1:size(main,3),:) = .25d0
  main(2,:,size(main,3)/2+1:size(main,3),:) = .5d0
  main(3,:,size(main,3)/2+1:size(main,3),:) = 7.d0*&
       sqrt(1.4d0*.25d0/.5d0)
  main(15,:,:,:) = .25d0*main(3,:,:,:)
end subroutine GodRieInit
subroutine ToroRiemannConvergence(out)
  implicit none
  real(8), dimension(3), intent(out) :: out
  real(8), dimension(:,:), allocatable :: num
  real(8), dimension(:,:), allocatable :: exact
  real(8), dimension(:), allocatable :: x
  real(8), dimension(21) :: left, right
  real(8) :: max_wave_speed
  real(8), dimension(4) :: riemann_middle_states
  real(8), dimension(5) :: riemann_wave_speeds
  integer :: n, nx, ierr
  real(8), dimension(0:4) :: rmserrors, dxes, x0, t_out
  real(8), dimension(3) :: fitted_poly
  real(8) :: max_value
  max_value = 0d0
  x0 = [.3d0,.5d0,.5d0,.4d0,.8d0]
  t_out = [.18d0,.12d0,.01d0,.03d0,.01d0]
  x0 = .3d0
  t_out = .2d0
  do n = 0, 4
     nx = 100*2**n
     allocate(exact(5,0:nx-1),x(0:nx-1),num(3,0:nx-1))
     num = 0d0
     call ReadToroDat("datToro/1D_Rie_Test_1"//char(48+n)//".dat",num,x,ierr)
     left = 0d0                          ; right = 0d0
     left(1:3) = num(:,0)                ; right(1:3) = num(:,nx-1)
     left(6) = (x(2)-x(1))               ; right(6) = left(6)
```

```
       left(10) = left(6)                   ; right(10) = left(10)
       left(14) = left(6)                   ; right(14) = left(14)
       left(21) = left(6)*left(10)*left(14) ; right(21) = left(21)
       call riemann_solve(left,right,1,nx,(x-x0(n))/left(6)/t_out(n),&
            exact,max_wave_speed,&
            riemann_middle_states,riemann_wave_speeds)
       open(file="datToro/exact"//char(n+48)//".dat",unit=100)
       write(100,*) exact(2,:)
       close(100)
!!$       write(*,*) exact(2,:) - num(2,:)
!!$       read(*,*)
       rmserrors(n) = sqrt(sum(&
            ((num(2,:)-exact(2,:))/maxval(abs(exact(2,:))))**2)/nx)
       max_value = max(max_value,maxval(abs(num(2,:))))
       dxes(n) = x(2)-x(1)
       deallocate(exact,x,num)
    end do
    fitted_poly(1:2) = polyfit(log(dxes),log(rmserrors),1)
    fitted_poly(3) = 0d0
!!$    call minpack_function_fitting(dxes,rmserrors,exponential_with_y_offset,&
!!$         fitted_poly,fvec,fjac,1d-3,info)
    out = fitted_poly
  end subroutine ToroRiemannConvergence
  subroutine ReadToroDat(file,out,x,ierr)
    implicit none
    integer, parameter :: maxnx = 10000
    character(len=*), intent(in) :: file
    real(8), intent(out), dimension(:,:) :: out
    real(8), intent(out), dimension(:) :: x
    integer, intent(out) :: ierr
    real(8), dimension(5) :: read_buffer
    integer :: iostatus, n
    open(unit=837347,file=file,status="old")
    do n = 1, maxnx
       read(837347,*,IOSTAT=iostatus) read_buffer
       if(iostatus.eq.0)then
          out(1,n) = read_buffer(4)
          out(2,n) = read_buffer(2)
          out(3,n) = read_buffer(3)
          x(n) = read_buffer(1)
       else if(iostatus<0)then
          exit
       else
          write(*,*) "Error reading ",trim(file)," in ReadToroDat!!"
          write(*,*) "n = ", n, "iostatus = ", iostatus
          stop
       end if
```

```
    end do
  end subroutine ReadToroDat
end module Godunov_tester
```

C.1.9 test_program.f90

Running `make check` compiles the *_tester.f90 modules and links them into this program in order to actually run the tests.

```
program test_program
  use generalutilitiestester
  use godunov_tester
  use riemann_tester
  use grid_motion_tester
  use UCS_tester
!  use ode_solvers_tester
  implicit none
  integer :: result, junk
  character(len=32) :: output_dir
  call get_command_argument(1,value=output_dir)
  write(*,*) "General Utilities test:"
  result = GUTest()
  junk = GUErrorReader(result)
  write(*,*) "Riemann tester:"
  result = RieTester(output_dir)
  junk = RieErrorReader(result)
!!$  write(*,*) "Grid Motion Tester:"
!!$  result = grid_motion_test()
!!$  junk = grid_motion_reader(result)
  write(*,*) "Godunov tester:"
  result = GodTester(output_dir)
  junk = GodErrorReader(result)
  write(*,*) "UCS_tester:"
  result = UCS_test_main()

!!$  write(*,*) "ODE_Solvers tester:"
!!$  result = ODEtester()
!!$  junk = ODEErrorReader(result)
  write(*,*) "Done"
end program test_program
```

C.1.10 Godunov_driver.f90

This is the driver routine for `Godunov.f90`.

```
module Godunovdriver
  use Godunov
  interface
     subroutine bc_func(main,nx,ny,nz)
       implicit none
       real(8), dimension(21,nx,ny,nz),intent(inout) :: main
       integer, intent(in) :: nx, ny, nz
     end subroutine bc_func
  end interface
contains
  subroutine prim_update(main,dt_out,dt_in,CFL,nx,ny,nz,options)
    implicit none
    real(8), dimension(:,:,:,:) :: main
    !f2py intent(in,out) :: main
    integer, intent(in) :: nx, ny, nz
    integer, dimension(:), intent(in) :: options
    real(8) :: dt_in, CFL, dt_out
    !f2py intent(out) :: dt_out
    ! Options is an array of integers that controls the behavior of prim_update
    ! and its various subroutines. It is organized as follows.
    ! Elements 1-100 (Fortran numbering) control prim_update itself
    ! Elements 101-200 control prim_update_FV
    ! Elements 201-300 control prim_update_HUI3D
    ! Elements greater than 300 are reserved for future expansion
    select case(options(1))
    case(1)
       call prim_update_FV(main,dt_out,dt_in,CFL,nx,ny,nz,options)
    case(2)
!!$       write(*,*) "Error in prim_update: "
!!$       write(*,*) "-Dimensionally split algorithms not yet implemented!"
!!$       stop
       call prim_update_HUI3D(main,dt_out,dt_in,CFL,nx,ny,nz,options)
    case default
       write(*,*) "Bad update_type value!!!"
       stop
    end select
  end subroutine prim_update
end module Godunovdriver
```