The Unified Coordinates, Verification, and the Method of Manufactured Solutions

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

2014

This thesis entitled:
The Unified Coordinates, Verification, and the Method of Manufactured Solutions
written by C. Nathan Woods
has been approved for the Department of Aerospace Engineering Sciences

---

Ryan P. Starkey

Prof. Rachel Goddard

Ms. Thora Nea

Hi there

Me too!

And last!

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)

The Unified Coordinates, Verification, and the Method of Manufactured Solutions

Thesis directed by Dr. Ryan P. Starkey

Often the abstract will be long enough to require more than one page, in which case the macro "\OnePageChapter" should *not* be used.

But this one isn't, so it should.

Dedication

To my loving wife, for all her patience and support.

Acknowledgements

Here's where you acknowledge folks who helped. But keep it short, i.e., no more than one page, as required by the Grad School Specifications.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

Insert text here

1.1 Grid Generation and Unified Coordinates

Insert text here

1.2 Code Verification

Insert text here

1.3 Numerical Integration

Insert text here.

Chapter 2

The Unified Coordinate System

Insert text here

2.1 Background

Insert text here

2.2 The Euler Equations in Unified Coordinates

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 coordinate to UCS coordinates. 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.

2.2.1 The Unified Coordinate System

UCS is defined by the coordinate transformation:

$$\begin{bmatrix} dt \\ dx \\ dy \\ dz \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ U & A & L & P \\ V & B & M & Q \\ W & C & N & R \end{bmatrix} \cdot \begin{bmatrix} d\lambda \\ d\xi \\ d\eta \\ d\zeta \end{bmatrix} \tag{2.1}$$

2.1 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:

$$\begin{bmatrix} \frac{\partial}{\partial t} \\ \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} = \begin{bmatrix} 1 & -U_\xi & -U_\eta & -U_\zeta \\ 0 & \frac{MR-NQ}{J} & \frac{CQ-BR}{J} & \frac{BN-CM}{J} \\ 0 & \frac{NP-LR}{J} & \frac{AR-CP}{J} & \frac{CL-AN}{J} \\ 0 & \frac{LQ-MP}{J} & \frac{BP-AQ}{J} & \frac{AM-BL}{J} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial}{\partial \lambda} \\ \frac{\partial}{\partial \xi} \\ \frac{\partial}{\partial \eta} \\ \frac{\partial}{\partial \zeta} \end{bmatrix} \tag{2.2}$$

In Eq. 2.2, we have defined the following relations:

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

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

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

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.

The UCS transformation in Eq. 2.1 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:

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

Upon expansion, and using the notation of Eq. 2.1, this constraint is equivalent to the following compatibility conditions:

$$\begin{array}{lll}
\frac{\partial A}{\partial \lambda} = \frac{\partial U}{\partial \xi} & \frac{\partial L}{\partial \lambda} = \frac{\partial U}{\partial \eta} & \frac{\partial P}{\partial \lambda} = \frac{\partial U}{\partial \zeta} \\
\frac{\partial B}{\partial \lambda} = \frac{\partial V}{\partial \xi} & \frac{\partial M}{\partial \lambda} = \frac{\partial V}{\partial \eta} & \frac{\partial Q}{\partial \lambda} = \frac{\partial V}{\partial \zeta} \\
\frac{\partial C}{\partial \lambda} = \frac{\partial W}{\partial \xi} & \frac{\partial N}{\partial \lambda} = \frac{\partial W}{\partial \eta} & \frac{\partial R}{\partial \lambda} = \frac{\partial W}{\partial \zeta} \\
\frac{\partial A}{\partial \eta} = \frac{\partial L}{\partial \xi} & \frac{\partial A}{\partial \zeta} = \frac{\partial P}{\partial \xi} & \frac{\partial L}{\partial \zeta} = \frac{\partial P}{\partial \eta} \\
\frac{\partial B}{\partial \eta} = \frac{\partial M}{\partial \xi} & \frac{\partial B}{\partial \zeta} = \frac{\partial Q}{\partial \xi} & \frac{\partial M}{\partial \zeta} = \frac{\partial Q}{\partial \eta} \\
\frac{\partial C}{\partial \eta} = \frac{\partial N}{\partial \xi} & \frac{\partial C}{\partial \zeta} = \frac{\partial R}{\partial \xi} & \frac{\partial N}{\partial \zeta} = \frac{\partial R}{\partial \eta}
\end{array} \tag{2.3}$$

The conditions of Eq. 2.3 are not independent. In particular, if the 9 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.

2.2.2 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:

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

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 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:

$$\oint_{\partial V} F_\mu = 0 \tag{2.4}$$

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:

$$\frac{\partial}{\partial x^\mu} F_\mu = 0 \tag{2.5}$$

Under the UCS transformation defined in Eqs. 2.1 and 2.2, Eq. 2.5 becomes:

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

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

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

Using these identities, it is possible to show:

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

Upon collecting terms, this becomes:

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

The next step is to eliminate the remaining non-conservative terms. It can be shown that $\frac{\partial J}{\partial \lambda} - \frac{\partial}{\partial \xi^j} \left(J\, U_i \frac{\partial \xi_j}{\partial x^i} \right)$ is identically zero[9], leaving 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:

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

In our notation, this becomes:

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

It is then possible to rewrite:

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

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:

$$\frac{\partial (J\,F_0)}{\partial \lambda} + \frac{\partial}{\partial \xi^j}\left(J\frac{\partial \xi_j}{\partial x^i}\left(F_i - U_i F_0\right)\right) = 0 \tag{2.6}$$

The corresponding weak form may be derived similarly.

$$\begin{aligned}&\textstyle\int_{\partial V} J\,F_0 d\xi_1 d\xi_2 d\xi_3 + \int_{\partial V} J\frac{\partial \xi_1}{\partial x^i}\left(F_i - U_i F_0\right) d\lambda d\xi_2 d\xi_3\\ &+ \textstyle\int_{\partial V} J\frac{\partial \xi_2}{\partial x^i}\left(F_i - U_i F_0\right) d\lambda d\xi_1 d\xi_3 + \int_{\partial V} J\frac{\partial \xi_3}{\partial x^i}\left(F_i - U_i F_0\right) d\lambda d\xi_1 d\xi_2 = 0\end{aligned} \tag{2.7}$$

Eq. 2.6 describes the behavior of the physical flow quantities under the UCS transformation. It is known from Eq. 2.3 that evolution equations also exist for the grid metric components. These may be handled by appending the time-dependent compatibility conditions to Eq. 2.6to yield an expanded equation set.

Using Eq. 2.2, it is possible to define:

$$\frac{D\xi_i}{Dt} \equiv \left(\frac{\partial}{\partial t} + u_j\frac{\partial}{\partial x^j}\right)\xi_i = \left(u_j - U_j\right)\frac{\partial}{\partial x^j}\xi_i \tag{2.8}$$

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

$$F_0 \equiv \begin{bmatrix} \rho J \\ \rho J u \\ \rho J v \\ \rho J w \\ \rho J e \\ A \\ B \\ C \\ L \\ M \\ N \\ P \\ Q \\ R \end{bmatrix} ; \; F_1 \equiv \begin{bmatrix} \rho J \frac{D\xi}{Dt} \\ \rho J \frac{D\xi}{Dt} u + J \frac{\partial \xi}{\partial x} p \\ \rho J \frac{D\xi}{Dt} v + J \frac{\partial \xi}{\partial y} p \\ \rho J \frac{D\xi}{Dt} w + J \frac{\partial \xi}{\partial z} p \\ \rho J \frac{D\xi}{Dt} e + J u_j \frac{\partial \xi}{\partial x^j} p \\ -U \\ -V \\ -W \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} ; \; F_2 \equiv \begin{bmatrix} \rho J \frac{D\eta}{Dt} \\ \rho J \frac{D\eta}{Dt} u + J \frac{\partial \eta}{\partial x} p \\ \rho J \frac{D\eta}{Dt} v + J \frac{\partial \eta}{\partial y} p \\ \rho J \frac{D\eta}{Dt} w + J \frac{\partial \eta}{\partial z} p \\ \rho J \frac{D\eta}{Dt} e + J u_j \frac{\partial \eta}{\partial x^j} p \\ 0 \\ 0 \\ 0 \\ -U \\ -V \\ -W \\ 0 \\ 0 \\ 0 \end{bmatrix} ; \; F_3 \equiv \begin{bmatrix} \rho J \frac{D\zeta}{Dt} \\ \rho J \frac{D\zeta}{Dt} u + J \frac{\partial \zeta}{\partial x} p \\ \rho J \frac{D\zeta}{Dt} v + J \frac{\partial \zeta}{\partial y} p \\ \rho J \frac{D\zeta}{Dt} w + J \frac{\partial \zeta}{\partial z} p \\ \rho J \frac{D\zeta}{Dt} e + J u_j \frac{\partial \zeta}{\partial x^j} p \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -U \\ -V \\ -W \end{bmatrix} \tag{2.9}$$

Using these redefined flux vectors, along with the spatial compatibility conditions of Eq. 2.3, the strong and weak forms of the transformed Euler equations are given exactly as before:

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

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

2.3 Unsteady Grids and Grid Motion Control

The equations defined by the UCS flux vectors in Eq. 2.9 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. 2.9 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:

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

Using Eq. 2.2, this may be written:

$$(u_i - U_i)\frac{\partial \eta}{\partial x^i} = (u_i - U_i)\frac{\partial \zeta}{\partial x^i} = 0 \tag{2.10}$$

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.

2.3.1 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:

$$\frac{\partial}{\partial\lambda}\left(\nabla_{\vec{x}}\xi\cdot\nabla_{\vec{x}}\eta\right)=0 \tag{2.11}$$

For two dimensional flow, $C=N=P=Q=0$ and $R=1$, so Eq. 2.11 becomes:

$$\frac{\partial}{\partial\lambda}\left(\frac{AL+BM}{\sqrt{A^2+B^2}\sqrt{L^2+M^2}}\right)=0 \tag{2.12}$$

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 $\frac{\partial}{\partial\lambda}\left(ST\right)=\left(L\dot{L}+M\dot{M}\right)\frac{T}{S}+\left(A\dot{A}+B\dot{B}\right)\frac{S}{T}$

Eq. 2.12 can then be rewritten:

$$0=\left(\dot{A}L+A\dot{L}+\dot{B}M+B\dot{M}\right)S^2T^2-\left(AL+BM\right)\left[\left(L\dot{L}+M\dot{M}\right)T^2+\left(A\dot{A}+B\dot{B}\right)S^2\right]$$

$$\Downarrow$$

$$0=\dot{A}S^2\left[T^2L-A\left(AL+BM\right)\right]+\dot{B}S^2\left[T^2M-B\left(AL+BM\right)\right]$$
$$+\dot{L}T^2\left[S^2A-L\left(AL+BM\right)\right]+\dot{M}T^2\left[S^2B-M\left(AL+BM\right)\right]$$

which leads to:

$$0=S^2\left(B\dot{A}-A\dot{B}\right)+T^2\left(L\dot{M}-M\dot{L}\right) \tag{2.13}$$

At this point, the compatibility conditions of Eq. 2.3 can be applied:

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

which leads to:

$$0=S^2\left(B\frac{\partial U}{\partial\xi}-A\frac{\partial V}{\partial\xi}\right)+T^2\left(L\frac{\partial V}{\partial\eta}-M\frac{\partial U}{\partial\eta}\right) \tag{2.14}$$

At this point, one might choose to solve Eq. 2.14 for U in terms of grid velocity magnitude and flow velocity angle θ, defining:

$$U=\exp\left(g\right)\cos\theta;\;\;V=\exp\left(g\right)\sin\theta$$

$$\Downarrow$$

$$\tfrac{\partial U}{\partial\xi}=\exp\left(g\right)\tfrac{\partial g}{\partial\xi}\cos\left(\theta\right)-\exp\left(g\right)\sin\left(\theta\right)\tfrac{\partial\theta}{\partial\xi}$$
$$\tfrac{\partial V}{\partial\xi}=\exp\left(g\right)\tfrac{\partial g}{\partial\xi}\sin\left(\theta\right)+\exp\left(g\right)\cos\left(\theta\right)\tfrac{\partial\theta}{\partial\xi}$$

This allows one to write Eq. 2.14 as:

$$\begin{aligned} 0 = S^2 \left(A \sin\left(\theta\right) - B \cos\left(\theta\right)\right) \frac{\partial g}{\partial \xi} + T^2 \left(M \cos\left(\theta\right) - L \sin\left(\theta\right)\right) \frac{\partial g}{\partial \eta} \\ + S^2 \left(A \cos\left(\theta\right) + B \sin\left(\theta\right)\right) \frac{\partial \theta}{\partial \xi} - T^2 \left(M \sin\left(\theta\right) + L \cos\left(\theta\right)\right) \frac{\partial \theta}{\partial \eta} \end{aligned} \tag{2.15}$$

Eq. 2.15 may then be solved using a variety of techniques.

Alternatively, one may return to Eq. 2.14, and derive instead an equation for U. Based on Eqs. 2.10 and 2.3, one can write:

$$V = v - \frac{B}{A}\left(u - U\right)$$

$$\Downarrow$$

$$V_{\xi^i} = v_{\xi^i} + \frac{B_{\xi^i} A + B A_{\xi^i}}{A^2}\left(U - u\right) + \frac{B}{A}\left(U_{\xi^i} - u_{\xi^i}\right)$$

Substituting this into Eq. 2.14 yields:

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

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

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

This can finally be written:

$$\begin{aligned} 0 = U_\eta + \frac{S^2 A}{T^2 J}\left(A v_\xi - B u_\xi\right) - \frac{L}{J}\left(A v_\eta - B u_\eta\right) \\ + \left(\frac{S^2}{T^2 J}\left(B_\xi A + B A_\xi\right) - \frac{L}{AJ}\left(B_\eta A + B A_\eta\right)\right)\left(U - u\right) \end{aligned} \tag{2.16}$$

In some situations, this form may be more advantageous.

2.3.2 Jacobian Preservation

For general three-dimensional flows, grid-angle preservation is no longer possible. One alternative approach is to preserve the Jacobian of the transformation:

$$\frac{\partial J}{\partial \lambda} = 0 \tag{2.17}$$

It is helpful to define a few useful variables:

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

and

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

$$\Downarrow$$

$$J = \vec{A} \cdot \vec{J}_1 = \vec{L} \cdot \vec{J}_2 = \vec{P} \cdot \vec{J}_3$$

Rewriting the material coordinates from Eq. 2.10:

$$\begin{aligned} J\tfrac{D\eta}{Dt} &= \varepsilon_{ijk}\left(u_i - U_i\right) P_j A_k = 0 \\ J\tfrac{D\zeta}{Dt} &= \varepsilon_{ijk}\left(u_i - U_i\right) A_j L_k = 0 \end{aligned} \tag{2.18}$$

and using the compatibility conditions from Eq. 2.3, the equation for preservation of the Jacobian can be written as:

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

which leads to

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

Differentiating Eq. 2.18 results in

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

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

$$\frac{\partial U_i}{\partial \xi} J_{1i} + \frac{\partial u_i}{\partial \eta} J_{2i} + \frac{\partial u_i}{\partial \zeta} J_{3i} + \left(u_i - U_i\right)\left(\frac{\partial}{\partial \eta} J_{2i} + \frac{\partial}{\partial \zeta} J_{3i}\right) \tag{2.19}$$

2.3.3 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. 3.4) 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.

2.4 Solving the UCS Equations

The equations defined by Eqs. 2.7 and 2.9 are complex, perhaps hopelessly so when the equations of grid motion control in Eqs. 2.8 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[4]. 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. The flow of information in such a UCS code is diagrammed in Fig. %%.

In order to use unsteady coordinates to simulate fluid flow in a physical system, provisions must be made for adding and removing grid points as they move into and out of the simulation

region. The most common way to accomplish this is by creating points at the upstream boundary whenever the upstream edge of the grid has moved a downstream distance of Δx, and by removing points as they pass out of the region.

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.

2.4.1 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:

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

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.

2.5 The Riemann Problem

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 $\mathbf{W}$ would be given by $\overline{\mathbf{W}} = \frac{1}{V}\int_V \mathbf{W}(\xi,\ \eta,\ \zeta)\, dV$. 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[12].

2.5.1 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. 2.9, 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[2]:

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

$$F'_0 = \begin{bmatrix} \rho J \\ \rho J\omega \\ \rho J\tau_1 \\ \rho J\tau_2 \\ \rho Je \\ A \\ B \\ C \end{bmatrix}; \quad F'_1 = \begin{bmatrix} \rho S(\omega - \Omega) \\ \rho S(\omega - \Omega)\omega + p \\ \rho S(\omega - \Omega)\tau_1 \\ \rho S(\omega - \Omega)\tau_2 \\ \rho S(\omega - \Omega)e + \omega p \\ -U \\ -V \\ -W \end{bmatrix}; \quad S_0 = \begin{bmatrix} 0 \\ s_{11} \\ s_{12} \\ s_{13} \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.20}$$

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

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

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. 2.20 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:

$$\overline{\frac{\partial \hat{e}_1}{\partial x^i}} = \frac{\left(\left(\frac{\partial \xi}{\partial x^i}\right)_L + \left(\frac{\partial \xi}{\partial x^i}\right)_R\right)}{\left\|\left(\frac{\partial \xi}{\partial x^i}\right)_L + \left(\frac{\partial \xi}{\partial x^i}\right)_R\right\|} \tag{2.21}$$

Both flow and grid velocity components are transformed using Eq. 2.21. 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.

2.5.2 Eigensystem of the grid-aligned equations

In order to compute the solution to the one-dimensional Riemann problem defined by Eq. 2.20 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):

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

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

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:

$$\begin{aligned} &\left(\sigma\frac{\partial F'_0}{\partial \mathbf{W}} - \frac{\partial F'_1}{\partial \mathbf{W}}\right) = 0 \\ &\Downarrow \\ &\frac{\rho^3\sigma^3\hat{\sigma}^3\left(S^2\frac{\gamma p}{\rho}-\hat{\sigma}^2\right)}{\gamma-1} \\ &\text{where } \hat{\sigma} \equiv J\sigma - S(\omega-\Omega) \\ &\Downarrow \\ &\sigma_1 = 0 (\text{multiplicity of } 3) \\ &\sigma_2 = \frac{S}{J}(\omega-\Omega)\,(\text{multiplicity of } 3) \\ &\sigma_\pm = \frac{S}{J}\left(\omega-\Omega\pm\sqrt{\frac{\gamma p}{\rho}}\right) \end{aligned} \tag{2.22}$$

The corresponding eigenvectors are:

$$
\begin{aligned}
r_1 &= \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}; \quad r_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}; \quad r_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \\
r_4 &= \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}; \quad r_5 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}; \quad r_6 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \\
r_+ &= \begin{bmatrix} 1 \\ \frac{1}{a^2} \\ -\frac{1}{a\rho} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}; \quad r_- = \begin{bmatrix} 1 \\ \frac{1}{a^2} \\ \frac{1}{a\rho} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}
\end{aligned}
\tag{2.23}
$$

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.

2.5.3 Riemann invariants and rarefaction wave relations

The generalized Riemann invariants are relations that hold true across smooth waves, and are given by the relations[12]:

$$\frac{dw_1}{{r_i}^1} = \frac{dw_2}{{r_i}^2} = \cdots = \frac{dw_m}{{r_i}^m}$$

$$\Downarrow$$

$$\frac{\partial\rho}{\partial p} = \frac{1}{a^2};\ \frac{\partial\omega}{\partial p} = \frac{\pm 1}{\rho a};\ \frac{\partial\tau_1}{\partial p} = 0;\ \frac{\partial\tau_2}{\partial p} = 0;$$

Relations for flow quantities may be derived based on an upstream state given by p_0,ρ_0,ω_0,$\tau_1 0$,$\tau_2 0$. Beginning with ρ, it can be shown:

$$\begin{gathered}\frac{\partial\rho}{\partial p} = \frac{\rho}{\gamma p} \Rightarrow \frac{\partial\rho/\partial p}{\rho} = \frac{1}{\gamma p} \Rightarrow \ln(\rho) = \frac{\ln(p)}{\gamma} + C_1 \Rightarrow \rho = C_1 p^{1/\gamma} \Rightarrow \rho = \rho_0 \alpha^{\frac{1}{\gamma}} \\ \alpha \equiv \frac{p}{p_0}\end{gathered} \tag{2.24}$$

Normal velocity may then be similarly derived:

$$\begin{gathered}\frac{\partial\omega}{\partial p} = \pm\frac{1}{\rho a} = \pm\left(a_0\rho_0 p_0 \alpha^{\frac{\gamma+1}{2\gamma}}\right)^{-1} \Rightarrow \frac{\partial\omega}{\partial\alpha} = \pm\frac{a_0}{\gamma}\alpha^{\frac{-(\gamma+1)}{2\gamma}} \Rightarrow \omega = \pm\frac{2\gamma}{\gamma-1}\frac{a_0}{\gamma}\alpha^{\frac{\gamma-1}{2\gamma}} + C_1 \\ \Downarrow \\ \omega = \omega_0 \pm \frac{2a_0}{\gamma-1}\left(\alpha^{\frac{\gamma-1}{2\gamma}} - 1\right)\end{gathered} \tag{2.25}$$

The tangential velocities are simple:

$$\frac{\partial\tau_1}{\partial p} = \frac{\partial\tau_2}{\partial p} = 0 \tag{2.26}$$

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. 2.22:

$$S_H = \frac{S}{J}\left(\omega_0 - \bar{\Omega} \pm a_0\right);\quad S_T = \frac{S}{J}\left(\omega_* - \bar{\Omega} \pm a_*\right) \tag{2.27}$$

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

$$\frac{\xi}{\lambda} = \frac{S}{J}\left[(\omega - \Omega) \pm a\right] \Rightarrow \omega = \frac{J}{S}\left(\frac{\xi}{\lambda}\right) \mp a + \Omega \tag{2.28}$$

Eqs. 2.25 and 2.28 can be combined to solve for the pressure ratio $\alpha \equiv \frac{p}{p_0}$:

$$\begin{gathered}
\frac{J}{S}\left(\frac{\xi}{\lambda}\right) \mp a + \Omega = \omega_0 \pm \frac{2a_0}{\gamma-1}\left(\alpha^{\frac{\gamma-1}{2\gamma}} - 1\right) \Rightarrow (\omega_0 - \Omega) \pm \frac{\gamma+1}{\gamma-1} \mp \frac{2a_0}{\gamma-1} - \frac{J}{S}\frac{\xi}{\lambda} = 0 \\
\Downarrow \\
\pm\left(\omega_0 - \Omega - \frac{J}{S}\frac{\xi}{\lambda}\right) + \frac{\gamma+1}{\gamma-1}a - \frac{2a_0}{\gamma-1} = 0 \Rightarrow \frac{2}{\gamma+1} \mp \frac{\gamma-1}{a_0(\gamma+1)}\left(\omega_0 - \Omega - \frac{J}{S}\frac{\xi}{\lambda}\right) = \alpha^{\frac{\gamma-1}{2\gamma}} \\
\Downarrow \\
\alpha = \left[\frac{2}{\gamma+1} \mp \frac{\gamma-1}{a_0(\gamma+1)}\left(\omega_0 - \Omega - \frac{J}{S}\frac{\xi}{\lambda}\right)\right]^{\frac{2\gamma}{\gamma-1}}
\end{gathered} \tag{2.29}$$

2.5.4 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[12]:

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

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:

$$\begin{aligned}
\rho &= \rho_0 \tfrac{\alpha(\gamma+1)+(\gamma-1)}{\alpha(\gamma-1)+(\gamma+1)} \\
\omega &= \omega_0 \pm \frac{(\alpha-1)a_0}{\sqrt{\frac{1}{2}\gamma[(\gamma+1)\alpha+(\gamma-1)]}} \\
\tau_1 &= \tau_{10} \\
\tau_2 &= \tau_{20} \\
S &= \tfrac{S}{J_0}\left[\omega_0 - \Omega \pm a_0\sqrt{\tfrac{\gamma+1}{2\gamma}(\alpha-1)+1}\right]
\end{aligned} \tag{2.30}$$

2.5.5 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.

2.5.6 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 Fig. %%. 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. 2.24, 2.25, and 2.30, it is possible to define a function of pressure representing the jump in normal velocity across the central slip line:

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

$\omega\left(\mathbf{W}_i\right)$ 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.

2.5.7 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 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:

$$\begin{aligned}
&w_R = w_{i+1} - \tfrac{1}{2}\left(w_{i+2} - w_{i+1}\right)\phi\left(\tfrac{w_{i+1}-w_i}{w_{i+2}-w_{i+1}}\right)\\
&w_L = w_i + \tfrac{1}{2}\left(w_i - w_{i-1}\right)\phi\left(\tfrac{w_{i+1}-w_i}{w_i-w_{i-1}}\right)\\
&\text{where } \phi \text{ is the minmod limiter given by :}\\
&\phi\left(w\right) = \max\left(0, \min\left(1, w\right)\right)
\end{aligned} \tag{2.31}$$

2.6 Algorithms

Hui[4] uses a dimensional splitting technique to solve the multidimensional Euler equations, as follows:

- For each coordinate direction n:
  - For each cell i, j, k:
    - For each interface $+$, $-$:
      - Apply MUSCL reconstruction using Eq. 2.31.
      - Transform flow and grid velocity to normal and tangential components using:

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

      - Solve the Riemann problems as described in section 3.6 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 using $\frac{\partial x_{+i}}{\partial \hat{e}^j} = \left(\frac{\partial \hat{e}_{+i}}{\partial x^j}\right)^{-1}$
    - 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. 2.5 or 2.4, and updated metric components.

– Update conserved variables using (e.g.):

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

$$\Downarrow$$

$${F_0}^{t_0+\Delta t} = {F_0}^{t_0} - \frac{\Delta t}{\Delta \xi_n} F_n$$

– Convert updated conserved variables to updated primitive variables.

2.6.1 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 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:

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

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

2.7 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.

2.7.1 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. 2.7.1, 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.

2.7.2 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. 2.2. This problem is identical to the one given by Hui[4], 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.

It can also be seen in Fig. 2.3 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, does resolve the slip line, and makes slightly more accurate predictions of shock locations, at the cost of a less-well-resolved expansion corner.

2.7.3 Basic boundary-layer effects

Many phenomena in hypersonics 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[8]:

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

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. 2.4. The presence of the boundary layer can be clearly seen in the curving of the inviscid flow in the otherwise uniform channel, as well as the formation of the oblique shock train.

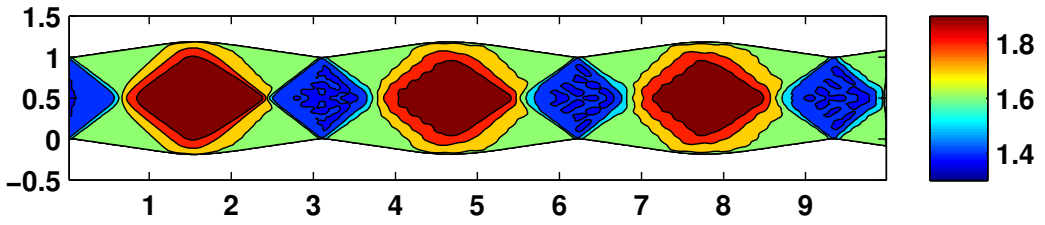


Figure 2.1: Computed Mach number for an under-expanded nozzle flow, showing the diamond-shock train

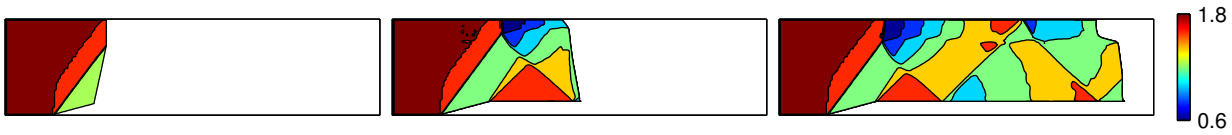
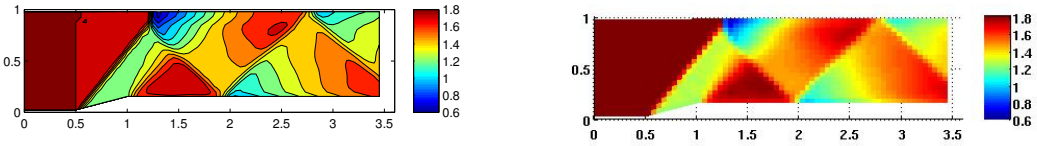
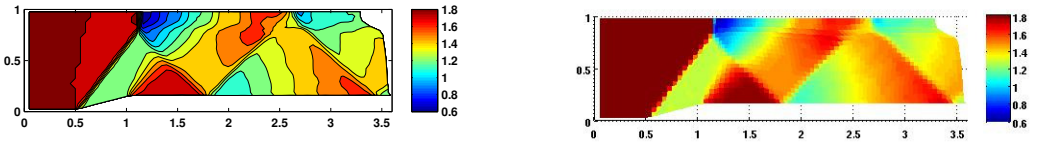


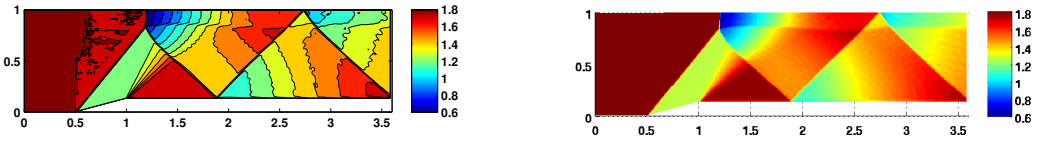
Figure 2.2: Computed Mach number for transonic duct flow at various times, showing the manner in which nodes flow to fill boundaries.



(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 2.3: Qualitative accuracy comparison between Unified ($h = 0.25$) (b) and Eulerian ($h = 0$) (a,c) simulations for a transonic duct flow. Notice the improved resolution of the slip line and the walls for the unified solution.

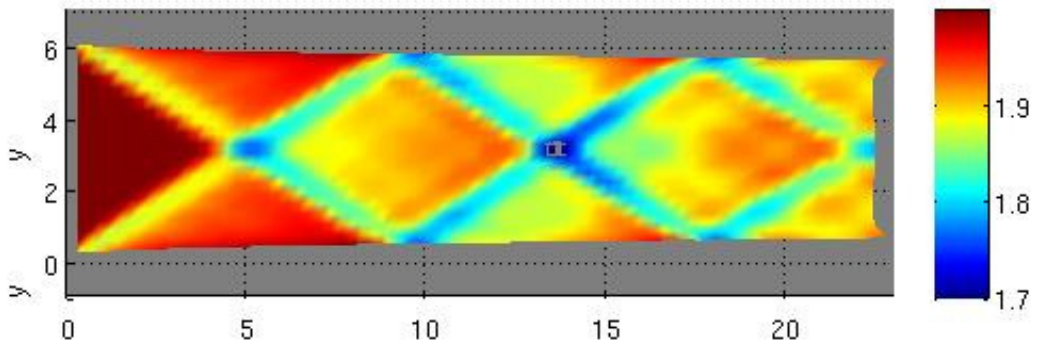


Figure 2.4: Oblique shock train produced by a turbulent boundary layer in an otherwise uniform channel

2.7.4 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 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. 2.5.

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. 2.6, 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.

2.8 Future Developments

Although UCS has shown promise for reducing the costs of CFD while raising accuracy, 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.

2.8.1 A first cut at better boundary conditions

Consider a two-dimensional duct flow containing a ramp, as in Fig. 2.2. How would one apply boundary conditions for this problem? The inflow, outflow, and top wall conditions are

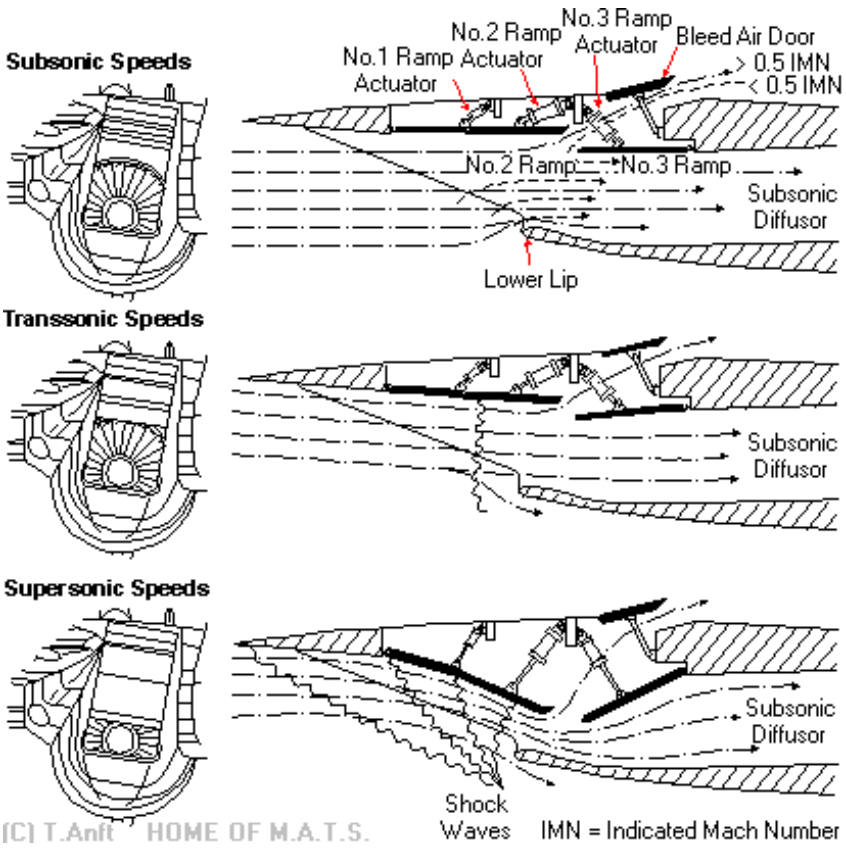


Figure 2.5: Diagram of the variable inlet geometry of the USAF F-14 Tomcat. Courtesy of: Home of M.A.T.S., Available at http://www.anft.net/f-14/f14-detail-airintake.htm, Accessed 25 Nov 2011

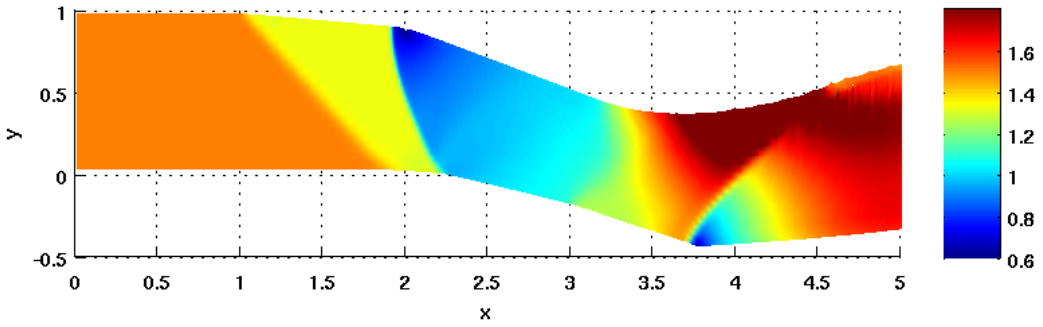


Figure 2.6: Mach number in a geometry modeled after Fig. 2.5.

simple enough, and can be easily implemented as array operations:

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

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
```

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

```
else
```

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

```
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.

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

$$\Delta\xi = \Delta\eta = \Delta\zeta = 1$$

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:

$$(\xi, \eta, \zeta)_{ijk} = (i + \xi_0,\ j + \eta_0,\ k + \zeta_0) \tag{2.32}$$

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. 2.32. 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.

2.8.2 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 Fig. %%. 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[3]

2.8.3 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 3.4. 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.

2.8.4 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.

2.8.5 Conclusion

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. Specific versions and their capabilities will be discussed in more detail in Chapter 3.

Chapter 3

BACL-Streamer and Verification of the Unified Coordinate System

Insert text here

3.1 Background

Insert text here

Be sure to talk about IMMS here, as per discussion in 4.

3.2 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.

v1.0 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.

3.3 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 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.

3.3.1 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

tt 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.

3.4 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.

3.4.1 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. 3.6, 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[4], 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:

$$(p, \rho, M, \theta) = \begin{cases} (0.25, 0.5, 7, 0) & : y > 0 \\ (1, 1, 2.4, 0) & : y < 0 \end{cases} \tag{3.1}$$

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. 3.1 and 3.2. In particular, it should be noted that the resolution of the slip line is greatly improved in the moving grid case. This is a well known effect of Lagrangian-esque simulations, and one of the beneficial features of UCS.

The two-dimensional Riemann problem can be solved exactly[11], and so convergence rates can be measured for this problem. Convergence is tested for both moving and stationary grids, with and without the $2^n d$-order MUSCL update. These convergence rates are shown in Fig. 3.3. 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.[7][6]

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.

3.4.2 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:

$$(p, \rho, M, \theta) = (1, 1, 1.8, 0)$$

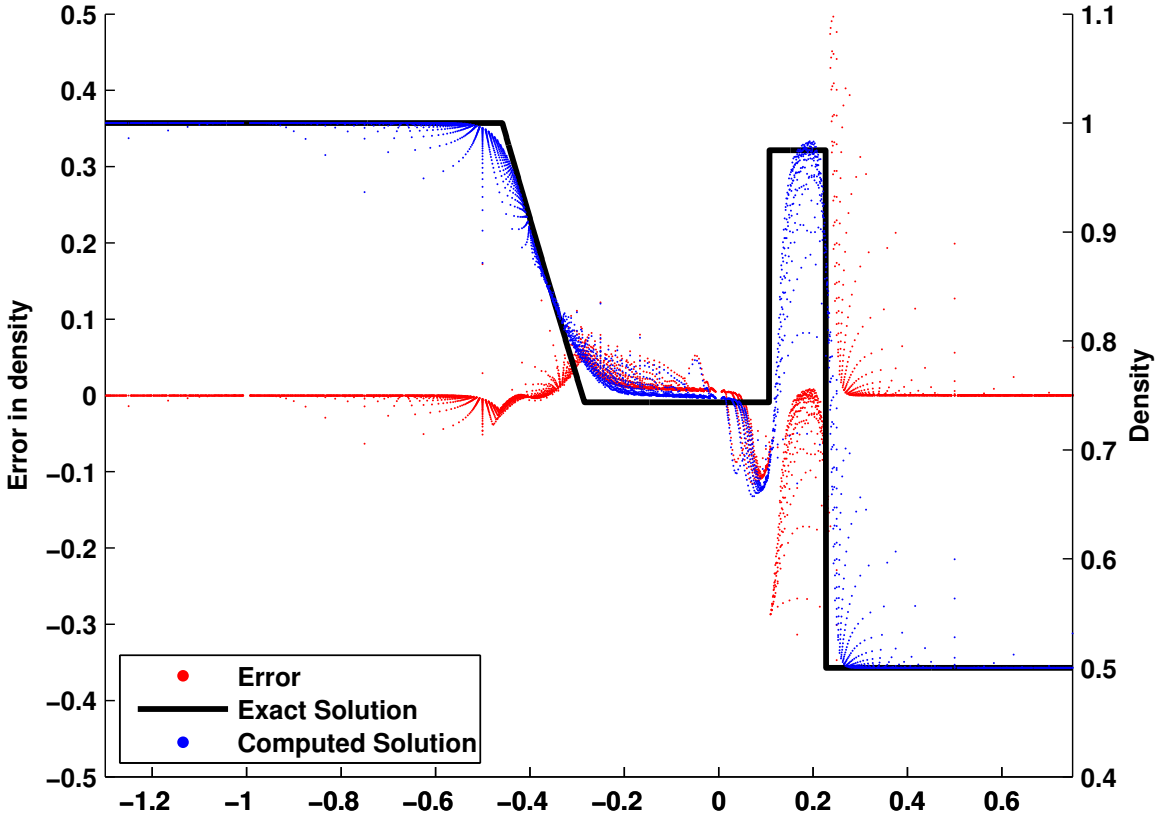


Figure 3.1: The 2-D Riemann problem with corresponding numerical error, stationary grid

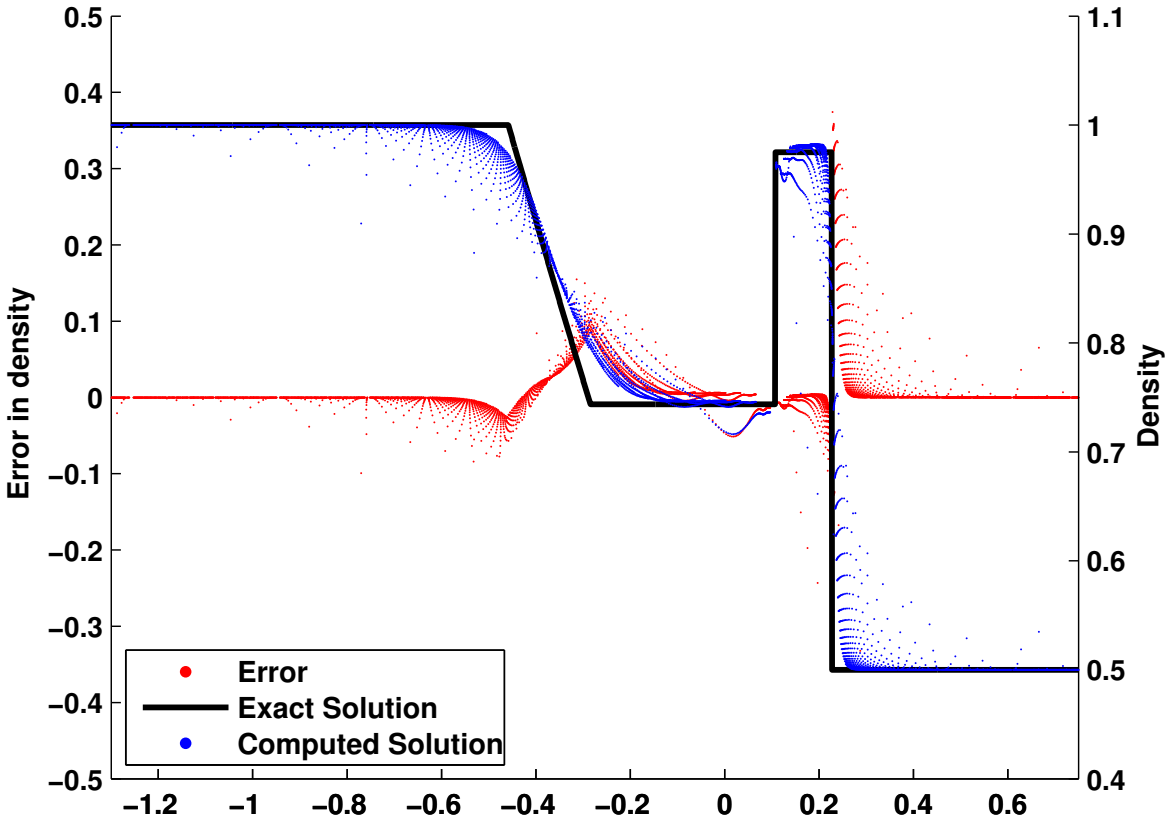


Figure 3.2: The 2-D Riemann problem with corresponding numerical error, moving grid

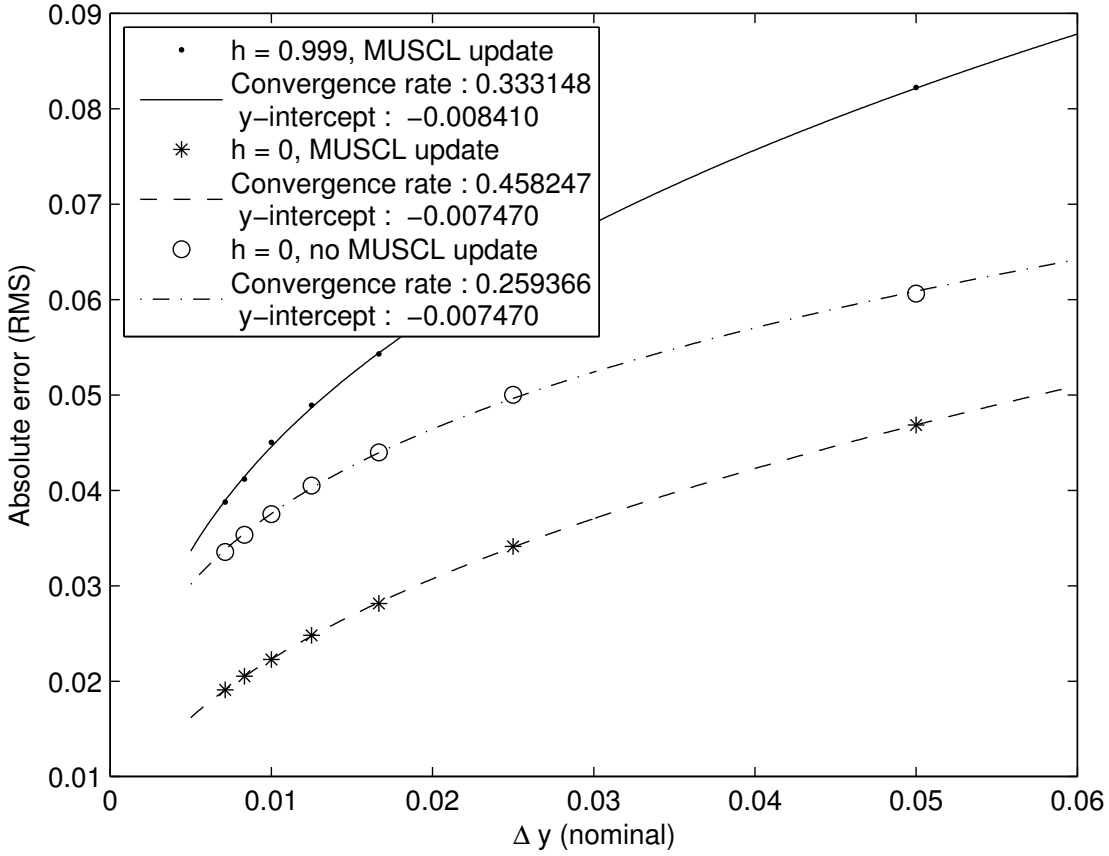


Figure 3.3: Order of convergence n of `BACL-Streamer` v2.0 for a two-dimensional Riemann problem.

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 again fall short of expectations. Like the Riemann 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.

3.4.3 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. 3.6. 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. 3.7.

3.4.4 Conclusion

Overall, this type of verification has highlighted the difficulties that must be overcome by any UCS program. Moving grids are not a simple thing to implement, especially when combined with problems that depend on the precise resolution of singular points. However, 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 v2.0 and the need for a three-dimensional solver, led directly to the development of v3.0.

Figure 3.4: Root-mean-squared error for the oblique shock problem. The appearance of grid instabilities leads to two distinct error curves with different rates of convergence n.

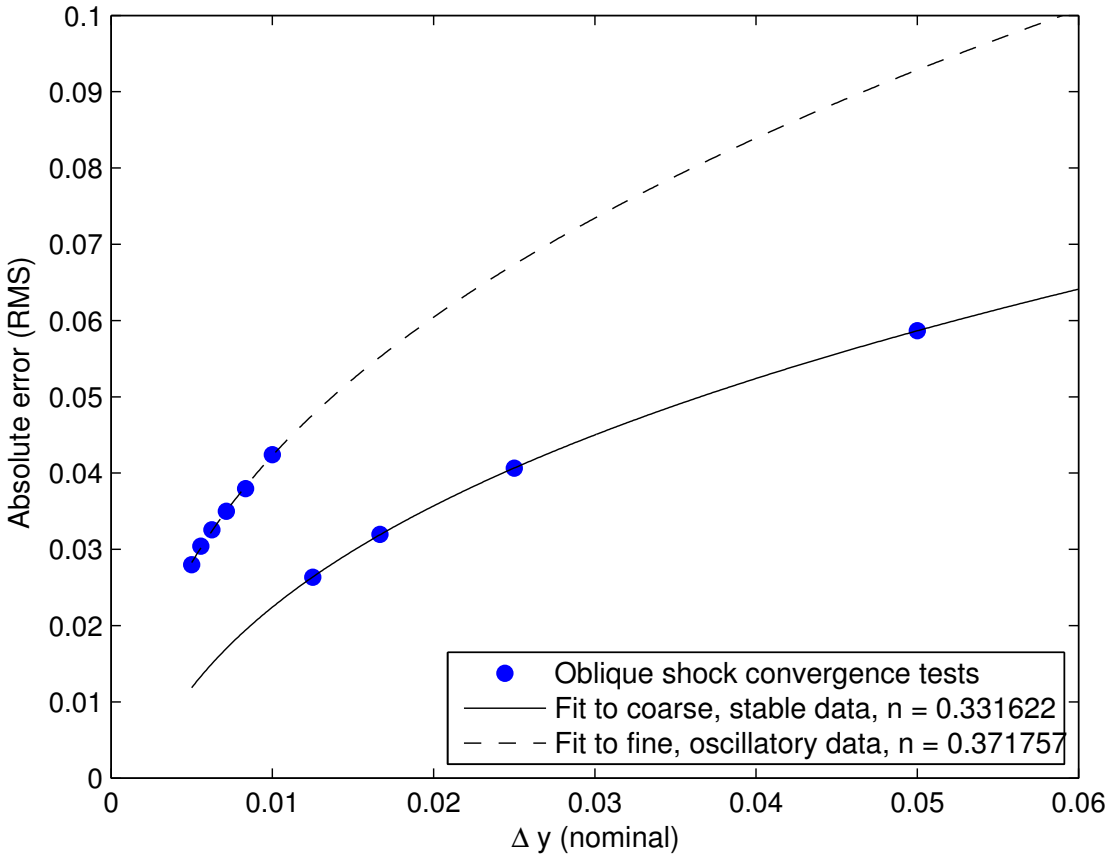
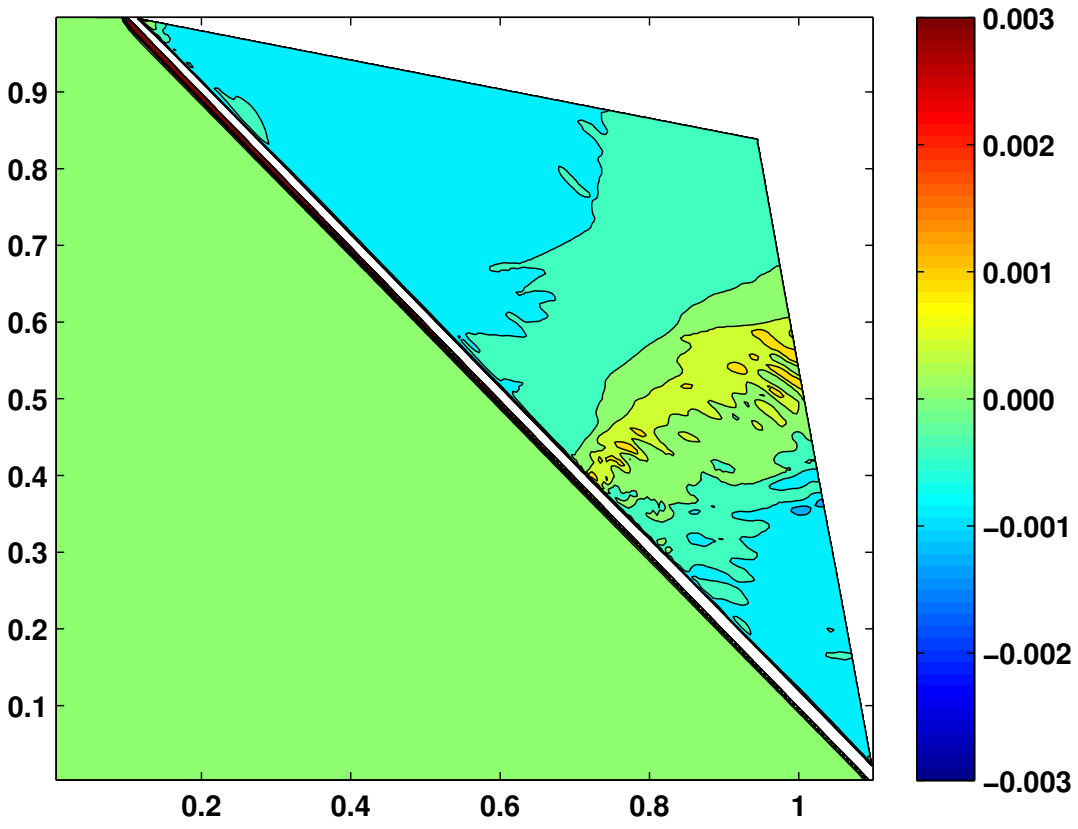


Figure 3.5: A plot of normalized error in pressure, highlighting the oscillations which propagate downstream from the oblique shock.



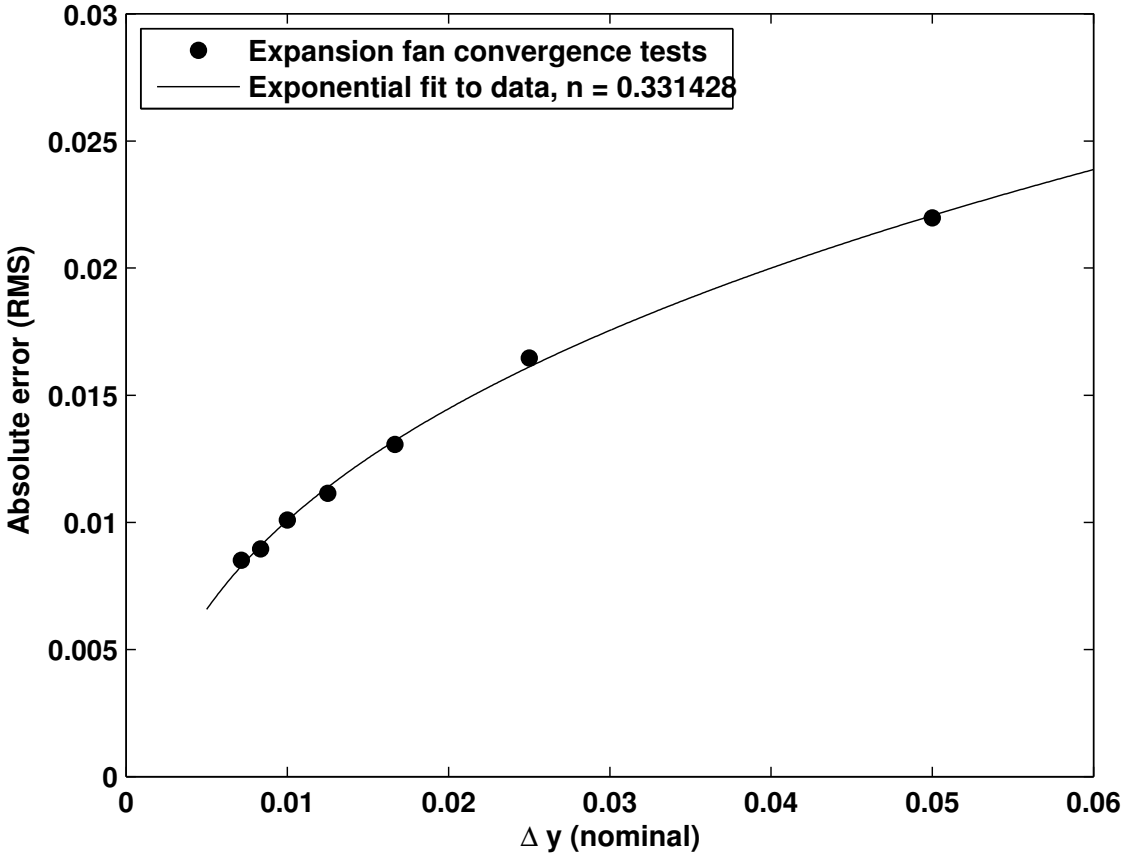


Figure 3.6: Root-mean-squared error for the Prandtl-Meyer expansion, with order of convergence n.

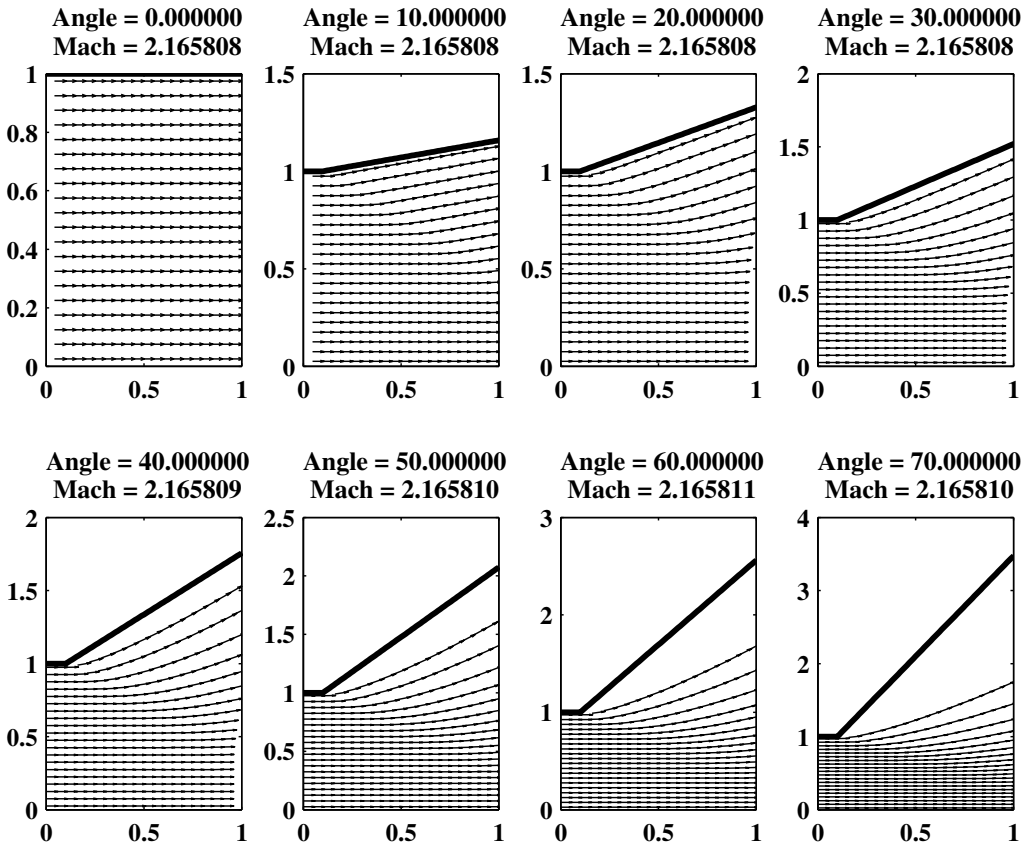


Figure 3.7: Computed streamlines for Prandtl-Meyer expansion at increasing expansion angles. Angles are given in degrees.

3.5 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, which is included as part of the SciPy Python library.

3.5.1 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.

3.5.2 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` 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, which requires that `main_data` be sized at least two points larger in every spatial array dimension, as shown in Fig. %%. Implementing boundary conditions in this way greatly simplifies the Core, and allows the difficult task of implementing boundary conditions to be handled in Python.

3.5.2.1 Core library

`libStreamer` is built using `make all`, and provides everything from basic matrix operations, 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. 2.15 is implemented.

3.5.2.2 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.

3.5.2.3 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 non-trivial UCS components remains incomplete.

3.5.3 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`, which covers basic simulation initialization and time-stepping, and `BoundaryConditions.py`, which is an experimental boundary conditions implementation.

3.5.3.1 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.

3.5.3.2 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 `main.py` and `BoundaryConditions.py` files.

3.5.3.3 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`, `BoundaryConditionsStuff.f90`, and the code-generated `FortranNormalVectors.` work together 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.

3.5.4 BACL-Manufactured and the method of manufactured solutions

`BACL-Streamer` v3.0 is also integrated with the `BACL-manufactured` Python package. `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.

In order compute source terms for use in the method of manufactured solutions, it is necessary to provide both the system of equations and the proposed manufactured solution. With these, it is possible to generate manufactured source terms for differential MMS, as well as the integrands required for computing integral source terms for integrative MMS. `BACL-manufactured` provides the machinery to do this quickly and easily, as efficiently as possible.

The package defines a `SympyEquation` base class to describe the behavior of both integral and differential balance laws. This class provides the following methods for computation of source terms:

- `balance_diff` - Returns symbolic representation of differential manufactured source terms.
- `balance_lambda_init` - Returns the manufactured source terms from `balance_diff` as a Python lambda function.
- `balance_integrate` - Returns integral form of manufactured source terms, integrated over some specified volume.

A `SympyEquation` 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.

The `SympyEquation` class is a general-purpose class, and is not intended for direct use. Instead, subclasses are used to define the equations for flux and source terms. Implementations are available for the linear heat equation and for the UCS Euler equations. These subclasses, along with a selection of appropriate manufactured solutions, are provided in equation-specific modules. This allows for easy expansion of the method to different equation sets.

`BACL-Streamer` also provides the `integration` module, which is a collection of tools used to efficiently evaluate multidimensional integrals based on symbolic integrands with known discontinuities. The numerical methods used are discussed in detail in Section 4.2, so the discussion here is limited to the manipulations required to derive integrals of the appropriate form for evaluation using `nquad`.

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.

Finally, the package also processes any symbolic discontinuities into the appropriate form for the numeric integration routine, as described in Section 4.3. The end result is a complete set of functions defining the critical surfaces of integration.

Once all of this functional machinery has been initialized, the `balance_integrate` function provided by the `SympyEquation` object can be called to evaluate the integral numerically, the results of which become the integrated source terms for use with integral manufactured solutions.

3.6 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. Moving grids have not been verified as yet. The discussion here of integrative manufactured solutions relies heavily on the use of fast, accurate methods for evaluation of multidimensional integrals subject to discontinuities, discussed in detail in Chapter 4.

Because it was designed to incorporate non-conservative, manufactured solutions, tt BACL-Streamer v3.0 includes functionality for the inclusion of manufactured source terms as part of the time-stepping scheme in `main.py`. The resulting equations are solved using a partial-time-step scheme similar to the dimensional splitting approximations, and described in Toro[12]. Differential source terms are incorporated using SciPy's ODE solvers, while integral source terms are treated exactly like the existing flux integrals.

To provide comparable testing with a different, though similar code, verification was also performed on Toro's NUMERICA[12][10]. As the NUMERICA library does not allow for non-conservative source terms, it is verified against the exact, one-dimensional Riemann problem given by the initial states

$$\mathbf{W} \equiv (p, \rho, u)$$

$$\mathbf{W}_L = (1.0,\ 1.0,\ 0.0)$$

$$\mathbf{W}_R = (0.1,\ 0.125,\ 0.0)$$

The solution to this problem is readily obtained by reference to the discussion in Section , or a numerical solution may be obtained from the Riemann solver used in the Godunov algorithm.

`BACL-Streamer` v3.0 was tested against the two-dimensional Riemann problem of Section 3.4.1, as well as the two different manufactured solutions. The first was given by:

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

This provides a smooth, yet non-physical, and provides an excellent test of the Euler solver with source terms. The second solution is similar, with an added discontinuous jump:

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

These two solutions are shown in Fig. 3.6

In all, five different verification tests were run, as parameterized in Table 3.6. 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 doubling the number of points. For the manufactured solutions, the baseline grid had 101 points, and coarsened this to obtain 51-, 26-, and 21-point grids corresponding to $\Delta x = 1, 2, 4, 5$, respectively. For the exact Riemann problem, a 101-point base grid is used to define 201- and 51-point grids, such that $\Delta x = 0.005, 0.01, 0.02$. For the two-dimensional, steady, Riemann problem, the initial grid is sized 60×100 with $\Delta x = \Delta y = 0.01$, which is used to derive 30×50 and 120×200 grids, with $\Delta x = \Delta y = 0.02, 0.005$, respectively.

	Differential steps	Grid sizes	$x \times y$ ranges
Smooth, differential, manufactured solution	$\Delta x = 1, 2, 4, 5$	$nx = 101, 51, 26, 21$	$\{0, 100\}$
Smooth, integral, manufactured solution	$\Delta x = 1, 2, 4, 5$	$nx = 101, 51, 26, 21$	$\{0, 100\}$
Unsteady Riemann problem (NUMERICA)	$\Delta x = 0.005, 0.01, 0.02$	$nx = 201, 101, 51$	$\{0, 1\}$
Discontinuous, integral, manufactured solution	$\Delta x = 1, 2, 4, 5$	$nx = 101, 51, 26, 21$	$\{0, 100\}$
Two-dimensional, steady, Riemann problem	$\Delta x = \Delta y = 0.005, 0.01, 0.02$	$nx \times ny = 30 \times 50$, $60 \times 100, 120 \times 200$	$\{0, 0.6\} \times \{-0.5, 0.5\}$

Once the grids have been defined, numerical solutions were computed and compared to exact

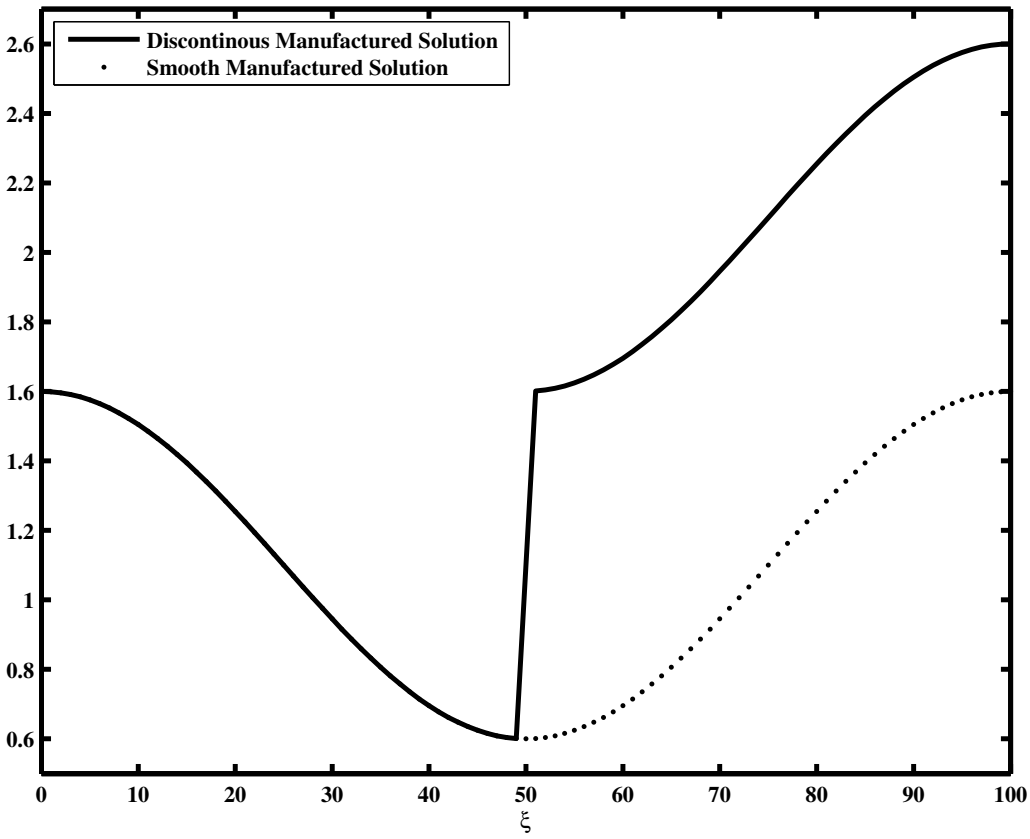


Figure 3.8: Manufactured solutions used for verification of `BACL-Streamer` v3.0

solutions. The RMS value of the pointwise-error is then used to fit the function $f(\Delta x) = A\Delta x^n$. The resulting fits and the functions that describe them are shown in Figs. 3.6 and 3.6. The equation fits are also given in Table 3.6

	Function Fit	Convergence Rate
Smooth, differential, manufactured solution	$y = 8.9 \times 10^{-4}\Delta x^{0.76}$	0.76
Smooth, integral, manufactured solution	$y = 7.7 \times 10^{-4}\Delta x^{0.54}$	0.54
Discontinuous, integral, manufactured solution	$y = 0.13\left(\frac{\Delta x}{500}\right)^{0.56}$	0.56
Unsteady Riemann problem (NUMERICA)	$y = 0.20\Delta x^{0.50}$	0.50
Steady, two-dimensional, Riemann problem	$y = 0.20\Delta x^{0.28}$	0.28

The convergence rates that result for differential MMS and integral MMS are quite comparable. The same agreement is seen when comparing the rate of convergence to discontinuous solutions, whether exact or manufactured. As before, a drop in convergence is observed for discontinuous solutions.

3.7 Future Work

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 development of integrative manufactured solutions has opened many new doors, and allows for a truly thorough verification of both v3.0 and the UCS system as a whole. Beyond the scope of UCS, integrative manufactured solutions have the potential to revolutionize the entire field of code verification, in much the same way that traditional MMS has done in the last decade.

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

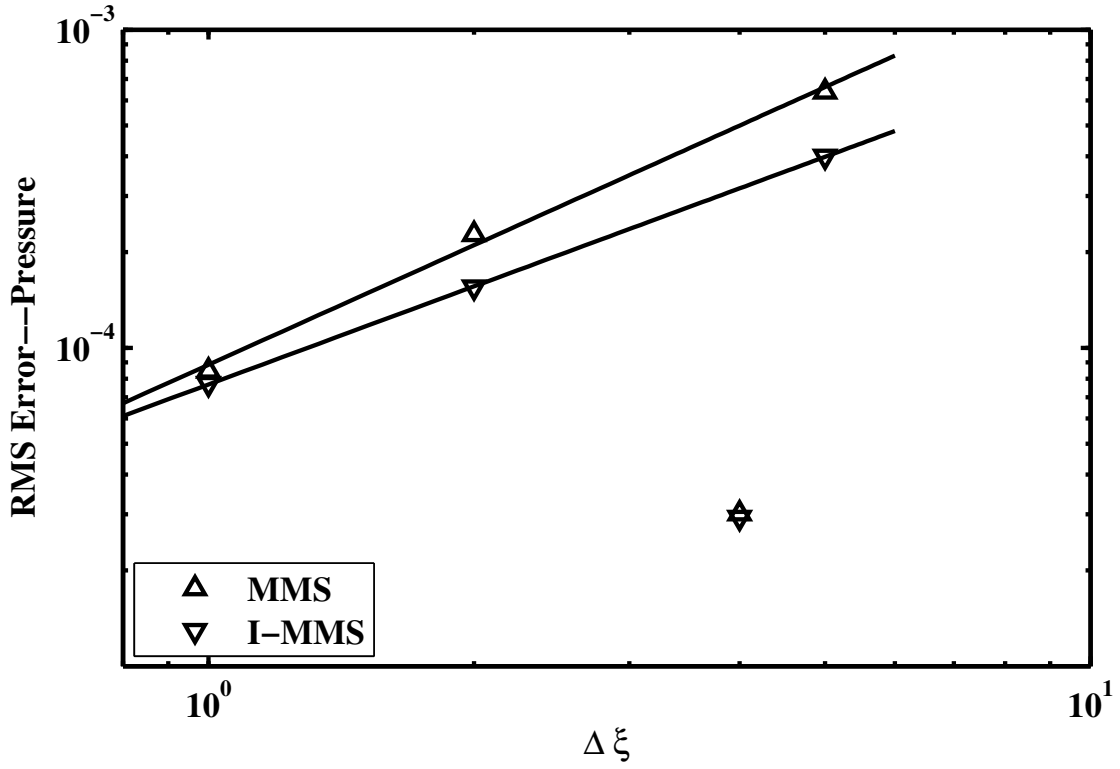


Figure 3.9: Measured convergence rates for `BACL-Streamer` v3.0 using both differential and integrative MMS with smooth manufactured solutions.

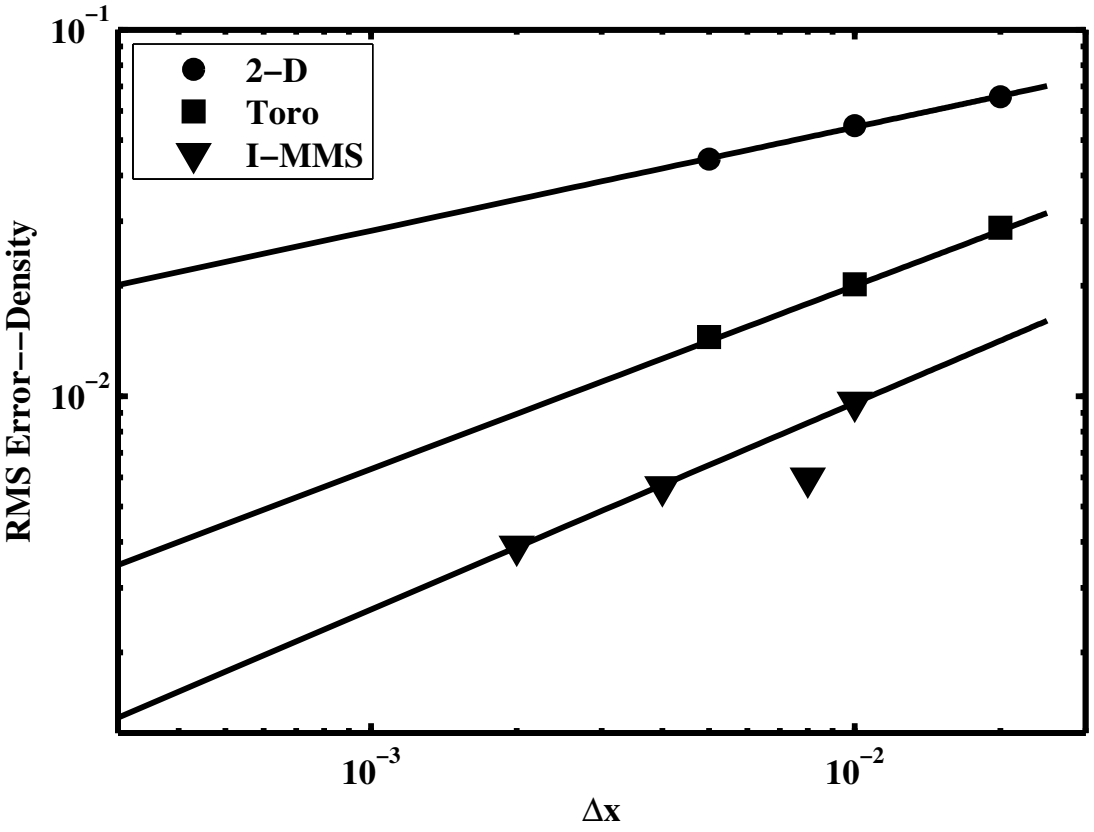


Figure 3.10: Convergence rates for discontinuous solutions: the two-dimensional Riemann problem; the one-dimensional Riemann problem computed with Toro's NUMERICA; the discontinuous manufactured solution.

algorithms, and it will provide designers with enough confidence in the method to use it on new projects.

Chapter 4

Integration of Discontinuous Functions

As discussed in Section 3.1, the principal difficulty inherent in implementing the method of manufactured solutions for integral equations lies in the evaluation of the flux and volume integrals required in order to compute the manufactured source terms. Multidimensional integration is itself a difficult problem, and is made more so by the presence of discontinuities in the integrand. As the intended use of these integrals is to verify the numerical accuracy of other codes, it is important that they be evaluated as accurately as possible, which is also made more difficult by the presence of discontinuities.

To resolve these difficulties and enable the use of integrative manufactured solutions, an n-dimensional integration routine was developed for general types of functions, and methods were devised for the tracking of discontinuities through iterated applications of one-dimensional integration. The resulting algorithm effectively removes discontinuities from the integrand, allowing the evaluation of the integrals to proceed with the level of accuracy that would be expected for a continuously differentiable integrand function.

4.1 Background

Multidimensional integration routines generally belong to one of two classifications. Monte Carlo methods rely on the fact that, over many random function samples taken with the integration region, the average value of the samples will converge to the value of the integral of the function over that region. Monte Carlo methods are especially useful because the computational cost of this

procedure does not increase with the dimensionality of the integral to be evaluated. There is also no additional difficulty introduced when integrating over complex integration domains, nor when integrating ill-behaved (singular, discontinuous) integrand functions. Unfortunately, convergence to the exact value of the integral is quite slow.

In contrast, quadrature methods are more complex, and generally make more assumptions about both the integration domain and the integrand function. Quadrature methods work by sampling the function at some pre-defined number of points, using the samples to fit a polynomial to the integrand, and then evaluating the integral of the polynomial. Most quadrature methods are one-dimensional in nature, but some multidimensional methods do exist.[**?**]. Because quadrature methods use polynomials to approximate the integrand function, difficulties arise when attempting to evaluate functions that cannot easily be represented as polynomials, such as discontinuous functions. For smooth functions, quadrature typically provides excellent accuracy when compared to Monte Carlo methods.

Smoothing of integrand functions can be achieved through subdivision of the integration domain along discontinuous surfaces, resulting in a set of integrals over irregular domains where the integrand is smooth. In multiple dimensions, this process is analogous to the problem of shock-fitting, and very difficult...

Although multidimensional subdivision of integrals is difficult, the same process is straightforward in one dimension. By using one-dimensional integration routines iteratively, the problem is greatly simplified. Some tools are ...

No available tools were able to do n-dimensional integrals, with the level of control necessary ...

Given the initial locations of any discontinuities, it is only necessary to propagate these through each level of integration in order to subdivide the integral appropriately at every level such that the resulting function is smooth.

4.2 Nquad Development

As discussed in Section 4.1, a tool needed to be developed for general n-dimensional integration using iterative application of one-dimensional quadrature. The SciPy library[5] provided the closest such tool. The flexible nature of `scipy.integrate.quad`, together with its support for multivariate functions, provided the basic functionality required for extension to n-dimensional problems, and it was selected as the baseline for development of `nquad`, a general-purpose, flexible routine for the evaluation of multidimensional integrals, with access to the entire suite of `Quadpack` integration tools.

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 are accepted and scheduled for inclusion in version 0.15.

4.2.1 `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 Fig. 4.2.1.

`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.

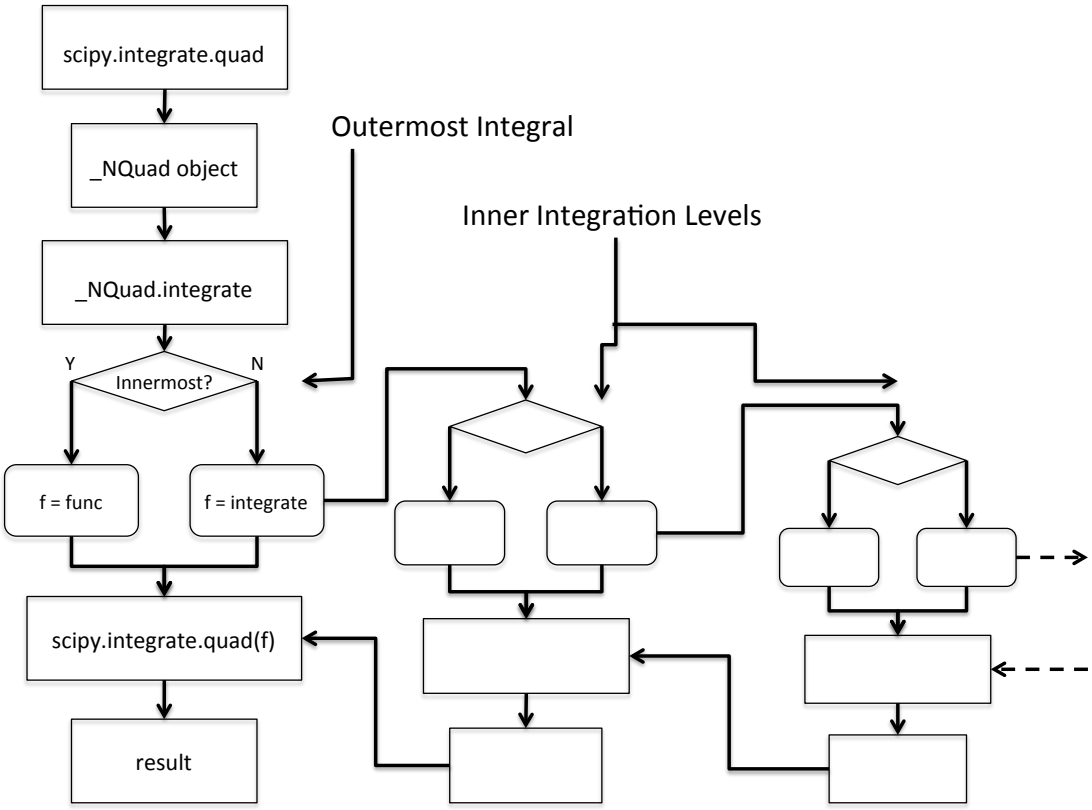


Figure 4.1: The `nquad` algorithm, diagrammed for 3+ dimensional integrals.

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 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[1]. When such information is provided through the `points` argument to `scipy.integrate.quad`, the integration domain is subdivided at those critical points, 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. At inner levels of integration, the function is being evaluated at specific values of the outer integration variables. These values can be used to compute values for the `range` and `opt` arguments corresponding to that level. 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.

4.2.2 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.

4.3 Discontinuity tracking

As discussed in Section 4.1, the principal requirement of any multidimensional integration routine for use in code verification through the integrative method of manufactured solutions is highly accurate evaluation of integrals of discontinuous functions. Monte Carlo methods are insufficiently accurate for this purpose, and quadrature methods do not accurately capture discontinuous behavior. Subdividing the integration domain avoids this issue, but is difficult to achieve in multiple dimensions. Subdivision is much simpler if the multidimensional integrals are expressed as iterated one-dimensional integrals using Fubini's theorem.

The key to understanding domain subdivision for iterated integrals is to recognize that each one-dimensional integration yields a new function. For example, one may consider the integration of a paraboloidal step function in three-dimensional space, given by:

$$F_0(x,y,z) \equiv \begin{cases} 1 & ; \quad x^2+y^2-z<0 \\ 0 & ; \quad else \end{cases} \tag{4.1}$$

Assuming that this function is integrated, with an integration order and volume given by:

$$\int_V F_0(x,y,z) = \int_0^2 \int_{-1}^1 \int_{-1}^1 F_0(x,y,z)\,dxdydz$$

It is then possible to define new functions based on the results of this integration: $F_1(y,z) \equiv \int_{-1}^1 F_0(x,y,z)\,dx$ and $F_2(z) \equiv \int_{-1}^1 F_1(x,y,z)\,dy$ One could also choose a different ordering, which might lead to the function: $F_{12}(x,y) \equiv \int_0^2 F_0 dz$

These four functions are illustrated in Fig. 4.3.

Each of these functions may be considered as an independent entity for the purposes of integration. That is, the behavior of $F_1(y,z)$ need not be considered when evaluating the integral $\int_{-1}^1 F_0(x,y,z)\,dx$. This means that, for the computation of $F_1(y,z)$, discontinuities can be ac-

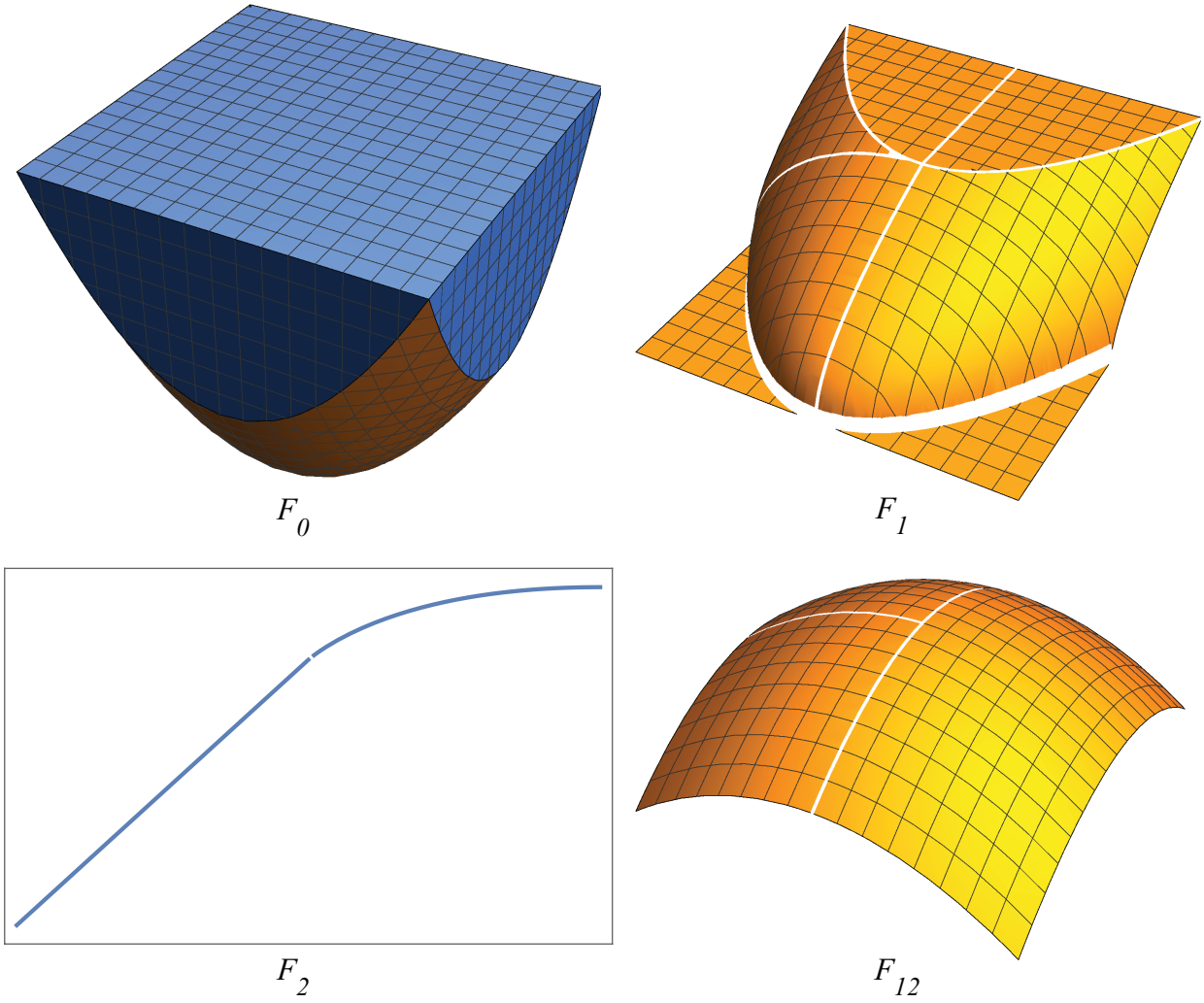


Figure 4.2: Progressive steps of integration of a paraboloidal step function, showing the propagation of discontinuities.

counted for simply by dividing the integration domain at the critical values of x that lie on the discontinuous surface for the appropriate values of y and z.

Subdividing the integration domain in this way essentially eliminates the effect of the discontinuous surface on the computation of $F_1(y,z)$, but this new function may also be discontinuous or otherwise-ill-behaved. It is therefore to devise a method for predicting such critical surfaces in F_1 based on knowledge of the behavior of F_0 and the integration step connecting the two functions.

If critical surfaces of some function $F_0(x_0,...,x_n)$ are denoted by the functions $f_{0k}(x_0,...,x_n) = 0$, then critical values of x_0, denoted by $x_0^*(x_1,...,x_n)$, can be found. The function $F_1(x_1,...,x_n)$ can be derived by integration of F_0 with respect to x_0, subdividing the integration domain at the critical values x_0^*.

The function $F_1(x_1,...x_n) \equiv \int_{x_{0\,\min}}^{x_{0\,\max}} F_0 dx_0$ may itself have critical surfaces, which result from one of two mechanisms.

The first source of critical surfaces in F_1 is orthogonality between the surface normal of the surfaces defined by the original functions f_{0k} and the initial direction of integration. That is, one can define new critical surfaces $f_{1l}(x_1,...,x_n) = f_{0k}(x_0^{**},x_1,...,x_n)$, where x_0^{**} are the zeros of $\frac{\partial f_{0k}}{\partial x_0}$.

Critical surfaces can also arise from the intersection between multiple surfaces, including those that bound the integration domain. For these, one can define new surfaces by solving the intersecting surfaces together to eliminate x_0

As an example, consider the paraboloidal step function F_0 given in Eq. 4.1. One critical surface in F_1 can be derived as $\frac{\partial}{\partial x}\left(x^2+y^2-z\right) = 2x = 0 \Rightarrow y^2 - z = 0$, while the intersection of the paraboloidal surface with the bounding surfaces $x = \pm 1$ yields the new surface $y^2 - z + 1 = 0$. Critical surfaces for F_2 may be similarly derived, though only one new critical value emerges, defined by $z - 1 = 0$.

It is important to note that, although changing the order of integration has no effect on the end result, the intermediate steps may be quite different, and these differences may affect the cost of integration. For instance, if the order of integration in the above problem is changed so that the integral in z is evaluated first, then the resulting function will have no discontinuities at

all, as seen in Fig. 4.3. Since dividing an integration domain increases the computational cost of that integration by at least a factor of 2, it clearly advantageous to choose the order of integration carefully so as to eliminate critical surfaces as quickly as possible.

4.4 Integration Verification

4.5 BACL-Manufactured

Insert text here

4.6 Integrative Manufactured Solutions

Insert text here

4.7 Future Work

Insert text here

Chapter 5

Conclusion

Insert text here

Bibliography

[1] AT&T Bell Laboratories, University of Tennessee, and Oak Ridge National Laboratory. Netlib.

[2] W H Hui. The Unified Coordinate System in Computational Fluid Dynamics. Communications in Computational Physics, 2(4):577–610, 2007.

[3] W. H. Hui, Z. N. Wu, and B. Gao. Preliminary Extension of the Unified Coordinate System Approach to Computation of Viscous Flows. Journal of Scientific Computing, 30(2):301–344, October 2007.

[4] WH Hui, PY Li, and ZW Li. A unified coordinate system for solving the two-dimensional Euler equations. Journal of Computational Physics, 153:596–637, 1999.

[5] Travis E Oliphant. SciPy: Open source scientific tools for Python. Computing in Science and Engineering, 9:10–20, 2007.

[6] Bojan Popov and Ognian Trifonov. Order of convergence of second order schemes based on the minmod limiter. Mathematics of computation, 75(256):1735–1753, 2006.

[7] F. Sabac. The Optimal Convergence Rate of Monotone Finite Difference Methods for Hyperbolic Conservation Laws. SIAM Journal on Numerical Analysis, 34(6):2306–2318, 1997.

[8] Herrmann Schlichting and Klaus Gersten. Boundary Layer Theory. Springer-Verlag, New York, 8th edition, 2000.

[9] P D Thomas and C K Lombard. Geometric Conservation Law and Its Application to Flow Computations on Moving Grids. AIAA Journal, 17(10):1030–1037, October 1979.

[10] E F Toro. NUMERICA: a Library of Source Codes for Teaching, Research and Applications. HYPER-EUL. Methods for the Euler equations., 1999.

[11] Euletorio Toro. The development of a Riemann solver for the steady supersonic Euler equations. Aeronautical Journal, 98:325–339, 1994.

[12] Euletorio Toro. Riemann Solvers and Numerical Methods for Fluid Dynamics. Springer, 2009.

Appendix A

Streamer Implementations

A.1 Streamer v1.0

A.2 Streamer v2.0

A.3 Streamer v3.0

Appendix B

scipy.integrate.nquad

Appendix C

BACL-Manufactured