# Introduction

## The state of CFD

## The problems of grid generation

## Structured vs. Unstructured grids

## Why UCS?

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.

# The Unified Coordinates

## The history of UCS

### Hui 2-, 3-D

The unified coordinates were first used to solve time-dependent flows in 19991, 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 20012, 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 later3. 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. %%. Unsteady boundary conditions were also easily modeled with UCS, as in Fig. %%.

### Hui Viscous

UCS has been applied to more than just the Euler equations. An application to the full Navier-Stokes equations4 showed not only that UCS was capable of resolving viscous flows, but also that the system was robust enough to handle complex phenomena like shock-induced boundary-layer separation with recirculation, as shown in Fig. %%, thus answering one of the major concerns about its Lagrangian methodology. Additional models to which UCS has been applied include multi-material flows5, magneto-hydrodynamics6, and gas-kinetic BGK simulations of a freely falling plate7,8.

#### Space-marching Euler

### Avalanche modeling

## The UCS transformation

### Grid velocity

#### Hui’s methods

##### Lagrangian

##### Grid-angle-preserving

##### Jacobian-preserving

##### Skewness-preserving

#### Limitations to Hui’s work

##### Do not follow boundaries exactly

##### Does not pin grid points to singular points

##### Elastic “pressure-based” grid possibilities?

### Compatibility conditions

#### Derivation

#### Typical solution methods

##### Finite difference?

##### Boundary value methods?

##### Impacts on solution stability

#### The full UCS equations

##### Strong conservation form for flow variables and metric components

We here present the full transformation of the Euler equations to Cartesian coordinates, while maintaining Cartesian velocity components. In all of the following, implied sums over greek indices are to be taken over four-dimensional space-time, and sums over latin indices are to be taken over three-dimensional space. In three dimensions, the Euler equations are given by:



We wish to express in the unified coordinate system given by the transformation:



may be written more succinctly as , which naturally leads to , which can be further written as:



where we have defined the following:



Geometric compatibility conditions are required in order for to be well-behaved. In particular, we require that partial derivatives commute:



This can be expanded to read (see also Hui9):



We now transform using :



If we multiply this by the Jacobian and apply the differential product rule, we find:



It can be shown that is identically equal to zero10, eliminating one term. It is likewise possible to show that the final non-conservative term is zero. 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 in terms of vector cross products:



If we rewrite:



It can easily be seen that must be identically zero, by exploiting the anti-symmetry of the Levi-Civita tensor. We are therefore left with the conservative, fully transformed Euler equations in unified coordinates:



describes the behavior of the physical flow quantities in the unified coordinates. We know from , however, that time evolution equations also exist for the grid metric components. This is best handled by appending the time-dependent compatibility conditions to to yield an expanded equation set.

We use to define the total derivative of transformed coordinates:



We are now in a position to write the total transformed Euler equations in the unified coordinate system:



##### Source terms for physical coordinates

##### Free specification of grid velocity, as long as transformation remains well-behaved (J>0?)

### Grid motion control

is dependent on some suitable choice for the grid motion components *U*, *V*, *W*. There is a great deal of freedom available in the choice of these components, provided only that the transformation does not become singular. Some very useful properties can be obtained through judicious choice of grid velocity. In particular, we require that  and  shall be material coordinates. In other words:



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

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, and *U* remains unspecified, which allows further controls to be applied in order to improve grid quality.

Common choices for *U* are: such that grid velocity is proportional to flow velocity; such at the angle between lines of constant  and lines of constant  is preserved (can ensure an orthogonal grid in two dimensions); such that the jacobian of the transformation remains constant in time. Methods of grid control are discussed in more detail in Section %%.

### Solution algorithm

is a much larger system of equations than the Euler equations , and some simplifying approximations must be made in order to compute practical solutions. The first and most essential approximation is to decouple the equations that control the evolution of the grid metric from those that govern the fluid flow. This reduces the problem to that of solving the Euler equations in curvilinear coordinates, along with a simple grid update step, and is referred to as the time-step-eulerian (TSE) approximation.

Additionally, many algorithms for solving the Euler equations are inherently one-dimensional, and must be extended to multidimensional flows in some way. The two most prominent methods are dimensional splitting approximations and finite-volume methods, which are described in more detail in Sec. %%.

#### Time-step-Eulerian

#### Dimension Splitting vs. Finite-Volume methods

#### 1st-order Godunov solver

The Godunov method for solving partial differential equations11 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 **W** would be given by . 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 Toro11.

##### Transformation to grid components

Godunov’s method for solving partial differential equations requires the solution of various one-dimensional Riemann problems, where the initial conditions and discontinuity are given by the interface between two adjoining cells. Therefore, in order to apply the Godunov method to solve, 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  as an orthonormal basis where  is normal to the cell interface, then we may write



Under this transformation, the tangential derivatives vanish at cell interfaces, and the remaining unsteady, one-dimensional equations become12:



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 unless  is constant across the cell boundary. It is therefore necessary to choose some suitable average to be applied at the cell boundary:



Both flow and grid velocity components are transformed using . Once transformed, the normal grid velocity component must also be averaged as before: . This averaged value is used both in the solution of the Riemann problem and in the computation of intercellular fluxes.

##### Eigensystem of governing equations

In order to compute the solution to the one-dimensional Riemann problem defined by and the initial conditions:



The first step is to compute the derivatives:





The eigenvalues can be computed as the solutions of the equation:



The corresponding eigenvectors are:



From these we may compute both the generalized Riemann invariants and the Rankine-Hugoniot relations.

##### Riemann invariants and rarefaction wave relations

The generalized Riemann invariants are relations that hold true across smooth waves, and are given by the relations11:



We begin with density:



Then normal velocity:



The tangential velocities are simple:



The rarefaction head and tail speeds are given by the eigenvalues :



It is finally necessary to account for pressure changes for points within the rarefaction wave. The slope of a characteristic for a rarefaction wave is:



can be combined with to solve for :



##### The Rankine-Hugoniot conditions and shock wave relations

The generalized Riemann invariants do not hold across discontinuous waves such as shocks. For such waves, the Rankine-Hugoniot conditions must be used11:



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:



##### Slip lines

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

##### 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 %%. 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 - and , it is possible to define a function of pressure representing the jump in normal velocity across the central slip line:

 u

 represents the normal velocity computed across the  wave given a central pressure  . 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.

#### Spatial accuracy and boundary interpolation 2nd-order MUSCL update

The Godunov method is inherently first-order accurate, but it is possible to boost the order of spatial accuracy using MUSCL interpolation 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:



#### Godunov dimensional splitting

##### Hui1 uses a dimensional splitting technique to solve the multidimensional Euler equations. The algorithm is as follows:

* For each coordinate direction *n*:
  + For each cell *i*, *j*, *k*:
    - For each interface *+*, *-*:
      * Apply MUSCL reconstruction using .
      * Transform the flow and grid velocity to normal and tangential components using: 
      * Solve the Riemann problems as described in section %% to find the flow variables at each interface: 
      * Transform interface velocity back to Cartesian components using: 
    - Update coordinate-appropriate grid metric components: 
    - Compute interface fluxes using interface flow variables and central metric variables, e.g.:
    - Compute new conserved quantities  using and updated metric components
    - Update conserved variables using (e.g.):
    - Convert updated conserved variables to updated primitive variables.

#### Finite-Volume

The dimensionally split algorithm above 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. Unfortunately, the FV approach is computationally unstable for two- and three-dimensional problems under the Godunov method without special treatment.

* Compute all cell interface fluxes:
  + For each interface:
    - Perform MUSCL interpolation if applicable
    - Transform velocity vectors to normal & 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:
* Compute conserved variables
* Update conserved variables using computed flux vectors
* Compute updated primitive variables

### Specification of grid motion

#### Grid-angle-preserving

##### In two-dimensional flow, it is possible to force the grid to move in such a way that the angle of intersection between lines of constant %xi and lines of constant %eta is preserved. That is:



We take for two-dimensional flow, and becomes:



If we define we can write which lead to We rewrite :



Which leads to:



At this point, we apply the UCS compatibility conditions:



We must also here choose whether to solve the equation for grid velocity components directly, or to solve for grid velocity magnitude. We follow the magnitude approach here, and define:



and similarly for %eta. This allows us to write as



can then be solved using a number of techniques.

#### Jacobian-preserving

##### For three-dimensional flows, grid-angle-preserving is no longer as useful. Preserving the jacobian of the grid transformation is different technique that can be used for 3-D flows. We first define some useful variables:



We also require that %eta and %zeta be material coordinates:



We finally rewrite the compatibility conditions:

.



The equation for preservation of grid jacobian can then be written as:



Applying the compatibility conditions yields



We can differentiate to find:



These can be combined to yield:



#### Elastic boundary-conforming?

# Verification of Codes – Almost copy/paste from SciTECH

## Introduction

The use of numerical simulations and mathematical models to predict the behavior of real-world systems continues to grow in almost every current field of study, beginning with the sciences and engineering, but now reaching into economics, finance, and the social sciences as well. While these models and simulations have been instrumental in advancing many fields, they have also led to their own share of disasters. As the demands of the day become increasingly complex, it is becoming increasingly critical that researchers know precisely the limits and limitations of both their models and the computer programs or codes that they use to solve them. The limitations of numerical simulation arise naturally out of the physical and numerical approximations that are used to solve them, and so it is important to both understand and quantify the errors introduced by these approximations in order to use simulations effectively in the design, prediction, and analysis of real-world systems. The National Research Council13 identified the three topics of Verification, Validation, and Uncertainty Quantification as crucial in quantifying this understanding, and specifically identified the development of manufactured solutions for complex mathematical models as a key tool for verification.

In order to better understand the errors inherent in a numerical simulation, Roache14 classified these errors as:

1. Errors that are a result of modeling approximations, such as incompressibility, continuity, 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, or round-off error.

The study and quantification of modeling errors for a given application is generally referred to as validation. The study and quantification of the remaining numerical and programming errors is known as verification. Verification is further subdivided into two parts, code or software verification, and solution verification. Code verification shows that the code or software does solve the mathematical model correctly within some domain of inputs, while solution verification estimates the expected error for a solution to a specific problem. This paper is primarily concerned with code verification.

Despite the overwhelming need for reliable verification of numerical simulations and scientific codes, the majority of such codes are verified principally by comparison with other, unverified codes, or by comparison with experimental data, both of which can hide important errors and behaviors. With the development of the powerful, general, and thorough Method of Manufactured Solutions (MMS) at the end of the twentieth century14–17, it became possible to convincingly verify any scientific code, provided the choice of manufactured solution was sufficiently smooth. The Integrative Method of Manufactured Solutions (IMMS) described herein extends MMS to codes that solve systems of equations that permit discontinuous solutions, through the use of the integral form of the equations which such weak solutions satisfy.

## Background

The most rigorous approach to verification commonly taken by developers today is to assemble a suite of exact solutions against which the code is compared17. If these tests do not show evidence to the contrary, then the code is assumed to be correct. Unfortunately, this process has no clear ending point, and therefore many such test suites are neither thorough nor complete, typically using simple exact solutions that do not fully exercise the governing equations, nor the boundary and initial conditions, nor the various alternative code paths that might be encountered in real-world applications of the code.

Recent developments in code verification have favored an alternative approach to code verification which uses the more precise “code order verification” defined by Knupp17, “The process by which one verifies the theoretical order-of-accuracy of the algorithm employed by the code to solve its governing equations.” This is done by measuring the rate at which the solution returned by the code approach an exact solution as the computational grid is refined, and comparing the results to the theoretical rate that would be expected. This kind of testing is remarkably sensitive14, but it relies on the availability of exact solutions that fully exercise the code as described above, so it is best paired with the method of manufactured solutions (MMS).

The basic idea of MMS is simple. Rather than finding a solution that fits an equation, one chooses a solution and derives the equation. A general description of the problem of mathematical modeling is to find solutions to the possibly vector-valued equation:



This is a very difficult problem to solve for general forms of *f* and *u*. A much easier problem is to simply choose some suitable form for *u(x)*, and derive a new equation:



where *Sm* is defined as the residual that results from the evaluation of *f* with the manufactured solution. It is also commonly referred to as a manufactured source term. Provided that the code being verified is capable of handling the additional manufactured source term, running the code with that source should yield the original, exact solution. Most importantly, the choice of manufactured solution is almost completely arbitrary, and so it can be chosen to fully exercise all important aspects of the code. Knupp17 gives the following guidelines for constructing useful 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 solution should be general enough that it exercises every term in the governing equation.
3. The solution should have a sufficient number of nontrivial derivatives.
4. Solution derivatives should be bounded by a small constant.
5. The manufactured solution 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

The demonstration of theoretical order of accuracy for such a manufactured solution constitutes code order verification, as described above.

The Center for Predictive Engineering and Computational Sciences (PECOS) at the University of Texas Austin has developed and maintains the Manufactured Analytical Solution Abstraction (MASA) library18, which provides a generalized interface for manufactured solution and source term evaluation.

The method of manufactured solutions works exceptionally well for most differential equations, because the differential source terms can be computed easily and analytically, often through the use of computer algebra systems. Unfortunately, many differential equations of interest do not admit continuous solutions for certain physically admissible and boundary conditions, and this has been a problem in MMS for some time15–19. Currently, these situations are handled either in a two-step process which verifies the code against smooth manufactured solutions and then against discontinuous exact solutions, or else a manufactured solution is chosen that is not discontinuous but merely steep enough to appear as a discontinuity when discretized. These approaches are problematic, because they must choose between a comprehensive test of all solution features in a single test problem and a faithful representation of the manufactured solution by the code being verified. The integrative approach to MMS resolves both of these problems cleanly.

The principal difficulty with applying MMS to discontinuous solutions is that such solutions are not differentiable everywhere, so they cannot satisfy the differential equations for which MMS was developed. These are solutions only in the weak sense, which is to say that they satisfy only the integral equations that result from integrating the original equations over some suitable computational volume, much as finite-volume and finite-element codes already do. This provides a clear path to reliable code verification for discontinuous solutions, using the integral form of the equations, and one implementation of this technique is found in the SENSEI code developed by Roy et al.20

## Manufacturing Integral Solutions

Consider a general form of integral equation that would be obtained by integrating a differential balance law over some volume:



In such an equation, the change in some quantity *u* contained within a volume *V* is given by the amount created/destroyed by sources/sinks *S* throughout the volume, combined with the flux *F* through the boundary of the volume . If there are no sources, then the total amount of *u* is conserved, and such equations are called conservation laws.

This form of integral balance law is extremely common, and is the basis behind finite-volume and finite-element codes. For a code such as this, an integral form of MMS is more suitable for verification, as it more closely reflects the mathematics internal to the code. This is especially true for equations that may admit discontinuous solutions, since the manufacture of such weak solutions for integral equations is both simple and straightforward.

Given some suitable choice for *u*(*x*), the following can be computed:



This yields a manufactured source term, analogous to that found in MMS, though integrated over the volume *V*. This term can then either be differentiated to yield *Sm*, or else it can be used in its integrated form directly as a source term in a scientific code, which should then yield the initial, potentially discontinuous, manufactured solution. This constitutes the Integrative Method of Manufactured Solutions (IMMS).

Given the complex form of the flux and source terms for most equations of interest, it is difficult to choose a manufactured solution that is complex enough to fully exercise a code, and yet yield fluxes and sources that are analytically integrable. Numerical integration tools have no such difficulties, and the errors they introduce are normally many orders of magnitude less than those found in a typical numerical simulation.

To summarize, applying IMMS to the verification of codes consists of three steps:

1. Choose a suitable manufactured solution.
2. Evaluate the necessary integrals in order to compute the integrated manufactured source term.
3. Incorporate the manufactured source into the code to be verified.

## Implementation

The Busemann Advanced Concepts Lab at the University of Colorado Boulder has developed prototype software that implements IMMS for balance laws as in BACL-IMMS is written primarily in the Python programming language, and leverages both Sympy21 and SciPy22 in order to ease the process of computing manufactured source terms for any user-defined equation for any solution form. The principal components of BACL-IMMS are:

1. An n-dimensional integration tool nquad that recursively applies QUADPACK23 integration routines to evaluate multidimensional integrals with discontinuous integrands.
2. A specialized integration module that wraps nquad for the integration of list quantities.
3. A base equation class that provides functions for the simple integration of surface integrals, volume integrals, and complete balance integrals.
4. Subclasses that define fluxes and sources for different equation sets.

### The nquad Integration Tool

The SciPy22 library for the Python programming language provides access to the one-dimensional integration routines drawn from the QUADPACK23 Fortran library. Although these routines are well-established and robust, IMMS requires the evaluation of integrals over multiple variables, which neither QUADPACK nor SciPy supported with enough generality. nquad provides the structure and interface required to evaluate these integrals by recursive application of the 1-D QUADPACK routines. This recursive approach to numerical integration is less efficient than true multidimensional integrators, but it provides one key advantage for computing discontinuities.

In IMMS, many of the multivariate integrands will be discontinuous at some point, but the locations of these discontinuities are known at the time the manufactured solution is defined. These locations can be automatically defined and passed to QUADPACK, thus allowing the automatic subdivision of the integration region, precisely along discontinuities. nquad allows the SciPy library to accomplish this with general n-dimensional integrals, using whichever QUADPACK routines are most appropriate, and it has been included in SciPy as of version 0.13.

### List Integration & Vector Integration

The integration module and base equation class used in BACL-IMMS greatly simplify the process of computing manufactured source terms through integration. The computation of list integrals is quite common in scientific computing, as in  , where *E* could be a list of such disparate quantities as mass density, momentum, and energy. The notation suggest the integration over a single region of an arbitrary number of functions, and BACL-IMMS allows this operation. The concepts of surface and volume integration are also intrinsic to IMMS, which relies on the generalized Stokes’ Theorem to convert volume integrals of undefined functions to surface integrals of bounded, discontinuous functions. In particular, the computation of manufactured source terms as in is automated in BACL-IMMS, so that the integrated manufactured source term can be intelligently computed based only on the chosen manufactured solution, the governing equations, and the optional locations of any discontinuities. It is in this module that discontinuities are automatically computed from the functional input form provided by the user: . This expression is provided symbolically using Sympy, and automatically converted into functions that return the location of the discontinuity in a form QUADPACK can process, e.g. .

### Equation-specific Subclasses

In order to serve a wide range of users, BACL-IMMS was designed from the beginning to be expandable. It provides all of the tools needed to perform a balance integration independent of any specific equation set. A simple interface allows the computation of manufactured source terms for any system of equations that can be put in integrative balance-law form as in , given only a symbolic definition of the flux and source terms that make up that system of equations. The linear heat equation has already been implemented, as have the three-dimensional Euler equations in the unified coordinate system9. In order to implement a new equation set, it is only necessary to create a new subclass that defines manufactured solutions, flux functions, and source functions, if any, and everything else is largely automatic.

## Testing

IMMS has many potential advantages in the world of software verification. Unlike traditional MMS, it can easily handle discontinuous solutions, and it is a more faithful representation of the underlying mathematical model for finite-volume and finite-element codes. However, it must be demonstrated that IMMS is accurate and reliable enough to be used in code verification, which requires its own testing. The obvious first test for a program of this type is to use it on known, exact solutions, for which *Sm* is identically zero. The results of integrating the fluxes and sources with such an exact solution should ideally be zero, within the limits of machine precision, but such accuracy is strongly dependent on the numerical integration routine used. For practical purposes, the error in numerical integration should be several orders of magnitude less significant than the expected error in the numerical solution.

### The Linear Heat Equation

The linear heat equation with constant material properties is given by:



A few exact solutions to this equation are given by Polyanin24:



In , *A*, *B*, *c*, and  are arbitrary constants, *n* is a positive integer, and *a* is the thermal diffusivity given by . The spatial coordinate  is given by , corresponding to a three-dimensional rotation of *x*. The value of *Sm* can be computed for various values of the solution parameters, and the resulting values provide a test of the numerical integration routine being used. A randomized set of solution parameters was used to generate a variety of exact solutions, and the value of the manufactured source terms was computed. Fig. %% shows the results of this computation. Each line represents a single test, and the value it takes at each discrete point on the x-axis corresponds to the value of that solution parameter for that test. The exact combinations of simulation parameters used for each test are randomly sampled from a discrete set of representative values. Results for BACL-IMMS are shown in Fig. %%. In the case of the linear heat equation, the error in the integration is many orders of magnitude below the value of the exact solution, which shows that IMMS maintains more than the level of accuracy needed for the verification of numerical codes.

### The Unsteady Euler Equations

The linear heat equation provides a good first test for IMMS, because it is well understood, with a number of exact solutions for comparison, but a more strenuous test case is needed in order to more fully explore the possibilities of IMMS for software verification. One area where IMMS would be very useful is in the field of computational fluid dynamics. The Euler equations for inviscid flow of a perfect gas are a system of five coupled, nonlinear equations that are known to admit solutions with multiple types of discontinuities, even when given smooth initial data. As such, they provide a much more strenuous test of BACL-IMMS’s integration routines.

As with the heat equation, BACL-IMMS is applied to various exact solutions for which the manufactured source terms should be identically zero. Four unsteady, one-dimensional Riemann problems are selected for this test, and they are defined by the initial states given in Table 1. These Riemann problems are also rotated in space to increase dimensionality, though for the sake of simplicity, only one rotation is used, and the resulting unsteady problem is two-dimensional.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Table 1 Initial states of Riemann problems | | | | | | |
|  | Left State | | | Right State | | |
| Problem # | P | **ρ** | *u* | P | **ρ** | *u* |
| 0 | 1.0 | 1.0 | 0.0 | 0.1 | 0.125 | 0.0 |
| 1 | 0.4 | 1.0 | -2.0 | 0.4 | 1.0 | 2.0 |
| 2 | 1000.0 | 1.0 | 0.0 | 0.01 | 1.0 | 0.0 |
| 3 | 0.1 | 1.0 | 0.0 | 100.0 | 1.0 | 0.0 |

The results from numerically computing manufactured source terms for the Euler equations are shown in Fig. %%, and it is immediately obvious that the integration is less precise for this much more complex set of equations, due to the more complex nature of the equations and the discontinuities that must be resolved. The error is small in comparison to both the problems being tested and the expected numerical error inherent in a numerical solution of the Euler equations, nonetheless care should be exercised when selecting numerical integrators, to be sure that they provide acceptable accuracy for verification of the specific equation set.

## Example Verification of Euler equations solver using IMMS

Although comparison with known solutions is important for the development of BACL-IMMS, and IMMS in general, the real purpose of the method is to enable codes to be convincingly verified for the solution of general partial differential equations in the presence of discontinuities. The Euler equations provide just such an example, since there are few exact solutions available, and the equations readily allow for the spontaneous generation of shocks, contact discontinuities, and other problematic features. For such a system, the IMMS is a natural fit.

The Busemann Advanced Concepts Lab has developed BACL-Streamer, an Euler solver as part of its research into design-oriented computational fluid dynamics (CFD)25–27W. BACL-Streamer is a finite-volume code that uses a first-order, exact Godunov method to solve the conservative Euler equations. It also allows for both forms of manufactured solutions, as well as non-conservative elements, by allowing the definition and use of source terms through the use of partial time steps as described in Toro11, where the contributions from the conservative fluxes and the non-conservative sources are computed separately. The advancement of conservative flow variables  as a result of differential source terms is handled by the time-advancement of the system of ordinary differential equations given by:



using the differential equation solvers found in the SciPy library22, while integral source terms are incorporated directly with the numerical fluxes as:



Several different solutions are needed for verification of IMMS, including exact solutions to provide baseline convergence rates, smooth manufactured solutions for comparison with MMS, and a discontinuous solution for IMMS verification. The manufactured solutions chosen are analytically simple, yet provide non-trivial manufactured source terms. The smooth solution is given by:



The discontinuous solution is similar, with a sharp discontinuity:



These solutions are shown in Fig. %%.

The most interesting exact solutions to the Euler equations involve flow discontinuities, and two Riemann problems are chosen to provide a baseline convergence test. The first is Sod’s shock tube problem given as problem #0 in Table %%. The second is a two-dimensional, steady, supersonic Riemann problem given by the boundary condition:



The solution for the two-dimensional, steady problem is shown in Fig. %%.

### Comparison of MMS and IMMS for Smooth Manufactured Solutions

The simulation is initialized with the correct solution and allowed to evolve forward in time. The final error in the numerical pressure is measured, and the RMS average is computed across the length of the domain. This RMS value can be computed at multiple levels of grid refinement in order to obtain a measured convergence rate and plots, as shown in Fig. %%. The convergence rate for IMMS is very close to MMS, and it appears that IMMS performs as well as traditional MMS for this type of problem.

### Computation of Convergence Rates for Discontinuous Solutions, Both Exact and Manufactured

Although the comparison with MMS is encouraging, the intended purpose of IMMS is the verification of codes with discontinuous solutions, which MMS cannot do. Therefore, the convergence of RMS error for the discontinuous manufactured solution in is computed using IMMS. The convergence rate for the discontinuous problem is clearly much lower than for the smooth problem. As a comparison, we also show convergence of two discontinuous, exact solutions. The steady, supersonic Riemann problem given in and shown in Fig. %% is also computed with BACL-Streamer, and its convergence rate measured. To check against another solver, Sod’s problem (test problem # 0 from Table %%), is solved using Toro’s NUMERICA11,28,29, and the convergence again measured. The results from this test are shown in Fig. %%. The drop in convergence rate is seen in all of these cases, and it appears to be due to the presence of discontinuities and the effects of shock smearing. The performance of IMMS is quite consistent with the behavior of the exact solutions, and it can be recommended for further use in code verification.

## Conclusion

The Integrative Method of Manufactured Solutions (IMMS) is a powerful new tool for the verification of scientific codes. For the first time, it is possible to incorporate one-step verification procedures for problems involving shocks, slip lines, material interfaces, and other forms of discontinuous solutions, while simultaneously testing smooth convergence as rigorously as the differential method of manufactured solutions allows. This one-step verification procedure can and should be used in all development codes that admit discontinuous solutions in order to ensure proper numerical simulation of mathematical models, to discover subtle coding mistakes, and to increase the trustworthiness of any codes thus verified. BACL-IMMS has been developed and is publically available27 to simplify this implementation, and the principles are readily adaptable to custom code implementations as well. The introduction of IMMS will accelerate the widespread adoption of strong code verification standards in computing throughout science.

## Order of convergence verification

### Measure rate of convergence of code to complex, exact, solution

### Compare convergence rate with that expected based on algorithm.

### Highly sensitive to even subtle errors

## Method of Manufactured Solutions

### Exact solutions are rarely complex enough to yield useful verification tests.

### Boundary conditions for exact solutions are typically simplistic.

### MMS begins with a manufactured solution and an analytically computed source term

#### Manufactured solution can be as complex as desired, but must be differentiable.

#### Boundary conditions are arbitrary, based on the value of the solution at the boundaries.

#### Source term is computed using computer-aided algebra (CAS) systems and code generation tools.

## Integrative Method of Manufactured Solutions

### Numerical integration accuracy

### IMMS performance (Roy’s method)

### Multidimensional integration with discontinuities

## Verification of BACL-Streamer & UCS

### Choice of exact & manufactured solutions

### Verification of Euler solver

### Verification of UCS solver under various forms of grid motion

### Verification of BCs?

# Challenges of Moving Grids

Although the unified coordinates provide many benefits in accuracy and grid generation, they are not without their own challenges. The main difficulties lie in connecting the unsteady computational coordinates in which the UCS equations are solved with the boundary conditions, which are defined in physical coordinates.

## A First Cut at Boundary Conditions

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



*refl* represents the vector reflection across the wall boundary, imposing a symmetry condition at the top wall at 0 degrees from horizontal.

For the bottom wall, however, things are more difficult. For example, one might try:

do i = 1, ni

if(cell(i)%x < start\_ramp .or. &

cell(i)%x > end\_ramp)then



else



end if

end do

This will certainly work. However, 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.

## A better method for boundary conditions

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



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 , it is possible to write a relation allowing the computational coordinates of a cell to be determined solely from the array indices:



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 . This may be done as a simple array slice operation, with no conditional evaluation required.

u

## 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 Fig. %%. Unfortunately, with the grid moving at an unpredictable rate, these singular points never align exactly with grid points, which automatically introduces error  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 layer4.

## 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 in Fig. %%. 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.

## 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.

# Code Iterations

## v.1.0

### Basic Fortran

### Variable-size grid handled using dynamic memory allocation and array copying.

### Code outline

## v.2.0

### Advanced Fortran

### Linked-list implementation of variable grid

#### Each node points to each other node

### Individual nodes represented as derived types

### Code outline

## v.3.0

### Code design

#### Space & time looping

#### Mixed-Language Programming

#### Modularity

### Unit tests

#### Direct testing of expected outcome

#### Grid convergence studies

#### “Does routine \_\_X\_\_ do what I want it to do?”

### Algorithmic implementation

#### Dimensional splitting

#### U = h u

##### Lagrangian-esque grid motion

##### Grid-angle-preserving grid motion

### Code outline

# Conclusion